

# Lab 4 Report

## Robotics Integration Group Project I

Yuwei ZHAO (23020036096)

Group #31 2025-11-30

### Abstract

Lab4 investigates the mathematical foundations and practical application of polynomial trajectory optimization for UAVs. The primary objective is to formulate trajectory generation as a Quadratic Programming (QP) problem to minimize derivatives of position, such as velocity and snap. We begin by analytically deriving the cost and constraint matrixes for single-segment minimum velocity problems, verifying that the optimal solutions align with the Euler-Lagrange equation. The analysis is then extended to multi-segment minimum snap trajectories, identifying the necessary waypoint, continuity, and boundary constraints required for a unique solution. Finally, these theoretical frameworks are applied to a drone racing scenario, where optimal trajectories are generated to navigate a quadrotor through a sequence of gates.

See Resources on [github.com/RamessesN/Robotics\\_MIT](https://github.com/RamessesN/Robotics_MIT).

## 1 Introduction

Trajectory generation is a core component of quadrotor control, ensuring smooth navigation by minimizing specific state derivatives. This laboratory focuses on **Polynomial Trajectory Optimization**, specifically transforming the variational problem of minimizing an integral cost into a numerical Quadratic Program (QP).

The report is structured in three parts:

### 1. Single-Segment Formulation

We analytically derive the cost matrix  $Q$  and constraint matrix  $A$  for a minimum velocity problem ( $r = 1$ ). We verify that the QP solution aligns with the theoretical optimum derived from the Euler-Lagrange equation.

### 2. Multi-Segment Extension

We extend the analysis to piece-wise polynomials over  $k$  segments. We derive the counting rules for waypoint, continuity, and boundary constraints to ensure a unique solution for high-order problems like Minimum Snap.

### 3. Application

Finally, we utilize this framework to generate optimal trajectories for a drone racing scenario, navigating a quadrotor through a sequence of gates.

## 2 Procedure

### 2.1 Individual Work

#### 2.1.1 Single-segment trajectory optimization

Consider the following minimum velocity ( $r = 1$ ) single-segment trajectory optimization problem:

$$\min_{P(t)} \int_0^1 (P^{(1)}(t))^2 dt \quad (1)$$

s.t.

$$P(0) = 0, \quad (2)$$

$$P(1) = 1, \quad (3)$$

with  $P(t) \in \mathbb{R}[t]$ , i.e.,  $P(t)$  is a polynomial function in  $t$  with real coefficients:

$$P(t) = p_N t^N + p_{N-1} t^{N-1} + \dots + p_1 t + p_0 \quad (4)$$

Note that because of constraint (2)  $P(0) = p_0 = 0$ , and we can parametrize  $P(t)$  without a scalar part  $p_0$ .

1. Suppose we restrict  $P(t) = p_1 t$  to be a polynomial of degree 1, what is the optimal solution of problem (1)? What is the value of the cost function at the optimal solution?

$$\therefore P(t) = p_1 t$$

$$\text{Let } t = 1 \therefore P(1) = p_1 \cdot 1 = p_1$$

$$\therefore P(1) = 1 \therefore p_1 = 1$$

$$\therefore \text{optimal solution: } P(t) = t.$$

$$\therefore P(t) = t \therefore P^{(1)}t = \frac{d}{dt}t = 1$$

$$\therefore \text{Cost} = \int_0^1 (1)^2 dt = 1.$$

2. Suppose now we allow  $P(t)$  to have degree 2, i.e.,  $P(t) = p_2 t^2 + p_1 t$ .

- Write  $\int_0^1 (P^{(1)}(t))^2 dt$ , the cost function of problem (1), as  $p^T Q p$ , where  $p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$  and  $Q \in S^2$  is a symmetric  $2 \times 2$  matrix.

$$\therefore P(t) = p_2 t^2 + p_1 t \therefore P^{(1)}(t) = 2p_2 t + p_1$$

$$\therefore \text{Cost} = \int_0^1 (2p_2 t + p_1)^2 dt = p_1^2 + 2p_1 p_2 + \frac{4}{3} p_2^2$$

In order to write into a  $2 \times 2$  matrix as  $p^T Q p$ , we have

$$[p_1, p_2] \cdot \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = p_1^2 + 2p_1p_2 + \frac{4}{3}p_2^2$$

$$\therefore Q_{11} = 1, Q_{22} = \frac{4}{3}$$

$$\therefore Q_{12} = Q_{21} \therefore 2Q_{12} = 2$$

$$\therefore Q_{12} = Q_{21} = 1 \Rightarrow Q = \begin{bmatrix} 1 & 1 \\ 1 & \frac{4}{3} \end{bmatrix}.$$

- Write  $P(1) = 1$ , constraint (3), as  $Ap = b$ , where  $A \in \mathbb{R}^{1 \times 2}$  and  $b \in \mathbb{R}$ .

$$\therefore P(1) = 1$$

$$\therefore P(1) = p_2(1)^2 + P_1(1) = p_1 + p_2 = 1$$

$$\therefore Ap = b \text{ and } p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} \therefore A = [1 \ 1], b = 1.$$

- Solve the Quadratic Program (QP):

$$\min_p p^T Q p \text{ s.t. } Ap = b \quad (5)$$

You can solve it by hand, or you can solve it using numerical QP solvers (e.g., you can easily use the quadprog function in Matlab). What is the optimal solution you get for  $P(t)$ , and what is the value of the cost function at the optimal solution? Are you able to get a lower cost by allowing  $P(t)$  to have degree 2?

