Lab 3: Post-Process & Motion Control

Contents

Submission

Optional. No submission required. This lab provides a platform to post process A* paths and implement your own motion controller. This is the early access version, Lab3. Please use Lab 3 (released later) .sh files to fulfil the requirements for the project.

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). Come to the lab as well. The lab will be opened on Tuesday, Wednesdays and Thursdays for W6 and Recess Weeks, 2 to 5pm. Please follow the timings and rules on **LumiNUS**.

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 3

A. Copy Files to Existing Workspace

Use your previous Lab 2 or Lab 1 folders.

Extract the **lab3** folder in **Lab3.zip** from **LumiNUS**.

Copy **lab3.sh** and **world.sh** from the **lab3** folder into your `~/ws` folder.

Copy `lab3/src/pkg/launch` folder into the folder `~/ws/src/pkg` folder

Copy `lab3/src/pkg/worlds` folder into the folder `~/ws/src/pkg` folder

Copy all the files in the `lab3/src/pkg/scripts` folder into the `~/ws/src/pkg/scripts` folder

Grant permissions to all files by running the following on the **terminal**:

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

Source the pkg if you haven't done so

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

B. Create New Workspace

Open **Files** (**nautilus**) and navigate to the **Home** folder.

Delete any **ws** folder from the **Home** folder.

Extract the **lab3** folder in **Lab3.zip** from **LumiNUS** into the **Home** folder.

Rename **lab3** folder as **ws**.

On the terminal, run:

```
cd ~/ws/src
catkin_init_workspace
cd ~/ws
catkin_make
source ~/ws/devel/setup.bash
chmod +x ~/ws/*.sh
chmod +x ~/ws/src/pkg/scripts/*.py
```

3. Introduction

This is the early access version for you guys to test your post processing code and motion model code. There will not be any pseudocode now. Please also use the `.sh` files in Lab3 (not this Lab3) for your project.

A. Launch Custom World Maps

In a terminal, run the shell script, **world.sh**, to open up a **Gazebo** world.

```
cd ~/ws
./world.sh
```

The code should give you the same result as using `turtlebot3_world` in the earlier labs. But this time, you can set the world and spawn coordinate parameters. It also launches a launch file, **world.launch**, located in the `~/ws/src/pkg/launch` folder.

Additional parameters to set the world boundaries and cell size will be available in Lab 3.

Available worlds are in the `~/ws/src/pkg/worlds` folder.

You should use this to launch your custom worlds while debugging, so you do not have to use the other shell script to keep opening or closing **Gazebo** while you test your code.

B. Launch Move and Main nodes

On a new terminal, run the shell script **lab3.sh**, to launch the **main** and **move** nodes. Make sure the gazebo launched from the previous section is still running. The robot will start running to 4 different goal positions

```
cd ~/ws
./lab3.sh
```

You should be able to see a post-processed path on the map. Post processing is not done on segments of the original A* path that have inflated cells. This is because the A* treats occupied cells as extremely high, but not infinite, g-cost cells, and inflation cells as extremely high g-cost cells which are significantly cheaper to access than occupied cells.

This is meant to avoid algorithmic problems when your map becomes inaccurate due to the motion model and a goal coordinate lies within an inflated or occupied zone when it shouldn't be.

The shell script sets the necessary environment variables and launches a **lab3.launch** file in the `~/ws/src/pkg/launch` folder. The launch file launches both the **move** and **main** nodes.

If you examine the **lab3.launch** file, the line that should open Gazebo using **world.launch** is commented out (using XML commenting format, between `<!--` and `-->`). This is meant to run only the python scripts, but not **Gazebo**, for debugging purposes. For the project, you would need to uncomment that line to launch everything simultaneously at one go.

C. Hidden Items

The following describes a list of important items which are hidden. If they are hidden, they are in **lab3_aux.pyc**. A warning – if your project 1 code looks vaguely like the one in the file, the section where it is plagiarised will automatically be zero, with no warning. The code is provided to enhance your learning, not for you to copy.

Hidden
• `OccupancyGrid` – Occupancy grid implementation and its methods. Lower level implementation than previous labs • `GeneralLOS` – General LOS. Slightly lower level implementation. • `GeneralIntLOS` – General LOS, optimised for integer calculations (i.e. between centre of cells). • `Astar` – A* planner, with integrated open list and lower level implementation. • `post_process` – The method to post process A* paths, which return a smoother path and its turning points. These turning points are passed to the motion controller. • `OdometryMM` – The odometry motion model from the lecture slides integrated (sensor fusion) with IMU information. The implementation uses a simple weighted average. It performs far more stably than the ones in Labs 1 and 2. • `get_v_w` – A rudimentary motion controller with no PID implementation. It's not so advanced.

D. Videos of Demonstration in LumiNUS Multimedia

You can find related videos in the LumiNUS Multimedia section. The section below documents their details.

