Where Am I?

Humberto MartnezBarrn y Robles

Abstract—Any robotics task involving a mobile robot requires the agent to be able to infer where it is based on information given to it by the outside world (such as sensors and given maps). This paper explores the implementation of Adaptive Montecarlo Localization in a simulated environment in which a mobile robot had to navigate through a given map while learning to predict its pose using a particle filter. Another approach, the Linear Kalman Filter (KF) and the Extended Kalman Filter (EKF) will also be discussed, although they were not deployed in a simulated environment. Two different robot models were used to test the performance of the localization parameters.

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

1 Introduction

Ever since the earliest applications of robotics, localization has been a far from trivial problem. In the end, humans wanted robots like those in the science fiction books and movies. A *walking* C3PO, or a *moving* R2D2. It's not really that fun to have a robot that can't move. However, it was not enough to make the robot move on its own; it had to move like a human would - with knowledge of where it is and where it is going.

Thus, the Localization Problem was formulated. In order for R2D2 to move to a certain position, he would first need to know *where it is*. Therefore, sensors would need to be introduced. Now, the robot would be able to sense nearby objects and obstacles in order to better control its navigation. Nevertheless, it would still not know where it is! The solution? Different approaches have been taken; some of them include the popular and widely used particle filters and Kalman filters.

The goal of this project was to deploy a mobile robot in a simulated environment and have it learn where it was by moving around (initially, the robot has no clue where in the map it may be). The goal of the project was to accomplish this task using a particle filter provided by a ROS package called *AMCL* (specifically, Adaptive Montecarlo, which will be explained shortly). Thus, the project consisted of two parts: a) building the robot in a URDF file, and b) adjusting the necessary parameters to make it successfully navigate to a goal position and simultaneously locate itself.

1.1 Building the robot

In this part, the robot would have to be built in two files: a .xacro and a .gazebo file. Each of these files would describe the robot for Gazebo and Rviz in order to deploy it in the environment. It was important to build the robot correctly because if it wasn't, then it might not be able to move, much less localize itself. Three robots were built in total: the first one was provided by Udacity in the classroom, the second was a new two-wheeled robot, and the third was a new four-wheeled robot.

1.2 Parameter tuning

This part consisted on researching the move_base and amcl packages' parameters in order to find the values which best helped each robot localize itself. The only way to get familiar with said parameters was reading the ROS wiki on them and playing around with them, noticing any changes in the robot's behavior.

2 Background

Now it is time to explain both Kalman filters and particle filters. Each approach hopes to localize the robot by predicting its position after a measurement update and then taking the error into account to improve itself. The filter chosen was Adaptive Montecarlo Localization because particle filters are far more adaptive than Kalman filters. The adaptive version of Montecarlo Localization (which is the same as a particle filter) exists in order to solve a far more complicated problem: the *kidnapped robot* problem, in which the robot starts moving around a map, and then it is moved by another entity, ending up in a different part of the map.

Normal Kalman and particle filters would be lost in this problem, but AMCL was created to dynamically adjust the number of particles in order to scatter fresh particles around the map every now and then, helping the filter stay up to date in case the robot gets moved around. Other approaches have been taken, and this is still a field of study, in which new things are being invented that will make this problem easier to solve.

2.1 Kalman Filters

Kalman filters are named after Rudolph E Kalman, a Hungarian scientist who proposed a formula to estimate the Apollo's trajectory. Since then, the filter has evolved and been used by the AI and robotics community frequently.

Nowadays, there's more than one type of Kalman filter. For example, the original Kalman filter (now called Linear Kalman Filter or LKF) is a great way to predict future

poses for moving objects and it is used in all kinds of autonomous vehicles to predict the trajectory of moving entities (such as cars, people, and other autonomous robots). In order to make these predictions, the LKF predicts the future position of the robot as a Gaussian with the mean representing where it thinks the robot is most likely to be and an uncertainty represented by the covariance. Then, a sensor measurement takes place, updating the robot's pose and comparing it with the prediction. The result of this comparison is a new prediction function, which is far more certain than both the measurement and the previous guess about the robot's current and future pose. However, this filter is limited to linear observation and transition models and Gaussian uncertainty distributions, so as soon as the model becomes nonlinear the filter is no good anymore.