$$\therefore \text{we have } \min_p p^T Q p \Leftrightarrow \min_{p_1, p_2} (p_1^2 + 2p_1p_2 + \frac{4}{3}p_2^2)$$

$$\text{and } Ap = b \Leftrightarrow p_1 + p_2 = 1$$

$$\text{Let } p_1 = 1 - p_2 \therefore \text{Cost} = (1 - p_2)^2 + 2(1 - p_2)p_2 + \frac{4}{3}p_2^2 = 1 + \frac{1}{3}p_2^2$$

$$\text{In order to make Cost minimum } \Rightarrow \begin{cases} p_1=1 \\ p_2=0 \end{cases} \therefore \text{Cost}_{\text{minimal}} = 1.$$

No, it remains the same value even though  $P(t)$  has degree 2.

3. Now suppose we allow  $P(t) = p_3t^3 + p_2t^2 + p_1t$ :

- Let  $p = [p_1, p_2, p_3]^T$ , write down  $Q \in S^3$ ,  $A \in \mathbb{R}^{1 \times 3}$ ,  $b \in \mathbb{R}$  for QP (5).

$$\therefore P(t) = p_3t^3 + p_2t^2 + p_1t$$

$$\therefore P_t^{(1)} = 3p_3t^2 + 2p_2t + p_1$$

$$\therefore [P_t^{(1)}]^2 = 9p_3^2t^4 + 4p_2^2t^2 + p_1^2 + \underbrace{12p_2p_3t^3}_{p_2p_3} + \underbrace{6p_1p_3t^2}_{p_1p_3} + \underbrace{4p_1p_2t}_{p_1p_2}$$

$\therefore$  we have

$$\left\{ \begin{array}{l} \text{item } p_3^2 : \int_0^1 9t^4 dt = \frac{9}{5} \\ \text{item } p_2^2 : \int_0^1 4t^2 dt = \frac{4}{3} \\ \text{item } p_1^2 : \int_0^1 1dt = 1 \\ \text{item } p_2 p_3 : \int_0^1 12t^3 dt = 3 \\ \text{item } p_1 p_3 : \int_0^1 6t^2 dt = 2 \\ \text{item } p_1 p_2 : \int_0^1 4tdt = 2 \end{array} \right.$$

$$\therefore Q = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \frac{4}{3} & \frac{3}{2} \\ 1 & \frac{3}{2} & \frac{9}{5} \end{bmatrix}$$

$$\because p_3(1)^3 + p_2(1)^2 + p_1(1) = 1 \Rightarrow 1 \cdot p_1 + 1 \cdot p_2 + 1 \cdot p_3 = 1 \\ \therefore A = [1 \ 1 \ 1], b = 1.$$

- Solve the QP, what optimal solution do you get? Do this example agree with the result we learned from Euler-Lagrange equation in class?**

From the above, we have the path that connects two points and minimizes the change in speed (energy) is always a straight line. That's regardless of the inclusion of higher-degree terms like  $t^2$  or  $t^3$ , the optimization drives their coefficients to 0. The curve connecting the two points that minimizes the velocity cost is always a straight line. Consequently, the value of the cost function remains 1.

Yes. By Euler–Lagrange equation, we have  $\frac{\partial L}{\partial P} - \frac{d}{dt} \frac{\partial L}{\partial P'} = 0$

$$\text{Since } L = (P')^2 \quad \therefore \frac{d}{dt}(2P') = 0 \Rightarrow P''(t) = 0$$

The condition  $P''(t) = 0$  implies that the optimal function must be linear. The QP result is indeed a linear function, which confirms that the theoretical result derived from calculus of variations.

#### 4. Now suppose we are interested in adding one more constraint to problem (1):

$$\min_{P(t)} \int_0^1 (P^{(1)}(t))^2 dt, \text{ s.t. } P(0) = 0, P(1) = 1, P^{(1)}(1) = -2 \quad (6)$$

Using the QP method above, find the optimal solution and optimal cost of problem (6) in the case of:

- $P(t) = p_2 t^2 + p_1 t$ , and
- $P(t) = p_3 t^3 + p_2 t^2 + p_1 t$ .

§ Case I. If  $P(t) = p_2 t^2 + p_1 t$ ,

$$\begin{cases} \text{For } P(1)=1: p_1 + p_2 = 1 \\ P^{(1)}(1)=-2: p_1 + 2p_2 = -2 \end{cases}$$

$$\therefore \text{we have } \begin{cases} p_1=4 \\ p_2=-3 \end{cases}$$

$$\therefore \begin{cases} P(t) = -3t^2 + 4t \\ \text{Cost} = p_1^2 + 2p_1p_2 + \frac{4}{3}p_2^2 = 4 \end{cases}$$

**§ Case II. If  $P(t) = p_3 t^3 + p_2 t^2 + p_1 t$ ,**

$$\begin{cases} \text{For } P(1)=1: p_1+p_2+p_3=1 \\ P^{(1)}(1)=-2: p_1+2p_2+3p_3=-2 \end{cases}$$

$$\therefore \begin{cases} p_1=4+p_3 \\ p_2=-3-2p_3 \end{cases}$$

From (3), we have:  $Q = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \frac{4}{3} & \frac{3}{2} \\ 1 & \frac{3}{2} & \frac{9}{5} \end{bmatrix}$

$$p = \begin{bmatrix} 4+p_3 \\ -3-2p_3 \\ p_3 \end{bmatrix} = \underbrace{\begin{bmatrix} 4 \\ -3 \\ 0 \end{bmatrix}}_{p_{\text{base}}} + p_3 \underbrace{\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}}_d$$

$$\text{For Cost}(p_3) = Ap_3^2 + Bp_3 + C, \text{ we have } \begin{cases} A=d^T Q d \\ B=2p_{\text{base}}^T Q d \end{cases}$$

