AA274A: Principles of Robot Autonomy I
Midterm 2025

Instructions

You have **5 hours** to complete this midterm from the time you begin or until the end of the exam window (**November 1 at Noon 12:00 PM PT**). The exam is open notes, but **collaboration with other students is strictly prohibited**. You may access official LaTeXand Python documentation (e.g., function syntax), but all other internet resources, including LLMs like ChatGPT and IDE-integrated AI tools, are strictly prohibited.

Short answer questions should be formatted using LaTeX. Questions requiring extensive equations or math can be handwritten and included as an image, but we highly recommend typesetting in LaTeXfor guaranteed legibility. For multiple choice questions, no explanations or intermediate work is required, only your answer choice. Feel free to use this LaTeXOverleaf template.

The exam is worth a total of **100 points**. There are **4 mandatory questions** each worth **25 points**.

Questions about the exam can be posted as **private posts on Ed**. However, we will not respond to content based questions, or to any questions that would give the questioner an advantage on the exam. Additionally, "my question was not answered on Ed" is not a valid excuse for submitting the exam late. Do not email the course staff mailing list with midterm questions.

Problem 1: Trajectory Optimization vs Trajectory Following (25 pts)

Background. In lecture we saw several algorithms for trajectory optimization and trajectory following. In this problem you will compare iterative LQR (iLQR) for trajectory optimization and gain-scheduled LQR (GS-LQR) for trajectory following. While these two algorithms have many similarities, they actually achieve quite different objectives: iLQR optimizes an open-loop dynamically feasible trajectory, while GS-LQR is a closed-loop controller to follow a given dynamically feasible trajectory. You will implement and compare these algorithms for a planar quadrotor robot model in a Colab Notebook. **Remember that you will need to make a copy of the notebook to save your work!**

The goal is to move the quadrotor from the origin to a hover point at $(p_x, p_y) = (1.5, 1.0)$ m.

Parameters: $m = 1.0$ kg, $g = 9.81$ m/s^2, $\ell = 0.25$ m, $I_{zz} = 0.025$ kg m^2, $\Delta t = 0.02$ s, and horizon $T = 150$ (3 s total). Each rotor thrust satisfies $0 \le T_i \le 2mg$. The nominal hover thrust is $T_{\text{hov}} = \frac{mg}{2}$.

(a) Planar Quadrotor Dynamics(5 pts)

To design either a trajectory optimizer or a tracking controller, we first need a mathematical model of the quadrotor's motion.

The planar quadrotor has state

$$x = \begin{bmatrix} p_x & v_x & p_y & v_y & \phi & \omega \end{bmatrix}^\top, \qquad u = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix},$$

The continuous-time equations of motion are:

$$\dot{x} = f_c(x, u) = \begin{bmatrix} v_x \\ -\frac{T_1+T_2}{m}\sin\phi \\ v_y \\ \frac{T_1+T_2}{m}\cos\phi - g \\ \omega \\ \frac{(T_2-T_1)\ell}{I_{zz}} \end{bmatrix}.$$

We will discretize these using forward Euler:

$$x_{t+1} = f_d(x_t, u_t) = x_t + \Delta t\, f_c(x_t, u_t),$$

with $\Delta t = 0.02$ s.

Implement both `f_continuous(x,u)` and `f_discrete(x,u,dt)` in the Colab notebook.

(b) The Optimal Control Problem and Running Cost (5 pts)

Both iLQR and GS-LQR minimize a cumulative quadratic cost over time. We first express the general form, then tailor it to the planar-quadrotor task.

The general finite-horizon quadratic objective is

$$J = \sum_{t=0}^{T-1} \Big[(x_t - x^\star)^\top Q (x_t - x^\star) + (u_t - u^\star)^\top R (u_t - u^\star) \Big] + (x_T - x^\star)^\top Q_f (x_T - x^\star),$$

where $x^\star, u^\star$ are the desired equilibrium values at the hover point.

