Principles of Robot Autonomy I: Midterm 2025

Christopher Luey
11/01/25

Problem 1

(a) **Planar Quadrotor Dynamics Implementation.** The continuous-time model $f_c(x, u)$ unpacks $x = [p_x, v_x, p_y, v_y, \phi, \omega]^\top$ and $u = [T_1, T_2]^\top$. It sets $\dot{p}_x = v_x$ and $\dot{p}_y = v_y$, then applies the given equations to compute $\dot{v}_x = -(T_1 + T_2)\sin\phi/m$, $\dot{v}_y = (T_1 + T_2)\cos\phi/m - g$, $\dot{\phi} = \omega$, and $\dot{\omega} = (T_2 - T_1)\ell/I_{zz}$. The result is returned as a NumPy array that matches the state dimension. The notebook implementation exported to Appendix A.1 provides the full code listing.

The discrete update $f_d(x, u, \Delta t)$ uses the requested forward Euler step $x + \Delta t\, f_c(x, u)$. This keeps the discrete model consistent with the continuous dynamics for any Δt (default 0.02 s); the corresponding code appears in Appendix A.1.

(b) **Running Cost Specialization.** With diagonal weights $Q = \mathrm{diag}(q_p, q_v, q_p, q_v, q_\phi, q_\omega)$ and $R = r_T I_2$, the per-step cost for the hover equilibrium $x^\star = [1.5, 0, 1.0, 0, 0, 0]^\top$, $u^\star = [T_{\text{hov}}, T_{\text{hov}}]^\top$, $T_{\text{hov}} = mg/2$ is

$$\ell(x_t, u_t) = q_p(p_x - 1.5)^2 + q_v v_x^2 + q_p(p_y - 1.0)^2 + q_v v_y^2 + q_\phi \phi^2 + q_\omega \omega^2 + r_T\left[(T_1 - T_{\text{hov}})^2 + (T_2 - T_{\text{hov}})^2\right].$$

This scalar form is implemented in the `running_cost` helper whose source is captured in Appendix A.2.

(c) **Linearization about a Reference Trajectory.**

(i) Continuous and Discrete Jacobians

Differentiating f_c component-wise yields the continuous Jacobians

$$A_c(x, u) = \frac{\partial f_c}{\partial x} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{T_1+T_2}{m}\cos\phi & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{T_1+T_2}{m}\sin\phi & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B_c(x, u) = \frac{\partial f_c}{\partial u} = \begin{bmatrix} 0 & 0 \\ -\frac{\sin\phi}{m} & -\frac{\sin\phi}{m} \\ 0 & 0 \\ \frac{\cos\phi}{m} & \frac{\cos\phi}{m} \\ 0 & 0 \\ -\frac{\ell}{I_{zz}} & \frac{\ell}{I_{zz}} \end{bmatrix}.$$

Forward Euler converts these to discrete Jacobians at $(x_t^{\text{ref}}, u_t^{\text{ref}})$ via $A_t = I + \Delta t\, A_c(x_t^{\text{ref}}, u_t^{\text{ref}})$ and $B_t = \Delta t\, B_c(x_t^{\text{ref}}, u_t^{\text{ref}})$. The corresponding code used by both controllers is listed in Appendix A.3.

(ii) Affine Linearization Form

Substituting the Jacobians above into $f_d(x, u) = x + \Delta t\, f_c(x, u)$ and expanding about $(x_t^{\text{ref}}, u_t^{\text{ref}})$ gives

$$x_{t+1} \approx f_d(x_t^{\text{ref}}, u_t^{\text{ref}}) + A_t\,(x_t - x_t^{\text{ref}}) + B_t\,(u_t - u_t^{\text{ref}}),$$

which matches the deviation form $\delta x_{t+1} = A_t\,\delta x_t + B_t\,\delta u_t$. The algebraic steps are summarized in Appendix A.4.

(iii) Linearization Helper Implementation

The helper `linearize_about` calls the analytic routines above, returning (A_t, B_t) together with $f_d(x_t^{\text{ref}}, u_t^{\text{ref}})$ for use by both iLQR and GS-LQR; see Appendix A.5 for the exported source.

(d) **Interpreting Results: iLQR vs. GS-LQR.** Both controllers were tested on "real-world" dynamics that incorporate hidden thrust scaling errors (actuator miscalibration) and acceleration biases (unmodeled disturbances), simulating realistic model mismatch. The generated trajectories, time histories, and cost convergence plots are shown in Figures 7–10 in Appendix A.6.

(i) Trajectory Shape Comparison

The planar trajectory comparison (Figure 10) reveals a dramatic performance difference. The GS-LQR controller (blue) successfully navigates from the origin to approximately $(1.5\,\text{m}, 1.0\,\text{m})$, reaching the goal marker via a smooth, curved path. In contrast, the iLQR trajectory (red) catastrophically diverges to $(20\,\text{m}, -16\,\text{m})$—far beyond the intended goal.

Examining the time-series plots clarifies this failure. The iLQR roll angle ϕ monotonically decreases from $0°$ to $-270°$ (Figure 7), indicating continuous rotation in one direction due to persistent control asymmetry under model mismatch. Meanwhile, GS-LQR shows an initial dip to approximately $-45°$ before recovering, then maintaining ϕ within roughly $-5°$ to $+30°$ with a peak around $t = 1\,\text{s}$ (Figure 9). The fundamental difference: GS-LQR employs closed-loop feedback that corrects for model errors at each timestep, whereas iLQR executes its pre-computed control sequence open-loop with no error correction. Consequently, accumulated model mismatch destroys the iLQR trajectory.

(ii) Cost and Control Effort

The total closed-loop rollout costs confirm the performance gap:

- **iLQR**: $J = 181{,}377$
- **GS-LQR**: $J = 1{,}644$ (110× lower)

Control effort profiles further illustrate the disparity. The iLQR thrusts exhibit violent oscillations in the first $0.5\,\text{s}$, with spikes reaching $\sim 20\,\text{N}$ (four times hover thrust) and one rotor briefly dropping to nearly $0\,\text{N}$. After this initial transient, thrusts settle to a constant $\approx 5\,\text{N}$, but by then the trajectory has already diverged irreparably. In contrast, GS-LQR shows initial transients in the first $0.5\,\text{s}$ with thrusts reaching $10\,\text{N}$ and one rotor briefly dropping near $0\,\text{N}$, but the feedback controller recovers and stabilizes thrusts near hover values ($\approx 5\,\text{N}$) by $t = 1\,\text{s}$, maintaining smooth control thereafter. GS-LQR thus achieves dramatically superior tracking accuracy despite both controllers experiencing challenging initial dynamics.

(iii) Sensitivity to Increased Control Weight

To assess sensitivity to the control penalty r_T, both controllers were rerun with $r_T = 0.01$ (a tenfold increase from the baseline 0.001). The resulting trajectories and time histories appear in Figures 11–14 in Appendix A.7.

Quantitative comparison:

Controller	r_T	Cost J	Ratio
iLQR	0.001	181,377	—
iLQR	0.01	178,994	0.99×
GS-LQR	0.001	1,644	—
GS-LQR	0.01	1,648	1.00×

Qualitative observations:

- **iLQR behavior:** The higher control penalty produces slightly different initial transients (Figure 11), with T2 reaching 20 N instead of T1, but the open-loop execution still diverges catastrophically under model mismatch. The trajectories are nearly identical—both reach approximately $(20\,\text{m}, -16\,\text{m})$ with ϕ decreasing to $-270°$. The cost remains similarly high ($\sim$ 179k vs. 181k) because tracking errors dominate, not control effort.
- **GS-LQR robustness:** The closed-loop GS-LQR cost is nearly unchanged (1,644 vs. 1,648), demonstrating that the feedback controller already operates efficiently. Comparing Figures 9 and 13, the trajectories are nearly indistinguishable: both reach $(1.5\,\text{m}, 1.0\,\text{m})$ with similar transient behavior. The initial thrust spikes are similar in both cases, and steady-state thrusts remain near 5 N. The attitude profiles show similar transients (dip to $-40°$, peak at $+25-30°$), indicating minimal sensitivity to the control weight.
- **Key insight:** When a controller successfully tracks the reference (GS-LQR), increasing r_T has minimal impact because the feedback naturally keeps the system near equilibrium, making control effort inherently small. When tracking fails catastrophically (iLQR), the cost is dominated by state errors (position and attitude divergence), not control effort, so r_T changes have negligible effect on total cost. The figures visually confirm these trajectories are essentially unchanged despite the tenfold increase in control penalty.

Overall, this sensitivity study confirms that *feedback* is the critical factor for robustness under model mismatch, while control penalty tuning is a secondary consideration that primarily affects maneuver aggressiveness in well-performing controllers.

Problem 2

(a) **Position of the calibration board in the world frame.**

We are given the following coordinate frame specifications:

- **World frame** W: Origin at ground level with $\hat{z}_W$ pointing upwards and $\hat{x}_W$, $\hat{y}_W$ across the ground (right-hand rule).
- **Camera frame** C: Origin 30 cm above W's origin. The camera's $\hat{x}_C$ points directly downwards, $\hat{y}_C$ points to the camera's left, and $\hat{z}_C$ points out from the lens (parallel to $\hat{x}_W$).
- **Board frame** B **(in camera coordinates)**: Located at 0 cm in C's x axis, 20 cm in C's y axis, and 100 cm in C's z axis.

Step 1: Rotation matrix from C to W.

From the orientation constraints:

- $\hat{z}_C$ is parallel to $\hat{x}_W \Rightarrow \hat{x}_W = \hat{z}_C$
- $\hat{x}_C$ points downwards $\Rightarrow \hat{z}_W = -\hat{x}_C$
- By the right-hand rule: $\hat{y}_W = \hat{y}_C$

The rotation matrix R_{WC} (expressing C's axes in W) is:

$$R_{WC} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Step 2: Position transformation.

