Udacity Robotics Software Nanodegree: Localization Project

Wisnu Prasetya Mulya

Abstract—This paper examines the implementation of AMCL and move_base ROS packages and the parameters tuning of them to navigate to a goal position. Two simulated robots are being the subjects of this paper: one whose properties are specified by Udacity and one that is made personally by the author. The results show that with certain parameter values, both robots could navigate into the goal position with the specified goal's XY and yaw tolerance of 3%.

Index Terms—Robot, IEEEtran, Udacity, LaTeX, Localization.

1 Introduction

The primary task of this paper is to examine the implementation of two ROS packages, the AMCL and the move_base, on the subject robots and to elaborate the role of each essential parameters for the packages to function and how to fine tune them in order to successfully overcome the challenge of navigating the subject robots to a goal position and orientation, which is specified by Udacity.

There are two simulated robots that are the subject in this paper: the one constructed by following Udacity guideline, which will be referred to as udacity-bot throughout the rest of the paper, and the one constructed personally by the author, which will be referred to as custom-bot throughout the rest of the paper.

The parameters of the udacity-bot are tuned independently, while the custom-bot adapts the working parameters from the udacity-bot and only differ in parameters pertaining to its dimension. Also, several parameter specifications for the udacity-bot are adapted from the ones illustrated in the packages' documentations and from the working parameters in the work of Kaiyu Zheng [1].

AMCL and move_base packages are used to perform localization and navigation stack respectively and their implementation in navigating the robots towards the goal position and orientation requires fine tuning of their parameters. This is an arduous task without a proper understanding of each parameter's role and how to adjust them differently for different robot. Choosing different values randomly for the parameters and examining how each affects the localization and the navigation of the robot will take a considerable amount of time. Therefore, an examination on this task becomes important, so that the results will guide for a quick setup of the parameters of both packages.

2 Background

The challenge in determining robots position and orientation is substantial, since first, the real world has innate random properties, such as uneven ground and random air resistance, and second, the sensors installed on the robots have their own noises that would affect the accuracy of the measurement data collected.

The notion has given a birth to the subject of localization in the robotics field. Localization is the process of determining the current position and orientation of robots by the means of filtering the errors resulting from natural randomness and innate noises of the sensors. Several prominent localization algorithms have been discovered to tackle the challenge effectively, but in this paper, the author focuses only on the implementation of the Monte Carlo Localization (MCL) algorithm, since several parameters of this algorithm can be adjusted according to the performance of the processing unit of the robots.

Further, alongside the MCL implementation, a navigation stack will be implemented as well in order to move the robots towards a goals position specified by Udacity. The navigation stack will move the robots' base towards the goal position and orientation, which would take into consideration the robots' odometry and sensors.

2.1 Kalman Filters

Kalman Filters are localization algorithms which use Gaussian distribution assumption in filtering noises from the inputs. Also, aside from assuming that the state variables are normally distributed, it assumes that they are linear. It works in producing accurate approximations, since it assigns weights to the sensor inputs based on their standard errors (variance of the Gaussian distribution).

Further, the drawbacks of only covering for normally distributed and linear states are improved in the Extended Kalman Filters. It works by performing the similar steps as in the regular Kalman Filters, but it approximates the non-linear states by using Taylor Series to approximate the locally linear function of the states.

2.2 Particle Filters

Particle Filters, such as the MCL, are localization algorithms which filter randomly generated particles in a given known map by assigning Bayesian probabilities to each particle in determining how accurate their position and orientation are with regards to the known map and the robot's inputs of its surrounding.

2.3 Comparison / Contrast

Even though both Kalman Filters and Particle Filters serve the same purpose, they differ in their performance based on the assumptions about the states that are to be approximated and the capability of the processing unit of the robot. Analyzing their benefits and disadvantages is therefore crucial in determining which algorithm to pick for the localization task and for future reference.