On the other hand, Extended Kalman filters or EKF's are a nonlinear version of the LKF, and works by taking the exact same steps, with one exception: the transition and observation models need to be differentiable - no longer linear. Thus, the EKF uses linear approximation to fit a nonlinear model. This filter, however, is still only good for Gaussian distributions of uncertainty.

2.2 Particle Filters

Particle filters use a discrete approximation of the robot's position, which contrasts with the KF's continuous approach. A particle filter scatters *particles* across the map at random positions and orientations, each with an equal *weight*. Each of these particles represents a virtual "robot", and therefore a possibility of the robot's position and orientation.

As the robot moves and measures its distance to certain landmarks on the map, the weight of each particle is updated based on how close (or far) the measured distance for each particle are to the sensor's information. Then, a *resampling* step takes place, where the same number of particles as the original are selected at random, although the particles with largest weights (and therefore the ones that are likely closest to the robot's pose) are more likely to be picked than the ones with smaller weights. Thus, after several movements, the particles converge to pretty much a single pose and orientation that fairly represents the robot's pose.

2.3 Comparison / Contrast

While particle filters need nice Gaussian uncertainty distributions (and preferably a linear model) in order to work properly, particle filters can adapt to any kind of distribution by making the problem discrete. Thus, each particle will hold a weight in order to infer the robot's position and orientation. However, particle filters require additional computation, and might end up performing far more operations than the KF. Both can achieve a result, but each also has advantages and disadvantages. Specifically, the KF's greatest limitation is that not all models fit it. This is why the particle filter was chosen for the project. Though more computation might be necessary, the particle filter is guaranteed to work on almost every kind of distribution given the right parameters (such as number of particles resampling periods, and, for the adaptive case, the frequency with which new particles will be scattered across the map).

3 Simulations

During the simulation, the robots performed remarkably well after parameters were tuned (a quite nontrivial task). However, the third robot (a four-wheeled model) did not behave nearly as well as the rest.

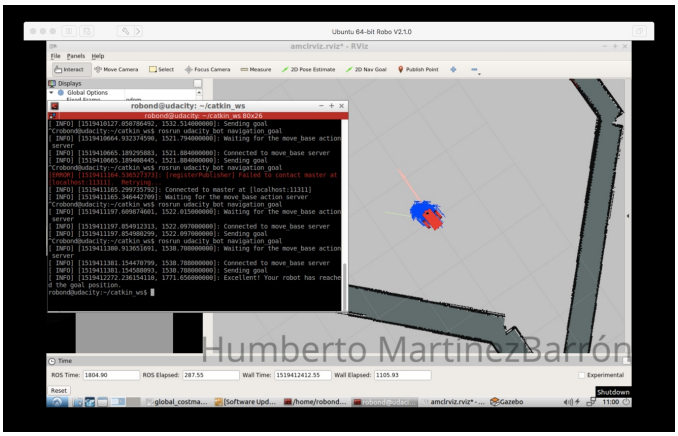

Fig. 1. The benchmark model at the target location.

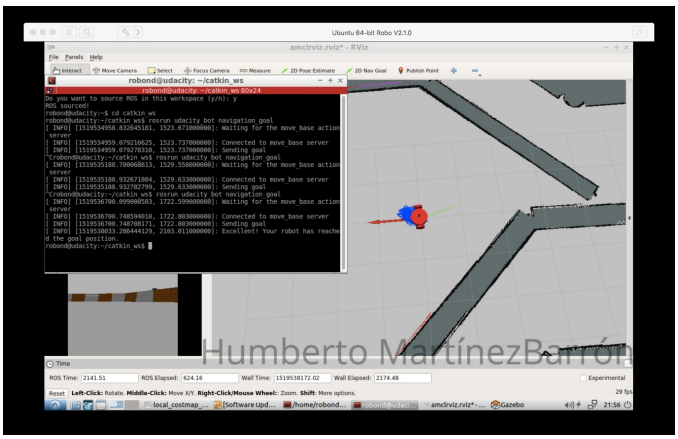

Fig. 2. The first personal model at the target location.

3.1 Achievements

The benchmark model, by the time all parameters had been tuned, was able to navigate to the goal position with a small yaw and xy error. Also, the localization results were the expected ones; the particles had converged to the robot's true position by the time it had reached the goal.

On the other hand, the first modified model was able to navigate to the goal and achieve a good localization result with almost the exact same parameters as the benchmark robot.

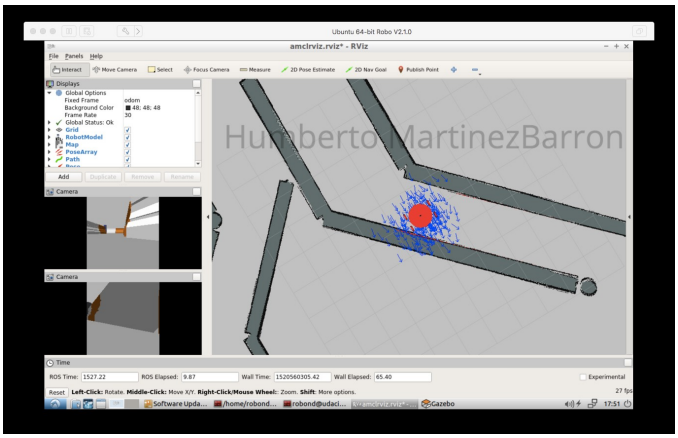

Fig. 3. The second personal model at the start location.

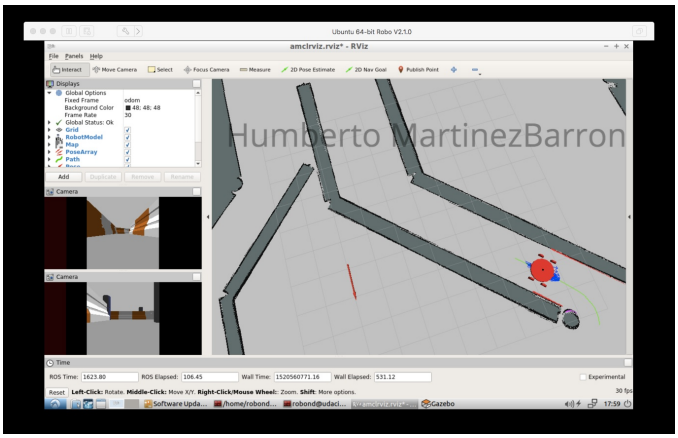

Fig. 4. The second personal model at the corridor.

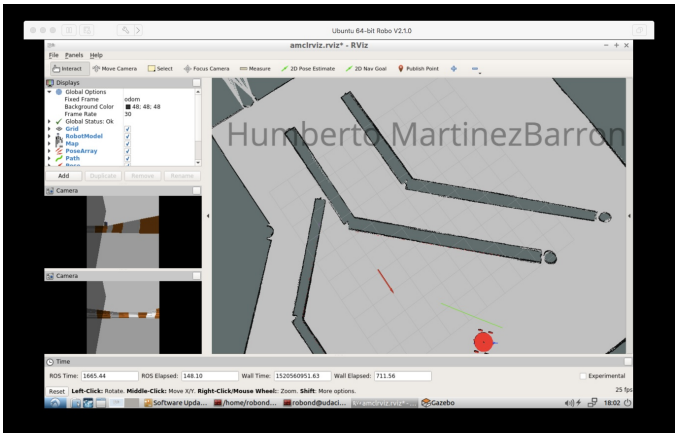

Fig. 5. The second personal model outside of the corridor.

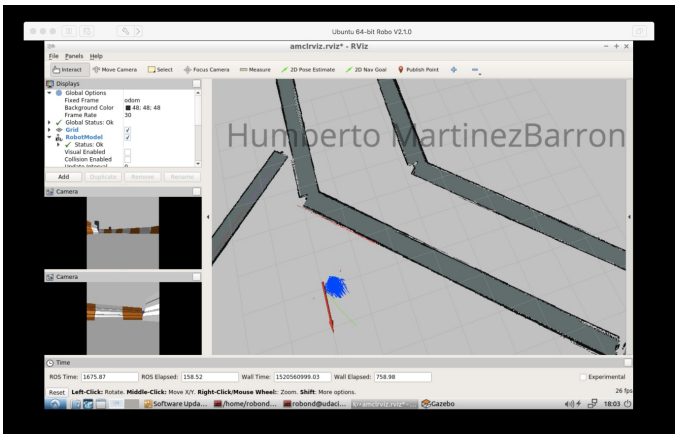

Fig. 6. The particle distribution for the second personal model.

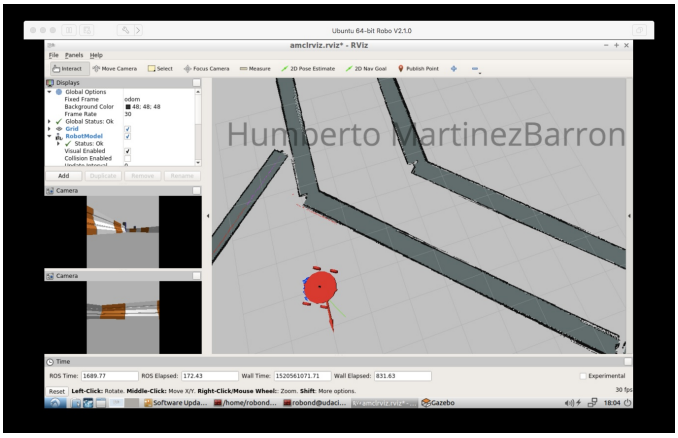

Fig. 7. The particle distribution for the second personal model at the xy goal.

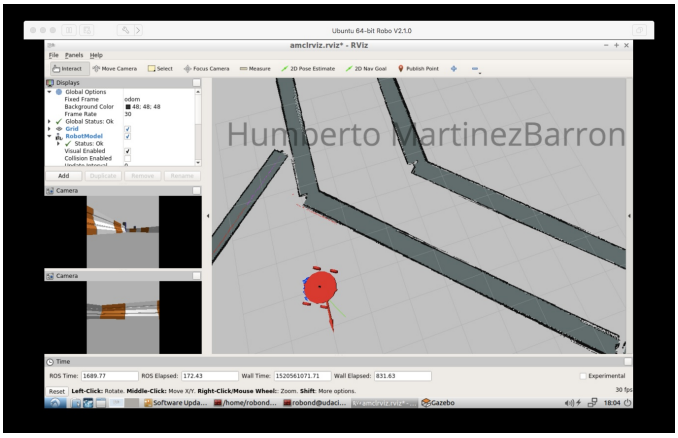

Fig. 8. The second personal model at the xy goal.

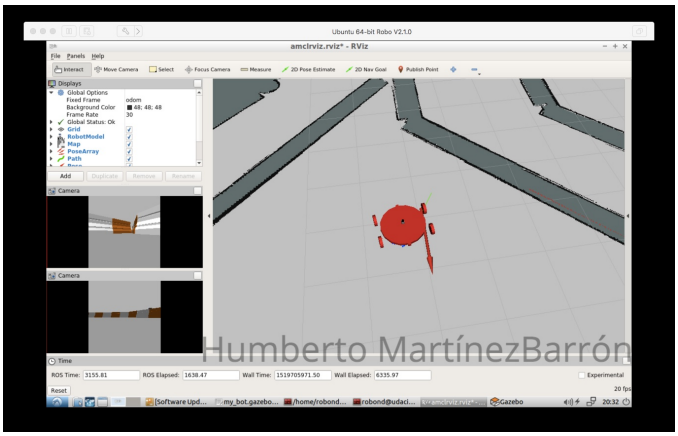

Fig. 9. The second personal model at the target location.

Finally, the second modified model (consisting of a four-wheeled robot with two cameras) was able to localize itself rather quickly and it reached the xy goal position without any trouble. However, it was unable to achieve the yaw goal.

