CALIFORNIA INSTITUTE OF TECHNOLOGY

ME/CS 132a, Winter 2013

Lab #2 Due: 15 March 2013

Introduction

Occupancy-grid maps (and their various forms) have been a fundamental component to a majority of robotic systems throughout the years. They define for the robot what the world looks like (in a simplistic sense) and allow the robot to navigate through that world without collisions – in some cases, such maps also allow the robot to interact with the world through manipulation. Robotic systems typically start without any knowledge of the world in which they are placed in. As such, the maps they build of the world only become useful when the robot is continually exploring the environment. Exploration is thus contingent on motion – so you can see how maps and motion go "hand-in-hand".

In this lab you will develop, implement, and test an Occupancy-Grid Map as well as a simple point-to-point motion controller to allow semi-autonomous exploration for populating the map.

The goals of this lab are to

1. write a simple motion controller that moves towards specified goal points;
2. implement an occupancy-based grid mapper that will be used next term for the path-planning algorithms you will implement; and
3. develop your C/C++ writing skills in preparation for ME132B next term (which will require a significant amount of coding for many of the algorithms that will be asked of you to implement).

Part I: Point-to-point Control [60 pts total]

1 `go_to_point()` function

[20 pts] Develop a simple point-to-point controller function call in C/C++ that generates the necessary input signals (motor commands) for a two-wheeled, non-holonomic robot to achieve a desired location (with any orientation at that location). This function should be called each timestep to re-generate inputs (to be applied that timestep) until the desired goal location has been reached.

Your function call should have the following form:

```
int go_to_point(double goal_x,  double goal_y,
                double robot_x, double robot_y, double robot_theta,
                double* r_dot, double* theta_dot);
```

- The first two arguments to the function (`goal_x`, `goal_y`) define where you want your robot to go (i.e. the "goal" location).
- The next three arguments (`robot_x`, `robot_y`, `robot_theta`) define the current state of the robot for that timestep – necessary information for your controller to determine the motor inputs.
- The remaining two arguments (`r_dot`, `theta_dot`) are the control inputs your function calculates and returns to you. The notation indicates those variables are pointers and memory for them needs to be allocated outside of the function.

NOTE: YOU CANNOT USE THE **GoTo** FUNCTION IN PLAYER/STAGE (e.g., as found in the Position2dProxy class). Even if you try and use it, you'll find it won't work on the robots in the lab. The purpose of this part of the lab is to have you think about different ways of commanding the robot to go from one point to the next. Be creative!

HINT: One approach you may want to consider is a state-feedback control law. State-feedback controllers are typically expressed by the following form:

$$\mathbf{u} = -\mathbf{K} \cdot \mathbf{e} \tag{1}$$

where $\mathbf{u}$ defines the control input to the system you intend to control (i.e. the robot), $\mathbf{e}$ defines an error state of the robot (i.e. the "error" or "how-far" the robot is from the end goal location) and $\mathbf{K}$ defines the control gains you adjust to get the desired behavior. In this particular case, one implementation of a state feedback control system could look like the following:

$$\begin{bmatrix} \dot{r} \\ \dot{\theta} \end{bmatrix} = - \begin{bmatrix} K_{rr} & K_{r\theta} \\ K_{\theta r} & K_{\theta\theta} \end{bmatrix} \cdot \begin{bmatrix} \delta_r \\ \delta_\theta \end{bmatrix} \tag{2}$$

where δ_r is the radial distance error to the goal and δ_θ is the angular heading error to the goal, in polar coordinates. Alternatively, a similar control law could be developed in Cartesian coordinates.

Note also that if using the above control law, special attention is needed to handle the discontinuity between $\pi/-\pi$.

2 Simulation-Component

[20 pts] Once you've written your `go_to_point` function, write a simple Player program called `lab2_part1.cc` that does the following:

- Load a list of points specified in a provided text file (named `goal_points.txt` that will have the following form):

$$\begin{matrix} goal1_x & goal1_y \\ goal2_x & goal2_y \\ \vdots & \vdots \end{matrix}$$

- Command the robot to each goal point in sequential order from the provided input file using your `go_to_point` function. You can consider a point "reached" when it has reached a certain distance radius threshold of $5cm$.