4. Noteworthy Features

A. Lower level Implementation

The previous lab codes are very inefficient, because they are designed to be readable. The hidden implementations are quite low level. Here are some tips to speed up your code:

- Use python lists `[]`, or lists of lists `[[]]`, instead of `numpy` arrays. `numpy` arrays should only be used if you need to use large, batch operations, which you do not exactly need for Project 1
- Use classes only when you only need to design custom methods on custom data. As much as possible, use low level objects like lists or tuples to store information.
- For large, reference or cached information, like `DEG2RAD`, use lists instead of tuples. That is because reading from lists are faster than reading from tuples. List access is faster because they are pointers to memory. Tuples store everything in one memory address and further operations must be done to parse their values and access the individual elements.
- For large, dynamically generated information, like A* paths, it is preferable to use lists of tuples `[()]`. Tuples are much faster to create than lists. However, we use lists to store the tuples because it makes accessing these tuples easier for subsequent operations. This is like a balance between speeds.
- Cache functions that are frequently called. For example, using `update_at_idx` in **lab3.py**.
- Pass position `(x, y)` or index `(i, j)` components individually, i.e. instead of passing `a_func(idx)`, pass `a_func(i, j)`. Instead of passing whole tuples, pass its individual components into functions instead, so we can avoid parsing their values, which are slow.
- Instead of assigning `a, b = list_tuple_or_method`, use `ta = list_tuple_or_method; a = ta[0]; b = ta[1]`. This avoids further checks on the former's operation.
- Strategically create data structures. Avoid putting information that is usually accessed side by side in different places. For example, when sorting in the open list, we may be tempted to access two different lists containing the f-cost or h-cost of every cell using the cell's index. Instead, you should just add the cell's f-cost and h-cost (without reference to the data structures containing them) into the open-list, along with the index.
- Avoid generating lists or tuples for small information. If you use a class, you should store this information in the class instance's properties using `self`. You should notice this when using `OdometryMM`, where the robot pose information is accessed from the instance using `.x`, `.y` and `.o`, instead of being returned as tuples.
- See the forum.

B. Environment Variables

The **Turtlebot** model is no longer required to be specified. Instead, the world (`WORLD`), starting positions (`X_POS` and `Y_POS`), goal positions (`GOALS`), cell size (`CELL_SIZE`) and occupancy grid boundaries (`MIN_POS` and `MAX_POS`) should be specified. The syntax is in the **lab3.sh** file.

You should have identical commands in your project.

C. Starting Position using Robot True Position

The robot true pose is used only to initialise the robot program, so debugging becomes easier (i.e. you can start anywhere on the map, and the map is seldom wrong). It should not be used anywhere else in the code. In the project, you should use the environment variables (not available in the current code yet) to set the robot initial pose.

D. Gazebo Restart and Clean Map with IMU Information

The IMU information is integrated into the odometry motion model using a simple weighted average. A weighted average of p and q is given by the following:

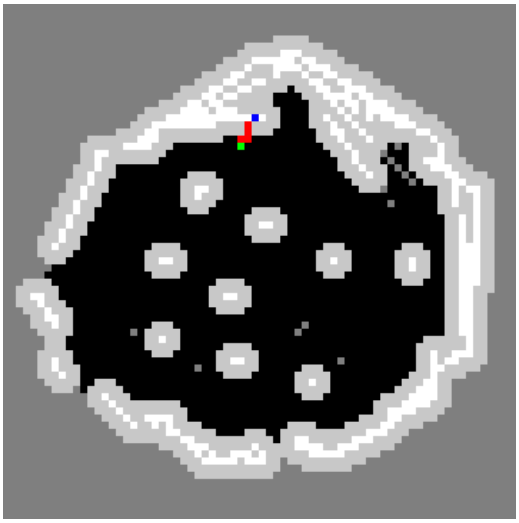
$$\Sigma = ap + bq, \qquad a + b = 1, \qquad 0 < \{a, b\} < 1$$

The IMU information is critical for improving motion estimation, so the map does not 'shake' a lot like previous labs. However, it alone is not useful because it is very noisy, and the signal to noise ratio for the linear acceleration is very low for a robot that moves slowly. So, you should fuse it with the wheel encoder information to obtain a more reliable motion estimation. You can use Kalman filters, but you can also do a rudimentary approach by tuning the weighted average.

The IMU topic will stop publishing if you press `Ctrl+R` in Gazebo. You should use `Ctrl+Shift+R` in Gazebo to reset instead. Occasionally, the map will be initialised wrongly when you run **lab3.sh**. Just `Ctrl+C` **lab3.sh**, `Ctrl+Shift+R` in Gazebo, and then run **lab3.sh** again.

You can see the demonstration with and without IMU videos in the **LumiNUS Multimedia** section.

