[CS-475] Assignment 5: Fast-SLAM

Michael Maravgakis
maravgakis@csd.uoc.gr

Release date: 12/04/2022
Deadline: 05/05/2022

1 Overview

In the last two assignments, you've implemented a particle filter for localization and mapping with known poses. The latter, was implemented by utilizing the ground truth pose of the robot and in the particle filter you where provided with a height map of the workspace. The goal of the map was to provide you with a metric for calculating the weight of each particle in order to be able to separate the most "popular" particles from the rest.

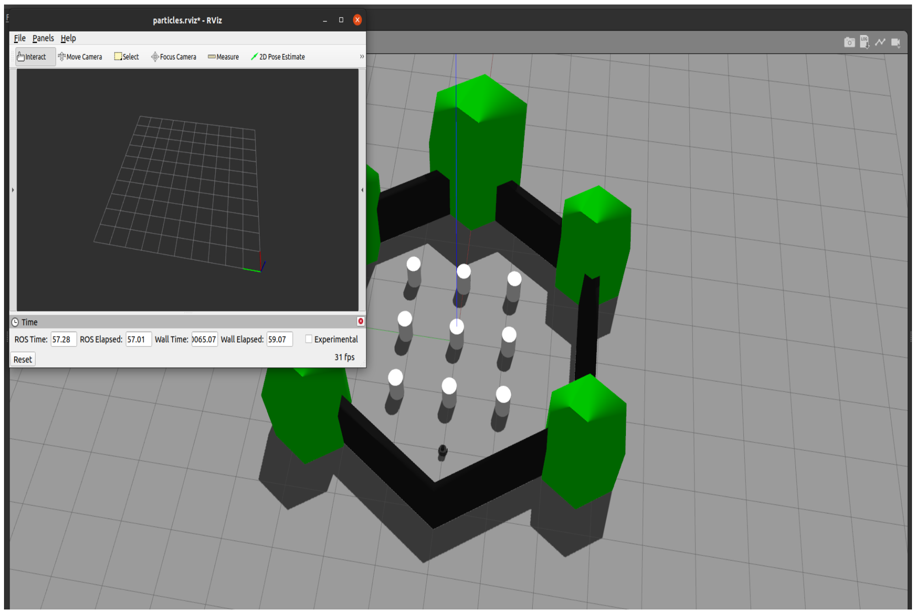
In real applications having either a map or the exact pose of the robot is a luxury and quite often is not the case. Suppose you just have a robot with some sensors and you want the robot to navigate autonomously in an unknown environment. The first task you need to figure out is how to place the robot inside a map that you don't even know its structure. That's where you need the Simultaneously Localization And Mapping (SLAM) algorithm. There are numerous SLAM implementations but in this assignment we are going to use the Fast-SLAM.

2 Installation

In your *.zip* file there is a package (*assign5/*) that you will need to copy to your catkin workspace. Then compile your workspace: `$ catkin_make`
The environment is much similar as in assignment 4, you can run the simulation by using:
`$ roslaunch assign5 burger.launch`
In case you can't see the new package, update the ros package manager:
`$ rospack profile` The first line of your node should be the following:

1. Ubuntu 18.04 + ROS melodic: `#! /usr/bin/env python`
2. Ubuntu 20.04 + ROS noetic: `#! /usr/bin/env python3`

You will need to implement a ROS node(*src/slam.py*) for the FastSLAM algorithm. You can reuse anything you've implemented so far. If you've completed the installation, you should see the following after you use `roslaunch` (RViz and gazebo):



ROS Time: 57.28 ROS Elapsed: 57.01 Wall Time:)065.07 Wall Elapsed: 59.07

Figure 1: Environment

3 Implementation[1]

SLAM is a chicken-egg problem meaning that in order to localize the robot you need a map and in order to create a map you need the pose of the robot. FastSLAM is a modified combination of your previous 2 assignments, namely particle filter and mapping. The idea is simple, you create the particle filter and you assign a map to **each particle** and by using the belief of the map you can calculate the weights and re-sample. Your implementation will be to reform the particle filter algorithm from assignment 3 and add mapping with known poses to each particle. This time the known pose is the belief of each particle's position. The fast slam algorithm in pseudo-code is the following:

[1]Read the whole assignment before beginning your implementation

Algorithm 1 FastSLAM algorithm

```
1: function FastSLAM($X_{t-1}, u_t, z_t$)
2:     $S \leftarrow \emptyset$
3:     for Every particle p in N do
4:         $x_t^{[k]} \leftarrow MotionUpdate(u_t, x_{t-1}^{[k]})$
5:         $w_t^{[k]} \leftarrow SensorUpdate(z_t, x_t^{[k]})$
6:         $m_t^{[k]} \leftarrow UpdateOccupancyGrid(z_t, x_t^{[k]}, m_{t-1}^{[k]})$
7:         $S \leftarrow X_t$
8:     end for
9:     LowVarianceResample(S)
10: end function
```

3.1 Initialization

You can assume that you know the size of the workspace beforehand(6x6). This information comes handy when you are initializing the size of the occupancy grid and also initialize the poses of the particles (x,y,yaw). This information makes the mapping problem static, i.e. you have one map fixed size and you want to fill it along the way. In case you don't even have that information you will need to dynamically allocate space and expand the occupancy grid as you go.

Do not get confused by the robot's position inside the grid in RViz. The algorithm will converge eventually at some point but every time you run the algorithm it will be on a different point (with respect to origin). The only thing that you need to care about is the particle's estimation with respect to the created map, meaning that translation and rotation of the whole occupancy grid are irrelevant. You can see figure 2 where this are the maps created by 3 particles but all of them are correct because the particle has the same position with respect to the local map.

At the initialization step, the map is defined as a 10x10(m) occupancy grid but feel free to change this if you find something more suitable. You will need to initialize the particles just like assignment 3 but this time you can assume that all the particles are somewhere in the center of the map. This step is crucial because if a particle is initialized at the edge it will not be able to create a map with measurement that fall outside of the boundary box.

3.2 Motion Update

The input for this function is the displacement on x/y-axis and how much the yaw (orientation) of the robot is changed. Do not forget that these measurements are with respect to the origin (assume GPS). In your code these variables are formalized as `dx, dy & dyaw` accordingly. In the motion update function you should update the position of each particle according to your control input. You will need to write a transformation that does the following: Figures out

