COM SCI 188: Introduction to Robotics (winter 2026)

Coding Assignment 3: SLAM and Motion Planning

1. Overview

In this assignment you are **not required to implement EKF-SLAM or PRM/RRT yourself**.

You are given complete, working reference implementations and visualization code.

You are explicitly allowed—and encouraged—to use AI coding assistants (“AI teammates”) to help you:

- read and understand the code,
- write scripts to run experiments,
- generate plots, tables, and boilerplate.

The main goal is to learn by **designing, running, and interpreting experiments**, not by re-deriving all the math.

By the end, you should have a concrete, experimental understanding of:

1. How **EKF-SLAM** behaves under different trajectories and noise conditions.
2. How **PRM** and **RRT** compare as sampling-based motion planners in the same environment.
3. Simple ways to **improve** PRM and RRT and when those improvements help.

2. Provided Code

You are given the Python files listed below. You may read and modify these files (or create additional scripts) as needed for your experiments.

`environment.py`

- Defines a 2D environment with circular obstacle landmarks and bounds.
- Provides:
  - `Environment`: landmarks, obstacle radius, bounds, collision checking, simple visualization.
    environment
  - `RobotSimulator`: simulates a differential-drive robot with process + measurement noise and produces landmark measurements.
    environment
- Also includes `motion_model` and `wrap_angle` used by both SLAM and planning.
  environment

`slam.py`

- `EKFSLAM`: complete EKF-SLAM implementation (state `mu`, covariance `P`, `predict`, `update`, landmark initialization, etc.).
  slam
- Helper methods:
  - `get_estimated_landmarks`, `get_estimated_path`, `get_landmark_covariances`, `get_pose_covariance`, etc.
    slam
- `get_covariance_ellipse`: converts a 2×2 covariance into ellipse parameters for plotting.
  slam

`motion_planning.py`

- `PRM`: complete probabilistic roadmap planner with:
  - sampling of collision-free nodes,
  - k-nearest-neighbor connections,
  - A* search on the roadmap with optional visualization callbacks.
    motion_planning
- `RRT`: complete RRT planner with:
  - biased sampling toward the goal,
  - step-limited extension,
  - collision checking and path reconstruction,
  - optional visualization callback.
    motion_planning

COM SCI 188: Introduction to Robotics (winter 2026)

`test_EKF-SLAM.py`

- Runs EKF-SLAM in the `Environment` using `RobotSimulator` and visualizes:
    - true vs estimated robot trajectory,
    - landmark estimates and their uncertainty ellipses,
    - robot pose uncertainty ellipses.
      test_EKF-SLAM
- The info box shows statistics such as pose error and landmark uncertainty over time.
  test_EKF-SLAM
- `main()` defines a specific control sequence (straight, turning, etc.), noise settings, and runs the demo.
  test_EKF-SLAM

`test_PathPlanning.py`

- `PRMVisualizer` and `RRTVisualizer` show how PRM and RRT build their structures over time.
  test_PathPlanning
- `demo_prm` and `demo_rrt` run each planner from a fixed start to goal in the given environment and visualize sampling, tree growth, and final path.
  test_PathPlanning
- `main()` creates a single `Environment`, start/goal pair, then calls both demos.
  test_PathPlanning

COM SCI 188: Introduction to Robotics (winter 2026)

Part A – EKF-SLAM: Trajectory and Uncertainty

Goal: Use the already-implemented EKF-SLAM to understand how the **robot's path** and **noise levels** affect localization and mapping performance.

You should **build on** `test_EKF-SLAM.py`. You can:

- modify the control sequence and noise parameters,
- and add code to log numerical metrics.

A1. Path Design: Which Trajectory is Better for SLAM?

Design at least **two qualitatively different control strategies** for the robot in the same environment:

Examples (you can choose your own, as long as they're clearly different):

- **Path A – "Greedy Forward"**
  The robot mostly drives forward and rarely loops back.
- **Path B – "Looping / Revisit"**
  The robot regularly revisits earlier areas and re-observes landmarks.