For this problem,

$$x^\star = [1.5,\, 0,\, 1.0,\, 0,\, 0,\, 0]^\top, \qquad u^\star = [T_{\text{hov}},\, T_{\text{hov}}]^\top, \quad T_{\text{hov}} = \tfrac{mg}{2}.$$

To get our cost function we need to:

(i) Substitute the planar-quadrotor state and control into the general objective above and **write out explicitly the per-timestep running cost** in scalar form. Use diagonal weights

$$Q = \text{diag}(q_p,\, q_v,\, q_p,\, q_v,\, q_\phi,\, q_\omega), \qquad R = r_T I_2.$$

Record the final simplified per-timestep running cost and implement it in the Colab notebook.

(c) Linearization about a Reference Trajectory (5 pts)

Both the iLQR and gain-scheduled LQR controllers rely on a local *affine linearization* of the nonlinear dynamics. In iLQR, this linearization is taken about the current nominal trajectory from the previous iteration. In GS-LQR, it is taken about a provided reference trajectory (e.g., a minimum-jerk path). In both cases, the linearization allows us to approximate the discrete-time dynamics by a first-order model valid for small perturbations.

The discrete dynamics are obtained using forward Euler:

$$x_{t+1} = f_d(x_t, u_t) = x_t + \Delta t\, f_c(x_t, u_t).$$

We can approximate these dynamics locally around a point $(x_t^{\text{ref}}, u_t^{\text{ref}})$ by:

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

where

$$A_t = \left.\frac{\partial f_d}{\partial x}\right|_{(x_t^{\text{ref}}, u_t^{\text{ref}})} = I + \Delta t\, A_c(x_t^{\text{ref}}), \qquad B_t = \left.\frac{\partial f_d}{\partial u}\right|_{(x_t^{\text{ref}}, u_t^{\text{ref}})} = \Delta t\, B_c(x_t^{\text{ref}}, u_t^{\text{ref}}).$$

In deviation form:

$$\delta x_{t+1} = A_t\, \delta x_t + B_t\, \delta u_t, \qquad \text{with } \delta x_t = x_t - x_t^{\text{ref}},\ \ \delta u_t = u_t - u_t^{\text{ref}}.$$

(i) First you will need to write expressions for the continuous-time Jacobians $A_c(x), B_c(x, u)$, then write explicit analytic expressions for the discrete Jacobians A_t, B_t at a general point $(x_t^{\text{ref}}, u_t^{\text{ref}})$. **Include the derivation for these expressions in your solutions and fill in the continuous and discrete Jacobian functions in the Colab notebook.**

(ii) Show analytically how to compute the linearization in affine form $x_{t+1} \approx f_d(x_t^{\text{ref}}, u_t^{\text{ref}}) + A_t(x_t - x_t^{\text{ref}}) + B_t(u_t - u_t^{\text{ref}})$.

(iii) Implement the provided Python function:

`linearize_about(x_ref, u_ref)`
which returns $A_t, B_t, x_{t+1}^{\text{ref}}$ following the equations above.

This function will be used by both controllers:

- **iLQR:** linearizes about the previous iteration as a reference trajectory,
- **GS-LQR:** linearizes about the given reference trajectory.

(d) Interpreting Results: iLQR vs. GS-LQR (10 pts)

After completing your linearization function, run the iLQR and GS-LQR blocks of the Colab notebook. To compare the methods, you will evaluate them on a "real-world" version of the planar quadrotor whose dynamics differ slightly from the model used for planning.

Generated plots. Each block will produce plots of:

- planar path $(p_x(t), p_y(t))$,
- roll angle $\phi(t)$,
- thrust inputs $T_1(t), T_2(t)$,
- cost vs. iteration (for iLQR).

To compare these methods use the generated plots (mentioned above) to provide short answers (1-3 sentences) to each of the following:

(i) **Trajectory shape.** Compare the trajectories produced by iLQR and GS-LQR.

(ii) **Cost and effort.** Compare the total closed-loop rollout cost J and cumulative control effort.

