Mobile Robotics Using Adaptive Monte Carlo Localization in ROS

Martin Bufi

Abstract—This paper describes the foundation about localization for mobile robotics. It then covers the creation and testing of two robot simulations in a ROS (Robot Operating system) simulation environment using Gazebo and RViz. Both robots use Adaptive Monte Carlo Localization (ACML) techniques combined with one of the offical ROS navigation plugins to successfully navigate a maze provided by Clearpath Robotics, and reach a predefined goal position. The two robots are then compared for efficiency in reaching the end goal with some future improvements to keep in mind.

Index Terms—Robot, IEEEtran, Mobile Robotics, Kalman Filters, Extended Kalman filters, Particle filters, Localization.

1 Introduction

Localization for mobile robotics is an emerging field with active research in improving techniques we have currently mastered. The idea of "Localization", means to determine a good approximation of the current position of the robot using a multitude of sensors with given uncertainties of each due to noise, such as a camera/Lidar (Light detection and Ranging) or uncertainties due to imperfect actuators/encoder counts when moving the robot

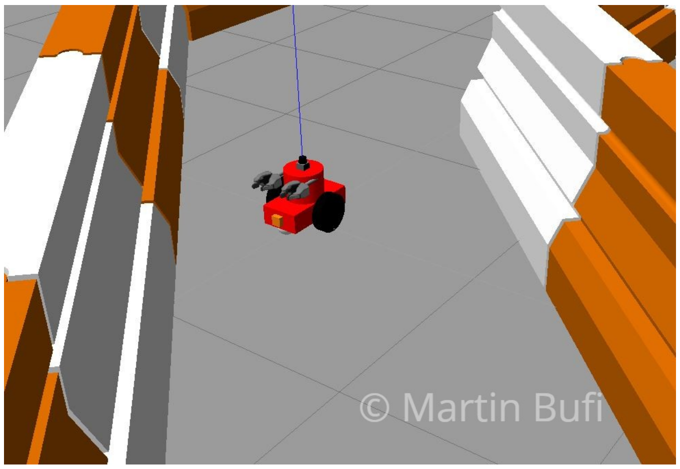


Fig. 1. MartinBOT traversing the maze

In this project, two robots were developed and then further tested in a simulation environment modeled after the Clearpath Robotics Jackal maze, as can be seen in the world definition file. Both robots successfully navigated the maze using Adaptive Monte Carlo localization (ACML). The benchmark definition of the first Robot, "*Udacitybot*", was provided as part of the initial process in the project, while the second robot "*Martinbot*", was created independently. The Martinbot uses the Willow Garage[1] PR2 Gripper URDF definition as described in the ROS.org Wiki[2].

The source files for this project can be found in github: https://github.com/mbufi/ROS_Localization_Project

2 Background / Formulation

When developing any sort of land, air, marine robot, Localization is a fundamental issue. In the non-ideal world we live in where sensors are inaccurate/noisy and actuators are imprecise, localization becomes a very difficult problem yet important problem to overcome when determining the location of a robot.

The two most common approaches to Localization are Kalman Filters or more advanced Extended-Kalman filters, and using particle filters.

2.1 Kalman Filters

Kalman filtering, also known as linear quadratic estimation (LQE), is very prevalent in current control system applications after solving the non-linear problem of trajectory estimation for the Apollo program. With the appropriate amount of optimization, it has proven to be very good at taking in noisy measurements in real time and providing relatively accurate predictions, which in the case of localization would be position.

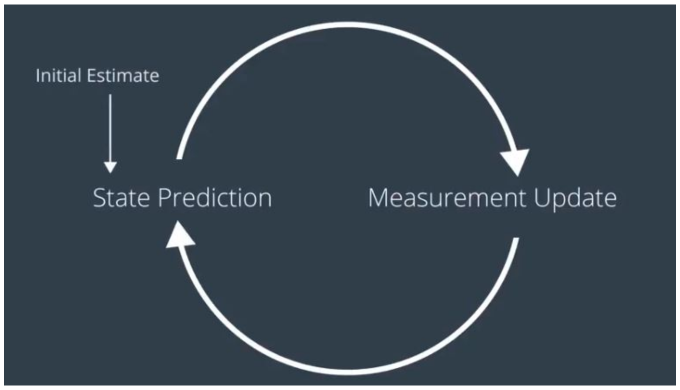


Fig. 2. Kalman filter in Action [2]

These are the equations that are implement the Kalman Filter in multiple dimensions:

$$x' = Fx$$

$$P' = FPF^T + Q$$

$$y = zHx'$$

$$S = HP'H^T + R$$

$$K = P'H^TS^{-1}$$

$$x = x' + Ky$$

$$P = (I - KH)P'$$

The Kalman Filter can recover successfully from inaccurate estimates, but it is very important to estimate the noise parameters as accurately as possible - as they are used to determine which of the estimate or the measurement to believe more. Overall, the filter is based on the following two assumptions:

- State space can be represented by a unimodal Gaussian distribution.
- Motion and measurement models are linear.

These two assumptions provide practical limits to modern day robotic applications, due to the nature of non-linear readings and behaviors found in both the hardware and environment. Therefore, Extended Kalman Filters (EKF) were introduced to address these issues.

