Lab 2: Path Planning

Contents

Submission

Please complete all questions. This assignment is worth 10 marks. Q1 to Q4 are worth 1 mark each. Q5 and Q6 are worth 2 marks each. If your run can be done within 0.2s per iteration (Q7) on the lab computer and with a display showing the real time update of the map, you get 2 marks to get full marks. If you implement either part of Q8, you will get one bonus mark that will be counted to your total score, capped at 100. The bonus is quite little, but it would be a requirement for the project.

Please submit your report **[your matric number].pdf**, together with **lab2.py** with your matric number on **Line 2** as a comment, in a zip file **[your matric number].zip** and submit to **LumiNUS**.

Submission is due **1 week** at the time of the start of your lab. If you start your lab on 28 January 2pm, it is due on 4 February 2pm.

3rd party libraries or high-level functions for navigation, localisation, and mapping are not allowed. Basic libraries like `numpy`, `cv2`, `rospy` and messages are allowed. Please clarify before using, otherwise marks will be voided.

Queries

Email Yan Kai (lai.yankai@u.nus.edu) if you have are unsure of anything, but please cc Prof. Prahlad (elepv@nus.edu.sg).

1. Assessing the Lab Computer

Turn on the computer.

Immediately press and hold `F12` until you see the list of operating system (OS).

Select the item containing Ubuntu and wait.

The password is `!Control1`.

2. Setup for Lab 2

A. Copy Files

Use **Files** (**nautilus**) for this step. If you still have the previous lab's folder, just put **lab2.py** and **lab2_aux.py** from `ws/src/pkg/scripts` of **lab2.zip** into `~/ws/src/pkg/scripts`. If you do not have the previous lab's folder, simply copy over `ws` in **lab2.zip** into `~` (home).

B. Source Files

Then, run the following in the terminal or `gedit ~/.bashrc` the following line in the terminal and add the following line, like previous labs:

```
source ~/ws/devel/setup.bash
```

If you chose to save it into the **.bashrc** file, remember to run `source ~/.bashrc` in the terminal after closing and saving the **Text Editor** (**gedit**).

C. Assign Executable

Always remember to run the executable for new python files:

```
chmod +x ~/ws/src/pkg/scripts/*.py
```

3. Introduction

In this lab you are going to design the A* algorithm and its open list.

A. Hidden and Shown Items

The following describes a list of important items which are hidden or shown. If they are hidden, they are in **lab2_aux.pyc**. If they are shown, they are in **lab2.py**

Hidden	Shown
• `Cell.set_occupancy` – Log odds Binary Bayes model for updating occupancy information (this is just a simple addition or subtraction really). • `PathPlanner` – A path planner implementation using the A* algorithm on the occupancy grid. • `OpenList` (not in lab2.py) – the open list used by the hidden A* implementation. This sorts according to the F-cost, followed by H-cost. Can be optimised further. • `gen_mask` – the algorithm used to generate a circular mask around a cell efficiently. This returns a list of integer indices of neighbouring cells that fall within a radius around (0, 0). • `MotionModel` – Implements the odometry motion model • `LOS` – Implements the general line algorithm • `inverse_sensor_model` – Self explanatory	• `OccupancyGrid` – The occupancy grid data structure and its associated methods. • `OccupancyGrid.update_at_idx` – The occupancy updating procedure. This writes inflation zones efficiently and resolves conflicts between range observations of the same iteration. • `OccupancyGrid.show_map` – Using `cv2` we can draw an image representing the occupancy grid, robot position, goal position and the optimal path quite quickly. • `Distance` – A more robust implementation of the octile distance. Required for reliable open-list sorting.

Some items are hidden because they are answers to the project and previous labs. Shown items are fundamental to the project. The intention of showing these items is to accelerate your learning. So please take a hard look at them.

B. Run Reference Code

The compiled python file contains both the hidden A* and open list implementations. You may only run them to benchmark your code's performance.