3.2 Benchmark Model

3.2.1 Model design

The benchmark robot model consists on a rectangular prism for a mobile base with dimensions 0.4 length, 0.2 width, and 0.1 height. The two wheels are made up of cylinders of radius 0.1 and length 0.05 with the hinges located on the y axis at 0.15 and -0.15. Also, it has two spherical casters (front and back) with a 0.05 radius. The camera is placed on the robot's "front face", and the hokuyo laser sensor is located at 0.15 along the x axis, 0 along the y axis, and 0.1 along the z axis.

3.2.2 Packages Used

The robot used the move_base and amcl packages, with the move_base subscribed to the laser_scan topic to receive the sensor's readings. Also, an odometry package was used.

3.2.3 Parameters

The localization parameters used are shown in table 1.

The value for controller_frequency was chosen to avoid getting too many "Control loop missed desired rate of 20.00000 Hz. The actual loop took (some number)". The value for minimum and maximum particles was chosen because a larger number could result in too many computations in order to make a prediction (more than necessary), while a smaller number might not perform the localization task correctly. The value for the likelihood maximum distance determines what is the maximum distance to "inflate" an obstacle (and therefore, it determines by how much the robot will try to stay away from possible obstacles in the map). The value 2.5 was chosen to make it stay safe and clear from any possible collisions. The recovery alpha values indicate an exponential decay for the frequency with which the robot will scatter new particles along the map. The values chosen were recommended by the ROS wiki, and worked fine. The kid_err parameter defines what the maximum error can be between the estimation and the ground truth. A smaller value will allow less error.

TABLE 1
AMCL Parameters

odom_frame_id	odom
odom_model_type	diff-corrected
base_frame_id	robot_footprint
global_frame_id	map
laser_model_type	likelihood_field
controller_frequency	1.0
min_particles	50
max_particles	250
laser_likelihood_max_dist	2.5
recovery_alpha_slow	0.001
recovery_alpha_fast	0.1
kid_err	0.0025
odom_alpha1	0.005
odom_alpha2	0.005
odom_alpha3	0.010
odom_alpha4	0.005
odom_alpha5	0.003

TABLE 2
Benchmark move_base Parameters

holonomic_robot	false
meter_scoring	true
pdist_scale	3.5
sim_time	1.0
occdist_scale	2.5
xy_goal_tolerance	0.05
yaw_goal_tolerance	0.025
obstacle_range	3.5
raytrace_range	3.0
transform_tolerance	0.2
robot_radius	0.4
inflation_radius	0.4
observation_sources	laser_scan_model

The parameters specified in table 2 are justifiable by noticing the robot is not holonomic, since it cannot move diagonally. The terminal recommended to set meter_scoring to "true" to avoid getting a warning. pdist_scale was set to 3.5 to make sure the robot follows the planned path, but it need not stick to it completely. sim_time was set to 1.0 to stick to the default value. occdist_scale was set to 2.5 to avoid collisions as much as possible. The xy and yaw goal tolerances were set to small values to make sure the robot accomplished the required specifications for the project. The obstacle and raytrace ranges were set to their corresponding values to take obstacles at dangerous distances into account for the costmap. The transform tolerance worked fine as long as it was larger than zero, and for hardware limitations it was not to be set too high. The robot and inflation radius were set to 0.4 because the robot's length is 0.4.

3.3 Personal Model (1)

3.3.1 Model design

The first personal model design consisted on a modified version of the benchmarked bot. The base is now a cylinder, and the hokuyo sensor is placed at the center of the base.

The camera is at the same spot as the last. The base has a radius of 0.2 and a length of 0.1. The wheels have the same radius, length, yaw, pitch, and yaw. However, their hinged are at 0.25 and -0.25 along the y axis.

3.3.2 Packages Used

The packages used for the personal model design were the exact same as those used for the benchmark model.

3.3.3 Parameters

The AMCL parameters are the exact same as those used for the benchmark bot, since the AMCL package worked perfectly, the changes had to be made to the move_base parameters, not the AMCL parameters. Table 3 shows the parameters used for the personal robot's move_base that are different from the ones used for the benchmark robot.