Use the `RobotSimulator` and `EKFSLAM` from `environment.py` and `slam.py` (as in `test_EKF-SLAM.py`) but modify the **control pattern** inside the main loop to realize your two different paths.

For each path:

1. Run EKF-SLAM for a fixed number of steps.
2. Collect and plot:
   - true vs estimated robot trajectory (`simulator.get_true_path()` vs `ekf.get_estimated_path()`).
   - landmark estimates (`ekf.get_estimated_landmarks()`) and their uncertainty ellipses.
3. Log or compute numerical metrics over time, such as:
   - **Pose error** (true vs estimated position, as `np.linalg.norm(true_xy - est_xy)`).
   - **Pose uncertainty** (e.g., sqrt of trace of pose covariance from `get_pose_covariance()`).
   - **Average landmark uncertainty** (e.g., mean trace of landmark covariances from `get_landmark_covariances()`).

COM SCI 188: Introduction to Robotics (winter 2026)

Questions to answer:

- Which path leads to **lower pose error** and **lower map uncertainty** by the end?
- How do the uncertainty ellipses for landmarks behave over time under each path?
- Give at least one concrete example (with a figure) where the robot revisits an area and uncertainty clearly shrinks. Explain what's happening in terms of EKF-SLAM.

A2. Noise Sensitivity

Now pick one of your paths (the one you think is "good") and study the effects of **noise**:

- Vary the **process noise** `R` (motion noise) used in EKF-SLAM and `RobotSimulator`.
- Vary the **measurement noise** `Q` (range/bearing noise).

For example, try:

- Low, medium, and high process noise (keeping measurement noise fixed).
- Low, medium, and high measurement noise (keeping process noise fixed).

For each setting:

1. Run the same control sequence.
2. Record pose error, pose uncertainty, and average landmark uncertainty over time.
3. Generate plots (e.g., error vs time) and/or summary tables.

Questions:

- How does increasing motion noise vs increasing measurement noise affect:
    - pose accuracy,
    - map accuracy,
    - overall uncertainty?
- Which type of noise seems more damaging for this task? Why?
- Does your "good" trajectory still perform reasonably well even with higher noise, or does it break down? Explain.

COM SCI 188: Introduction to Robotics (winter 2026)

Part B – PRM vs RRT: Planning in the Same Environment

Goal: Use the provided PRM and RRT implementations to compare their behavior on the same obstacle field.

You should build on `test_PathPlanning.py`:

- Reuse the `Environment`, `PRM`, `RRT`, and visualizers.
- Add your own scripts (e.g., `experiments_planning.py`) that:
    - run multiple trials,
    - log metrics (success rate, path length, etc.),
    - and produce plots/tables.

B1. Start/Goal Scenarios

The provided `main()` chooses `start = [1, 1]` and `goal = [17, 17]` in a fixed landmark layout. Design at least **two different planning scenarios** by changing `start` and `goal` and editing the environment (obstacle positions or radii):

1. **Scenario 1 – "Relatively Open":** Start/goal positions that can be connected without having to squeeze between very tight obstacles.
2. **Scenario 2 – "Constrained / Narrow Passage":** Start/goal positions that force the path to go near or between obstacles (narrower corridors).

For each scenario:

- Run **PRM** and **RRT**.
- Save visualizations of:
    - roadmap / tree growth,
    - final path.

Before you run large experiments, write a short **prediction**: which planner do you expect to perform better in each scenario, and why?

COM SCI 188: Introduction to Robotics (winter 2026)

B2. Metrics and Single-Scenario Comparison

Pick one scenario (your choice, but state which one).

Define and measure at least:

- **Success rate**: fraction of runs where a valid path is found (within a time or iteration limit).
- **Average path length**: sum of Euclidean distances between consecutive waypoints, as is already printed in `demo_prm` and `demo_rrt`.
- **Planning effort**: e.g.,
    - number of iterations (`max_iter` in RRT),
    - number of sampled nodes in PRM (`n_samples`),
    - or wall-clock time if you want.

