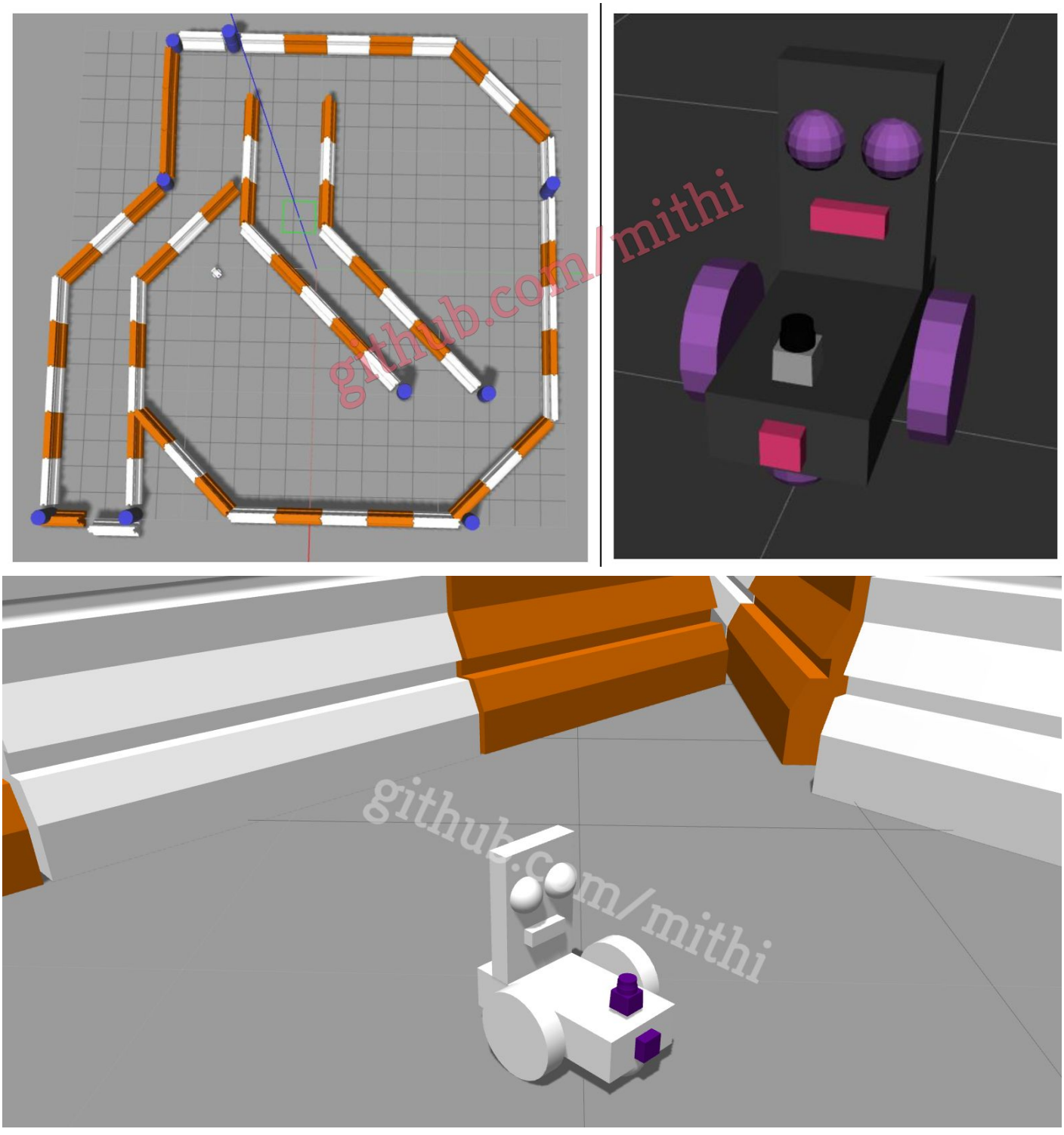
Robotics Nanodegree Term 2 Project 2 - version 2 - Localization

Mithi Sevilla | Jan 8, 2018 | medium.com/@mithi | github.com/mithi

Abstract

In this project, I used the given starting files to build a custom robot model in a simulation environment - Gazebo and RViz, in ROS. This is a differential drive robot with a camera and a lidar sensor. I used the Adaptive Monte-Carlo Localization or AMCL package to localize the robot inside the map. To achieve good results, I tuned some parameters for my robot to achieve the robot's desired position and orientation.



