The Adaptive Monte Carlo Localization Algorithm: a Basic Implementation

Pinto, Jorge

Abstract—An algorithm for an Adaptative Monte Carlo Localization (AMCL) task was implemented. Two designs of robot models (benchmark vs personal) were created to evaluate the algorithm on a provided map. A goal set point was established via an executing script so the robots could move autonomously. Although the objective was accomplished in both case studies, differences in performance was observed. The number of particles generates by the algorithm was modified to absorb the loss of position information due to uncertainties. Whether a design is chosen over the one depends on the application the robot is intended, as well as the task's requirements in terms of time for completion and uncertainty.

Index Terms—Monte Carlo, Localization, Gazebo, Navigation.

1 Introduction

THE Monte Carlo localization (MCL) algorithm was defined as an option to the existing Markov probabilistic frameworks [1] for global localization and tracking position in robots. Although procedures may prove to be useful under particular circumstances, the Adaptive MCL approach showed to be advantageous in terms of computational requirements; in this project, such scheme will be evaluated by developing a localizing algorithm in a simulated environment, for both a provided robot model ("Udacity_bot) and a personalized robot. These models will also be required to reach a goal position defined by the script *navigation_goal.cpp* and after the algorithm has been deployed. Other examples are suggested in the literature for applications of this method in different case studies [2]–[4].

2 Background

As [1] affirms, sensor-based localization has been acknowledged as a common challenge in autonomous robots. The term localization is defined as "a version of on-line temporal state estimation" [1], whenever a mobile robot is able to determine its current position x,y and orientation θ over a global coordinate frame. As the reader may infer however, such task involves a series of issues roboticists deal with everyday, where global localization and position tracking are the stand outs of the navigation task.

In this sense, many approaches have been developed in the past to improve the task of determining the position of a mobile robot from either a known or an unknown position. The former is used for position tracking, and is attainable my including small corrections into its sensed odometry. The latter on the other hand, is characteristic for the global localization task and it becomes difficult as the robot tries to determine its position from scratch (e.g, the hijacked robot problem).

Therefore, several methods have been proposed for crack down the issue. Next subsections include a description of the usual frameworks followed for this task, with a particular focus on those analyzed in this project.

2.1 Kalman Filters

Kalman filters have been used in the past for position estimation and tracking [5]. Easy to implement, it has proven to work assuming that robot's belief is describable by a Gaussian multivariate function. In addition, many linear models can benefit from this model due to its simplicity although, in case of non-linear models, a linearisation is necessary before the filter can be properly applied (the so called Extended Kalman Filter or EKF). Nonetheless, given the nature of the robot's movement, the distribution of its odometry may be multi-modal thus rendering the global localization task unfeasible under this scheme.

2.2 Particle Filters

Particle filters (i.e., sampling techniques) have also been applied for localization tasks. Markov localization [6] has been in the radar for some time and it represents the robot's current position (or belief) as a discrete multi-modal distribution against different possible positions, and update it afterwards whenever it location changes. Such approach implies discretizing the state space of the robot in term of the searched for resolution (e.g. uncertainty).

In this project, MCL is kept as a variation of the "sampling/importance resampling" procedure [1], as it allows to determine the probability of a computed set of random samples or particles (i.e., belief) of the type ((x,y,θ),p) where

1) x,y,θ represents a robot's position and
2) p a weighting factor similar to a discrete probability

New samples are then generated after a motion command, with such likelihood determined by the factor p. As the robot moves, further particles would be discarded/generated in function of their uncertainty regarding its position. For the whole definition, reader is suggested to check out the referenced literature [1], [6].

2.3 Comparison

Some differences exist among the above described approaches due to their probabilistic nature:

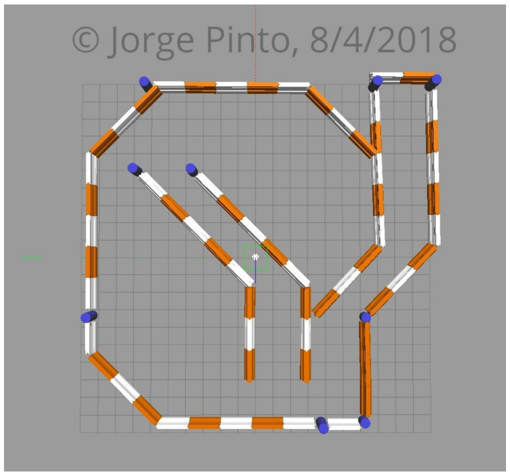


Fig. 1: Top view of the Map provided in Gazebo®

- MCL can determine a robot's global position due to its flexibility in representing multi-modal distributions, and this makes the method more robust than EKF.
- EKF is proved to be more efficient in terms of memory usage and time due to its focus over Gaussian distributions in addition to its ease of implementation for linear models.
- Given the randomness of the generated samples, fine resolutions would be more uncertain under MCL as compared to the EKF method.