TABLE 3
Personal move_base Parameters (1)

pdist_scale	2.5
sim_time	2.0
occdist_scale	3.0
obstacle_range	3.0
raytrace_range	2.5
robot_radius	0.45
inflation_radius	0.45

pdist_scale was reduced to prove that the robot could reach the target location with an even lower attachment to the navigation plan. sim_time was increased to test whether a larger number would speed up the simulation process - if it did, it was not noticeable. occdist_scale had to change because a robot with a cylindrical base had more trouble turning on its own axis than the benchmark robot, so it would need to do more to avoid obstacles. Finally, obstacle and raytrace ranges were decreased to test whether the previous parameter values had been larger than necessary (they were).

3.4 Personal Model (2)

3.4.1 Model design

The second personal model consisted on a disk for a base (a cylinder with a 0.05 length and 0.4 radius), four wheels with the same radius, length, roll, pitch, and yaw as the benchmark robot, although their hinges were now placed at (0.225, 0.45, 0), (-0.225, 0.45, 0), (0.225, -0.45, 0), and (-0.225, -0.45, 0). The hokuyo joint was now placed at (0, 0, 0.1). Also, two cameras were used instead of one. One was placed at the front of the robot (0.4, 0, 0), and the back camera was placed at (-0.4, 0, 0) with a 3.1415 yaw. Mass values were also largely decreased in order to attempt to make the robot move more quickly (which seemed to work).

3.4.2 Packages

The same packages were used for this model as for the benchmarked model design.

TABLE 4
Personal move_base Parameters (2)

pdist_scale	5.0
sim_time	2.0
occdist_scale	3.0
obstacle_range	5.0
raytrace_range	2.0
transform_tolerance	0.4
robot_radius	0.6
inflation_radius	0.6

3.4.3 Parameters

The parameters that changed from the benchmarked model's configuration are shown in table 4.

pdist_scale was brought up to 5.0 to attempt at making the robot stick to the navigation plan as much as possible. Again, sim_time had no apparent effect on the result, it was left from the previous parameter configuration (Personal model 1 - see table 3). occdist_scale was set at 3.0 to make the robot try to avoid collisions. xy and yaw goal tolerances were unchanged in order to attempt to get the same navigation results for this model. The obstacle and raytrace ranges were changed in order to help the robot avoid collisions. Transform tolerance was set to 0.4 as another attempt to make the robot stick to the plan. Finally, the robot and inflation radius were selected thus in order to make the robot take into account that it is now bulkier and must therefore plan the navigation route to go through places it can fit in.

4 Results

4.1 Localization Results

4.1.1 Benchmark

The provided robot model achieved the desired results after considerable parameter tuning. At first, the main problem was the navigation package; the height and width of the local and global costmaps seemed to play a major role in the robot's navigation capabilities. After the size of the costmap exceeded a certain value (approximately 10.0), the navigation stack would no longer work. After getting the robot to be able to plan its route and navigate towards it, the new problem would be that the path plan would try to make the robot go between the wall and a column in the map. To fix this, the robot and inflation radius were set to a larger value in order to make the robot take into account its own size before attempting to fit through places it simply could not go through.

After tuning the parameters necessary for a decent navigation, it was now necessary to tune the parameters for the AMCL package. However, none of the parameters explained and available in the ROS wiki seemed to explain why the localization results were so poor. As it turns out, the diff-corrected odometry model needs tuning of specific parameters in order to improve its performance. Thus, these parameters were researched in order to understand how they could affect the performance of the AMCL package (see table 3's odom_alpha 1 -

5 values). A template for the parameters used was found in a github repository authored by user hershwg (https://github.com/ros-planning/navigation/blob/jade-devel/amcl/test/small_loop_crazy_driving_prg_corrected.xml). The exact same alpha values were used for the AMCL package, and a satisfactory result was achieved with these values. The arrows converged to practically a single location with a uniform orientation approximately one movement after exiting the corridor while following a smooth path with no unexpected behavior.

