MOBILE ROBOTICS

Planning, Mapping & Localization

Building and evaluating the core autonomy stack in Python & ROS2

Cole Helmer · Robotics Engineering (RE 607) · Widener University

Three graduate robotics projects — path planning and optimal control, occupancy-grid mapping with graph SLAM, and Monte Carlo localization — implemented from scratch in Python and ROS2 and demonstrated in Gazebo/RViz simulation.

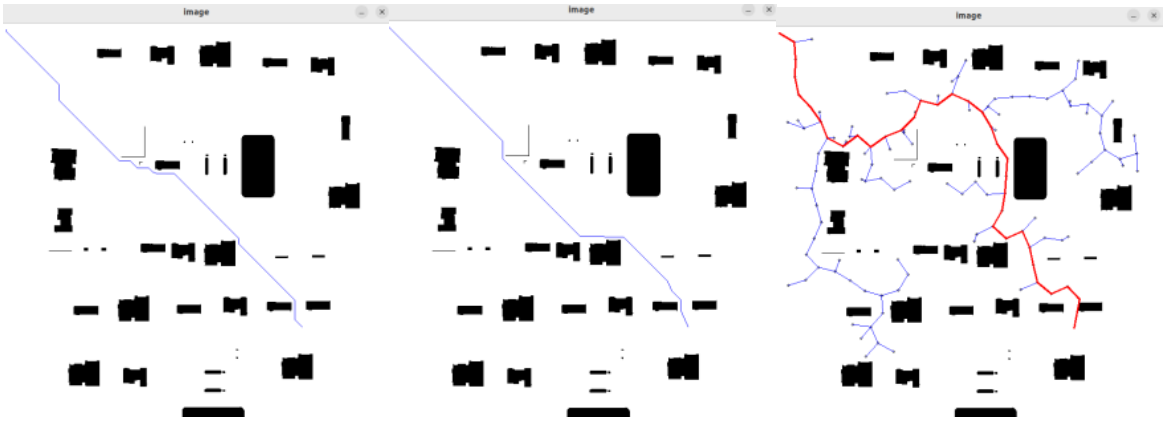
Overview

Autonomy in a mobile robot comes down to a handful of tightly-coupled problems: deciding how to move through the world, building a usable picture of that world, and knowing where you are inside it. Across a semester of graduate robotics work I implemented the core of that stack from the ground up in Python and ROS2, testing each piece in Gazebo and RViz simulation. This report brings three of those projects together — **path planning and optimal control**, **occupancy-grid mapping with graph SLAM**, and **Monte Carlo localization** — into a single picture of how a robot plans a route, maps its surroundings, and locates itself within them. Each was built and demonstrated in simulation, which kept the focus where it mattered: the algorithms themselves rather than hardware integration.

01

Path Planning & Optimal Control

The first project compared three classic planners on the same obstacle map: Dijkstra's algorithm, A*, and a Rapidly-exploring Random Tree (RRT). All three reach the same goal, but the way they get there is revealing. Dijkstra's produces clean, straight-line paths — it is guaranteed to find the shortest route, so it commits to direct segments from start to finish. A* lands on very similar paths but does it far more efficiently: where Dijkstra's might take three or four moves to step around an obstacle, A*'s heuristic resolves the same detour in a single turn. RRT takes an entirely different posture — rather than marching toward the goal, it seeds the start and end states and spreads outward, branching to probe the open space until a connected path emerges. It trades the optimality of the other two for the ability to explore wide, unstructured areas quickly.

