Localisation: Where Am I?

Shane Reynolds

March 16, 2019

Abstract

Reliable localisation of a mobile robot in a known environment is an important problem in robotics. Higher order tasks, such as navigation, rely on the robot understanding where it is in an environment. This paper explores the workings of two widely used approaches which solve the localisation problem: Kalman filters; and Monte Carlo Localisation. An off-the-shelf Monte Carlo Localisation implementation is tested in ROS, using simulation software called Gazebo. The simulation experiment explores the package robustness against changes in robot body geometries and sensor locations.

1 Introduction

Suppose a robot has an unknown pose in an environment for which it has a map. The robot takes sensor readings, and based on these observations, must infer a set of poses where it could be located in the environment. This scenario is known as the *localisation problem* [1, 2]. Put simply, localisation is the problem of estimating a mobile robot's location and orientation relative to its environment, given sensor data [3]. This is not as straight forward as it appears given robot actuators are subject to small random performance perturbations, and sensors provide imperfect measurement due to noise. Compounding these problems, the robot may not even know it's initial pose relative to the environment. This paper provides a discussion on two approaches to solving the localisation problem. The first of these solutions is the Extended Kalman Filter (EKF): an adaptation of the Kalman Filter, suited to the estimation of non-linear system responses. The second solution is a particle filter method called Monte Carlo Localisation (MCL). The paper concludes with an application of MCL in a Gazebo simulation using an off-the-shelf MCL package in ROS. The MCL implementation is trialled across two different robot models providing opportunity to assess robustness of tuned parameters, and an analysis of parameter tuning for reliable performance.

2 Background

The mobile robot localisation problem comes in three flavours. The simplest involves tracking the robot's pose relative to its environment - called *position tracking*. This scenario requires an initial known estimate of the robot's pose relative to the environment. As the robot takes actions and senses the environment the robot pose is updated [4]. Often initial pose estimates are unavailable, since the robot may not know where it is - this scenario represents a more meaningful problem version called the *global localisation problem*. The goal here is for the robot to determine it's pose from scratch. Finally, the most challenging problem type is referred to as the *kidnapped robot problem* which sees a localised robot tele-ported to another location on the map without knowledge of the move. This is different to the *global localisation problem* scenario because, after tele-porting, the robot incorrectly believes it is somewhere other than its current location. This final scenario is used to test whether a localisation algorithm can recover from a catastrophic failure.

2.1 Kalman Filters

Kalman Filters can be used to solve the simplest localisation problem: position tracking. In an ideal world, position tracking is a trivial task in which the robot knows it's starting location, and updates position with perfect actuation and perfect sensor readings. In the real world, however, this is not the case. Actuation is not perfect, and is subject to minor perturbations, or wheel slippage can occur. Furthermore, sensor readings are noisey. The implications are that both movement and measurement are imprecise as they are subject to stochastic errors. Consider the a mobile robot moving along a one dimensional trajectory. Let the discrete time position be x_j, at time j. We note that the position at time j is dependent on the previous position at time $j-1$, denoted as x_{j-1}, plus any movement action taken by the robot, denoted as u_j. Additionally, as previously noted, the robot's actuators are not perfect and are subject to noise, ω_j. We can write x_j as follows:

$$x_j = ax_j + bu_j + \omega_j \tag{1}$$

This description is sometimes easier to understand when visualised as a computational block diagram, shown in Figure 1.

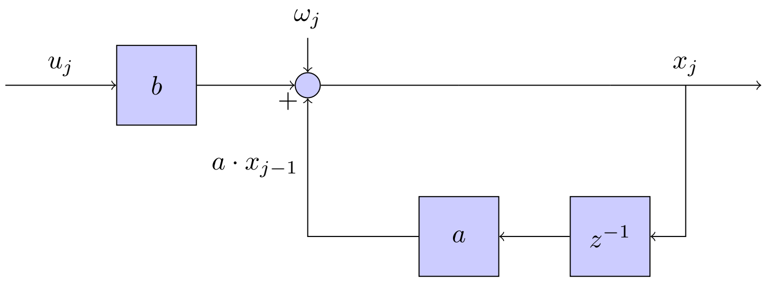

Figure 1: Block diagram illustrating robot position, x_j, at discrete time j updated using control action, u_j, and the previous position x_{j-1}. Stochastic perturbation is introduced using ω_j.

Almost always random phenomena are hidden from the observer behind a dynamical system - Kalman described this as follows:

> *A random function of time may be thought of as the output of a dynamic system excited by an independent Gaussian random process.* [5]

It is true for robotic motion that we must attempt to estimate the robot's stochastically perturbed motion, through the use of odometry sensors, or laser range finders, which are subject to stochastic noise. It must be noted, however, that it is not necessarily true that these stochastic perturbations follow a Gaussian distribution. Letting the measurement of signal x_j be represented by z_j, we can write the following equation defining the measurement of x_j, subject to noise ν_j, as:

$$z_j = hx_j + \nu_j \tag{2}$$

Equation (2) can be represented in a computational graph as shown in Figure 2.

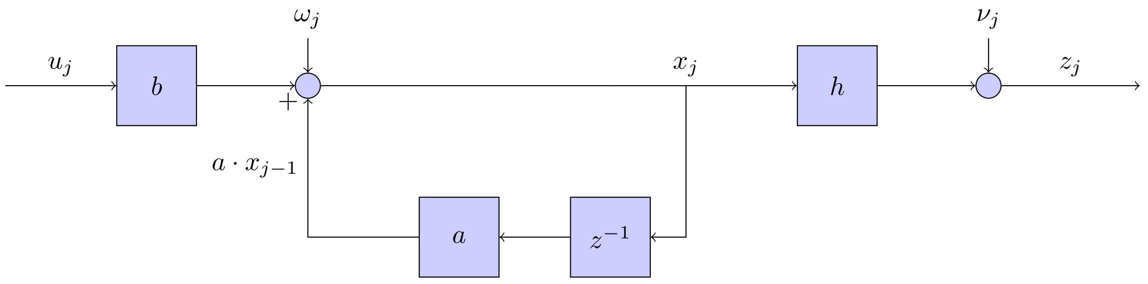

Figure 2: Block diagram showing measurement of position, z_j. This is an extended iteration of the block diagram shown in Figure 1. Stochastic variance due to imperfect sensors is added with ν_j.