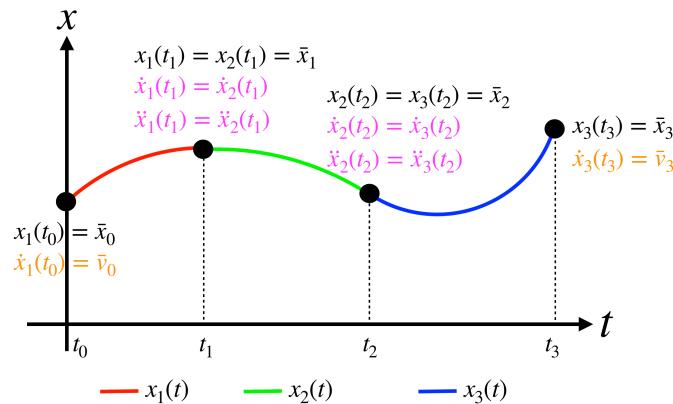
$$\therefore Qd = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \frac{4}{3} & \frac{3}{2} \\ 1 & \frac{3}{2} & \frac{9}{5} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{1}{6} \\ -\frac{1}{5} \end{bmatrix}$$

$$\therefore A = \frac{2}{15}, B = 1 \quad \therefore \text{Cost} = \frac{2}{15}p_3^2 + p_3 + 4$$

$$\text{In order to make cost minimum } \Rightarrow p_3 = -\frac{15}{4} \Rightarrow \text{Cost}_{\text{minimal}} = \frac{17}{8}.$$

## 2.1.2 Multi-segment trajectory optimization

- Assume our goal is to compute the minimum snap trajectory ( $r = 4$ ) over  $k$  segments. How many and which type of constraints (at the intermediate points and at the start and end of the trajectory) do we need in order to solve this problem? Specify the number of waypoint constraints, free derivative constraints and fixed derivative constraints.



$$\text{Cost} = \int (x^{(4)}(t))^2 dt \Rightarrow x^{(2r)}(t) = 0 \Rightarrow x^{(8)}(t) = 0$$

**Step 1: Determine the number of Unknowns**

Given the cost function  $\text{Cost} = \int (x^{(4)}(t))^2 dt$ , the Euler-Lagrange equation yields the necessary condition:

$$x^{(2r)}(t) = 0 \stackrel{r=4}{\Rightarrow} x^{(8)}(t) = 0$$

Integrating this equation **8 times**, we obtain a polynomial of degree  $2r - 1 = 7$ :

$$P(t) = p_7 t^7 + p_6 t^6 + \dots + p_1 t + p_0$$

- ∴ Each segment has  $N + 1 = 8$  unknown coefficients and there are  $k$  segments.
- ∴ Total Unknowns =  $8k$ , which means we need  $8k$  constraints to solve for a unique solution.

(1) For *Waypoint Constraints*:

For each segment  $i$ , the position at start  $t_{i-1}$  and end  $t_i$  is fixed.

$$\therefore 2 \text{ constraints} \times k \text{ segments} = 2k \text{ constraints.}$$

(2) For *Free Derivative Constraints*:

At the  $(k - 1)$  intermediate waypoints, the trajectory must be smooth.

Continuity is required for derivatives up to  $2r - 2 = 6$  (i.e., 1<sup>st</sup> to 6<sup>th</sup> derivatives).

$$\therefore 6 \text{ constraints} \times (k - 1) \text{ points} = 6(k - 1) \text{ constraints.}$$

(3) For *Fixed Derivative Constraints*:

At the start  $t_0$  and end  $t_k$  of the entire trajectory, we fix derivatives up to  $r - 1 = 3$  (Velocity, Acc, Jerk).

$$\therefore 3 \text{ (start)} + 3 \text{ (end)} = 6 \text{ constraints.}$$

**> Proof:**

$2k + 6(k - 1) + 6 = 8k$ , which confirms that total number of constraints is  $8k$ .

**2. Can you extend the previous question to the case in which the cost functional minimizes the  $r$ -th derivative and we have  $k$  segments?**

From the method above, we have Total number of constraints =  $2rk$ .

Specifically,

(1) For *Waypoint Constraints*:

$2k$  (Start and End positions for each segment).

(2) For *Free Derivative Constraints*:

$(k - 1) \cdot (2r - 2)$  (Continuity of 1<sup>st</sup> to  $(2r - 2)$ <sup>th</sup> derivatives at intermediate points).

(3) For *Fixed Derivative Constraints*:

$2(r - 1)$  (Fixing 1<sup>st</sup> to  $(r - 1)$ <sup>th</sup> derivatives at  $t_0$  and  $t_k$ ).

**> Proof:**

$2k + (k - 1)(2r - 2) + 2(r - 1) = 2rk$ , which confirms that total number of constraints is  $2rk$ .

## 2.2 Team Work

### 2.2.1 Drone Racing

In this section, we implemented the `trajectory_generation_node` in C++ to enable the quadrotor to autonomously navigate through a sequence of gates. The implementation details are divided into three functional blocks: initialization, trajectory generation, and command publishing.