The followings are the characteristics of the Kalman Filters, including its benefits and disadvantages:

- Assumes Gaussian distribution for the state variables
- More efficient than Particle Filters
- More complex in implementation than Particle Filters
- Memory and resolution cannot be controlled

Also, below are the characteristics of the Particle Filters, including its benefits and disadvantages:

- Assumes any distribution for the state variables
- More simple to be implemented than Kalman Filters
- Memory and resolution can be controlled
- Less efficient than Kalman Filters

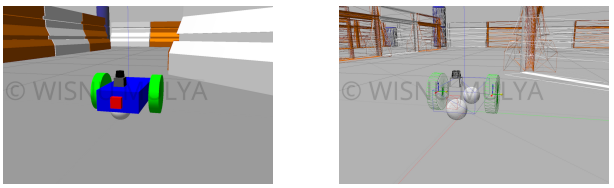
3 Simulations

The two robots that are being the subject in this paper are built in simulation. Specific details regarding their models, packages used, and the parameters used in the AMCL and move_base packages are elaborated in the following subsections.

Further, the specification of the machine of which the simulation is conducted is as follow:

- Machine: Mid-2014 Macbook Pro Retina 13-inch with macOS 10.13.3
- Virtual Machine Software: VMware Fusion Pro 10.1.1
- Virtual Machine Image: Linux 4.8.0-58-generic Ubuntu 16.04.2 LTS with 2 processor cores and 4MB memory

3.1 Benchmark Model



(a) udacity-bot model (b) udacity-bot wireframe

Fig. 1: Udacity-bot Model

The benchmark model of the robot whose configurations are being specified by Udacity is called the udacity-bot in this paper. The udacity-bot has a box base, two wheels, two casters, a camera, and a Hokuyo laser rangefinder.

3.1.1 Model design

The udacity-bot configurations is elaborated in Table 1.

TABLE 1: Udacity-bot Model Design

Property	Value
Chassis	
Shape	Box
Dimension	0.4 x 0.2 x 0.1
Mass	15
Caster Radius	0.05
Caster mu	0
Caster mu2	0
Caster slip1	1.0
Caster slip2	1.0
Back Caster Position (X,Y,Z)	-0.15, 0.0, -0.05
Front Caster Position (X,Y,Z)	0.15, 0.0, -0.05
Wheels	
Shape	Cylinder
Length	0.05
Radius	0.1
Mass	5
RPY	0.0, 1.5707, 1.5707
Joint Type	Continuous
Joint Axis	Y
Joint Damping	1.0
Joint Friction	1.0
Left Wheel Joint Origin (X,Y,Z)	0.0, 0.15, 0.0
Right Wheel Joint Origin (X,Y,Z)	0.0, -0.15, 0.0
Camera	
Shape	Box
Dimension	0.5 x 0.5 x 0.5
Mass	0.1
Joint Type	Fixed
Joint Origin (X,Y,Z)	0.2, 0.0 0.0
Hokuyo Rangefinder	
Shape	Mesh File
Joint Type	Fixed
Joint Origin (X,Y,Z)	0.15, 0.0, 0.0
Differential Drive Controller	
Update Rage	10
Wheel Separation	0.4
Wheel Diameter	0.2
Torque	10

3.1.2 Packages Used

There are two ROS packages used for this robot:

- AMCL
- move_base

3.1.3 Parameters

The parameters of the packages are elaborated in Table 2.

3.2 Personal Model

(a) custom-bot model (b) custom-bot wireframe

Fig. 2: Custom-bot Model

The second robot model, which is made by the Author, is called the custom-bot in this paper. As shown in Figure 2, the custom-bot has a cylinder base, two spherical wheels, two casters, a camera, and a Hokuyo laser rangefinder. Thus, the custom-bot only differs in its base and wheels from the udacity-bot.

TABLE 2: Udacity-bot Packages' Parameters

Property	Value
AMCL	
Odom Model Type	diff-corrected
Minimum Particles	20
Maximum Particles	100
Transform Tolerance	0.3
Laser Model Type	likelihood_field
laser_z_hit	0.95
laser_z_rand	0.05
Odom alpha1	0.001
Odom alpha2	0.001
Odom alpha3	0.001
Odom alpha4	0.001
move_base	
Controller Frequency	5.0
Maximum X Velocity	1.0
Yaw Goal Tolerance	0.04
XY Goal Tolerance	0.03
Sim Time	4.0
Meter Scoring	True
Path Distance Scale	2.0
Obstacle Range	2.5
Ray Trace Range	3.5
Transform Tolerance	0.3
Inflation Radius	1.75
Global & Local Update Frequency	5.0
Global & Local Publish Frequency	5.0
Global & Local Resolution	0.02
Global Width & Height	5.0
Local Width & Height	1.0

3.2.1 Model design

The custom-bot configurations is elaborated in Table 3.