- Exit when the last goal point has been reached.

Test it in the simulated Stage environment using the `freespace.cfg` file. This configuration file specifies a relatively large world to test your function results without having to worry about obstacles.

3 Lab-Component

[20 pts] This portion of the assignment is to be tested during your allotted lab time. During your lab time, demonstrate to the TAs that your `lab2_part1.cc` program works in simulation. Once verified, the TAs will provide you with a unique `goal_points.txt` file, listing a set of goal points for your robot to achieve.

4 What you will submit

To be clear about what gets submitted (and how) for this part of the lab, here's a checklist:

- C/C++ source code for `go_to_point` function
- C/C++ source code for `lab2_part1.cc` program
- Instructions on how to run your program for the TAs to test completeness of your algorithm.
- **lab time:** demonstrate your point-to-point function works for any point in the workspace of the robot.

Part II: Occupancy-Grid Maps [60 pts total]

1 `occupancy_grid_mapping()` function

[20 pts] The performance of occupancy grid maps are generally a function of the resolution of the chosen grid cells and also the quality of the sensor measurements themselves. Additionally, the pose of the robot also plays a significant role as uncertainty in robot pose can propogate into "smearing" of data in the overall map.

In this part of the lab, we will assume that the robot pose from wheel odometry is perfect[1]. With this assumed perfect pose, fix the reference frame for the map to be coincident with the world origin and choose a **map size of** 100×100 **cells with a cell resolution of** $5cm$**. This will cover a floor spacing of** $5m \times 5m$**.** Understand that anything beyond the perimeter of $5m$ on either side of the world origin will be outside the bounds and cannot be represented in our chosen map. Other implementations of occupancy grid maps exist which center the map on the robot and allow obstacles in the immediate vicinty of the robot to be captured. We will not deal with moving maps in this lab.

[1] Understanding that wheel odometry is never perfect, we will make this assumption anyway for this part of the lab and see the effects of this poor decision in the generated results.

Algorithm 1 occupancy_grid_mapping$(\{l_{t-1,i}\}, x_t, z_t)$

1: acquire measurements z_t from sensor
2: acquire state x_t from robot odometry
3: **for** all cells $\mathbf{m}_i \in m$ **do**
4: **if** $\mathbf{m}_i$ in perceptual field of z_t **then**
5: $l_{t,i} = l_{t-1,i}$+ **inverse_range_sensor_model**$(\mathbf{m}_i, x_t, z_t)$ - l_0
6: **else**
7: $l_{t,i} = l_{t-1,i}$
8: **end if**
9: **end for**
10: return $l_{t,i}$

Following Algorithm 1 presented in class[2], implement in C/C++ a probability-based occupancy grid map function call, using the "log odds" ratio to avoid numerical instabilities associated with zero probabilities:

$$l_{t,i} = \log \frac{p(\mathbf{m}_i | z_{1:t}, x_{1:t})}{1 - p(\mathbf{m}_i | z_{1:t}, x_{1:t})} \tag{3}$$

where z_t represents a scan measurement of 180 laser range points, x_t represents the state of the robot, and $\mathbf{m}_i$ represents the i^{th} grid cell of map m. In your implementation, use the following probability values for **free** (p_{free}), **occupied** (p_{occ}), and **prior** (p_0):

$$\left\{ \begin{array}{ccc} p_0 & = & 0.7 \\ p_{free} & = & 0.10 \\ p_{occ} & = & 0.90 \end{array} \right\} \rightarrow \left\{ \begin{array}{ccc} l_0 & = & \log \frac{p_0}{1-p_0} \\ l_{free} & = & \log \frac{p_{free}}{1-p_{free}} \\ l_{occ} & = & \log \frac{p_{occ}}{1-p_{occ}} \end{array} \right\} \tag{4}$$

Recall that the **inverse_range_sensor_model**() function is sensor dependent. In this lab, you will be using a laser range finder, which fits the sensor model presented by Thrun *et al.*[3]. However, that model is computationally expensive. Instead, you will be implementing one that is much faster but requires some computational geometry (Algorithm 2):