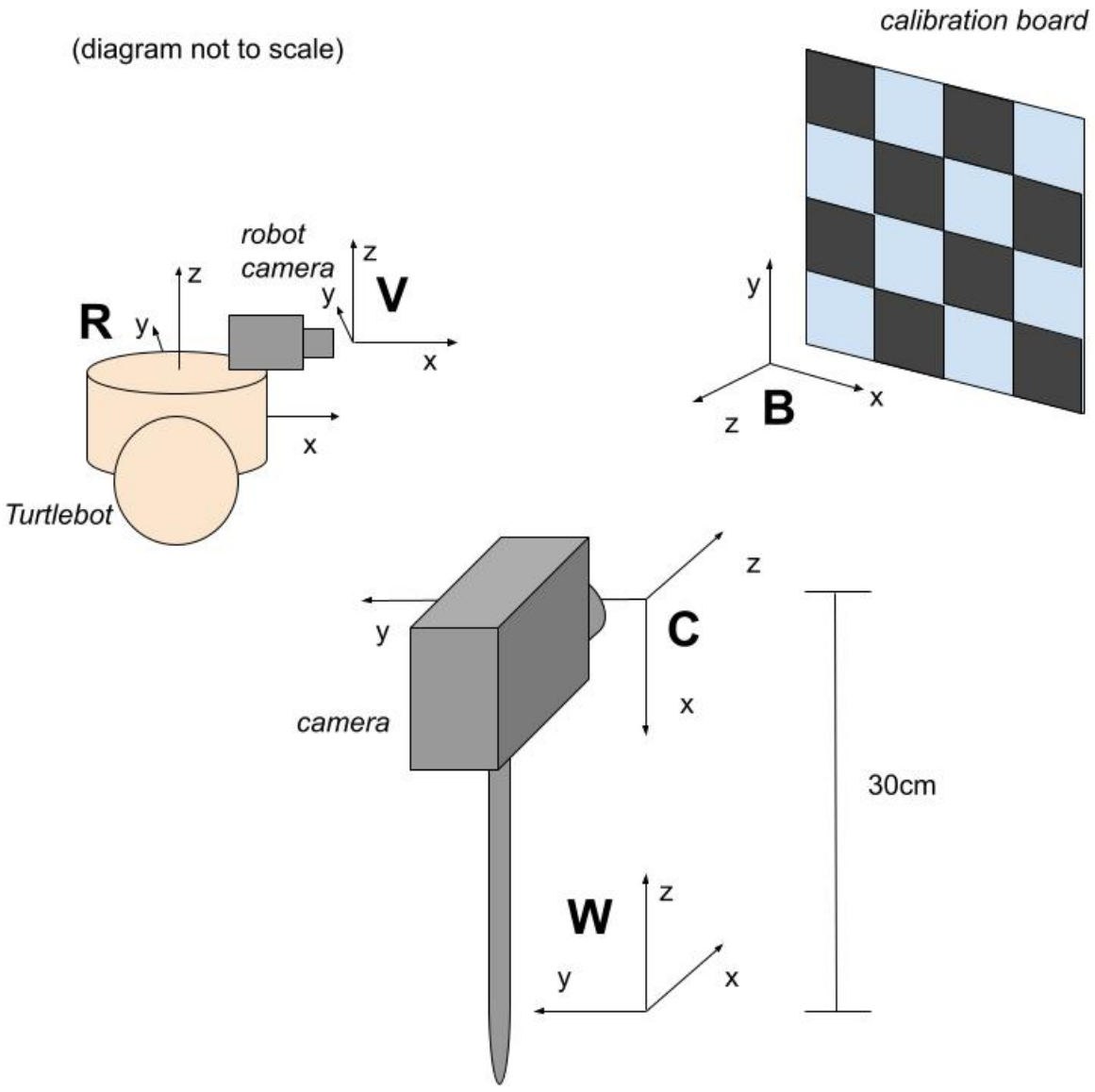
(iii) **Sensitivity to weights.** Rerun both controllers after increasing r_T by a factor of 10. Describe qualitatively what changes in tilt, thrust, and trajectory smoothness you observe.