3.2.2 Packages Used

Similar to the udacity-bot, there are two ROS packages used for this robot:

- AMCL
- move_base

3.2.3 Parameters

The parameters of the packages are elaborated in Table 4.

4 Results

There are two measurements that are being observed: the time for the localization to converged and the time for the robot to get to the goal position and orientation. The summary of the result is given under Table 5.

The parameters of AMCL and move_base for both robots are all the same, except for the `footprint` and `robot_radius` where the udacity-bot has the former and the custom-bot has the latter.

Another difference is the parameters applied to the differential drive controller plugin. For the `wheelSeparation` parameter, the udacity-bot is set to 0.4, while the custom-bot is set to 0.2. For the `wheelDiameter` parameter, the udacity-bot is set to 0.2, while the custom-bot is set to 0.1.

The observed results show that the custom-bot appears to be faster in getting to the goal position and also faster for its localization to converge. However, both of their localization process seems to converge at the time when the robot is making a u-turn.

TABLE 3: Custom-bot Model Design

Property	Value
Chassis	
Shape	Cylinder
Radius	0.2
Length	0.1
Mass	15
Caster Radius	0.05
Caster mu	0
Caster mu2	0
Caster slip1	1.0
Caster slip2	1.0
Back Caster Position (X,Y,Z)	-0.1, 0.0, -0.05
Front Caster Position (X,Y,Z)	0.1, 0.0, -0.05
Wheels	
Shape	Sphere
Radius	0.05
Mass	5
RPY	0.0, 0.0, 0.0
Joint Type	Continuous
Joint Axis	Y
Joint Damping	1.0
Joint Friction	1.0
Left Wheel Joint Origin (X,Y,Z)	0.0, 0.1, -0.05
Right Wheel Joint Origin (X,Y,Z)	0.0, -0.1, -0.05
Camera	
Shape	Box
Dimension	0.5 x 0.5 x 0.5
Mass	0.1
Joint Type	Fixed
Joint Origin (X,Y,Z)	0.2, 0.0 0.0
Hokuyo Rangefinder	
Shape	Mesh File
Joint Type	Fixed
Joint Origin (X,Y,Z)	0.15, 0.0, 0.1
Differential Drive Controller	
Update Rage	10
Wheel Separation	0.2
Wheel Diameter	0.1
Torque	10

Also, when making a u-turn, the custom-bot is observed to have poorer performance in following the global path, while the udacity-bot is observed to not deviate as much as the custom-bot.

4.1 Localization Results

4.1.1 Udacity-bot

The moment when the udacity-bot achieves the goal position and orientation is portrayed in Figure 3.

4.1.2 Custom-bot

The moment when the custom-bot achieves the goal position and orientation is portrayed in Figure 4.

5 Discussion

The discussion of the results is divided into two sections: the discussion regarding the parameters and the discussion regarding the results observed.

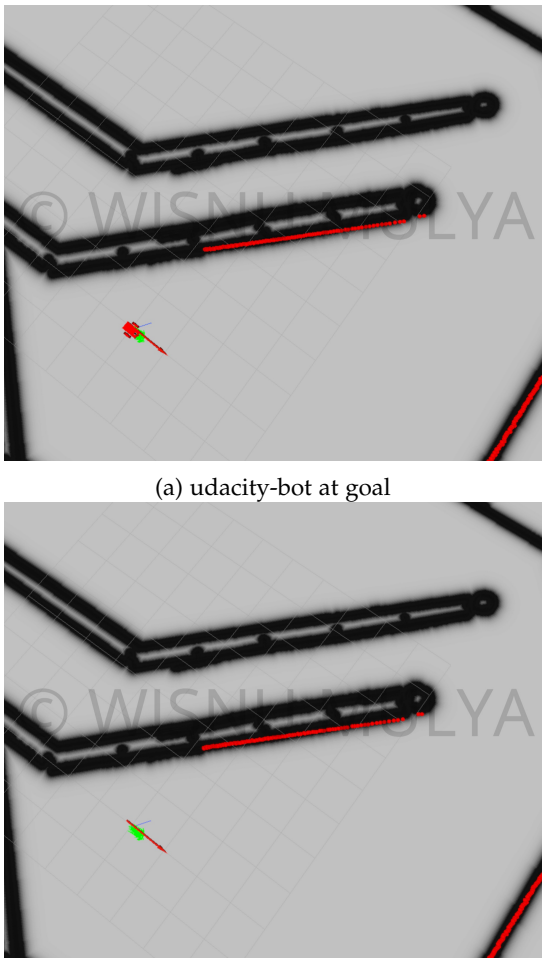
Further, regarding the parameters, an independent tuning is only conducted in the parameters for the udacity-bot, while the custom-bot copies all of the parameters of udacity-bot's, except for the ones pertaining to its dimension. The custom-bot has similar parameters due to that they achieve a good job in navigating the custom-bot to the goal position and orientation with no significant issue.