Whether a technique is chosen over the other one would depend on the case of study and the number of variables therein involved. As a matter of fact, given the nature of this project, the Adaptive MCL technique was selected for determining the robot's position.

3 Simulations

In this project, two different robot models were analyzed in Gazebo®and Rviz®simulation environments, for which the map shown in Fig. 1 was provided beforehand. Keep in mind that as a common features, both models were required to have a camera (sight), a laser (surroundings and range detection) and two actuators (side wheels); therefore, three gazebo plug-ins were included in a definition file before implementation.

Moreover, all the necessary ROS kinetic packages were installed in Linux®for the robots' motion adaption. Nonetheless, other ingredients were required in this regard.

3.1 Packages

Both ROS *AMCL* and *Navigation Stack* packages were added: the former adjusted the number of particles generated by the MCL localization algorithm as the robot navigates over time; for the latter, the move_base package was written so the robot could implement a moving action given a local goal in a provided map. In both cases, different parameters were provided (see Figs.5 and 7).

3.2 Topics

The general channels for information transmission was used. However, given the requirement of both robot models for localization and position tracking (impersonated by ROS *AMCL* and *Navigation Stack* packages), two topics stand out:

1) **odom**: From nav_msgs/Odometry.msg. In terms of *Navigation Stack* package, as per ROS documentation, "the navigation stack uses tf to determine the robot's location in the world and relate sensor data to a static map. However, tf does not provide any information about the velocity of the robot. Because of this, the navigation stack requires that any odometry source publish both a transform and a nav_msgs/Odometry message over ROS that contains position and velocity information" [7]. On the other hand, it is required by *AMCL* to dynamically adapt the number of particles as the robot's position change.
2) **scan**: From sensor_msgs/LaserScan.msg, whose information is generated by the placed sensor ("hokuyo sensor" in this case) for range and surroundings detection during motion planning.

3.3 Benchmark Model

The "Udacity_bot" was provided in an initial stage for setting up the simulation environment. Fig.2 provides an example of such instance in steady-state.

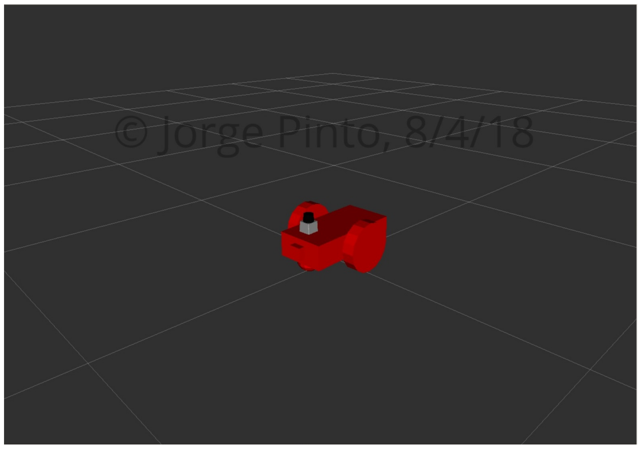


Fig. 2: Model provided for simulation.

3.3.1 Model design

As observed in the figure, the "Udacity_bot" had the following elements:

1) Base: Rectangular, of dimensions 0.4x0.2x0.1 meters, and mass 10 kgs.
2) Actuators: Two cylindrical wheels of radius 0.1 and length 0.05 meters, and mass 5kgs each, located at each side of the base.
3) Camera: Located in front side of the base, as a 0.05x0.05x0.05 meters box.
4) Hokuyo-sensor: Placed on top-front side of the base.
5) Caster Wheels: Located in both front and back side of the base to avoid continuous tilting over the

vertical axis of the robot. Each conceived as a sphere of radius 0.05 meters.