The position of the camera origin in W is:

$$p_{WC} = \begin{bmatrix} 0 \\ 0 \\ 0.3 \end{bmatrix} \text{ m}$$

The position of the board origin in C is:

$$p_{CB} = \begin{bmatrix} 0 \\ 0.2 \\ 1.0 \end{bmatrix} \text{ m}$$

The position of the board in W is:

$$p_{WB} = p_{WC} + R_{WC}\, p_{CB} = \begin{bmatrix} 0 \\ 0 \\ 0.3 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0.2 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0.3 \end{bmatrix} + \begin{bmatrix} 1.0 \\ 0.2 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.2 \\ 0.3 \end{bmatrix} \text{ m}$$

Answer: The position of the lower left corner of the calibration board (origin of frame B) in the world frame is:

$$\boxed{p_{WB} = \begin{bmatrix} 1.0 \\ 0.2 \\ 0.3 \end{bmatrix} \text{ m} = \begin{bmatrix} 100 \\ 20 \\ 30 \end{bmatrix} \text{ cm}}$$

(b) **Conversion between board frame and world frame.**

We must find the homogeneous transformation matrix T_{WB} that transforms coordinates from frame B to frame W.

Step 1: Determine rotation R_{CB} (from B to C).

We are given two orientation constraints:

- The y axis of B points opposite to the x axis of C: $\hat{y}_B = -\hat{x}_C$
- The z axis of B is rotated 150° about $\hat{y}_B$ relative to the z axis of C

First, express $\hat{y}_B$ in frame C:

$$\hat{y}_B = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}_C$$

Next, $\hat{z}_B$ is obtained by rotating $\hat{z}_C = [0,0,1]_C^\top$ by 150° about $\hat{y}_B = [-1,0,0]_C^\top$. Rotation about $[-1,0,0]^\top$ by 150° is equivalent to rotation about $[1,0,0]^\top$ by −150°:

$$R_x(-150^\circ) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-150^\circ) & -\sin(-150^\circ) \\ 0 & \sin(-150^\circ) & \cos(-150^\circ) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\frac{\sqrt{3}}{2} & \frac{1}{2} \\ 0 & -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix}$$

Applying this to $\hat{z}_C$:

$$\hat{z}_B = R_x(-150^\circ)\,\hat{z}_C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\frac{\sqrt{3}}{2} & \frac{1}{2} \\ 0 & -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{bmatrix}_C$$

Using the right-hand rule to find $\hat{x}_B = \hat{y}_B \times \hat{z}_B$:

$$\hat{x}_B = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ \frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} 0 \cdot (-\frac{\sqrt{3}}{2}) - 0 \cdot (-\frac{1}{2}) \\ 0 \cdot 0 - (-1) \cdot (-\frac{\sqrt{3}}{2}) \\ (-1) \cdot (-\frac{1}{2}) - 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\sqrt{3}}{2} \\ -\frac{1}{2} \end{bmatrix}_C$$

Therefore, the rotation matrix R_{CB} (columns are B's axes expressed in C) is:

$$R_{CB} = \begin{bmatrix} \hat{x}_B & \hat{y}_B & \hat{z}_B \end{bmatrix}_C = \begin{bmatrix} 0 & -1 & 0 \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \end{bmatrix}$$

Step 2: Compute rotation R_{WB}.

Using the chain rule for rotations: $R_{WB} = R_{WC}\,R_{CB}$

$$R_{WB} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \end{bmatrix}$$

Step 3: Homogeneous transformation matrix.

The homogeneous transformation matrix is:

$$\boxed{T_{WB} = \begin{bmatrix} -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} & 1.0 \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} & 0.2 \\ 0 & 1 & 0 & 0.3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad T_{WB} = \begin{bmatrix} -0.5 & 0 & -0.866 & 1.0 \\ -0.866 & 0 & 0.5 & 0.2 \\ 0 & 1 & 0 & 0.3 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

where the rotation block R_{WB} captures the board's orientation and the translation vector p_{WB} from part (a) specifies its position.

(c) **Finding the position of the robot.**

We are given the following additional information:

- **Robot frame R and view frame V**: Both have $\hat{x}$ facing the front of the robot and $\hat{z}$ facing upwards.
- **Position of V in R**: $p_{RV} = [10, 0, 4]^\top$ cm $= [0.1, 0, 0.04]^\top$ m.
- **Orientation of V relative to B**: The $\hat{z}_V$ (upwards) is parallel to $\hat{y}_B$, and $\hat{x}_V$ is rotated $(180 + a)^\circ$ from $\hat{z}_B$ about these vectors, where a is the last digit of the SUID.
- **Position of B in V**: $p_{VB} = [70, 0, 22.6]^\top$ cm $= [0.7, 0, 0.226]^\top$ m.

For this solution, I use $a = 1$ (last digit of my SUID 006952321).

Step 1: Determine rotation R_{VB} (from B to V).

Given that $\hat{z}_V$ is parallel to $\hat{y}_B$, we have $\hat{z}_V = \hat{y}_B$.

The constraint that $\hat{x}_V$ is rotated $(180 + a)^\circ = 181^\circ$ from $\hat{z}_B$ about the axis $\hat{z}_V = \hat{y}_B$ means:

$$\hat{x}_V = R_y(181^\circ)\, \hat{z}_B$$

where the rotation is about the y-axis of frame B.

Using $\cos(181^\circ) \approx -0.9998$ and $\sin(181^\circ) \approx -0.0175$:

$$R_y(181^\circ) = \begin{bmatrix} \cos(181^\circ) & 0 & \sin(181^\circ) \\ 0 & 1 & 0 \\ -\sin(181^\circ) & 0 & \cos(181^\circ) \end{bmatrix} \approx \begin{bmatrix} -0.9998 & 0 & -0.0175 \\ 0 & 1 & 0 \\ 0.0175 & 0 & -0.9998 \end{bmatrix}$$

Therefore, $\hat{x}_V = R_y(181^\circ) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}_B \approx \begin{bmatrix} -0.0175 \\ 0 \\ -0.9998 \end{bmatrix}_B$

With $\hat{y}_V = \hat{z}_V \times \hat{x}_V = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}_B \times \begin{bmatrix} -0.0175 \\ 0 \\ -0.9998 \end{bmatrix}_B \approx \begin{bmatrix} -0.9998 \\ 0 \\ 0.0175 \end{bmatrix}_B$

The rotation matrix R_{VB} (columns are B's axes in V) is:

$$R_{VB} = \begin{bmatrix} \hat{x}_B & \hat{y}_B & \hat{z}_B \end{bmatrix}_V = \begin{bmatrix} -0.0175 & -0.9998 & 0 \\ 0 & 0 & 1 \\ -0.9998 & 0.0175 & 0 \end{bmatrix}^\top = \begin{bmatrix} -0.0175 & 0 & -0.9998 \\ -0.9998 & 0 & 0.0175 \\ 0 & 1 & 0 \end{bmatrix}$$

Step 2: Compute R_{WV} and R_{WR}.

Since R and V have the same orientation (both x forward, z up):

$$R_{RV} = R_{VR} = I_{3\times 3}$$

Using $R_{WV} = R_{WB}\, R_{BV} = R_{WB}\, R_{VB}^{\top}$:

$$R_{WV} = \begin{bmatrix} -0.5 & 0 & -0.866 \\ -0.866 & 0 & 0.5 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.0175 & -0.9998 & 0 \\ 0 & 0 & 1 \\ -0.9998 & 0.0175 & 0 \end{bmatrix} \approx \begin{bmatrix} 0.8746 & 0.4848 & 0 \\ -0.4848 & 0.8746 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Since $R_{WR} = R_{WV}$, we have the same rotation matrix for the robot.

Step 3: Find position p_{WR}.

Using the transformation chain:

$$p_{WB} = p_{WR} + R_{WR}\, p_{RV} + R_{WV}\, p_{VB}$$

Rearranging:

$$p_{WR} = p_{WB} - R_{WR}\, p_{RV} - R_{WV}\, p_{VB}$$

Since $R_{WR} = R_{WV}$:

$$p_{WR} = p_{WB} - R_{WV}\,(p_{RV} + p_{VB})$$

Computing $p_{RV} + p_{VB} = \begin{bmatrix} 0.1 \\ 0 \\ 0.04 \end{bmatrix} + \begin{bmatrix} 0.7 \\ 0 \\ 0.226 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0 \\ 0.266 \end{bmatrix}$ m

$$R_{WV}\,(p_{RV} + p_{VB}) \approx \begin{bmatrix} 0.8746 & 0.4848 & 0 \\ -0.4848 & 0.8746 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.8 \\ 0 \\ 0.266 \end{bmatrix} = \begin{bmatrix} 0.700 \\ -0.388 \\ 0.266 \end{bmatrix} \text{ m}$$

$$p_{WR} = \begin{bmatrix} 1.0 \\ 0.2 \\ 0.3 \end{bmatrix} - \begin{bmatrix} 0.700 \\ -0.388 \\ 0.266 \end{bmatrix} \approx \begin{bmatrix} 0.300 \\ 0.588 \\ 0.034 \end{bmatrix} \text{ m}$$

Answer: With $a = 1$, the position of the robot (origin of frame R) in the world frame is:

$$\boxed{p_{WR} \approx \begin{bmatrix} 0.300 \\ 0.588 \\ 0.034 \end{bmatrix} \text{ m} = \begin{bmatrix} 30.0 \\ 58.8 \\ 3.4 \end{bmatrix} \text{ cm}}$$

(d) **Axis of wheel rotation.**

Answer: Option 1: Vector $\hat{y}$ in coordinate frame R

(e) **Axis of robot movement.**

Answer: Option 2: Vector $\hat{x}$ in coordinate frame R

Problem 3

3a. Robust Line Fitting with RANSAC

(i) Implementation and Results

Using $N_{\text{iter}} = 150$, $\epsilon = 0.08$, and seed 274, the RANSAC routine identified a dominant line with slope $\hat{m} = 1.8921$ and intercept $\hat{b} = 0.7693$. Twenty-six of the 100 samples satisfied the inlier threshold, matching the green points in Fig. 1. The full implementation used to generate these values is provided in Appendix B.1.

