ME 132 a: Lab 2

Shir Aharon
Tiffany Huang
Steven Okai

March 13, 2013

1 Part I: Point-to-point Control

Notes on how to run code:

The mainloop for this section is in the `lab2_part1.cc` file, with helper functions in the `lab2_shared.h` file. For interfacing with the Player/Stage system, we reuse the code from the previous lab in `common_functions` and `cmdline_parsing` files.

The code is currently hardcoded to expect a file named `goal_points.txt` which consists of the (x, y) points to which the robot should attempt to drive to.

1.1 go_to_point() function

For this function, we start by computing the relative angle that the robot needs to turn, and the distance it needs to drive. The `theta_dot` returned is half of the remaining distance to turn, which is done to prevent overshooting the necessary rotation amount before the next time step. The `r_dot` calculation is a little more complicated. First, if the current direction is more than 22.5° off, then the robot is told not to move and just turn. This prevents motion in the incorrect direction. Now again to prevent overshooting the destination or moving too far off from the direct line connecting the current robot location and the goal point, the speed returned is at most the distance to travel. This is divided by 10 for overshoot reasons, but can be scaled back up by the log of the angle offset. Thus the closer to being pointed directly at the goal point, the faster we can safely move.

These parameters were determined somewhat by trial and error. The log scaling for the speed was done in an attempt to move as fast as was deemed "safe". The constraints on the turning speed and the 22.5° restriction come from experimentally attempting to drive around the map without colliding into walls. In particular, this configuration seemed to allow driving in the center of a 1m wide path on the occupancy map.

To handle real world constraints, if the speed is under some threshold for 0, then we make the rotation a large enough value (except if close to 0). This accounts for the motor power that does not have very fine control, but prevents turning if the robot isn't supposed to move. Additionally, we check if we are close to a wall in which case we simply stop all motion.

1.2 Simulation Component

The code for this is in the `lab2_part1.cc` file. It simply connects to the robot and then iterates through the goal points given in a loop that checks if close enough to the current point, and then getting updating speed/direction values.

1.3 Lab Component

Demoed in lab.

2 Part II: Occupancy-Grid Maps

Notes on how to run code:

The mainloop for this section is in the `lab2_part2.cc` file, with helper functions in the `lab2_shared.h` file. Again, for interfacing with the Player/Stage system, we reuse the code from the previous lab in the `common_functions` and `cmdline_parsing` files.

As before, the code is currently hardcoded to expect a file named `goal_points.txt` which consists of the (x, y) points to which the robot should attempt to drive to.

Graphics Method

The code generates a file named `data.txt` which stores the probability values from the occupancy grid by directly writing them out in a comma separated form. Subsequent grids are written out after the previous one

in each time step. This is then plotted in matlab by the function `lab2_part2_plot` in `lab2_part2_plot.m`. The matlab code has the sizes hardcoded in, along with the image to use.

2.1 occupancy_grid_mapping() function

The `occupancy_grid_mapping` function simply checks if each point is within the 90° on each side of the robot that the laser scanner can detect. If this is the case, the `inverse_range_sensor_model` is called on that grid point. This function finds the laser data point that has a matching angle to the line between the robot and the current point (i.e. on the correct ray). Now the correct probability is returned based on the range of that laser point. If the same as the distance of the point, it is marked occupied. If shorter, than it is marked free. Otherwise, it is marked as unknown. If no laser data is within 0.57° of the point, then it is marked as unknown.

2.2 Simulation Component

The driving code is in the `lab2_part1.cc` file. It simply connects to the robot and then iterates through the goal points given in a loop that checks if close enough to the current point, and then getting updating speed/direction values. At each time step, it updates the occupancy grid and writes it into the data file.

2.3 Lab Component

Demoed in lab.

2.4 Group of 3

2.4.1 Surprises About the Method

Compared to the book, the forward sensor model method proposed in the Thrun paper was surprising because the algorithm cannot be performed incrementally considering that the proposed algorithm needs all the data before it can begin to maximize the occupancy map. The new method is also computationally expensive. Many calculations are required to compute the expected log likelihood of all the data and maximize it. Due to these two traits, this method has very limited real time applications. Many systems require a real time mapping algorithm, so it is surprising that this proposed approach cannot be used in such situations. Additionally, the paper mentions that the algorithm keeps the noise in the result usually and does not always find a unique optimum map, which makes the method seem a bit unreliable.

2.4.2 Computationally Expensive Components

The most computationally expensive components of this approach seem to be maximizing the expected data log likelihood because the computation for this part has an iterative nature and involves multiple exponentials, Gaussians, and sums, which all take a while to compute. This aspect of the approach requires multiple passed through data, which is also slow and does not scale well with grid size.

2.4.3 Difficulties in Implementation

The difficulty in implementing this algorithm seems to lie in achieving a decent computational speed because of the computationally costly expected log likelihood maximization with the E (expectation) and M (maximization) steps. The paper mentions that they can get a map in less than a minute on a low-end PC, which is a very long time, especially for real time applications. Other problems that could occur when implementing this approach are that α cannot be computed to determine the occupancy uncertainty so an α has to be set by hand and the algorithm cannot update incrementally, which prohibits real time applications.

2.4.4 Approximations

An approximation that can be made to make the implementation of the proposed method workable, but perhaps not sufficient for use in practice is approximating the marginal posteriors used in calculating the residual occupancy uncertainty because calculating the marginal posteriors is computationally intractable. Additionally, the constraint on when the optimum map is reached could be relaxed to reduce the number of iterations the method requires for a faster computational speed. Next, Gaussian approximations can replace some of the Gaussians in the log likelihood calculations to speed up the interior of the loop. Another possibility is to use a local max instead of searching for a global max to find the optimum map and to run the search for a max on every few incremental points for faster convergence. One other option is to combine this algorithm with the conventional occupancy grid map algorithm by running the conventional algorithm in real time and the proposed algorithm in the paper in the background. This approach would provide a more practical real time application for the proposed method in the paper, but would require additional work to maintain coherency in the grid data as this model lags behind the newest data.