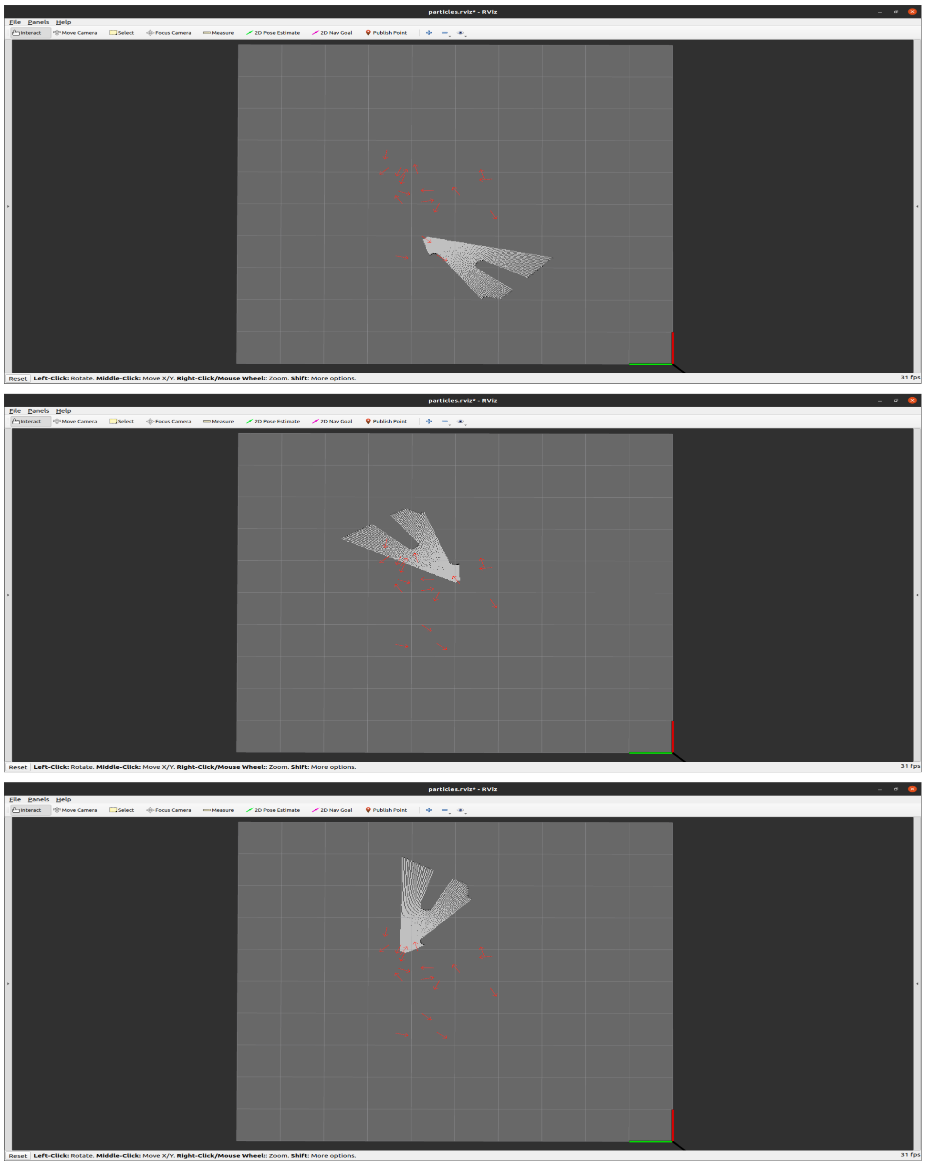


Figure 2: Results

by how much the actual robot has moved in the direction that it is facing and updates all the particles with that information.

Since the displacement is given to you with error you do not want to simply update the all particles by `di` where i is x, y, or yaw. You should rather update them by: $di \pm random(-error, +error)$. What this does is takes into account the erroneous measurement and tries to apply the idea that *"the robot didn't move by exactly this value (di), but something close to that"*. If you publish the particles after this step, you should see all particles wiggle a bit. Try to move the robot around and check if all the particles are following the path of the robot.

3.3 Update Occupancy Grid

Although this step comes after the weight calculation, I suggest to implement this first because you will actually need this in order to calculate weights and also you can check if this step is done correctly. This part is actually pretty straight-forward. You re-implement your previous assignment (Mapping with known poses) for every particle. You consider each particle's pose to be the real one and you raytrace the beam. You loop your beam from minimum scan range to your measurement's range, for every angle and add the log free probability to every cell except the measurement's one where you add the log occupied.

Once you've implemented this part, you can check the robustness of it by using only one particle and moving the robot around. Toggle off the noise (set global variable "ToggleNoise =0") and you should observe the perfect (almost) mapping. Now add one more particle and visualize its map, if the map is built up such that the new particle is the actual robot you are good to go. (Remember each time you can only view one map). The expected result at this point is something like figure 2.

3.4 Sensor Update

This function will update the weights for each particle given a scan measurement (z_t). This is a bit tricky because you will need to take into account computational power and time required for this step. The way to compute the weights is by comparing the already obtained map of each particle with the given measurement.

$$w_t^{[i]} = \frac{1}{|d - z_t| + 1} \tag{1}$$

where $w_t^{[i]}$ is the weight of particle i in time t, d is the calculated distances from the particle's map and z_t is the current scan measurement. z_t contains range measurements on all direction (0-360) so what you need to do is calculate the expected measurements by tracking down a hypothetical scan from the particle's new position and comparing to the real ones. More specifically:

$$|d - z_t| = \sqrt{(d_0 - z_0)^2 + (d_1 - z_1)^2 + ... + (d_{360} - z_{360})^2} \tag{2}$$

In order to compute the d vector you will need to assume that any map cell with a value greater than a certain threshold is occupied. For example, d_0 is the distance between the particle and the first occupied cell in the direction that the particle is facing. d_{30} will be computed using the same process but with the first occupied cell in the direction + 30 degrees. (Do not forget to normalize the weights to 1)

Toggle the noise ON and the navigate the robot around and print the weights, you should see them differentiate. High weight means that the update step where you add a random value to your measurement approximates better the real displacement.