E. A* Accesses Inflated and Occupied Cells



After running around the map, the robot position will become inaccurate. Hence, the original goal coordinates supplied may come to lie in an inflated or occupied cell (see image). If the A* avoids searching the occupied or inflated positions, the scripts will fail fatally. Hence, it is important to be able to accept a goal position in these zones.

A very high g-cost can be associated to occupied cells, and a significantly lower g-cost can be associated to inflated cells. However, both g-costs must be extremely large compared to the g-costs of unknown and open areas.

You may end up hitting an obstacle but that is *far less likely to happen* than the algorithm failing because A* is unable to find a path in an inflated zone. You can see this occurring in **NoIMU&GoalInf.mp4** in the **LumiNUS Multimedia** section.

F. Differences in LOS Methods

The implemented LOS classes do not have `.calculate()` methods. Instead you have to run `.init(si, sj, ei, ej)` to initialise the LOS instance with start index `(si, sj)` and end index `(ei, ej)`.

Then use `.next()` to retrieve the next index. Every time `.next()` is called, the next index in the initialised direction will be set in the instances' `.i` and `.j` properties. *Nothing* is returned from the `next()` method. Refer to **lab3.py lines 168 to 172**.

G. Post Process Method

The post process method accepts the A* path, inflation information and a LOS class instance as arguments. It returns a smoothened path and its corresponding turning points.

When post processing, cells in the A* path should be checked to see if there are LOS to each other. A cell has LOS to another cell if there are no inflated cells between them. The objective is to create a smoother path with less turning points for the robot to follow.

It accepts only inflation information of every cell to check for LOS, instead of additional occupancy information. This is because an occupied cell is guaranteed to be inflated.

The LOS class that is passed into the post processed method is an integer optimisation of the General LOS algorithm, `GeneralIntLOS`. The integer optimisation works only if the line starts and ends in the centre of cells. The original `GeneralLOS` works regardless of where the lines begin or end.

Take note that because an A* path may lie in an inflated cell, *you should only process non-inflated cells*. A* paths on inflated cells should not be processed as the LOS can only be checked on non-inflated cells.

The post process method is not currently well implemented, as it preserves a whole A* path segment (between turning points) should any one of the segment's cells is inflated. This will be fixed in Lab 3.

However, the way it processes the segments in free or unknown cells should be check against your code. It checks in both directions, and this method corresponds to method **1b** in the post processing lecture.

No trajectories are generated using splines, and no artificial potential fields are used.

H. Redo A* and Post Processing if Post Processed Path Lies on Inflation

Do not redo the two if the path lies on only unknown or free cells. Recalculation takes up computational resources and may overwrite your generated trajectory (e.g. splines). If you do so, the robot may move very slowly, as points are always regenerated, and your script may always defer to the closest point (hence small error and low speed).

I. Motion Controller

A rudimentary, ad-hoc P-controller with some enhancements is used to guide the robot. It takes the turning points generated from the post processing method as waypoints to move the robot to, with speed and acceleration limits. It can also move backwards if the objective is behind the robot.

The function `get_v_w` in the **move** node (**lab3_move.py**) returns the linear forward velocity `v` and angular velocity `w` which would be published to the robot to move it via the **cmd_vel** topic.

J. Main and Move Nodes

The **main** node is in **lab3.py**. The **move** node is in **lab3_move.py.**

The **main** node does the map update, A* and post process methods. It iterates every 0.2s. Each iteration should last around 0.07s before waiting for the next.

The move node does the odometry and motion control. It iterates every 0.05s. Each iteration should last around 0.03s before waiting for the next.

All timings are based on the lab computers.

K. Main and Move Topics

The main node, in **lab3.py**, publishes the **main** topic. The **move** node, in **lab3_move.py**, publishes the **move** topic. Both topics contain `Float64MultiArray` messages for your reference, so you can modify your messages easily. It is essentially a list. Use `rqt`, or `rostopic echo main` or `rostopic echo move` in the terminal *while the scripts are running* to examine the topics

The **main** topic contains information about the operating mode and a target coordinate for the robot to move to. The operating mode is by default `0`, until the robot reaches all goals and sets it to `1`, which will request the **move** node to shut down.

The **move** topic contains information about the robot pose, as well as an indicator which would be `0` if it is running.

5. How to Use

Examine all the information above as well as comments in the scripts.

The codes provide a platform for you to test your post processing method and motion controller, as it is understandable that programming should not be a major component for assessment.

In the **main** node (**lab3.py**), replace the `post_process` method on **line 188** with your own post processing method.

In the **move** node (**lab3_move.py**), replace **line 150** with your own motion controller, using the positional error `err_pos` and positional error `err_ang`.

You can try to replace the **lab3_aux** module's methods with your own for Project 1.

Do not plagiarise the **lab3_aux.pyc** code. You should use your own implementation. This is a tool to accelerate your learning. Its quite easy to smell plagiarism.