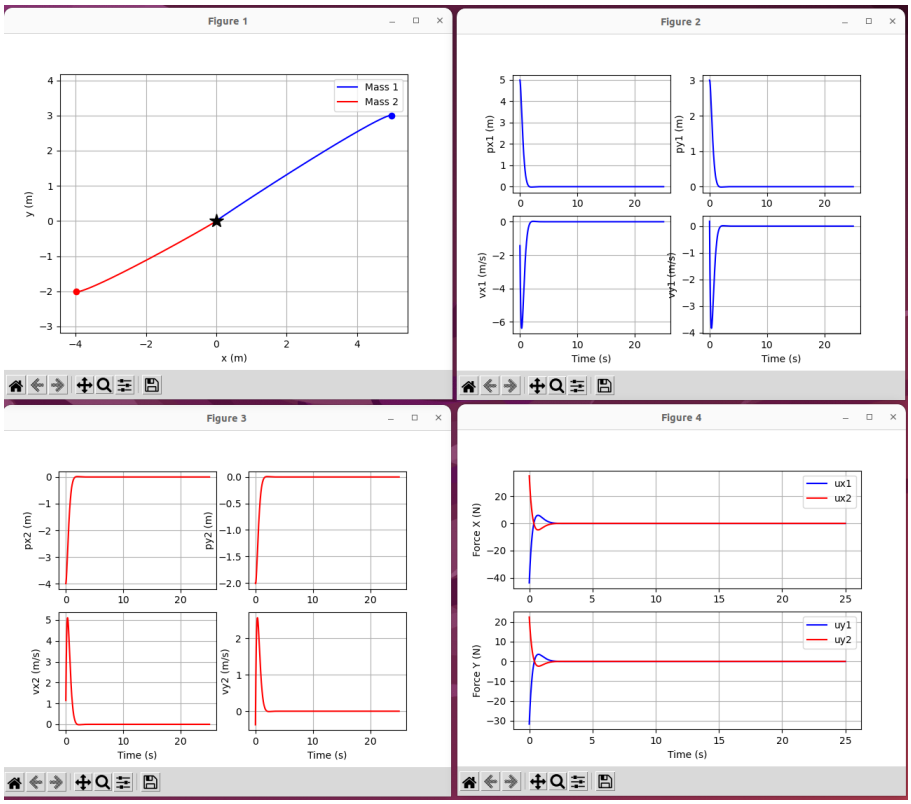


Dijkstra's — shortest, direct | A* — same route, fewer turns | RRT — space-filling search

The three planners on one map: Dijkstra's and A converge on near-identical shortest paths, while RRT explores outward through the free space before committing to a route.*

The control half of the project extended a linear-model trajectory-following example to a two-mass system under **LQR optimal control**. Where the original example tracked a single mass, I defined two state vectors and combined them into an eight-state model — four states per mass — then built block-diagonal A and B matrices and scaled each mass's cost matrices independently. The result is a

single LQR controller driving both masses along their reference trajectories at once.



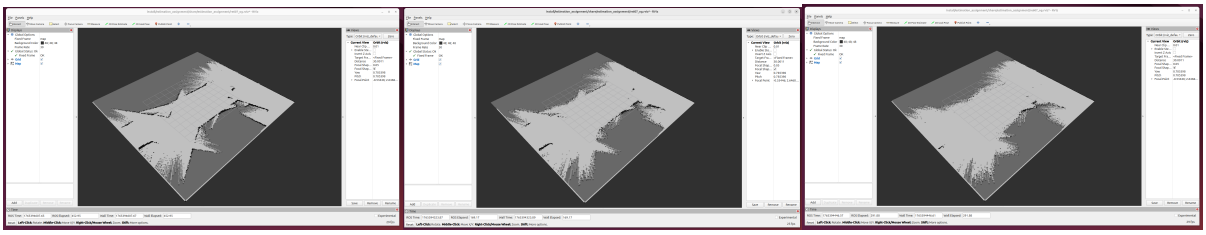
Two-mass LQR trajectory following — both masses tracked simultaneously by one controller built from block-diagonal state and cost matrices.

02

Estimation & Mapping — Occupancy Grids and Graph SLAM

A robot cannot plan through a world it cannot see. This project built an **occupancy grid** directly from the robot's laser scans as it roamed. Each incoming scan is range-checked against the grid, and every cell it crosses has its log-odds occupancy updated — free space drives the odds down, returns drive them up — before the accumulated grid is published as a map. The heart of the work was exactly that pipeline: confirm a cell falls within sensor range, update its occupancy from the laser measurement, and publish the result as the map converges.

