# ADAPTING MONTE CARLO LOCALIZATION TO UTILIZE FLOOR AND WALL TEXTURE DATA

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Stephanie Krapil

September 2014

COMMITTEE MEMBERSHIP

| | |
|---|---|
| TITLE: | Adapting Monte Carlo Localization to Utilize Floor and Wall Texture Data |
| AUTHOR: | Stephanie Krapil |
| DATE SUBMITTED: | September 2014 |
| COMMITTEE CHAIR: | Associate Professor John Seng, Ph.D.<br>Department of Computer Science |
| COMMITTEE MEMBER: | Professor Franz Kurfess, Ph.D.<br>Department of Computer Science |
| COMMITTEE MEMBER: | Assistant Professor Foaad Khosmood, Ph.D.<br>Department of Computer Science |

ABSTRACT

Adapting Monte Carlo Localization to Utilize Floor and Wall Texture Data

Stephanie Krapil

Monte Carlo Localization (MCL) is an algorithm that allows a robot to determine its location when provided a map of its surroundings. Particles, consisting of a location and an orientation, represent possible positions where the robot could be on the map. The probability of the robot being at each particle is calculated based on sensor input.

Traditionally, MCL only utilizes the position of objects for localization. This thesis explores using wall and floor surface textures to help the algorithm determine locations more accurately. Wall textures are captured by using a laser range finder to detect patterns in the surface. Floor textures are determined by using an inertial measurement unit (IMU) to capture acceleration vectors which represent the roughness of the floor. Captured texture data is classified by an artificial neural network and used in probability calculations.

The best variations of Texture MCL improved accuracy by 19.1% and 25.1% when all particles and the top fifty particles respectively were used to calculate the robot's estimated position. All implementations achieved comparable performance speeds when run in real-time on-board a robot.

# ACKNOWLEDGMENTS

Many thanks to:

- Dr. John Seng for advising me on this project.
- My parents for their unconditional support.
- The Cal Poly Robotics Club for the use of their club room and for owning a large quantity of tablecloths.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

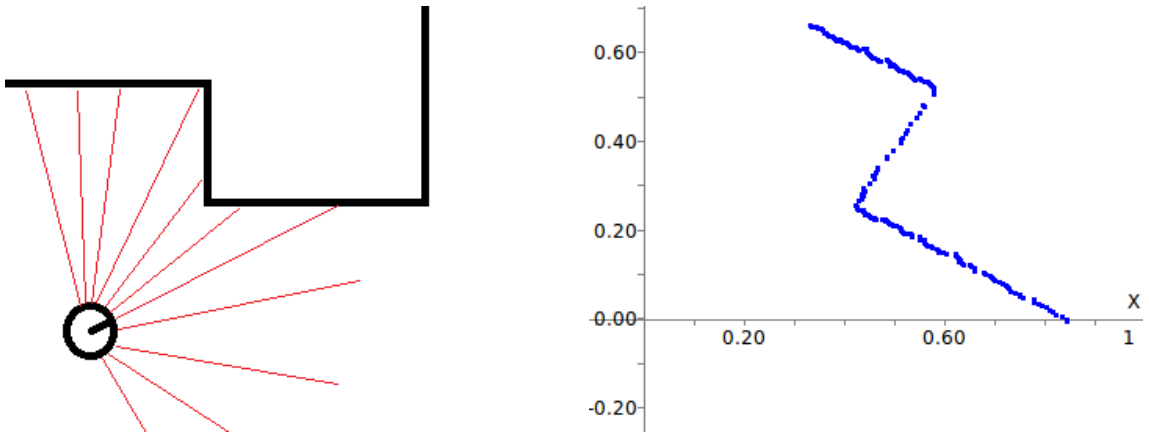
# CHAPTER 1

## Introduction

With interest in commercial robots growing, there is a constant search to improve robot navigation. An important part of this problem is locating the physical position of the robot. Robot localization is a problem defined as estimating a robot's location and orientation, which together are referred to as a robot's pose. To do this, a robot compares the data coming back from its sensors to a preexisting map.

The problem of localization is simple: have a robot locate itself given a map of the surrounding area. Unfortunately, the solution is much more complex than the problem. Robots rely on sensors to view the world around them, but sensors can only return numeric values. This means that to locate itself, the robot must first be able to extract information on its surroundings from a series of numbers. A common example is extracting straight lines from arrays of distance data, such as seen in Figure 1. Lines can be recognized as walls, and often form other landmarks such as corners or doorways.

Once that is done, the data must be compared to a map. However, this brings up the problem of representing a map so the robot can understand it. A common method, called a grid map, represents an area as a grid, and for each square identifies it as free or occupied. Comparing the robot's sensor data to this map is tricky, as the robot could be at any location and facing any direction and calculations need to be done in real-time so the robot does not crash.




(a) Robot scanning in building (b) Laser scan from robot

**Figure 1.1: Example scan from a laser range finder**

There are different variations of the robot localization problem. The first is global localization, where the robot has a map of the area but has to figure out its position without any idea of its position. The second is position tracking, where the robot begins with knowledge of its approximate position and then tries to account for sensor and motor errors while driving. While global localization and position tracking are the two main types of localization, another interesting problem in the field is that of the kidnapped robot [19]. This localization problem occurs when the robot has determined its position, but is then kidnapped by a human and taken to a new position. Since the robot believes itself to be at the old position, it must be able to recognize that its position is no longer correct and restart global localization to determine its new position. This particular thesis will be focusing on the global localization and position tracking problems.

One obvious solution for keeping track of a robot's position is a GPS sensor. However, GPS cannot completely solve the robot localization problem for two reasons. First, GPS sensors are typically accurate down to a meter. When working with a robot, that meter could be the difference between a clear path and one obstructed by a tree or wall. Second, GPS only tracks position  not orientation. While one might argue orientation could be estimated from previous GPS readings, it would not be

perfectly accurate. Consider the case of a robot spinning in place. For that case, the GPS will be remaining roughly the same but orientation will be changing constantly between each sensor reading. Theoretically, the robot could calculate how far it has turned based upon encoders or its wheels' velocities, but wheels are prone to slippage which adds error to the calculations and sensor readings can be corrupted by noise. Error could recreate the first situation, where the robot thinks it is facing a clear path when it is in truth facing an obstruction.

There are several algorithms for localizing a robot. The main ones can be sorted into three categories: Kalman filtering [18, 19], Markov localization [2, 19], and Monte Carlo methods [5, 4, 19]. In [7], Gutmann and Fox determined that adaptive Monte Carlo Location (AMCL) largely outperforms combined Markov localization and Kalman filtering methods in terms of accuracy under standard conditions and is better at dealing with sensor noise. With those two conditions, it is the method chosen on which to build this thesis work.

This thesis improves on MCL by adding textures to improve the algorithm accuracy. Ten variations of MCL utilizing combinations of wall textures and floor textures along with different weighting schemes were implemented and tested both for performance and accuracy.

The remainder of this work is organized as follows. The next section presents background information on Monte Carlo Localization and identifying surface textures of floors and walls. Section 4 describes an adaption to Monte Carlo Localization that integrates classification of floor and wall surfaces. Section 5 details the experimental setup, while Section 6 discusses the results. Finally, the paper concludes by describing what can be taken away from this work and possible avenues for future research.

# CHAPTER 2

# Background

As mentioned previously, there are several different algorithms for localizing a robot. This thesis focuses on Monte Carlo Localization. Monte Carlo Localization has many advantages. It can be used to solve global localization, position tracking, and the kidnapped robot problems. It uses less computing resources than other algorithms. It is able to keep track of multiple location predictions at once. Section 2.1 will describe the algorithm.

The goal of this thesis is to improve the accuracy of MCL, meaning the location the algorithm predicts the robot is at will be closer to the physical location of the robot. Since traditionally MCL only uses information on the position of landmarks for localization, data on the surface texture of the surrounding environment will be added into the algorithm to improve accuracy. The focus will be on identifying floor and wall textures of the surrounding environment, as discussed in Section 2.2.

## 2.1 Monte Carlo Localization

Monte Carlo Localization (MCL) was first proposed by Fox, Burgard, Dellaert, and Thrun in [5]. Unlike previous algorithms, which represented the probability of the robot being in every possible position, MCL simplifies the representation into a sample set of possible robot locations. Most algorithms used single-modal distributions (i.e. Gaussian) to model the probability of a robot being at a particular position on the map, which left them unable to represent multiple likely locations. Using a sample

set of locations allows MCL to model multi-model probabilities, which means the robot could track several likely positions at once. This makes MCL especially suited for solving the global localization problem. Since its proposal in 1999, Monte Carlo Localization has become a popular method for solving the localization problem.

### 2.1.1 Algorithm

Localization is the process of determining the probability of the robot being at a pose given its sensor inputs and previous movements. MCL represents this probability as a sample set of robot poses, which are called particles. This section will detail the Monte Carlo Localization algorithm as described in [5, 4, 19].

The probability density $p(x_k|Z_k)$ is represented by a set of N random samples. It represents the probability that a robot is at location $x_k$ given the input from its sensors, $Z_k$. Since the probability density function (PDF) and sample sets are derived from the same probability distribution, it is possible to generate a set of samples from the probability density function or the probability density function from a set of samples. As such, it is able to use less computational resources representing the probability than non-MCL methods.

The core of MCL is the idea of sampling and importance resampling from the set of particles. At each iteration of the algorithm, resampling occurs until the remaining particles are highly likely to represent the robot's actual location.

In each iteration of MCL, two steps are performed. The first starts with a set of particles, $S_{k-1}$, which is generated from the previous iteration or, if starting, drawn from a probability density equally distributed over the environment. For each particle in the set, a motion model (discussed further in Section 2.1.2) is applied to translate each particle according to the robot's actual movement.

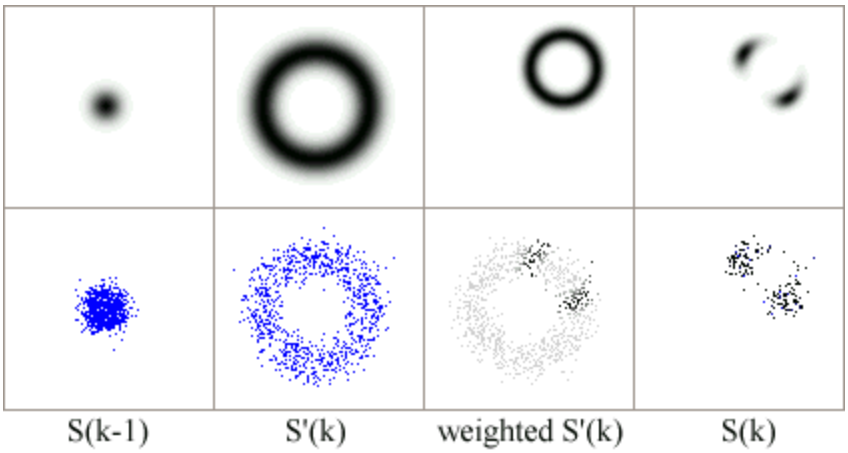



**Figure 2.1: Probability densities (top) and corresponding particles sets (bottom) for one MCL iteration** [4]

In the second step, each translated particle is weighted according to sensor readings. If the sensor readings match what the robot would see if it were at a particular particle location, it is weighted higher than if there is no match. From this set of weighted particles, a new set of particles is drawn, which become the starting set for the next iteration of the algorithm. The two steps are repeated until the particles converge on the robot's location.

Figure 2.1 shows one iteration of MCL. It starts with a cluster of particles, all near the same location but having different orientations. In the second panel, the robot moves forwards and the particles are translated accordingly. In the third, the particles are weighted according to the robot's sensor readings, with the darker particles being more likely. In the final panel, the sample set for the next iteration is pulled from the weighted samples.

The basic algorithm is shown in Algorithm 2.1.1. Lines 3 - 7 implement the first step of MCL with motion and sensor models that represent the robot. Lines 8 - 11 implement the second step of MCL. One thing to note is in line 8 all weights are normalized so $\Sigma wt_{[i]} = 1$.

**Algorithm 2.1.1** Monte Carlo Localization [19]

1: **function MCL**$(X_{t-1}, u_t, z_t, m)$
2: $\quad \bar{X}_t = X_t = \emptyset$
3: $\quad$ **for p = 1 to P do**
4: $\quad\quad x_t^{[p]} = sample\_motion\_model(u_t, x_{t-1}^{[p]})$
5: $\quad\quad w_t^{[p]} = measurement\_model(z_t, x_{t-1}^{[p]}, m)$
6: $\quad\quad \bar{X}_t = \bar{X}_t + < x_t^{[p]}, w_t^{[p]} >$
7: $\quad$ **end for**
8: $\quad$ **for p = 1 to P do**
9: $\quad\quad$ *draw i with probability* $\propto w_t^{[i]}$
10: $\quad\quad$ *add* $x_t^{[i]}$ *to* $X_t$
11: $\quad$ **end for**
12: $\quad$ **return** $X_t$
13: **end function**

As MCL continues to run, it narrows down its prediction to one likely pose. Unfortunately, if this pose is incorrect there is no way for the algorithm to recover. To solve this problem, a variation of MCL, called Adaptive Monte Carlo Localization (AMCL) is used [7, 19]. This variant tracks the short-term and long-term average of the robot's probability and adds random samples if the robot's current particle set is not properly representing its position.

The code for AMCL is shown in Algorithm 2.1.2. The first step of MCL remains roughly the same, aside from calculating the average probability. Lines 10 and 11 update the short- and long-term probabilities, which is then used in line 13 to add random samples as an alternative to the original resampling step. AMCL is implemented as the baseline used in experiments.

### 2.1.2 Kinematics

MCL can be used interchangeably with different robot kinematics models. A kinematic model translates the movement of a robot's wheels into motion. It also represents that motion in a global plane. The model is used in Algorithms 2.1.1 (line

**Algorithm 2.1.2** Adaptive Monte Carlo Localization [19]

1: **function AMCL**$(X_{t-1}, u_t, z_t, m)$
2: **static** $w_{slow}, w_{fast}$
3: $\bar{X}_t = X_t = \emptyset$
4: $w_{avg} = 0$
5: **for p = 1 to P do**
6: $x_t^{[p]} = sample\_motion\_model(u_t, x_{t-1}^{[p]})$
7: $w_t^{[p]} = measurement\_model(z_t, x_{t-1}^{[p]}, m)$
8: $\bar{X}_t = \bar{X}_t + < x_t^{[p]}, w_t^{[p]} >$
9: $w_{avg} = w_{avg} + \frac{1}{P} w_t^{[p]}$
10: **end for**
11: $w_{slow} = w_{slow} + \alpha_{slow}(w_{avg} - w_{slow})$
12: $w_{fast} = w_{fast} + \alpha_{fast}(w_{avg} - w_{fast})$
13: **for p = 1 to P do**
14: **if probability is** $max(0.0, 1.0 - \frac{w_{fast}}{w_{slow}})$ **then**
15: **add random pose to** $X_t$
16: **else**
17: **draw** $i \in \{1, \ldots N\}$ **with probability** $\propto w_t^{[i]}$
18: **add** $x_t^{[i]}$ **to** $X_t$
19: **end if**
20: **end for**
21: **return** $X_t$
22: **end function**

4) and 2.1.2 (line 6) to translate the particles according to the robot's movement. For instance, if the robot drives forward for 1 m, all of the particles would shift their position in the direction they are oriented by 1 meter.

Kinematic models vary based on a robot's form of locomotion (i.e. wheels, legs, treads) and how they are mounted on the robot. Kinematics for a differential drive robot will be explained here as that is the wheel setup used for experimentation. [17] has a more detailed explanation. A differential drive robot has two fixed wheels, one on each side of the robot. Each wheel may be given separate motor commands and operates independently of the other.

There are two different reference frames against which robot motion is measured. The first is the global reference frame. The global reference frame is the coordinate-

axis corresponding to the plane the robot exists on. The second is the local reference frame, which corresponds to a coordinate-plane located at the robot's location. A diagram of the two frames is pictured in Figure 2.2. $\theta$ denotes the angular distance between the global and local reference frames, and the robot's location is indicated by a single point P.