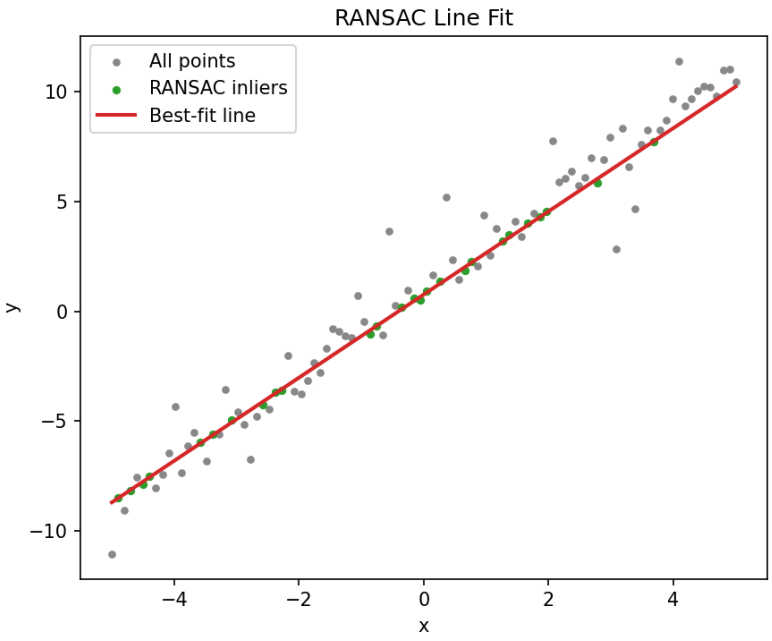

Figure 1: RANSAC line estimate highlighting inliers (green) and the best-fit line (red).

3b. Umeyama Point-Cloud Alignment

(i) Estimated Transform

Applying the closed-form Umeyama method (see Appendix B.2) yields

$$R = \begin{bmatrix} 0.7676 & -0.0013 & 0.6409 \\ 0.2347 & 0.9311 & -0.2791 \\ -0.5964 & 0.3647 & 0.7150 \end{bmatrix}, \quad t = \begin{bmatrix} 0.0195 \\ 0.0300 \\ -0.0107 \end{bmatrix}.$$

(ii) Alignment Accuracy

Transforming the source cloud with (R, t) results in a root-mean-square error of 0.0653 m across all correspondences, confirming that the recovered transform nearly reproduces the noisy target.

(iii) Visualization

Figure 2 contrasts the raw and aligned point clouds. After alignment, the transformed source overlaps the target in all three axes, while the pre-alignment view shows the original rotation and translation offsets.

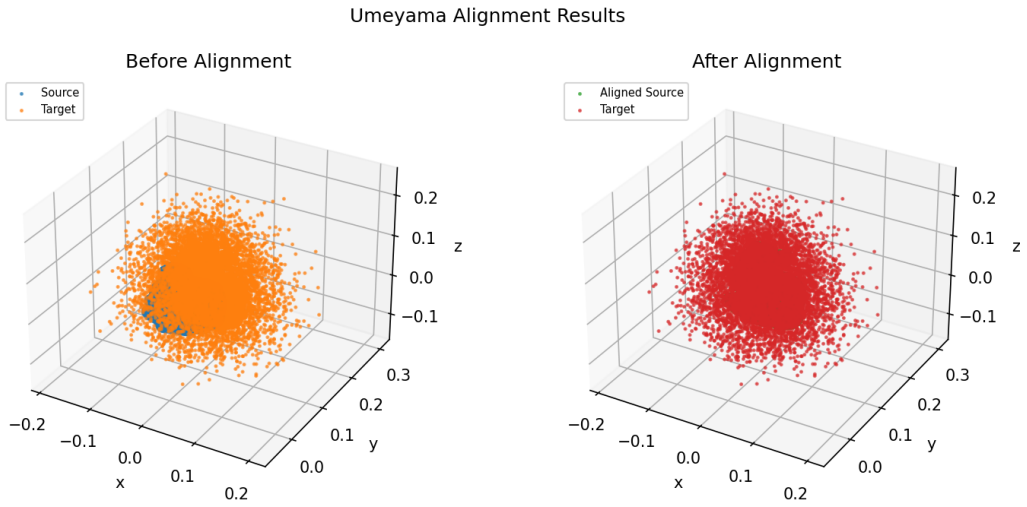

Figure 2: Umeyama alignment before (left) and after (right) applying (R, t) to the Stanford bunny source cloud.

3c. Robust Registration via RANSAC

(i) Consensus Set

With $s = 3$, $N_{\text{iter}} = 200$, $\tau = 0.002$, and seed 274, the RANSAC loop retained 1635/8171 correspondences as inliers.

(ii) Refit Transform and Accuracy

Refitting Umeyama on the inliers produced the summary below:

- **Final rotation** $\hat{R}$

$$\hat{R} = \begin{bmatrix} 0.7639 & -0.0110 & 0.6453 \\ 0.2380 & 0.9342 & -0.2659 \\ -0.5999 & 0.3567 & 0.7162 \end{bmatrix}$$

- **Final translation** $\hat{t}$

$$\hat{t} = \begin{bmatrix} 0.0200 \\ 0.0300 \\ -0.0100 \end{bmatrix}$$

- **Inlier RMSE**: 5.64×10^{-17} m (numerically zero within floating-point tolerance).

(iii) Visualization and Comparison

Figure 3 overlays the raw clouds and the aligned inlier set. The left panel shows the original source (blue) and target (orange) point clouds before alignment. The right panel displays the final result: aligned source inliers (green) and target inliers (red) overlap tightly, while rejected correspondences appear in grey.

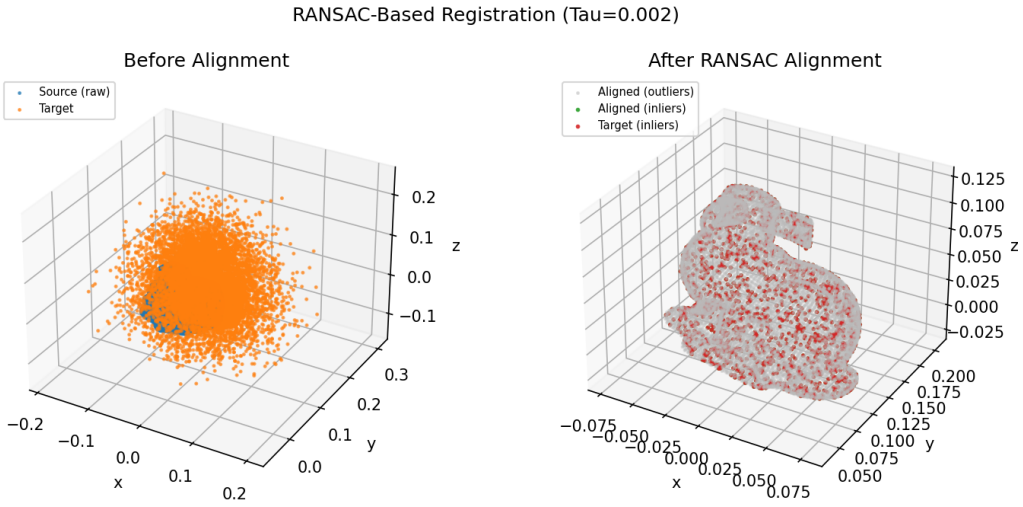

Figure 3: RANSAC registration. Left: original source (blue) and target (orange). Right: aligned source inliers (green) and target inliers (red); grey points denote rejected correspondences.

Comparison to direct Umeyama: The RANSAC-based registration substantially improves alignment quality compared to applying Umeyama to all correspondences directly. The global fit from part (b) achieved RMSE = 0.0653 m and exhibits visible misalignment in Figure 2, whereas the RANSAC-refined model reduces inlier RMSE to 5.64×10^{-17} m (numerically zero) and produces near-perfect overlap in Figure 3 by identifying and discarding ∼80% of correspondences as outliers.

Problem 4

(a) **Body-frame half-space formulation.**

(i) Scalar collision tests

We express each face of the drone's bounding box as an independent half-space test on the query point $q_b = (q_{b,x}, q_{b,y})^\top$ expressed in the drone body frame. The rectangle spans b_x along the body x-axis and b_y along the body y-axis, so the four scalar inequalities are

$$q_{b,x} - \frac{b_x}{2} \leq 0, \quad -q_{b,x} - \frac{b_x}{2} \leq 0, \quad q_{b,y} - \frac{b_y}{2} \leq 0, \quad -q_{b,y} - \frac{b_y}{2} \leq 0.$$

A collision occurs precisely when all four conditions are satisfied simultaneously.

(ii) Matrix representation

Stacking the scalar inequalities produces the requested matrix form $Aq_b + b \leq 0$ with

$$A = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}, \qquad b = -\frac{1}{2}\begin{bmatrix} b_x \\ b_x \\ b_y \\ b_y \end{bmatrix}.$$

These coefficients are exactly those used inside the collision checker for the RRT implementation; the helper sourcing this computation is exported in Appendix C.1.

(b) **World-to-body transformation.**

(i) Translation vector τ

The pose $x = (x, y, \theta)^\top$ centres the body frame at the drone's geometric centre, so obstacle points are first shifted by

$$\tau = \begin{bmatrix} x \\ y \end{bmatrix}.$$

This translation enforces $q - \tau$ as the world-frame displacement from the body origin.

(ii) Rotation matrix R

A positive θ corresponds to a counter-clockwise rotation, yielding the standard planar rotation matrix

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Applying R rotates vectors about the origin of B by θ.

(iii) Transforming obstacle points

Combining the translation and rotation gives $q_b = R^\top(q - \tau)$ for any obstacle $q \in \mathbb{R}^2$. The code implementation of τ and R appears in Appendix C.2.