Introduction

The goal of this project is to learn and have hands-on experience localizing a robot in a simulated ROS environment using the AMCL package. AMCL outputs pose estimates after taking a laser-based map, laser scans and transform messages. It initializes a particle filter according to the parameters provided.

Particle filtering works as follows: random guesses of the robot's state called *particles* are generated and upon observation of the environment, it discards particles inconsistent with the aforementioned observations. It replaces the inconsistent particles with the consistent ones each iterations and in the end hopefully most particles converge where the robot actually is. *AMCL builts on top of this base algorithm in hopes of increasing the efficiency of filtering by adapting the sample size during the estimation process. AMCL adopts an approximation technique of particle merging and splitting (PM&S) according to the spatial similarity of particles. In which, particles are merged by their weight based on the discrete partition of the running space of mobile robot.* (http://ieeexplore.ieee.org/document/5512017/)

Background

Localization is a subproblem of navigation which is one of the most challenging competencies required of an autonomous artificially-intelligent mobile robot. Localization tries to find the answer to the following question: given a robot with an internal map, sensor observations and behavior measurement, how can we estimate its position and orientation as it moves around? Localization is challenging because it has to be performed in real-time with reasonable accuracy given the the sensor observations made by the robot is unreliable, and the behavior measurement (dead-reckoning) is also inaccurate. We need MCL to be able to intelligently infer where the robot is given the current sensor observations against a known map.

Model Configuration

AMCL

This packages takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates. On startup, amcl initializes its particle filter according to the parameters provided.

Parameter: Value	Reason for selecting
amcl.launch `<node pkg="amcl" type="amcl" name="amcl" output="screen">` `<remap from="scan" to="udacity_bot/laser/scan"/>` `<param name="odom_frame_id" value="odom"/>` `<param name="odom_model_type" value="diff-corrected"/>` `<param name="base_frame_id" value="robot_footprint"/>` `<param name="global_frame_id" value="map"/>` `<param name="min_particles" value="500"/>` `<param name="max_particles" value="1000"/>` `<param name="update_min_d" value="0.4"/>` `<param name="update_min_a" value="0.5"/>` `<param name="resample_interval" value="2"/>` `<param name="transform_tolerance" value="0.75"/>` `</node>`	- If minimum and maximum allowed number of particles are too high, then the filtering process will take a longer amount of time, however if these are too low then the particles might not even converge to near the actual location of the robot - The **update_min_d** and **update_min_a** is the translational and rotational movement before performing a filter update, if this is too low, then updating the filter will eat up computational power even though nothing has happened that much yet, but if this is too high then the update might be too outdated and many things might have already happened, same reasoning goes for the **resample_interval** - The **transform tolerance** is *(Time with which to post-date the transform that is published, to indicate that this transform is valid into the future.)* Is set a little bit higher because my computer is slow

BASE_LOCAL_PLANNER

This package provides a controller that drives a mobile base in a place. It connects the path planner to the robot. The planner creates a kinematic trajectory for the robot to get from the start

location to the goal location. The job of the controller is to use the value from the planner to determine the linear and angular velocity that the robot uses. I tuned the following parameters:

Parameter: Value	Reason for selecting
base_local_planner_params.yaml `controller_frequency: 2.0` `TrajectoryPlannerROS:` `  max_vel_x: 3.0` `  min_vel_x: 0.2` `  max_vel_y: 0.0` `  min_vel_y: 0.0` `  max_vel_theta: 0.5` `  min_in_place_vel_theta: 0.1` `  acc_lim_theta: 1.0` `  acc_lim_x: 2.5` `  acc_lim_y: 0.0` `  holonomic_robot: false` `  meter_scoring: true` `  yaw_goal_tolerance: 0.05` `  xy_goal_tolerance: 0.2`	- The robot is non-holonomic because it is a differential drive robot having two control inputs (velocity of each wheel) but three degrees of freedom (x, y axis and orientation) - The linear (x axis) and angular (theta) velocity and acceleration and velocity limits are tweaked from the default values so that the robot does not easily overshoot the goal point the path-planner sends. These values are more conservative than aggressive so the robot is slow. - The velocities and acceleration in the y axis are set to zero because since it is a differential drive robot the wheels can only go forward or backward not sideways - I tweaked the tolerance to be a little bit more lax because so that it stops at the desired location faster but at the same time I felt it was visually acceptable - Meter scoring is set to true to assume that the goal distance are expressed in units of meters - Controller frequency was set to sa smaller number to lessen the warning that the controller loop fails to meet the desired frequency because my computer is slow. SOURCE:: http://wiki.ros.org/base_local_planner