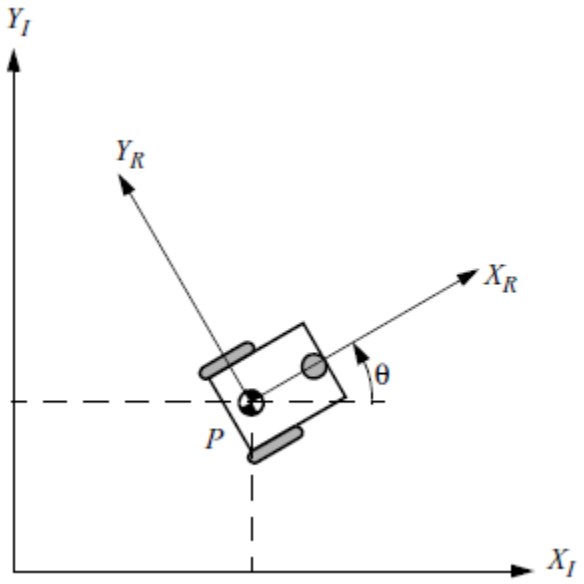



**Figure 2.2: The global reference frame (I) and the robot local reference frame (R) [17]**

To solve for the robot's final location, the robot's motion in the local reference frame must be converted to a location in the global reference frame. To do this, the robot motion is viewed as a single translation to update its location and a single rotation to update its orientation. This can be calculated as a function of the

robot's current pose using the orthogonal rotation matrix, shown in Equation 2.1. The orthogonal rotation matrix is a function of θ.

$$R(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

A differential drive robot has two wheels of diameter d. Assuming P is a point centered between the wheels, then l is the distance from a wheel to P. The spinning speed of the wheels are $\varphi_1$ and $\varphi_2$ for the right and left wheels respectively. By combining these with the orthogonal rotation matrix to account for the disparity between local and global reference frames, it is possible to calculate the position of the robot in the global reference frame.

$$\xi_I = R(\theta)^{-1}\xi_R \tag{2.2}$$

Both wheels of a differential drive robot contribute to the translation speed of P along the x-axis in the local reference frame. Adding the individual contributions of the wheels provides how far the robot location is translated. If one wheel moves while the other remains stationary, P will move with half the speed, and the robot will pivot around its stationary wheel. All movement is along the local reference x-axis since a differential drive robot is only capable of moving forwards or backwards. Since the robot is incapable of sideways movement, movement along the local reference y-axis will always remain 0.

To determine the rotational portion of the robot's movement, the wheels' speeds must be converted to angular motion (Equations 2.3 and 2.4). Since forward movement of the left wheel will result in clockwise motion while movement of the right

wheel results in counterclockwise motion, one of the angular velocities will be negative to account for opposite directions.

$$\omega_1 = \frac{r\varphi_1}{2l} \tag{2.3}$$

$$\omega_2 = \frac{-r\varphi_2}{2l} \tag{2.4}$$

Like the translational speeds, once the individual contributions of the wheels have been calculated, they are added together to determine the final rotational position of the robot, yielding Equation 2.5. This equation provides the final x, y, and θ that represent the robot's position in the global reference frame.

$$\xi_I = R(\theta)^{-1} \begin{bmatrix} \frac{r\varphi_1}{2} + \frac{r\varphi_2}{2} \\ 0 \\ \frac{r\varphi_1}{2l} + \frac{-r\varphi_2}{2l} \end{bmatrix} \tag{2.5}$$

## 2.2 Floor and Wall Classification

Terrain classification is the problem of identifying the surface a robot is driving on. An example would be distinguishing between concrete, grass, asphalt, or gravel. Terrain classification is an important problem in the field of robotics. By identifying a surface, it is possible to improve the control algorithms of a robot to make it more accurate or efficient. For instance, if the robot is known to slip more on gravel, a different steering method could be utilized to reduce slippage.

When describing terrain, words like smooth, bumpy, or rough are used. It is difficult to translate these into something a robot can understand, since there is no such thing as a tactile sensor. Instead, other sensors are used to make up for the lack.

Laser range finders, cameras, inertial measurement units (IMUs), and more have all been used in attempts to differentiate between terrain types. Laser range finders can be used to determine the distance to the nearest object, and the difference in distance between bordering laser beams can identify how smooth a surface is [11, 10, 15, 9, 1]. Cameras often focus on the color to differentiate between textures [16, 6, 11, 14]. IMUs are capable of returning acceleration and position vectors, which can be compared to determine what surfaces a robot is traveling on [8, 20].

There is a wide variety of ways to solve problems of terrain classification. At its core, the classification is a form of pattern matching. Some of the methods used include boosting [11, 9], neural networks [8, 14], and computer vision methods [16, 6, 11, 14]. Often the data will undergo statistical analysis [20] or modifications such as fuzzy logic [8] to help with classification. Some of the different variations will be discussed in Section 3.

# CHAPTER 3

## Related Work

Adding terrain data to improve MCL is a relatively new idea. Many of the implementations, such as those described here, rely on camera or other vision based systems. This section will describe two existing MCL implementations that utilize visual data and then describe a number of terrain classification implementations that are able to classify surroundings without the use of images.

In [16], images are used to help MCL adapt to a dynamic environment. As the robot drives, pictures are taken and stored in a database. Feature matching between the current image and stored images is performed, which is then used to calculate how similar the current robot pose is to the one the image was taken at. Based on the difference, particles near the saved location are injected into the algorithm to help the robot narrow down its position. The algorithm reduces the number of times a robot gets lost due to obstruction of its surroundings.

The work of [6] is the closest to what this thesis is attempting. An omni-directional camera is used to create a 3D texture map of the environment the robot is operating in. The map is a series of images that represent the surrounding area. Comparison between textures is computed via a pairwise pixel comparison, and they are considered to match if the difference falls beneath a certain threshold. The modified algorithm converges faster than a distance-only MCL implementation.

In [8], terrain classification is accomplished by filtering acceleration and angular velocity vectors with fuzzy logic. The output of the fuzzy logic is then fed into a

neural network which classifies terrain as flat, rugged, grassy, inclined, or unclassified. 100% accuracy was achieved for flat, rugged, and inclined surfaces, while grass was recognized with 80% accuracy.

[20] utilizes an IMU strapped directly to a robot to gather information on acceleration and angular velocity to extract features. The features are then narrowed down using Sequential Forward Floating Feature Selection and fed into a Linear Bayes Normal Classifier which determines terrain type. The features and classifier are modified based on the velocity of the robot, which helps it to adapt.

[11] uses both exteroceptive sensors, a LIDAR and camera, to identify a terrain offline and a proprioceptive sensor, an IMU, to identify a terrain based on how a robot reacts to it. A classifier for each is trained using boosting, specifically the AdaBoost algorithm. Good results were achieved for each classifier separately, and the authors hypothesize that data correlation could be used to further the work.

In [14], a laser that projects a stripe of light onto the ground is combined with a camera to create a vision system that can operate in varying levels of brightness. Based on how the light projected is disrupted, a probabilistic neural network classifies it as safe or unsafe for driving.

In [10], a laser range finder is used to identify walls of a building for a gondola-type service robot. A Kalman filter is used to differentiate between windows and wall surfaces. The resulting surface type is then used to adjust the robot's control algorithm for traversing the walls.

[9] uses a single laser range finder scan to locate people in an environment based on their shapes. The work tests both circle fitting and boosting techniques against a bounding box variation. The boosting technique used the AdaBoost algorithm and out performed the other two variations.

[15] uses a laser range finder to categorize indoor environments, such as offices, laboratories, or kitchens. In addition to the standard distances to the nearest objects obtained by the laser, reflectance data is also utilized. The 3D laser scans combined with reflectance data allows histograms of local binary patterns to be formed, which are combined into a feature vector and classified using support vector machines.

In [1], a subclass of Markov Random Fields are used to efficiently segment 3D laser scans. Based on training data, the algorithm is able to classify each scan point separately. One example, on a dataset obtained from outside data, is able to differentiate between buildings, trees, shrubs, and the ground.

# CHAPTER 4

## Adapting Monte Carlo Localization

AMCL utilizes a map of positions of a building's walls to allow for indoor robot localization. This thesis adds additional maps to incorporate textures of the floor surfaces the robot is driving on and the wall surfaces directly in front of the robot. By adding this additional information, particles located on textures that match the ones the robot is driving on or where the robot is sensing them will be weighted higher. This will allow more incorrect particles to be eliminated at earlier iterations of the algorithm. The following subsections will detail the floor texture and wall texture maps added to AMCL.

### 4.1 Floor Texture MCL

The first concept this thesis explored is how identifying floor surface textures could affect MCL. Theoretically, by identifying the type of surface a robot is driving on (i.e. tile, carpet) it would allow particles located on dissimilar surfaces to be eliminated.

Floor surface identification is done using sensor readings from an inertial measurement unit (IMU). A single reading of acceleration vectors is sufficient to distinguish between surfaces in the test area.

### 4.1.1 Floor Texture Map

For each floor texture, the area in which it exists on the map must be defined. These could be defined in a grid map, but that would be memory consuming. Instead, to do this, the boundary of each texture area will be defined. However, these floor areas are often irregular shapes that are not conducive to defining the boundaries of.

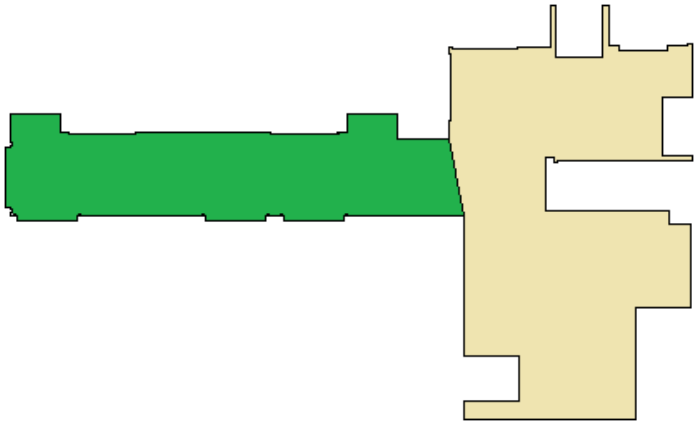
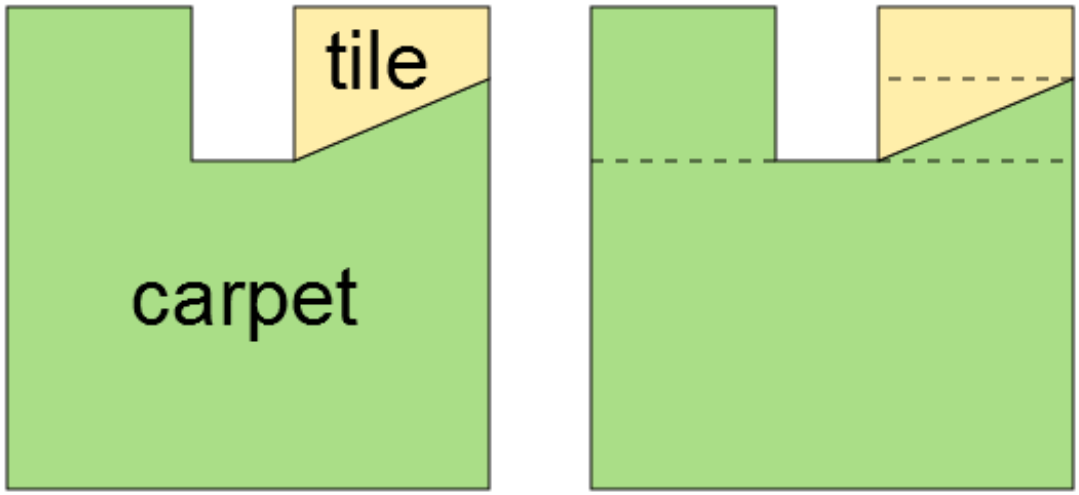


**Figure 4.1: Test area with carpet (green) and tile (tan)**

Ideally, it would be possible to define each unbroken surface as one complete area. Creating a map would be easy if you could just assume each area is a rectangle and ignore the irregularities in the walls. However, consider the following case: Figure 4.1. First, the area containing carpet is bounded by a non-rectangular region. This could be solved by defining a linear function to bound the region and assume everything greater than the function line is tile while anything less is carpet. Yet consider how that line would extend past the wall. If extended, it would incorrectly define part of the tiled area as carpet.

As seen in the example, approximating the texture areas will not work. Because of that, each irregularity in the wall must be defined.

The easiest way to do this is to divide the floor areas into smaller areas that can be more easily defined. For the sake of this thesis, it is assumed the smaller areas will




(a) Floor surfaces (b) Division into shapes

**Figure 4.2: Sample division of floor area consisting of two different surfaces into shapes**

be right triangles or rectangles. For both shapes, at least one side will be parallel to the x-axis of the map and another with the y-axis of the map. This allows for two advantages. First, the side of each shape is a straight line, which can be defined by a linear function. Second, with both shapes it will be possible to define bounds for the x,y coordinates. This is important because it allows a particle to be easily checked against an area of floor surface to determine if it falls within that area and obtain the texture of the area. Figure 4.2 shows an example of a building with two different floor surfaces and how it can be divided into shapes to represent the textures.

Thus, the floor of a building will be saved as a list of rectangles and right triangles. The shapes are defined by linear functions and bounds are defined for both the x and y components. Each shape will also be affiliated with a texture.

### 4.1.2 Comparing Floor Texture

Retrieving the floor surface beneath a particle is straight forward. The particle is checked against the list of rectangles and right triangles until it is found within the

bounds of a shape. Once that shape has been identified, the texture classification corresponding to that shape is returned or -1 if the pose falls outside map boundaries. See Algorithm 4.1.1.

**Algorithm 4.1.1** Algorithm used by FloorTextureMap to retrieve the texture of provided robot pose

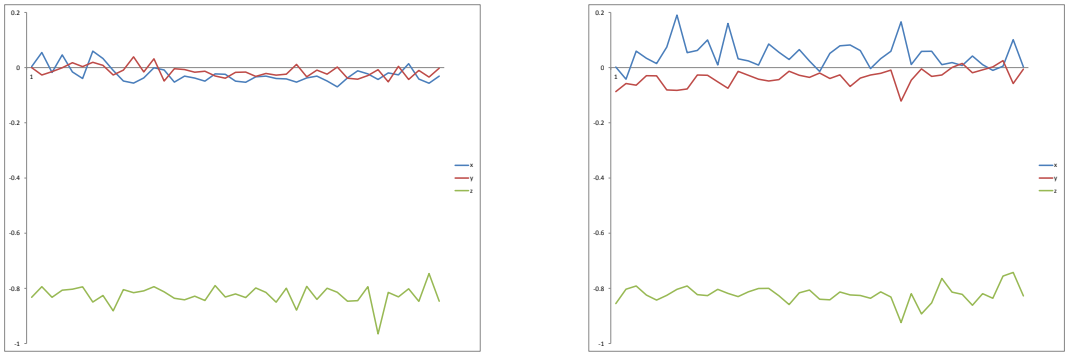
```
1: function getFloorTexture(pose)
2:     for all floor sections do
3:         if pose is located in floor section then
4:             return corresponding texture
5:         end if
6:     end for
7:     return −1
8: end function
```

The returned texture can then be compared against the floor surface the robot is driving over. Depending on whether or not the surfaces are identical, the particle will be weighted accordingly. This is further described in Section 4.3.

### 4.1.3 Artificial Neural Network Training

Since some IMU readings, like accelerations, could be affected by robot movement, training sets include readings taken from different robot motion. This will include while the robot is driving forwards, spinning in place clockwise, spinning in place counterclockwise, turning right, turning left, and driving backwards.

The ANN takes an input of acceleration vectors in the x, y, and z directions, as seen in Figure 4.1.3. The vectors are obtained straight from an instantaneous reading of the IMU mounted on the robot and are unmodified. The network produces a binary output, which identifies the surface as carpet or not carpet. Since the floor of the test area consists of only two textures, this equates to identifying the surface as tile or carpet.



(a) Carpet (b) Division into shapes

**Figure 4.3: Sample IMU acceleration vectors (Gs)**

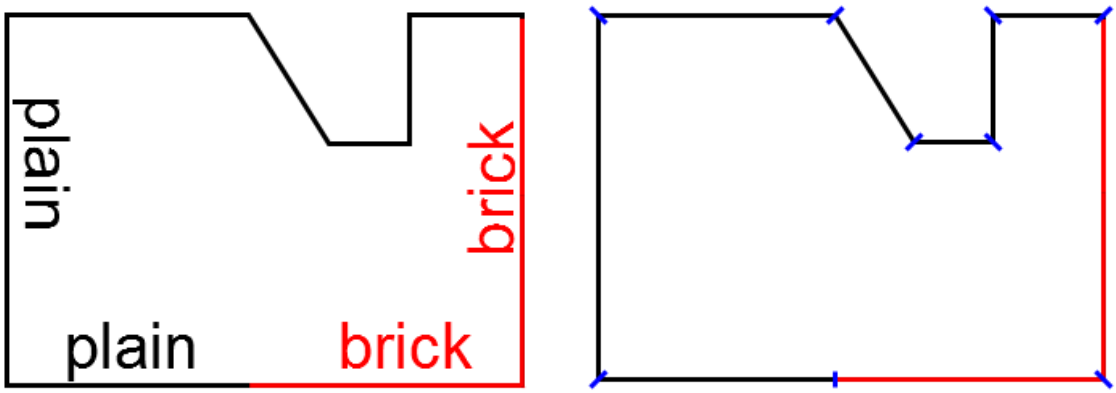
Three layers make up the ANN: an input layer, a hidden layer, and an output layer. Both the input and the hidden layers consist of three neurons, while the output layer consists of one neuron. After training, the network was able to classify test data with ~7% mean squared error.

## 4.2 Wall Texture MCL

The second concept this thesis explored is how analyzing the surface of a wall could affect MCL. Theoretically, by utilizing the laser range finder to detect a pattern in the wall surfaces, particles could be eliminated quicker. If the robot can use its laser to identify it is staring at, for instance, a brick wall, all particles that are facing a wall of some other surface could be eliminated.

### 4.2.1 Wall Texture Map

To save all of the wall textures, the wall locations must be saved. Rather than populating a full grid map, bounded functions are saved which identify where the walls would be located on a grid map. Linear functions were used for ease of implementation. This is viable because most buildings do not contain curved walls. If the case




(a) Wall map with different surfaces (b) Division into walls

**Figure 4.4: Sample division of Wall Texture Map into list of walls**

of a curved wall in a building does arise, a curve could be approximated by a series of linear functions.

An important aspect of saving the linear functions is identifying their bounds. By limiting the functions, it is possible to describe where the walls start and stop on the grid map. Since each wall is essentially representing which squares would be filled on a grid map, bounds are inclusive.

A wall is defined as two things. It will either consist of uninterrupted plane from end point to end point or will be divided so each segment only consists of one texture. Figure 4.2.1 shows an example of the latter where the wall surface changes in the middle of the bottom wall. That bottom wall would be saved as two different walls to define the surface boundaries.

### 4.2.2 Comparing Wall Texture

For a given particle, it must be possible to determine the surface of the wall directly in front of it (if such a wall exists). Since each particle consists of an x,y coordinate pair and an angle representing difference from the x-axis, it is possible to extrapolate a linear function that extends directly through the particle, as in Figure 4.5. Once an

extrapolated function is obtained, intersection points are calculated for each of the walls represented by the map. Whichever wall contains an intersection point located closest to the particle is identified and its stored texture returned.

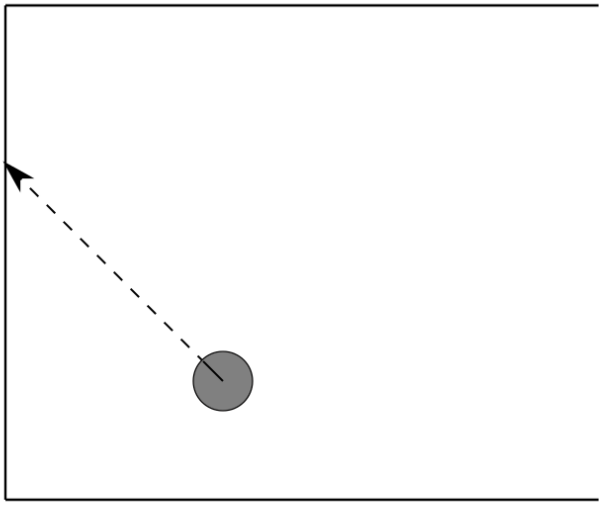


**Figure 4.5: A line extrapolated from a robot pose intersecting a wall**

To actually obtain the extrapolated function, trigonometry is used. Starting with a robot pose, the coordinates are defined as (x, y, θ). Since $a$ is the supplement of $\theta$, it can be calculated as $a = \pi - \theta$. Once $a$ is calculated, it is possible to calculate $y' - y = x\ tan(a)$. With $y' - y$ and $x$, the slope of the hypotenuse, which is the extrapolated line, is known to be $\frac{y' - y}{x}$. The slope and the line's y-intercept ($y'$) are then used to determine the function representing the extrapolated line, which is $y_f = \frac{y' - y}{x} x_f + y'$. This line can then be used to determine where the robot's vision intersects with a wall.

However, the function is not enough yet. Since the robot is only facing one direction, that is the only part of the line that matters and running calculations with

the rest would waste resources. To indicate that any portion of the line extending behind the robot can be ignored, the function is assigned bounds. In this case, the area the function is valid in is $x_{bound} = [0, x]$ and $y_{bound} = [y, \infty]$.

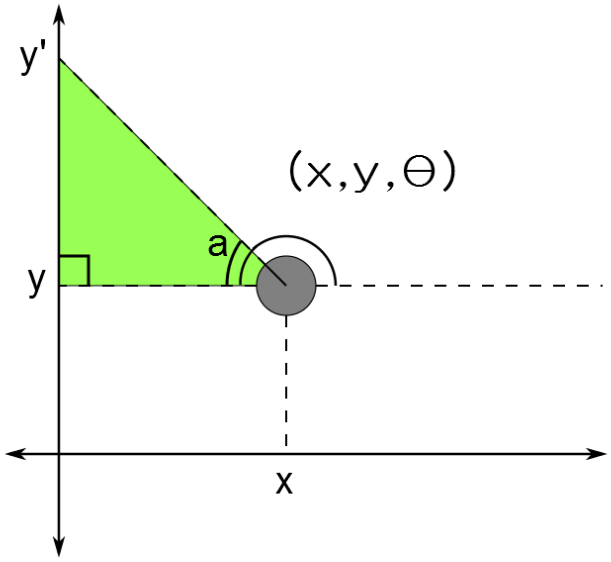



**Figure 4.6: Triangle used to calculate the line extrapolated from a robot pose**

Algorithm 4.2.1 shows how to get the texture of a robot pose from a WallTextureMap.

### 4.2.3 Artificial Neural Network Training

Data gathered on walls from a laser range finder is subject to a number of parameters. Training sets will be compiled of laser readings perpendicular to the wall, some tilted to the left, some tilted to the right, and from different distances away.

The ANN takes an input of twenty consecutive raw laser readings from the laser scan, as pictured in Figure 4.8. Utilizing the data, the ANN does a binary classifi-

**Algorithm 4.2.1** Algorithm used by WallTextureMap to retrieve the texture of provided robot pose

```
1: function getWallTexture(pose)
2:     extrapolate line from pose
3:     for all walls do
4:         if extrapolated line intersects wall then
5:             if intersection is within the bounds of functions then
6:                 if min(distance between intersection and pose) then
7:                     save pose
8:                 end if
9:             end if
10:        end if
11:    end for
12:    if no intersections within function bounds then
13:        return −1
14:    else
15:        return texture of saved pose
16:    end if
17: end function
```

cation to determine if the wall being scanned is part of one selected surface. For the experiment conducted, it was classifying the wall as a row of lockers (Figure 4.7) or not a row of lockers (Figure 4.8).

The ANN is structured with three layers: an input layer, a hidden layer, and an output layer. Both the input and the hidden layers consist of twenty neurons, while the output layer consists of one. After training, the ANN was able to classify test data with $\tilde{1}\%$ mean squared error.

### 4.2.4 Variations

For ease of implementation, the robot only utilizes a subset of the large laser scan. The subset consists of a number of readings from the center of the laser scan. This is used to identify the surface directly in front of the robot if it is facing a surface. Classification of a wall surface using these points is further detailed in Section 4.2.3.

Two variations of Wall Texture MCL were devised, as described below.

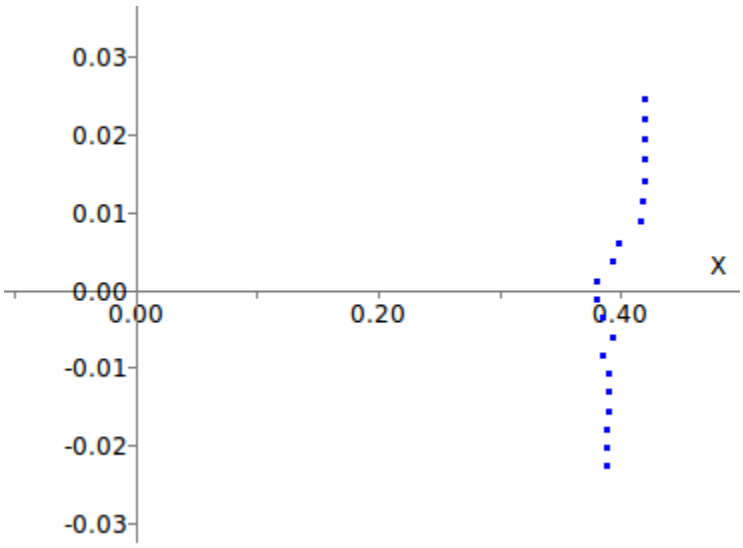



**Figure 4.7: Laser scan of lockers**

#### 4.2.4.1 Front

The first variation of Wall Texture MCL utilizes a single subset of the laser scan. The subset is taken so it examines the wall directly in front of the robot, as seen in Figure 4.9a.

As such, the algorithm requires only one set of data to be fed through the ANN and classified. However, after examining several use cases and running initial tests, it became apparent that the robot was often unable to utilize the laser scans without altering its path to specifically target driving at a wall [FIG?]. This led to the second variation of Wall Texture MCL, which is described next.

#### 4.2.4.2 Sides

The second variation of Wall Texture MCL utilizes three subsets of the laser scan. The first subset is directly in front of the robot, as described in [SEC]. The second and third subsets are taken from the beginning of the laser scan and the end of the

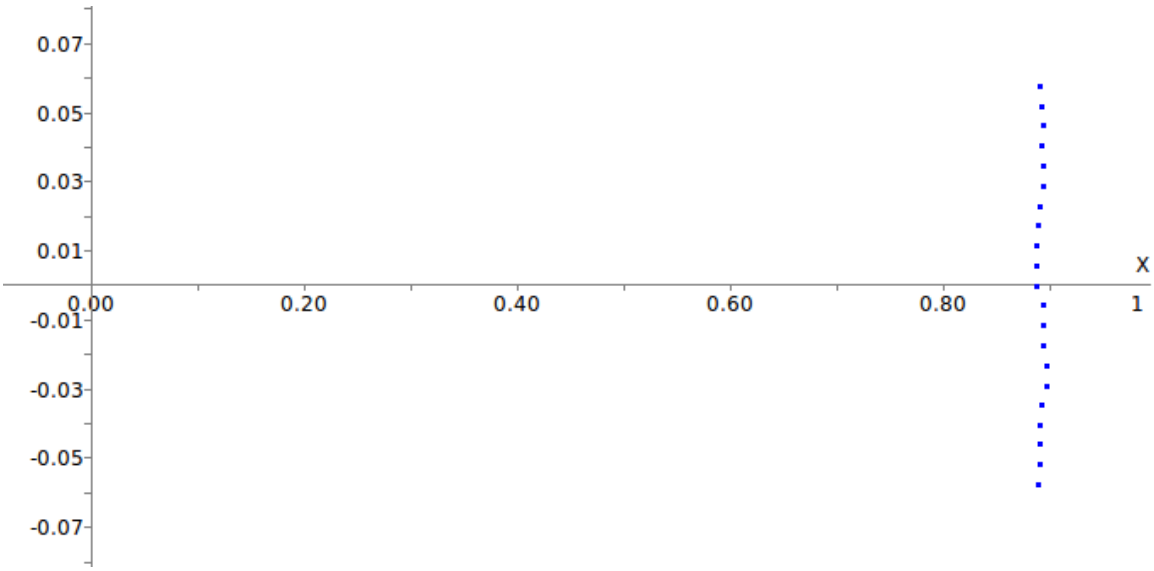
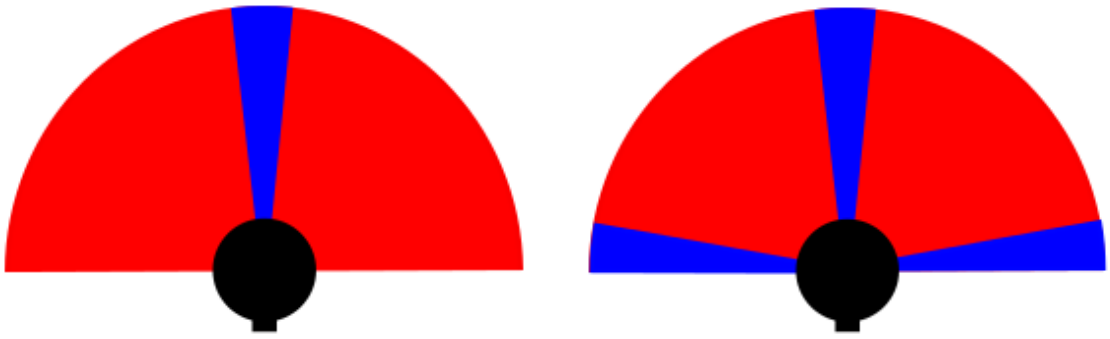



**Figure 4.8: Laser scan of plain wall**

laser scan, both of which are approximately 90 degrees left and right of the front subsection. This is pictured in Figure 4.9b.

Unlike the previous version of Wall Texture MCL described in Section 4.2.4.1, this version requires the classification of three different subsets per iteration of the algorithm. However, it also allows the robot to utilize more of its environment without going out of its way to face a wall. The main use-case this version addresses is driving along a wall. If a robot is driving down a long hallway, the front subset of the laser



(a) Front (b) Sides

**Figure 4.9: Subsections (blue) of the laser scan (red) used by the Wall Texture MCL variations**

scan may not pick up a wall in front of it for several minutes. By using subsets directly left and right of the robot, the algorithm is able to continue factoring in the wall surface texture into its weight calculations, which could help it pick out subtle landmarks, such as a door from surrounding wall.

## 4.3 Incorporating Textures Into MCL

While identifying the floor and wall textures adds more data about the robot's location, one main question is how should that information be used to affect the set of particles used by the MCL algorithm. Two main weighting schemes were explored by this thesis.

Both weighting schemes build on a starting weight, which is the particle weight calculated using a standard MCL observation model. The weight represents how likely the particle is the physical position of the robot based upon the latest laser sensor reading.

### 4.3.1 Double Weights

The first variation modifies the particle weights multiplicatively. For each iteration of the algorithm, the floor and/or wall surface(s) the robot senses are classified. This will be referred to as the sensed texture and is done before any analysis of the particles starts. Next, as each particle is processed, the floor texture, right wall texture, front wall texture, and left wall texture are calculated from the maps as described in [SEC - imu and laser] as needed. If the difference between each sensed texture and each map lookup texture is under a certain threshold, the textures are considered to match and the starting weight of the particle is doubled. If the difference exceeds the threshold, the starting weight of the particle is divided in half.

Based on their starting weights, the probability of the robot being at specific particles can vary drastically with this method. If the starting weight is small, multiplying or diving the weight by two may not change it much. However, large starting weights are adjusted much more. Doubling an already large weight makes a particle significantly more likely to survive to the next round of the algorithm.

Another effect of larger weights is the rolling averages used by AMCL will remain high. This reduces the number of random particles added during an iteration of the algorithm. If the average remains high, AMCL is less likely to consider the robot lost and will focus on manipulating the existing particles rather than drawing new ones to add to the sample set.

### 4.3.2 Add Weights

The second variation modifies particle weights additively. Like the other weighting scheme, if the floor or wall textures pulled from the particles position on the map match the textures of the environment around the robot, the weight will be increased. However, instead of multiplying the weight, a fixed constant value is added to the starting weight. If the textures do not match, the same fixed constant value is subtracted from the starting weight. Weights need to remain positive, so if subtraction results in a negative value, it is automatically set to an infinitesimally small value instead.

Unlike the previous weighting scheme, the impact of matching textures remains constant no matter the starting weight. This allows low weighted particles to quickly increase, since adding a constant provides a greater increase than doubling a low starting weight.

Adding and subtracting constants to and from the weights does not affect the rolling averages used by AMCL as much as doubling the weights. With the addition and subtraction, the averages change less drastically from iteration to iteration.

# CHAPTER 5

# Experimentation

Experiments were conducted to determine the accuracy and performance of the proposed MCL variations. To test accuracy, a robot was driven around a building which contained a variety of floor and wall textures. At selected points, the robot's physical location was compared to the algorithm's estimated position. To test performance, the time the algorithm took to complete single iterations was measured for different particle set sizes.

## 5.1 Methodology

This section describes which portions of the thesis libraries were used for and details modifications made to facilitate testing.

### 5.1.1 Libraries

The implementations used for testing were built on an open source library called the Mobile Robot Programming Toolkit (MRPT) [13]. The library provides interfaces for retrieving data from various robotics sensors as well as several implementations of localization and Simultaneous Localization and Mapping (SLAM). While the library does have an implementation of Monte Carlo Localization that could have been used, the algorithm was re-implemented for the purpose of testing the new variations.

The algorithm was implemented again, but several classes that provided the underlying functionality for the algorithm were utilized. Classes representing 2D particles, standard grid maps, and laser scan readings were used. Utilizing MRPT had two main advantages. First, a grid map representation did not have to be invented. The existing CObservationGridMap2D has already been optimized and expedites map creation by allowing maps to be developed from an image, as seen in [PIC]. Second, a sensor model had already been implemented that was able to compare a laser scan against a position in the grid map and determine the likelihood of the robot currently being there. This sensor model served as the starting weight utilized in the weighting schemes from Section 4.3.

In addition to some classes from MRPT, an external library called Fast Artificial Neural Network was utilized to create the ANNs used for texture classification [12]. The library is capable of creating customized ANNs using user provided parameters and training data. This particular library was chosen because it was originally designed to quickly process images for use on robots.