#### 1. State Initialization and Safety Hover

The node subscribes to the `/current_state` topic to retrieve the UAV's real-time position and orientation. We converted the ROS `Odometry` messages into `Eigen::Vector3d` for internal calculations.

Crucially, to prevent the drone from drifting or crashing before the race starts, we implemented a **safety hover logic**. When the trajectory container is empty (idle state), the node continuously captures the current position and publishes it as the desired setpoint with zero velocity and acceleration. This ensures the drone maintains a stable hover at the starting gate until the race track waypoints are received.

#### 2. Trajectory Optimization and Constraints

Upon receiving the gate waypoints, we utilized the `mav_trajectory_generation` library to formulate the optimization problem.

- **Position Constraints:** We differentiated between intermediate gates and the endpoints. Intermediate waypoints were added as position constraints allowing non-zero velocity (fly-through), while the final waypoint was constrained using `makeStartOrEnd(...)` to enforce zero velocity and acceleration, ensuring a safe stop.
- **Yaw Unwrapping Strategy:** A critical challenge was handling the discontinuity of orientation angles (e.g., the jump from  $\pi$  to  $-\pi$ ). We implemented a yaw unwrapping algorithm that checks the difference between consecutive waypoints. If  $|\psi_k - \psi_{k-1}| > \pi$ , we added or subtracted  $2\pi$  to the target yaw. This ensures the generated yaw trajectory is continuous and prevents the drone from spinning unnecessary full circles.

#### 3. Trajectory Sampling and Publishing

The generated polynomial trajectory is continuous, but the controller requires discrete setpoints. We set up a ROS timer running at **100Hz**. At each time step  $t$ , the node:

- Checks if  $t$  exceeds the total trajectory duration.
- Evaluates the polynomial to extract the desired position  $x_{d(t)}$ , velocity  $v_{d(t)}$ , acceleration  $a_{d(t)}$ , and yaw  $\psi_{d(t)}$ .
- Packages these values into a `MultiDOFJointTrajectoryPoint` message.
- Converts the `Eigen` types back to `geometry_msgs` and publishes them to the `/desired_state` topic.

## 2.2.2 Environment Preparation

The experimental environment was set up by updating the codebase and configuring the workspace as follows:

### 1. Codebase Synchronization

First, the lab repository was updated to acquire the latest packages:

```
1 cd ~/labs  
2 git pull
```

Shell

Upon verification, the `~/labs/lab4` directory contained the required packages: `planner_pkg`, `trajectory_generation_pkg`, and dependencies.

### 2. Workspace Configuration

The new packages were copied into the ROS workspace source directory:

```
1 cp -r ~/labs/lab4/ ~/vnav_ws/src  
2 cd ~/vnav_ws
```

Shell

The `src` directory now includes `controller_pkg`, `tesse-ros-bridge`, `planner_pkg`, `trajectory_generation_pkg`, and dependencies. The workspace was then rebuilt and sourced:

```
1 catkin build  
2 source devel/setup.bash
```

Shell

Then, copy the [simulator](#) to `~/vnav-builds/lab4/`.

```
1 cd ~/vnav-builds/lab4/  
2 chmod +x lab4.x86_64
```

Shell

### 3. Simulator Setup

The [Unity simulator binary](#) was downloaded to `~/vnav-builds/lab4/` and configured with execution permissions:

```
1 cd ~/vnav-builds/lab4/  
2 chmod +x lab4.x86_64
```

Shell

#### 4. Execution Sequence

The complete command sequence to launch the simulation environment and the autonomous racing stack is listed below:

```

1  # 1. Launch Unity Simulator
2  cd ~/vnav-builds/lab4/
3  ./lab4.x86_64
4
5  # 2. Establish ROS Bridge
6  roslaunch tesse_ros_bridge tesse_quadrotor_bridge.launch
7
8  # 3. Start Trajectory Follower (Controller + Traj Generation)
9  roslaunch trajectory_generation traj_following.launch
10
11 # 4. Trigger Race (Waypoint Publisher)
12 roslaunch planner_pkg traj_gen.launch

```

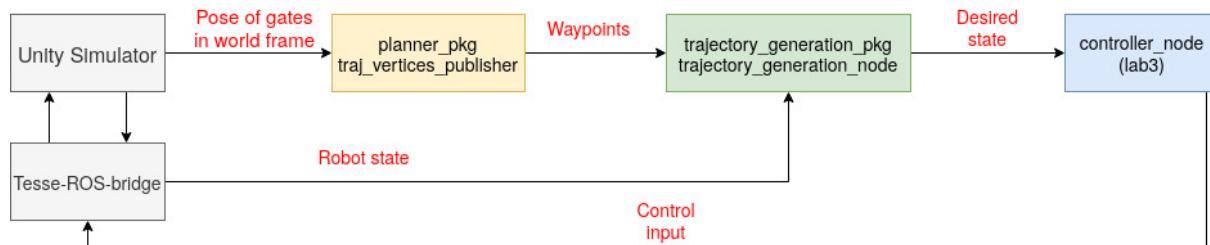


Figure 2: The Tesse ROS Bridge establishing communication between the Unity physics engine and the ROS control stack.

The visualization above shows the Unity simulator running alongside the ROS bridge, which facilitates real-time data exchange between the drone model in Unity and the ROS nodes handling trajectory generation and control.

#### 2.2.3 Simulation Results

The complete system was tested in the Unity simulator. The drone successfully generated a smooth trajectory passing through all gates and completed the race without collision.

