EECS C106A: Remote Lab 8 - Building Occupancy Grids with STDR Simulator*

Fall 2020

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

---

If you get stuck at any point in the lab you may submit a help request during your lab section at `https://tinyurl.com/106alabs20`. You can check you position on the queue at `https://tinyurl.com/106Fall20LabQueue`.

Note: For all labs this semester you may collaborate with a lab partner but we expect everyone to do every part of the labs themselves. You should work closely with your partner to overcome obstacles in the labs but each member of the team must do the lab themselves.

Contents

1 Getting started with Git **2**
    1.1 Setting up STDR Simulator . . . 2

2 The ROS parameter server **3**

3 Generating & updating the occupancy grid **4**
    3.1 Testing your occupancy grid . . . 4

Introduction

In this lab, we will build and test one of the most useful data structures 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:

*Developed by David Fridovich-Keil and Laura Hallock, Fall 2017. Updated by Valmik Prabhu, Nandita Iyer, Ravi Pandya, and Philipp Wu, Fall 2018. Converted to remote by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).

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

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.

1 Getting started with Git

Our starter code for this lab is a ROS package called `lab8_starter`. It is on Git for you to clone and so that you can easily access any updates we make to the starter code. First, create a new ROS workspace called `lab8`.

```
mkdir -p ~/ros_workspaces/lab8/src
cd ~/ros_workspaces/lab8/src
catkin_init_workspace

cd ~/ros_workspaces/lab8
catkin_make
```

Next, clone the starter code package into the `src` directory of your workspace, and build it.

```
cd ~/ros_workspaces/lab8/src
git clone https://github.com/ucb-ee106/lab8_starter.git
cd ~/ros_workspaces/lab8
catkin_make
source devel/setup.bash
```

We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

1.1 Setting up STDR Simulator

Recall that we introduced you to STDR Simulator in Lab 4. We will again be using it today in Lab 8. First, clone the appropriate packages:

```
cd ~/ros_workspaces/lab8/src
git clone https://github.com/stdr-simulator-ros-pkg/stdr_simulator.git
```

In Lab 4 we had you create a new package and file in order to use the turtlebot teleop keyeboard launch file to control our STDR Sim robot model instead. You can go back to Lab 4 and repeat these steps if you need to or you can find and copy over the package.

```
cp -r ~/ros_workspaces/lab4/src/stdr_teleop ~/ros_workspaces/lab8/src
```

If you choose to copy over the files, be sure to change any references to `lab4_starter` to `lab8_starter` in order for it to work.

Now build and source your workspace (this may take a few minutes)

```
cd ~/ros_workspaces/lab8
catkin_make
source devel/setup.bash
```

2 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 you can specify when launching 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.

Inside your `lab8_starter` package you will find two source files and two launch files 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 (one of them should look very familiar!).

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.

Task 1: 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

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Run `demo.launch` with no errors.
- Explain each parameter you have loaded in `Load Parameters`.

---

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

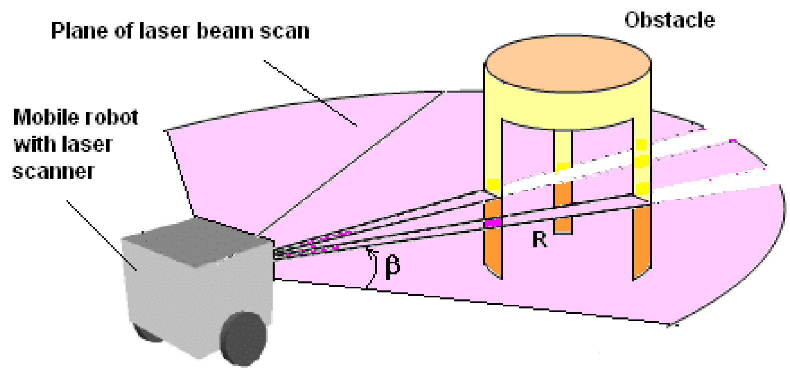

Figure 1: Diagram of a mobile robot with a laser scanner.

3 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.* 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 receives 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 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 `robot0`, while the sensor frame is `robot0_laser_0`.

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.

Task 2: Complete the `SensorCallback` function. When you're done, try running the launch file again, and make sure you don't get any error messages.

3.1 Testing your occupancy grid

Look back at `demo.launch` 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.

Now let's try testing out our occupancy grid! First we need to start up our simulation.

```
roslaunch lab8_starter stdr_maze_env.launch
```

Next run `demo.launch` (there should be no errors)

```
roslaunch lab8_starter demo.launch
```

We then need to start up RViz and our keyboard teleop node

```
roslaunch stdr_launchers rviz.launch
roslaunch stdr_teleop stdr_keyboard.launch
```

Task 3: In RViz, find and add the appropriate topic where your occupancy grid is being published to. You should be able to see a mostly red and blue map being generated in RViz as you drive your robot 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?

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Demonstrate the odometry-based localization and the associated map.
- Compare your map with the one you generated in Lab 4 using the `gmapping` demo.
- Explain any bottlenecks in the code — what's the slowest part of the computation?
- Change a parameter of the launch file from the command line.