TABLE 4: Custom-bot Packages' Parameters

Property	Value
AMCL	
Odom Model Type	diff-corrected
Minimum Particles	20
Maximum Particles	100
Transform Tolerance	0.3
Laser Model Type	likelihood_field
laser_z_hit	0.95
laser_z_rand	0.05
Odom alpha1	0.001
Odom alpha2	0.001
Odom alpha3	0.001
Odom alpha4	0.001
move_base	
Controller Frequency	5.0
Maximum X Velocity	1.0
Yaw Goal Tolerance	0.04
XY Goal Tolerance	0.03
Sim Time	4.0
Meter Scoring	True
Path Distance Scale	2.0
Obstacle Range	2.5
Ray Trace Range	3.5
Transform Tolerance	0.3
Robot Radius	0.2
Inflation Radius	1.75
Global & Local Update Frequency	5.0
Global & Local Publish Frequency	5.0
Global & Local Resolution	0.02
Global Width & Height	5.0
Local Width & Height	1.0

TABLE 5: Result Summary

	Udacity-bot	Custom-bot
Convergence Time (second)	409	278
Navigation Time (second)	895	810



(a) udacity-bot at goal

(b) udacity-bot at goal no base

Fig. 3: Udacity-bot at Goal Position

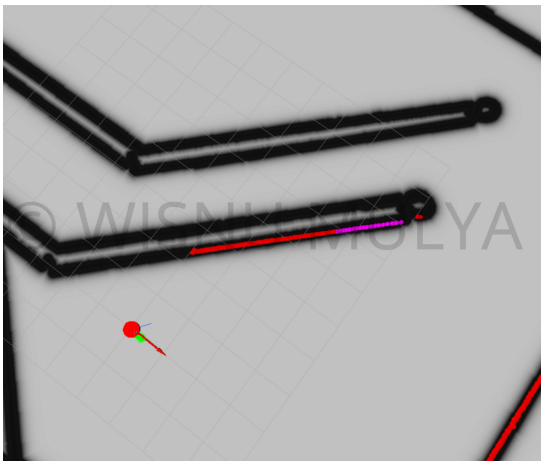
5.1 Parameters Discussion

5.1.1 AMCL

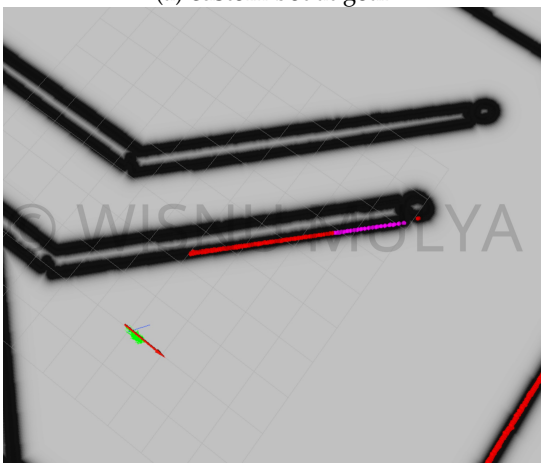
- The `min-particles`, `max-particles`, and the `transform_tolerance` are tuned based on the performance of the processing unit in conducting localization. It is observed by the author that the values of 20, 100, and 0.3, respectively, are in relation with the 5Hz value of `update_frequency` and `publish_frequency` in producing a map visualization that is not flickering and significantly less frequency-inconsistency warnings thrown.
- For the laser parameters, the `laser_model_type` used is the `likelihood_field`, since it is suggested by Udacity as the one that is more computationally efficient and more reliable in simulation. Two additional parameters are `z_laser_hit` and `z_laser_rand` which are to specify the accuracy of the laser sensor, which are both set to their default values, 0.95 and 0.05 respectively.
- For the odometry, since the differential drive controller is used, the `odom_model_type` is set to `diff-corrected`. Further, as suggested in the documentation of AMCL package, that specification entails only to set the alphas from `odom_alpha_1` to `odom_alpha_4` and they are all set to a very small value of 0.001, since the noises are non-existent in the simulation.

