# Instruction for Project 4

## AMCL (Adaptive Monte Carlo Localization)

### Introduction

In Project 3, you've built a map of the given world using a SLAM package. Once a map is acquired with SLAM, you don't have to run SLAM repeatedly in order to navigate within the world. You can just exploit the acquired map. Specifically, you can localize the robot, or figure out the pose (position, orientation) of the robot within that fixed map using localization algorithms. The localization algorithm that you'll use in Project 4 is Adaptive Monte Carlo Localization (AMCL). We will give you a pre-made map, and an AMCL ROS package. You'll now navigate your robot based on the localization information. In other words, in the main, you'll subscribe to the output of AMCL instead of Gazebo model states to receive the pose information of the robot.

In Project 4, you will implement all the things needed for the final project. In final project, your goal is to make the car follow the track. However, the track will be not exactly same as the project 4. Therefore, you need to generalize your path generating code for arbitrary map. Your code must select some waypoints on the track so that the RRT algorithm can find appropriate path for the car.

### Adaptive Monte Carlo Localization (Theory) 1)

Monte Carlo localization (MCL), also known as particle filter localization, is an algorithm for robots to localize using a particle filter. Given a map of the environment, the algorithm estimates the position and orientation of a robot as it moves and senses the environment. The algorithm uses a particle filter to represent the distribution of likely states, with each particle representing a possible state, i.e., a hypothesis of where the robot is. The algorithm typically starts with a uniform random distribution of particles over the configuration space, meaning the robot has no information about where it is and assumes it is equally likely to be at any point in space. Whenever the robot moves, it shifts the particles to predict its new state after the movement. Whenever the robot senses something, the particles are resampled based on recursive Bayesian estimation, i.e., how well the actual sensed data correlate with the predicted state. Ultimately, the particles should converge towards the actual position of the robot.

The state of the robot depends on the application and design. For example, the state of a typical 2D robot may consist of a tuple $(x, y, \theta)$ for position $x, y$ and orientation $\theta$. The belief, which is the

robot's estimate of its current state, is a probability density function distributed over the state space. In the MCL algorithm, the belief at a time t is represented by a set of $X_t = \{x_t{}^1, x_t{}^2, \dots, x_t{}^M\}$. Each particle contains a state, and can thus be considered a hypothesis of the robot's state. Regions in the state space with many particles correspond to a greater probability that the robot will be there—and regions with few particles are unlikely to be where the robot is. The algorithm assumes the Markov property that the current state's probability distribution depends only on the previous state (and not any ones before that), i.e., $X_t$ depends only on $X_{t-1}$. This only works if the environment is static and does not change with time. Typically, on start-up, the robot has no information on its current pose so the particles are uniformly distributed over the configuration space.

Given a map of the environment, the goal of the algorithm is for the robot to determine its pose within the environment. At every time t the algorithm takes as input the previous belief $X_{t-1} = \{x_{t-1}{}^1, x_{t-1}{}^2, \dots, x_{t-1}{}^M\}$, an actuation command $u_t$, and data received from sensors $z_t$; and the algorithm outputs the new belief $X_t$

Algorithm MCL$(X_{t-1}, u_t, z_t)$:
$\bar{X}_t = X_t = \emptyset$
for $m = 1$ to $M$:
$x_t^{[m]} =$ motion_update$(u_t, x_{t-1}^{[m]})$
$w_t^{[m]} =$ sensor_update$(z_t, x_t^{[m]})$
$\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
endfor
for $m = 1$ to $M$:
draw $x_t^{[m]}$ from $\bar{X}_t$ with probability $\propto w_t^{[m]}$
$X_t = X_t + x_t^{[m]}$
endfor
return $X_t$

Monte Carlo localization may be improved by sampling the particles in an adaptive manner based on an error estimate using the Kullback–Leibler divergence (KLD). Initially, it is necessary to use a large M due to the need to cover the entire map with a uniformly random distribution of particles. However, when the particles have converged around the same location, maintaining such a large sample size is computationally wasteful.

AMCL (or KLD-sampling) is a variant of Monte Carlo Localization where at each iteration, a sample size $M_x$ is calculated. The sample size $M_x$ is calculated such that, with probability $1 - \delta$, the error between the true posterior and the sample-based approximation is less than $\epsilon$. The variables $\delta$ and $\epsilon$ are fixed parameters. The main idea is to create a grid (a histogram) overlaid on the state

space. Each bin in the histogram is initially empty. At each iteration, a new particle is drawn from the previous (weighted) particle set with probability proportional to its weight. Instead of the resampling done in classic MCL, the AMCL algorithm draws particles from the previous, weighted, particle set and applies the motion and sensor updates before placing the particle into its bin. The algorithm keeps track of the number of non-empty bins, $k$. If a particle is inserted in a previously empty bin, the value of $M_x$ is recalculated, which increases mostly linear in $k$. This is repeated until the sample $M$ is the same as $M_x$.

It is easy to see AMCL culls redundant particles from the particle set, by only increasing $M_x$ when a new location (bin) has been filled. In practice, AMCL consistently outperforms and converges faster than classic MCL.

## Make AMCL package and Implement your code

1. Extract project4.zip file to the src folder in your catkin workspace. You may change mode of the files by "chmod -R 777 project4_skeleton"
2. In terminal, type "sudo apt-get install ros-kinetic-navigation ros-kinetic-scan-tools"
3. type "cd ~/catkin_ws" and "catkin_make"
4. Copy and Paste TODO 1 parts in main.cpp from project3 code.
5. Complete TODO 2 parts in set_waypoints() function in main.cpp.

After the completion of TODO parts, you can test the localization while navigating by executing "*roslaunch project4 project4.launch*", since the AMCL usage is already implemented by TAs. However, in order to improve performance by tuning, you can modify some key parts of the AMCL package.

1) AMCL requires a robot odometry which estimates the pose of the robot from the robot's internal states such as speed and steering angle. AMCL then compensate the error of this estimation which is called odometry drift. You will use laser_scan_matcher package to generate the odometry. http://wiki.ros.org/laser_scan_matcher

2) In localization.launch file, there are AMCL configurations that TAs set. There are filter parameters, laser model parameters, and odometry model parameters which are adjustable. You can refer to http://wiki.ros.org/amcl for descriptions of each parameters.

For project 4, the set_waypoints() function should make waypoints from an arbitrary map which has form of a circuit track. Except for the start point and the goal point, it is not recommended to assign fixed points to waypoints (for final project). A simple way to generate waypoints is to divide

the map into a few parts (eg. by 4 quadrants), and randomly sample a point from empty space of each part of the map. However, if you wish, you can generate waypoints in your own ways.

To make your waypoints far from walls, you can set the variable **"int waypoint_margin"**, and you can access specific point in the map by **"map_margin.at<uchar>(i, j)".** If it has the value of 255, it means it's color is white, and 0 means black. Also, if the point you selected is in the i-th row, j-th column in the image, the position of the point on the real coordinate will be **x=res*(i-map_origin_x)** , and **y=res*(j-map_origin_y).**

You should notice that the input map image at /project4/src/slam_map.pgm is rotated. This is because of the problem of map_server package. The map will be re-rotated in the main code, so **map_margin** has the size of 540x179, while slam_map.pgm has the size of 179x540. Do not rotate the slam_map.pgm file manually outside of the main.cpp code.

## Submission Format

Compress your project 2 and project 4 folder into a single compressed file. Then, upload it on eTL. The name of the compressed file should be "**IS_Project_04_[TeamName].tar.gz**".

## Reference

1) https://en.wikipedia.org/wiki/Monte_Carlo_localization