### 5.1.2 Algorithm Modifications

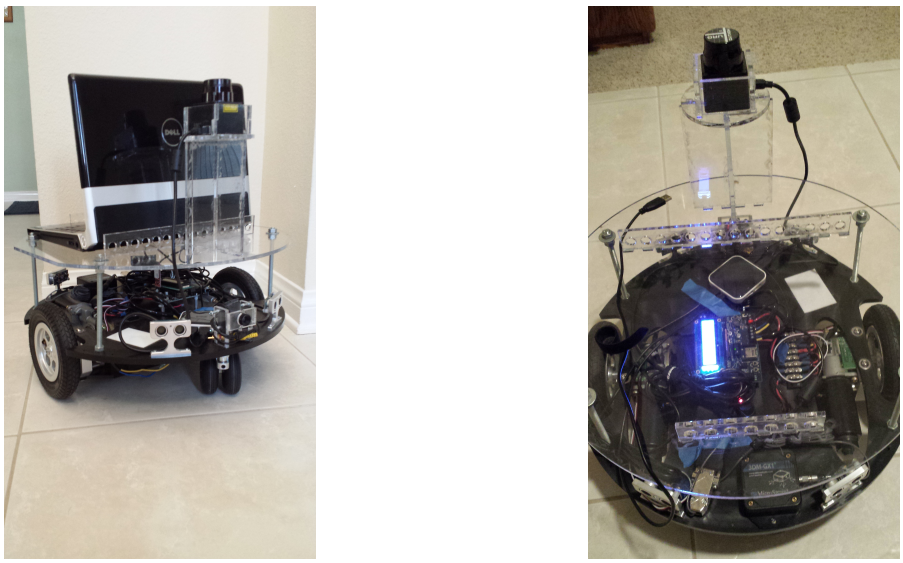
#### 5.1.2.1 Accuracy

One change was made to the MCL algorithm variations used for accuracy testing. Normally, the algorithm would run consistently, including when the robot is sitting still. However, the robot needed to be halted so physical measurements could be taken. Instead of continuing to run while the robot is stopped, the algorithm instead pauses its processing of particles and waits until the robot is in motion again. If the robot has been still for a sufficient period of time, which indicates it has reached a point where measurements are being recorded, the particle set is saved to file so it can be used for later processing.

#### 5.1.2.2 Performance

Normally MCL is a constantly repeating algorithm that has no fixed end. For performance testing, the number of iterations the algorithm would run was specified ahead of time.

## 5.2 Equipment



(a) (b)

**Figure 5.1: Robot used for testing**

The robot used in experiments is a modified Parallax MadeUSA chassis kit, pictured in Figure 5.2. It utilizes a differential drive setup with motors separately controlled by an Aithon board [3]. Motor commands are sent to the Aithon board via Bluetooth through an Android app. The app allows both of the main motors to be controlled separately.

The two main sensors used for localization are a Hokuyo URG-04LX-UG01 laser range finder (Figure 5.1b top) and a MicroStrain 3DM-GX1 IMU (Figure 5.1b bottom). These are connected directly to a laptop with an Intel Core 2 Duo running

Ubuntu 13.04. The Aithon board is also connected to the laptop and periodically sends motor velocity information to the running localization program. While the Aithon board sends data to the laptop, the laptop does not send anything back to the Aithon board. This means the robot is in no way autonomous and any localization is done passively. For a complete data flow diagram, see Figure 5.2.

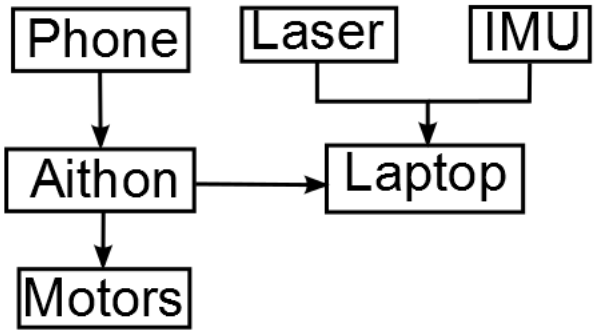



**Figure 5.2: Movement of data through hardware components**

## 5.3 Environment

Experiments were run in the Bonderson building on the Cal Poly campus. Approximately half of the building was utilized. Doors separating the two main areas of the building were closed and tests were conducted in the main hallway area, seen in Figure 5.3. The hallway area itself is approximately 38 meters long by 9 meters wide. There are a number of different surfaces in this portion of the building alone.

The wall surfaces vary between standard paint, concrete, shiny metal (elevator door), glass, and painted metal (lockers) seen in Figure 5.4. The glass actually presented a problem for the robot, as the laser range finder is not able to consistently sense it. To overcome this problem, glass surfaces in the robot's field of view were covered during test runs.

**Figure 5.3: Main hallway**

The flooring of the building consists of linoleum tiles and industrial carpeting, pictured in Figure 5.3. An inch-wide rubber strip separates the two, but will not be counted as a separate texture for experimentation purposes.

Tests in the building were conducted after hours to ensure a static environment. Since MCL assumes it is only seeing objects on the reference map, this ensured the robot did not become confused if a person moved into its field of view.

**Figure 5.4: Wall surfaces**

## 5.4 Data Collection

Two sets of experiments were performed so data on both the accuracy and performance of the algorithm could be obtained. Accuracy tests were run by driving a robot around a building and recording its location. Performance tests were done by recording the amount of time algorithm iterations take to complete.

### 5.4.1 Accuracy

For each implementation, multiple test runs were conducted. The approximate path the robot took each time can be seen in Figure 5.6. At eleven different points along the way, the robot stopped so measurements could be taken. Both the physical location and the estimated location of the robot are recorded at each point. The physical location is determined by measuring the robot's location in the building's frame of reference. The estimated location is determined based upon the set of particles held

(a) Carpet

(b) Tile

**Figure 5.5: Floor surfaces**

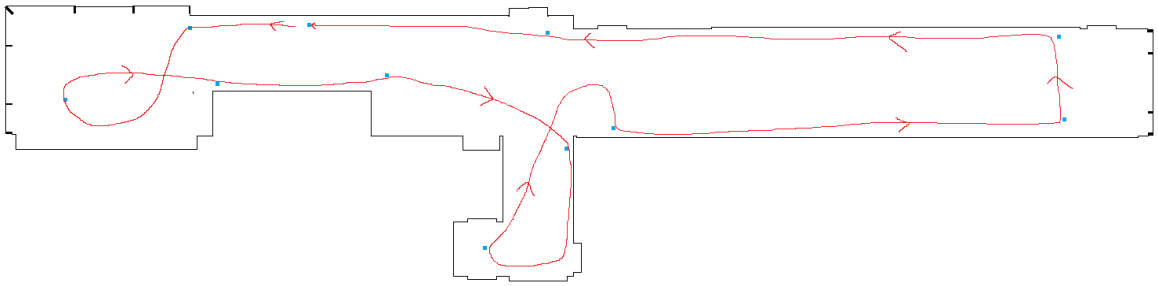


**Figure 5.6: Overview of building with robot test route and measuring points**

by the robot. The particle set is saved at each point so the estimated location can be determined later off-line.

For each set of particles, the average location was calculated for both all particles and the fifty highest weighted particles. The distance between each average and the corresponding physical location for that point is calculated, as seen in Table 6.1. Expanded tables are provided in A.1 and A.2. Selected scatter plots of the particles are also provided.

### 5.4.2 Performance

In addition, data on the performance of the algorithm was gathered. Each implementation was run with 1000, 5000, and 10000 particles, and the length of each iteration was recorded using a software timer. The time per iteration was collected by running each variation for 100 iterations and averaging the total time.

The same maps of the test area were used for this test, however the robot was propped up to drive in place rather than actually move around. In addition, the robot was blocked in on three sides so the worst case of the algorithm would run. This mainly affected the laser and both versions. The front laser versions would be forced to always make one wall comparison while the side laser versions would always make three wall comparisons. In reality, having the robot being blocked in on three sides for extended periods of time is highly unlikely, but it serves to get a worst-case time.

# CHAPTER 6

## Results

Data was collected on eleven different algorithms: the baseline and ten variations of Texture MCL (which collectively refers to any MCL variation utilizing floor and/or wall textures). The Texture MCL variations fall into three general categories based on the sensors used: IMU, laser, and both IMU and laser. The laser category also has two variations which use different subsets of a laser scan. Finally, each variation was tested with two different weighting schemes, one that adds a constant to the particle weights and one that multiplies the weights by a constant. A brief description of each has been provided below.

**standard-mcl**

This is the unmodified MCL algorithm that serves as the baseline for testing. The AMCL variation (Algorithm 2.1.2) is implemented to allow for recovery from incorrect guesses.

**imu-mcl-double**

This is a variation of MCL that uses floor textures and adjusts the particle weights multiplicatively.

**imu-mcl-add**

This is a variation of MCL that uses floor textures and adjusts the particle weights additively.

**laser-mcl-front-double**

This is a variation of MCL that uses wall textures directly in front of the robot and adjusts particle weights multiplicatively.

**laser-mcl-front-add**

This is a variation of MCL that uses wall textures directly in front of the robot and adjusts particle weights additively.

**laser-mcl-sides-double**

This is a variation of MCL that uses wall textures directly in front of the robot and to its left and right. Particle weights are adjusted multiplicatively.

**laser-mcl-sides-add**

This is a variation of MCL that uses wall textures directly in front of the robot and to its left and right. Particle weights are adjusted additively.

**both-mcl-front-double**

This is a variation of MCL that uses both floor textures and wall textures directly in front of the robot. Particle weights are adjusted multiplicatively.

**both-mcl-front-add**

This is a variation of MCL that uses both floor textures and wall textures directly in front of the robot. Particle weights are adjusted additively.

**both-mcl-sides-double**

This is a variation of MCL that uses both floor textures and wall textures to the front, left, and right of the robot. Particle weights are adjusted multiplicatively.

**both-mcl-sides-add**

This is a variation of MCL that uses both floor textures and wall textures to the front, left, and right of the robot. Particle weights are adjusted additively.

## 6.1 Accuracy

Table 6.1 contains the average distances between an algorithm's estimated position and the robot's physical location. Two estimated positions are derived, one from the entire particle set and one from the fifty highest weighted particles for each point. Extended tables are available in Appendix A.

At first glance, the numbers in Table 6.1 are incredibly large. However, each number is the average of 11 points measured 5 different times, for a total of 55 points. If the robot got lost right before a measurement point, that point will have a very large distance between the estimated location and the physical position of the robot. The large distance is an outlier that drags the averages up. An example of this can be seen in the tables of Appendix A. The robot commonly got lost around points 8 and 9 because it had trouble determining how far it had traveled down the symmetrical hallway. As a result, while the rest of the estimated positions may be close to their corresponding physical locations, points 8 and 9 had many differing by at least 10 meters.

Another factor that contributes to larger values is the use of a velocity model for robot motion. There are two options for modeling robot motion: sensors, such as encoders, or

Figure 6.1 shows the numbered points where measurements were recorded inside the building. Each number corresponds to a location (a black box) in the plots of particles (colored dots) below. The plots show where the particles were distributed across the map when the robot was stopped at a point for measurements.

The *standard-mcl* implementation output an average distance between estimated location and physical location of 4.91. As seen in Figure 6.1, the algorithm managed to converge near the correct location for all but points 8 and 9. While there is a small

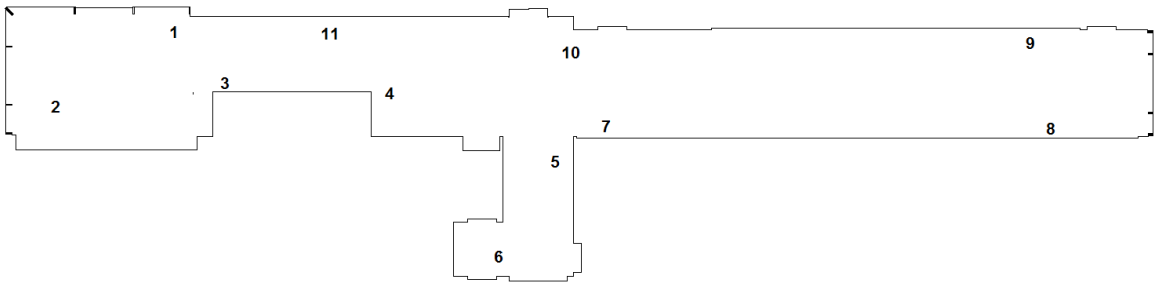


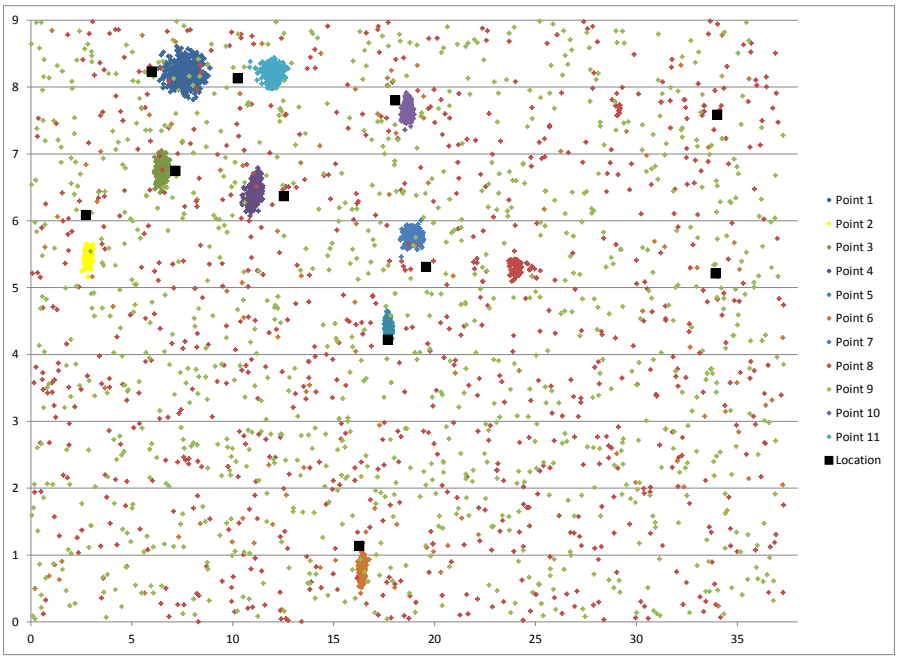

**Figure 6.1: Numbered measuring points in the test area**

**Table 6.1: Average Distances Between Robot's Estimated and Physical Locations for Given Particles (m)**

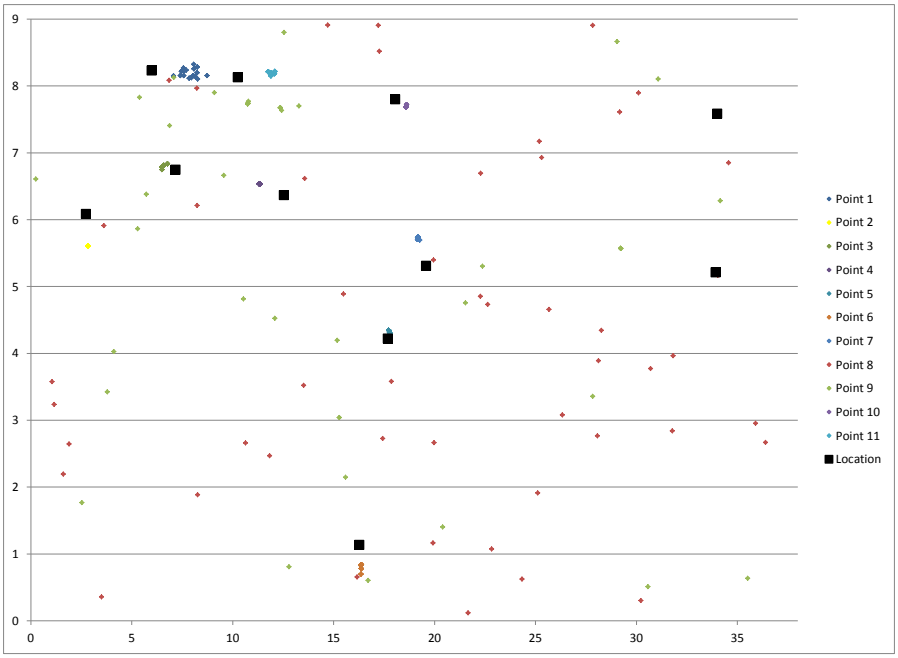
| Implementation | All Particles | Top 50 Particles |
|---|---|---|
| **standard-mcl** | 4.91 | 4.42 |
| **laser-mcl-front-double** | 4.84 | 3.8 |
| **laser-mcl-front-add** | 4.39 | 3.31 |
| **laser-mcl-sides-double** | 4.46 | 4.14 |
| **laser-mcl-sides-add** | 4.8 | 4.96 |
| **both-mcl-front-double** | 4.66 | 4.27 |
| **both-mcl-front-add** | 5.61 | 5.18 |
| **both-mcl-sides-double** | 3.97 | 3.39 |
| **both-mcl-sides-add** | 5.52 | 5.47 |
| **imu-mcl-double** | 4.52 | 4.09 |
| **imu-mcl-add** | 4.99 | 4.27 |

cluster of red particles when all particles are plotted to indicate where the robot thought it was for point 8, a look at the top 50 plotted reveals that few to none of the particles in that cluster were rated highly. To provide better results than the baseline, the texture implementations can do two things: develop actual clusters for points 8 and 9 or shorten the distance between the clusters and physical location for the rest of the points.