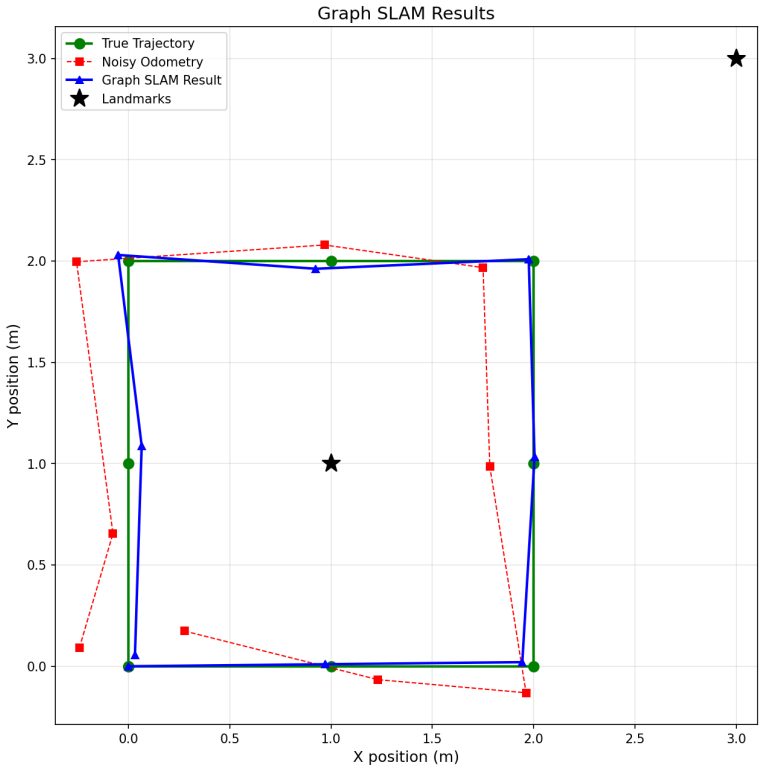
What made it interesting was watching the map degrade under sensor noise. With clean odometry the grid is crisp and well-defined, with minimal scatter and walls you could navigate from. As I introduced position and especially orientation noise, the walls thicken, scatter points multiply, and the outer edges blur; by the highest-noise setting the general shape survives but the walls are no longer cleanly readable. It is a concrete look at how localization error propagates straight into map quality.



No noise — clean, readable | Position + orientation noise | High noise — shape only

The same occupancy grid under increasing sensor noise — from a clean, well-defined map to one where only the general structure survives.

Alongside the mapping work, a **graph-SLAM** formulation tackled the complementary problem: correcting drifting odometry into an accurate trajectory. Plotting the true path against the raw noisy odometry and the SLAM-corrected estimate shows the optimizer pulling a badly-drifting odometry track back onto the ground truth — the same uncertainty that blurs the map, resolved at the trajectory level.



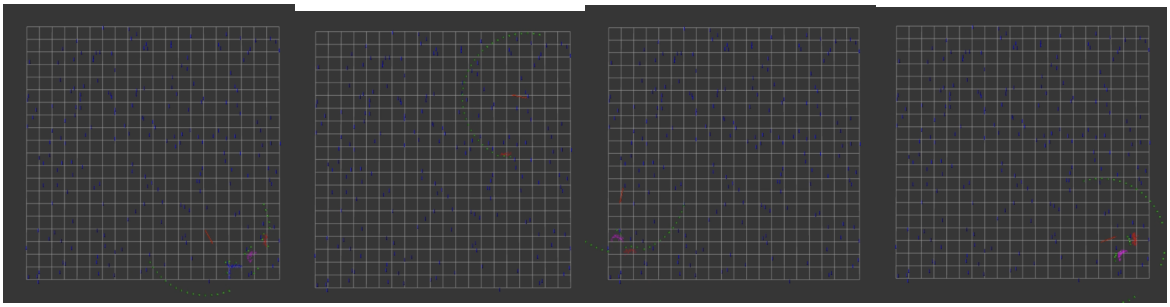
Graph SLAM: the corrected estimate (blue) recovers the true trajectory (green) from badly-drifting odometry (red).

03

Monte Carlo Localization

The final project closed the loop on localization with a particle filter — **Monte Carlo Localization**. I ran 200 particles, each one a hypothesis about where the robot actually is; the filter's job is to kill off the bad guesses and reinforce the good ones over time. I implemented the key pieces: a *handle-observation* step that propagates each particle forward from the odometry and assigns it a weight, a *resampling* step that continually redistributes particles toward the higher-weight hypotheses, and an *error-analysis* step that scores each particle against the incoming measurements to decide which survive.

In simulation the convergence is clean. The robot starts in the corner with several loose particle clouds representing near-total uncertainty about its pose. As measurements arrive, the particles conform into a couple of dominant groups as the better hypotheses gain weight, and by the final frames they collapse into tight clusters locked onto the robot — noticeably more precise on the second pass through the space.



1 — dispersed 2 — grouping 3 — converging 4 — locked on

Monte Carlo Localization converging: loose particle clouds (1) tighten into weighted groups (2–3) and finally lock onto the robot's pose (4).

Reflection

Taken together, these three projects are the backbone of mobile-robot autonomy: planning a path, building and maintaining a map, and staying localized within it. Implementing each in simulation let me focus on the algorithms themselves — the search and optimization, the probabilistic estimation, the sensor-fusion bookkeeping — rather than the mechanics of a physical platform. The thread running through all three is the same: real sensor data is noisy, and good autonomy is mostly about reasoning carefully under that uncertainty.