Simulator/Architecture Details

Kevin Robb

Contents

1 Introduction

I've created a rudimentary simulator entirely from scratch in Robot Operating System (ROS Noetic) for this project. In order to work with any filter or localization method that I want to test/demo, I designed it to contain several "nodes" which each handle an important part of the simulation; these nodes communicate with each other by publishing or subscribing to certain topics, and all nodes run simultaneously.

Some base scripts/nodes run in every demo to setup the structure and generate data for the filter being examined. All filters have the option to subscribe to the following:

- Control Commands:
  - Forward velocity.
  - Angular velocity.
- Measurements; for all landmarks within range/FOV of vehicle's camera:
  - Landmark's unique ID.
  - Range to landmark from vehicle position.
  - Bearing to landmark relative to vehicle orientation.
- True Landmark Map, for localization-only filters.
- True Vehicle Pose, for mapping-only filters.
- Occupancy Grid Map.

For instance, the EKF-SLAM node will subscribe to the control commands and the landmark measurements, and will publish its current estimate of the "state", which includes the estimated vehicle pose and estimated landmark positions.

In this 2D simulation, the robot/vehicle is completely defined in the space by its x, y, and θ components, i.e., its 2D position in the world and its global heading, which is in the range $[-\pi, \pi]$, with $\theta = 0$ being to the right. The world frame can be defined however we like; for simplicity, we usually assume the robot's starting pose is the origin, but in my architecture I define a global "map" coordinate system w.r.t. which the robot's initial pose is specified. This helps the plots to keep everything visible and centralized.

2 Environment

The environment has no border, and consists of features called landmarks. A map is then simply a dictionary mapping each landmark's ID to its position.

Note: In a real-world application, these would be unique, recognizable features, such as AprilTags or other visual fiducials, brightly colored balls that can be easily thresholded in HSV and clustered, or known objects that can be fit to a depth pointcloud with RANSAC. Regardless, we assume a layer of abstraction from what a "landmark" actually is; a different ROS node could be dedicated to using real-world data of some kind to generate abstract landmark measurements of this form.

We can generate a random map by specifying some parameters.

- `num_landmarks`.
- `bound`. All landmarks will lie within ±`bound` in both x and y.
- `min_sep`. No landmarks will be within `min_sep` of another landmark.

We can then create a map by randomly generating coordinates within bounds and saving or rejecting them based on `min_sep` until `num_landmarks` has been reached. For simplicity, the IDs are the integers 0 to `num_landmarks` $- 1$.

I've also made an option to generate landmarks in a discrete grid.

We have a second map in the form of a color image that will be thresholded to form a binary occupancy grid. This is used for path-planning, but does not obstruct vision or prevent vehicle motion. A more robust simulation would potentially handle collisions and perform raycasting to determine line-of-sight, but these details were low priority for the goals of this project.

3 Control Commands

The robot can be controlled once per timestep with a certain odometry, consisting of a forward velocity and angular velocity, resulting in that distance being achieved in the next timestep (For simplicity in the math, we assume a timestep is one "time unit" in length, so $d = v \cdot t = v$). We call this command $\boldsymbol{u}$ by convention. Thus, the command for some timestep t will be $\boldsymbol{u}_t = \begin{bmatrix} \mathrm{d}\,d & \mathrm{d}\,\theta \end{bmatrix}^T$. I've allowed for these commands to be generated in a few ways:

- A full trajectory is generated up front, and commands are fed one at a time into the simulation. This trajectory generation is described in the following section.
- The user clicks on the screen, and the vehicle directly pursues this point.
- The user clicks on the screen, and the `goal_pursuit_node` uses A* to find the optimal path there through free space, and navigates with Pure Pursuit.
- The `goal_pursuit_node` autonomously chooses goal points, and navigates to them.

After a command is chosen for a timestep, the `sim_node` enacts a noisy version of it on the "real" vehicle, determines what the robot measures given the true vehicle pose and true landmark positions, and publishes a noisy version of these measurements for the filters to use.

4 Trajectory Generation

The goal for full trajectory generation is to create a non-trivial robot path that will come close to all landmarks at least once. This gives the best paths for evaluating different filters and environments.

I did this by treating the map as a travelling salesman problem (TSP); this is basically a general approach in graph theory for creating a path that visits all nodes in some graph. The simplest solution to the TSP is the nearest-neighbors heuristic. This strategy entails choosing the nearest node to the current node that has not yet been visited, and choosing to go there next. This continues until all nodes have been added to a list. This is usually not the most efficient TSP solution, but since our objective is to just move around the map, it will suffice.

My trajectory generator creates such a list for the map that has been provided, and then generates odometry commands within constraints to get within some threshold of each landmark. When a landmark is "visited", it is moved to the end of the list; this way, the vehicle will continue moving until time runs out, rather than stopping once it has visited all landmarks.

This strategy helps to minimize the uncertainty in the EKF estimates, since the uncertainty grows every timestep and can only shrink when a landmark is detected. The trivial approach to truly minimize uncertainty would be to find a landmark and then sit stationary in front of it for the duration, but this isn't very interesting.

We track the true position for the entire trajectory (since it's needed for further planning and for generating measurements) and also save a noisy version of all odometry commands that has the process noise $\boldsymbol{v}$ applied to it. This ensures the EKF isn't being fed perfect data, and brings this simulation closer to reality. The noisy commands are published one at a time to the EKF at some specified frequency `DT`.

5 Measurements

Generating measurements given the map and true vehicle position is fairly straightforward. We compare all landmark positions to the robot's to get a distance and bearing to each. We then reject all landmarks whose range is further than the robot's maximum vision range or whose bearing is outside the robot's constrained FOV. For all suitable landmarks that satisfy both vision constraints, their ID, range, and bearing are saved and obfuscated with the measurement noise $\boldsymbol{w}$.

Thus, our measurement $\boldsymbol{z}$ at some timestep will be zero, one, or more $\boldsymbol{z}_{t,id} = \begin{bmatrix} id & r & \beta \end{bmatrix}^T$, where id specifies the landmark that has been detected. Any number of landmarks can be detected on a single timestep, and the measurements for all are concatenated together to be used by the filter.

6 Visualization

I've created a dedicated `plotting_node` which shows a live view of the ground truth map and trajectory overlaid with the EKF's current estimate for the map and the vehicle pose. This is the best way to get an idea of the effectiveness of the EKF for Landmark SLAM, and see how this effectiveness may change depending on the random layout of the map and the trajectory.