All of the elements mentioned above were hinged to the base by links defined in the respective .xacro file.

3.4 Personal Model

Fig.3 provides an image of the personalized robot in steady-state. Observe that its differences regarding the former model were an established requirement to carry out the simulations.

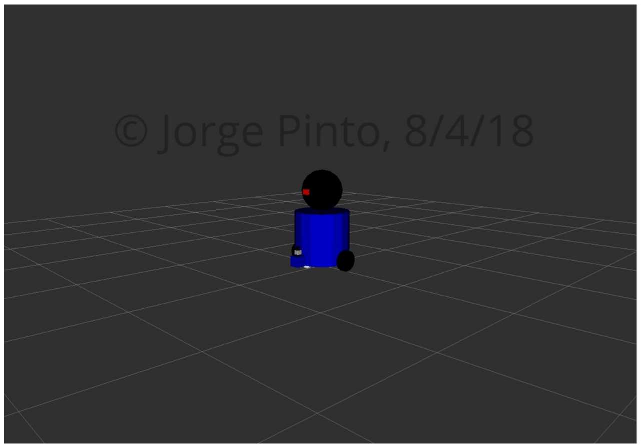


Fig. 3: Model independently created

3.4.1 Model design

The figure above portrays the general arrangement of elements of the personalized robot (could be thought of a ”friendly” trash can), comprised of

1) Base: Cylindrical, of radius 0.27 and length 0.5 meters, and mass 10 kgs.
2) Head: Spherical of radius 0.2 meters and 5 kgs of mass.
3) Actuators: Two cylindrical wheels of radius 0.1 and length 0.05 meters, and mass 5kgs each, located at each side of the base.
4) Camera: Located in the front of the head (the nose of the robot), as a 0.05x0.05x0.05 meters box.
5) Hokuyo-sensor: Placed on the bottom-front side of the body, on a 0.1x0.1x0.1 meters box with mass 0.1 kgs.
6) Caster Wheels: Located in both front and back side of the base to avoid continuous tilting over the vertical axis of the robot. Each conceived as a sphere of radius 0.05 meters.

Similarly, these elements were hinged to the base by links defined in the respective .xacro file, completely independent from that of the Benchmark's.

3.5 Package's Parameters

The move_base package includes a series of parameters that aids its implementation per definition. Each file (4) impacted a specific functionality within the package and although some parameters could have been left out, it was decided to keep them just for the sake of analysis. As a starting point, consider benchmark model's configuration files included in Figs.4 and 5. Conversely, the personal model's configuration files are comprised in figures 6 and 7.

The reader would find interesting that many of these parameters were kept constant and only a limited amount were modified between the experiences.

4 Results

For both models, the localization task was achieved by running the AMCL package right after loading them in Rviz ®. To successfully reach the goal position, the *navigation_goal.cpp* script had to be ran once the robot had found itself in the map; a local map layer displaying ”cost” areas shown by colors (blue for low risk of finding an obstacle, cyan for high risk) determined the burden involved in reaching different positions along the way, while red and violet dots represented the obstacles detected by the laser and transmitted via the **scan** topic. The path planning task for the robot was thus performed by the move_base package after starting the script. For both benchmark and personal model, the *transform_tolerance* parameter (i.e., ”time with which to post-date the transform that is published of the AMCL package, to indicate that this transform is valid into the future” [8]) was kept at 0.3 seconds.

4.1 Benchmark Model

In terms of localization, the benchmark model was successfully localized by the AMCL package.