(c) **World-frame inequality.** Substituting $q_b = R^\top(q-\tau)$ from part (b) into the body-frame inequality $Aq_b + b \leq 0$ from part (a) yields

$$AR^\top(q - \tau) + b \leq 0 \quad \Longrightarrow \quad \underbrace{AR^\top}_{A_W} q + \underbrace{\left(b - AR^\top \tau\right)}_{b_W} \leq 0.$$

This gives the world-frame collision test $A_W q + b_W \leq 0$, where:

$$A_W = AR^\top \in \mathbb{R}^{4\times 2}, \qquad b_W = b - A_W \tau \in \mathbb{R}^4.$$

The matrix A_W rotates each half-space normal into the world frame, while b_W shifts the offsets to account for the drone's translation. The code implementation of this computation appears in Appendix C.3.

(d) **RRT implementation and evaluation.** The `is_free_state` method of the `MidtermRRT` class in `P4_rrt.py` was completed to implement the collision detection logic derived in parts (a)–(c). Given a pose (x, y, θ) and obstacle point cloud, the method computes translation $\tau = [x, y]^\top$ and rotation matrix R (part b), constructs body-frame matrices A and b (part a), transforms them to world coordinates as $A_W = AR^\top$ and $b_W = b - A_W\tau$ (part c), then evaluates $A_W q + b_W \leq 0$ for each obstacle point. If all four inequalities are satisfied for any point, a collision is detected and the method returns `False`. The complete implementation is provided in Appendix C.1.

With `is_free_state` completed, the notebook `midterm_p4_rrt.ipynb` was executed to test the collision-aware RRT planner on three different obstacle maps. The notebook cells were run sequentially, generating the exploration trees and solution paths shown in Figures 4–6. Detailed interpretations of each map's results follow.

Map 1: Cluster maze

The first map contains dense clusters of obstacles. The RRT was configured with step size $\epsilon = 10\,\text{px}$ and a $20 \times 10\,\text{px}$ bounding box for the drone. Starting from $(10, 20)$ with initial heading $\pi/4$ and targeting $(100, 90)$ with heading $\pi/6$, the planner successfully finds a collision-free path containing 19 nodes (Fig. 4).

The blue tree shows the exploration process, revealing how the RRT grows toward the goal while avoiding obstacle clusters. The green path represents the final solution extracted by backtracking from the goal node. The collision checker effectively prevents the rotated rectangle from intersecting any obstacle points, confirming that the world-frame inequality $A_W q + b_W \leq 0$ correctly accounts for both translation and rotation of the drone body.

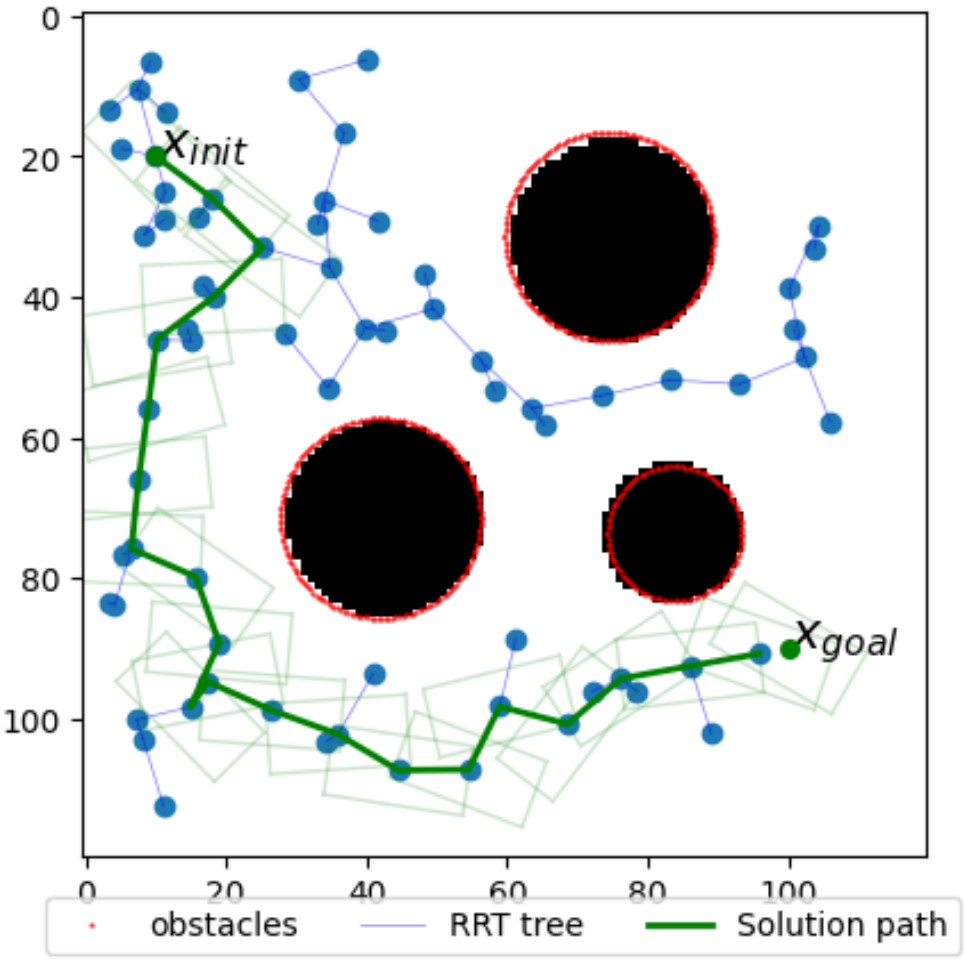

Figure 4: Cluster maze: the RRT (blue tree, green solution) navigates between obstacle clusters without intersecting any sampled points.

Map 2: Narrow-gap maze

The second map features a challenging narrow corridor that tests the planner's ability to navigate tight spaces. The drone footprint was reduced to 10×5 px and the step size decreased to $\epsilon = 5$ px to enable finer-grained exploration. Starting from $(20, 30)$ with heading $-\pi/6$ and targeting $(90, 25)$ with heading 0, the planner discovers a 31-node solution that threads through the gap (Fig. 5).

This result demonstrates the critical importance of the orientation-aware collision detection implemented in `is_free_state`. The planner must carefully adjust the drone's heading θ to align the rectangular body with the corridor opening. The world-frame collision test $A_W q + b_W \leq 0$ properly accounts for this rotation via the matrix $A_W = AR^\top$, allowing the planner to find configurations where the rotated rectangle clears the walls. Without this capability, a point-mass planner would incorrectly assume the gap is traversable, leading to collisions when the drone's physical extent is considered.

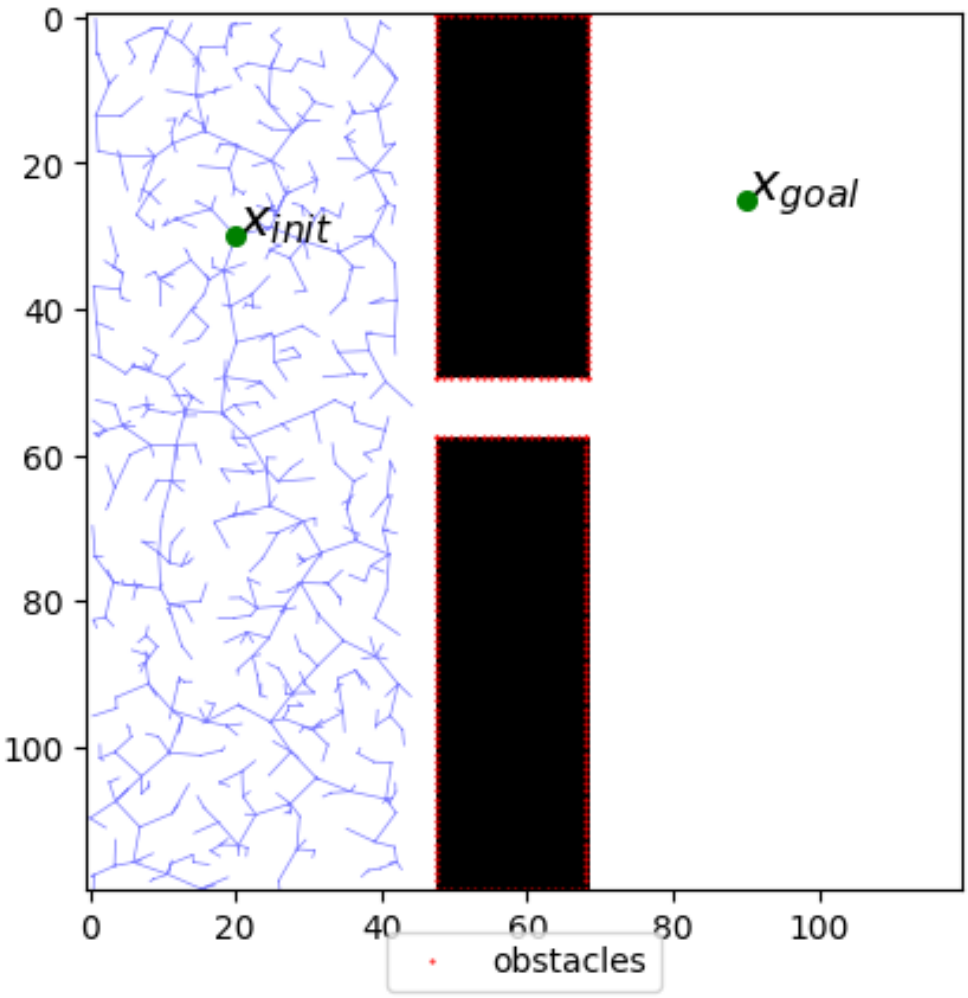

Figure 5: Gap maze: the planner discovers an orientation-aware path that clears the narrow opening.

Map 3: Super Mario Land Level 4-3

The third map is a large-scale test inspired by Super Mario Land Level 4-3, containing 3,110 obstacle points extracted from the classic Game Boy level. Despite the dense environment, the RRT with drone dimensions 15×12 px and step size $\epsilon = 10$ px successfully plans a 53-node path from Mario's starting position $(183, 37)$ to Princess Daisy at $(545, 59)$ (Fig. 6).

