CS3027 Continuous Assessment

Konrad Dryja

November 15, 2017

Abstract

Results of my work throughout CS3027 Robotics during first semester 2017-2018 academic year.

1 Preface

1.1 Preliminaries

To be able to run all of nodes and packages, several adjustments to paths needs to be made, mainly path wise, so the exact components are visible to each other. Let's start with the launch file located in launch_files/assessment.launch – it's going to be our starting point when running all the nodes written by me. Make sure to modify paths of map_server, stage and add_gaussian_noise so they actually point to the actual location of the packages within your ROS environment. We will also have to modify our map files to utilise the provided files – change "image:" entry within launch_files/map.yaml to reflect location of your map.png. I have also made use of several extra pip packages which were tremendous help with dealing with concepts such as Gaussian Probability or array manipulations. Make sure that you have them installed and ready:

- Scipy (tested on version 1.0.0 / 25)
- Numpy (tested on version 1.13.3)

If not, be sure to install them using pip. If operating with multiple Python versions, make sure to use the correct version (i.e. pip2 / pip2.7 if using environment with main Python3). Environment used for development:

- Arch Linux / Ubuntu 16.04
- ROS Kinetic
- Python 2.7.14
- GCC 7.2.0

1.2 Modifications to the provided files

It important to mention that I have slightly adjusted map.world file to slightly slow down simulation speed (from 50ms to 100ms) to match real time which ultimately avoids odometry readings discrepancies. I have also altered launch file by removing dummy_localization (as it is no longer needed and has been completely replaced by my own nodes) and appending real_robot_pose with rviz_info which are uniform to use with every developed component.

1.3 Project Overview

Project consists of two ROS packages and one external file:

- assessment – contains majority of the code produced for the purposes of CS3027.
- add_gaussian_noice – provided by the course coordinator containing laser scan adjusted by random noise.
- pixeldraw.py (located in assessment/resources) – pre-processing script to divide provided map onto cells which are used in path planning (see pathplanning.py module for further explanation).

And finally, beforementioned assessment package contains following files (later referred as nodes) – all located in the scripts folder:

- real_robot_pose.py
- rviz_info.py
- pathplanning.py
- drive.py
- localization.py
- image_processing.py

2 Design Choices

2.1 Map representation

To begin this paragraph I wanted to briefly go through the concepts and ideas behind path-planning algorithms which could be utilised in terms of the course. Throughout my research on actual map representation I had looked into various methods such as:

- Occupancy grid
- Exact cell decomposition
- Approximate cell decomposition
- Potential field model

Ultimately I have settled for the approximate cell decomposition. One of the biggest turning points for me was the ease of implementation through a simple recursive call on 4 different areas which later could be easily converted to say a map, or graph representation of obtained cells. Due to the performance issues, speed and fact that I only have to perform decomposition only once (since I can assume that the map won't change) I have decided to do it only once and afterwards import the results into my actual ROS node through pickle python library (by compressing the created Graph structure after decomposition).

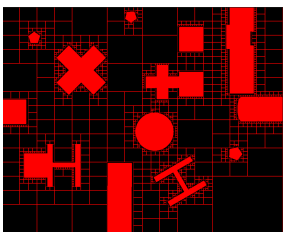

Figure 1: Representation of decomposed cells

Figure 1 represents how the map looks like post-decomposition. Every cell represents area where the robot can move freely without any collisions or issues. "Robby" can also travel to any of the direct neighbouring cells as it wishes. Everything is based on the recursive call for which pseudo-code is included below:

Algorithm 1 Adaptive cell decomposition.

```
1: procedure DECOMPOSE(array, depth)
2:     if no obstacle in array then
3:         Mark array as cell
4:         return
5:     else
6:         DECOMPOSE(top left of array)
7:         DECOMPOSE(top right of array)
8:         DECOMPOSE(bottom left of array)
9:         DECOMPOSE(bottom right of array)
10:    end if
11: end procedure
```

Nodes and files including logic for path-planning: pathplanning.py, resources/pixeldraw.py

2.2 Pathplanning

Now, I should receive approximately 500 discrete cells – which is another advantage of the method over, say, occupancy grid where I would need to consider 1000*800 pixels – those values quickly add up and it is important to reduce run-time as it may be crucial when the robot needs to act quickly.