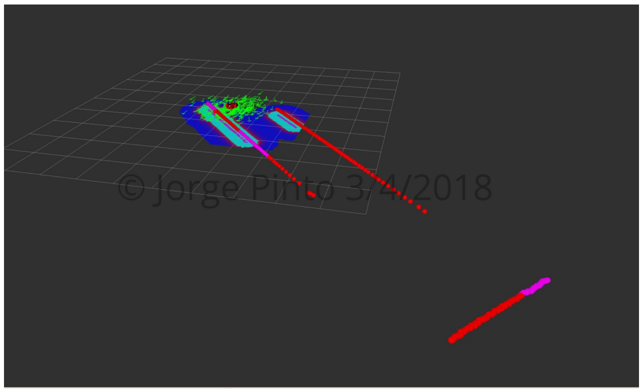


Fig. 8: PoseArray for Benchmark Model

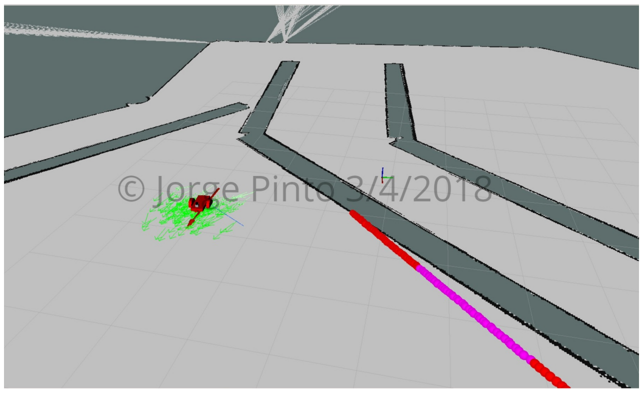


Fig. 9: Goal position for Benchmark Model

Parameter	Value
holonomic_robot	false
acc_lim_x	0.5
acc_lim_y	0.5
acc_lim_theta	0.5
max_vel_x	1.0
min_vel_x	0.5
max_vel_theta	1.0
min_vel_theta	-1.0
yaw_goal_tolerance	0.1
xy_goal_tolerance	0.2
sim_time	1.0
controller_frequency	10
sim_granularity	0.5
pdist_scale	4
gdist_scale	3
meter_scoring	true

(a) base_local_planner_params.yaml

Parameter	Value
map_type	costmap
obstacle_range	2.5
raytrace_range	3.0
transform_tolerance	0.3
robot_radius	0.3
inflation_radius	1.5
observation_sources	laser_scan_sensor
laser_scan_sensor	sensor_frame: hokuyo data_type: LaserScan topic: /udacity_bot/laser/scan marking: true clearing: true

(b) costmap_common_params.yaml

Fig. 4: Configuration Files for Benchmark Model (first half)

Parameter	Value
global_frame	odom
robot_base_frame	robot_footprint
update_frequency	1.0
publish_frequency	1.0
width	5.0
height	5.0
resolution	0.05
static_map	false
rolling_window	true

(a) local_costmap_params.yaml

Parameter	Value
global_frame	map
robot_base_frame	robot_footprint
update_frequency	1.0
width	30.0
height	30.0
resolution	0.05
static_map	true
rolling_window	false

(b) global_costmap_params.yaml

Fig. 5: Configuration Files for Benchmark Model (second half)

Fig.8 shows an array of such random positions before setting the goal position. The parameters set for the package included a minimum of 150 and maximum of 300 generated particles during the run. After the goal was set, the robot started to move neither smoothly, nor clumsily. The blue areas of the local map updated as the robot moved forward, while kept aligned with the calculated path. It did not take much time for the robot to reach the position defined by the *navigation_goal* script. Moreover, one could think that an oscillation around the goal could have happened due to accumulated errors but it was not the case. Fig.9 shows the orientation of the robot model after the *navigation_goal* script finished.

4.2 Personal Model

The personal model was also successfully localized by the AMCL package; Fig.10a shows the array of random positions generated by the algorithm before running the goal script as well as the local map of ”cost” areas surrounding the robot. The difference observed in random generated positions however comes from the different limits of particles set for the AMCL package. In this case, minimum and maximum numbers of particles were set to 50 and 100 respectively after some tuning attempts.

Setting the *navigation_goal* script provided the robot with a motion path (shown as a solid blue line in Fig.10b), calculated from the ”cost” areas displayed by the local map. In this case, the robot did move to the desired position, but it took a longer time than the benchmark’s. Moreover, while tuning the parameters shown in Figs.6 and 7 the robot sometimes lost its path to the goal. In this scenario, it had to reach a wall beforehand to recalculate its path. Nonetheless, the robot’s end orientation was not different from the one set by the script even though some oscillations were observed around the goal.