This result validates the computational efficiency of the world-frame collision checking approach. Rather than transforming all 3,110 obstacle points into the body frame for each candidate pose, the implementation precomputes $A_W = AR^\top$ and $b_W = b - A_W\tau$ once per pose, then performs only matrix-vector multiplications $A_W q$ for each obstacle point. This vectorized approach keeps the collision detection tractable even for large point clouds, enabling real-time planning in environments with thousands of obstacles. The successful path shown in the figure demonstrates that the half-space formulation from parts (a)–(c) scales effectively to practical robotics applications.

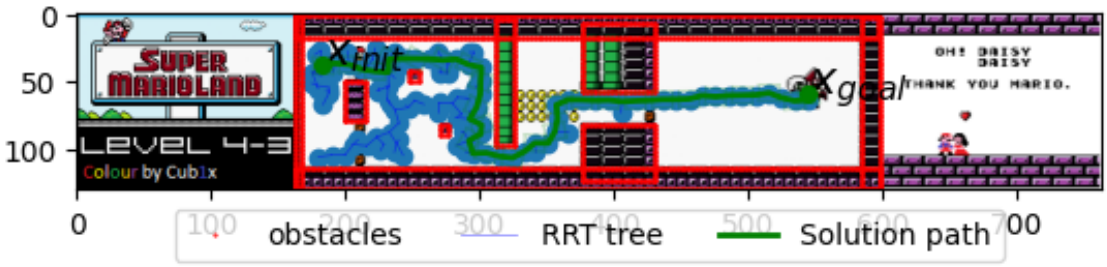

Figure 6: Super Mario Land Level 4-3 inspired map: the RRT weaves past dense obstacles to reach Daisy safely.

A Problem 1 Details

A.1 1a. Planar Quadrotor Dynamics Listing

```
def f_continuous(x, u):
    ##########################################################################
    ########################## YOUR CODE HERE ################################
    # Write the continuous dynamics for the planar drone
    T1, T2 = u
    px, vx, py, vy, phi, omega = x
    thrust_sum = T1 + T2
    dx = np.zeros_like(x, dtype=float)
    dx[0] = vx
    dx[1] = -(thrust_sum / m) * np.sin(phi)
    dx[2] = vy
    dx[3] = (thrust_sum / m) * np.cos(phi) - g
    dx[4] = omega
    dx[5] = ((T2 - T1) * ell) / Izz
    ##########################################################################
    return dx

def f_discrete(x, u, dt=dt_default):
    ##########################################################################
    ########################## YOUR CODE HERE ################################
    # Write the discrete dynamics for the planar drone
    dx = x + dt * f_continuous(x, u)
    ##########################################################################
    return dx
```

A.2 1b. Running Cost Helper

```
def running_cost(x, u):
  px, vx, py, vy, phi, omega = x
  T1, T2 = u
  ex = px - goal[0]
  ey = py - goal[2]
  cost = (
    q_p * ex**2 +
    q_v * vx**2 +
    q_p * ey**2 +
    q_v * vy**2 +
    q_phi * phi**2 +
    q_omega * omega**2 +
    r_T * ((T1 - T_hov)**2 + (T2 - T_hov)**2)
  )
  return cost
```

A.3 1c. Continuous and Discrete Jacobians

Differentiating each component of $f_c(x, u)$ with respect to x and u yields the matrices A_c and B_c used for first-order linearization. Applying forward Euler gives $A_t = I + \Delta t\, A_c$ and $B_t = \Delta t\, B_c$, which are exported below for exact reuse in the controllers.

```
def continuous_jacobians(x, u):
    px, vx, py, vy, phi, omega = x
    T1, T2 = u
    A_c = np.zeros((6, 6))
    B_c = np.zeros((6, 2))

    ############################################################################
    ########################## YOUR CODE HERE ##################################
    A_c[0, 1] = 1.0
    A_c[1, 4] = -((T1 + T2) / m) * np.cos(phi)
    A_c[2, 3] = 1.0
    A_c[3, 4] = -((T1 + T2) / m) * np.sin(phi)
    A_c[4, 5] = 1.0

    B_c[1, 0] = -np.sin(phi) / m
    B_c[1, 1] = -np.sin(phi) / m
    B_c[3, 0] = np.cos(phi) / m
    B_c[3, 1] = np.cos(phi) / m
    B_c[5, 0] = -ell / Izz
    B_c[5, 1] = ell / Izz
    ############################################################################

    return A_c, B_c

def discrete_jacobians(A_c, B_c, dt):
    ############################################################################
    ########################## YOUR CODE HERE ##################################
    state_dim = A_c.shape[0]
    A_t = np.eye(state_dim) + dt * A_c
    B_t = dt * B_c
    ############################################################################

    return A_t, B_t
```

A.4 1c.2 Affine Linearization Form

Starting from $f_d(x, u) = x + \Delta t\, f_c(x, u)$, a first-order Taylor expansion about $(x_t^{\text{ref}}, u_t^{\text{ref}})$ produces

$$x_{t+1} \approx f_d(x_t^{\text{ref}}, u_t^{\text{ref}}) + A_t\,(x_t - x_t^{\text{ref}}) + B_t\,(u_t - u_t^{\text{ref}}),$$

with A_t and B_t defined as in Appendix A.3. Rearranging gives the deviation dynamics $\delta x_{t+1} = A_t\,\delta x_t + B_t\,\delta u_t$ used in both iLQR and GS-LQR.

A.5 1c.3 Linearization Helper

The `linearize_about` helper wraps the analytic Jacobians, returning the affine model coefficients and the nominal next state for any reference pair $(x_t^{\text{ref}}, u_t^{\text{ref}})$.

```
def linearize_about(x_ref, u_ref, dt=dt_default):
    """
    Affine first-order discrete-time expansion about (x_ref, u_ref):
        x_ref{t+1} ~= f_d(x_ref, u_ref) + A_t (x_ref) + B_t (u_ref)
    """
    A_c, B_c = continuous_jacobians(x_ref, u_ref)
    ########################################################################
    ########################## YOUR CODE HERE ##############################
    A_t, B_t = discrete_jacobians(A_c, B_c, dt)
    x_next_ref = f_discrete(x_ref, u_ref, dt)
    ########################################################################

    return A_t, B_t, x_next_ref
```

A.6 1d.1 Baseline Controller Results ($r_T = 0.001$)

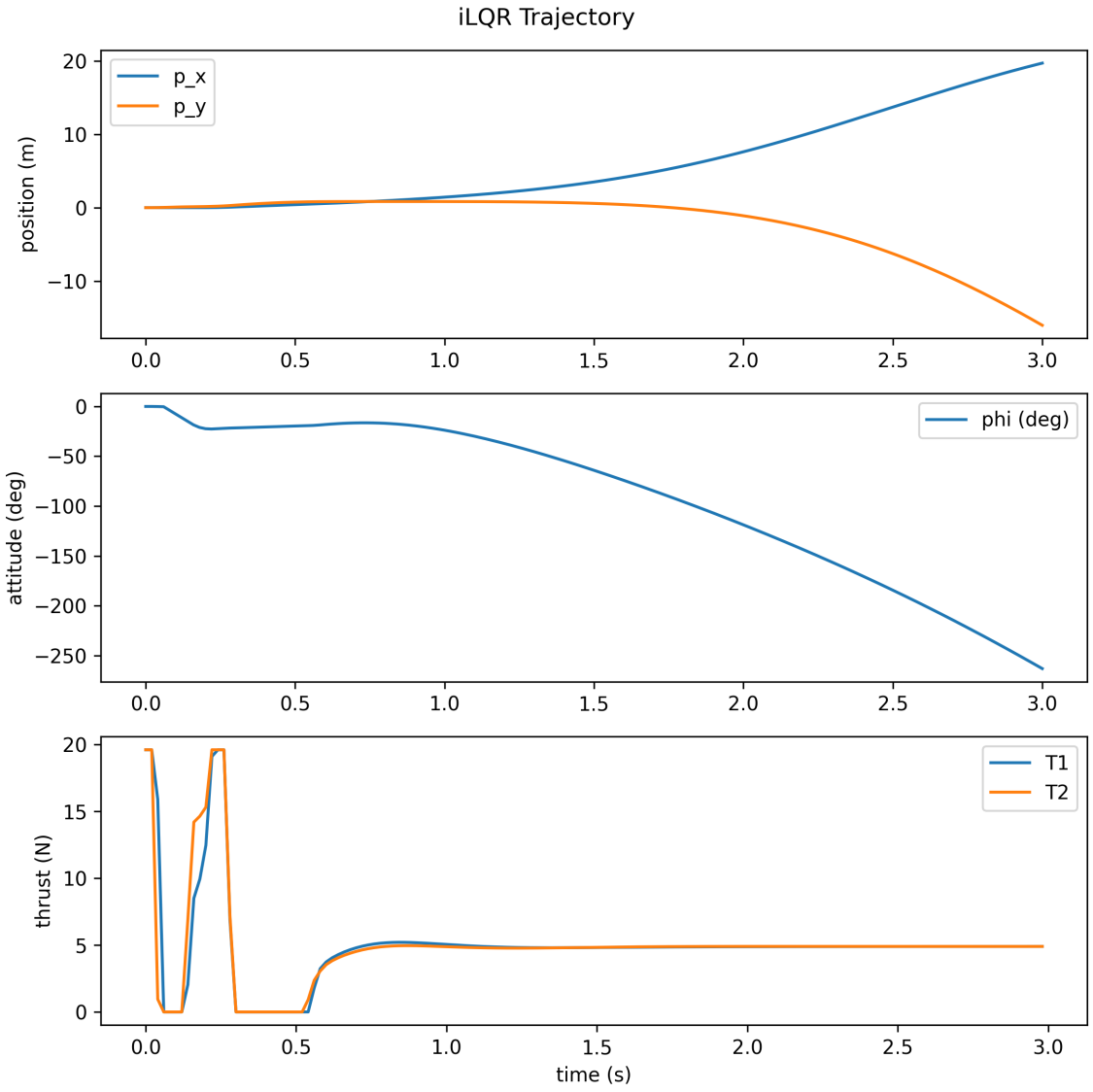

Figure 7: iLQR trajectory on real-world dynamics. The roll angle monotonically decreases to $-270°$, and thrusts exhibit violent oscillations (reaching 20 N) in the first 0.5 s before settling to constant values. The trajectory diverges to $(20\,\text{m}, -16\,\text{m})$, indicating catastrophic failure of the open-loop control sequence under model mismatch.