4.1.2 Student

The first personal robot (named my_bot1) achieved a great localization and navigation result. The arrows had converged to a single point with a fairly uniform orientation after the robot left the corridor (inside the corridor, the distance to the landmarks would be fairly equal for several poses of the robot within the corridor, but once it began turning, the arrows converged to a single pose in two movements) while following a smooth path with no unexpected behavior. The same AMCL parameters were used. Nevertheless, it was necessary to change the occ_dist, obstacle_range, and raytrace_range parameters. The robot and inflation radius were changed to fit the new dimensions of the robot's cylindrical base.

The second personal robot (named my_bot) also achieved very good localization results. However, navigation was far poorer - even with a pdist_scale value set to 5.0, the robot was unable to stick to the navigation plan. Nevertheless, the localization package worked rather well; the arrows converged once the robot was out of the corridor in only a few moves (two to three moves), with the only unexpected behavior being the lack of adherence to the path plan. The path the robot followed was still smooth, but inconsistent with the plan.

4.2 Technical Comparison

The first personal robot was expected to achieve similar results as the benchmark robot, since only minor design modifications were made. The results show that such changes do not have a transcendent effect on the robot's localization performance. Also, it seemed that a change in the position of the sensor (no longer at the front of the robot's "body", but at center of it) did not have any major effects.

On the other hand, when more wheels were added (my_bot), localization continued to work acceptably while navigation fell apart at the end (note that the robot could still reach the xy goal position, but it would never reach the yaw goal). Thus, the change from using only two wheels to using four makes a huge difference in how the robot is going to be driven. Nevertheless, it seemed once the AMCL parameters were fine-tuned, they would work on different robot models. Also, different positions and heights were experimented for the hokuyo joint. First, at the back, but this position was unable to provide the bot with enough information about its surroundings in order to achieve successful navigation. Then, at the center (ground level), where the sensor took too much of the wheels of the robot into account (the cylindrical base is very thin, so if the hokuyo is placed too low, it will count the wheels as obstacles and think it is eternally stuck). Also, if the sensor was placed too far to the front or back, it might miss very important information about its surroundings, which would affect its performance considerably, given the my_bot model was wider than the others. This is why the hokuyo joint is located at the center of the base, at a height of 0.1.

5 Discussion

The benchmark model worked well because of well-tuned AMCL parameters. The odom_alpha's resulted crucial in order to achieve the desired results. The values used by hershwg's model worked great for all three models. Perhaps they were so finely tuned they were able to apply to several types of models with a certain structure.

The first personal model (my_bot1) achieved the desirable results for the same reasons as the benchmark model did; since its structure was almost the same, the AMCL package was expected to (and it did) perform similarly on both robot models.

The second personal model, however (my_bot), performed poorly when it came to navigation, but satisfyingly when it came to localization. This result is due to the drive plugin used (according to user grandia in the robotics slack community). Thus, a solution for the poor performance might have been finding a plugin to drive a four-wheeled robot rather than try to force a two-wheel driving plugin to work for a different kind of robot.

5.1 Topics

- Which robot performed better?
  The benchmark model definitely performed better than the other two models, although the difference is not as noticeable (especially when talking about localization). Still, the second personal robot did perform noticeably worse as far as navigation goes.
- Why did it perform better? (opinion)
  Perhaps it performed better because the four-wheeled robot did not have a plugin for driving itself correctly. Still, the benchmark model performed better than the first personal model because the parameters and structure had been thought through more carefully, making it easier for it to drive along the map.
- How would the 'Kidnapped Robot' problem be approached?
  For the 'kidnapped robot' problem, perhaps the key parameters to tune would be the recovery_alpha values because the robot would need to spread new particles on the map in order to figure out where it has been 'kidnapped' to. These two parameters represent an exponential decay for when the new particles will be scattered to recover. Thus, the better tuned this parameter is, the better the robot will know when to produce new particles in order to localize itself after being kidnapped.

- In what types of scenario could localization be performed?
  When a map is already known and non-mobile landmarks have been saved to the map with coordinates, the localization problem can be solved with any of the two filters explained in this report.
- Where would MCL/AMCL be used in an industry domain?
  MCL might be best applied when there is no risk of losing important information during the resampling step of the particle filter. For example, perhaps MCL would be used where the kidnapped robot scenario is extremely unlikely (such as warehouse robots who are left to operate on their own, never being moved by humans), or where a convergence limit tells the robot when it knows its pose well enough in order to carry out its tasks, such as am autonomous car (as long as it is *left alone* and the map is known).

