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.

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:

$$\min_{P(t)} \int_0^1 \left(P^{(1)}(t)\right)^2 dt \tag{1}$$

s.t.

$$P(0) = 0, \tag{2}$$

$$P(1) = 1, \tag{3}$$

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

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

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?**

   $\because P(t) = p_1 t$

   Let $t = 1 \therefore P(1) = p_1 \cdot 1 = p_1$
   $\because P(1) = 1 \therefore p_1 = 1$

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

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

   $\therefore$ 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 $\int_0^1 \left(P^{(1)}(t)\right)^2 dt$, 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 2×2 matrix.**

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

     $\therefore$ 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 2×2 matrix as p^TQp, we have

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

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

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

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

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

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

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

- **Solve the Quadratic Program (QP):**

$$\min_{\boldsymbol{p}}\boldsymbol{p^TQp}\ \boldsymbol{s.t.}\ \boldsymbol{Ap=b} \tag{5}$$

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?

$\because$ we have $\min_p p^TQp\Leftrightarrow\min_{p_1,p_2}\left(p_1^2+2p_1p_2+\frac{4}{3}p_2^2\right)$

and $Ap=b\Leftrightarrow p_1+p_2=1$

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

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).**

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

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

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

$\therefore$ we have

$$\begin{cases} \text{item } p_3^2 : \int_0^1 9t^4 dt = \frac{9}{5} \\ \text{item } p_2^2 : \int_0^1 4t^2 dt = \frac{4}{3} \\ \text{item } p_1^2 : \int_0^1 1 dt = 1 \\ \text{item } p_2 p_3 : \int_0^1 12t^3 dt = 3 \\ \text{item } p_1 p_3 : \int_0^1 6t^2 dt = 2 \\ \text{item } p_1 p_2 : \int_0^1 4t dt = 2 \end{cases}$$

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

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

- **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 $\frac{\partial L}{\partial P} - \frac{d}{dt}\frac{\partial L}{\partial P'} = 0$

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

  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 theroretical result derived from calculus of variations.

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

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

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 $P(t) = p_2 t^2 + p_1 t$,

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

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

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

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

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

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

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

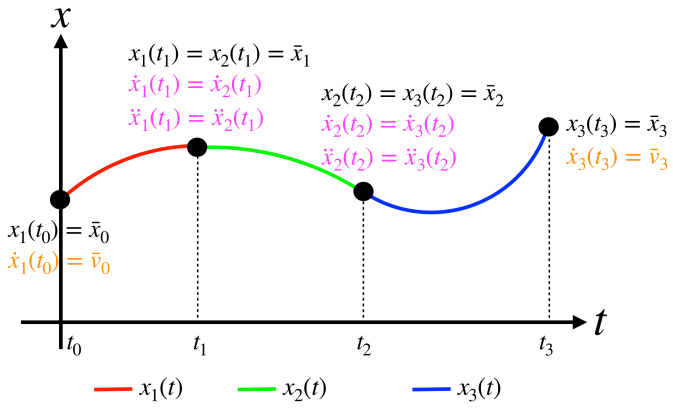
$\therefore Qd=\begin{bmatrix} 1 & 1 & 1 \\ 1 & \frac{4}{3} & \frac{3}{2} \\ 1 & \frac{3}{2} & \frac{9}{5} \end{bmatrix}\cdot\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}=\begin{bmatrix} 0 \\ -\frac{1}{6} \\ -\frac{1}{5} \end{bmatrix}$

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

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

1. **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.**



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

Step 1: Determine the number of Unknowns

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

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

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

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

$\because$ Each segment has $N + 1 = 8$ unknown coefficients and there are k segments.
$\therefore$ 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 constraints $\times$ k segments = $2k$ 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^{st} to 6^{th} derivatives).
$\therefore$ 6 constraints $\times$ $(k-1)$ points = $6(k-1)$ 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 (start) +3 (end) = 6 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 $\text{Total number of constraints} = 2rk$.
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^{st} to $(2r-2)^{\text{th}}$ derivatives at intermediate points).

(3) For *Fixed Derivative Constraints*:
$2(r-1)$ (Fixing 1^{st} to $(r-1)^{\text{th}}$ 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 π 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 2π 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 Enviroment 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:

```shell
cd ~/labs
git pull
```

   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:

```shell
cp -r ~/labs/lab4/. ~/vnav_ws/src
cd ~/vnav_ws
```

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

```shell
catkin build
source devel/setup.bash
```

   Then, copy the simulator to `~/vnav-builds/lab4/`.

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

3. **Simulator Setup**
   The Unity simulator binary was downloaded to `~/vnav-builds/lab4/` and configured with execution permissions:

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

4. **Execution Sequence**

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

```shell
# 1. Launch Unity Simulator
cd ~/vnav-builds/lab4/
./lab4.x86_64

# 2. Establish ROS Bridge
roslaunch tesse_ros_bridge tesse_quadrotor_bridge.launch

# 3. Start Trajectory Follower (Controller + Traj Generation)
roslaunch trajectory_generation traj_following.launch

# 4. Trigger Race (Waypoint Publisher)
roslaunch planner_pkg traj_gen.launch
```

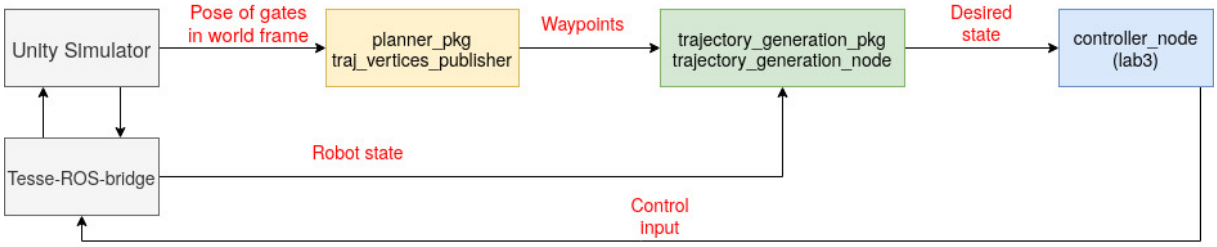


Figure 2: The Tesse ROS Bridge establishing communication between the Unity physics engine and the ROS control stack.

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.

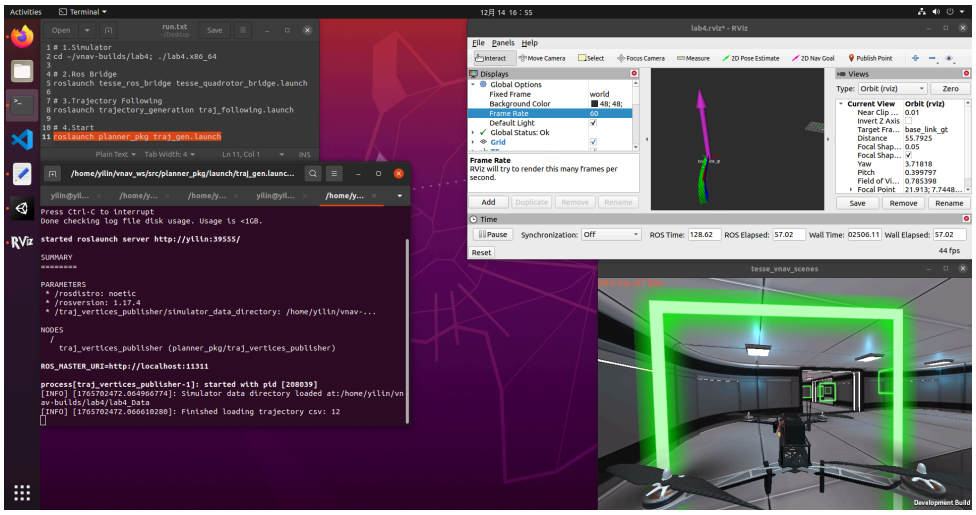


Figure 3: The quadrotor autonomously navigating through the gates in the Unity simulator environment.