2.2 Extended Kalman Filters

The EKF simply addresses those concerns by using the multi-dimensional Taylor Series to linearize a non-linear motion or measurement function. The equation for a multi-dimensional Taylor Series is presented below.

$$T(x) = f(a) + (x-a)^T Df(a) + \frac{1}{2!}(x-a)^T D^2 f(a)(x-a)$$

You will see that it is very similar to the 1-dimensional Taylor Series. As before, to calculate a linear approximation, we only need the first two terms.

$$T(x) = f(a) + (x-a)^T Df(a)$$

You may notice a new term, Df(a). This is the Jacobian matrix, and it holds the partial derivative terms for the multi-dimensional equation.

$$Df(a) = \frac{\delta f(a)}{\delta x}$$

From here, the KF equations are then modified slightly to use the partial derivative terms. The rest of the math is not in the scope of this paper. (Please refer to other publications for the full math behind this)

While the EKF algorithm addresses the limitations of the Kalman Filter, the mathematics is relatively complex and the computation uses considerable CPU resources making it somewhat unsuitable to perform on a small embedded processor.

$$\cancel{x' = Fx} \quad \rightarrow \quad x' = f(x)$$

$$P' = FPF^T + Q$$

$$\cancel{y = z - Hx'} \quad \rightarrow \quad y = z - h(x')$$

$$S = HP'H^T + R$$

$$K = P'H^TS^{-1}$$

$$x = x' + Ky$$

$$P = (I - KH)P'$$

Fig. 3. Highlighted in blue are the Jacobians that replaced the measurement and state transition functions.

2.3 Particle Filters

The ultimate goal of Monte Carlo pose estimation (MCL) is to determine where the robot is located in a given environment. That is, we must get x, y, and Θ of the robot on the map. For this purpose, MCL calculates the probability that the robot can be located. The powerful Monte Carlo localization algorithm estimates the posterior distribution of a robots position and orientation based on sensory information. This process is known as a recursive Bayes filter.

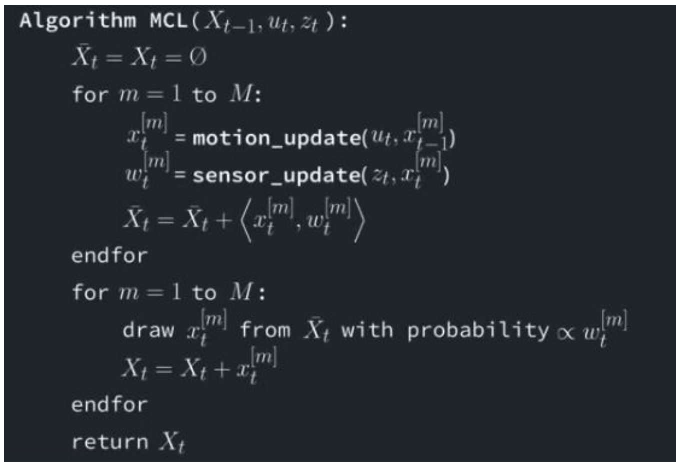
Using a **Bayes filtering** approach, roboticists can estimate the state of a dynamical system from sensor measurements.

In mobile robot localization, its important to be acquainted with the following definitions:

- **Dynamical** system: The mobile robot and its environment
- **State**: The robots pose, including its position and orientation.
- **Measurements**: Perception data(e.g. laser scanners) and odometry data(e.g. rotary encoders)

The goal of Bayes filtering is to estimate a probability density over the state space conditioned on the measurements. The probability density, also known as posterior is called the **belief**.

Therefore, particle filters begin operation by distributing particles throughout a known map and then removing those that are least likely represent the current position of the robot. It does this through the following algorithm:



```
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($z_t, x_t^{[m]}$)
        $\bar{X}_t = \bar{X}_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
    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 $X_t$
```

Fig. 4. Monte Carlo Localization [3]

1) **Initialization**: Since the robots initial pose (position, orientation) is unknown, the particles are randomly arranged within the range where the pose can be obtained with N particles. Each of the initial particle weighs 1/N, and the sum of the weight of particles is 1. N is empirically determined, usually in the hundreds. If the initial position is known, particles are placed near the robot.
2) **Prediction**: Based on the system model describing the motion of the robot, it moves each particle as the amount of observed movement with odometry information and noise.
3) **Update**: Based on the measured sensor information, the probability of each particle is calculated and the weight value of each particle is updated based on the calculated probability.
4) **Pose Estimation**: The position, orientation, and weight of all particles are used to calculate the average weight, median value, and the maximum weight value for estimating pose of the robot.
5) **Resampling**: The step of generating new particles is to remove the less weighed particles and to create new particles that inherit the pose information of the weighted particles. Here, the number of particles N must be maintained.