5.1.2 move_base

- For the global and local parameters, both of their `update_frequency` and `publish_frequency` are set to 5.0, since they are both dependent upon the processing unit performance and in relation with several AMCL parameters mentioned previously, it produces less warnings and less flickering map. The other parameter with the same value is the `resolution`, which for both global and local is set to 0.2. However, for the `width` and `height`, the global has a value of 5.0 and the local has a value of 1.0, since a small value for both of them results in a better behavior of the robot in following the local and global path.
- Some parameters for the common costmap are set according to the illustration code in the package's documentation. They are `obstacle_range` and `raytrace_range` which are set to 2.5 and 3.0 respectively. Then, `footprint` is specified for the udacity-bot, since it has a box chassis, while `robot_radiu` is specified for the custom-bot, since it has a cylindrical chas-



(a) custom-bot at goal



(b) custom-bot at goal no base

Fig. 4: Custom-bot at Goal Position

sis. Also, the `transform_tolerance` and the `controller_frequency` are set to 0.3 and 5.0Hz respectively, which are observed to work well with other parameters for both packages in reducing warnings thrown. Another parameter is the `inflation_radius`, which determines how far does the path needs to stay away from the obstacles, and it is set to 1.75 according to the paper written by Kaiyu Zheng [1], since it produces the path that are in the middle of obstacles.

- For the base local planner parameters, the `yaw_goal_tolerance` and `xy_goal_tolerance` are set to 0.03, since it has been observed to be the lowest value of decimal to the power of two which will keep the robot from rotating too long in adjusting its orientation when goal position is achieved. Then the `max_vel_x` is set to be 1.0, so that it has faster speed rather than its default. Further, the meter_scoring is set to be `true` and the `sim_time` and `pdist_scale` are set to 4.0 (as suggested by the paper by Kaiyu Zheng [1]) and 2.0 respectively, since they result in the local path being close to the global's.

5.2 Result Discussion

It has been observed that the custom-bot performs better than the udacity-bot with regards to both how fast the particles converges and how fast the robot achieves the goal position and orientation.

Since the only differing parameters from both robots are their chassis' shapes, dimensions, and their wheels, the difference in performance might be explained by the size of their wheels.

Even though the torque specified to both robots are the same, the smaller size of the custom-bot's wheels might make it travel faster and thus, the convergence of the localization particles is also faster, since it makes the u-turn first (of which it has been encountered that the convergence seems to happen when the robot is making a u-turn).

Further, the problem of 'Kidnapped Robot' when using MCL is observed to be poor. A test of deactivating AMCL when performing navigation towards the goal and reactivating it again (thus emulating the condition of the robot being kidnapped) results in the robot getting off-tracked from the navigation path and seem to not be able to getting back on track.

Overall, the AMCL has performed a great job in approximating the position and orientation of both robots, even though both of them have a different shape and size. Using it with a navigation stack such as move_base is appropriate in a real life situation that involves navigating a robot in a known map. For example, the task of navigating a robot in a warehouse would be effectively conducted by using AMCL.

6 Conclusion / Future work

In conclusion, AMCL is appropriate in conducting the task of localization when there is restriction in the capability of the processing unit of the robot, since its parameter could be tuned to lower the accuracy, but resulting in a better and faster performance. Its application is thus, specifically effective in a robot which has a low computing capability, for which its performance can be improved by sacrificing accuracy.

Further, a significant issue in completing the task in this paper is that it still takes a considerable amount of time for the robot to navigate towards the goal position and orientation. An improvement in the future which would result in faster processing will impact the performance significantly. This might be achieved by allocating a greater processing power and memory to the simulator (which translate in a faster processing unit in the real world) and also simpler sensors that would reduce the input data size and resulting in a faster process.

References

[1] K. Zheng, "Ros navigation tuning guide," *eprint arXiv:1706.09068*, 2017.