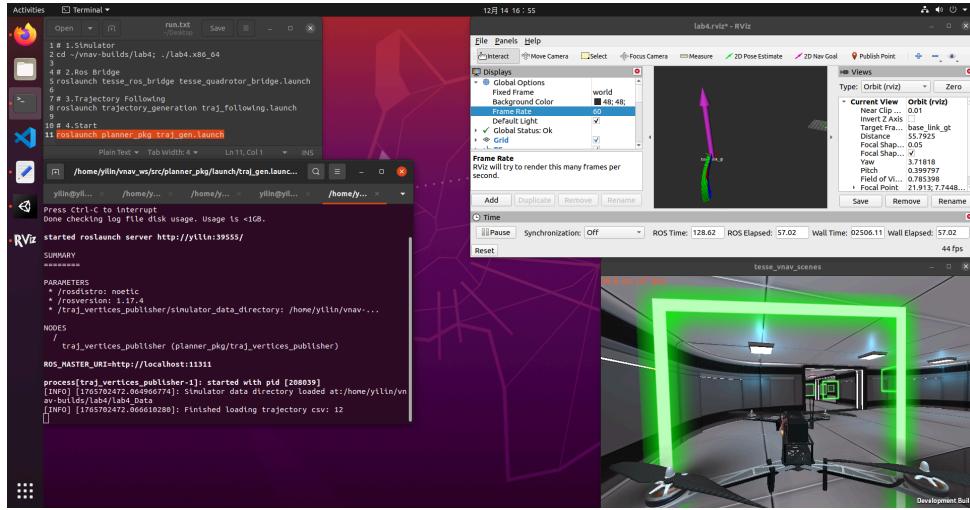


Figure 3: The quadrotor autonomously navigating through the gates in the Unity simulator environment.

The visualization above confirms that the generated trajectory (visualized by the red path in Rviz/Unity) effectively connects the gate vertices while maintaining smoothness, and the drone's actual path closely tracks this reference.

### 3 Reflection and Analysis

The implementation process highlighted the importance of robust software design in robotics integration.

- Handling Discontinuities:** The mathematical optimization assumes continuous functions. Practical implementation issues, such as the modular nature of angles (Yaw), required specific algorithmic handling (unwrapping) to bridge the gap between linear algebra and physical rotation.
- System Timing:** The synchronization between the trajectory generator and the controller was critical. We observed that the sampling frequency of the trajectory node (100Hz) significantly impacted the flight smoothness. A lower frequency caused “stuttering” in the control commands, while a higher frequency ensured the geometric controller received smooth derivatives (velocity and acceleration feed-forward terms), which are essential for aggressive maneuvers.

### 4 Conclusion

In this lab, we successfully linked the theoretical framework of polynomial trajectory optimization with a practical ROS-based control system.

We derived the constraints for single-segment and multi-segment optimization, verifying that Minimum Snap trajectories require  $8k$  constraints for a unique solution. In the team work section, we implemented the C++ logic to convert these mathematical constraints into a func-

tional software module. The successful drone racing simulation demonstrated that the QP-based approach effectively generates feasible, smooth, and high-speed trajectories for complex 3D environments.

## 5 Source Code

- *trajectory\_generation\_node.cpp*

```
1 #include <eigen_conversions/eigen_msg.h>
2 #include <geometry_msgs/PoseArray.h>
3 #include <nav_msgs/Odometry.h>
4 #include <ros/ros.h>
5 #include <trajectory_msgs/MultiDOFJointTrajectory.h>
6
7 #include <mav_trajectory_generation/polynomial_optimization_linear.h>
8 #include <mav_trajectory_generation/trajecotry.h>
9
10 #include <tf/transform_datatypes.h>
11 #include <tf2_geometry_msgs/tf2_geometry_msgs.h>
12 #include <eigen3/Eigen/Dense>
13
14 class WaypointFollower {
15     [[maybe_unused]] ros::Subscriber currentStateSub;
16     [[maybe_unused]] ros::Subscriber poseArraySub;
17     ros::Publisher desiredStatePub;
18
19     // Current state
20     Eigen::Vector3d x; // current position of the UAV's c.o.m. in the world
21     frame
22
23     ros::Timer desiredStateTimer;
24
25     ros::Time trajectoryStartTime;
26     mav_trajectory_generation::Trajectory trajectory;
27     mav_trajectory_generation::Trajectory yaw_trajectory;
28
29     void onCurrentState(nav_msgs::Odometry const& cur_state) {
30         // PART 1.1 - 将 ROS 消息转换为 Eigen 向量
31         tf::pointMsgToEigen(cur_state.pose.pose.position, x);
32     }
33
34     void generateOptimizedTrajectory(geometry_msgs::PoseArray const& poseArray)
35     {
36         if (poseArray.poses.size() < 1) {
37             ROS_ERROR("Must have at least one pose to generate trajectory!");
38             trajectory.clear();
39             yaw_trajectory.clear();
40             return;
41         }
42     }
43 }
```

```

41     if (!trajectory.empty()) return;
42
43     const int D = 3; // dimension of each vertex in the trajectory
44     mav_trajectory_generation::Vertex start_position(D), end_position(D);
45     mav_trajectory_generation::Vertex::Vector vertices;
46     mav_trajectory_generation::Vertex start_yaw(1), end_yaw(1);
47     mav_trajectory_generation::Vertex::Vector yaw_vertices;
48
49     // Convert the pose array to a list of vertices
50     // Start from the current position and zero orientation
51     using namespace mav_trajectory_generation::derivative_order;
52     start_position.makeStartOrEnd(x, SNAP);
53     vertices.push_back(start_position);
54
55     start_yaw.addConstraint(ORIENTATION, 0);
56     yaw_vertices.push_back(start_yaw);
57
58     double last_yaw = 0;
59
60     for (auto i = 0; i < poseArray.poses.size(); ++i) {
61         // PART - 1.2
62
63         // --- 1. Process position vertex ---
64         Eigen::Vector3d pos_eigen;
65         tf::pointMsgToEigen(poseArray.poses[i].position, pos_eigen);
66
67         mav_trajectory_generation::Vertex pos_vertex(D);
68
69         // If it is the last point, it must be the end point, and the velocity
70         // acceleration is forced to be 0
71         if (i == poseArray.poses.size() - 1) {
72             pos_vertex.makeStartOrEnd(pos_eigen, SNAP);
73         } else {
74             // If it is an intermediate point, only position constraints are
75             // added to allow passage at a certain speed
76             pos_vertex.addConstraint(POSITION, pos_eigen);
77         }
78         vertices.push_back(pos_vertex);
79
80         // --- 2. Process yaw vertex ---
81         double current_yaw = tf::getYaw(poseArray.poses[i].orientation);
82
83         while (current_yaw - last_yaw > M_PI) current_yaw -= 2 * M_PI;
84         while (current_yaw - last_yaw < -M_PI) current_yaw += 2 * M_PI;

```

```

85     mav_trajectory_generation::Vertex yaw_vertex(1);
86     yaw_vertex.addConstraint(ORIENTATION, current_yaw);
87     yaw_vertices.push_back(yaw_vertex);
88
89     last_yaw = current_yaw;
90 }
91
92 std::vector<double> segment_times;
93 const double v_max = 15.0;
94 const double a_max = 10.0;
95 segment_times = estimateSegmentTimes(vertices, v_max, a_max);
96 for(int i = 0; i < segment_times.size(); i++) {
97     segment_times[i] *= 0.6;
98 }
99
100 // Position
101 const int N = 10;
102 mav_trajectory_generation::PolynomialOptimization<N> opt(D);
103 opt.setupFromVertices(vertices, segment_times,
104 mav_trajectory_generation::derivative_order::SNAP);
105 opt.solveLinear();
106
107 // Yaw
108 mav_trajectory_generation::PolynomialOptimization<N> yaw_opt(1);
109 yaw_opt.setupFromVertices(yaw_vertices, segment_times,
110 mav_trajectory_generation::derivative_order::SNAP);
111 yaw_opt.solveLinear();
112
113 mav_trajectory_generation::Segment::Vector segments;
114 opt.getTrajectory(&trajectory);
115 yaw_opt.getTrajectory(&yaw_trajectory);
116 trajectoryStartTime = ros::Time::now();
117
118 ROS_INFO("Generated optimizes trajectory from %lu waypoints",
119 vertices.size());
120 }
121
122 void publishDesiredState(ros::TimerEvent const& ev) {
123     if (trajectory.empty()) {
124         // If there is no trajectory yet, publish the current position as the
125         // desired position
126         trajectory_msgs::MultiDOFJointTrajectoryPoint hover_point;
127
128         hover_point.time_from_start = ros::Duration(0.0);
129
130         // 1. Set the position to the current position x
131         geometry_msgs::Transform transform;

```

```

128     tf::vectorEigenToMsg(x, transform.translation);
129
130     transform.rotation = tf::createQuaternionMsgFromYaw(0);
131     hover_point.transforms.push_back(transform);
132
133     // 2. Set the speed to 0
134     geometry_msgs::Twist velocity;
135     velocity.linear.x = 0; velocity.linear.y = 0; velocity.linear.z = 0;
136     velocity.angular.x = 0; velocity.angular.y = 0; velocity.angular.z =
137     0;
138     hover_point.velocities.push_back(velocity);
139
140     // 3. Set the acceleration to 0
141     geometry_msgs::Twist accel;
142     accel.linear.x = 0; accel.linear.y = 0; accel.linear.z = 0;
143     accel.angular.x = 0; accel.angular.y = 0; accel.angular.z = 0;
144     hover_point.accelerations.push_back(accel);
145
146     // Issue hover commands
147     desiredStatePub.publish(hover_point);
148     return;
149 }
150
151 // PART 1.3
152 trajectory_msgs::MultiDOFJointTrajectoryPoint next_point;
153
154 // 1. Calculate the time from the beginning of the trajectory to the
155 // present
156 ros::Duration time_from_start = ros::Time::now() - trajectoryStartTime;
157 next_point.time_from_start = time_from_start;
158
159 double sampling_time = time_from_start.toSec();
160
161 // 2. The time is prevented from exceeding the total length of the
162 // trajectory
163 if (sampling_time > trajectory.getMaxTime())
164     sampling_time = trajectory.getMaxTime();
165
166 // Getting the desired state based on the optimized trajectory we found.
167 using namespace mav_trajectory_generation::derivative_order;
168 Eigen::Vector3d des_position = trajectory.evaluate(sampling_time,
169 POSITION);
170 Eigen::Vector3d des_velocity = trajectory.evaluate(sampling_time,
171 VELOCITY);
172 Eigen::Vector3d des_accel = trajectory.evaluate(sampling_time,
173 ACCELERATION);