The *both-mcl-front-double* implementation, seen in Figure 6.1 managed to barely improve on the baseline. It yielded an improvement of 5.1% when all particles were analyzed and only 3.2% when the top 50 were used.

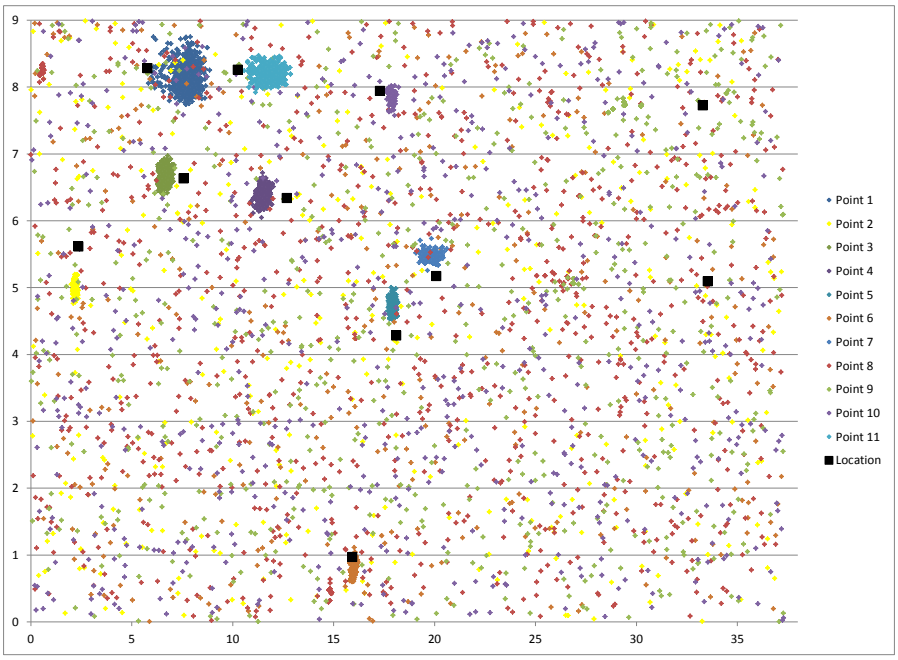



(a) All particles

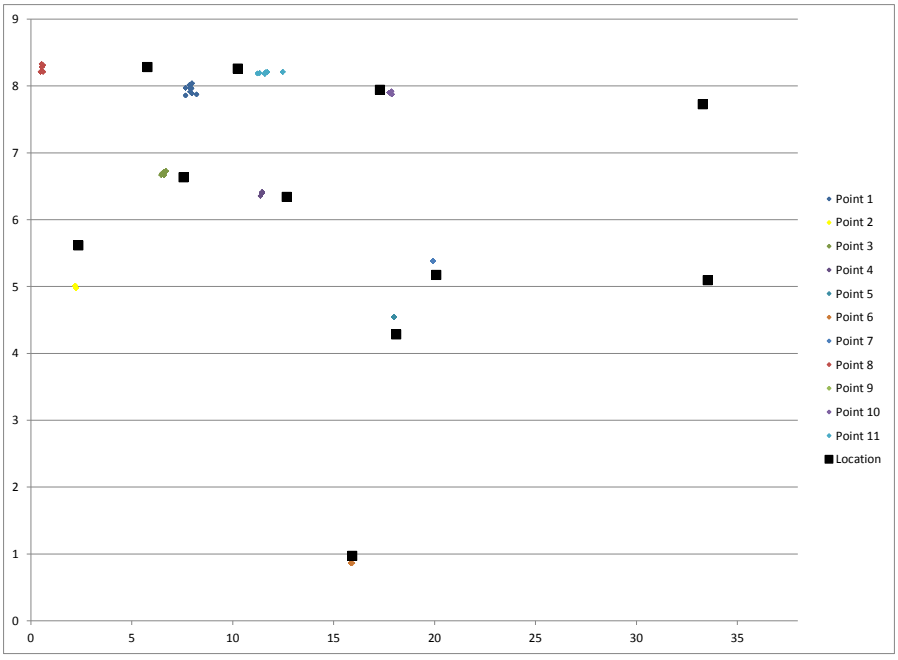



(b) Top 50 particles

**Figure 6.2: Particle distribution from *standard-mcl* run 1**

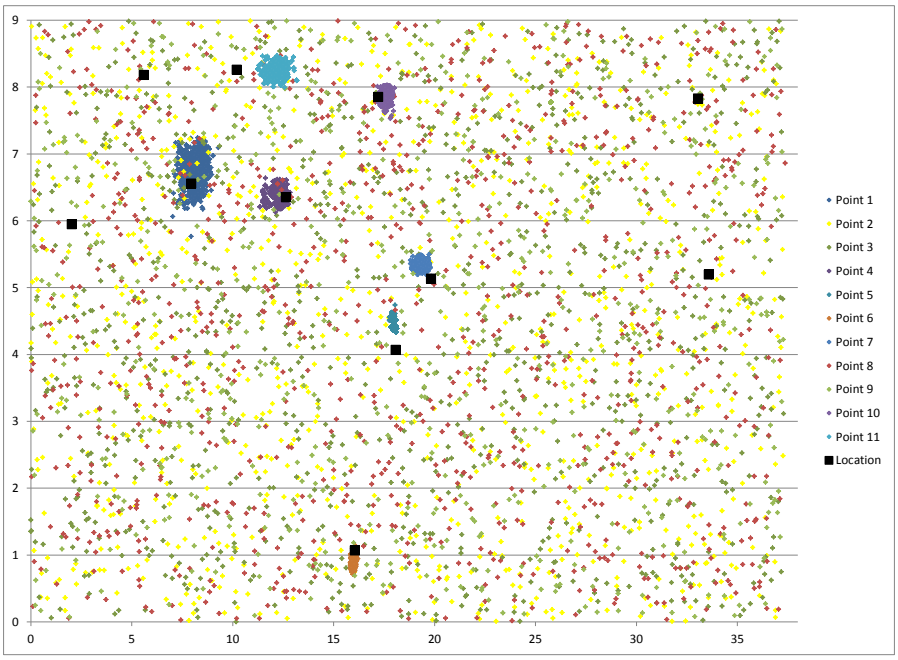



(a) All particles

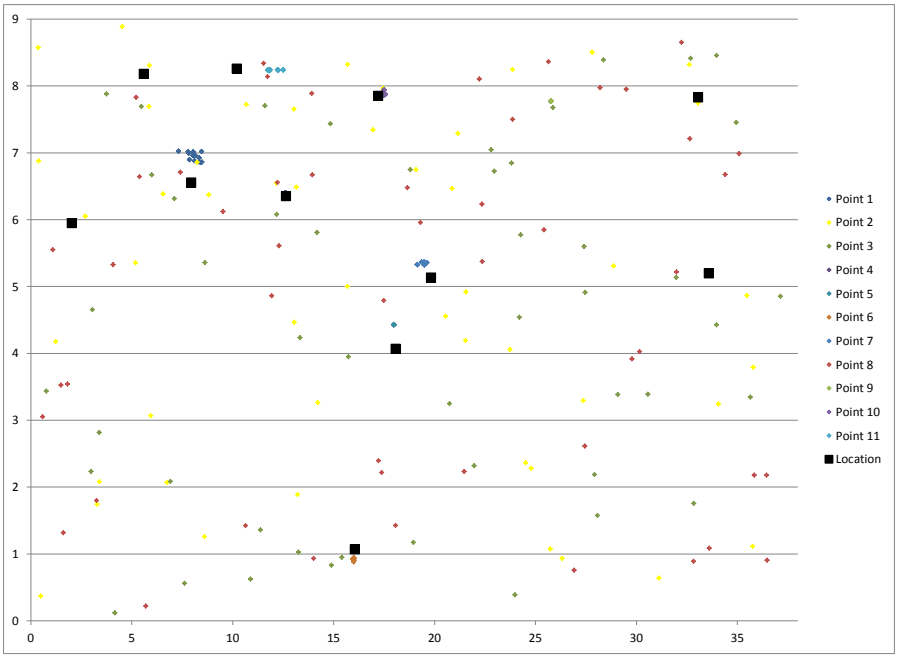



(b) Top 50 particles

**Figure 6.3: Particle distribution from *both-mcl-front-double* run 3**




(a) All particles




(b) Top 50 particles

**Figure 6.4: Particle distribution from *both-mcl-front-add* run 1**

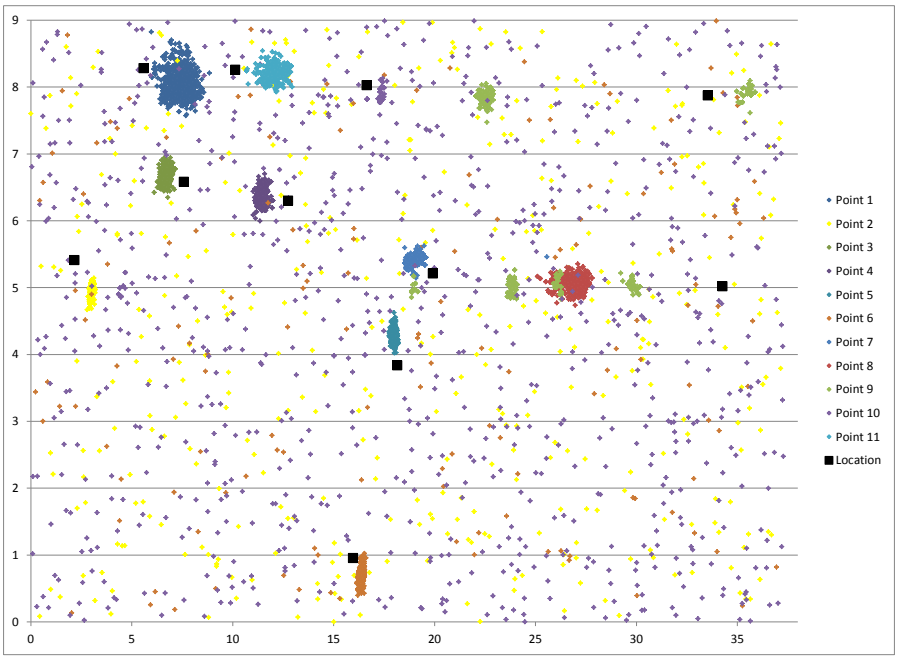
The *both-mcl-front-add* implementation performed as one of the worst out of all of the implementations. It resulted in a 14.1% decrease of accuracy when looking at all particles and a 17.2% decrease when only the top 50 were examined. Figure 6.1 shows that that the algorithm had difficulty converging on locations. From the particles scattered over the all particles plot, it can be determined that the algorithm was injecting many random samples into the particle set. This would be done frequently if the average weights that control the randomness remain constantly low.

The implementation *both-mcl-sides-double* managed to achieve the largest increase in accuracy over the baseline for all particles and the second largest for the top fifty, at 19.1% and 23.2%. Looking at Figure 6.1, two things are worth noting. First, the random particles scattered across the chart are less dense than the other implementations, meaning more of the particles managed to converge on their location. Second, there were actually solid estimates for points 8 and 9. While not as precise as they could be, this implementation managed to determine that the robot was in the long portion of the hallway and tried to converge nearby.

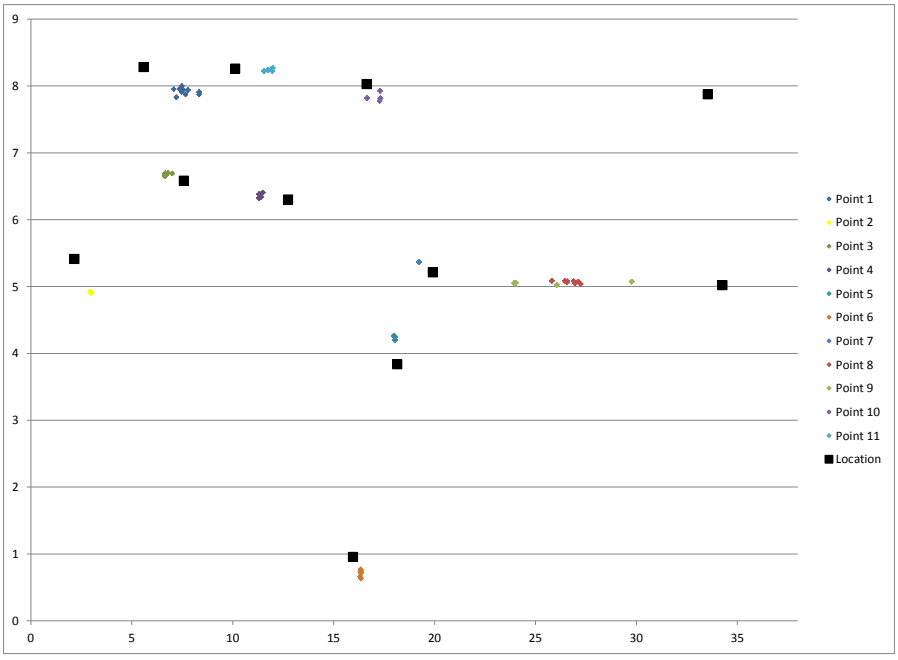
The other implementation that performed poorly in comparison to the baseline was *both-mcl-sides-add*. With 12.2% and 23.7% decreases in accuracy for all particles and the top 50 respectively, it did slightly better than *both-mcl-front-add* for all particles but did worse for the top 50. Looking at Figure 6.1, it is possible to see small clusters forming for some of the more random points. However, being small the clusters have a low weight, and sometimes cannot adjust the rolling averages enough to force a stop to creating random particles.

The implementation *imu-mcl-double* managed to increase the location accuracy by 8% and 7.4% compared to the baseline.

The implementation *imu-mcl-add* managed to obtain worse accuracy than the baseline for all particles but better accuracy for the top 50. One possible reason for