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

This lab effectively bridged the gap between optimal control theory and real-world robotics application. Our analysis is divided into two perspectives: theoretical insights from the individual derivation and practical challenges encountered during the team implementation.

3.1 Theoretical Insights

The derivation of the Quadratic Programming (QP) formulation for trajectory generation revealed several key insights:

1. **Optimality of Polynomials:** The analytical solution to the minimum velocity problem ($r = 1$) confirmed that the optimal path connecting two points with minimal energy is a straight line. This aligns perfectly with the Euler-Lagrange equation ($P^{(2r)}(t) = 0$), where for $r = 1$, $P''(t) = 0$, implying a linear function. Even when higher-degree polynomials (degree 2 or 3) were allowed, the optimization naturally drove the coefficients of higher-order terms to zero to minimize the cost, demonstrating the robustness of the variational approach.
2. **Constraint Complexity:** Extending the problem to multi-segment minimum snap trajectories highlighted the combinatorial nature of constraints. We derived that for a system minimizing the r-th derivative over k segments, exactly $2rk$ constraints are required. This theoretical bound is critical for system design; missing a single continuity constraint at an intermediate waypoint can lead to a singular matrix, making the trajectory unsolvable.

3.2 Practical Implementation Challenges

Translating the math into C++ code for drone racing introduced engineering complexities not captured by the QP formulation alone:

1. **Yaw Discontinuities:** While position trajectories are defined in Euclidean space $\mathbb{R}^3$, orientation exists in the special orthogonal group $\mathrm{SO}(3)$ (represented here by Yaw angle). A naive implementation of yaw constraints led to "unnecessary spins" when the target angle crossed the $pmpi$ boundary. Implementing a **yaw unwrapping algorithm** was essential to linearize the angular space for the polynomial solver.
2. **Time-Scaling vs. Control Limits:** To achieve faster lap times (Extra Credit), we utilized time scaling. We observed a strict trade-off: compressing the trajectory time increases the required centripetal acceleration ($a = \frac{v^2}{r}$). If this exceeds the drone's physical thrust-to-weight ratio, the geometric controller saturates and fails to track. This required iterative tuning of both the time scaling factor and the controller gains (k_x, k_v, k_R) to find the boundary of stability.
3. **System Synchronization:** The frequency of the trajectory sampler proved vital. A low sampling rate (e.g., 5Hz) caused aliasing and jerky control inputs, while 100Hz provided smooth feed-forward derivatives, essential for high-speed tracking.

4 Conclusion

In this laboratory, we successfully developed a complete pipeline for autonomous quadrotor navigation, moving from mathematical derivation to simulation-based racing.

In the **Individual Work**, we established the theoretical foundation by formulating the trajectory generation as a Quadratic Program. We analytically verified that the QP solution matches the calculus of variations result and generalized the constraint counting rules for high-order minimum snap trajectories ($2rk$ constraints).

In the **Team Work**, we implemented this framework in ROS. By leveraging the `mav_trajectory_generation` library, we solved the multi-segment optimization problem to navigate a drone through a complex race course. Key technical achievements included:

- Developing a robust state machine with safety hover logic.
- Solving the yaw wrap-around problem for continuous orientation control.
- Fine-tuning the geometric controller and time-scaling parameters to achieve high-speed, collision-free flight.

The final simulation results validated our approach, demonstrating that polynomial optimization is a powerful tool for generating feasible, smooth, and aggressive trajectories for agile aerial robots.