The fundamental problem that the Kalman Filter tries to answer is: how can an accurate estimate of a hidden stochastic signal be found when it is observed as the output of a dynamical system? The first step to answering this is to create a mathematical model of the system we can use to find an estimate of the measurement, z_j, which we call $\hat{z}_j$. This estimate is based on an estimation of the hidden variable, which we denote with $\hat{x}_j^-$. The mathematical model that provides these estimates does not factor in stochasticity seen in actual signals x_j and z_j. This is the *a priori* estimate of x_j, and is mathematically described as:

$$\hat{x}_j^- = a\hat{x}_{j-1} + bu_j \tag{3}$$

The *a priori* estimate is used to predict an output estimate, $\hat{z}_j$. In turn, $\hat{z}_j$, is used to estimate the difference between predicted signal and the observed signal, referred to as the residual:

$$z_j - \hat{z}_j = z_j - h\hat{x}_j^- \tag{4}$$

If the residual is small, then our estimate is good. If it is large then our estimate is not good. We can use the residual in (4) to update our *a priori* estimate, $\hat{x}_j^-$:

$$\hat{x}_j = \hat{x}_j^- + k(z_j - h\hat{x}_j^-) \tag{5}$$

A visualisation of the computational architecture for the process can be seen in Figure 3. We see the Kalman filter process decomposed into two distinct phases:

1. the state prediction phase in which the residual is used to update the *a priori* providing a state prediction, known as the *posterior belief*
2. the measurement update phase in which the *posterior belief* is updated with control actions u_j and used as the new *a priori*, which is combined with sensor readings to compute the new residual.

The task of determining k, used to refine our estimate, is at the heart of the Kalman filtering process. So far we have only considered problems of a single dimension - determining k in this domain is a fairly simple task, however, the problem is complicated further when considering robotic motion in multiple dimensions. Most mobile robotic motion can be thought of in a 2D planar environment, and expressed using coordinates x, and y, and orientation θ. Adding to the problem dimensionality we may also want to estimate other important locomotion variables such as velocity. Suppose that state was now represented as vector of variables. We define this using bold typeface $\mathbf{x}_j$. Similarly, control actions are represented as $\mathbf{u}_j$, and the measurement for each state variable in $\mathbf{x}_j$ is represented by $\mathbf{z}_j$. Without loss of generality equation (1) becomes:

$$\mathbf{x}_j = \mathbf{A}\mathbf{x}_j + \mathbf{B}\mathbf{u}_j + \boldsymbol{\omega}_j \tag{6}$$

Similarly, equation (2) becomes:

$$\mathbf{z}_j = \mathbf{H}\mathbf{x}_j + \boldsymbol{\nu}_j \tag{7}$$

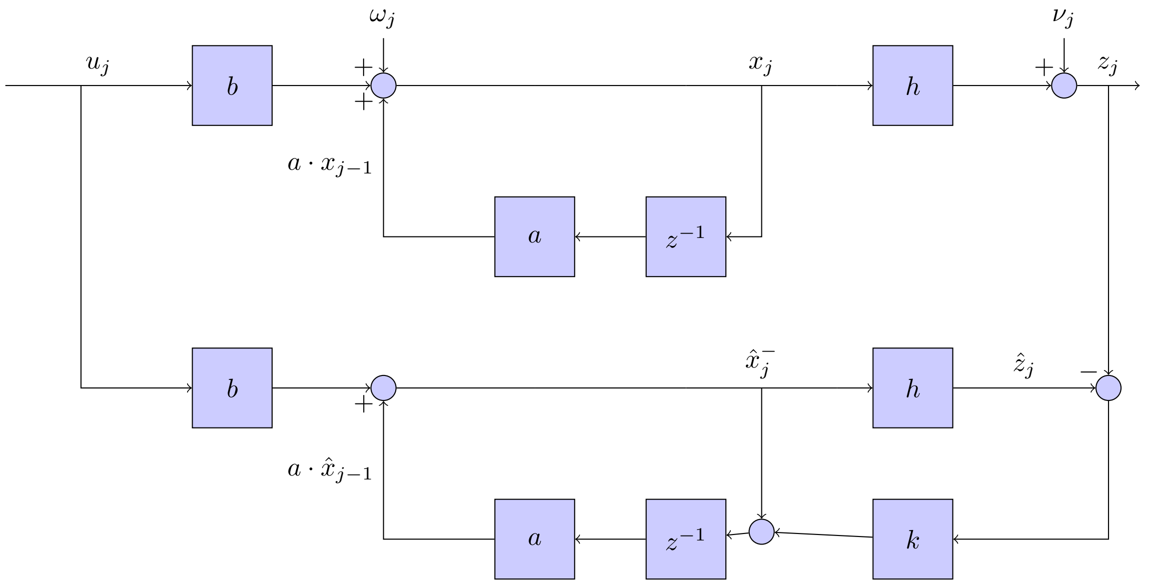

Figure 3: Block diagram showing the actual measurement model (above), and the prediction model (below). The outputs of the two models are compared and fed to a feedback topology. Core to the Kalman filter implementation is the selection of weights k in the bottom right hand corner.

The *a priori* estimate of the state variable, shown in equation (3), in multidimensional form, is given by:

$$\hat{\mathbf{x}}_j^- = \mathbf{A}\hat{\mathbf{x}}_j + \mathbf{B}\mathbf{u}_j \tag{8}$$

The *a priori* update, shown in equation (5) is given by:

$$\hat{\mathbf{x}}_j = \hat{\mathbf{x}}_j^- + \mathbf{K}_j(\mathbf{z}_j - \mathbf{H}\hat{\mathbf{x}}_j^-) \tag{9}$$

The error for the *posterior*, $\mathbf{e}_j$, can be written as:

$$\mathbf{e}_j = \mathbf{x}_j - \hat{\mathbf{x}}_j \tag{10}$$

A common way to measure the accuracy of $\mathbf{e}_j$ is using the covariance matrix, denoted in the literature as $\mathbf{P}_j$. This can be expressed as follows:

$$\mathbf{P}_j = \mathbb{E}[\mathbf{e}_j \cdot \mathbf{e}_j] \tag{11}$$

The Kalman filter attempts to select the gain, $\mathbf{K}$, such that the *posterior* covariance is minimised. Applying a classical optimisation technique, we arrive at the following expression:

$$\frac{\partial \mathbf{P}_j}{\partial \mathbf{K}_j} = \frac{\partial \mathbb{E}[(\mathbf{x}_j - \hat{\mathbf{x}}_j)(\mathbf{x}_j - \hat{\mathbf{x}}_j)^T]}{\partial \mathbf{K}_j} = 0 \tag{12}$$