Problem 2: Coordinate Transforms (25 pts)

Let's consider a scenario where we have a Turtlebot moving around an enclosed space and whose movements we are tracking. The robot's regular navigation equipment has failed, and so we're currently trying to work out its current position based on the view of its camera.

Separately from the robot, we have a camera with a known position fixed in the environment, and both it and the robot are currently looking at a calibration board also fixed in the environment. For the time being, the robot isn't moving, so we'll only consider the positions of the objects in the environment and not their velocities. To describe the positions of objects in 3D space in this environment, we have five coordinate frames:

- W, the world frame, which is fixed to the floor.
- C, the camera frame, which is fixed to the camera.
- B, the board frame, which is fixed to the calibration board.
- V, the view frame, fixed to the camera mounted on the Turtlebot.
- R, the robot frame, fixed to the chassis of the Turtlebot.

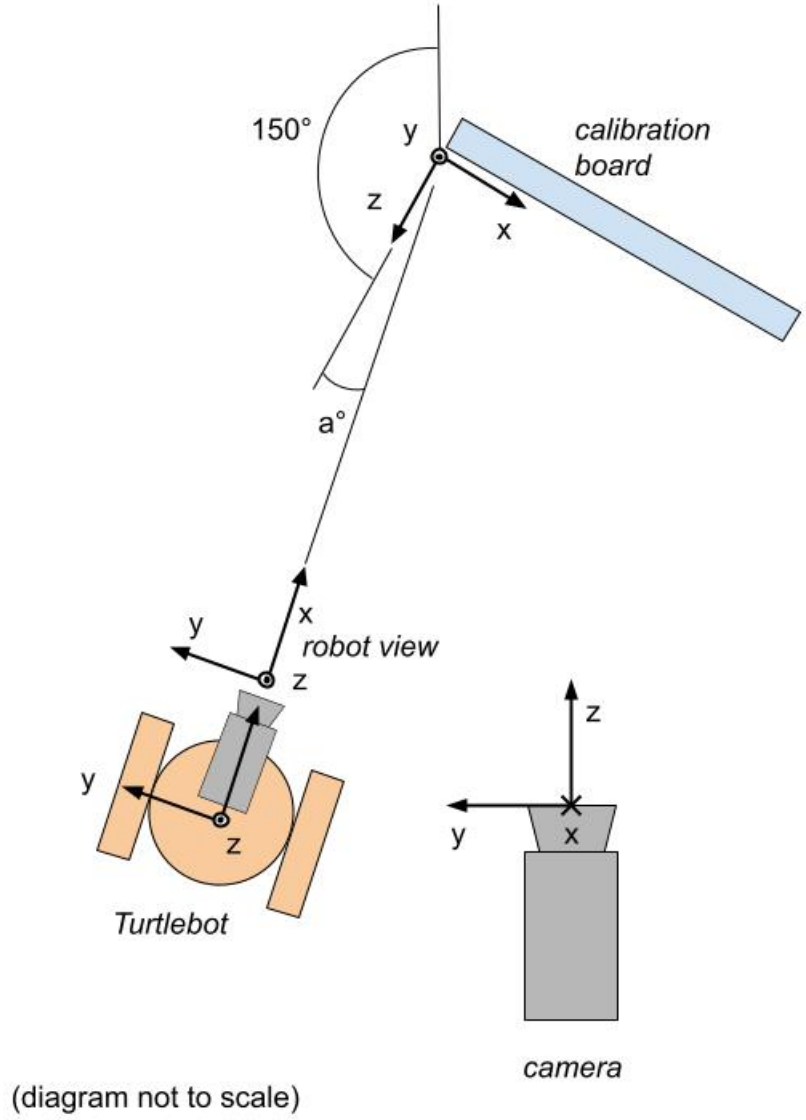


(a) Position of the calibration board in the world frame (3 pts)

Let's start by finding the position of the calibration board in the world frame, using the camera's view of the board.