I have utilised the graph and nodes structures that I used for the CS2521 continuous assessment which allows me to easily store each cell, its neighbours and distances to those neighbours. Afterwards implementing any of the path-planning algorithms becomes trivial since I only needed to code a simple A* using Euclidean distance as a heuristic function. All the operations are performed on the occupancy grid array (reshaped from 1d to 2d – 800x1000) thanks to the implementation of functions allowing me to easily swap around Map Frame coordinates (e.g x=2.5, y=-3.5) to Occupancy Grid (e.g. row=253, column=575).

Last, but not least, to obtain the shortest path overall (given all goals) I had to implement a brute-force solution in non-deterministic polynomial time by getting every available permutations of those lists, getting shortest path out of all of those permutations and publishing the path to driving node.

2.3 Driving

As opposed to my friends who were focusing on proportional controllers and driving based on the distance remaining I have taken a slightly different stance which is driving based on the time elapsed. Pseudo-code below represents the through process and algorithm flow when running the node:

Algorithm 2 Driving module pseudo-code.

```
1: while time elapsed * speed <= distance do
2:     keep moving forward
3: end while
```

One of the issues that I have encountered is the fact that sometimes due to the computing inaccuracies the robot ends up slightly beyond the requested point. I believe it may be caused with the CPU overload and trying to catch up with ROS simulation. The problem probably could be avoided by instead of relying on time, I would base it on distance travelled – both approaches have their drawbacks. Pseudo-code above is easily applied to the list of destinations that I have to visit obtained from path-planning module so the implementation becomes trivial. One thing to note is that due to my laptop's capabilities (where a VM performed even worse) I had to decide against using my own localisation solution to help the robot localise itself within the environment, instead I am utilising /base_pose_ground_truth (while still having odom errors enabled which are corrected as-you-go).

2.4 Localisation

That is the part that I am the most proud of. I have managed to fully implement Monte Carlo Localisation as a node in my ROS project. It was probably the most difficult project and I had to consider plenty of approaches to the problem. The biggest issue wasn't choice of localisation method – it was the actual act of perception and motion update that caused me the most problems, especially the former as it was exceptionally computational dependant constantly taking massive chunks of my CPU power.

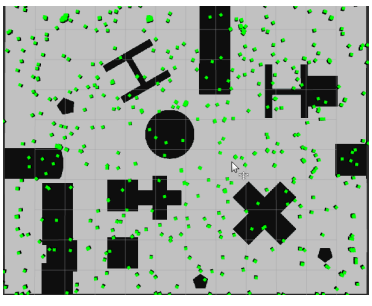

Figure 2: Particles as displayed in Rviz

To start off, I'm placing 800 random particles across the map (as shown on figure 2) – each of them with three properties: X, Y and Theta. Now I needed a way to tell how likely it is for a particle to appear in a given position.

I have considered two methods of performing perception update at all – use either Beam Sensor Model or Likelihood Field Model. Unfortunately I didn't quite comprehend the latter so I focused on the beam sensor model which in principle is trivial – simply track every pixel that laser beam covers and check occupancy grid to see whether it is obstacle or not. If it is, return the expected distance – and keep repeating till you reach the maximum distance which is 3 meters in our case. Once I have obtained my expected reading I referred to the actual – obtained from the noisy scanner. Thanks to the access to the add_gaussian_noise I read out the properties such as standard deviation which I plugged into SciPy's Probability Density Function so I will get the proportional probability of how likely the reading is.

Now robot casts 30 different laser proximity beams. Due to the precious computation time I had to divide those by 6 leaving me with 5 of them – which is more than enough in our environment. By adding all of those probabilities I got a total probability of given particle being at a given point of the map. So now I've got all my particles and their probabilities – all is left is sampling. Simply pick 500 particles based on the weighted probability and add them to the next iterations. Remaining 300 are randomised again to allow recovery from errors.

As when it comes to clustering of the particles, the location obtained from my localisation is always based on the particle with the highest probability. Both the actual position and and believed position are printed to the console to check how accurate my reading are and for debugging purposes.