Algorithm 2 inverse_range_sensor_model(m, x_t, z_t)

1: define polygon P formed by the vertices in measurement z_t
2: **for** all cells $\mathbf{m}_i \in m$ **do**
3: **if** $\mathbf{m}_i$ inside P **then**
4: return l_{free}
5: **else if** $\mathbf{m}_i$ on boundary of P **then**
6: return l_{occ}
7: **else**
8: return l_0
9: **end if**
10: **end for**

One of the biggest challenges of this part of the lab is getting your mapping algorithm to run fast enough to do the necessary computations within the expected control rate of your `go_to_point`

[2]Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. "Probabilistic Robotics"
[3]Probabilistic Robotics, Ch. 9 Table 9.2

motion controller. The biggest sink of your processing time will be in the map update, particularly determining freespace in the world. You will likely have to find a fast and efficient way of determining whether a map cell lies inside or outside the (likely concave) polygon defined by the set of all laser points in a given scan. You should make a serious effort at solving this yourself before consulting external references.

2 Simulation-Component

[20 pts] Once you've written your **occupancy_grid_mapping** function, write a simple Player C/C++ program called `lab2_part2.cc` that does the following:

- Using the functionality developed in Part 1, load a list of points specified in a provided text file (named `goal_points.txt`) and command the robot to navigate to each point sequentially, using the same metric of a $5cm$ radius threshold for deciding when a point is "reached".
- While the robot is executing its commanded motions, continually call the *occupancy_grid_map*() function to generate a probabilistic occupancy grid map for each timestep.
- Either write a visualizer in C/C++ or save a copy of the occupancy grid map at each timestep such that you can generate a movie (online or offline) illustrating the development of the probabilistic occupancy grid map overall timesteps of robot travel.
- Exit when the last goal point has been reached.

Test it in the simulated Stage environment using the `occupancy.cfg` file. This configuration file specifies a relatively large world with some obstacles to test your mapping function results. Generate a `goal_points.txt` file that acts like a trail of breadcrumbs, leading your robot throughout the world one point at a time (you will learn next term how to autonomously plan motions for your robot to follow).

3 Lab-Component

[20 pts] This portion of the assignment is to be tested during your allotted lab time. During your lab time, demonstrate to the TAs that your `lab2_part2.cc` program works in simulation. Once verified, the TAs will provide you with a unique `goal_points.txt` file, listing a set of goal points (also like breadcrumbs) for your robot to achieve and will observe the output of your occupancy grid map.

4 If you are a group of 3, or thirst for more

If your group has three members, then in addition to the above, you must also consider a method that does not assume all cells are independent. To that end, consider the maximum a posteriori occupancy grid algorithm as appearing in Table 9.3 (page 301) of Thrun *et al.* (2006). Read the extended technical note from Thrun (linked on the class website) describing the forward sensor models and how they can be used in an Expectation-Maximization algorithm.

Prepare a summary of what needs to be done to complete that implementation, **but do NOT implement this approach**:

In particular, answer the following questions:

- What is surprising in the method proposed in the paper, compared to the book?
- What strikes you as the most computationally expensive components of this approach?
- Is the difficulty in achieving a decent computational speed, or just to get an implementation to work at all? (i.e. is this solution even tractable?)
- Are there approximations that can be made to make the implementation workable, but perhaps not sufficient for use in practice?

The point value of this part will blend into that of the above, with a nominal division of each of the two approaches being 50% of the total; e.g., the "lab component" will be worth 10 points for each approach.

Groups comprising two students **are not required to do this and will not receive extra credit if they do so**. However, if you implement and demonstrate this approach, we are happy to evaluate your work and give you feedback.

5 What you will submit

To be clear about what gets submitted (and how) for this part of the lab, here's a checklist:

- C/C++ source code for `occupancy_grid_mapping` function
- C/C++ source code for `lab2_part2.cc` program
- Instructions on how to run your program for the TAs to test completeness of your algorithm.
- **lab time:** demonstrate your occupancy grid mapper works via the output display images in your visualizer (real-time) OR run a MATLAB or Python script (or other major language) to playback saved images of your occupancy grid map.