- The origin of W is fixed to the ground, with its z vector pointing upwards and the x and y vectors set across the ground according to the right-hand rule (i.e. cross-multiplying the x and y vectors results in a vector in the direction of z).
- The origin of C is placed $30cm$ directly above the origin of W, with the camera's z vector pointing directly out from the lens, the x vector pointing directly downwards and the y vector pointing to the camera's left; the z vector of C is parallel to the x vector of W.
- The origin of B is at the same height as the camera (so a distance of $0cm$ in the camera's x axis), at a distance of $20cm$ in the camera's y axis and at a distance of $100cm$ in the camera's z axis.

Find the position of the lower left corner of the calibration board (that is, the origin of its coordinate frame) in the world frame. Write your answer as a vector.



(b) Conversion between board frame and world frame (7 pts)

The calibration board surface is facing in a different direction than the camera lens:

- The y axis of B is pointing in the opposite direction from the x axis of C.
- The direction of the z vector of B is rotated 150° about the y axis of B relative to the direction of the z vector of C.

Write the homogeneous transformation matrix to change the coordinates of a position expressed in the coordinate frame B to an expression in the coordinate frame W.

(c) Finding the position of the robot (10 pts)

Although we don't know the position of the robot in the coordinate frame W, which is what we ultimately want to find, we know the position of V relative to R and can infer the position and orientation of the calibration board relative to V from the robot's view of it.

- The robot frame R and the view frame V both have the x direction facing the front of the robot and the z vector facing upwards.
- The origin of frame V is $10cm$ from the origin of R in its x direction, $0cm$ from the origin of R in the y direction and $4cm$ above the origin of R in the z direction.
- The z vector of V, aiming directly upwards, is parallel to the y axis of B, and the x vector of V is rotated around these vectors such that the x vector of V is rotated $(180 + a)^\circ$ from the z vector of B, where a is the last digit of your SUID number.
- We've lined up the robot so that the origin of the frame B is $70cm$ from the origin of V in the x direction of V, $22.6cm$ from the origin of V in the z direction of V and $0cm$ from the origin of V in its y direction.

Find the position of the robot, defined by the origin of the coordinate frame R, in the coordinate frame W. State the value of a in your case along with your answer.

(d) Axis of wheel rotation (2.5 pts)

Multiple Choice: If we are working with a differential drive robot (such as a Turtlebot) with its position expressed as the midpoint of its two drive wheels, and we wanted to express the rotation of these wheels as angular velocities in 3D space, **select the vector we can use to express the axis of rotation for both wheels**.

1. Vector y in coordinate frame R
2. Vector y in coordinate frame V
3. Vector x in coordinate frame R
4. Vector z in coordinate frame W

(e) Axis of robot movement (2.5 pts)

Multiple Choice: If we are working with a differential drive robot (such as a Turtlebot) with is position expressed as the midpoint of its two drive wheels, and we wanted to express the instantaneous velocity of the chassis in 3D space, **select the vector we can use to express the direction of displacement for the robot**.

1. Vector x in coordinate frame B
2. Vector x in coordinate frame R
3. Vector z in coordinate frame R
4. Vector z in coordinate frame V

Problem 3: Robust Estimation and Point-Cloud Alignment (25 pts)

Modern robot perception often requires fitting simple geometric models to noisy data and aligning 3D point clouds captured from different viewpoints. In this problem you will explore both ideas. You will first use the **RANSAC** algorithm to robustly fit a line to noisy 2D data, and then apply the **Umeyama** method, a closed-form, SVD-based algorithm, to estimate a rigid transformation between two 3D point sets. We will be implementing these methods in a Colab Notebook. **Remember that you will need to make a copy of the notebook to save your work!**

(a) RANSAC for robust 2D line fitting (5 pts)

We begin with a simple model fitting task. You are given a set of $N = 100$ 2D points $\{(x_i, y_i)\}$ saved in the file `line_2d_points.npy`. Most points lie near a true line $y = mx + b$ but about 10 % are outliers. The goal is to estimate (m, b) robustly.

RANSAC repeatedly samples a minimal subset, estimates a candidate model, and tests how many points agree with it (the inliers). After many iterations, the model with the largest inlier set is selected and refit to all inliers.

Use the following fixed parameters throughout this part:

$$\varepsilon = 0.08 \text{ (inlier threshold)}, \qquad N_{\text{iter}} = 150.$$

The perpendicular distance from a point (x_i, y_i) to a line $y = mx + b$ is

$$d_i = \frac{|mx_i - y_i + b|}{\sqrt{m^2 + 1}}.$$

Algorithm. The RANSAC loop for this task is summarized below.

Algorithm 1 RANSAC line fitting (fixed parameters)

Require: points $\{(x_i, y_i)\}_{i=1}^{N}$, threshold $\varepsilon = 0.08$, iterations $N_{\text{iter}} = 150$
1: Initialize best_inliers $\leftarrow \emptyset$, best_model $\leftarrow (0, 0)$
2: **for** $k = 1$ to N_{iter} **do**
3: Randomly choose two distinct points $(x_a, y_a), (x_b, y_b)$
4: Estimate candidate model:
$$m = \frac{y_b - y_a}{x_b - x_a}, \qquad b = y_a - mx_a$$
5: Compute distances d_i for all points and mark inliers where $d_i < \varepsilon$
6: **if** inlier count $>$ |best_inliers| **then**
7: Update best_inliers and best_model
8: **end if**
9: **end for**
10: Refit $(\hat{m}, \hat{b})$ using least squares on all best_inliers
11: **return** final $(\hat{m}, \hat{b})$

Implement this algorithm in `ransac_line(points)` using the parameters above. Then plot the results: display all points, highlight inliers (e.g., in green), and overlay the best-fit line in red. Report in your solution: the estimated $(\hat{m}, \hat{b})$ values, and the number of inliers.

(b) 3D point-cloud alignment with the Umeyama method (10 pts)

Next, we align two 3D point sets $\mathcal{P} = \{p_i\}$ and $\mathcal{Q} = \{q_i\}$ related by a rigid transform

$$q_i \approx R\,p_i + t.$$

Here $R \in SO(3)$ is a rotation matrix and $t \in \mathbb{R}^3$ is a translation vector. Our goal is to find (R, t) that minimize

$$\frac{1}{N}\sum_i \|q_i - (Rp_i + t)\|_2^2.$$

You are provided two clouds:

- `bunny_vertices.npy`: the original Stanford bunny (source),
- `bunny_target.npy`: the same points after a random rotation, translation, and small Gaussian noise.

We assume the correspondences are known and one-to-one.

The **Umeyama method** gives a closed-form least-squares solution using the **singular value decomposition (SVD)** of the covariance matrix between the two centered point sets.

Here is the complete algorithm you should implement.

Algorithm 2 Umeyama rigid alignment (no scale)

Require: Source points $P = \{p_i\}$, target points $Q = \{q_i\}$

1: Compute centroids:
$$\bar{p} = \frac{1}{N}\sum_i p_i, \qquad \bar{q} = \frac{1}{N}\sum_i q_i$$

2: Center the data:
$$P' = [p_i - \bar{p}], \qquad Q' = [q_i - \bar{q}]$$

3: Compute cross-covariance:
$$H = (P')^\top Q'$$

4: Perform SVD: $H = U\,\Sigma\,V^\top$

5: Compute rotation:
$$R = V\,\mathrm{diag}(1, 1, \det(VU^\top))\,U^\top$$

6: Compute translation:
$$t = \bar{q} - R\,\bar{p}$$

7: **return** (R, t)

This method works because SVD provides orthogonal matrices U, V that maximize $\mathrm{tr}(RH)$ over all valid rotations $R \in SO(3)$, making it the least-squares optimal rigid alignment.

You will implement this algorithm exactly as written in `umeyama_alignment(P,Q)`. Use `np.linalg.svd` for the decomposition. After computing (R, t), apply the transformation $Rp_i + t$ to the source cloud and compare it visually to the target cloud.

Report the following results in your solutions: the estimated R and t matrices, the root-mean-square alignment error (RMSE), and a 3D scatter plot showing both clouds before and after alignment.