Computationally, this filter is much more efficient than the Kalman Filter. As well, Monte Carlo Localization is not subject to the limiting assumptions of Kalman Filters as outlined above; *a unimodal Gaussian probability distribution and linear models of measurement and actuation*. It is actually Multimodal Discrete, meaning it does not confine to a specific probability distribution. These differences are labeled in the table below:

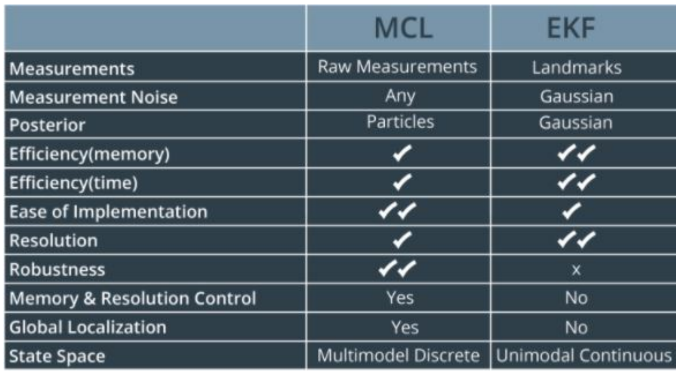
Therefore, Monte Carlo Localization is used in this project due to its efficiency and non-limiting factors.

3 ROS Navigation Stack

3.1 Overview

The main aim of the ROS navigation package is to move a robot from the start position to the goal position, without making any collision with the environment. The ROS Navigation package comes with an implementation of several navigation related algorithms which can easily help implement autonomous navigation in the mobile robots.

The user only needs to feed the goal position of the robot and the robot odometry data from sensors such as wheel encoders, IMU, and GPS, along with other sensor data streams such as laser scanner data or 3D point cloud from sensors like Kinect. The output of the Navigation package will be the velocity commands which will drive the robot to the given goal position. The Navigation stack contains implementation of the standard algorithms, such as SLAM, A*(star), Dijkstra, AMCL, and so on, which can directly be used in the application.



	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency(memory)	✓	✓✓
Efficiency(time)	✓	✓✓
Ease of Implementation	✓✓	✓
Resolution	✓	✓✓
Robustness	✓✓	x
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete	Unimodal Continuous

Fig. 5. MCL vs. EKF [3]

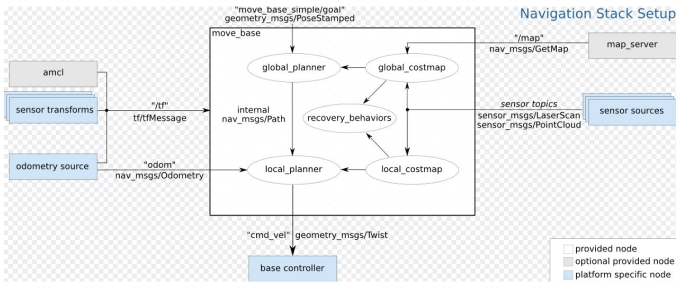


Fig. 6. ROS Navigation Stack: Relationship between essential nodes and topics on the navigation packages configuration

http://wiki.ros.org/move_base

The move_base node is from package called move_base. The main function of this package is to move a robot from its current position to a goal position with the help of other Navigation nodes. The move_base node inside this package links the global-planner and the local-planner for the path planning, connecting to the rotate-recovery package if the robot is stuck in some obstacle and connecting global costmap and local costmap for getting the map.

The move_base node is essentially an implementation of SimpleActionServer which takes a goal pose/location with message type (geometry_msgs/PoseStamped). We can send a goal position to this node using a SimpleActionClient node. The move_base node subscribes the goal from a topic called move_base_simple/goal, which is the input of the Navigation stack, as shown in the previous diagram.

When this node receives a goal pose, it links to components such as global_planner, local_planner, recovery_behavior, global_costmap, and local_costmap, generates the output which is the command velocity (geometry_msgs/Twist), and sends to the base controller for moving the robot for achieving the goal pose.

3.2 Packages linked to Move_Base

- **global-planner**: This package provides libraries and nodes for planning the optimum path from the current position of the robot to the goal position, with respect to the robot map. This package has implementation of path finding algorithms such as A*, Dijkstra, and so on for finding the shortest path from the current robot position to the goal position.
- **local-planner**: The main function of this package is to navigate the robot in a section of the global path planned using the global planner. The local planner will take the odometry and sensor reading, and send an appropriate velocity command to the robot controller for completing a segment of the global path plan. The base local planner package is the implementation of the trajectory rollout and dynamic window algorithms.
- **rotate-recovery**: This package helps the robot to recover from a local obstacle by performing a 360 degree rotation.
- **clear-costmap-recovery**: This package is also for recovering from a local obstacle by clearing the costmap by reverting the current costmap used by the Navigation stack to the static map.
- **costmap-2D**: The main use of this package is to map the robot environment. Robot can only plan a path with respect to a map. In ROS, we create 2D or 3D occupancy grid maps, which is a representation of the environment in a grid of cells. Each cell has a probability value which indicates whether the cell is occupied or not. The costmap-2D package can build the grid map of the environment by subscribing sensor values of the laser scan or point cloud and also the odometry values. There are global cost maps for global navigation and local cost maps for local navigations.

3.3 Packages interfaced to Move_Base

- **map-server**: Map server package allows us to save and load the map generated by the costmap-2D package.
- **AMCL**: AMCL is a method to localize the robot in map. This approach uses particle filter to track the pose of the robot with respect to the map, with the help of probability theory. In the ROS system, AMCL can only work with maps which were built using laser scans.