In a terminal, first launch **Gazebo**:

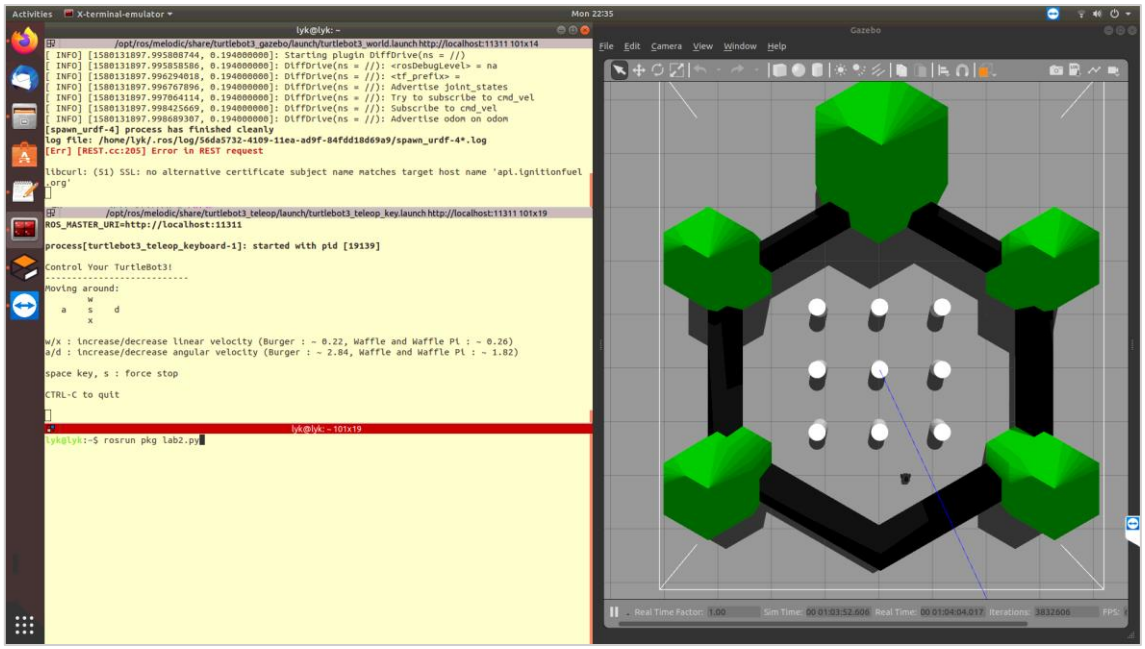
```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Wait for **Gazebo** to finish launching.

Open a new terminal (`Ctrl+Shift+O` if on **Terminator**). Run the following to teleoperate your robot.

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

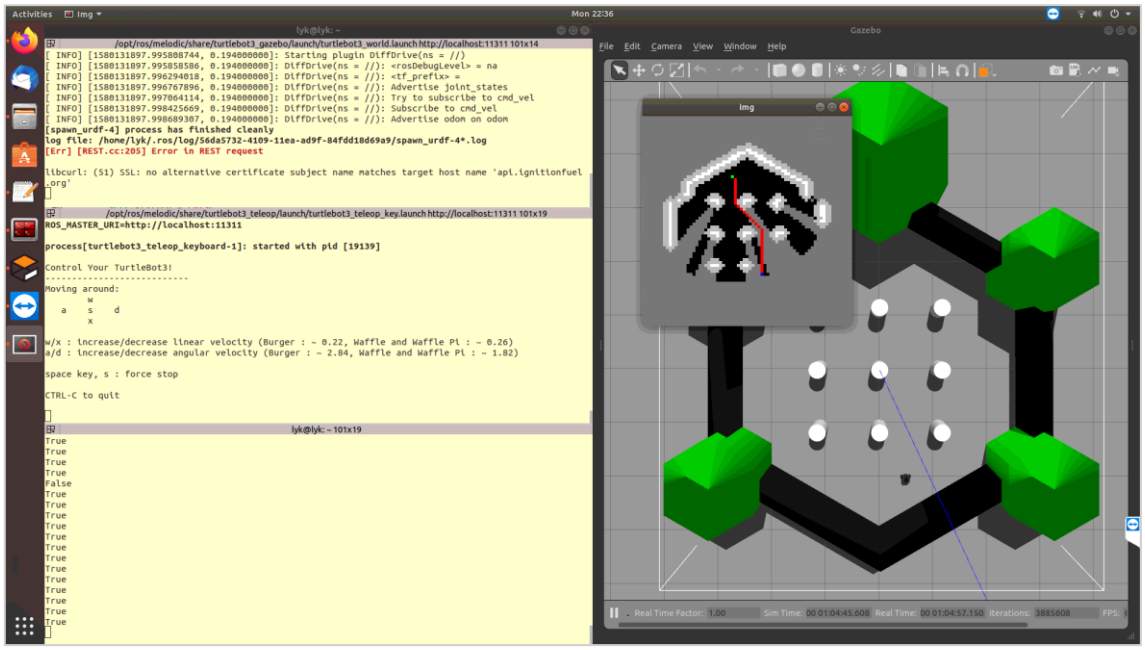
Open a third terminal. You may want to drag and snap your terminals on one side of the screen and **Gazebo** on the other, as such:



On the third terminal, run the script using:

```
rosrun pkg lab2.py
```

You should now see the picture of the occupancy grid and a path (red) between the robot (green dot) to a goal position (blue dot). The white areas are occupied. The black areas are free areas. Both must be sufficiently observed to show up in the image. The light grey areas are inflation zones and will exist depending on the occupied cell that is causing it. The dark grey areas are unknown zones.



At every 0.2s, the path is re-planned with A* based on where the robot is. You can teleoperate the robot to observe that it is the case. The goal position is arbitrary. If it takes more than 0.2s to re-plan and show the image, the terminal will print a 'False'. Otherwise it is 'True'. 5 prints would correspond quite accurately to 1s, provided that not many iterations exceed 0.2s.

The algorithm must be very fast for effective motion estimation and path planning, otherwise, the robot has to move very slowly. State of the art implementations are much faster than this.

When you are done, `Ctrl+C` on all terminals to stop all programs.

4. Understanding the Code

Now let's launch a python shell. In one of the terminals, run the following:

```
python
```

The shell executes statements line by line. We can use this to debug or test some ideas.

Once the shell opens, run the following **in the shell** to import `numpy` methods:

```
from numpy import *
```

A. Computing with Irrational Numbers

One of the major problems with calculating irrational numbers is its accuracy in binary form. It is simply impossible to represent some combination of irrational numbers accurately, for example, those with square root of two.

Let's consider an example. We should expect that $d_0 + d_1 = d$ for the simple arithmetic as shown:

$$d_0 = 10\sqrt{2} + 3$$

$$d_1 = 3\sqrt{2} + 10$$

$$d = 13\sqrt{2} + 13$$

In the shell, we test this:

```
d0 = 10 * sqrt(2) + 3
```

Then,

```
d1 = 3 * sqrt(2) + 10
```

Finally,

```
d = 13 * sqrt(2) + 13
```

Now we equate the results:

```
d0 + d1 == d
```

You should see a `False`.

You can see quite clearly why this is the case if you run:

```
d0 + d1
```

And then compare the floating-point value with

```
d
```

This floating-point approximation in many machine languages is problematic when we want to compute octile distances and costs. The f-cost is the sum of the g-cost and h-cost, which are typically octile distances for an 8-connected path planner. Even if we know that two f-costs are identical, the machine would give us different values. You could see quite clearly if we try to compute the following costs:

First, we compute g_0 and h_0 by combining two lines into one:

```
g0 = 4 * sqrt(2) + 10; h0 = 10 * sqrt(2) + 2
```

Then, g_1 and h_1

```
g1 = 8 * sqrt(2) + 1; h1 = 6 * sqrt(2) + 11
```

We then find f_0 and f_1:

```
f0 = g0 + h0; f1 = g1 + h1
```

Q1. Use the shell to determine the following:

a. **What is the value of f0?**
b. **What is the value of f1?**

B. Relating Irrational Computation to Octile Distances

To remedy the problem, we delay the computation of irrational numbers to the last possible step, and avoid computation with them whenever we want to manipulate calculations involving them.

Now, run the following code:

```
og0 = 4; cg0 = 10; oh0 = 10; ch0 = 2
```

```
og1 = 8; cg1 = 1; oh1 = 6; ch1 = 11
```

```
of0 = og0 + oh0; cf0 = cg0 + ch0
```

```
of1 = og1 + oh1; cf1 = cg1 + ch1
```

When summing up the f-costs, we use the coefficients of the `sqrt(2)` (ordinal steps) and the coefficients of the cardinal steps. Because they are usually small ($\ll 1.8 \times 10^{308}$ for `float64`) whole numbers, there are no precision losses when representing them.