(c) Robust 3D registration via RANSAC (10 pts)

Point-cloud correspondences can contain mismatches. To make the Umeyama alignment robust, we will wrap it inside a RANSAC loop that rejects outliers. In each iteration we hypothesize a transform from a minimal sample of 3 correspondences, score it on all correspondences, and keep the transform with the largest consensus set. Afterward we refit on the inliers.

Use the following fixed parameters:

$$s = 3\ , \quad \tau = 0.002 \quad N_{\text{iter}} = 200, \quad \text{seed} = 274.$$

Assume $P \in \mathbb{R}^{N\times 3}$ and $Q \in \mathbb{R}^{N\times 3}$ hold one-to-one correspondences by row index (q_i corresponds to p_i). Define the residual for a hypothesis (R, t) as

$$r_i \ = \ \|R\,p_i + t - q_i\|_2\,,$$

and call a pair an inlier if $r_i < \tau$.

Here we need to accomplish the following: implement a RANSAC loop that repeatedly samples 3 non-collinear pairs, estimates (R, t) with your Umeyama implementation, counts inliers with the rule above, remembers the best hypothesis, and finally refits $(\hat{R}, \hat{t})$ on only the inliers from that best hypothesis. When two hypotheses tie in inlier count, break the tie by choosing the one with smaller inlier RMSE.

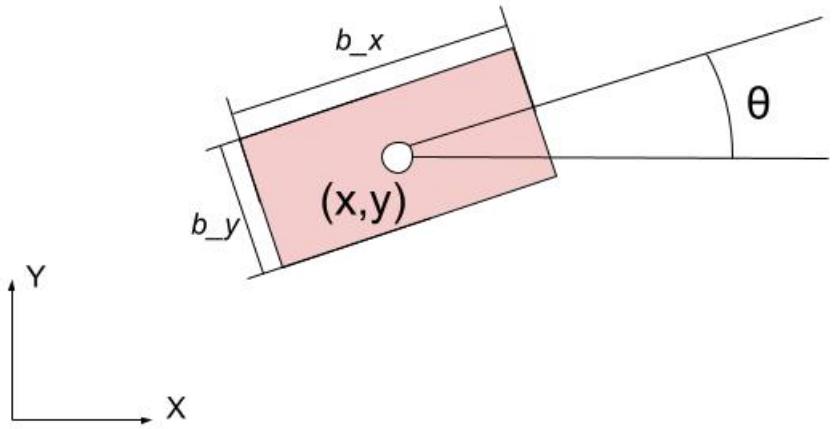
Algorithm 3 RANSAC for rigid registration (Umeyama as inner solver; fixed parameters)

Require: paired points $P, Q \in \mathbb{R}^{N\times 3}$, $\tau = 0.02$, $N_{\text{iter}} = 200$, $s = 3$, RNG seed = 274
Initialize best_inliers $\leftarrow \emptyset$, best$(R, t) \leftarrow$ none
for $k = 1$ to N_{iter} **do**
  Randomly choose $s = 3$ *distinct* indices; let P_s, Q_s be those rows
  If the three p's are nearly collinear, resample (e.g., check that area$(\triangle p) > 10^{-6}$)
  $(R_k, t_k) \leftarrow$ Umeyama(P_s, Q_s) ▷ use your SVD method from part (b)
  For all i, compute $r_i = \|R_k p_i + t_k - q_i\|_2$
  $\mathcal{I} \leftarrow \{i : \ r_i < \tau\}$ ▷ consensus set
  if $|\mathcal{I}| >$ |best_inliers| **or** ($|\mathcal{I}|$ = |best_inliers| and RMSE$_{\mathcal{I}}(R_k, t_k)$ is smaller) **then**
    best_inliers $\leftarrow \mathcal{I}$, best$(R, t) \leftarrow (R_k, t_k)$
  end if
end for
$(\hat{R}, \hat{t}) \leftarrow$ Umeyama$(P_{\text{best_inliers}}, Q_{\text{best_inliers}})$ ▷ refit on inliers
return $(\hat{R}, \hat{t},$ best_inliers$)$