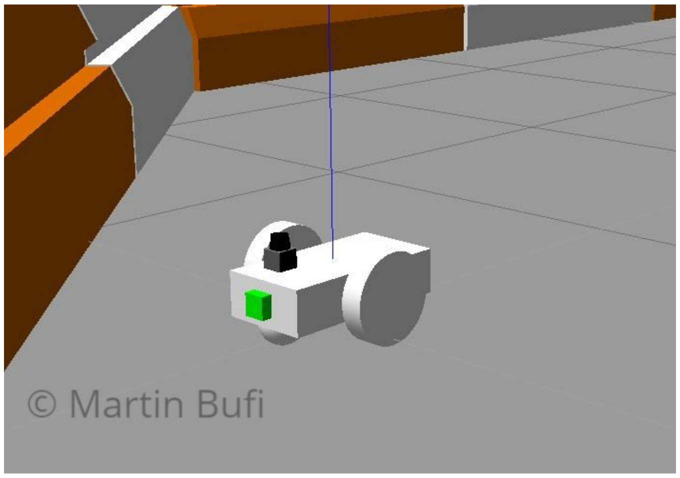


Fig. 7. UdacityBot displayed in Gazebo

4 Building the Mobile Robots in ROS

4.1 Benchmark Udacity Model - UdacityBot

4.1.1 Model Design

4.1.2 Required files for Navigation

The **UdacityBot** package contains a launch file that launches navigation related nodes and packages, and configuration files such as xml file, yaml file that configures various parameters, map file, and rviz configuration file. Below are details of these files.

- **udacity_bot/launch/udacity_world.launch**:
  The *udacity_world.launch* file spawns the robot (details found in the xacro file) into the specific world. It also launches the saved Rviz configuration.
- **udacity_bot/launch/amcl.launch**:
  The *udacity_bot/config/amcl.yaml* file contains various parameter settings of Adaptive Monte Carlo Localization (AMCL) and is used with udacity_bot/launch/amcl.launch file.
- **/config/movebase_params.yaml** :
  This configuration file configures the parameter of move_base that supervises motion planning.
- **config/base_local_planner_params.yaml** :
  A package that ultimately transmits the speed command to the robot and sets the parameters for it.
- **config/base_local_planner_params.yaml** :
  A package that ultimately transmits the speed command to the robot and sets the parameters for it.

Navigation uses the occupancy grid map. Based on this occupancy grid map, each pixel is calculated as an obstacle, a non-movable area, and a movable area using the robots pose and surrounding information obtained from the sensor. In this calculation, the concept of costmap is applied. The below files are for configuring parameters of the costmap.
/config/costmap_common_params.yaml
/config/global_costmap_params.yaml
/config/local_costmap_params.yaml

The **costmap_common_params.yaml** has common parameters where: **global_costmap_params.yaml** file is required for the global area motion planning, while **local_costmap_params.yaml** file is required for the local area motion planning. All the files are commented to allow inspection of each of the parameters that was modified. **For the scope of this report, descriptions of each parameter were excluded. They can be seen on Github.**

4.1.3 Launching udacity_world.launch

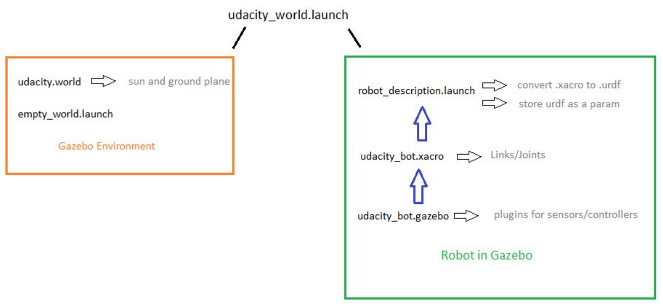


Fig. 8. Files required for udacity_world.launch

4.2 Key parameters to consider for proper AMCL localization

The most challenging part of the localization process is tuning the AMCL filter. Below are some of the most important factors when fine tuning the UdacityBot, as well as the MartinBot. The rest can be read about on the Github repository:

- **Min particles and Max particles** parameter for the filter. These two values are set to 5000 by default, but were greatly reduced to reduce computation time. On a system with more power, both default values would be acceptable.
- **transform_tolerance** specifies the delay in transform (tf) data that is tolerable in seconds. This parameter serves as a safeguard to losing a link in the tf tree while still allowing an amount of latency the user is comfortable with to exist in the system. For example, a transform being 0.2 seconds out-of-date may be tolerable, but a transform being 8 seconds out of date is not. If the tf transform between the coordinate frames specified by the global_frame and robot_base_frame parameters is transform_tolerance seconds older than ros::Time::now(), then the navigation stack will stop the robot. This is a key parameter affecting the stability of the robot model which has to be larger than the update frequency. It was therefore set to 10Hz.
- **robot_radius** is used to set either the footprint of the robot or the radius of the robot if it is circular. In the case of specifying the footprint, the center of the robot is assumed to be at (0.0, 0.0) and both clockwise and counterclockwise specifications are supported.
- **inflation_radius** sets the inflation radius for the costmap. The inflation radius should be set to the maximum distance from obstacles at which a cost should be incurred. For example, setting the inflation radius at 0.95 meters means that the robot will treat all paths that stay 0.95 meters or more away from obstacles as having equal obstacle cost.

The Github repository contains the rest of the parameters used, each commented to get a feel for what they do and why they were changed. There are too many to cover in this report, so the rest have been kept out due to space.