5 Source Code

- ***trajectory_generation_node.cpp***

```cpp
#include <eigen_conversions/eigen_msg.h>
#include <geometry_msgs/PoseArray.h>
#include <nav_msgs/Odometry.h>
#include <ros/ros.h>
#include <trajectory_msgs/MultiDOFJointTrajectory.h>

#include <mav_trajectory_generation/polynomial_optimization_linear.h>
#include <mav_trajectory_generation/trajectory.h>

#include <tf/transform_datatypes.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <eigen3/Eigen/Dense>

class WaypointFollower {
  [[maybe_unused]] ros::Subscriber currentStateSub;
  [[maybe_unused]] ros::Subscriber poseArraySub;
  ros::Publisher desiredStatePub;

  // Current state
  Eigen::Vector3d x;  // current position of the UAV's c.o.m. in the world frame

  ros::Timer desiredStateTimer;

  ros::Time trajectoryStartTime;
  mav_trajectory_generation::Trajectory trajectory;
  mav_trajectory_generation::Trajectory yaw_trajectory;

  void onCurrentState(nav_msgs::Odometry const& cur_state) {
    // PART 1.1 – 将 ROS 消息转换为 Eigen 向量
    tf::pointMsgToEigen(cur_state.pose.pose.position, x);
  }

  void generateOptimizedTrajectory(geometry_msgs::PoseArray const& poseArray)
  {
    if (poseArray.poses.size() < 1) {
      ROS_ERROR("Must have at least one pose to generate trajectory!");
      trajectory.clear();
      yaw_trajectory.clear();
      return;
    }