5 Discussion

Most of the differences observed are due to the simple fact that the models are constructively different. The size of the

Parameter	Value
holonomic_robot	false
acc_lim_x	0.5
acc_lim_y	0.5
acc_lim_theta	0.5
max_vel_x	0.7
min_vel_x	0.5
max_vel_theta	2.0
min_vel_theta	-2.0
yaw_goal_tolerance	0.1
xy_goal_tolerance	0.2
sim_time	1.5
controller_frequency	1
sim_granularity	0.8
pdist_scale	4
gdist_scale	3
meter_scoring	true

(a) base_local_planner_params.yaml

Parameter	Value
map_type	costmap
obstacle_range	2.5
raytrace_range	3.0
transform_tolerance	0.3
robot_radius	0.4
inflation_radius	0.7
observation_sources	laser_scan_sensor
laser_scan_sensor	sensor_frame: hokuyo
	data_type: LaserScan
	topic: /my_droid/laser/scan
	marking: true
	clearing: true

(b) costmap_common_params.yaml

Fig. 6: Configuration Files for Personal Model (first half)

Parameter	Value
global_frame	odom
robot_base_frame	robot_footprint
update_frequency	1.0
publish_frequency	1.0
width	5.0
height	5.0
resolution	0.05
static_map	false
rolling_window	true

(a) local_costmap_params.yaml

Parameter	Value
global_frame	map
robot_base_frame	robot_footprint
update_frequency	1.0
width	30.0
height	30.0
resolution	0.05
static_map	true
rolling_window	false

(b) global_costmap_params.yaml

Fig. 7: Configuration Files for Personal Model (second half)

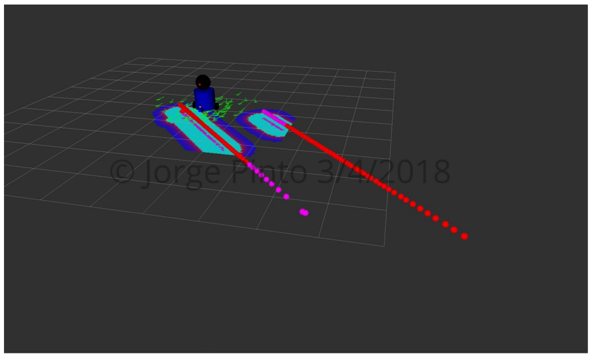
personal model plays an important role during the simulation in terms of available space for maneuvering, speed and attitude while on movement, and that is accountable to the longer time it took to reach the goal position.

In this sense of ideas, the benchmark model could be thought of having a performed better than the personal one, but it depends on the application they could have been intended to. For instance, assume that the task at hand is to carry trash from one point to another; there the ”trash can” would have performed better than the benchmark model in terms of possible capacity for payload. On the other hand, if the task would have been to only map the circuit shown in Fig.1, definitely the benchmark model is the winner in terms of time and motion behavior.

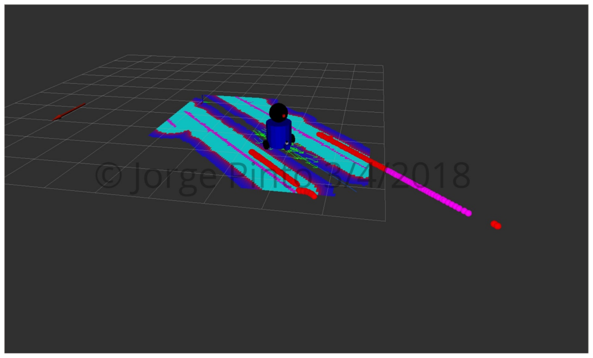
The sensor position plays also an important role when mapping the environment. It was thought at the beginning that the sensor could be placed at the top of the personal model's head. However, it proved to be inadequate due to the height the obstacles present in the map being lower. At such scenario, the robot would have not detected any wall whatsoever, making it very hard to reach the goal posed by *navigation_goal.cpp*.

Moreover, the PoseArray (or amount of generated particles) proved to affect the performance of both robots in the long run. For the personal model for instance, the limit of particles generated had to be modified as a high number of them produced uncertainty while in motion, which made the robot to lost the motion path generated by the move_base package. Hitting a wall did sort out the issue, but it turned out to be a poor solution in terms of performance.