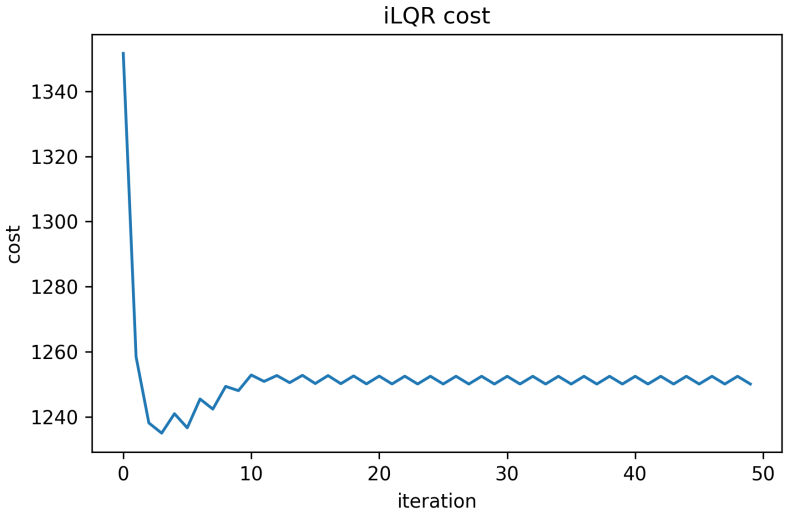

Figure 8: iLQR cost convergence during the planning phase. The algorithm converges to a locally optimal solution within $\sim$10 iterations under the nominal model.

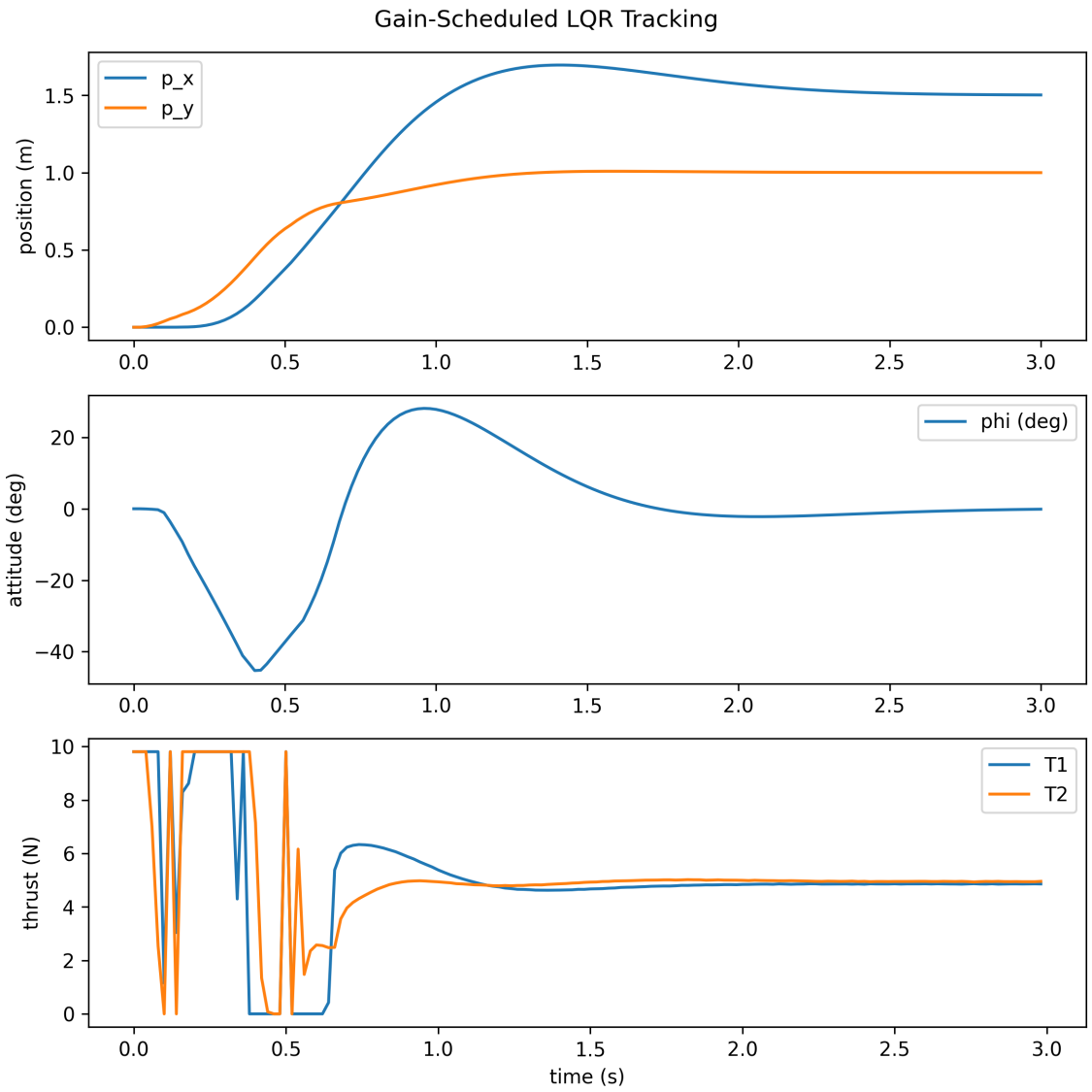

Figure 9: GS-LQR trajectory on real-world dynamics. Roll angle shows initial transients (dip to $-45°$, peak at $+30°$) before stabilizing near $0°$. Thrusts exhibit initial spikes up to 10 N with one rotor briefly near 0 N, then recover to hover values by $t = 1$ s. The trajectory successfully reaches $(1.5\,\text{m}, 1.0\,\text{m})$, demonstrating stable closed-loop tracking despite model errors.

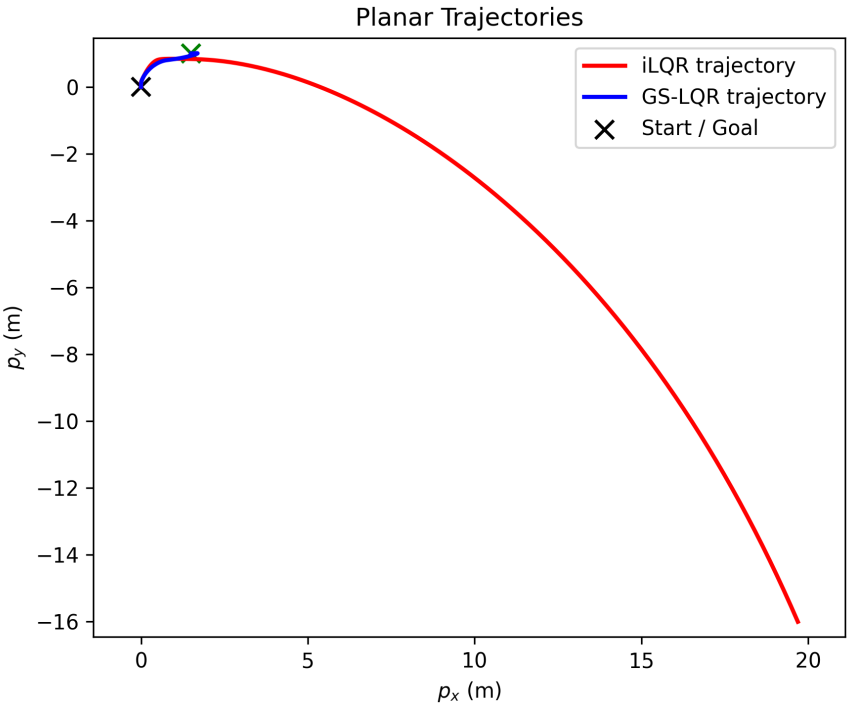

Figure 10: Planar trajectory comparison. GS-LQR (blue) successfully reaches the goal at $(1.5, 1.0)\,\mathrm{m}$, while iLQR (red) diverges catastrophically to $(20, -16)\,\mathrm{m}$ due to accumulated model errors in open-loop execution.