4.3 Personal Model - MartinBot

4.3.1 Model Design

The Martin_bot underwent many modifications:

- **materials.xacro** to be used for coloring.
- Change of base size.
- **New torso link (fixed to chassis)**: Used to place the scanner at a higher position, and attach the grippers. As used for extra weight and inertia.
- **Grippers (based on PR2** attached to torso link for both weight and aesthetics. This was done by downloading the PR2 gripper. This modification was inspired by the ROS tutorial on ”building your own robot model : R2D2”.[1]
- **Laser scanner** at a higher altitude.

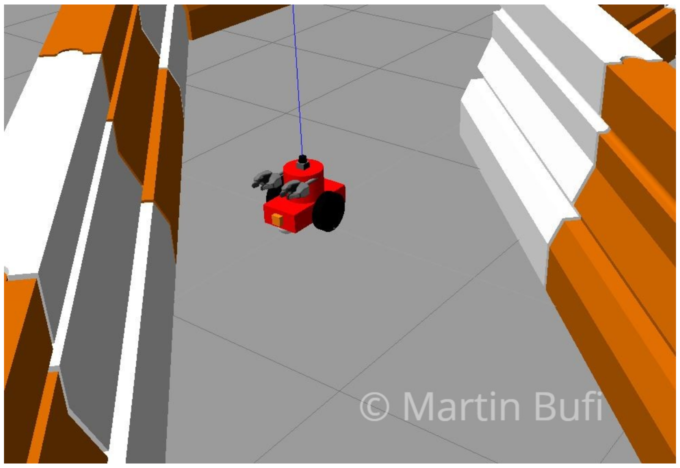


Fig. 9. MartinBOT : Front

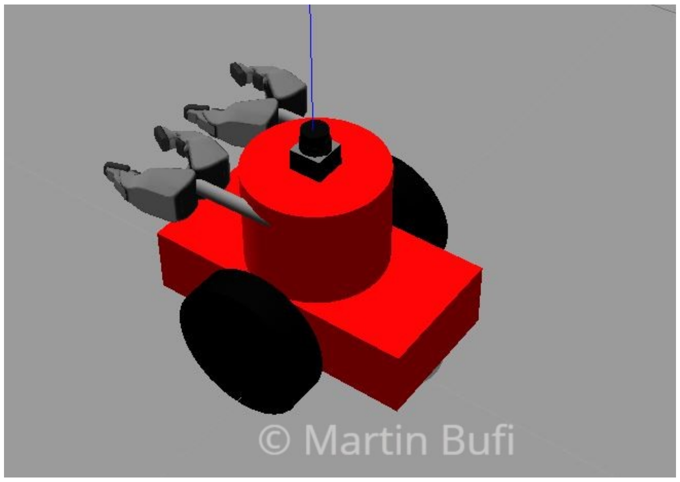


Fig. 10. MartinBOT : Back

4.3.2 Packages Used

The packages from the UdacityBot were reused.

4.3.3 Parameters Used

The Parameters from the UdacityBot were reused.

5 SIMULATIONS

Having all the models built in Gazebo and ready for use, the next step is to simulate the robots in the Jackal maze with the help of RViz .