Solving equation (12) yields the following result for the Kalman filter gain:

$$\mathbf{K}_j = \frac{\mathbf{P}_j\mathbf{H}^T}{\mathbf{H}\mathbf{P}_j\mathbf{H}^T + \mathbf{R}} \tag{13}$$

The denominator of equation (13) contains the measurement noise, represented by matrix $\mathbf{R}$. In fact, the denominator is an important calculation step that maps the state prediction covariance, $\mathbf{P}_j$, into the measurement space - often this is expressed with the variable $\mathbf{S}$, that is:

$$\mathbf{S} = \mathbf{H}\mathbf{P}_j\mathbf{H}^T + \mathbf{R} \tag{14}$$

In addition to the *posterior* error, $\mathbf{e}_j$, there is an error for the *a priori*, denoted $\mathbf{e}_j^-$. The covariance matrix for $\mathbf{e}_j^-$ is given by $\mathbf{P}_j^-$. The final step to arriving at an algorithm for the Kalman filter is being able to recursively update both the *posterior* and *a priori* covariance matrices. The *a priori* error covariance update is derived from considering the error expression $\mathbf{x}_j - \hat{\mathbf{x}}_j^-$, and is expressed using the movement noise matrix $\mathbf{Q}$. The expression is as follows:

$$\mathbf{P}_j^- = \mathbf{A}\mathbf{P}_{j-1}\mathbf{A}^T + \mathbf{Q} \tag{15}$$

The *posterior* update can be derived from equation (11), and is expressed as:

$$\mathbf{P}_j = (\mathbf{I} - \mathbf{K}_j\mathbf{H})\mathbf{P}_j^- \tag{16}$$

Equations (8), (9), (13), (14), (15), and (16) make up the Kalman Filter algorithm. A single pass of the Kalman Filter algorithm can be seen in Algorithm 1 - the pass starts with the *a priori* state estimate update, and concludes with the *posterior* error covariance update. The full algorithm would see the code run continuously, in a loop, iteratively updating state and covariance estimations.

Algorithm 1 Kalman Filter

```
1: $\hat{\mathbf{x}}' \leftarrow \mathbf{A}\hat{\mathbf{x}} + \mathbf{B}\mathbf{u}$ (A Priori State Estimate Update)
2: $\mathbf{P}' \leftarrow \mathbf{A}\mathbf{P}\mathbf{A}^T + \mathbf{Q}$ (A Priori Error Covariance Update)
3: $\mathbf{S} \leftarrow \mathbf{H}\mathbf{P}'\mathbf{H}^T + \mathbf{R}$ (Covariance Update In Measurement Space)
4: $\mathbf{K} \leftarrow \mathbf{P}'\mathbf{H}^T\mathbf{S}^{-1}$ (Kalman Gain Update)
5: $\mathbf{x} \leftarrow \mathbf{x}' + \mathbf{K}(\mathbf{z} - \mathbf{H}\mathbf{x}')$ (Posterior State Estimate Update Using Kalman Gain)
6: $\mathbf{P} \leftarrow (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}'$ (Posterior Error Covariance Update)
```

One of the major assumptions made when using the Kalman Filter is that motion and measurement models are linear. Unfortunately, most robot motion models exhibit non-linearity (REFERENCE). The main issue here is that non-linearity violates the other major assumption made when using Kalman Filters: state space variables can be modelled using a Gaussian distribution. For example, if some Gaussian distributed random variable X is used in a non-linear function $Y = f(X)$, then the resulting random variable Y is no longer Gaussian. To address this issue *motion models* and *measurement models* are linearised using a first order multidimensional Taylor series approximation, for short distances (REFERENCE). This does not materially change the structure of Algorithm 1, however, does require some additional computation in execution.

2.2 Particle Filters

Particle filters can be used to solve the *global localisation* problem, that is, it can localise a robot in an environment when its initial position is unknown. This section looks at a specific type of particle filter known as Monte Carlo Localisation (MCL). The central idea of MCL is the estimation of a posterior distribution across robot poses, often referred to as the *belief*. The *belief* is defined as probability density over the pose state space, which is conditioned on sensor measurement data, and odometric action data. Mobile robot pose, $\mathbf{x}$, is made up of location coordinates x and y and orientation θ. Letting observation from a laser range finder be represented by $\mathbf{z}$, and odometric sensor data by represented by $\mathbf{u}$, we can mathematically express the belief as follows:

$$Bel(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{z}_t, \mathbf{u}_{t-1}, \mathbf{z}_{t-1}, \mathbf{u}_{t-2}, \ldots, \mathbf{z}_0) \tag{17}$$

Particle filters estimate belief recursively, meaning that an update rule from time t to time $t+1$ is required, and an initial belief is also needed. The initial belief characterizes the initial knowledge about the system state - if little is known about the initial system state, then a uniform distribution can be used over the state space. To derive the update equation we start by expressing (17) using Bayes rule:

$$Bel(\mathbf{x}_t) = \frac{p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)\ p(\mathbf{x}_t|\mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)}{p(\mathbf{z}_t|\mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)} \tag{18}$$

The denominator is a constant value relative to $\mathbf{x}_t$, and (4) can be written more compactly as:

$$Bel(\mathbf{x}_t) = \eta\ p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)\ p(\mathbf{x}_t|\mathbf{u}_{t-1}, \ldots, \mathbf{z}_0) \tag{19}$$

where η is a constant value:

$$\eta = p(\mathbf{z}_t|\mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)^{-1} \tag{20}$$

A recursive relation seeks to provide an update at each time step. Information from the past is factored into the belief as changes occur. Therefore it can be reasoned that only new information presented to the system from current state is important when modifying the belief. This simplifying assumption, referred to as the *Markov assumption*, suggests that future data is independent of past data given knowledge of the current state. Mathematically, this allows equation (19) to be re-written as:

$$Bel(\mathbf{x}_t) = \eta\ p(\mathbf{z}_t|\mathbf{x}_t)\ p(\mathbf{x}_t|\mathbf{u}_{t-1}, \ldots, \mathbf{z}_0) \tag{21}$$

Partitioning over $\mathbf{x}_{t-1}$ equation (21) can be written as:

$$Bel(\mathbf{x}_t) = \eta\ p(\mathbf{z}_t|\mathbf{x}_t) \int p(\mathbf{x}_t, \mathbf{x}_{t-1}|\mathbf{a}_{t-1}, \ldots, \mathbf{z}_0)\ d\mathbf{x}_{t-1} \tag{22}$$