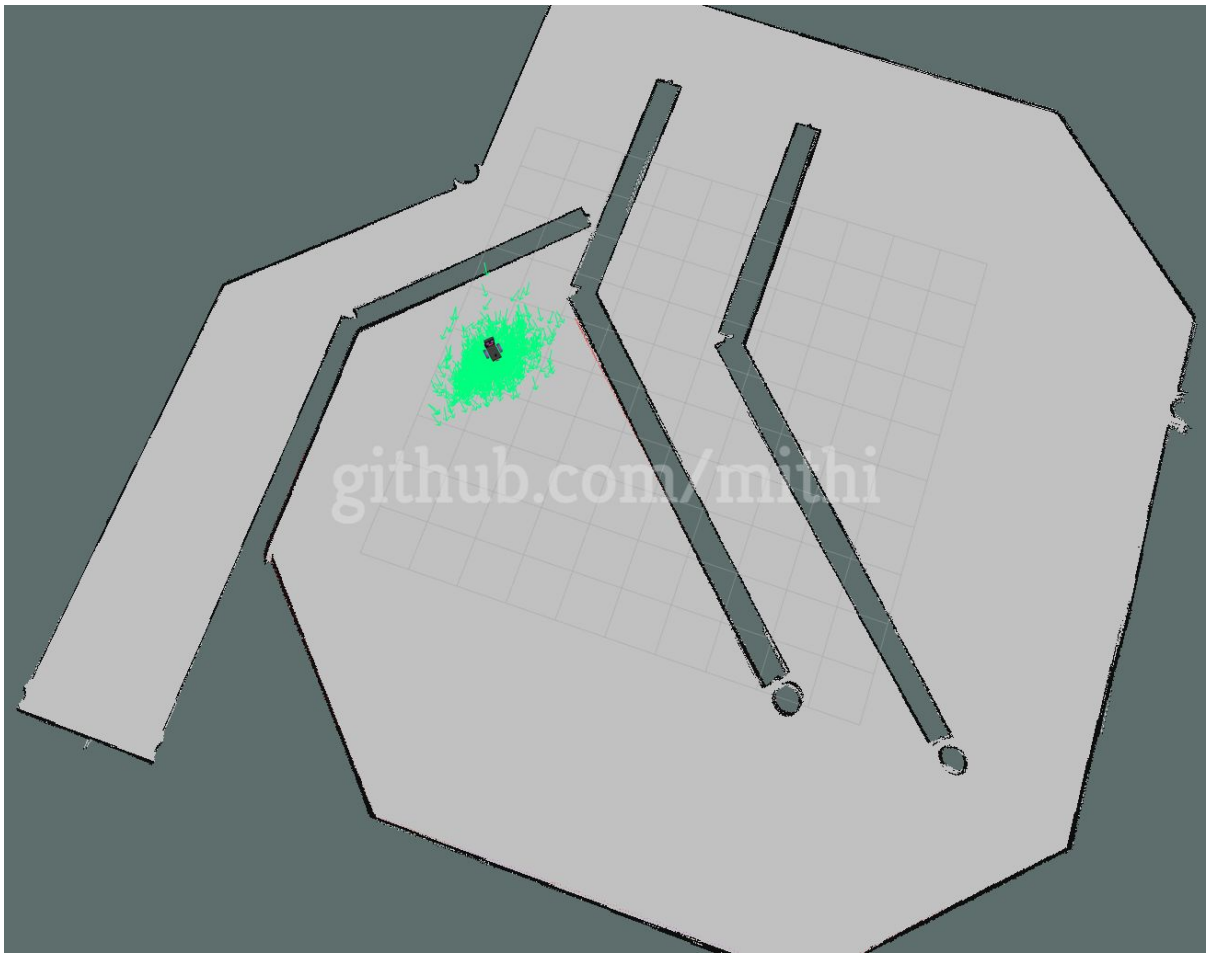
COST_MAPS

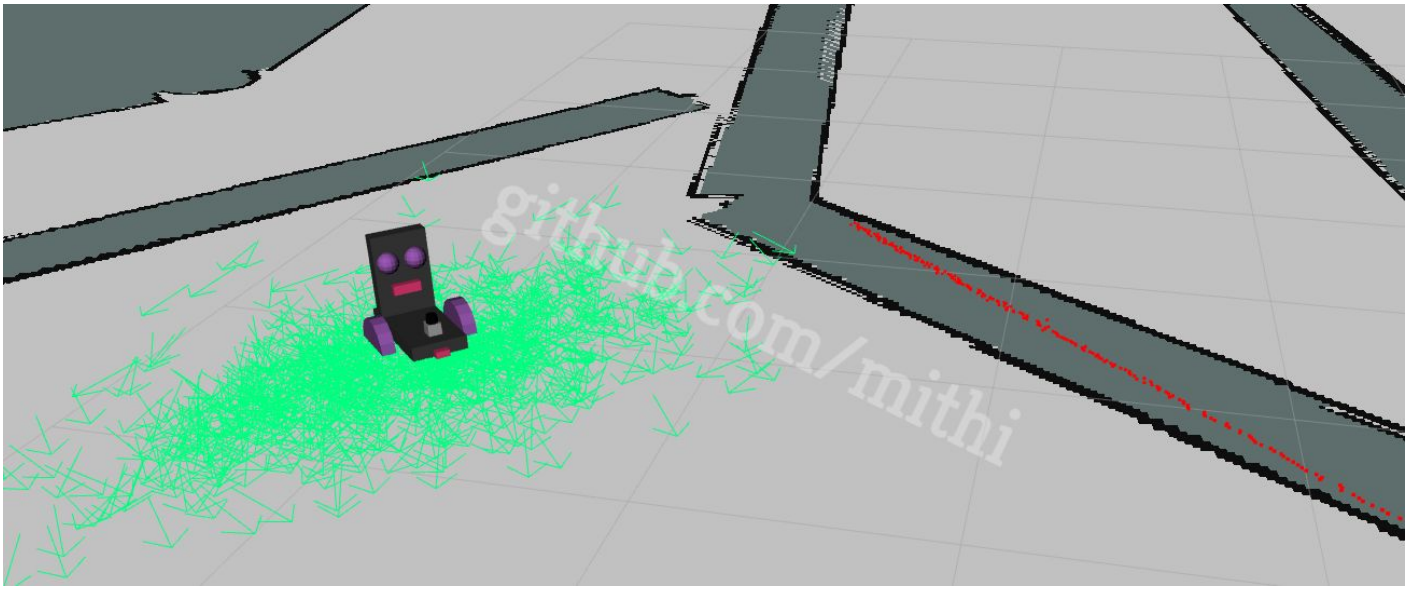
This package builds a 2D map of costs which takes in sensor data from the world and builds an occupancy grid. The cost are inflated based on a user specified radius. The global costmap is for the whole environment while the local cost map is just considering the environment within a small radius of the robot's current location.

