EE106A: Lab 8 - Building Occupancy Grids with TurtleBot*

Fall 2018

Goals

By the end of this lab, you should be able to:

- Use the ROS parameter server to set parameter values that can be shared across multiple nodes
- Understand and explain how an occupancy grid works and when to use one
- Map out the lab space using your own custom occupancy grid
- Point out any important deficiencies in your implementation

---

Contents

Introduction

In this lab, we will build and test one of the most useful datastructures in mobile robotics: the occupancy grid.[1] The key idea behind the occupancy grid is to represent space as — you guessed it — a grid, in which every cell, or *voxel*, is either occupied or free. Since nothing is ever really certain in life (i.e., measurements are noisy), occupancy grids actually keep track of the *probability* that each cell is occupied. When the robot receives a measurement of the environment, typically from a laser scanner, it updates these probabilities to incorporate the new information.

This lab is broken up into three phases:

1. Learn how to use the ROS parameter server.
2. Write the key steps in an occupancy grid update.
3. Test your implementation and identify any shortcomings.

*Developed by David Fridovich-Keil and Laura Hallock, Fall 2017. Updated by Valmik Prabhu, Nandita Iyer, Ravi Pandya, and Philipp Wu, Fall 2018

[1] A Google search for "occupancy grid" turns up lots of great references that go into more detail.

1 The ROS parameter server

We haven't really exposed you to the ROS parameter server before, but since it is one of the more useful features of ROS, we want you to get some practice using it.[2] ROS parameters are key-value pairs that ROS allows you to specify when launching a nodes (e.g., in a `launch` file) that may be queried by those nodes at run-time. This can be an extremely useful tool for writing flexible code and for enforcing that multiple nodes hold the same value for some particular variable.

Open the `lab8_resources.zip` file, which contains a ROS workspace called `lab8` that you should copy into your `ros_workspaces` folder (or wherever you keep your ROS workspaces). Make sure to initialize the workspace before building. Inside the `lab8` workspace, you'll find a package called `mapping`, which contains two source files and a launch file, along with the usual `CMakeLists.txt` and `package.xml` files. Open each of these files and make sure that they all make sense to you.

Inside the file `demo.launch`, you'll see that a number of command-line arguments are declared (along with default values). These arguments are then mapped to specific parameters in a node called `mapper`. These parameters will need to be read in by that node at run-time.

Open the file `occupancy_grid_2d.py` and locate the `LoadParameters` function. We've loaded one parameter for you, but you'll need to finish this function by loading the rest. Note that two of the variables in `occupancy_grid_2d.py`, `x_res` and `y_res`, are not on the parameter server. How do you think you should generate these variables (You should not be editing the launch file)?

Checkpoint 1

Get a TA to check your work when you've finished writing the `LoadParameters` function. When you run the launch file, you shouldn't see any error messages complaining about failing to load parameters.

---

2 Generating & updating the occupancy grid

Now for the fun part! In the file `occupancy_grid_2d.py` file, locate the function `SensorCallback` and fill in the details. The main idea here is that each grid cell contains the *log-odds ratio of occupancy*. That is, if p_{ij} is the probability of occupancy at cell (i, j), then the cell actually stores log-odds $\ell_{ij} \triangleq \log\left(\frac{p_{ij}}{1-p_{ij}}\right)$. This may seem like an unnecessary mathematical complication, but it's actually very useful: if we stored probabilities directly, we'd run into trouble trying to keep all of our probabilities positive when performing updates.

When a scan ray terminates at a particular cell, that cell's log-odds ratio is incremented by some small amount — i.e., $\ell \longleftarrow \ell + \Delta_{occ}$ — and then thresholded for numerical stability. Likewise, when the ray passes through a cell (and does not terminate there), that cell's log-odds ratio is decremented by some other amount — i.e., $\ell \longleftarrow \ell + \Delta_{free}$, where by convention Δ_{free} is negative — and similarly thresholded. In particular, these increments are computed as the log-odds ratios corresponding to *the probability that a cell is occupied given that a ray terminates there* and *the probability that a cell is occupied given that a ray passes through it*, respectively. Note that if our sensors were perfect, these values would correspond to 1 and 0, respectively; if that were the case, what would the log-odds update values be?