  On the other hand, AMCL might be the best option when it is uncertain whether or not the robot will be moved by humans or other entities. For example, home robots might have better chances to not get lost in a home with AMCL. Also, an autonomous exploration bot might be better off using AMCL to look for an object in an environment where it interacts with animals or other beings that might move (or 'kidnap') it.

6 Conclusion / Future work

During this project, three different robot models were tested in a gazebo environment while performing AMCL: a two-wheeled benchmark model and two personal models: a two-wheeled model and a four-wheeled model. The benchmark model was able to localize itself fairly well; making the arrows converge after a few moves after exiting the corridor. The two-wheeled personal model achieved very similar results to those observed for the benchmark model. Finally, the four-wheeled robot was unable to reach the target yaw, although it did reach the xy position with a fairly good localization result.

To deploy this project on hardware, it would be necessary to adjust the parameters according to real-world sensor readings and complications. A laser sensor might be useful, although an RGBD camera might also work. The AMCL package and ROS nodes connected to each of the moving parts would need to be downloaded onto the motherboard.

Industrial applications for the topics explored throughout this report include home service robots, autonomous vehicles (cars and maybe planes - where a three-dimensional particle filter would need to be implemented, and landmarks would need to be chosen given a map and sensors that can detect them), humanoid robots working in places with known maps, etc.

Another point to consider would be the tradeoff between accuracy and processing time. Since it is important for the robot to actually move and not take forever thinking, sometimes accuracy can be sacrificed to make the processing take less time. In the example of the second personal robot model, this tradeoff was not taken into account too carefully. Therefore, the robot takes a rather long time to start moving, but once it does, it will do its best to stay as close to the path as possible even with the limitations of a two-wheeled drive plugin.

6.1 Modifications for Improvement

- Sensor type
  Perhaps using an RGBD camera might be easier (or at least cheaper) to implement than a hokuyo laser sensor, so therefore trying to simulate the same environment with this type of sensor might help give an idea of how an easier-to-build robot might behave.
- Sensor Location
  If the hokuyo was placed at the front or back of the robot, it would sometimes miss some details of the map. However, depending on the base's geometry and the robot's size, perhaps some of those details might become reasonably expendable.
- Driving plugins
  A four-wheeled robot drive plugin might have helped my_bot model drive itself better and reach the target yaw, although it would take some time to find the correct one (or maybe even code it).
- Sensor Amount
  An attempt to place a laser sensor on the back of the robot and another at the front failed due to poor coordination with the rest of the active ROS topics. Nevertheless, perhaps more sensors can help have a better understanding of the details of the map surrounding the bot and therefore lead to a better and smoother path planning. Also, in real-life hardware implementations, perhaps that additional information could mark a major difference in how the robot drives itself around the map.
- 3D localization
  According to Sebastian Thrun (Artificial Intelligence for Robotics, Udacity), multi-dimensional spaces will cause a particle filter to scale exponentially (as well as for Histogram filters, a topic not covered during this project). Therefore, the recommendation for multi-dimensional spaces is implementing a Kalman filter. However, if a non-unimodal distribution is encountered when performing 3D localization, a Kalman filter might not be able to adapt to the model. Therefore, the computationally expensive 3D particle filter might need to be implemented - the only way to know it to try it!

6.2 Hardware Deployment

1) What would need to be done?
   In order to deploy this AMCL implementation onto a real hardware, the sensors would first need to be tuned and a way to make them communicate with the ROS packages would need to be found (as well as a way to make the ROS packages communicate with the moving parts of the robot).

2) Computation time/resource considerations?
Also, AMCL implementation on hardware might work fine for few dimensions, although once more dimensions are introduced, the computation expense starts to scale exponentially. Thus, for multi-dimensional problems it will be better to try to adapt the problem to a unimodal distribution in order to implement a Kalman filter (which scales linearly).