5.1 Simulating the UdacityBot

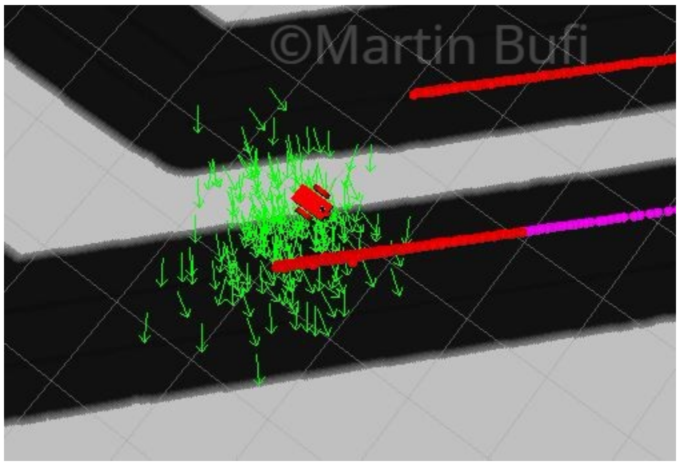


Fig. 11. Udacity bot shown at the Start position with Particle filter running

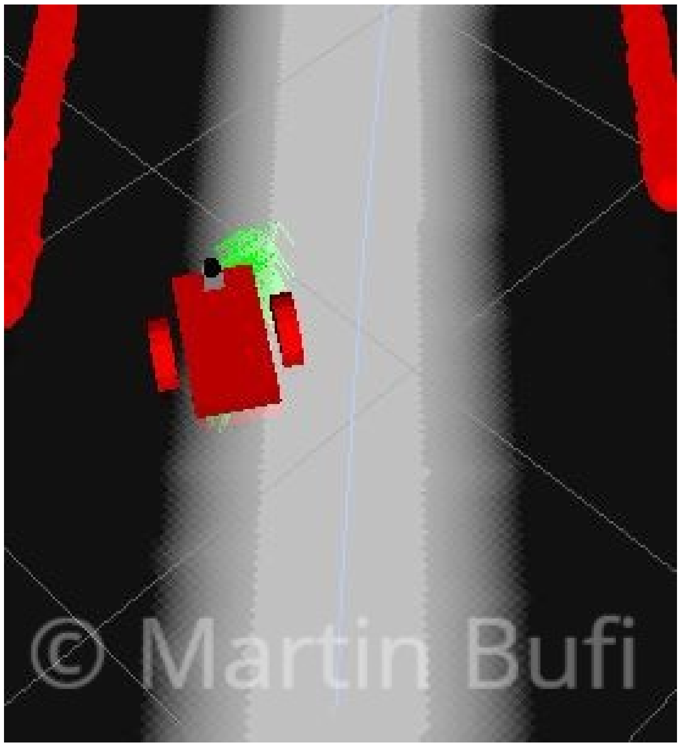


Fig. 12. Particle filter adjustment moving through a tight hallway

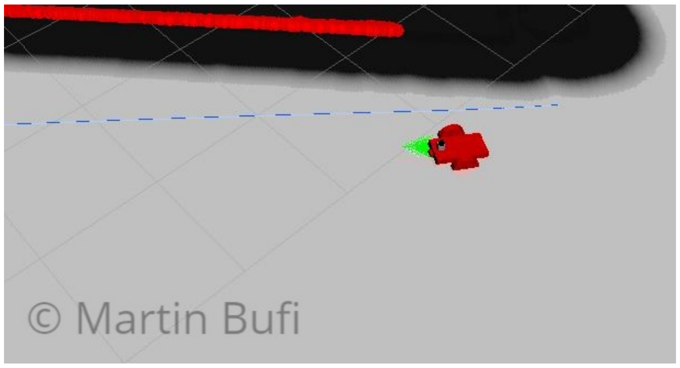
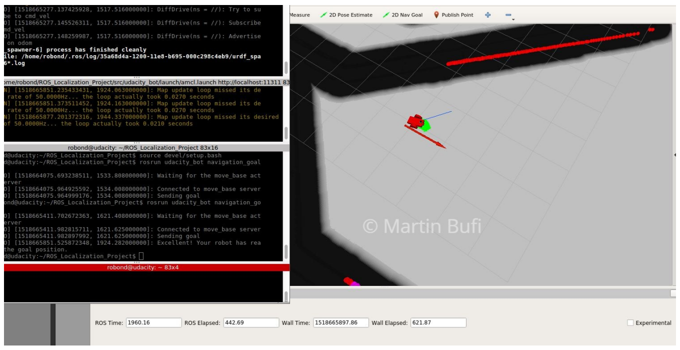


Fig. 13. Particles still accurate entering a large space



```
O] [1518665277.137425928, 1517.516000000]: DiffDrive(ns = //): Try to su
be to cmd_vel
O] [1518665277.145526311, 1517.516000000]: DiffDrive(ns = //): Subscribe
md_vel
O] [1518665277.148259987, 1517.516000000]: DiffDrive(ns = //): Advertise
 on odom
_spawner-6] process has finished cleanly
ile: /home/robond/.ros/log/35a68d4a-1200-11e8-b695-000c298c4eb9/urdf_spa
6*.log
```

```
ome/robond/ROS_Localization_Project/src/udacity_bot/launch/amcl.launch http://localhost:11311 83
N] [1518665851.235433431, 1924.063000000]: Map update loop missed its de
 rate of 50.0000Hz... the loop actually took 0.0270 seconds
N] [1518665851.373511452, 1924.163000000]: Map update loop missed its de
 rate of 50.0000Hz... the loop actually took 0.0270 seconds
N] [1518665877.201372316, 1944.337000000]: Map update loop missed its desired
of 50.0000Hz... the loop actually took 0.0210 seconds
```

```
robond@udacity: ~/ROS_Localization_Project 83x16
d@udacity:~/ROS_Localization_Project$ source devel/setup.bash
d@udacity:~/ROS_Localization_Project$ rosrun udacity_bot navigation_goal

O] [1518664075.693238511, 1533.808000000]: Waiting for the move_base act
erver
O] [1518664075.964925592, 1534.008000000]: Connected to move_base server
O] [1518664075.964999176, 1534.008000000]: Sending goal
ond@udacity:~/ROS_Localization_Project$ rosrun udacity_bot navigation_go

O] [1518665411.702672363, 1621.408000000]: Waiting for the move_base act
erver
O] [1518665411.982815711, 1621.625000000]: Connected to move_base server
O] [1518665411.982897992, 1621.625000000]: Sending goal
O] [1518665851.525872348, 1924.282000000]: Excellent! Your robot has rea
the goal position.
d@udacity:~/ROS_Localization_Project$
```

```
robond@udacity: ~ 83x4
```

ROS Time: 1960.16 ROS Elapsed: 442.69 Wall Time: 1518665897.86 Wall Elapsed: 621.87 Experimental

Fig. 14. Challenge Completed: UdacityBot at the final Pose.

5.2 Simulating the MartinBot

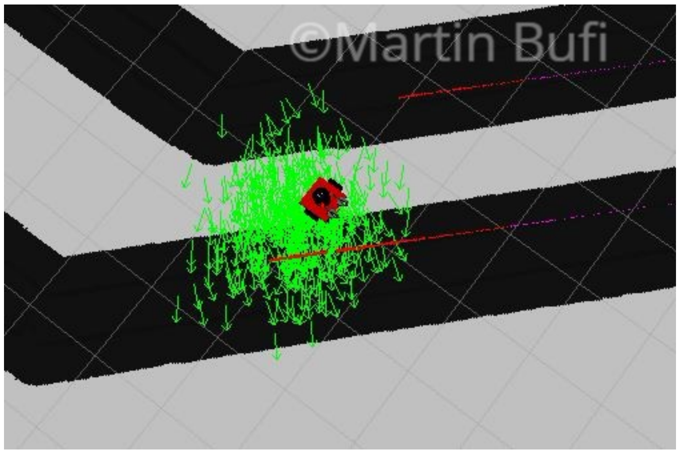


Fig. 15. Martinbot shown at the Start position with Particle filter running

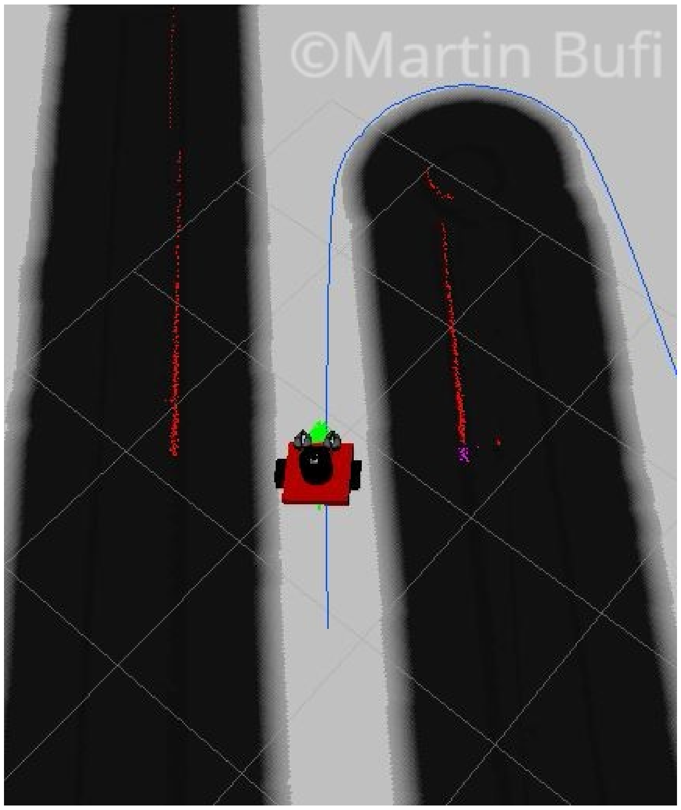


Fig. 16. Particle filter adjustment moving through a tight hallway

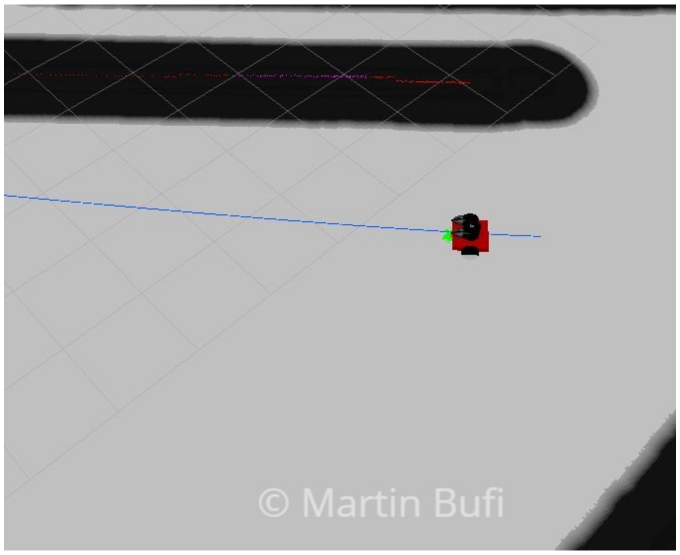
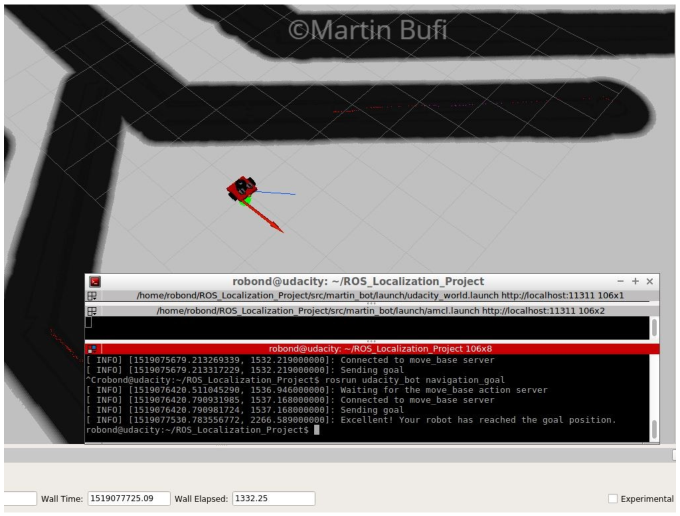