Applying Bayes' equation (22) can be written as:

$$Bel(\mathbf{x}_t) = \eta\ p(\mathbf{z}_t|\mathbf{x}_t) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)\ p(\mathbf{x}_{t-1}|\mathbf{u}_{t-1}, \ldots, \mathbf{z}_0)\ d\mathbf{x}_{t-1} \tag{23}$$

Exploiting the Markov property again the first expression in the integral can be simplified. The second expression can be written using the *Bel* notation as follows:

$$Bel(\mathbf{x}_t) = \eta\ p(\mathbf{z}_t|\mathbf{x}_t) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{t-1})\ Bel(\mathbf{x}_{t-1})\ d\mathbf{x}_{t-1} \tag{24}$$

There are two key components to equation (24): the first is probability $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{t-1})$ which is referred to as the *motion model*; and the second is the probability $p(\mathbf{z}_t|\mathbf{x}_t)$ which is referred to as the *sensor model*.

2.2.1 Motion model $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{t-1})$ and sensor model $p(\mathbf{z}_t|\mathbf{x}_t)$

Under noise free ideal conditions the robot moves from state $\mathbf{x}_{t-1}$ to state $\mathbf{x}_t$ with certainty given some control action $\mathbf{u}_{t-1}$. The assumption of idealness allows the use of kinematic equations to fully describe the robot's motion. As previously discussed, however, physical robot motion is subject to uncertainty since actuators are not ideal, which means there is uncertainty in pose $\mathbf{x}_t$. The *motion model*, $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{t-1})$, describes a posterior density over successors to pose $\mathbf{x}_{t-1}$. A graphical depiction of what this looks like can be seen in Figure 4. This shows the distribution over where the robot could be located given some control action. The *sensor model* provides the probability, or likelihood, of obtaining the sensor measurements, $\mathbf{z}_t$, given a robot pose, $\mathbf{x}_t$ and control action $\mathbf{u}_{t-1}$, assuming that we have a known map.

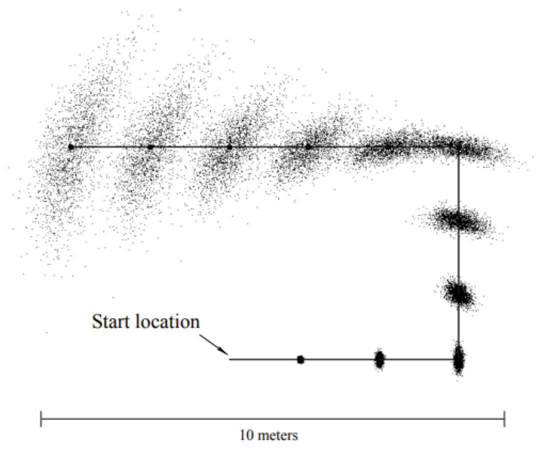

Figure 4: We see that the uncertainty of where the robot is located increases as it moves. Uncertainty is indicated by highly dispersed points.

2.2.2 MCL Implementation

The state space for mobile robot localisation is continuous, meaning that implementing the recursive localisation equation shown in (24) is computationally intractable. Particle filter algorithms attempt to represent the belief $Bel(\mathbf{x}_t)$ using a set of m weighted samples distributed according to $Bel(\mathbf{x}_t)$. Mathematically, we can express this as:

$$Bel(\mathbf{x}_t) \approx \{\mathbf{x}^{(i)}, \omega^{(i)}\}_{i=1,\dots,m} \tag{25}$$

The sampled particles, $\mathbf{x}^{(i)}$, are discrete hypotheses of the robot's pose drawn from $Bel(\mathbf{x}_t)$. The $\omega^{(i)}$, called the *importance factor*, are derived based on how likely the hypothesis particle is, given the sensor measurements and known map. The idea is that over time the set of sample particles converge on the true robot pose in the environment. This means that the algorithm needs to discard particles that are unlikely, and keep particles that are likely. Indeed the MCL algorithm can be thought of as featuring two distinct components:

1. **Motion and sensor update**: this phase sees the robot undergo a control action which is applied to its pose, as well as the pose of all particles. Additionally, sensor measurements are obtained, and then these sensor readings are used to determine new likelihood weights, ω, for each of the particles.
2. **Resampling**: this phase allows us to discard particles which are not likely (low weights), and keep particles with high likelihood (i.e. large weights). This is undertaken by randomly sampling m times from the pool of particles, with replacement. Higher ω weights mean it is more likely the particle will feature in the re-sampled set.

The MCL algorithm pseudocode is shown in Algorithm 2 below.

Algorithm 2 Monte Carlo Localisation

```
1: procedure MCL($X_{t-1}$,$u_t$,$z_t$)
2:     $\bar{X}_t = X_T = \emptyset$
3:     for $m = 1$ to $M$ do
4:         $x_t^{[m]}$ = motion_update($u_t, x_{t-1}^{[m]}$)
5:         $\omega_t^{[m]}$ = sensor_update($z_t, x_t^{[m]}$)
6:     for $m = 1$ to $M$ do
7:         draw $x_t^{[m]}$ from $\bar{X}_t$ with probability $\propto \omega_t^{[m]}$
8:         $X_t = X_t + x_t^{[m]}$
9:     return $X_t$
```

2.3 Comparison & Model Selection

Kalman filters and MCL present very different approaches to solving the localisation problem. Kalman filters make underlying assumptions about the distribution of stochastic perturbation to model actuation and sensor noise in order to predict true robot pose. Put simply there is an assumption of Gaussian distributed noise, and as a result the pose posterior is also Gaussian. In contrast, MCL does not make this restrictive assumption and instead takes samples, or particles, to estimate the pose distribution. This means MCL is more robust than a Kalman filter approach given that it can handle situations which violate KF distribution assumptions. The principal drawback of MCL is that it is much more expensive in terms of memory requirements and computational processing. To properly estimate the pose distribution a large number of samples (particles) need to be stored and updated for each discrete time step in the algorithm. The main advantage of MCL over KF is that MCL can recover from wheel slip events, that is, the MCL algorithm can solve the *global localisation* problem. KF, on the other hand, can only be used for *position tracking* - this was the principal motivation for selecting an MCL algorithm for implementation in simulation.

3 Simulations

Gazebo, a physical simulation environment, was used to test an off-the-shelf implementation of MCL on two mobile robot platforms: a benchmark model, and an Alternative model. ROS was used to implement and run the various process nodes which make up the robot model, and RVIZ was used to visualise sensor data, costmaps, and navigation paths. A basic simulated environment was used to test MCL performance for spawned robots - map topology can be seen in Figure 5.

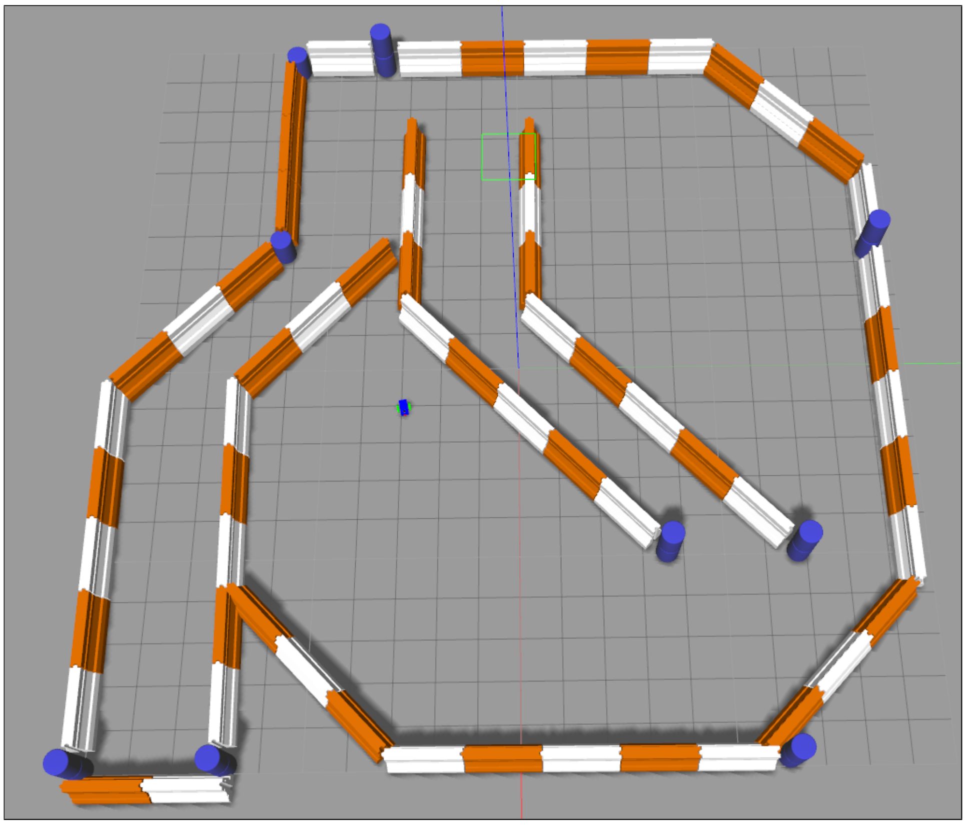

Figure 5: Simple environment topology was used to test the robot model MCL performance

The benchmark robot model was designed for easy implementation of the MCL package with ROS, Gazebo, and RVIZ. The second robot model, based heavily on the benchmark model, was designed to assess how different physical sensor locations impact MCL performance. Further, modifications were made to the physical robot body to determine if robot chassis geometries affect MCL performance. To evaluate MCL performance, each robot was provided an identical goal pose (location and orientation). Robots had to localise in the environment, and navigate to the goal pose. Performance metrics included: time taken to localise; and time taken to reach goal pose.

3.1 Benchmark Model

3.1.1 Model Design

The benchmark robot model consisted of a rectangular prism chassis, along with two cylindrical wheels placed in a differential drive configuration in the middle of the chassis. Frictionless, semi-spherical, casters were placed in symmetrical locations on either end of the chassis to ensure stability. A camera, modelled with a red cube, was placed on the front of the chassis. A laser range finder was placed on top of the chassis, towards the camera location.

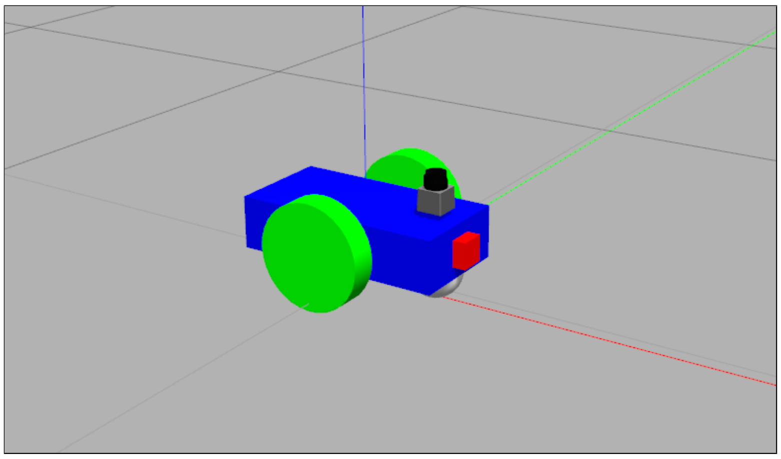

Figure 6: Benchmark robot model is comprised of a rectangular prism chassis, and two wheels arranged in a differential drive configuration on either side of the chassis. The chassis is equipped with a front mounted RGB camera, and a laser range finder. Frictionless casters provide stability underneath the chassis.

Figure 6 shows the robot model loaded into an empty world using Gazebo. The robot was equipped with a differential drive controller, called `libgazebo_ros_diff_drive.so`, which allowed ROS to control the robot wheels. Similarly, controllers for the camera and laser range finder were equipped. These were called `libgazebo_ros_camera.so` and `libgazebo_ros_laser.so`, respectively. The robot dimensions can be seen in Table 1.

Table 1: Descriptive measurement summary for the benchmark model

Dimension	**Measurement** (m)
Chassis Length	0.4
Chassis Width	0.2
Chassis Height	0.1
Wheel Radius	0.1
Wheel Placement	Center

3.1.2 Alternative Model

An alternative model was developed by modifying the benchmark design. The original chassis was retained, however, the cylindrical wheels were shifted to the rear, in a differential drive configuration. Two frictionless semi-spherical casters were set symmetrically at either end of the chassis. The camera was left in its original location.

To provided an elevated location for the laser range finder the height of the robot chassis at the rear was elevated. Figure 7 shows the robot model loaded into an empty world using Gazebo. The robot was equipped with controllers for the differential drive, camera, and laser range finder identical to the benchmark model. Dimensions can be seen in Table 2.

Table 2: Descriptive measurement summary for the Alternative model

Dimension	**Measurement** (m)
Chassis Length	0.4
Chassis Width	0.2
Chassis Height	0.1
Wheel Radius	0.1
Wheel Placement	Rear

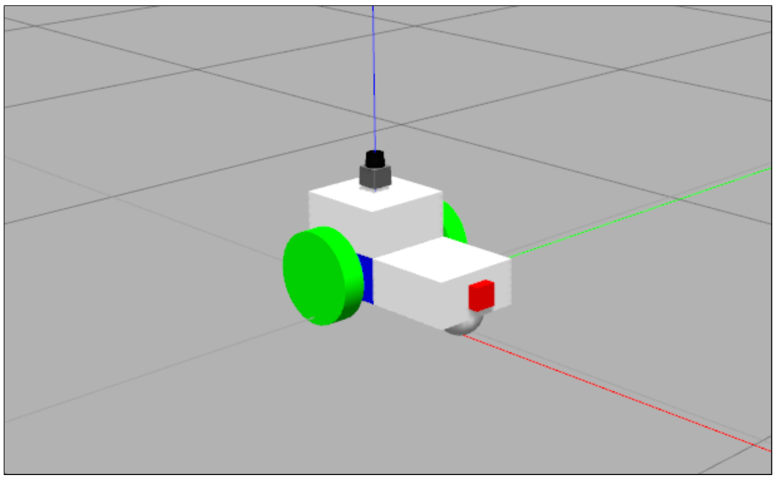

Figure 7: Alternative robot model is comprised of a rectangular prism chassis, and two wheels arranged in a differential drive configuration towards the chassis rear. The model is equipped with a front mounted RGB camera, and a laser range finder. Frictionless casters provide stability underneath the chassis.

3.2 Packages

Two main packages are required to implement MCL in simulation. The first is the `amcl` package - an MCL package that alters the number of particles used to improve algorithm efficiency. The second package is the `move_base` package which is used to generate cost maps and navigation paths for the robot model to reach goal locations. Sections 3.2.1 and 3.2.2 discuss the `amcl` and `move_base` packages, respectively.

3.2.1 `amcl` package

The `amcl` package is a localisation system for a robot traversing 2D map topologies, which is implemented in ROS. The package implements a Monte Carlo localisation approach that uses a particle filter to track the pose of the robot for a known map. Unlike standard MCL algorithms, this package implements an adaptive filter, which changes the number of sampled particles dynamically to ensure that error introduced by the sample-based representation is bounded. Some of the more important parameters for this package include:

- **`min_particles`** and **`max_particles`**: amcl dynamically adjusts the number of particles for each iteration, between some given range. A range with a high maximum may be too computationally expensive, for a given system.
- **`update_min_a`** and **`update_min_d`**: upon receiving a laser scan, amcl checks the linear and angular displacement of the robot from the last laser scan and will only provide a filter update if these values are greater than the values specified in these parameters
- **`transform_tolerance`**: the duration for which a published transform is valid
- **`laser_min_range`** and **`laser_max_range`**: define the minimum and maximum scan ranges that will be considered for the laser range finder
- **`laser_max_beams`**: how many evenly spaced beams in each scan to be used when updating the filter

3.2.2 move_base package

The move_base package provides the robot with a navigation and path planning tool. Given a goal in the world, the move_base package will provide action to reach it with a mobile base. This is comprised of a global path planner, and a local path planner. The move_base node, once initialised, will also maintain two costmaps: one for the global planner, and one for the local planner. These maps are integral to accomplish navigation tasks. The move_base node subscribes to a number of different topics. These include:

- an environment map provided by a map server;
- laser scans from a laser range finder;
- point cloud data from RGBD cameras;
- odometric data from the robot odometry;
- sensor transforms from a transform topic; and
- transform updates from amcl nodes

The move_base node receives a simple goal pose, and outputs a series of commands to the robot's base controller in order to achieve the input goal. The visual graph depicting the navigation stack set up can be seen in Figure XXXX.

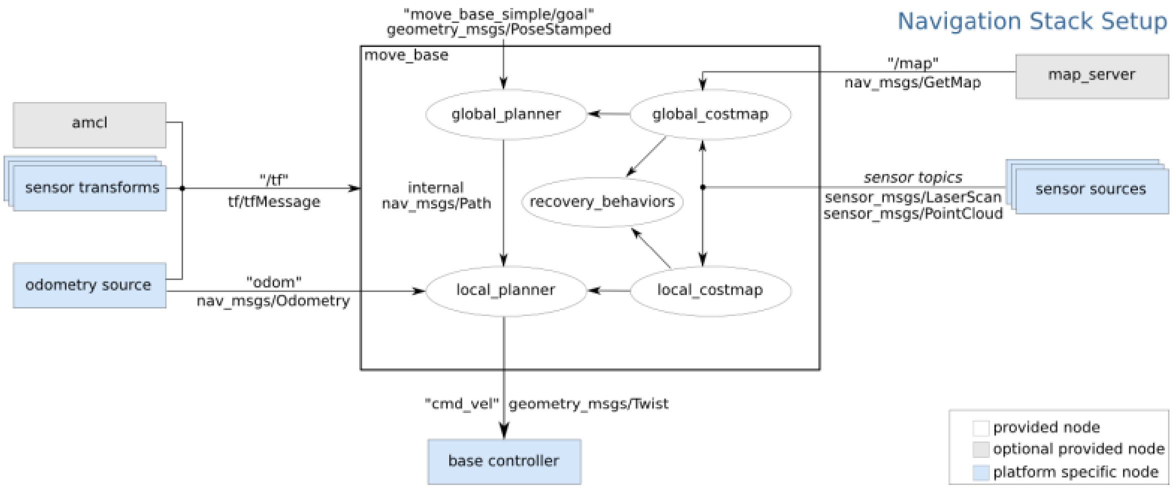

Figure 8: A depiction of the navigation stack used to localise the robot in a known environment using and MCL implementation. The move_base node subscribes to the output of the amcl node. Additionally, a map server provides move_base with a ground truth map. The move_base node also receives odometry and sensor data, and provides output to the control node.

Some of the more important parameters for this package include:

- **controller_frequency**: the rate in Hz at which to run the control loop and send velocity commands to the base
- **update_frequency**: the frequency in Hz for the map to be updated
- **publish_frequency**: the frequency in Hz for the map to be published
- **inflation_radius**: the radius in meters to which the map inflates obstacle cost values
- **obstacle_range**: the maximum range in meters at which to insert obstacles into the costmap using sensor data
- **raytrace_range**: the maximum range in meters at which to raytrace out obstacles from the map using sensor data

3.3 Parameter Tuning

Parameters were tuned for the benchmark model using trial and error. This was undertaken iteratively until a set of parameters were found allowing error free movement over short linear distances. Parameters were tuned one at a time by entering a test value for the model configuration, and then loading the model into the world with active `amcl` and `move_base` nodes. Desired poses were provided a short distance from the original pose, and performance was evaluated visually. This process was continued until satisfactory localisation and navigation behaviours emerged for the benchmark model - the tuned parameter values can be seen in Table 3.

Table 3: Parameter value summary for `amcl`, and `move_base`

Parameter	Value
`controller_frequency`	10.0
`update_frequency`	20.0
`publish_frequency`	20.0
`transform_tolerance`	0.2
`inflation_radius`	0.3
`obstacle_range`	8.0
`raytrace_range`	10.0

4 Results

Robot models were deployed separately in the environment using identical parameter configurations for their `move_base` and `amcl` nodes. Both models successfully generated a costmap, which was used for navigation. The physical environment, the ground truth map, and the local costmap, for the benchmark model, can be seen in Figures 9, 10, and 11, respectively. Robots, provided with a goal pose, were required to localise in the environment and navigate to the goal pose. Successful localisation saw particle convergence around the correct robot pose, which was determined by visual inspection. Time taken to localise, and time taken to reach the goal were measured. Each robot underwent five separate trials.

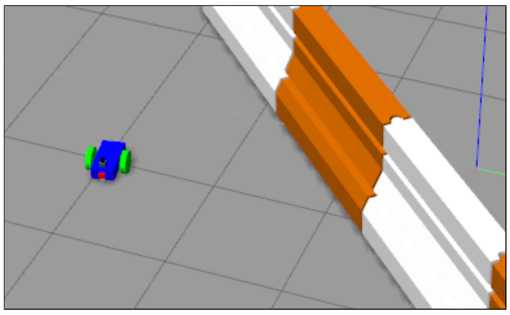

Figure 9: Actual physical environment. The robot is located near a barrier in the environment.

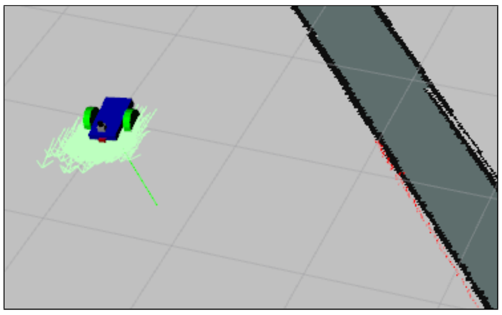

Figure 10: The ground truth map which is provided to `move_base` via a map server node.

The benchmark model was able to localise in all five trials and avoided collisions with environment obstacles. The robot became stuck in two trials, and was unable to recover. Interestingly, the robot did not get stuck on the environment obstacles, rather, it became stuck on an *invisible* barrier. The remaining three trials saw the robot successfully navigate to the desired goal pose. The average localisation time over the three successful trials was 0.49 seconds, and the average time to goal was 1 minute and 36 seconds. The full set of results can be seen in Table 4.

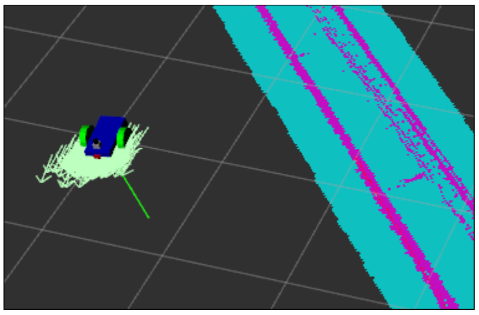

Figure 11: A local costmap generated by the `move_base` node inflates objects to help avoid collisions.

The Alternative model was able to localise in all five trials and avoided collisions with environment obstacles. The robot became stuck in three trials, and was unable to recover. As with the benchmark model, the robot did not become stuck on actual physical obstacles. The remaining two trials saw the robot successfully navigate to the desired goal pose. The average localisation time over two successful trials was 0.375 seconds, and the average time to goal was 1 minute and 21 seconds. The full set of results can be seen in Table 4.

Table 4: Five independent trials in which the robot was required to navigate to a given goal pose for benchmark and Alternative models. Listed results show the time taken to localise and the time taken reach goal pose.

	Benchmark Model		**Alternative Model**	
Trial	**Localisation**	**Goal**	**Localisation**	**Goal**
1	0.48	1.36	`stuck`	`stuck`
2	0.47	1.36	`stuck`	`stuck`
3	`stuck`	`stuck`	`stuck`	`stuck`
4	0.52	1.37	0.39	1.21
5	`stuck`	`stuck`	0.36	1.21

Benchmark model performance, during a successful run, can be seen in Figures 12, 14. and 16. Figure 12 shows the benchmark model initialised in the environment. The green arrows represent the distribution of particles used by the `amcl` node. Note they are uniformly distributed around the robot. Figure 14 shows the robot navigating towards the goal location - the green arrows are starting to converge on the robots true pose within the environment map. Finally, Figure 16 shows the robot at the goal pose. The arrows are tightly clustered around the robot demonstrating that `amcl` has localised the robot within the environment. Figures 13, 15, and 17 show the same sequence of events for the Alternative robot model on a successful run.

A YouTube video of the Benchmark robot model successfully localising in the environment and reaching the desired goal location can be seen here:

`put_in_url`

A YouTube video of the Alternative robot model successfully localising in the environment and reaching the desired goal location can be seen here:

`put_in_url`

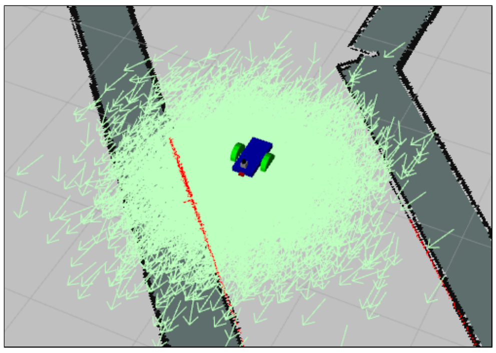

Figure 12: Benchmark model before the localisation process has commenced. Note the distribution of green pose arrows around the robot indicating a high level of pose uncertainty.

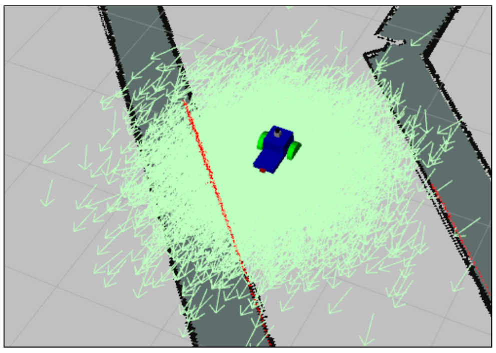

Figure 13: Alternative model before the localisation process has commenced. Note the distribution of green pose arrows around the robot indicating a high level of pose uncertainty.

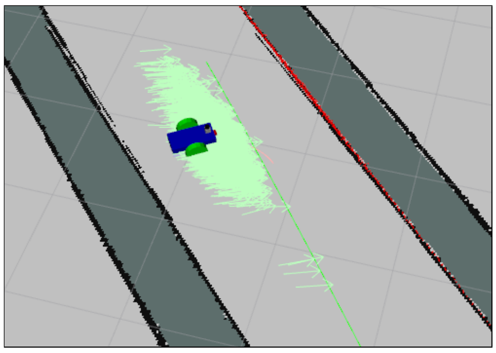

Figure 14: Benchmark model navigating the environment as `amcl` localises the robot. The distribution of green arrows is beginning to converge on the correct pose.

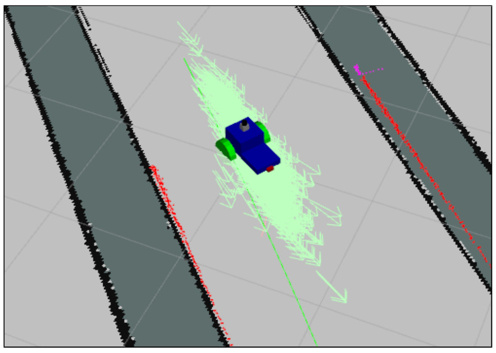

Figure 15: Alternative model navigating the environment as `amcl` localises the robot. The distribution of green arrows is beginning to converge on the correct pose.

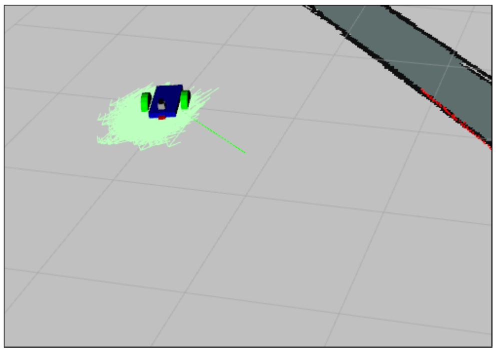

Figure 16: Benchmark model at the goal location. The tight clustering of green arrows indicates particle convergence on true robot location for `amcl`

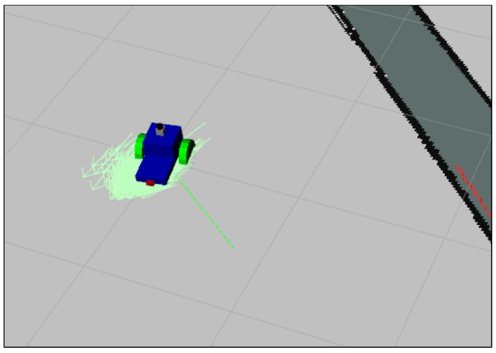

Figure 17: Alternative model at the goal location. The tight clustering of green arrows indicates particle convergence on true robot location for `amcl`

5 Discussion

The benchmark model appears the most stable as it was able to navigate to the goal pose more reliably than the Alternative model. It must be noted, however, that this may simply be an artefact of the stochastic initialisation and stochastic pose evolution of the particles that are used in the `amcl` node. The `amcl` node randomly initialises the distribution of particles, and each discrete time step adds random perturbations to particle movement, and sensor measurements. Given that the `amcl` outputs to the `move_base` node, it may be the case that both models are equally likely to get stuck - more trials need to be undertaken to determine if both models are equally likely to get stuck. No immediate reason is presented as to why the robot models are getting stuck. ROS does not flag any errors or warnings. Possible causes include poor parameter configuration, or a lack of adequate system resources. Further experimentation is required to determine the cause.

Time to localisation results indicated that both robot models performed similarly suggesting `amcl` is not materially affected by modifications to robot geometries and sensor locations. Time taken to reach the goal location was noticeably different. On average, the Alternative robot model was able to reach the goal location 15 seconds faster than the benchmark. There is no apparent reason why `move_base` provided more efficient navigation paths for the Alternative model. Observed time differences may again be due to the stochastic nature of `amcl`. To further understand this result, more trials need to be undertaken. Additionally, different starting locations could be used, which may lead to more robust conclusions.

6 Conculsion

Two robot models with differing body geometries and sensor locations were tested to determine the robustness of `amcl` and `move_base` packages, used to implement MCL on a mobile robot platform. Experimental simulation provided some evidence that time taken to localise within an environment is not affected by changes to robot geometries and sensor locations, however, this was not the case for algorithm stability and navigation path efficiency. The Alternative model appeared to experience reduced stability becoming stuck in the environment more often, but was able to navigate to the goal pose faster than the benchmark model. Ultimately these findings are inconclusive, and more experimental trials need to be undertaken to determine if results were due to stochastic variation or not.

References

[1] I. J. Cox, "Blanche: An experiment in guidance and navigation of an autonomous robot vehicle," *IEEE Trans. Robotics Automat*, vol. 7, pp. 193–204, 1991.

[2] C. M. Wang, "Location estimation and uncertainty analysis for mobile robots," *Proceedings from IEEE International Conference on Robotics and Automation*, pp. 1231–1235, 1988.

[3] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localisation for mobile robots," *Artificial Intelligence*, 2001.

[4] S. Thrun, D. Fox, and W. Burgard, "Markov localisation for mobile robots in dynamic environments," *Journal of Atrificial Intelligence Research*, vol. 11, pp. 391–427, 2001.

[5] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME - Journal of Basic Engineering*, vol. 82 (Series D), pp. 35–45, 1960.