A.7 1d.2 Increased Control Weight Results ($r_T = 0.01$)

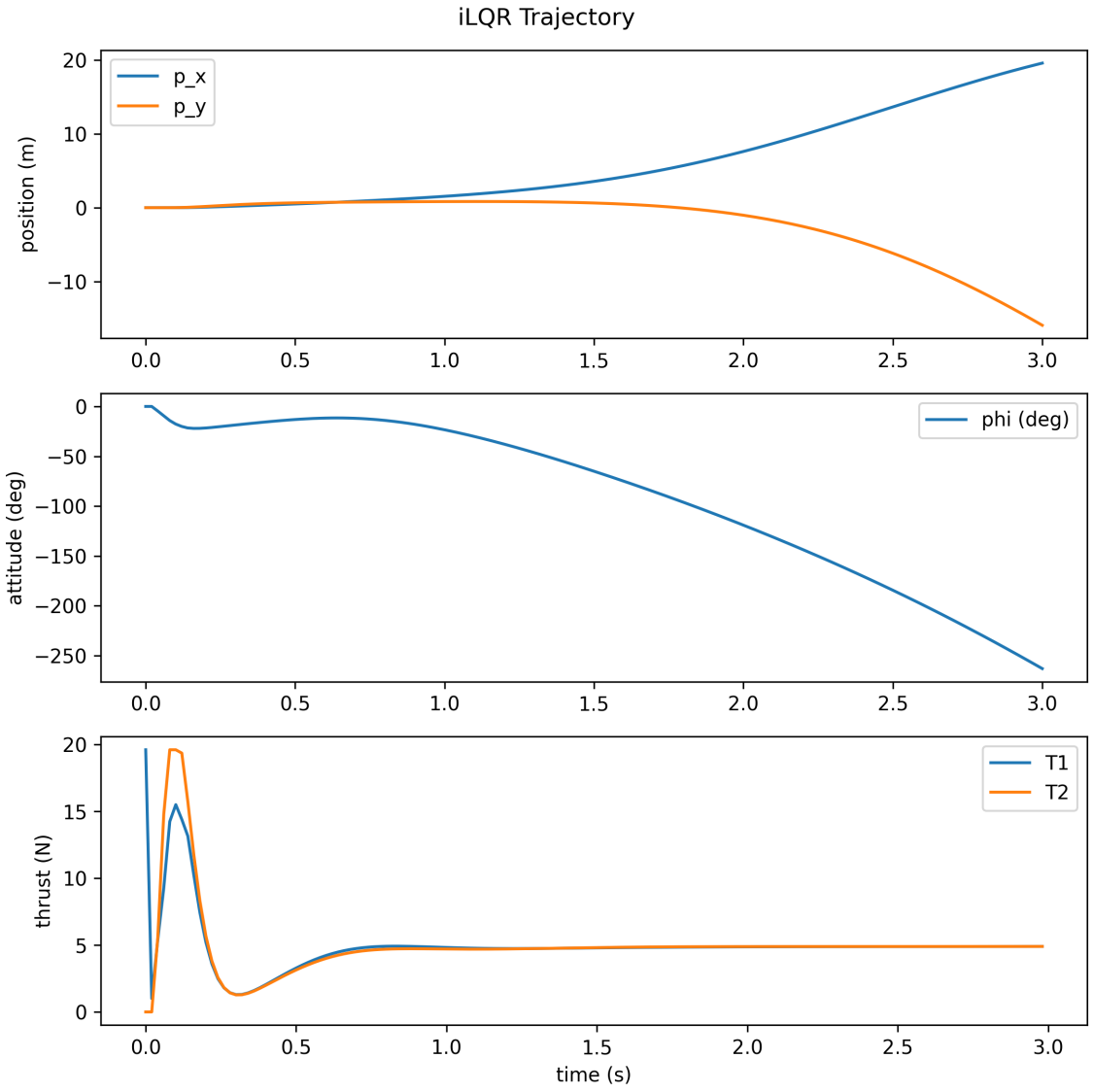

Figure 11: iLQR trajectory with $r_T = 0.01$. The trajectory is nearly identical to the baseline case, diverging to $(20\,\text{m}, -16\,\text{m})$ with roll angle reaching $-270°$. The thrust transients differ slightly (T2 spikes to 20 N instead of T1), but the catastrophic failure persists, demonstrating that control penalty tuning cannot compensate for lack of feedback under model mismatch.

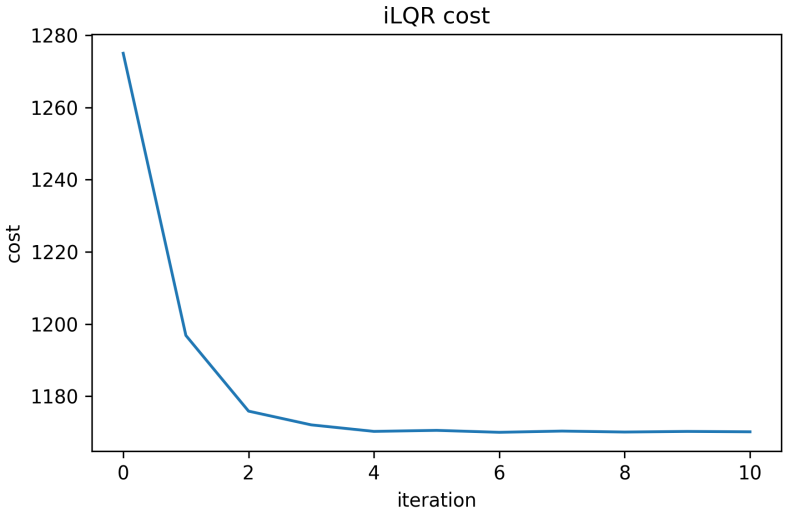

Figure 12: iLQR cost convergence with increased control penalty. The final cost is higher due to the increased weight on control effort.

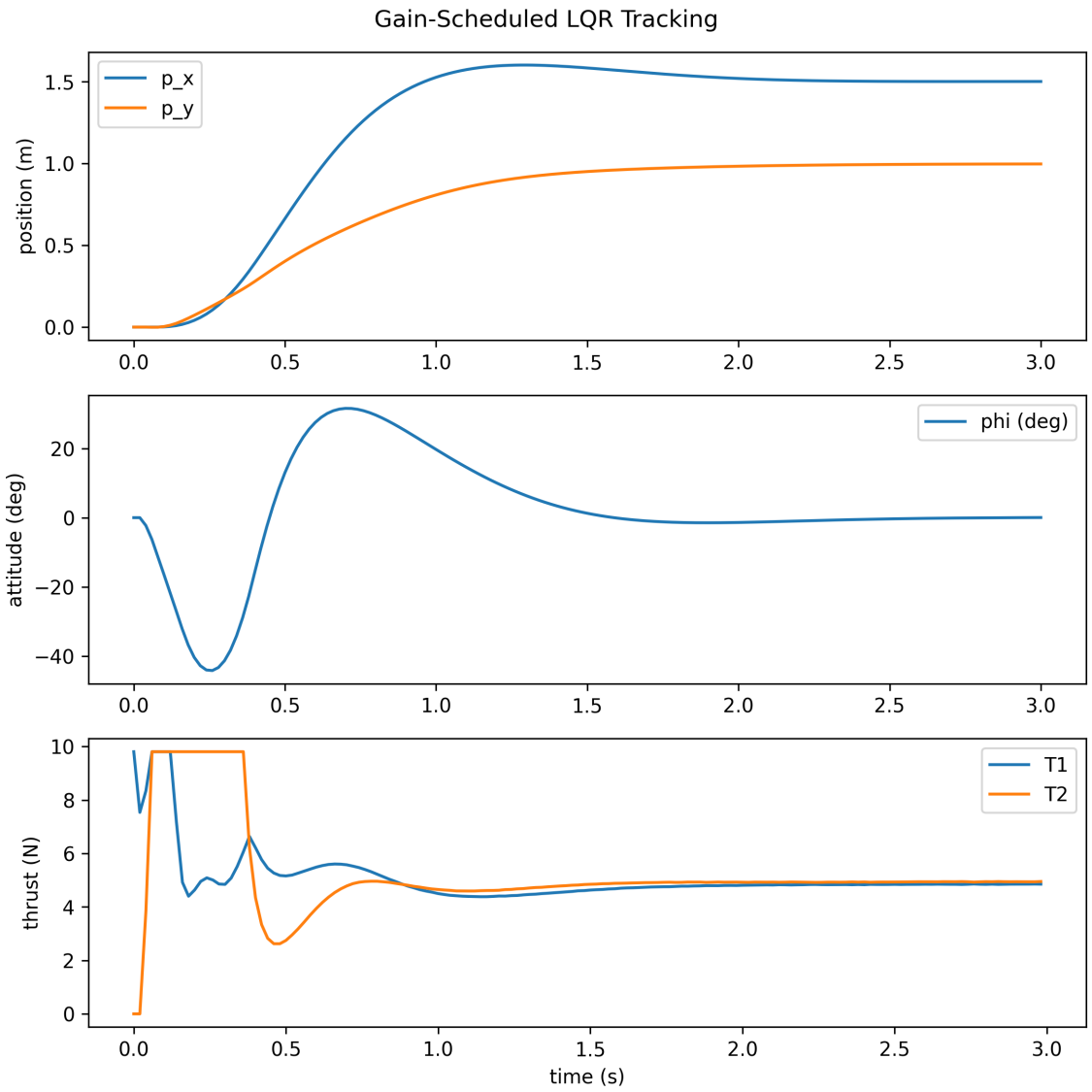

Figure 13: GS-LQR trajectory with $r_T = 0.01$. The trajectory is nearly indistinguishable from the baseline case, successfully reaching $(1.5\,\text{m}, 1.0\,\text{m})$ with similar transient behavior (dip to $-40°$, peak at $+30°$) and thrust profiles. The minimal change demonstrates that feedback control naturally operates efficiently, making the system insensitive to control penalty tuning.

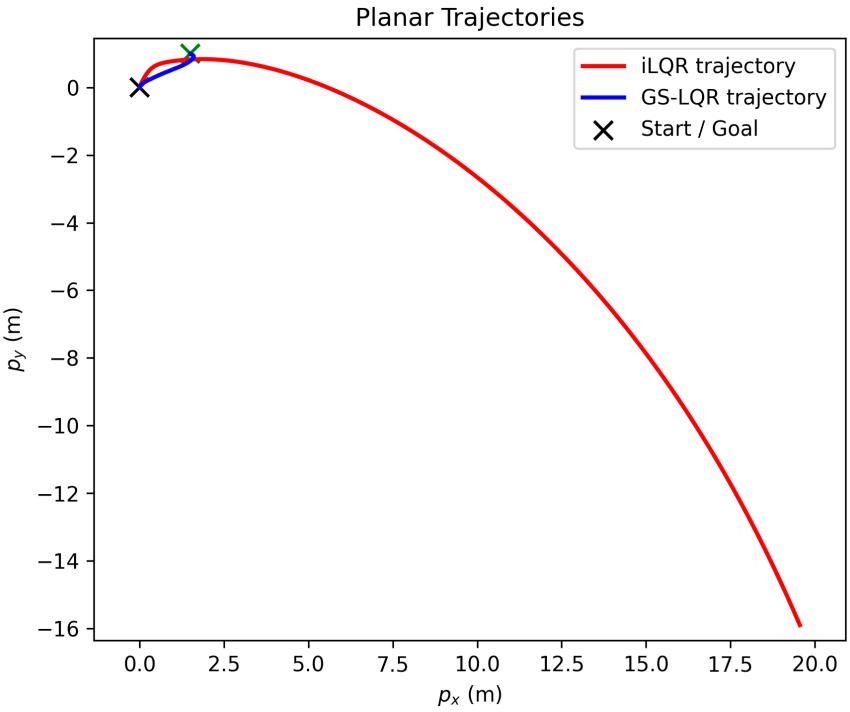

Figure 14: Planar trajectory comparison with increased control weight ($r_T = 0.01$). The trajectories are visually identical to the baseline case: iLQR (red) diverges to $(20\,\text{m}, -16\,\text{m})$ while GS-LQR (blue) reaches the goal at $(1.5\,\text{m}, 1.0\,\text{m})$. The insensitivity to the tenfold increase in control penalty confirms that feedback robustness, not parameter tuning, determines performance under model mismatch.