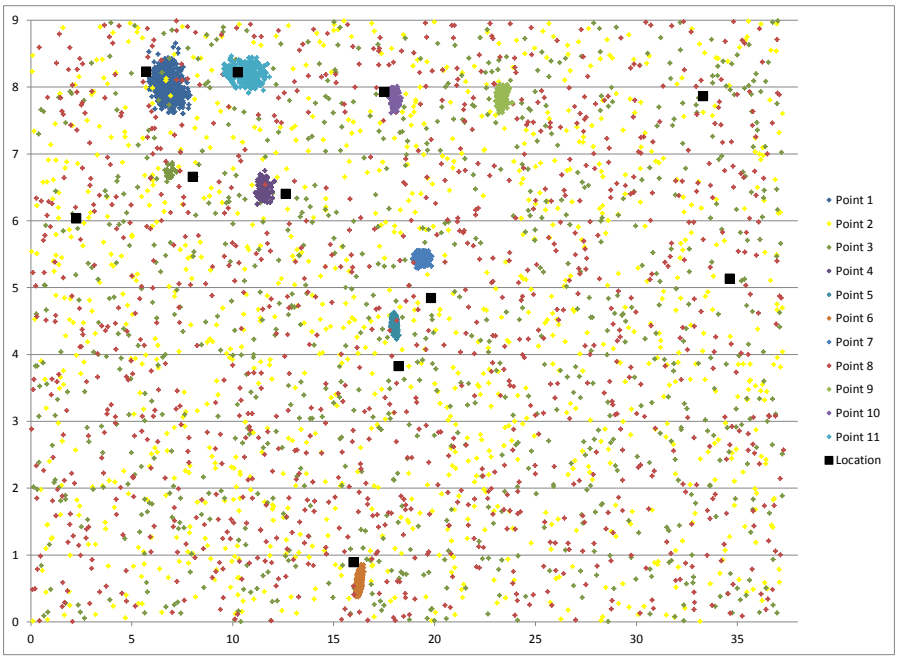



(a) All particles

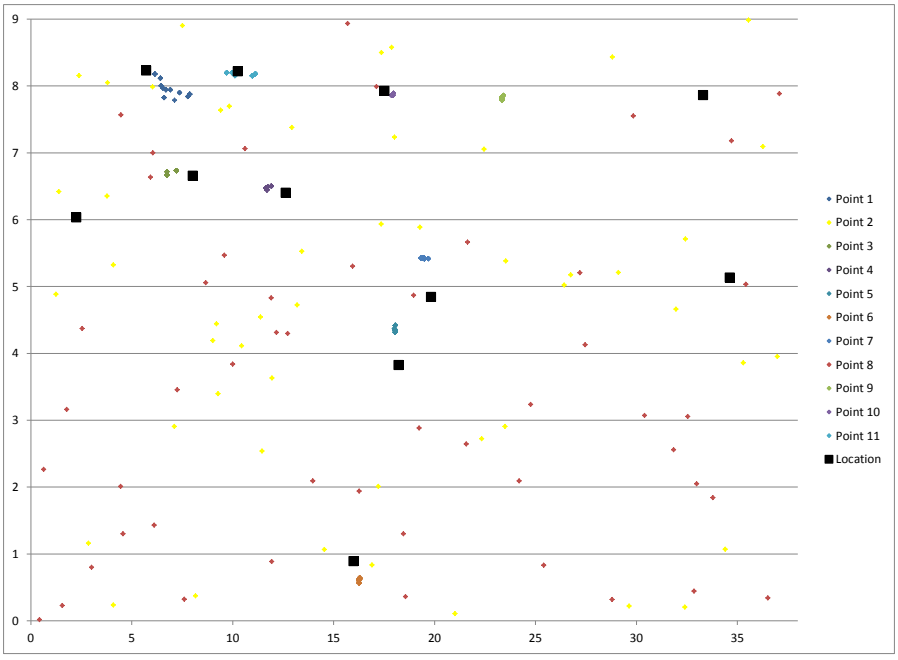



(b) Top 50 particles

**Figure 6.5: Particle distribution from *both-mcl-sides-double* run 2**

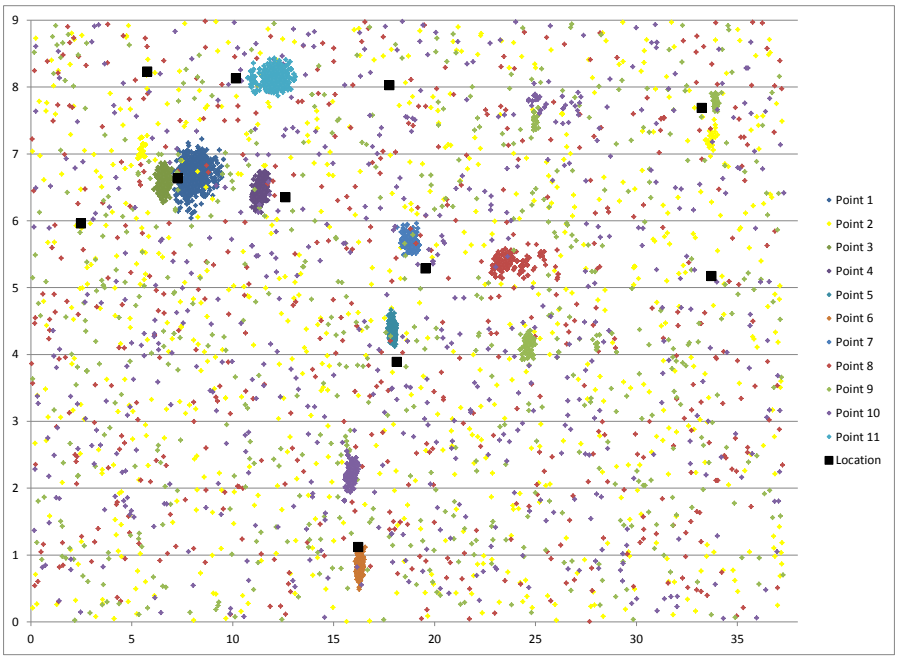



(a) All particles

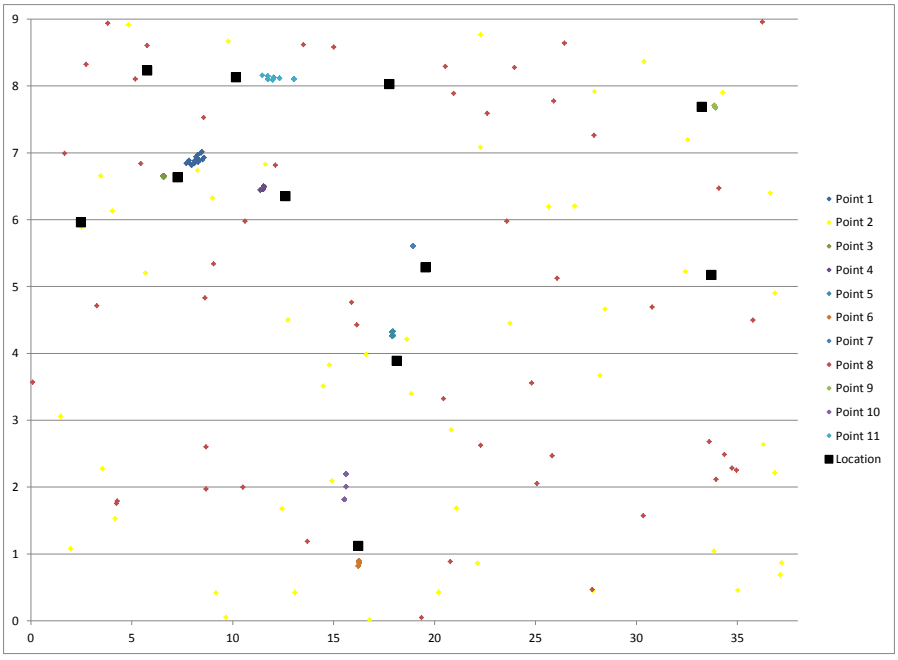



(b) Top 50 particles

**Figure 6.6: Particle distribution from *both-mcl-sides-add* run 1**

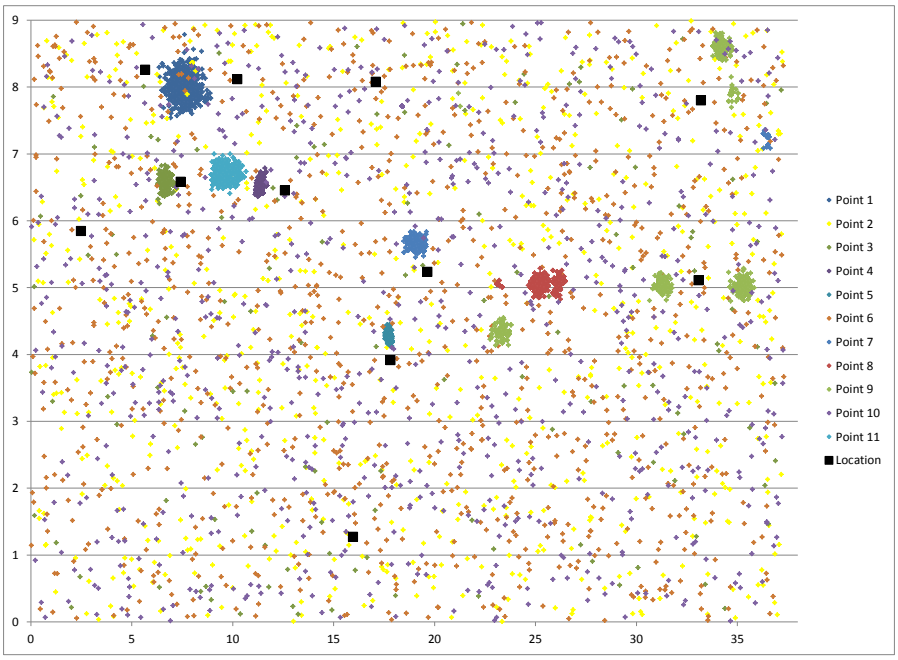



(a) All particles

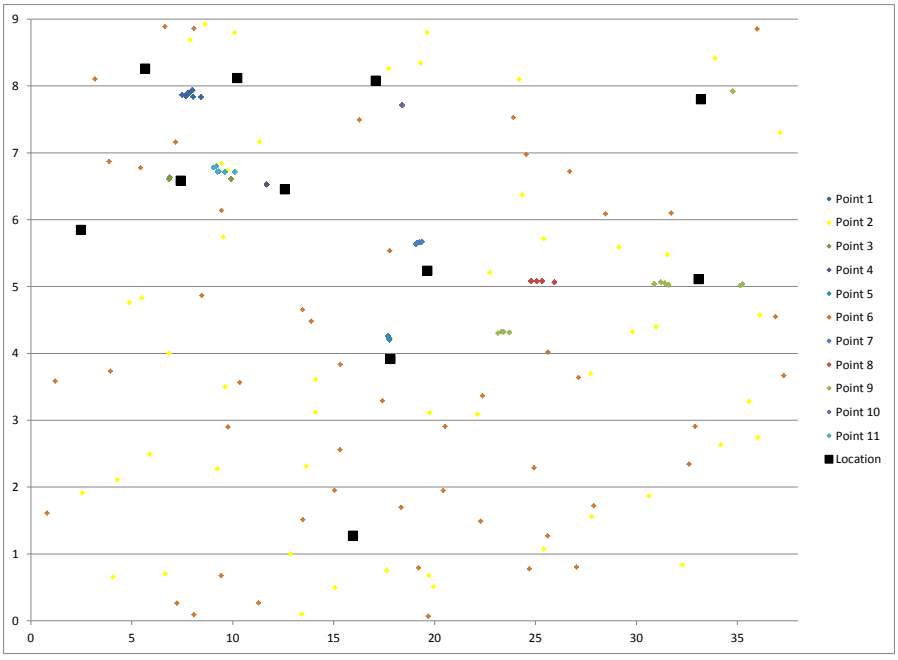



(b) Top 50 particles

**Figure 6.7: Particle distribution from *imu-mcl-double* run 2**




(a) All particles




(b) Top 50 particles

**Figure 6.8: Particle distribution from *imu-mcl-add* run 3**

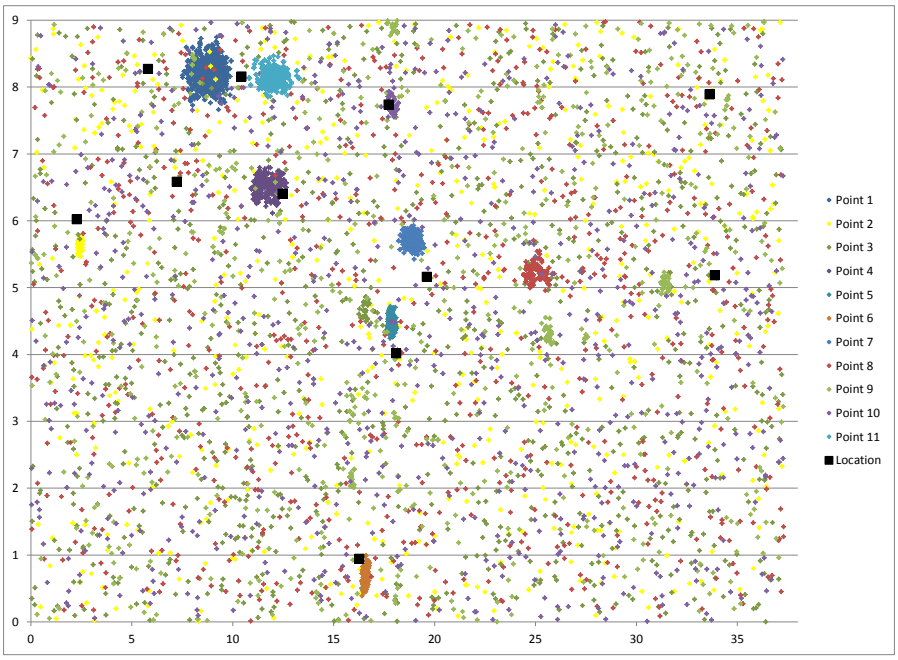
this occurrence is the robot could have been lost and started recovering right before measurements were taken. This would result in scattered points, with small clusters that would be weighted higher and make it into the top 50. It is also worth nothing that the differences in accuracy were small: -1.5% and 3.4%.

The *laser-mcl-front-double* implementation posted increases in accuracy of 1.4% and 13.9% over the baseline. One interesting thing about Figure 6.1's top 50 chart is how there are almost no scattered particles. Having little to no outliers in an individual cluster would help make the estimated location more accurate.

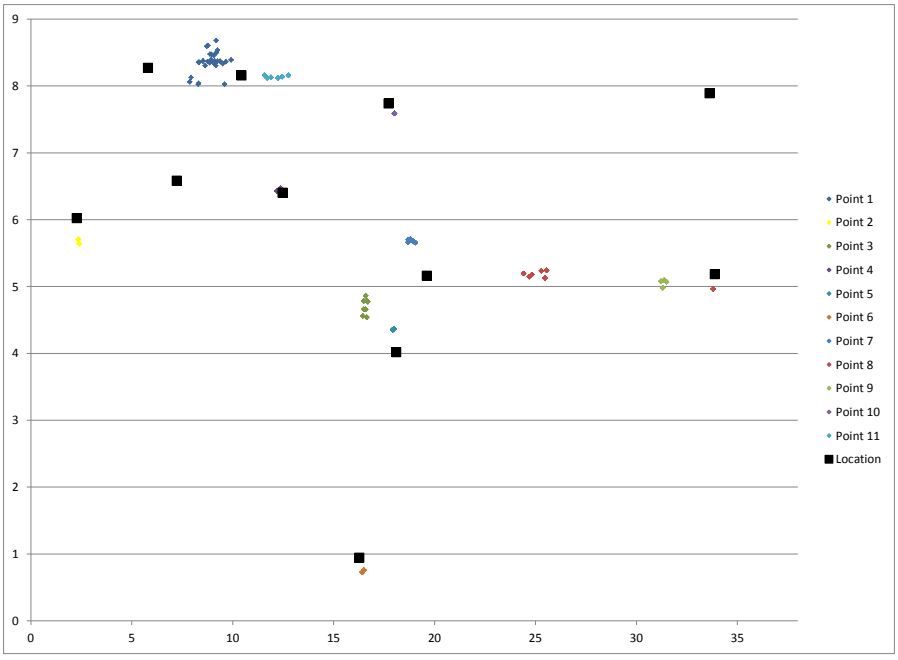
The *laser-mcl-front-add* implementation managed to achieve the largest increase in accuracy for the top 50 particles at 25.1%. It also improved on the baseline for all particles by 10.6%. Glancing at Figure 6.1, that seems surprising. One thing to take into account here is the measurements being used. For each point, averages of particles are collected. This means large scattered sets will have an average somewhere in the middle of the map, in this case rather close to two of the points the scattered particles should belong to.

The *laser-mcl-sides-double* implementation improved on the baseline's averages by 9.3% and 6.2%. While Figure 6.1 still shows scattering for the all particles, it is not nearly as dense as some of the other implementations. The top 50 particles chart has almost every particle in a cluster, but the clusters are farther away from the actual robot location than some of the other implementations.

The *laser-mcl-sides-add* implementation managed to improve accuracy by 2.4% for all particles but decrease in accuracy by 12.2% for the top 50. This implies random particles affected the averages. As mentioned earlier, scattered points will average to the center of the chart, which can improve the average if in reality the particles are off to one side.