Write a script that:

1. Runs PRM **N** times and RRT **N** times (e.g., N = 20–50), with different random seeds.
2. Collects the metrics above.
3. Outputs at least one table comparing PRM vs RRT on this scenario.

Questions:

- Which planner has a higher success rate?
- Which produces shorter paths on average?
- Are there clear failure modes (e.g., RRT failing to “discover” a corridor, PRM failing because the roadmap is too sparse)?

Include **at least one** screenshot or figure illustrating a “typical” successful run for each planner.

COM SCI 188: Introduction to Robotics (winter 2026)

B3. Parameter Studies

Now systematically vary parameters to see how they affect performance:

For PRM (in `motion_planning.py`):

- Vary `n_samples` (e.g., 100, 200, 400, 800).
- Optionally vary `k_neighbors`.

For each parameter setting:

- Run multiple trials, record success rate and average path length.
- Plot these metrics vs `n_samples` (and/or `k_neighbors`).

For RRT (in `motion_planning.py`):

- Vary `step_size` (e.g., 0.2, 0.5, 1.0).
- Vary `goal_sample_rate` (e.g., 0.0, 0.1, 0.3, 0.7).

Questions:

- How does increasing `n_samples` for PRM affect:
  - success rate,
  - path length,
  - planning time?
    Is there a point where more samples don't help much?
- How does `step_size` in RRT influence:
  - speed of finding a solution,
  - path quality,
  - likelihood of missing narrow passages?
- When is a **higher goal bias** (`goal_sample_rate`) helpful, and when does it seem to hurt (e.g., by causing too many attempts to extend toward the goal through blocked regions)?

Summarize what you observe with at least one or two plots per planner.

COM SCI 188: Introduction to Robotics (winter 2026)

Part C – Improving PRM and RRT

Now that you understand the baseline behavior, propose and implement at least **one improvement** to **each** planner.

You are free to be creative, but here are some suggestions:

PRM Improvements

Examples:

- **Smarter sampling:** Bias more samples toward "interesting" regions such as near obstacles or in the middle of the map.
- **Better connectivity:** Make `k_neighbors` depend on local density or distance.
- **Path post-processing:** After finding a path, implement a simple shortcutting algorithm that tries to connect non-consecutive waypoints directly if the straight-line path is collision-free.

RRT Improvements

Examples:

- **Adaptive goal bias:** Start with low `goal_sample_rate` and increase it over time, or vice versa.
- *Basic RRT* flavor:* After adding a new node, optionally "rewire" it to a cheaper parent if that's easy to add.
- **Path smoothing:** Once a path is found, run a shortcutting procedure to reduce unnecessary detours.

For each improvement:

1. **Describe** what you changed (1–2 paragraphs).
2. **Design an experiment** where this improvement should make a difference (e.g., a narrow passage or cluttered area).
3. **Compare baseline vs improved** planner quantitatively (same metrics as before).
4. Include at least one figure showing a "before vs after" path.
5. Explain whether the improvement behaved as you expected and why.

COM SCI 188: Introduction to Robotics (winter 2026)

Using AI / Coding Agents

You may (and probably should) use AI tools (ChatGPT, Copilot, etc.) to:

- help you understand the provided code,
- help you write experiment scripts and plotting code,
- suggest ways to structure your analysis.

However:

- **You** are responsible for:
    - verifying that the code runs and does what you think it does,
    - debugging and fixing anything that breaks,
    - designing experiments that actually test something meaningful,
    - and writing **your own** explanations and analysis.

In your report, include a short paragraph describing **how** you used AI ("I used AI to generate plotting code, but wrote the experimental design and analysis ourselves," etc.).

Deliverables

Submit:

1. **Code / scripts** you used with a **README**
2. **Report (PDF, ~3–5 pages)**
    - Brief description of what you did in Parts A–C.
    - Figures (plots, screenshots of visualizations) with captions.
    - Tables summarizing quantitative results.
    - Clear answers to the guiding questions in each part.
    - Short description of your use of AI tools.