You will implement this algorithm in the provided notebook. After implementing the function, run it on the provided bunny point sets. **In your write-up, report the following results: the number of inliers found, the final refit rotation $\hat{R}$ and translation $\hat{t}$, and the RMSE evaluated on inliers. Include a 3D scatter plot showing (i) the raw source and target, and (ii) the aligned source $(\hat{R}p_i + \hat{t})$ overlaid with the target. Conclude with a short paragraph (1-2 sentences) stating whether the robust version improved alignment compared to using Umeyama on all correspondences directly.**

Problem 4: RRT (25 pts)

In this problem, you will modify the RRT algorithm for a robot that can change its attitude (that is, its orientation in space) and has its volume represented with a bounding box, while using a point cloud to represent obstacles in space. Let's say we are working with a drone moving in 2D space, represented by a rectangle large enough to include all its body and propellers; the drone pose is listed with the vector $(x, y, \theta)^\intercal$ where x is the position of the drone (at the geometric center of the rectangle) in the horizontal axis, y is the position of the drone in the vertical axis, and θ is the angle of the drone's horizontal axis relative to the horizontal axis of the world coordinate frame.



In order to prevent the drone from colliding with its surroundings, we'll represent the presence of obstacles as a point cloud in space, contained in the set $P = \{p_1, p_2, \ldots, p_N\}$ where $p_i \in \mathbb{R}^2$, and use the dimensions of the rectangle representing the drone to create four intersected halfspaces. The height of the rectangle representing the drone is b_y, and the width of the rectangle is b_x.

Once we've used this to establish a criterion to determine whether new candidate points sampled by RRT are either valid or rendered invalid by a collision, we can integrate this rule into an implementation of RRT that can traverse 2D space while dodging any points in P.

(a) Defining collisions in body frame (5 pts)

When we are evaluating an abstract query point q in 2D space to see if it's contained in the intersection of all four halfspaces that collectively define a collision, each of the four inequalities that define these halfspaces are independent of the others, so they can all be evaluated simultaneously using a matrix A and vector b. Using q_b, a 2D query point expressed in the coordinate frame centered on the drone, **find the values in $A \in \mathbb{R}^{4\times 2}$ and $b \in \mathbb{R}^4$ required for the inequality $Aq_b + b \leq 0$ in terms of real numbers and the values b_x and b_y that describe the dimensions of the drone**. (Hint: If the drone is colliding with the query point, all four scalar inequalities should be true. If any of the four scalar inequalities is false, there is no collision.)

(b) Transforming from world frame to body frame (5 pts)

Each point in P is defined in the world coordinate frame, so we'll need to perform a coordinate transformation to convert the points in the world frame to the body frame so we can use the inequalities we described in the previous part of this question. Each point in the set P is arranged such that $p_i = (p_i^{(x)}, p_i^{(y)})^\intercal$, with the first and second entries representing its position in the x and y axes of the world frame, respectively. **Define**

the values of $\tau \in \mathbb{R}^2$ and $R \in \mathbb{R}^{2\times 2}$ in terms of x, y and θ, where τ is a vector describing the current position of the drone and R is a rotation matrix to rotate points about the origin by θ. (A positive θ value represents a counter-clockwise rotation.)

(c) Defining collisions in world frame (5 pts)

Using the relationship we just determined, we can substitute the query point q_b with q, the equivalent query point expressed in the axes of the world frame. **Use R and τ to write the inequality $Aq_b + b \leq 0$ in terms of q.**

(d) Implementing collision detection for RRT (10 pts)

Use the code in the Colab Notebook linked below to implement RRT with collision checking: specifically, **complete the function `is_free_state` of the `MidtermRRT` class in the file `P4_rrt.py`, then run the Colab Notebook** to run the algorithm on three maps with included point clouds representing the obstacles. **Remember that you will need to make a copy of the notebook to save your work!** To save and edit a copy of the notebook:

- Go to `File > Save a Copy in Drive`.
- Open your copy in a new tab, where you will have edit access.

Link: Colab Notebook for Implementing RRT with Collision Detection

Include the plots generated throughout the Colab Notebook in your submission.