Parameter: Value	Reason for selecting
costmap_commons_params.yaml `map_type: costmap` `robot_base_frame: robot_footprint` `obstacle_range: 2.5` `raytrace_range: 3.0` `robot_radius: 0.5` `inflation_radius: 1.0` `update_frequency: 1.0` `publish_frequency: 1.0` `static_map: true` `rolling_window: false` `transform_tolerance: 0.5` `resolution: 0.1` `laser_scan_sensor: {` `  sensor_frame: hokuyo,` `  data_type: LaserScan,` `  topic: /udacity_bot/laser/scan,` `  marking: true,` `  clearing: true` `}` **global_costmap_params.yaml** `global_costmap:` `  global_frame: map` `  width: 40.0` `  height: 40.0` **local_costmap_params.yaml**	- The **update frequency** (how much the map is updated per second) and **publish frequency** (how much the cost map displays information to others) are chosen to be very low because because my computer is not that powerful, so having a high frequency will just bombard the network with computations which will just be dropped. - The **static_map** is set to True because the map of the environment does not change, the **rolling_window** is set to false because of this. - The **inflation_radius** inflates the obstacles to increase the likelihood that the robot does not bump the obstacles. At first this was lower but having small value makes the robot bump to obstacles so this was increased. - The **robot_radius** is set based on the actual footprint of the robot - The **width** and **height** for the local costmap is set to just 4 meters and 40 meters for the global costmap as this seems to be sufficient values. - The **transform_tolerance** specifies the delay in transform (tf) data that is tolerable in seconds, I set this a little higher than the default value because my computer is slow - I changed the **resolution** from 0.05 meters per cell to 0.1 meters per cell, thinking that this might be tolerable and make the computation a little bit faster. - The **global_frame** values for global and local costmap values are the coordinate frames with respect to the map and the robot respectively - The **laser_scan_sensor** values are default values given from Udacity's original files - The **obstacle_range** and **raytrace_trange** are copied from the navigation tutorial as it says: *The "obstacle_range" parameter determines the maximum range sensor reading that will result in an obstacle being put into the costmap. Here, we have it set at 2.5 meters, which means that the robot will only update its map with information about obstacles that are within 2.5 meters of the base. The "raytrace_range" parameter determines the range to which we will raytrace freespace given a sensor reading. Setting it to 3.0 meters as we have above means that the robot will attempt to clear*

`local_costmap:` `  global_frame: odom` `  width: 4.0` `  height: 4.0`	*out space in front of it up to 3.0 meters away given a sensor reading.* **SOURCE:** - http://wiki.ros.org/costmap_2d - http://wiki.ros.org/navigation/Tutorials - http://wiki.ros.org/navigation/Tutorials/RobotSetup

Results





Discussion

The good thing is that I ran this project several times and I was able to navigate to my goal location and orientation each time. The bad things is that it takes 10 minutes to 15 minutes each time, I am thinking that maybe with better parameters even with my slow computer the robot would be able to navigate to the goal location quicker (hopefully less than five minutes).

Future Work

Processing time is almost as important as accuracy to localize a robot. If the robots takes a long time to localize itself it will take a long time for the robot to reach its destination if the goal is navigation. However, if the the accuracy is not that good, it might not be able to reach its desired destination at all in the first place which is its purpose. We have to balance the tradeoff between processing time and accuracy. Instead of aiming best-accuracy-slow inference time or worst accuracy- fast inference time, we should aim for good accuracy-good inference time. For future work, I hope I would tweak the parameters even further so that even with my slow computer the robot would be able to navigate to the goal location quicker (hopefully less than five minutes). Adding more precise sensors would increase the accuracy of localization, and a faster processor or more performant implementation would decrease inference time. The kidnapped robot problem refers to a situation where an autonomous robot is carried to an arbitrary location and the robot has to figure out where it is based on the reading from its on-board sensors. The AMCL method would work on this type of situation provided that it is carried to a location which it will be able to recognize, or in other worlds the aforementioned location is part of its known internal map. To reiterate, if it is carried to a location that exists outside its known internal map, it won't be able to localize itself at all as AMCL relies on comparing its observation to what theoretically would be the observation in a specific location of a known map. This is of course under the assumption that reliable accurate sensors are installed properly to the robot and the locations of these sensors are known, considered and incorporated in the implementation of the algorithm. Specific types of scenarios that should be accounted for include a scenario where the particles do not converge to a location for one reason or another. The algorithm should refresh or re-generate random particles occasionally in hopes of being able to to localize better the next time around.

Other References:

- http://ieeexplore.ieee.org/document/5512017
- https://github.com/turtlebot/turtlebot_apps/tree/indigo/turtlebot_navigation
- https://github.com/richardw05/mybot_ws
- https://github.com/husky/husky/tree/indigo-devel/husky_navigation
- http://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch