# Mobile Robot Localization Using Adaptive Monte Carlo Algorithm for ROS

Ross Lloyd

**Abstract**—The aim of this paper was to establish parameter settings for ROS amcl and move_base nodes, such that virtual robots of two designs could navigate robustly in a ROS-based, previously mapped environment. The robots also were to localise themselves accurately within the space and at a given goal location. Both robots were able to attain good localization and to plan and maneuver without hindrance to the goal. Additionally, a challenge in which one of the robots was to follow a set of given waypoints without using the ROS move_base node was completed.

**Index Terms**—ROS, Localization, Particle Filter, Udacity.

## 1 Introduction

In 1991, Professor Ingemar J. Cox described the localization problem as *"the most fundamental problem to providing a mobile robot with autonomous capabilities"* [4]. Localisation is the capacity of an autonomous agent to know its place and the places of objects in its environment, and is fundamentally a coordinate transformation problem: The robot's software seeks to define its own coordinates within the frame of its surroundings. It is the foundation of the robot's physical agency.

The 80's and 90's saw a great deal of progress with using Kalman Filters, which this paper will touch on, and later in Markov Localisation. Point or feature based methods, geometric methods, occupancy grid mapping, and *map matching* began to yield impressive results. However all of these techniques share the common root of Bayes Probality Theorem, which will be discussed later.

In order of difficulty, localization can be broken down into three main classes: First *position tracking*, where the initial pose is known and changes in any one dimension can be measured; Second is *Global Localization*, where the initial starting pose is unknown and all locations on the map have an equal probability; And finally the *Kidnapped Robot Problem* wherein the robot has localized itself, reducing the probability distribution to a point, and then being "kidnapped" - blindfolded and suddenly moved to a new location, such that any established probability distribution is unable to consider the new coordinates without some kind of recovery mechanism.

With such a complex task, clearly performance and accuracy are both of prime importance to the robotic designer. The higher accuracy, the more calculations you must perform; the more calculations, the more time any algorithm must take due to increasing complexity. For this reason, this paper will study the use of the Adaptive Monte Carlo Localisation filter, which boasts variable computation complexity through the dynamic manipulation of probabilistic "guesses", as well as a very rapid means of converging on the actual position.

## 2 Background

### 2.1 Kalman Filters

Originally designed to help guide rockets in the Apollo Moon Missions, and now able to produce GPS measurements to within sub-metre accuracy, track features in computer images, and accomplish sensor fusion in robotics, Kalman Filters (KF) are able to produce fast, accurate estimates of real world variables. They can do this from inputs which contain noise and a host of other elements affecting their accuracy. They accomplish this by estimating, for each time step, a Joint Probability Distribution of those variables.

There are three main types - the standard, linear Kalman Filter; The non-linear Extended Kalman Filter, and the highly non-linear Unscented Kalman Filter. To take standard Kalman Filters as an example: A KF acts in two main steps, a measurement update, and a movement update. The filter makes a guess of its current position based on the previous measurement, and then checks it against the new measurements after the robot has moved. This 'best guess' is a unimodal Gaussian Distribution, with the most likely position being represented by the peak (mean) of the distribution curve.

For multivariate Kalman Filters, the vectors for the mean and covariance take the form of **Figure 1** below:

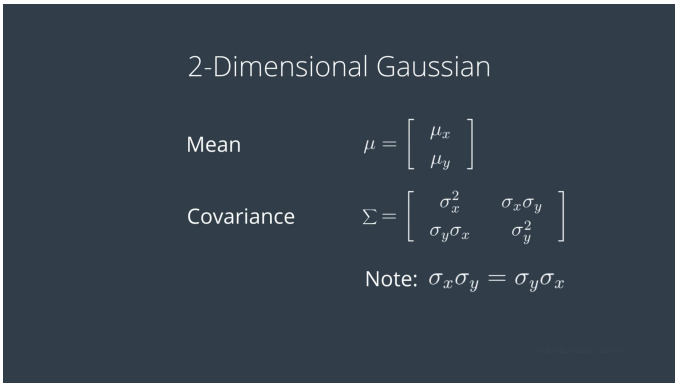



Fig. 1. Vectors containing the mean and covariance of the distribution. *Credit: Udacity*

The Multivariate Gaussian Distribution is given by **Figure 2** below:

$$p(x) = \frac{1}{(2\pi)^{\frac{D}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

Fig. 2. Equation giving the Multivariate Distribution. *Credit: Udacity*

The Multivariate KF is an *n-dimensional* version of the 1-dimensional KF. The equations needed to calculate it are as follows:

State Prediction:

$$x' = Fx$$

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

Measurement Update:

$$y = z - Hx'$$

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

Calculation of Kalman Gain:

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

Calculation of Posterior State and Covariance:

$$x = x' + Ky$$

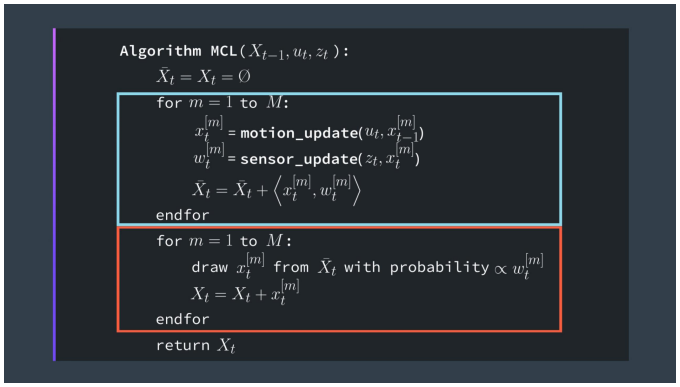
$$P = (I - KH)P'$$

Fig. 3. Equations for calculating Kalman Filter. *Credit: Udacity*

Kalman Filters make two main assumptions: 1) That the measurement models are linear and 2) That the state space can be represented by a unimodal distribution. These assumptions are limiting as many real world applications do not behave this way. Normally, if we pass a Gaussian distribution through a linear transformation, the output is another Gaussian distribution. However, if we do this with a non-linear function, the output is non-Gaussian, which is computationally very expensive. To recoup the speed and efficiency the KF is famous for, the Extended Kalman Filter (EKF) is instead used, which linearises a local portion of the function, centred on the mean. This simplification is effective and allows the KF to be applied to non-linear functions.

### 2.2 Particle Filters

A *Monte Carlo Localization filter*, more generally known as a *Particle Filter*, uses sensor measurements to keep track of a robot's pose. The algorithm takes as inputs the previous belief about the robot's pose, the actuation command and the sensor measurements. The belief is given by the Bayes Filter as per figure 4b, where X$t$ is the state at time $t$, and Z1...$t$ are the measurements from time 1 up to t.




$$Bel(X_t) = P(X_t|Z_{1...t})$$

Fig. 4. a) MCL Algorithm b) Bayes equation. *Credit: Udacity*

With reference to figure 4a, the initial belief is obtained by generating a random set of particles from 1 to *M*. In a first 'for' loop, the hypothetical state $X_t^{[m]}$ is estimated, and then a sensor update is performed. An *importance factor* $X_t^{[m]}$ or weight is assigned to the particles based on how close they are to this initial hypothetical estimate. Then the motion and measurement are added to the previous state $\bar{X}_t$.

Then the second 'for' loop, or *importance sampling* loop, is invoked. Its purpose is to select the particles with the highest weights, and so highest probability of being correct, and discard the rest. The selected particles are then redrawn as the new belief $\bar{X}_t$. The cycle then repeats until a precise location is determined.

### 2.3 Comparison / Contrast

So what are the benefits and disadvantages of using the EKF or MCL algorithms? Figure 6 below summarizes, however the main points are that MCL is easier to program than EKF; That MCL can approximate any distributions, while KF can only approximate Gaussian distributions. MCL is not limited by linear gaussian state space assumptions as in EKF; It is possible to control the computational load of MCL by changing the particle density, however with EKF you cannot.

For the purposes of this paper, we will be working with MCL only, however in ROS the adaptive version of the algorithm, which is capable of varying the number of particles M as the state estimate converges, is used.

## 3 Simulations

### 3.1 Achievements

In the case of both the classroom supplied robot and the novel robot, localization was achieved with respect to the assessment goal.

Success in localisation was subjectively assessed based on the appearance of the 'success' message' and a visual assessment of alignment with the goal. In the case of the *class supplied robot* (udacibot), some variation from run to run

| | MCL | EKF |
|---|---|---|
| Measurements | Raw measurements | Landmarks |
| Measurement Noise | Any | Gaussian |
| Posterior | Particles | Gaussian |
| Efficieny (memory) | Good | Very Good |
| Efficiency (time) | Good | Very Good |
| Ease of Implementation | Very Good | Good |
| Resolution | Good | Very Good |
| Robustness | Very Good | None |
| Memory and Resolution Control | Yes | No |
| Global Localization | Yes | No |
| State Space | Multimodal Discrete | Unimodal Continuous |

Fig. 5. Benefits and Disadvantages of the Algorithms.

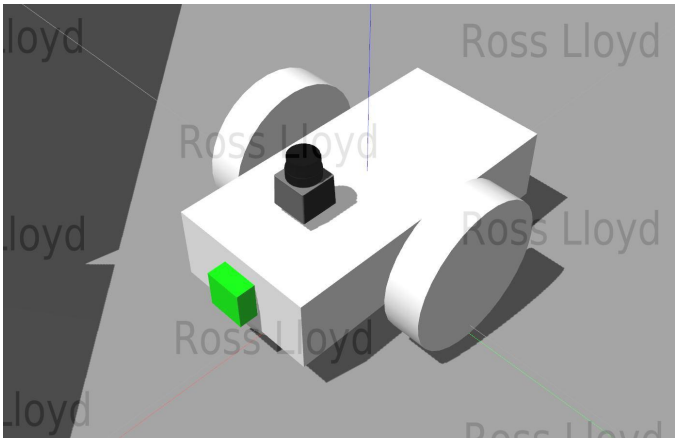



Fig. 6. Class provided 'udacibot'

was observed, mainly in rotational orientation. In the case of the *novel robot* (six bot), the performance was consistently better in both positional and angular orientation, in general aligning almost perfectly with the goal.

The challenge portion of the project was also completed, in which the robot must follow a set series of waypoints *without* using the move base node. A short video accompanies this report.

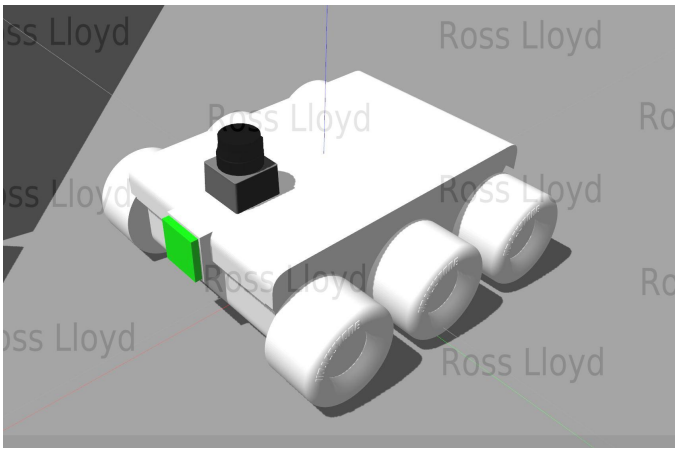



Fig. 7. Self created 'six bot'

## 3.2 Benchmark Model

### 3.2.1 Model design

Figure 9 below illustrates the layout and design of the benchmark model:

### 3.2.2 Parameters

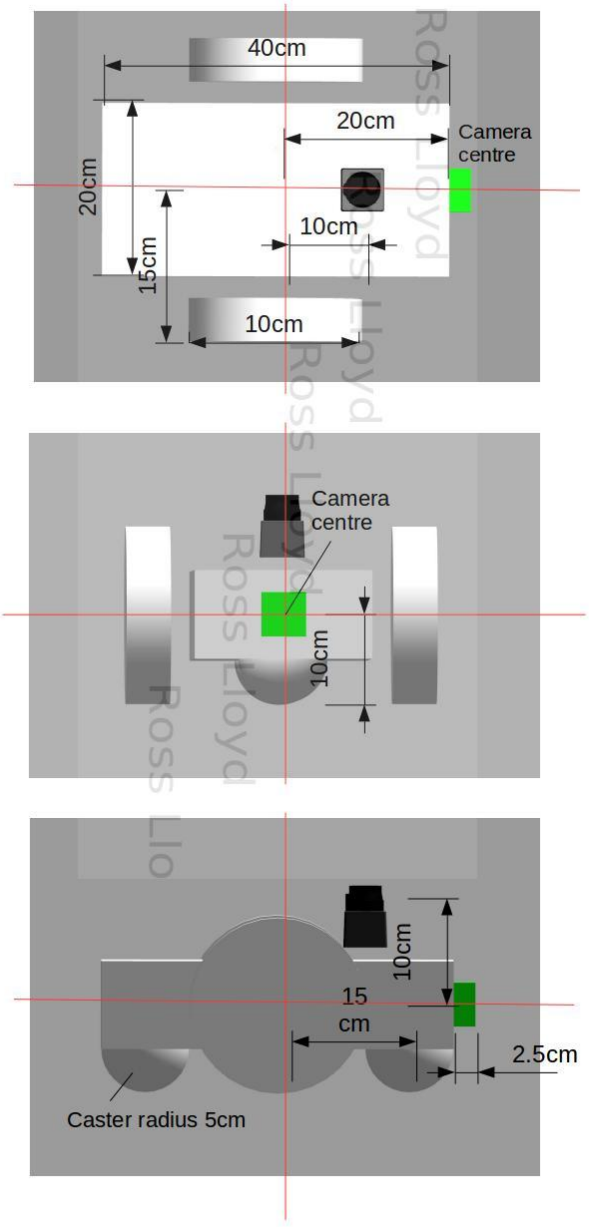



Fig. 8. 'udacibot' layout and design

### 3.2.3 Packages Used

The main packages used in the project are the move_base and amcl packages. For the former several topics under the NavFN, TrajectoryPlannerROS, global and local costmap, action server and current_goal headings are vital to navigation. For amcl, amcl_pose is the main topic of interest.

| Parameter | Chosen Value | Low | Correct | High |
|---|---|---|---|---|
| Speed max vel x | 0.35 | Duration to goal: High | Robot corners well and no unusual behaviour | Robot is unable to steer accurately even with increased theta speed / acceleration due to momentum |
| Inflation radius | 0.3 | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| cost scaling factor | 6.0 | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| Footprint | [-0.2, -0.2], [-0.2, 0.2], [0.2, 0.2], [0.2, -0.2] | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| Footprint padding | 0.01 | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| gdist scale | 0.5 | Robot ignores local trajectory and may hit walls | Robot attempts to stay with the path but can vary slightly from the local path | Robot can get stuck if it cannot navigate to the local goal |
| pdist scale | 1.0 | Robot does not move | Robot attempts to stay with trajectory but can vary slightly from the global path | No higher value available (value is normalized) |
| occdist scale | 0.01 | Robot will not move | Robot will leave appropriate distance to obstacles | Robot hits obstacles |
| obstacle range | 5.0 | Far from walls, robot loses accuracy and / or path planning | Robot plans and localizes well | No impact but waste of resources if larger than local costmap size |
| raytrace range | 6.0 | No observable impact | Obstacle placement remains accurate | No observable impact |
| local costmap W&H | 6.0 x 6.0 | No lower values implemented | Robot is able to navigate to all points on the map including project goal | Robot is able to navigate to all points on the map besides given project goal |
| Update frequency | 4.0Hz | Too slow and navigation is impaired | Robot navigation is responsive without overtaxing system | Too fast and errors occur as update loop is missed |

move_base and costmap parameters used for **udacitybot** and their effects

Fig. 9. 'udacibot' Move base and costmap parameters

| Parameter | Chosen Value | Low | Correct | High |
|---|---|---|---|---|
| Translational update rate | 0.4 | Performance issues. Must lower particles, leads to inaccurate map matching | AMCL Pose converges within 5-6 updates and is accurate. | AMCL pose struggles to converge |
| Rotational update rate | Pi / 8 | Performance issues. Must lower particles, leads to inaccurate map matching | AMCL Pose converges quickly and is accurate. | AMCL pose struggles to converge |
| initial_pose | 0 0 0 0 0 0 | No observable effect under normal conditions but would be relevant in "kidnapped robot" scenario | No observable effect under normal conditions but would be relevant in "kidnapped robot" scenario | No observable effect under normal conditions but would be relevant in "kidnapped robot" scenario |
| min_particles | 200 | No observable effect | OK | Performance problems |
| max_particles | 8000 | Reasonable localization performance | Most accurate localization performance | Slower performance and possible errors relating to update loop timing |
| odom_alpha 1-4 | 0.0 | Robot cannot navigate | Odometry is noiseless. Correct setting allows navigation | Robot cannot navigate |

amcl parameters used for **udacitybot** and their effects

Fig. 10. 'udacibot' amcl parameters

#### *3.2.4 Parameters*

Figures 9 and 10 show the relevant *changed* (all others default) parameters for the move base configuration and amcl node, with a record of observed effects of each when set to low, correct and high values. The functions of each of the settings are as follows:

**Move base parameters:**

- **max vel x:** This item sets the top linear speed that the robot can assume.
- **inflation radius:** A critical factor in calculating costmap values in the region of the wall and obstacles. See also discussion section for further analysis.
- **cost scaling factor:** Sets the gradient of the decay curve, or how steep or shallow the cost drops off in the region of obstacles.
- **footprint:** Nominally the outline of the robot, again critical for calculating the costmap. Can be set smaller or larger.
- **Footprint padding:** Provides a buffer zone to the footprint that further influences the costmap calculations.
- **gdist scale:** Affects how much the robot tries to adhere to the local planner goal.
- **pdist scale:** Affects how much the robot tries to adhere to the global planner goal.
- **occdist scale:** Affects how much the robot tries to avoid obstacles.
- **obstacle range:** The distance within which the robot will add detected objects to the obstacle costmap.
- **raytrace range:** The distance within which the robot will remove obstacles from the costmap when free space is detected.
- **local costmap width and height:** The size of area considered in local planner calculations.
- **Update frequency:** The rate at which the robot's position and trajectory is refreshed and which the processor must be capable of updating to.

**amcl parameters:**

- **Translational update rate:** The linear distance the robot travels before making an update to the amcl pose.
- **Rotational update rate:** The angle the robot must rotate through before making an update to the amcl pose.
- **Initial pose:** Acts as a primer to the localization algorithm to minimize starting localization error.
- **min particles:** The smallest particle cloud that is permissible to the adaptive part of the algorithm when calculating the amcl pose.
- **max particles:** The largest particle cloud that is permissible to the adaptive part of the algorithm when calculating the amcl pose.
- **odom alpha:** Refers to the expected noise in the odometry information.

### 3.3 Personal Model

#### *3.3.1 Model design*

Figure 11 below shows the design and layout of 'sixbot'. Sensor positions are largely unchanged aside from slight elevation of the camera and a lowering of the laser scanner due to the lower profile of the wheels.

#### *3.3.2 Parameters*

The design created for the "Personal Model" section of the report was a six-wheeled buggy with a similar sensor layout to the benchmark model. The aim was to attain fully functioning skid-steer with 6 wheels as this is a common configuration in the literature and wider world. This proved more difficult than it seems due to the lack of documentation on the various skid-steer implementations. Models for chassis and base were created in Solidworks and exported as STL.

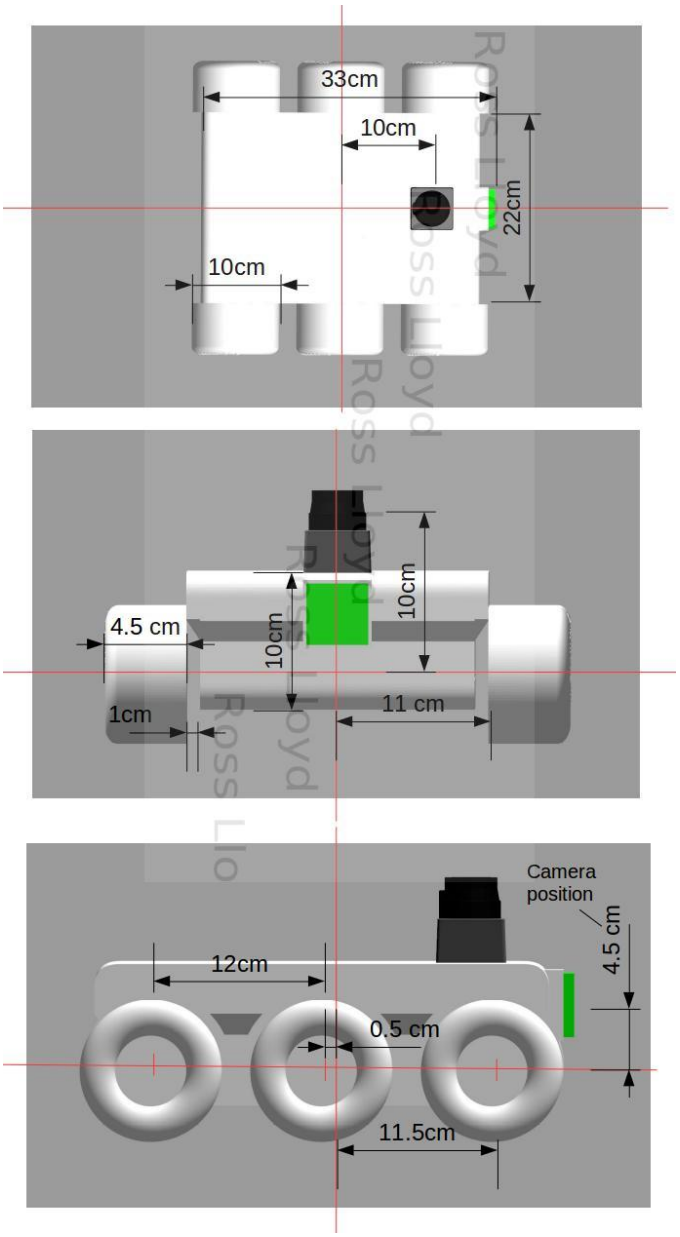



Fig. 11. 'sixbot' layout and design

| Parameter | Chosen Value | Low | Correct | High |
|---|---|---|---|---|
| Speed max vel | 0.3 | Duration to goal: High | Robot corners well and no unusual behaviour | Unusual Physics Behaviour |
| Inflation radius | 0.25 | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| cost scaling factor | 6.0 | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| Footprint | [-0.2, -0.12], [-0.2, 0.12], [0.2, 0.12], [0.2, -0.12] | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| Footprint padding | default | Robot moves too close to wall, gets stuck | Robot moves close to wall but does not collide | Path planning fails due to distance from wall on tight corners |
| gdist scale | 0.35 | Robot ignores local trajectory and may hit walls | Robot attempts to stay with the path but can vary slightly from the local path | Robot can get stuck if it cannot navigate to the local goal |
| pdist scale | 1.0 | Robot does not move | Robot attempts to stay with trajectory but can vary slightly from the global path | No higher value available (value is normalized) |
| occdist scale | 0.01 | Robot will not move | Robot will leave appropriate distance to obstacles | Robot hits obstacles |
| obstacle range | 5.0 | Far from walls, robot loses accuracy and / or path planning | Robot plans and localizes well | No impact but waste of resources if larger than local costmap size |
| raytrace range | 6.0 | No observable impact | Obstacle placement remains accurate | No observable impact |
| local costmap W&H | 6.0 x 6.0 | No lower values implemented | Robot is able to navigate to all points on the map including project goal | Robot is able to navigate to all points on the map besides given project goal |
| Update frequency | 4.0Hz | Too slow and navigation is impaired | Robot navigation is responsive without overtaxing system | Too fast and errors occur as update loop is missed |

move_base and costmap parameters used for **sixbot** and their effects

Fig. 12. 'sixbot' Move base and costmap parameters

| Parameter | Chosen Value | Low | Correct | High |
|---|---|---|---|---|
| Translational update rate | 0.4 | Performance issues. Must lower particles, leads to inaccurate map matching | AMCL Pose converges within 5-6 updates and is accurate. | AMCL pose struggles to converge |
| Rotational update rate | Pi / 8 | Performance issues. Must lower particles, leads to inaccurate map matching | AMCL Pose converges quickly and is accurate. | AMCL pose struggles to converge |
| initial_pose | 0 0 0 0 0 0 | No observable effect under normal conditions but would be relevant in "kidnapped robot" scenario | No observable effect under normal conditions but would be relevant in "kidnapped robot" scenario | No observable effect under normal conditions but would be relevant in "kidnapped robot" scenario |
| min_particles | 200 | No observable effect | OK | Performance problems |
| max_particles | 8000 | Reasonable localization performance | Most accurate localization performance | Slower performance and possible errors relating to update loop timing |
| odom_alpha 1-4 | 0.0 | Robot cannot navigate | Odometry is noiseless. Correct setting allows navigation | Robot cannot navigate |

amcl parameters used for **sixbot** and their effects

Fig. 13. 'sixbot' amcl parameters

#### *3.3.3 Packages Used*

In order to operate the 6 wheel skid steer configuration, the package **diffDrivePlugin6W** from **hector gazebo plugins** was used (https://wiki.ros.org/hector_gazebo_plugins). The same packages as applied in the benchmark model were also included, excepting of course the **differential drive controller** which was applied to the two wheel benchmark robot. Note that a 4 wheel skid steer configuration is possible with the latter, however within this project it would not function adequately (leaving the middle two wheels as casters) and so the 6w plug in was selected. Additionally, for the camera the **libgazebo_ros_camera.so** plugin was used, and for the laser scanner, the **libgazebo_ros_laser.so** plugin.

#### *3.3.4 Parameters*

See figures 12 and 13.

### 3.4 Personal Model - Challenge

The main changes were to the amcl node settings, where the maximum particles was reduced to 2000 to prevent an error due to performance issues. This stemmed from the need to set the update rates to the maximum possible, i.e. the update rate of the laser scan. Separate launch files were created, entitled **udacity_world_wp.launch** and *amcl_wp.launch* for use with the challenge.

## 4 Results

In addition to the results presented below, please also see the following youtube videos of the benchmark robot, novel design of robot and of the challenge portion featuring the novel design of robot. (Note: The viewer may wish to set the playback speed to double as the robot motion is relatively slow). If these hyperlinks do not work, the full URL's can be found in the bibliography, [1][2] and [3].

## 4.1 Localization Results

### *4.1.1 Benchmark Robot 'Udacibot'*

With reference to figures 14, 15 and 16, benchmark results for localization were reasonable. It can be observed that the particle cloud has converged on the robots position, though one or two arrows are not completely on-heading (see discussion for possible reasons and fixes).

At the **update rate** selected (0.4m translational, 0.4 rad rotational), the particles can be said to converge after roughly 25 seconds. If the update rate is set to the laser scan rate, the convergence is almost immediate as soon as the sim is loaded, however this has a negative impact on ultimate localization accuracy, and - dependent on particle cloud size, can result in performance issues and failure of the amcl node (see later section on the "challenge" where this relationship was explored).

Once appropriate values were found for **pdist scale, gdist scale, inflation radius, cost scaling factor and footprint padding**, the costmap calculated by the costmap2d node lead to a global and local path that took the robot close to the wall, but not so close that collisions occurred or the robot moved into a too-high cost region, resulting in stuck behaviours (either freezing or attempting fruitlessly to revolve or reverse).

One point to note is that the benchmark robot will occasionally pause at the hard U-turn around the wall, before its travel to the goal. This is due to a slight salient in the costmap which causes the robot to pause and back up. This can be minimised by further increasing the **inflation radius** to "smooth out" the costmap, however it also functions as a useful indicator that recovery behaviours are functioning as intended. [Note: for more general navigation, an inflation radius of 0.25 is more appropriate as, for goals other than the prescribed position, an inflation radius of 0.3 can be prohibitive. 0.25 works for the goal position and all other positions, 0.3 just for the prescribed position]

Once **linear speed** is set to an appropriate value (see figure 9), the robot is able to progress smoothly without understeering, continually finding a route to the global path via the local path.

An unusual observation was made with respects to **local map size and the path planning** ONLY to the udacity-supplied goal: When the local map size was the same as the global map size, and when navigating to a user assigned position, the robot did not experience problems with computing an efficient route. However, when planning a route to the goal position, the global planner would attempt to navigate in the opposite direction, as though attempting to reach the goal via the wall in the "alleyway" at the edge of the map. Reducing the local map size such that this wall could not be included in the local map forced the robot to always choose a direct path to the goal, with the trade off that it was necessary to compute the global path at each update.

When the hokuyo sensor was set to the class-supplied height and forward setting, from time to time the robot would freeze as it detected its own wheels or chassis. For this reason the height and forward placement was adjusted until no such interference took place.

Further analysis of the impact of various parameters in move base and amcl nodes can be seen in figures 9 and 10.

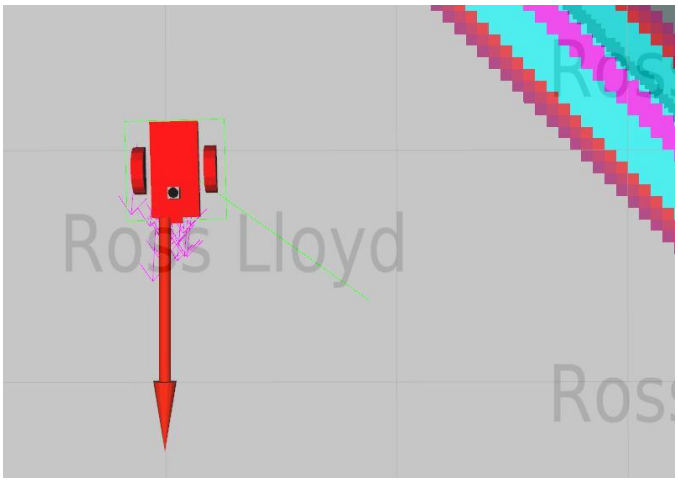



Fig. 14. Plan view of udacitybot at goal

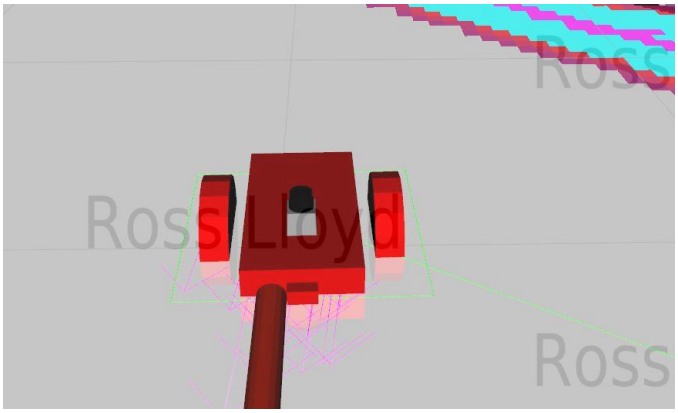



Fig. 15. Front view of udacitybot at goal

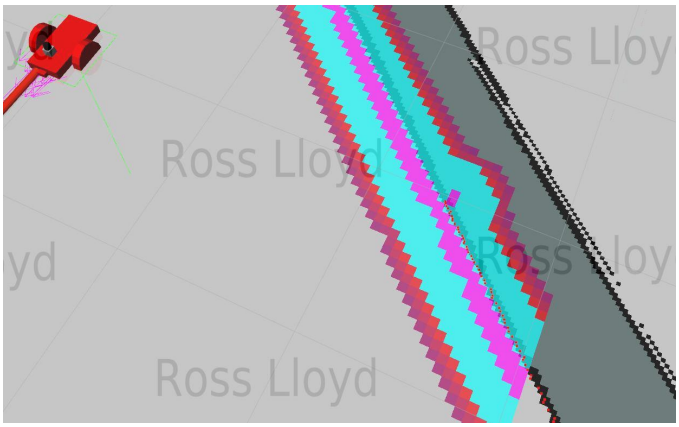



Fig. 16. Udacitybot at goal, laser and map accuracy

### *4.1.2 Novel Robot 'Sixbot'*

With reference to figures 17, 18 and 19, benchmark results for the novel robot localization were slightly superior to the benchmark model. It can be observed that the particle cloud

has converged on the robots position, though again one or two arrows are not completely on-heading (see discussion for possible reasons and fixes). The 6 wheel skid steer allows for some slight sideways translation and this seems to aid in acquiring a good orientation and alignment with the goal.

At the **update rate** selected (0.4m translational, 0.4 rad rotational), the particles can be said to converge after roughly 30 seconds, somewhat slower as the robot's speed must be lower to avoid some unusual physics behaviour in gazebo if the speed is set too high.

Again, if the update rate is set to the laser scan rate, the convergence is almost immediate as soon as the sim is loaded, with the same caveats that this has a negative impact on ultimate localization accuracy, and - dependent on particle cloud size, can result in performance issues and failure of the amcl node (see later section on the "challenge" where this relationship was explored).

Once appropriate values were found for **pdist scale, gdist scale, inflation radius, cost scaling factor and footprint padding**, the costmap calculated by the costmap2d node lead to a global and local path that took the robot close to the wall, but not so close that collisions occurred or the robot moved into a too-high cost region, resulting in stuck behaviours (either freezing or attempting fruitlessly to revolve or reverse).

Once **linear speed** is set to an appropriate value (see figure 12), the robot is able to progress smoothly, continually finding a route to the global path via the local path. As mentioned elsewhere, some unusual physics behaviour from the diffdrive6W addon prevented linear speed settings greater than or equal to ¿= 4.0 m/s. 3.0m/s provided the greatest reliability from the physics engine.

As with the benchmark robot, an unusual observation was made with respects to **local map size and the path planning** ONLY to the udacity-supplied goal: When the local map size was the same as the global map size, and when navigating to a user assigned position, the robot did not experience problems with computing an efficient route. However, when planning a route to the goal position, the global planner would attempt to navigate in the opposite direction, as though attempting to reach the goal via the wall in the "alleyway" at the edge of the map. Reducing the local map size such that this wall could not be included in the local map forced the robot to always choose a direct path to the goal, with the trade off that it was necessary to compute the global path at each update.

Further analysis of the impact of various parameters in move base and amcl nodes can be seen in figures 12 and 13.

### 4.2 Challenge

Please see the accompanying challenge video showing the robot accurately navigating to a set series of waypoints [3]. In this section, the robot was able to turn to and move linearly to the set of waypoints provided in the c++ node, **drive_bot.h** and **drive_bot.cpp**. Though there is some variation in the position at each waypoint, the robot does not accumulate error significant enough to cause failure.

### 4.3 Technical Comparison

The prime difference between the benchmark (udacibot) and novel (sixbot) platforms is the drive mechanism and wheel

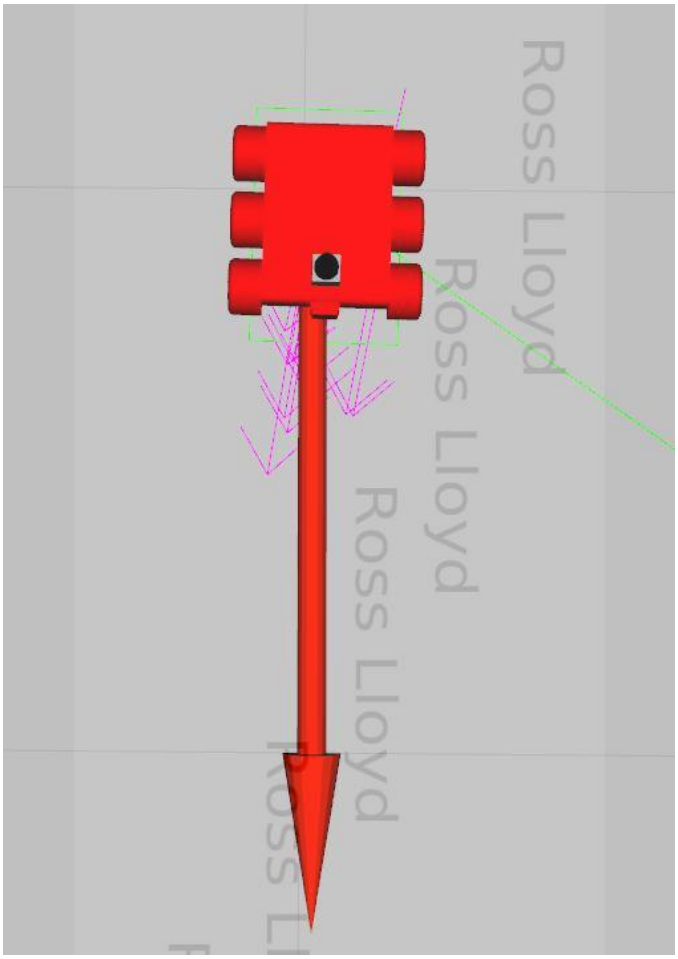



Fig. 17. Sixbot at goal, laser and map accuracy


Fig. 18. Sixbot at goal, laser and map accuracy

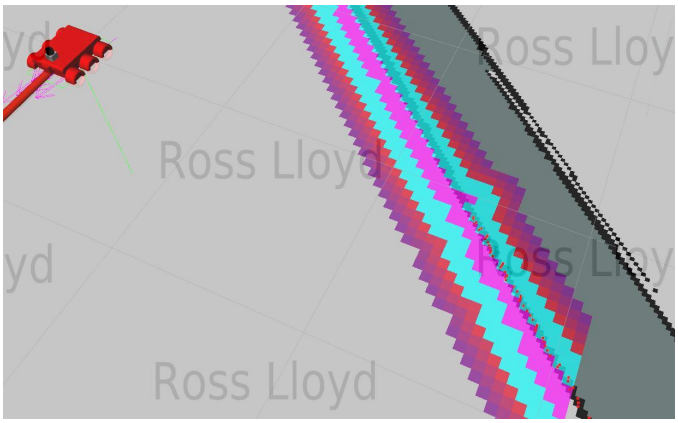



Fig. 19. Sixbot at goal, laser and map accuracy

layout. The udacitybot is a pure differential drive, two wheel platform, with frictionless casters on the front and back to provide balance. This confers excellent angular movements as the wheels form a pure couple, however mechanically in terms of grip and maneuverability it loses out to multi-wheel platforms.

In the ideal case, the 6 wheeled robot should be able to outperform the two wheel robot due to superior grip and multiple wheel drive inputs, however, due to a physics problem in gazebo it is not possible to drive the sixbot at significantly higher speeds to take advantage of the added traction and controllability. Ultimately the two wheel robot is actually the faster of the two with top speeds being 0.35m/s and 0.3m/s for the udacibot and sixbot respectively, though on some runs the sixbot has been able to navigate accurately up to 0.39m/s.

Path following and obstacle avoidance in both robots is excellent, though the six wheel robot seems to cope better with angular and linear motion combined i.e. cornering, with the two wheel udacibot tending to oscillate slightly as it corners.

Some problems were encountered with the hokuyo placement in the default udacibot. Due to its vertical and forward placement, it would sometimes register its own wheels and chassis as an obstacle, causing the robot to freeze. In order to avoid this problem some additional offset was included. This was not an issue with the sixbot due to the much smaller wheels.

Inflation radius was set slightly higher for udacibot at 0.3 (versus the sixbot at 0.25), due to the increased chassis length and its impact on the costmap calculation. This was also reflected in the footprint.

There is a minor difference in the gdist scale set for udacibot (0.5) as the lower limit applied to sixbot, when applied to udacibot, tended to lead to oscillation around the goal position.

Most other values, particularly those relating to move base and planning, are identical.

Values for amcl are also the same.

# 5 Discussion

## 5.1 Topics

*a) Which robot performed better and why?*. The performance of the two robots was very similar, though the ultimate orientation to goal was superior with the sixbot. This is most likely due to the 6 wheel drive's ability to slightly translate sideways at the final position, which helps compensate for any lack of alignment before the robot begins its turn to final heading. In terms of path planning and following, both robots performed extremely well due to well selected parameters for the move base node, in particular the cost scaling factor which kept the robots close to walls but with enough clearance to not get stuck. If this factor is too aggressive, it is possible for the robot to get stuck despite not hitting a wall, as the costmap becomes too "deep" too quickly. From run to run there were only minor differences in localization performance for the sixbot vs udacibot, with by far the largest factor being the update rate. If the rate was too high, the localization would ultimately drift as the robots neared the goal, but if too low the pose array would not converge.

*b) 'Kidnapped Robot' problem.* The kidnapped robot problem can be presented thus: The robot has sufficiently localized itself in the room and the particle filter has converged. This means that particles only exist around the robot's known position and nowhere else, due to the resampling process. Next, the robot is suddenly 'teleported' to a new location in the room. There are no particles in this location with which to localize the robot. In order to solve this problem, an 'injection of random particles' is implemented, such that the space includes a selection of possible evenly, but sparsely, distributed seeds around which a kidnapped robot may localize itself. This can be accomplished by using the Augmented Monte Carlo Localization Algorithm, which injects said particles.

*c) In What types of scenario could localization be performed?* Localization is appropriate in static environments, that is, ones where the map remains unchanged. In situations where objects move, such as buildings with people, the algorithm is unable to localize according to a known map and will therefore fail without additional algorithms, such as **state augmentation** or **outlier rejection**.

*d) AMCL in industry domains.* AMCL and MCL would be of greatest use in mostly static environments such as warehouses, or perhaps as a patrol robot in unoccupied buildings.

## 5.2 Challenge

In order to make the robot navigate to set waypoints, it was necessary to set the amcl update rate to the maximum, i.e. the rate of the laser scan. This was because the normal update rate for the project was not able to trigger the simple logic that determines when the robot is at one of the waypoints. This logic incorporated a small threshold calculation to allow for some error, but the update needed the fastest possible rate to perform robustly.

A class based approach was used in the c++ code (see **drive_bot.cpp** and **drive_bot.h**) which held the current

waypoint information, current pose, and which made simple decisions about which way to turn and when to move forwards.

The main challenge was setting the threshold such that variation in either direction e.g. left or right of a waypoint, could be attained, without allowing the robot to go so far off target that errors would multiply. This threshold was 6 percent for forward motion and 0.5 percent for angular motion. The latter is particularly important as a rotational error has a much larger impact on final positioning especially when the robot moves relatively large distances after the rotation.

The linear and rotational speeds were kept very low in order to maximise the chance of an accurate 'trigger' when reaching the waypoint.

Actual localization relative to the map is poor, due to the drift that results with very high amcl update rates. However as the requirement was for the robot to navigate to a set of waypoints, this small drift was deemed acceptable.

## 6 Conclusion / Future work

The AMCL algorithm is a robust means of identifying and tracking a robot's pose within a known environment, and estimate that pose (position / orientation) with a good deal of accuracy whilst not being computationally heavy. Using a novel design of robot, the results show that AMCL is applicable to other designs of robot with alternate drive systems.

As the dimensions of the base have such an impact on the costmap calculation, a square base has some advantage over an elongated one. In an elongated robot, the inflation layer must be tied to the robot's longest dimension, which reduces the size of the navigable area (or conversely, increases the area in which a high cost can be found adjacent to walls). The very best shape for this purpose would be a small circular one as with the Turtlebot.

It would be useful to add a 360 degree lidar sensor so that the robot has 'all-around' vision. This would allow it to take in twice as much data to pass to the localization algorithm which may lead to faster convergence. It may further be beneficial to implement short range sensors such as ultrasound, given that the laser scanner has a minimum range, for example one on each side of the robot.

The AMCL algorithm could also be applied in 3 dimensional applications, such as with drones or robots that are able to scale walls, in which case either a 3d lidar or RGBD camera may be used to generate voxel data. An application of drone technology is in underground space mapping in hazardous environments, such as mine prospecting or rescue situations.

### 6.1 Challenge

The waypoint challenge code could be improved by adapting it to take any series of waypoints from a .txt file, which would require extending the turn / move logic and adding a command to parse the file for the waypoints. It was decided not to do this due to time constraints.

### 6.2 Hardware Deployment

To deploy AMCL on hardware a processor of appropriate speed and memory must be used, as well as being able to attach a laser scanner and camera (at least). As accuracy and processing speed are effectively traded off against one another by the number of particles 'M', the use of an adaptive algorithm, which can change the number of particles as the cloud converges, will make best use of the available resources. This would require mounting on an appropriate mobile base with wheel encoders, separate motors and drivers for each wheel and a battery.

## 7 References / Bibliography

1 Udacity RoboND - Where Am I? Project - Benchmark Robot, https://www.youtube.com/watch?v=seA7nMnv6ws
2 Udacity RoboND - Where Am I? Own Robot Design, https://www.youtube.com/watch?v=seA7nMnv6ws
3 Udacity Nanodegree - Where Am I? Challenge, https://www.youtube.com/watch?v=oWdIWyFbrJI
4 S. Thrun, W Burgard, D Fox, Probabilistic Robotics, *The MIT Press, 2006*