```

```

168     Eigen::VectorXd des_orientation = yaw_trajectory.evaluate(sampling_time,
169     ORIENTATION);
170
171     // Populate next_point
172
173     // A. Fill Transform (position + pose)
174     geometry_msgs::Transform transform;
175     tf::vectorEigenToMsg(des_position, transform.translation);
176     tf::quaternionTFToMsg(tf::createQuaternionFromYaw(des_orientation(0)),
177     transform.rotation);
178     next_point.transforms.push_back(transform);
179
180     // B. Fill Velocity (linear velocity + angular velocity)
181     geometry_msgs::Twist velocity;
182     tf::vectorEigenToMsg(des_velocity, velocity.linear);
183     velocity.angular.x = 0;
184     velocity.angular.y = 0;
185     velocity.angular.z = 0;
186     next_point.velocities.push_back(velocity);
187
188     // C. Fill Acceleration (linear acceleration)
189     geometry_msgs::Twist accel;
190     tf::vectorEigenToMsg(des_accel, accel.linear);
191     accel.angular.x = 0;
192     accel.angular.y = 0;
193     accel.angular.z = 0;
194     next_point.accelerations.push_back(accel);
195
196
197 public:
198     explicit WaypointFollower(ros::NodeHandle& nh) {
199         currentStateSub = nh.subscribe(
200             "/current_state", 1, &WaypointFollower::onCurrentState, this);
201         poseArraySub = nh.subscribe("/desired_traj_vertices",
202                                     1,
203                                     &WaypointFollower::generateOptimizedTrajectory
204                                     this);
205         desiredStatePub =
206             nh.advertise<trajectory_msgs::MultiDOFJointTrajectoryPoint>(
207                 "/desired_state", 1);
208         desiredStateTimer = nh.createTimer(
209             ros::Rate(100), &WaypointFollower::publishDesiredState, this);
210         desiredStateTimer.start();
211     }

```

```

212 };
213
214 int main(int argc, char** argv) {
215     ros::init(argc, argv, "trajectory_generation_node");
216     ros::NodeHandle nh;
217
218     WaypointFollower waypointFollower(nh);
219
220     ros::spin();
221     return 0;
222 }
```

- *controller\_node.cpp*

```

1 #include <ros/ros.h> C++
2
3 #include <tf2_geometry_msgs/tf2_geometry_msgs.h>
4 #include <tf2/LinearMath/Quaternion.h>
5 #include <tf2/utils.h>
6 #include <mav_msgs/Actuators.h>
7 #include <nav_msgs/Odometry.h>
8 #include <trajectory_msgs/MultiDOFJointTrajectoryPoint.h>
9 #include <cmath>
10
11 #define PI M_PI
12
13 #include <eigen3/Eigen/Dense>
14 #include <tf2_eigen/tf2_eigen.h>
15
16 class controllerNode{
17     ros::NodeHandle nh;
18
19     // PART 1: Declare ROS callback handlers
20     ros::Subscriber des_state_sub, cur_state_sub;
21     ros::Publisher propeller_speeds_pub;
22     ros::Timer control_timer;
23
24     // Controller parameters
25     double kx, kv, kr, komega;
26
27     // Physical constants (we will set them below)
28     double m;           // mass of the UAV
29     double g;           // gravity acceleration
30     double d;           // distance from the center of propellers to the
                          C.o.m.
```

```

31     double cf,           // Propeller lift coefficient
32             cd;           // Propeller drag coefficient
33
34     Eigen::Matrix3d J;   // Inertia Matrix
35     Eigen::Vector3d e3;  // [0,0,1]
36     Eigen::MatrixXd F2W; // Wrench-rotor speeds map
37
38     // Controller internals
39     // Current state
40     Eigen::Vector3d x;    // current position of the UAV's c.o.m. in the world
41     // frame
42     Eigen::Vector3d v;    // current velocity of the UAV's c.o.m. in the world
43     // frame
44     Eigen::Matrix3d R;    // current orientation of the UAV
45     Eigen::Vector3d omega; // current angular velocity of the UAV's c.o.m. in
46     // the *body* frame
47
48     // Desired state
49     Eigen::Vector3d xd;   // desired position of the UAV's c.o.m. in the world
50     // frame
51     Eigen::Vector3d vd;   // desired velocity of the UAV's c.o.m. in the world
52     // frame
53     Eigen::Vector3d ad;   // desired acceleration of the UAV's c.o.m. in the
54     // world frame
55     double yawd;         // desired yaw angle
56
57     double hz;            // frequency of the main control loop
58
59     static Eigen::Vector3d Vee(const Eigen::Matrix3d& in){
60         Eigen::Vector3d out;
61         out << in(2,1), in(0,2), in(1,0);
62         return out;
63     }
64
65     static double signed_sqrt(double val){
66         return val > 0 ? sqrt(val) : -sqrt(-val);
67     }
68
69     public:
70     controllerNode():e3(0,0,1),F2W(4,4),hz(1000.0){
71         // PART 2: Initialize ROS callback handlers
72         xd = Eigen::Vector3d::Zero();
73         vd = Eigen::Vector3d::Zero();
74         ad = Eigen::Vector3d::Zero();
75         yawd = 0.0;
76         kx, kv, kr, komega = 0, 0, 0, 0;
77     }