Before starting any edits, read through the inline comments and try to understand what the function is doing at each step. This callback function recieves a `sensor_msgs/LaserScan` message, which represents a single line depth scan around the robot (as would be generated by a LIDAR). The scan begins at some angle, gathers range information

[2] See `http://wiki.ros.org/rospy_tutorials/Tutorials/Parameters` for a more detailed description of the server's purpose and usage.

at a certain angular increment, and ends at some second angle. Use `rosmsg show` or the online ROS documentation to see the contents of this message.

The callback function iterates through each ray of the scan using the `enumerate` function (look up the documentation for this function if you don't understand what it's doing). The first thing you'll be implementing is finding the angle of the ray in the *fixed frame*. A quick look at `demo.launch` shows that the fixed frame is called `odom`, while the sensor frame is `base_link`. The frame `odom`, which stands for odometry [3], is coincident with `base_link` when the robot is turned on. As the robot moves, `odom` moves in the opposite direction relative to `base_link`. Thus, if you set your fixed frame in RViz as `odom`, you'll see `base_link` moving as the turtlebot moves in the real world. Note: if you move the turtlebot manually (say by picking it up), the odometry won't be able to detect it and the `odom` frame will be wrong. If you do this, restart the bringup sequence on your turtlebot to reset the odom frame.

The next thing you'll be doing is "walking" backwards along the ray from the scan point to the sensor, updating the log-odds in each voxel the ray passes through. The `numpy.arange` function can be helpful in defining your loop. The function `PointToVoxel`, defined below `SensorCallback`, may be useful as well. If a voxel is occupied, you should increase the log odds at that voxel by your occupied update value, thresholding it at your occupied threshold value. If a voxel is free, you should increase the log odds at that voxel by your free update value, thresholding it at your free threshold value. Remember that you should only be updating each voxel once per ray.

When you're done, try running the launch file again, and make sure you don't get any error messages.

3 Testing your occupancy grid

Look back at the launch file again. You'll notice that the node's main source file is `mapping_node.py`, not `occupancy_grid_2d.py`. (Although this project is small by most standards, it is generally good practice to separate the actual executable node file from other files implementing different classes that your node uses.) Examine how the `mapping_node.py` file creates an occupancy grid, initializes it, and on success just idles. If you trace that initialization call into the `OccupancyGrid2d` class, you'll see that initialization loads all parameters, registers publishers and subscribers, and sets up any other class variables. If any of that fails, it returns `False`, which causes the whole node to crash. This is a very safe way to build your system because it minimizes the chance that your code crashes mid-operation. We strongly encourage you to use this sort of architecture in your projects.

Bringup the TurtleBot with a minimal launch, then launch the 3D sensor. (Refer to Lab 6 for instructions; make sure to set your `ROS_MASTER_URI` appropriately!) Run the launch file from the `mapping` package, and open `RViz`. Find and visualize the topic on which the occupancy grid is being published. You'll also need to change the fixed frame to `odom` and add `tf` to the display so you can see where the TurtleBot is. Add the `/scan` topic to the display to show the laser scanner's output in real time.

Drive the TurtleBot around with the `turtlebot_teleop` node we used in Lab 6. You should see the floorplan begin to emerge as you drive around. Do you notice any systematic errors? Where are they coming from, and how would you address them?

Next, experiment with changing some of the parameters defined in your parameter server. While you can simply change the values in your launch file, it's cleanest (and most convenient) to set them via command line so you can experiment with many different values without changing the defaults. (Hint: You've actually done this before using Baxter/Sawyer — `electric_gripper` is a parameter value!)

Experiment with changing the downsampling rate parameter. What is the downsampling rate's function, and why is it important? (The comments in the `occupancy_grid_2d.py` file might be helpful here.)

Lastly, experiment with changing the resolution of the map. (Note that the length of each cell isn't explicitly defined in the parameter server but can be calculated from the values there; which parameters do you need to modify to make the cells larger and smaller?) How does your map behave differently? Do you notice any change in error patterns?

[3] Odometry is the use of data from motion sensors to estimate change in position over time. It is used in robotics by some legged or wheeled robots to estimate their position relative to a starting location (wiki).

Checkpoint 2

Get a TA to check your work. Demonstrate the odometry-based localization and the associated map. How does it compare with the map you generated in Lab 6 using the `gmapping` demo? Describe any shortcomings you notice, and hypothesize why they exist. Explain any bottlenecks in the code — what's the slowest part of the computation?