```

```cpp
    if (!trajectory.empty()) return;

    const int D = 3;  // dimension of each vertex in the trajectory
    mav_trajectory_generation::Vertex start_position(D), end_position(D);
    mav_trajectory_generation::Vertex::Vector vertices;
    mav_trajectory_generation::Vertex start_yaw(1), end_yaw(1);
    mav_trajectory_generation::Vertex::Vector yaw_vertices;

    // Convert the pose array to a list of vertices
    // Start from the current position and zero orientation
    using namespace mav_trajectory_generation::derivative_order;
    start_position.makeStartOrEnd(x, SNAP);
    vertices.push_back(start_position);

    start_yaw.addConstraint(ORIENTATION, 0);
    yaw_vertices.push_back(start_yaw);

    double last_yaw = 0;

    for (auto i = 0; i < poseArray.poses.size(); ++i) {
      // PART - 1.2

      // --- 1. Process position vertex ---
      Eigen::Vector3d pos_eigen;
      tf::pointMsgToEigen(poseArray.poses[i].position, pos_eigen);

      mav_trajectory_generation::Vertex pos_vertex(D);

      // If it is the last point, it must be the end point, and the velocity acceleration is forced to be 0
      if (i == poseArray.poses.size() - 1) {
          pos_vertex.makeStartOrEnd(pos_eigen, SNAP);
      } else {
          // If it is an intermediate point, only position constraints are added to allow passage at a certain speed
          pos_vertex.addConstraint(POSITION, pos_eigen);
      }
      vertices.push_back(pos_vertex);


      // --- 2. Process yaw vertex ---
      double current_yaw = tf::getYaw(poseArray.poses[i].orientation);

      while (current_yaw - last_yaw > M_PI) current_yaw -= 2 * M_PI;
      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,
          mav_trajectory_generation::derivative_order::SNAP);
104       opt.solveLinear();
105
106       // Yaw
107       mav_trajectory_generation::PolynomialOptimization<N> yaw_opt(1);
108       yaw_opt.setupFromVertices(yaw_vertices, segment_times,
          mav_trajectory_generation::derivative_order::SNAP);
109       yaw_opt.solveLinear();
110
111       mav_trajectory_generation::Segment::Vector segments;
112       opt.getTrajectory(&trajectory);
113       yaw_opt.getTrajectory(&yaw_trajectory);
114       trajectoryStartTime = ros::Time::now();
115
116       ROS_INFO("Generated optimizes trajectory from %lu waypoints",
          vertices.size());
117     }
118
119     void publishDesiredState(ros::TimerEvent const& ev) {
120       if (trajectory.empty()) {
121           // If there is no trajectory yet, publish the current position as the
              desired position
122           trajectory_msgs::MultiDOFJointTrajectoryPoint hover_point;
123
124           hover_point.time_from_start = ros::Duration(0.0);
125
126           // 1. Set the position to the current position x
127           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 =
            0;
137         hover_point.velocities.push_back(velocity);
138
139         // 3. Set the acceleration to 0
140         geometry_msgs::Twist accel;
141         accel.linear.x = 0; accel.linear.y = 0; accel.linear.z = 0;
142         accel.angular.x = 0; accel.angular.y = 0; accel.angular.z = 0;
143         hover_point.accelerations.push_back(accel);
144
145         // Issue hover commands
146         desiredStatePub.publish(hover_point);
147         return;
148     }
149
150     //  PART 1.3
151     trajectory_msgs::MultiDOFJointTrajectoryPoint next_point;
152
153     // 1. Calculate the time from the beginning of the trajectory to the
        present
154     ros::Duration time_from_start = ros::Time::now() - trajectoryStartTime;
155     next_point.time_from_start = time_from_start;
156
157     double sampling_time = time_from_start.toSec();
158
159     // 2. The time is prevented from exceeding the total length of the
        trajectory
160     if (sampling_time > trajectory.getMaxTime())
161       sampling_time = trajectory.getMaxTime();
162
163     // Getting the desired state based on the optimized trajectory we found.
164     using namespace mav_trajectory_generation::derivative_order;
165     Eigen::Vector3d des_position = trajectory.evaluate(sampling_time,
        POSITION);
166     Eigen::Vector3d des_velocity = trajectory.evaluate(sampling_time,
        VELOCITY);
167     Eigen::Vector3d des_accel = trajectory.evaluate(sampling_time,
        ACCELERATION);
```

```
    Eigen::VectorXd des_orientation = yaw_trajectory.evaluate(sampling_time, ORIENTATION);

    // Populate next_point

    // A. Fill Transform (position + pose)
    geometry_msgs::Transform transform;
    tf::vectorEigenToMsg(des_position, transform.translation);
    tf::quaternionTFToMsg(tf::createQuaternionFromYaw(des_orientation(0)), transform.rotation);
    next_point.transforms.push_back(transform);

    // B. Fill Velocity (linear velocity + angular velocity)
    geometry_msgs::Twist velocity;
    tf::vectorEigenToMsg(des_velocity, velocity.linear);
    velocity.angular.x = 0;
    velocity.angular.y = 0;
    velocity.angular.z = 0;
    next_point.velocities.push_back(velocity);

    // C. Fill Acceleration (linear acceleration)
    geometry_msgs::Twist accel;
    tf::vectorEigenToMsg(des_accel, accel.linear);
    accel.angular.x = 0;
    accel.angular.y = 0;
    accel.angular.z = 0;
    next_point.accelerations.push_back(accel);

    desiredStatePub.publish(next_point);
  }

public:
  explicit WaypointFollower(ros::NodeHandle& nh) {
    currentStateSub = nh.subscribe(
        "/current_state", 1, &WaypointFollower::onCurrentState, this);
    poseArraySub = nh.subscribe("/desired_traj_vertices",
                                1,
                                &WaypointFollower::generateOptimizedTrajectory
                                this);
    desiredStatePub =
        nh.advertise<trajectory_msgs::MultiDOFJointTrajectoryPoint>(
            "/desired_state", 1);
    desiredStateTimer = nh.createTimer(
        ros::Rate(100), &WaypointFollower::publishDesiredState, this);
    desiredStateTimer.start();
  }
```

```
};

int main(int argc, char** argv) {
  ros::init(argc, argv, "trajectory_generation_node");
  ros::NodeHandle nh;

  WaypointFollower waypointFollower(nh);

  ros::spin();
  return 0;
}
```