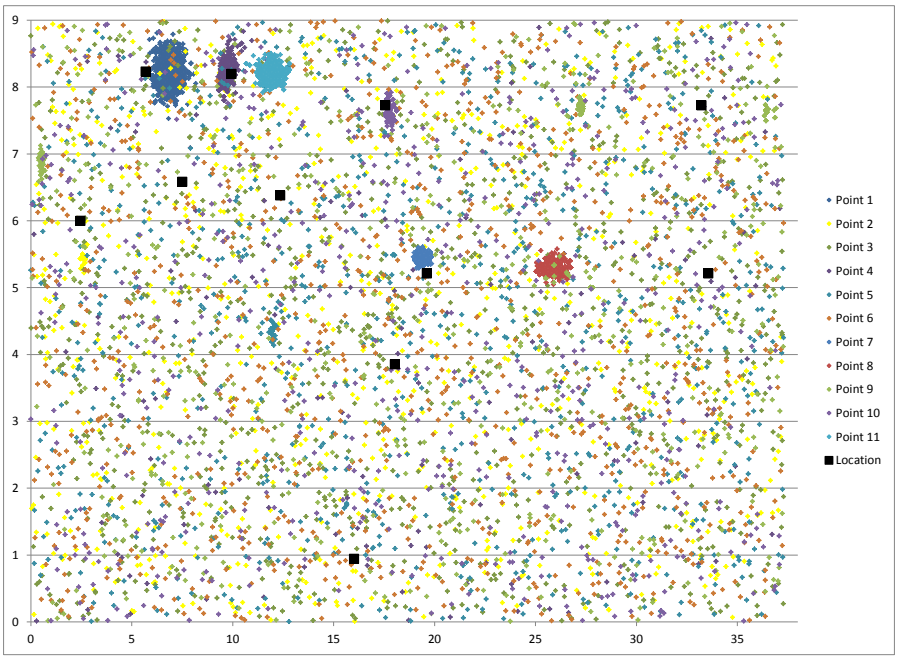



(a) All particles

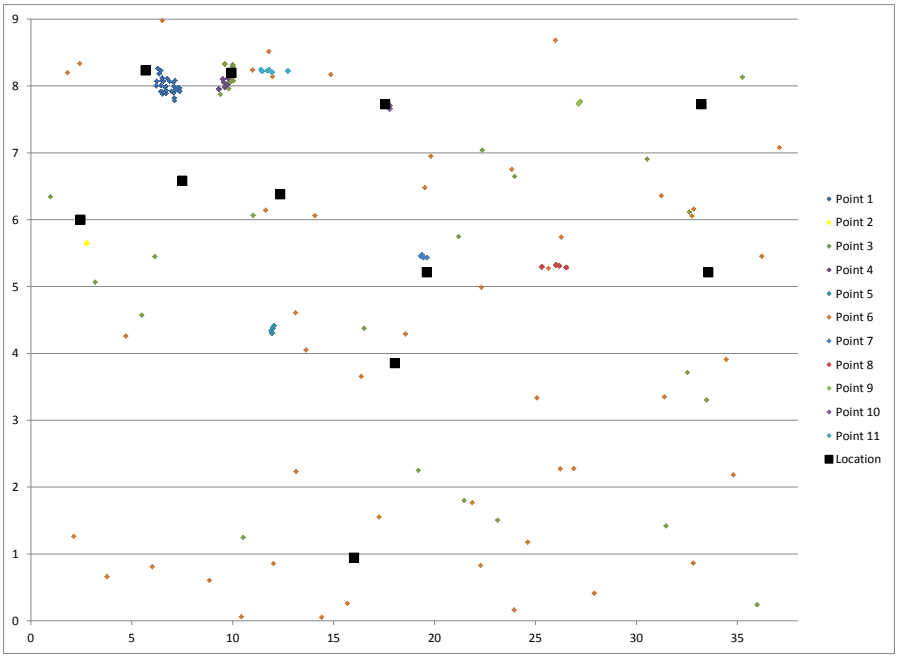



(b) Top 50 particles

**Figure 6.9: Particle distribution from *laser-mcl-front-double* run 2**

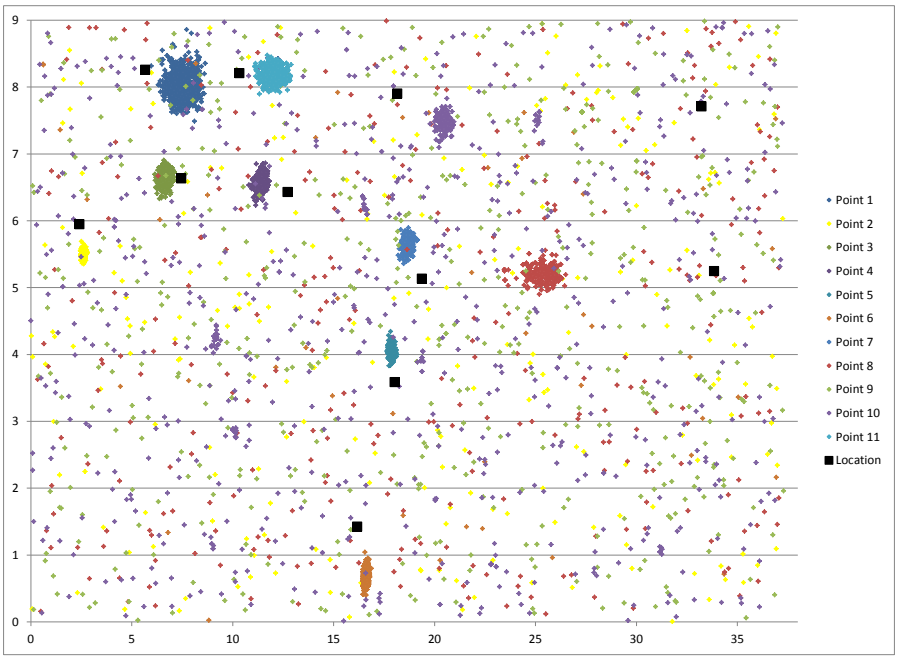



(a) All particles

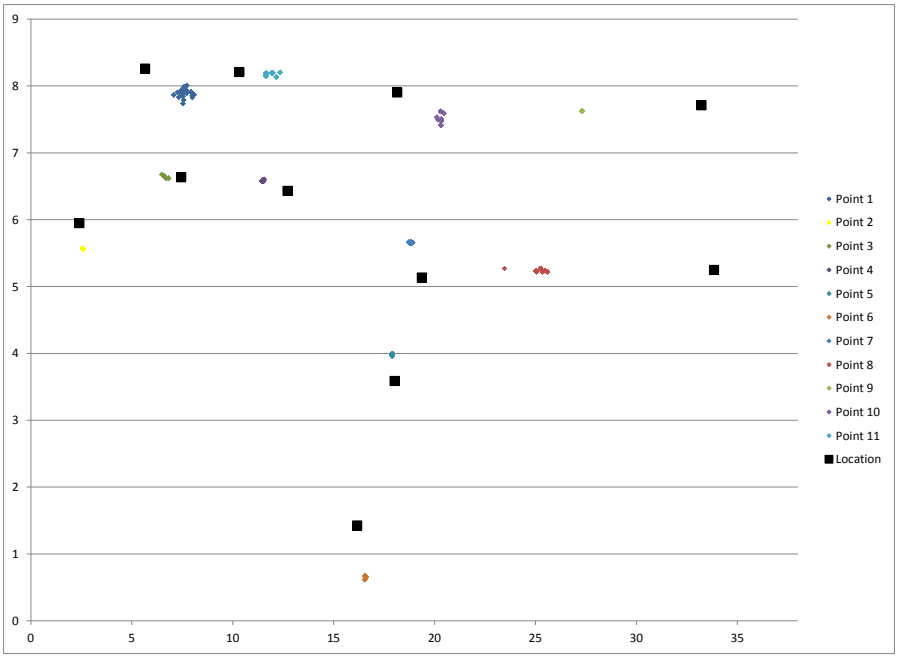



(b) Top 50 particles

**Figure 6.10: Particle distribution from *laser-mcl-front-add* run 3**

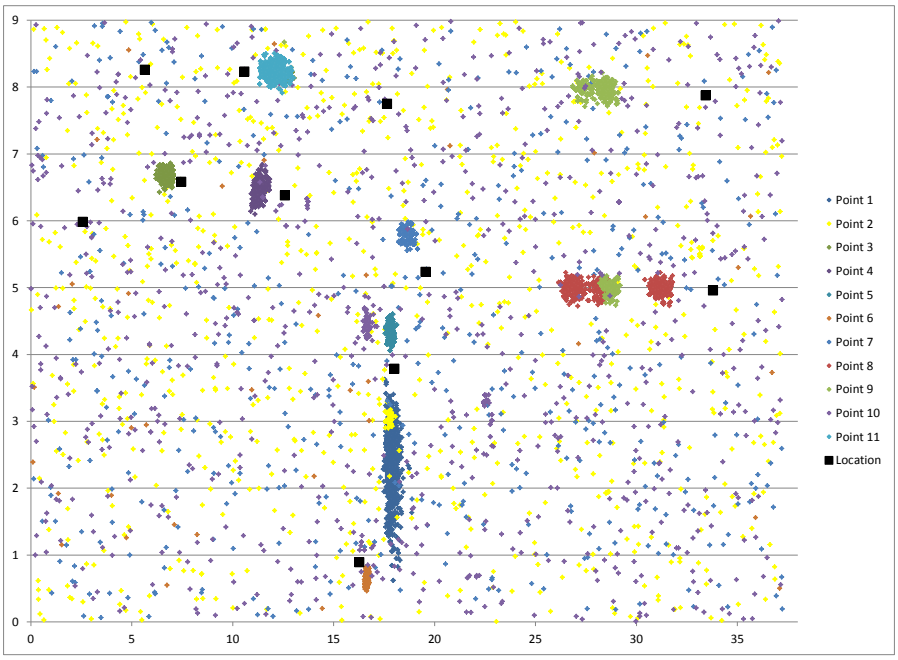



(a) All particles

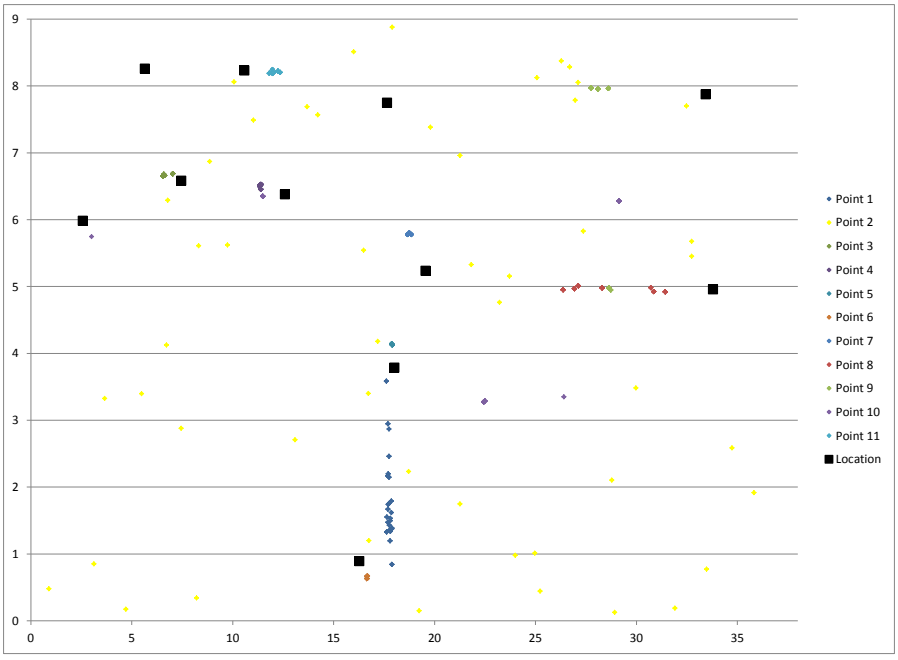



(b) Top 50 particles

**Figure 6.11: Particle distribution from *laser-mcl-sides-double* run 2**




(a) All particles




(b) Top 50 particles

**Figure 6.12: Particle distribution from *laser-mcl-sides-add* run 1**

## 6.2 Performance

MCL is designed to run in real-time. However, as the number of particles increases, the number of calculations that must be done increases as well. Table 6.2 lists the average amount of time it takes for a single iteration of the algorithms to complete. Figure 6.13 plots the number of particles against the length of one iteration of the algorithm.

**Table 6.2: Average Algorithm Iteration Length (s)**

| Implementation | Number of Particles | | |
|---|---|---|---|
| | **1000** | **5000** | **10000** |
| **standard-mcl** | 0.25 | 0.25 | 0.38 |
| **imu-mcl-add** | 0.26 | 1.24 | 2.39 |
| **imu-mcl-double** | 0.25 | 1.28 | 2.54 |
| **laser-mcl-front-add** | 0.25 | 1.24 | 2.42 |
| **laser-mcl-front-double** | 0.26 | 1.33 | 2.6 |
| **laser-mcl-sides-add** | 0.27 | 1.32 | 2.7 |
| **laser-mcl-sides-double** | 0.28 | 1.41 | 2.7 |
| **both-mcl-front-add** | 0.25 | 1.25 | 2.37 |
| **both-mcl-front-double** | 0.26 | 1.25 | 2.76 |
| **both-mcl-sides-add** | 0.25 | 1.25 | 2.26 |
| **both-mcl-sides-double** | 0.26 | 1.27 | 2.32 |

Data was gathered for particle sets of size 1000, 5000, and 10000. At 1000, the algorithms all complete an iteration in approximately a quarter of a second. However, as the particle sets increase in size, the texture MCL implementations take longer to complete while standard MCL remains under half a second per iteration.

The main cause of the slow-down is determining a particle's texture, since the particle must be compared to every wall and much of the floor sections as well. However, all of the performance tests were run as the worst case scenario. This means the sensors were accessed and the extra calculations done every iteration of the algorithm. While the best case scenario would remain the same for the floor texture versions because the IMU is accessed every turn regardless, the wall texture

versions using the laser would in reality have a better performance since wall textures are only utilized when the robot closes in on a wall. While they would still perform worse than *standard-mcl*, only utilizing the laser every few iterations would improve the performance.

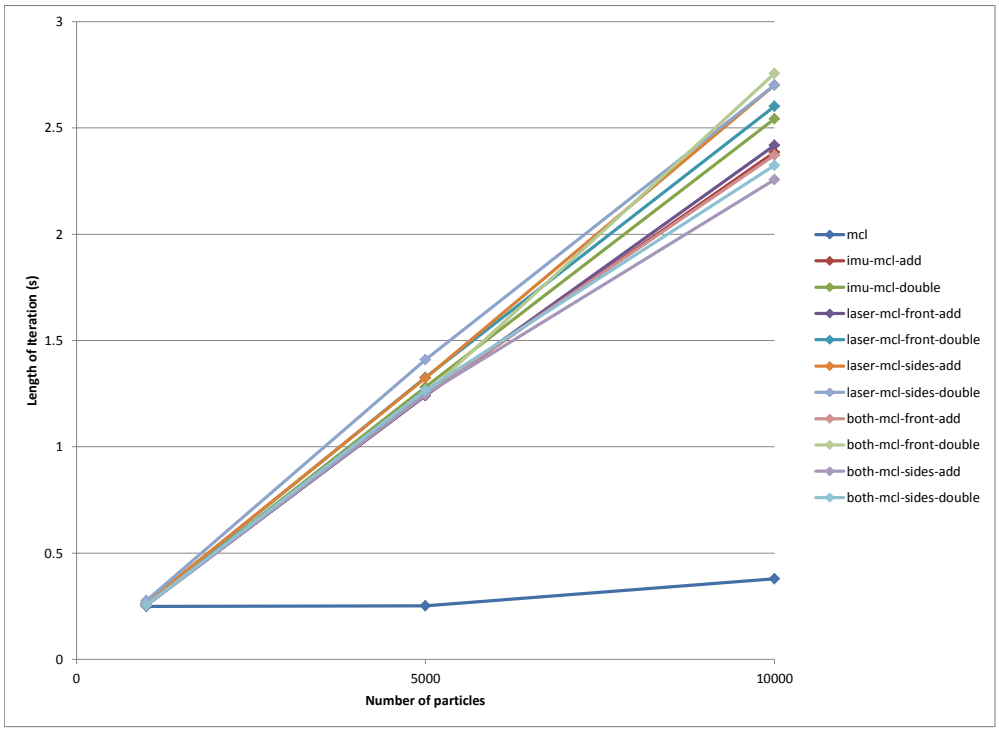



**Figure 6.13: Average time an algorithm iteration takes to complete for different particle set sizes**

# CHAPTER 7

## Future Work

There are a number of directions further research on Texture MCL could take. One option is refining the variations presented here. The texture maps and lookup algorithms could be optimized further, perhaps being implemented in parallel to increase performance. Different ANN structures could be tried to further improve accuracy. Various constants, such as $\alpha_{slow}$ and $\alpha_{fast}$ from Algorithm 2.1.2 or the weight modifications discussed in Section 4.3, could be tested to determine the optimal values.

Texture MCL could also work well with other sensors. For instance, photoresistors could be used to capture the color of surfaces. A camera could be used to extract patterns from the environment that may not appear to a laser or IMU. Since MCL has also been proven to work with sonar sensors, it would be interesting to create a version that does not require an expensive laser range finder to run.

After seeing the accuracy achieved with the IMU, it would be interesting to determine if MCL could be done solely using data derived from the floor. Cracks in the floor or bumps that separate different floor surfaces could potentially be used as landmarks. This would take a lot of the trouble out of operating in a dynamic environment since it would be virtually impossible for something to get between the robot and the ground.

# CHAPTER 8

## Conclusion

Wall Texture MCL provided more accurate results than Floor Texture MCL, with improvements measured against a standard MCL implementation run in the Bonderson building on Cal Poly's campus. The best Floor Texture implementation improved accuracy by 8% and 7.4%, when estimated positions were calculated with all and the top 50 particles respectively. The best Wall Texture implementation improved accuracy by 10.6% and 25.1% with all particles and the top 50 particles.

Combining the Floor and Wall Texture data methods yielded mixed results. On one hand, *both-mcl-sides-double* produced the most consistently best accuracy increase, with improvements of 19.1% for all particles and 23.2% for the top 50 particles. On the other, *both-mcl-front-add* and *both-mcl-sides-add* were the only two variations that had decreases in accuracy for both all particles and the top 50 particles. Since *both-mcl-front-double* produced positive increases in accuracy for both sets of particle sizes, it appears the weighting scheme is responsible. When the two surface texture types are combined and weights are modified with a multiplicative scheme, the algorithm becomes more accurate, while using an additive scheme reduces the accuracy.

Any accuracy gained using Texture MCL comes with the cost of performance as particle sets grow larger. Unlike standard MCL, Texture MCL implementations increase the amount of time they need to complete an iteration almost linearly. However, recall that performance tests simulated the worst case scenario, which will rarely

occur for some implementations. For Wall Texture MCL implementations, extra calculations for the weights only occur when the robot is near a wall, and rarely for all three sides (right, front, and left) in one iteration.

That being said, the implementations have many variables that could be adjusted for further optimization. The algorithm could also be parallelized to process multiple particles at once. Since this is a first attempt at adding textures to MCL, there is no doubt it could be improved or adapted. Hopefully, this thesis has provided a foundation for future work.