B Problem 3 Details

B.1 3a. RANSAC Line Implementation

The RANSAC estimator samples point pairs to hypothesize $y = mx + b$, scores them with the $\epsilon = 0.08$ inlier threshold, and refines the winning model with a least-squares fit over the inliers.

```
def ransac_line(points, eps=0.08, n_iter=150):
    """
    Robust line fitting y = m x + b using RANSAC.

    Args:
        points: (N,2) array of [x_i, y_i].
        eps:    inlier distance threshold (default 0.08).
        n_iter: number of RANSAC iterations.

    Returns:
        best_m, best_b, inliers_mask
    """
    N = points.shape[0]
    best_m, best_b = 0.0, 0.0
```

```
    best_inliers = np.zeros(N, dtype=bool)
    rng = np.random.default_rng(SEED_LINE)

    for k in range(n_iter):
        # —— Sample two distinct points ——
        idx = rng.choice(N, size=2, replace=False)
        (x_a, y_a), (x_b, y_b) = points[idx]

        # —— Fit candidate line y = m x + b ——
        m = (y_b - y_a) / (x_b - x_a)
        b = y_a - m * x_a

        # —— Compute distances and find inliers ——
        denom = np.hypot(m, 1.0)
        distances = np.abs(m * points[:, 0] - points[:, 1] + b) / denom
        inliers = distances <= eps
        if np.sum(inliers) > np.sum(best_inliers):
            best_inliers = inliers
            best_m, best_b = m, b

    # —— Refit line using least squares on all inliers ——
    if np.sum(best_inliers) >= 2:
        A = np.column_stack([points[best_inliers, 0], np.ones(np.sum(best_inliers))])
        y = points[best_inliers, 1]
        best_m, best_b = np.linalg.lstsq(A, y, rcond=None)[0]
    else:
        A = np.column_stack([points[:, 0], np.ones(N)])
        y = points[:, 1]
        best_m, best_b = np.linalg.lstsq(A, y, rcond=None)[0]
        best_inliers = np.ones(N, dtype=bool)

    return best_m, best_b, best_inliers
```

B.2 3b. Umeyama Alignment Implementation

The Umeyama routine computes centroids, factors the cross-covariance with SVD, enforces a proper rotation, and recovers the translation before returning (R, t).

```
def umeyama_alignment(P, Q):
    """
    Closed-form least-squares rigid alignment (Umeyama method).
    Args:
        P: (N,3) source points
        Q: (N,3) target points
    Returns:
        R (3,3), t (3,)
    """
```

```
    ### YOUR CODE HERE: compute centroids
    p_bar = P.mean(axis=0)
    q_bar = Q.mean(axis=0)

    ### YOUR CODE HERE: center the data
    P_centered = P - p_bar
    Q_centered = Q - q_bar

    ### YOUR CODE HERE: covariance and SVD
    H = P_centered.T @ Q_centered
    U, S, Vt = np.linalg.svd(H)

    ### YOUR CODE HERE: compute rotation and translation
    V = Vt.T
    D = np.eye(3)
    D[-1, -1] = np.linalg.det(V @ U.T)
    R = V @ D @ U.T
    t = q_bar - R @ p_bar

    return R, t
```

B.3 3c. RANSAC Registration Implementation

The RANSAC wrapper samples minimal triples, filters degenerate sets, evaluates consensus with the Umeyama inner solver, and refits on the best inliers.

```
def ransac_umeyama(P, Q, correspondences, tau=0.02, n_iter=200):
    """
    Robust Umeyama alignment using RANSAC.

    Args:
        P, Q: source and target point clouds (N,3)
        correspondences: (N,2) index pairs
        tau: inlier threshold
        n_iter: number of RANSAC iterations
    Returns:
        best_R, best_t, best_inliers
    """

    N = correspondences.shape[0]
    best_R = np.eye(3)
    best_t = np.zeros(3)
    best_inliers = np.zeros(N, dtype=bool)
    best_rmse = np.inf
    rng = np.random.default_rng(SEED_REG)

    P_corr = P[correspondences[:, 0]]
```

```
Q_corr = Q[correspondences[:, 1]]

for _ in range(n_iter):
    # === TODO(c1): randomly sample 3 correspondence pairs ===
    for _ in range(100):
        idx = rng.choice(N, size=SAMPLE_SIZE, replace=False)
        Ps = P_corr[idx]
        Qs = Q_corr[idx]
        area = np.linalg.norm(np.cross(Ps[1] - Ps[0], Ps[2] - Ps[0]))
        if area > 1e-6:
            break
    else:
        continue

    # === TODO(c2): estimate transform using Umeyama ===
    R_k, t_k = umeyama_alignment(Ps, Qs)

    # === TODO(c3): compute transformed distances for all correspondences ===
    P_all = (P_corr @ R_k.T) + t_k
    errors = np.linalg.norm(P_all - Q_corr, axis=1)
    inliers = errors < tau
    inlier_count = int(inliers.sum())
    if inlier_count > 0:
        rmse_k = np.sqrt(np.mean(errors[inliers]**2))
    else:
        rmse_k = np.inf

    best_count = int(best_inliers.sum())
    if (inlier_count > best_count) or (inlier_count == best_count and rmse_k < be
        best_inliers = inliers
        best_R, best_t = R_k, t_k
        best_rmse = rmse_k

# === TODO(c4): refit final (R, t) using inlier subset ===
if best_inliers.any():
    P_best = P_corr[best_inliers]
    Q_best = Q_corr[best_inliers]
    best_R, best_t = umeyama_alignment(P_best, Q_best)
else:
    best_R, best_t = umeyama_alignment(P_corr, Q_corr)

return best_R, best_t, best_inliers
```

C Problem 4 Details

C.1 4d. Complete `is_free_state` Implementation

The complete `is_free_state` method from the `MidtermRRT` class in `P4_rrt.py` (Problem 4d) implements the collision detection logic derived in parts (a)–(c). The method extracts the translation vector τ and angle θ from the pose, constructs the rotation matrix R (lines 11–14), defines the body-frame half-space matrices A and b (lines 16–32), transforms them to the world frame as $A_W = AR^\top$ and $b_W = b - A_W\tau$ (lines 34–36), then evaluates $A_W q + b_W \leq 0$ for each obstacle point (lines 38–41), short-circuiting once a collision is detected.

```
def is_free_state(self, obstacle_cloud, x, bx, by):
    # Returns True if the robot in pose "x" is not in collision with the obstacle

    # State information
    tau = np.reshape(x[:2],(2,1)) #displacement
    theta = x[2] #angle

    ########## Code starts here ##########
    # Hint: If one of the points in the point cloud is contained in the drone body

    # Rotation matrix
    c, s = np.cos(theta), np.sin(theta)
    R = np.array([[c, -s],
                  [s,  c]])

    # Matrix A defining the planes of the rectangle boundaries
    A = np.array([
        [1.0, 0.0],
        [-1.0, 0.0],
        [0.0, 1.0],
        [0.0, -1.0],
    ])

    # b defining distance to the planes
    half_width = 0.5 * bx
    half_height = 0.5 * by
    b = np.array([
        -half_width,
        -half_width,
        -half_height,
        -half_height,
    ])

    R_world_to_body = R.T
    A_world = A @ R_world_to_body
    b_world = b.reshape(-1, 1) - A_world @ tau
```

```
        for point in obstacle_cloud:
            q_world = point.reshape(2, 1)
            if np.all(A_world @ q_world + b_world <= 0):
                return False

        return True

        ########## Code ends here ##########
```

C.2 4b. Frame Transform Computation

The excerpt highlighted here shows the translation and rotation definitions together with the world-to-body transform $q_b = R^\top(q - \tau)$ used in the collision test.

```
        tau = np.reshape(x[:2],(2,1)) #displacement
        theta = x[2] #angle

        ########## Code starts here ##########
        # Hint: If one of the points in the point cloud is contained in the drone bod

        # Rotation matrix
        c, s = np.cos(theta), np.sin(theta)
        R = np.array([[c, -s],
                      [s,  c]])
```

C.3 4c. World-Frame Inequality

This segment pre-multiplies the rotation and translation to form A_W and b_W, enabling direct evaluation of the inequality $A_W q + b_W \leq 0$ in world coordinates.

```
        # Matrix A defining the planes of the rectangle boundaries
        A = np.array([
            [1.0, 0.0],
            [-1.0, 0.0],
            [0.0, 1.0],
            [0.0, -1.0],
        ])

        # b defining distance to the planes
        half_width = 0.5 * bx
        half_height = 0.5 * by
        b = np.array([
            -half_width,
            -half_width,
            -half_height,
            -half_height,
```

```
])

R_world_to_body = R.T
A_world = A @ R_world_to_body
b_world = b.reshape(-1, 1) - A_world @ tau
```