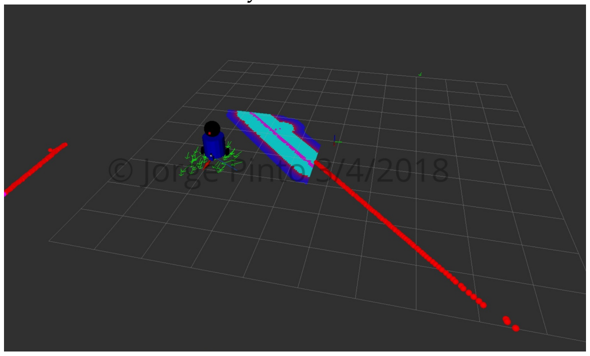
That been said, a situation like that of the ”Kidnapped Robot” could be sorted out by decreasing the number of particles generated in time. That's the main difference between MCL and AMCL as pointed out in [1]: the belief of the robot changes by time as approximate distributions with increased uncertainty are calculated, representing in turn the gradual loss of information in terms of position. Scenario and industry-wise whether MCL or AMCL is used depends entirely of the application for which these robots



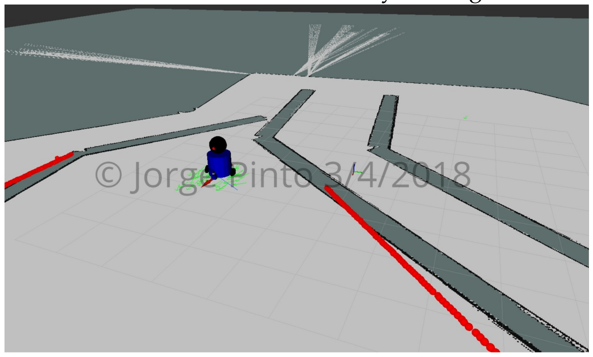
(a) PoseArray for Personal Model.



(b) Personal model on its way to the goal.



(c) Personal model at the goal.



(d) Personal model at the goal in map view

Fig. 10: Results gathered for the acquired data

are intended: a more precise localization task would require the latter, while the former could be used to simply map an unknown environment.

Therefore, the last affirmation is the reason to understand why the smaller benchmark robot performed better than the bigger trash can: the presence of slippage and drift on the biggest model due to its constructive properties while on movement (mass, inertial distribution, etc). As a matter of fact, parameters like robot_radius, inflation_radius, minimum and maximum speeds, sim_time, controller_frequency and sim_granularity were those modified to overcome the slippage and get the personal model in the right goal spot.

6 Conclusion / Future work

Two robot models were deployed to localize themselves in a preloaded map and reach a goal position provided by an external script. Different ROS packages were used: the AMCL and move_base packages were implemented to run in each model. Different results were gathered in terms of performance and time to reach to goal position. Constructively, the personal model turned out to be less attractive than the benchmark's: it would be to costly to produce such a robot given versus the poor results obtained. Industry-wise, any robot can be used for a localization task but the difference will reside in the uncertainty the operator is willing to accept during the deployment.

6.1 Recommendations for Improvement

To make the personal model attractive a good choice would be to make it smaller and therefore lighter. Drift and slippage from AMCL could be further reduced in consequence. Hardware wise, and independent and powerful processor would be necessary if more sensors are to be installed to the robot (if accuracy is of importance for the application or is height in to be detected in the map). For instance, in case a 3D localization task, much space would be required to compute the sets of incoming data.

References

[1] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, vol. 2, pp. 1322–1328, IEEE, 1999.

[2] D. Fox, W. Burgard, H. Kruppa, and S. Thrun, "A probabilistic approach to collaborative multi-robot localization," *Autonomous robots*, vol. 8, no. 3, pp. 325–344, 2000.

[3] D. Fox, S. Thrun, W. Burgard, and F. Dellaert, "Particle filters for mobile robot localization," in *Sequential Monte Carlo methods in practice*, pp. 401–428, Springer, 2001.

[4] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artificial intelligence*, vol. 128, no. 1-2, pp. 99–141, 2001.

[5] S. J. Julier and J. K. Uhlmann, "New extension of the kalman filter to nonlinear systems," in *Signal processing, sensor fusion, and target recognition VI*, vol. 3068, pp. 182–194, International Society for Optics and Photonics, 1997.

[6] D. Fox, W. Burgard, and S. Thrun, "Markov localization for mobile robots in dynamic environments," *Journal of Artificial Intelligence Research*, vol. 11, pp. 391–427, 1999.

[7] WikiROS, "Publishing odometry information over ros," 2016.

[8] WikiROS, "Amcl package summary," 2016.