- ***controller_node.cpp***

```cpp
#include <ros/ros.h>

#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/utils.h>
#include <mav_msgs/Actuators.h>
#include <nav_msgs/Odometry.h>
#include <trajectory_msgs/MultiDOFJointTrajectoryPoint.h>
#include <cmath>

#define PI M_PI

#include <eigen3/Eigen/Dense>
#include <tf2_eigen/tf2_eigen.h>

class controllerNode{
  ros::NodeHandle nh;

  // PART 1: Declare ROS callback handlers
  ros::Subscriber des_state_sub, cur_state_sub;
  ros::Publisher propeller_speeds_pub;
  ros::Timer control_timer;

  // Controller parameters
  double kx, kv, kr, komega;

  // Physical constants (we will set them below)
  double m;              // mass of the UAV
  double g;              // gravity acceleration
  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
      frame
41    Eigen::Vector3d v;     // current velocity of the UAV's c.o.m. in the world
      frame
42    Eigen::Matrix3d R;     // current orientation of the UAV
43    Eigen::Vector3d omega; // current angular velocity of the UAV's c.o.m. in
      the *body* frame
44
45    // Desired state
46    Eigen::Vector3d xd;    // desired position of the UAV's c.o.m. in the world
      frame
47    Eigen::Vector3d vd;    // desired velocity of the UAV's c.o.m. in the world
      frame
48    Eigen::Vector3d ad;    // desired acceleration of the UAV's c.o.m. in the
      world frame
49    double yawd;           // desired yaw angle
50
51    double hz;             // frequency of the main control loop
52
53    static Eigen::Vector3d Vee(const Eigen::Matrix3d& in){
54      Eigen::Vector3d out;
55      out << in(2,1), in(0,2), in(1,0);
56      return out;
57    }
58
59    static double signed_sqrt(double val){
60      return val > 0 ? sqrt(val) : -sqrt(-val);
61    }
62
63  public:
64    controllerNode():e3(0,0,1),F2W(4,4),hz(1000.0){
65        // PART 2: Initialize ROS callback handlers
66        xd = Eigen::Vector3d::Zero();
67        vd = Eigen::Vector3d::Zero();
68        ad = Eigen::Vector3d::Zero();
69        yawd = 0.0;
70        kx, kv, kr, komega = 0, 0, 0, 0;
71
```

```
72          des_state_sub = nh.subscribe("desired_state", 1, &
            controllerNode::onDesiredState, this);
73          cur_state_sub = nh.subscribe("current_state", 1,
            &controllerNode::onCurrentState, this);
74          propeller_speeds_pub = nh.advertise<mav_msgs::Actuators>("/
            rotor_speed_cmds", 1);
75          control_timer = nh.createTimer(ros::Duration(1.0/hz),
            &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:\nkx: %f \nkv: %f \nkr: %f \nkomega: %f\n", kx,
            kv, kr, komega);
83
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 << 1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0;
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&
        des_state){
102         //  PART 3: Objective - fill in xd, vd, ad, yawd
103         xd << des_state.transforms[0].translation.x,
104               des_state.transforms[0].translation.y,
105               des_state.transforms[0].translation.z;
106
107         vd << des_state.velocities[0].linear.x,
108               des_state.velocities[0].linear.y,
109               des_state.velocities[0].linear.z;
110
111         ad << des_state.accelerations[0].linear.x,
112               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
```

```
    Eigen::Vector3d b2d = (b3d.cross(b1d_desired)).normalized();
    Eigen::Vector3d b1d = (b2d.cross(b3d)).normalized();

    Eigen::Matrix3d Rd;
    Rd.col(0) = b1d;
    Rd.col(1) = b2d;
    Rd.col(2) = b3d;

    er = 0.5 * Vee(Rd.transpose() * R - R.transpose() * Rd); // Orientation error
    eomega = omega; // Rotation-rate error

    // Desired wrench
    double f = (-kx * ex + -kv * ev + m * g * e3 + m * ad).dot(R * e3);
    Eigen::Vector3d M = -kr * er - komega * eomega + omega.cross(J * omega);

    // Recover the rotor speeds from the wrench
    Eigen::Vector4d W;
    W << f, M.x(), M.y(), M.z();
    Eigen::Vector4d omega_sq = F2W.colPivHouseholderQr().solve(W);

    Eigen::Vector4d rotor_speeds;
    for (int i = 0; i < 4; i++) {
        rotor_speeds(i) = signed_sqrt(omega_sq[i]);
    }

    // Populate and publish the control message
    mav_msgs::Actuators control_msg;
    control_msg.angular_velocities.clear();
    for (int i = 0; i < 4; i++) {
        control_msg.angular_velocities.push_back(rotor_speeds(i));
    }
    propeller_speeds_pub.publish(control_msg);
  }
};

int main(int argc, char** argv){
  ros::init(argc, argv, "controller_node");
  controllerNode n;
  ros::spin();
}
```