```

```

72     des_state_sub = nh.subscribe("desired_state", 1, &
73     controllerNode::onDesiredState, this);
74     cur_state_sub = nh.subscribe("current_state", 1,
75     &controllerNode::onCurrentState, this);
76     propeller_speeds_pub = nh.advertise<mav_msgs::Actuators>("/
77     rotor_speed_cmds", 1);
78     control_timer = nh.createTimer(ros::Duration(1.0/hz),
79     &controllerNode::controlLoop, this);

76
77     // PART 6: Tune your gains!
78     nh.getParam("kx", kx);
79     nh.getParam("kv", kv);
80     nh.getParam("kr", kr);
81     nh.getParam("komega", komega);
82     ROS_INFO("Gain values:\n kx: %f \n kv: %f \n kr: %f \n komega: %f\n", kx,
83     kv, kr, komega);

84     // Initialize constants
85     m = 1.0;
86     cd = 1e-5;
87     cf = 1e-3;
88     g = 9.81;
89     d = 0.3;
90     J << 1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,1.0;

91
92     // F2W matrix
93     double d_by_sqrt2 = d/std::sqrt(2.0);
94     F2W <<
95         cf,           cf,           cf,           cf,
96         cf*d_by_sqrt2, cf*d_by_sqrt2, -cf*d_by_sqrt2, -cf*d_by_sqrt2,
97         -cf*d_by_sqrt2, cf*d_by_sqrt2, cf*d_by_sqrt2, -cf*d_by_sqrt2,
98         cd,           -cd,           cd,           -cd;
99 }
100
101 void onDesiredState(const trajectory_msgs::MultiDOFJointTrajectoryPoint&
102 des_state){
103     // PART 3: Objective - fill in xd, vd, ad, yawd
104     xd << des_state.transforms[0].translation.x,
105             des_state.transforms[0].translation.y,
106             des_state.transforms[0].translation.z;
107
108     vd << des_state.velocities[0].linear.x,
109             des_state.velocities[0].linear.y,
110             des_state.velocities[0].linear.z;
111
112     ad << des_state.accelerations[0].linear.x,
113             des_state.accelerations[0].linear.y,

```

```

113         des_state.accelerations[0].linear.z;
114
115     tf2::Quaternion quat;
116     tf2::fromMsg(des_state.transforms[0].rotation, quat);
117     yawd = tf2::getYaw(quat);
118 }
119
120 void onCurrentState(const nav_msgs::Odometry& cur_state){
121     // PART 4: Objective - fill in x, v, R and omega
122     // Position
123     x << cur_state.pose.pose.position.x,
124             cur_state.pose.pose.position.y,
125             cur_state.pose.pose.position.z;
126
127     // Velocity
128     v << cur_state.twist.twist.linear.x,
129             cur_state.twist.twist.linear.y,
130             cur_state.twist.twist.linear.z;
131
132     // Orientation
133     tf2::Quaternion quat;
134     tf2::fromMsg(cur_state.pose.pose.orientation, quat);
135     Eigen::Quaterniond eigen_quat(quat.w(), quat.x(), quat.y(), quat.z());
136     eigen_quat.normalize();
137     R = eigen_quat.toRotationMatrix();
138
139     // Angular velocity
140     Eigen::Vector3d omega_world;
141     omega_world << cur_state.twist.twist.angular.x,
142                     cur_state.twist.twist.angular.y,
143                     cur_state.twist.twist.angular.z;
144
145     omega = R.transpose() * omega_world;
146 }
147
148 void controlLoop(const ros::TimerEvent& t){
149     Eigen::Vector3d ex, ev, er, eomega;
150     // PART 5: Objective - Implement the controller!
151     ex = x - xd; // position error
152     ev = v - vd; // velocity error
153
154     // Rd matrix
155     Eigen::Vector3d F_des = -kx*ex - kv*ev + m*g*e3 + m*ad;
156     Eigen::Vector3d b3d = F_des.normalized();
157     Eigen::Vector3d b1d_desired(cos(yawd), sin(yawd), 0);
158

```

```

159     Eigen::Vector3d b2d = (b3d.cross(b1d_desired)).normalized();
160     Eigen::Vector3d b1d = (b2d.cross(b3d)).normalized();
161
162     Eigen::Matrix3d Rd;
163     Rd.col(0) = b1d;
164     Rd.col(1) = b2d;
165     Rd.col(2) = b3d;
166
167     er = 0.5 * Vee(Rd.transpose() * R - R.transpose() * Rd); // Orientation
168     eomega = omega; // Rotation-rate error
169
170     // Desired wrench
171     double f = (-kx * ex + -kv * ev + m * g * e3 + m * ad).dot(R * e3);
172     Eigen::Vector3d M = -kr * er - komega * eomega + omega.cross(J * omega);
173
174     // Recover the rotor speeds from the wrench
175     Eigen::Vector4d W;
176     W << f, M.x(), M.y(), M.z();
177     Eigen::Vector4d omega_sq = F2W.colPivHouseholderQr().solve(W);
178
179     Eigen::Vector4d rotor_speeds;
180     for (int i = 0; i < 4; i++) {
181         rotor_speeds(i) = signed_sqrt(omega_sq[i]);
182     }
183
184     // Populate and publish the control message
185     mav_msgs::Actuators control_msg;
186     control_msg.angular_velocities.clear();
187     for (int i = 0; i < 4; i++) {
188         control_msg.angular_velocities.push_back(rotor_speeds(i));
189     }
190     propeller_speeds_pub.publish(control_msg);
191 }
192 };
193
194 int main(int argc, char** argv){
195     ros::init(argc, argv, "controller_node");
196     controllerNode n;
197     ros::spin();
198 }
```