```
f0 = of0 * sqrt(2) + cf0
```

```
f1 = of1 * sqrt(2) + cf1
```

If you now compare their values, they are now identical.

If you compare to those in Question 1, one of the values may be the same as this. What matters is that your program yield identical values for truly identical values, more so than the floating-point inaccuracies that are yielded when calculating one single value. That is because floating point inaccuracies ($\ll 10^{-10}$) are typically negligible if we compare two truly non-identical values, so we do not care if the inaccuracies differ from the true value.

Now, look for the `Distance` class in the **lab2.py**. The motivation behind designing this class is to eliminate the floating-point problems associated with computing the costs of ordinal (diagonal) steps and ordinal steps.

You may want to refer to https://rszalski.github.io/magicmethods/ for a better introduction to Python's class magic methods, but it is not necessary.

Examine the code closely and answer the questions:

Q2. For the Distance class:

a. **Draw a 3x3 grid and overlay 8 arrows from the centre square. Label the arrows "ordinal" or "cardinal" correctly.**
b. **The magic method `__eq__` allows a `Distance` object to be compared to another using an `==` operator. Hence if two `Distance` objects are identical, comparing them using `==` will return a `True`. What are the two properties that are being compared to ensure that they are identical?**
c. **Which magic method implements the addition of `Distance` so that the `.total` property of two `Distance` objects are identical if they are truly identical?**
d. **Which static method calculates the octile distance between two indices?**

C. Cell Class

The occupancy grid is implemented differently as Lab 1. Instead of a single large `numpy` matrix with numbers, It is now a list of list containing `Cell` classes. We use classes so that we can have more flexibility defining their properties and methods. Note, the lecture uses the notation Vertex to generally refer to individual units of graph-like data structures like occupancy grids. To make it easier to write, Vertex is replaced with `Cell`. They mean the same thing.

`reset_for_planner` resets the properties of the cells for re-planning the path. **A re-plan occurs every 0.2s to account for new observations**.

`set_occupancy` uses a hidden log-odds binary implementation based on the observed occupancy state.

`is_planner_free` checks if the cell can be treated as free by a path planner

`set_g_cost` and `set_h_cost` sets the g-cost and h-cost respectively and recalculates the f-cost.

Q3. For the Cell class:

a. **What are the true values (a number or symbol) assigned to the `g_cost`, `h_cost` and `f_cost` properties of all `Cell` objects at the start of planning?**
b. **What are the three characteristics of the cells which would cause `is_planner_free` to return `True`?**

D. OccupancyGrid Class

The `OccupancyGrid` class contains methods that converts real world coordinates (`pos`) to map indices (`idx`), and checks if the supplied indices are in the map. It also contains methods that retrieve cells and to update several cells at once. It is basically an interface to the memorised map.

Examine the code closely and answer the following:

Q4. For the OccupancyGrid class:

a. **Given tuples `(1,4)` and `(1.23, 3.921)`, which is an `idx` and which is a `pos` (both values may not be associated to one another)? Can both be `pos`?**
b. **Given a cell at index `(1, 2)` of the `OccupancyGrid` object `occ_grid`, write the code to return `True` if the cell is occupied or `False` if the cell is not (free or unknown) in one line. Use `idx2cell` and search the `Cell` class for a method that returns the occupancy.**
c. **What is the method that sets the inflation zone whenever there is a change in the state of occupancy of one cell?**
d. **What is the variable in the `for` loop in part c that represents the index of the surrounding, inflated cells?**

5. Implementing the Algorithms

First, we switch from the reference code to the A* and open list you are going to design. The methods where you need to implement A* and the open list are in **lab2.py**, in the classes `Astar` (**lines 392 to 474**) and `OpenList` (**lines 476 to 525**). Replace **line 585** (`PathPlanner = lab2_aux.Astar`) with the following to switch to the class `Astar` you are going to design:

```
PathPlanner = Astar
```

The skeleton code is given for both the A* and open list implementations. You may use the skeleton code and comments starting with an exclamation mark (`# !`) to locate the positions where work is needed.

Otherwise, you can just implement your own A* path planner but please use the given `OccupancyGrid`, `Cell` and `Distance` classes and their methods.

Q5. Design `Astar` class from lines 392 to 474 (line numbers are before editing). The method to design is `get_path`

Q6. Design `OpenList` class from lines 476 to 525 (line numbers are before editing). The methods to design are `add` and `remove`

Q7. With your `Astar` and `OpenList` class, run lab2.py within 0.2s per iteration (all printed statements on the terminal start with True) on the lab computer, with an image / player showing the map update in real time.

You get 1 bonus mark for doing **either of** the following:

Q8. (Optional) Do either of the following:

a. **Implement your own `Cell.set_occupancy` method, `subscribe_wheels` method, `gen_mask` method, `OdometryMM` class, `LOS` class, `inverse_sensor_model` method, and set your own `MAX_RNG`.**
b. **Implement Jump Point Search by using a `JPS` class and setting `PathPlanner = JPS`. Use some timing (profiling) tool to show that it is faster than A*.**

6. Remove Files

We have now come to the end. We must tidy up the computer for others to use it.

Backup any files you want to keep for this lab.

Stop all running programs on the terminals using `Ctrl+C` or `Ctrl+Z` on every terminal.

Then run the following on the terminal to remove the folder `ws` with all its contents:

```
cd ~
rm -rf ~/ws
```