Fig. 17. Particles still accurate entering a large space



```
robond@udacity: ~/ROS_Localization_Project
/home/robond/ROS_Localization_Project/src/martin_bot/launch/udacity_world.launch http://localhost:11311 106x1
/home/robond/ROS_Localization_Project/src/martin_bot/launch/amcl.launch http://localhost:11311 106x2
robond@udacity: ~/ROS_Localization_Project 106x8
[ INFO] [1519075679.213269339, 1532.219000000]: Connected to move_base server
[ INFO] [1519075679.213317229, 1532.219000000]: Sending goal
^Crobond@udacity:~/ROS_Localization_Project$ rosrun udacity_bot navigation_goal
[ INFO] [1519076420.511045290, 1536.946000000]: Waiting for the move_base action server
[ INFO] [1519076420.790931985, 1537.168000000]: Connected to move_base server
[ INFO] [1519076420.790981724, 1537.168000000]: Sending goal
[ INFO] [1519077530.783556772, 2266.589000000]: Excellent! Your robot has reached the goal position.
robond@udacity:~/ROS_Localization_Project$
```

Wall Time: 1519077725.09 Wall Elapsed: 1332.25 Experimental

Fig. 18. Challenge Completed: Martinbot at the final Pose.

Note: all pictures can be seen on the github at a higher resolution.

6 Results and Discussion

6.1 General Localization Results

Both robots were able to successfully complete the course in under 10 minutes and performed equally well. Weight did not factor into the equation as what was expected. There were times the robots would drive in circles due to uncertainties, most likely due to inaccurate sensor readings and/or model tuning. This is something that will need to be further investigated in the future. This behaviour can be seen in the image below, where MartinBot begins moving north for a few minutes before calculating the proper trajectory. This error in navigation was the main reason for the delay in total time to reach the end goal.

It was also interesting that the laser scanner placement had no impact on the performance. There were indications that it would pick up the grippers occasionally, but

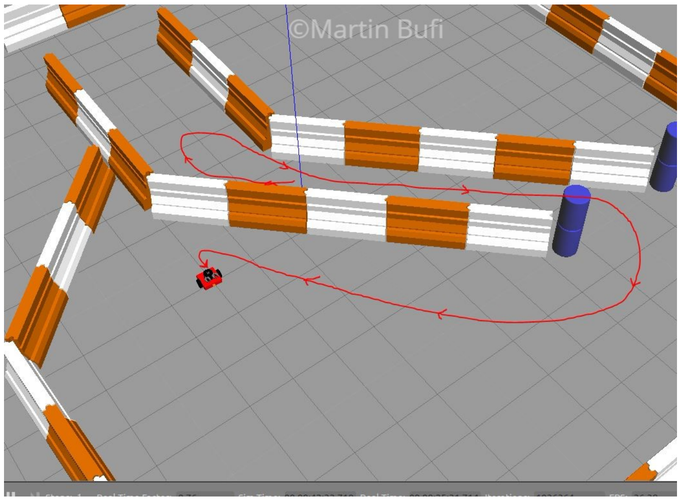


Fig. 19. Path taken to reach end position

it performed just like the UdacityBot due to the height of the barriers. If the barriers were lowered, the scanner would have much more trouble picking up the walls, almost certainly resulting in a robot collision.

After thorough development and testing, below are a few sentiments on the AMCL localization algorithm:

- One scenario that needs to be accounted for when researching localization techniques for mobile robots, would be to classify the behavior of an algorithm when the robot abruptly disappears from one location and shows up at another. **AMCL does not work well for the kidnapped robot problem.** problem.
- MCL/AMCL works very well in any industry domain where clear barriers/paths guide the robot during its trajectory path. The ground should be flat, clear of obstacles is the laser scanner is to be placed above ground level (which was tested with MartinBot).

7 Conclusion / Future work

In conclusion, both robots reached the end goal position in approximately 10 minutes. Both robots were able to avoid any collision with the barriers, thus proving the concept of proper localization when given a known map. One of the issues to look into would be the navigation algorithm and how the move base reacted to its commands due to having most of the time to reach the goal spent on following the wrong path and stopping for long extended periods of time. For future work, it will be important to further look into speeding up the navigation planner to allow more real time decision making as to what twist commands need to be sent and their update frequency to allow a smoother ride with less stops. Until the navigation planning is improved, neither of the ROS stacks implemented in this project could be put towards a commercial product due to inefficiencies and jitter in the movement. Also, different sensor placement would be useful. Placement of the laser scanner worked for this case, but in a commercial product, one cannot expect ideal maps and environment to work gracefully with all the sensors.

7.1 Real time Hardware Deployment in a commercial product

Once the navigation stack were to be improved, the next step would to apply the ROS navigation stack to real time hardware.

- Deploy the stack on a Jetson TX2 board running ROS and Ubuntu 16.04.
- Base: Create2 by iRobot.
- Scanner: hokuyo scanner.
- Camera: Standard webcam or camera on the Jetson dev board.

8 References

[1] ROS.org, *Building a Visual Robot Model with URDF from Scratch*. http://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch

[2]Udacity, *What's a Kalman filter?*. Udacity.com

[3]Udacity, *What's MCL?*. Udacity.com

[4]kaiyuzheng, *ROS Navigation Guide*. http://kaiyuzheng.me/documents/navguide.pdf