3.5 Low variance re-sampling

Low variance resampling is the process of picking up the good particles and eliminate the bad ones. By good and bad I mean large and small weight. You can find the algorithm at *p.110, table 4.4 Probabilistic robotics.* You will need to reformulate the algorithm a bit to match the python numbering principals (starting counting from 0 and not 1)

4 Results

This implementation is a bit frustrating so don't worry if you don't have the optimal map or the algorithm doesn't converge. Try to figure out if you are getting the expected results at each individual step. If you've implement the algorithm correctly the result should be something like the following:

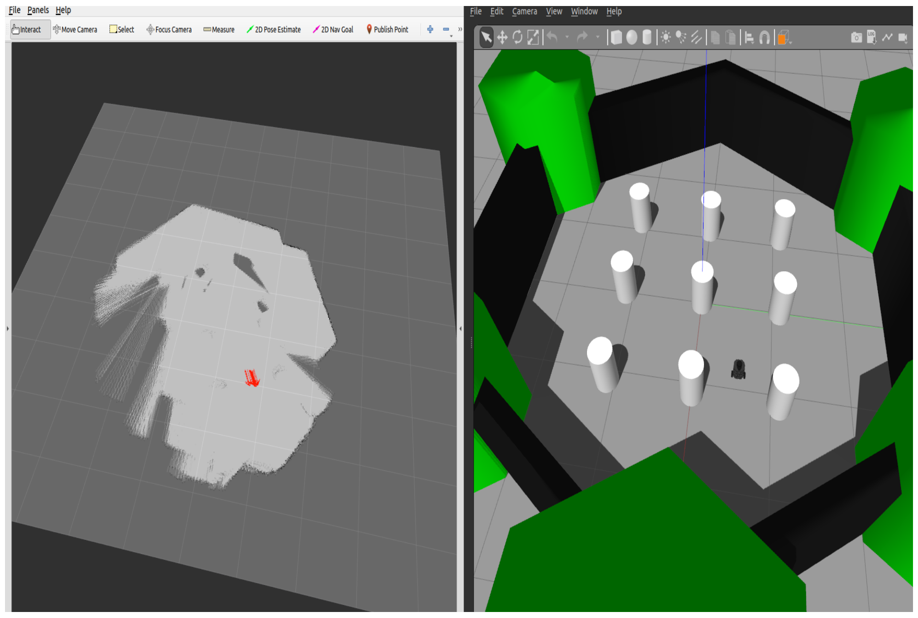


Figure 3: Final result (not the full map

Notice that the cylinders are not appeared inside the created map. I believe this happens due to the error. When a particle gets a scan measurement and it doesn't know its exact position, since the mapping is happening with respect to the estimates pose, it overlays small obstacles.

5 Tips

1. If you use the full scan measurement the algorithm will be painfully slow (if it runs at all). You can subsample the scan measurements and instead of taking 0-360 take for example -30 to 30 degrees measurements and discard the rest. This will greatly help with the computation time that is needed to be real-time.
2. Adjust map_resolution to a value that makes the algorithm run real time at your computer.
3. Visualize and debug each individual step
4. Publish the map of the highest weight particle each time
5. The template I gave you is just a sketch of my implementation. You can re-write the whole code if you want from scratch. If so, keep it structured and commented so it would be easy to read.

6 Submission

Sent your node (`slam.py`) attached via email at: **maravgakis@csd.uoc.gr** with subject `"[CS-475] Assignment 5 submission"` Don't forget to mention your name and registration number. The deadline is at **05/05/2022 23:59**