Bibliography

[1] D. Anguelov, B. Taskarf, V. Chatalbashev, D. Koller, D. Gupta, G. Heitz, and A Ng. Discriminative learning of markov random fields for segmentation of 3d scan data. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 169–176 vol. 2, June 2005.

[2] Wolfram Burgard, Dieter Fox, Daniel Hennig, and Timo Schmidt. Estimating the absolute position of a mobile robot using position probability grids. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 896–901. AAAI Press, 1996.

[3] Connor Citron, Brian Gomberg, and John Seng. The aithon board: A case study in commercialization of a student project. *J. Comput. Sci. Coll.*, 29(4):202–209, April 2014.

[4] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA99)*, May 1999.

[5] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 343–349, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[6] Yu Fu, S. Tully, G. Kantor, and H. Choset. Monte carlo localization using 3d texture maps. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 482–487, Sept 2011.

[7] D. Gutmann, J.-S. Fox. An Experimental Comparison of Localization Methods Continued. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[8] R. Jitpakdee and T. Maneewarn. Neural networks terrain classification using inertial measurement unit for an autonomous vehicle. In *SICE Annual Conference, 2008*, pages 554–558, Aug 2008.

[9] Hadi Kheyruri and Daniel Frey. Comparison of people detection techniques from 2d laser range data, 2010.

[10] Yoon J. Park C.-W. Kim, D.Y. In *2013 Proceedings of the 30th ISARC, Montral, Canada*, pages 685–690, 2013.

[11] Ambroise Krebs, Cdric Pradalier, and Roland Siegwart. Comparison of boosting based terrain classification using proprioceptive and exteroceptive data. In Oussama Khatib, Vijay Kumar, and George J. Pappas, editors, *Experimental Robotics*, volume 54 of *Springer Tracts in Advanced Robotics*, pages 93–102. Springer Berlin Heidelberg, 2009.

[12] Contributors listed at `http://leenissen.dk/fann/wp/authors/`. FANN Fast Artificial Neural Network Library, Accessed 2014.

[13] Contributors listed at `http://www.mrpt.org/Authors/`. MRPT (Mobile Robot Programming Toolkit), 2014.

[14] Liang Lu, C. Ordonez, Jr. Collins, E.G., and E.M. DuPont. Terrain surface classification for autonomous ground vehicles using a 2d laser stripe-based structured

light sensor. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 2174–2181, Oct 2009.

[15] Oscar Martnez Mozos, Hitoshi Mizutani, Hojung Jung, Ryo Kurazume, and Tsutomu Hasegawa. Categorization of indoor places by combining local binary pattern histograms of range and reflectance data from laser range finders. *Advanced Robotics*, pages 1455–1464, 2013.

[16] J. Perez, F. Caballero, and L. Merino. Integration of monte carlo localization and place recognition for reliable long-term robot localization. In *Autonomous Robot Systems and Competitions (ICARSC), 2014 IEEE International Conference on*, pages 85–91, May 2014.

[17] Roland Siegwart and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Bradford Company, Scituate, MA, USA, 2004.

[18] Jens–steffen Gutmann, Thilo Weigel, and Bernhard Nebel. A fast, accurate, and robust method for self-localization in polygonal environments using laser-range-finders. *Advanced Robotics Journal*, 14:2001, 2000.

[19] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[20] D. Tick, T. Rahman, C. Busso, and N. Gans. Indoor robotic terrain classification via angular velocity based hierarchical classifier selection. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3594–3600, May 2012.

# APPENDIX A

Charts

**Table A.1: Average Distances Between Robot's Estimated and Physical Locations for All Particles**

| Implementation | Run | Point | | | | | | | | | | | Avg Dist/Run | Avg Dist/Imp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | |
| mcl | 1 | 1.66 | 0.55 | 0.65 | 1.42 | 0.16 | 0.28 | 0.74 | 14 | 16.28 | 0.62 | 1.71 | 3.46 | 4.91 |
| | 2 | 2.72 | 12.19 | 0.65 | 1.32 | 0.57 | 0.42 | 12.61 | 0.39 | 11.62 | 2.62 | 0.09 | 4.11 | |
| | 3 | 4.69 | 24.72 | 0.73 | 1.62 | 0.4 | 0.65 | 0.98 | 17.13 | 15.46 | 2.14 | 1.72 | 6.38 | |
| | 4 | 2.53 | 8.86 | 0.92 | 1.31 | 0.47 | 0.52 | 0.79 | 18.5 | 15.57 | 2.11 | 1.77 | 4.85 | |
| | 5 | 11.84 | 16.62 | 0.61 | 1.32 | 0.36 | 0.18 | 0.45 | 12.55 | 15.21 | 2.45 | 1.78 | 5.76 | |
| laser-mcl-front-double | 1 | 3.04 | 16.32 | 0.65 | 1.26 | 0.48 | 0.44 | 1.82 | 9.8 | 14.57 | 5.26 | 0.15 | 4.89 | 4.84 |
| | 2 | 2.97 | 9.29 | 11.45 | 0.65 | 0.49 | 0.35 | 0.88 | 13.46 | 14.24 | 2.84 | 1.58 | 5.29 | |
| | 3 | 1.54 | 7.06 | 11.28 | 2.16 | 0.47 | 0.6 | 0.74 | 13.52 | 12.75 | 5.4 | 1.51 | 5.18 | |
| | 4 | 2 | 7.41 | 0.77 | 1.39 | 0.49 | 0.54 | 12.17 | 15.05 | 10.96 | 0.36 | 1.96 | 4.83 | |
| | 5 | 3.63 | 8.72 | 0.78 | 1.62 | 0.53 | 0.27 | 0.55 | 11.22 | 12.11 | 3.26 | 1.65 | 4.03 | |
| laser-mcl-front-add | 1 | 1.44 | 11.64 | 11.44 | 0.09 | 0.47 | 3.99 | 0.8 | 6.95 | 9.57 | 2.88 | 1.97 | 4.66 | 4.39 |
| | 2 | 4.44 | 15.24 | 9.48 | 0.43 | 0.48 | 2.45 | 0.39 | 6.05 | 2.46 | 2.83 | 2.9 | 4.29 | |
| | 3 | 1.16 | 12.34 | 11.35 | 2.07 | 0.82 | 4.52 | 0.33 | 7.64 | 11.3 | 2.23 | 1.97 | 5.06 | |
| | 4 | 1.53 | 5.55 | 0.69 | 1.28 | 0.45 | 1.81 | 0.8 | 12.37 | 8.47 | 0.7 | 1.69 | 3.21 | |
| | 5 | 3.4 | 16.12 | 0.71 | 1.35 | 0.45 | 0.25 | 0.96 | 14.87 | 9.19 | 2.83 | 1.92 | 4.73 | |
| laser-mcl-sides-double | 1 | 1.25 | 0.6 | 0.71 | 1.17 | 0.59 | 2.43 | 1 | 13.01 | 11.48 | 3.28 | 1.68 | 3.38 | 4.46 |
| | 2 | 1.83 | 4.23 | 0.8 | 1.32 | 0.47 | 0.82 | 0.87 | 10.42 | 9.96 | 2.66 | 1.67 | 3.19 | |
| | 3 | 3.25 | 15.37 | 0.83 | 1.31 | 0.44 | 0.39 | 0.93 | 13.61 | 15.07 | 3.4 | 1.3 | 5.08 | |
| | 4 | 4.44 | 15.54 | 9.57 | 5.06 | 0.57 | 0.75 | 0.59 | 13.18 | 9.8 | 2.98 | 1.87 | 5.85 | |
| | 5 | 1.84 | 4.66 | 9.18 | 6.51 | 2.21 | 4.03 | 0.38 | 14.77 | 6.97 | 0.52 | 1.5 | 4.78 | |
| laser-mcl-sides-add | 1 | 13.56 | 15.98 | 0.47 | 1.26 | 0.55 | 0.35 | 1.07 | 4.86 | 5.2 | 3.39 | 1.51 | 4.38 | 4.8 |
| | 2 | 5.04 | 18.46 | 8.18 | 0.52 | 0.61 | 3.92 | 1.08 | 0.6 | 2.1 | 3.45 | 1.56 | 4.14 | |
| | 3 | 3.33 | 17.16 | 9.2 | 4.79 | 0.58 | 0.92 | 5.31 | 3.27 | 6.99 | 4.37 | 1.55 | 5.22 | |
| | 4 | 5.28 | 15.65 | 0.74 | 1.18 | 0.47 | 1.78 | 0.34 | 10.81 | 10.18 | 3.15 | 1.49 | 4.64 | |
| | 5 | 3.38 | 15.7 | 11.77 | 0.44 | 0.49 | 0.47 | 10.32 | 11.77 | 1.85 | 3.42 | 1.95 | 5.6 | |

| Implementation | Run | Point | | | | | | | | | | | Avg Dist/Run | Avg Dist/Imp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | |
| **both-mcl-front-double** | **1** | 3.32 | 14.12 | 6.51 | 2.33 | 0.54 | 0.28 | 0.33 | 10.33 | 11.58 | 3.75 | 1.24 | 4.94 | 4.66 |
| | **2** | 13.36 | 9.87 | 0.8 | 1.25 | 0.31 | 0.39 | 0.74 | 15.69 | 7.25 | 3.52 | 1.62 | 4.98 | |
| | **3** | 1.97 | 5.62 | 0.93 | 1.19 | 0.45 | 1.4 | 0.4 | 15.95 | 10.4 | 2.45 | 1.54 | 3.84 | |
| | **4** | 4.46 | 16.97 | 0.86 | 1.21 | 0.4 | 0.29 | 0.57 | 12.67 | 5.59 | 3.58 | 1.4 | 4.36 | |
| | **5** | 4.53 | 14.91 | 0.76 | 1.33 | 0.48 | 0.3 | 0.73 | 11.89 | 17.65 | 3.04 | 1.38 | 5.18 | |
| **both-mcl-front-add** | **1** | 2.92 | 16.36 | 10.72 | 0.31 | 0.41 | 0.17 | 0.55 | 14.84 | 12.44 | 0.37 | 1.94 | 5.55 | 5.61 |
| | **2** | 1.51 | 2.74 | 8.62 | 1.82 | 0.97 | 4.26 | 0.34 | 14.76 | 13.39 | 3.25 | 1.98 | 4.88 | |
| | **3** | 11.84 | 16.81 | 11.01 | 6.62 | 0.48 | 4.2 | 0.23 | 13.8 | 9.52 | 2.9 | 1.3 | 7.16 | |
| | **4** | 3.66 | 16.7 | 9.4 | 3.26 | 1.92 | 4.3 | 4.1 | 14.92 | 1.52 | 3.66 | 0.41 | 5.8 | |
| | **5** | 2.94 | 16.45 | 9.81 | 2.92 | 0.5 | 2.92 | 0.94 | 6.76 | 2.82 | 3.69 | 1.47 | 4.66 | |
| **both-mcl-sides-double** | **1** | 1.93 | 9.19 | 0.98 | 0.99 | 0.59 | 2.74 | 0.95 | 11.58 | 6.07 | 4.82 | 0.24 | 3.64 | 3.97 |
| | **2** | 1.88 | 7.25 | 0.9 | 1.3 | 0.47 | 0.85 | 0.8 | 7.37 | 8.15 | 3.53 | 1.89 | 3.13 | |
| | **3** | 2.28 | 9.93 | 1.22 | 0.87 | 0.39 | 0.36 | 0.65 | 13.61 | 10.41 | 4.19 | 0.94 | 4.08 | |
| | **4** | 1.97 | 7.52 | 3.41 | 6.74 | 0.45 | 0.4 | 0.79 | 15.68 | 6.13 | 3.83 | 1.67 | 4.42 | |
| | **5** | 3.78 | 14.15 | 7.69 | 0.92 | 0.28 | 0.09 | 0.68 | 13.28 | 7.68 | 0.61 | 1.48 | 4.6 | |
| **both-mcl-sides-add** | **1** | 1.09 | 16.03 | 9.34 | 1.09 | 0.62 | 0.4 | 0.72 | 16.44 | 9.95 | 0.47 | 0.46 | 5.15 | 5.52 |
| | **2** | 12.04 | 16.69 | 0.21 | 1.3 | 0.48 | 0.42 | 0.33 | 15.62 | 3.67 | 3.13 | 1.75 | 5.06 | |
| | **3** | 13.91 | 16.33 | 1.64 | 0.23 | 0.49 | 3.8 | 0.91 | 11.9 | 8.64 | 1.65 | 1.75 | 5.57 | |
| | **4** | 3.11 | 16.93 | 10.57 | 3.03 | 3.08 | 4.63 | 17.09 | 3 | 0.53 | 3.82 | 1.69 | 6.13 | |
| | **5** | 2.87 | 16.3 | 9.87 | 12.09 | 0.95 | 4.34 | 0.19 | 0.34 | 11.96 | 1.86 | 1.62 | 5.67 | |
| **imu-mcl-double** | **1** | 2.06 | 0.48 | 1.11 | 1.26 | 0.5 | 0.69 | 0.9 | 12.92 | 12.47 | 1.14 | 1.85 | 3.22 | 4.52 |
| | **2** | 2.81 | 13.66 | 0.69 | 1.23 | 0.51 | 0.27 | 0.85 | 12.91 | 10.71 | 4.34 | 1.88 | 4.53 | |
| | **3** | 3.29 | 9.55 | 10.02 | 1.4 | 0.57 | 0.79 | 0.87 | 12.39 | 7.28 | 2.1 | 1.88 | 4.56 | |
| | **4** | 5.73 | 18.97 | 10.79 | 0.51 | 0.41 | 0.36 | 0.61 | 11.38 | 9.71 | 3.48 | 1.63 | 5.78 | |
| | **5** | 2.15 | 5.33 | 0.66 | 1.26 | 1.14 | 4.15 | 1.05 | 9.6 | 7.81 | 15.65 | 0.77 | 4.51 | |
| **imu-mcl-add** | **1** | 1.9 | 19.96 | 11.26 | 1.44 | 1.1 | 4.28 | 0.77 | 8.87 | 12.25 | 2.85 | 1.67 | 6.03 | 4.99 |
| | **2** | 4.69 | 14.79 | 8.41 | 0.45 | 0.44 | 4.22 | 0.82 | 0.77 | 3.67 | 3.1 | 1.78 | 3.92 | |
| | **3** | 2 | 13.13 | 0.93 | 1.17 | 0.38 | 4.26 | 0.65 | 7.71 | 2.02 | 3.19 | 1.51 | 3.36 | |
| | **4** | 2.13 | 18.33 | 7.94 | 1.13 | 0.42 | 4.54 | 1.02 | 13.42 | 3.67 | 5.03 | 1.63 | 5.39 | |
| | **5** | 1.74 | 16.79 | 11.01 | 0.51 | 0.41 | 2.8 | 7.45 | 18.23 | 7.49 | 0.46 | 1.75 | 6.24 | |

**Table A.2: Average Distances Between Robot's Estimated and Physical Locations for the Top 50 Particles**

| Implementation | Run | Point | | | | | | | | | | | Avg Dist/Run | Avg Dist/Imp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | | |
| **standard-mcl** | **1** | 1.87 | 0.48 | 0.59 | 1.23 | 0.13 | 0.34 | 0.58 | 14.39 | 18.27 | 0.55 | 1.66 | 3.65 | 4.42 |
| | **2** | 2.62 | 0.52 | 0.54 | 1.1 | 0.51 | 0.31 | 12.97 | 0.26 | 0.31 | 4.63 | 0.04 | 2.17 | |
| | **3** | 4.72 | 16.35 | 0.66 | 1.18 | 0.3 | 0.54 | 0.92 | 30.92 | 10.24 | 0.25 | 1.48 | 6.14 | |
| | **4** | 2.63 | 1.27 | 0.93 | 1.29 | 0.42 | 0.57 | 0.71 | 33.86 | 13.91 | 1.2 | 1.79 | 5.33 | |
| | **5** | 11.95 | 18.68 | 0.79 | 1.26 | 0.29 | 0.17 | 0.38 | 6.9 | 10.4 | 0.31 | 1.81 | 4.81 | |
| **laser-mcl-front-double** | **1** | 2.8 | 17.08 | 0.62 | 1.15 | 0.41 | 0.37 | 5.33 | 4.94 | 9.89 | 13.88 | 0.33 | 5.16 | 3.8 |
| | **2** | 3.14 | 0.33 | 9.51 | 0.15 | 0.38 | 0.26 | 0.92 | 7.84 | 3.64 | 0.33 | 1.69 | 2.56 | |
| | **3** | 1.42 | 0.56 | 7.37 | 6.89 | 0.33 | 0.6 | 0.73 | 15.35 | 8.53 | 13.68 | 1.5 | 5.18 | |
| | **4** | 2.07 | 0.47 | 0.76 | 1.22 | 0.38 | 0.39 | 9.29 | 12.15 | 6.49 | 0.24 | 1.85 | 3.21 | |
| | **5** | 3.35 | 0.52 | 0.68 | 1.46 | 0.44 | 0.27 | 0.43 | 7.56 | 9.7 | 5.84 | 1.71 | 2.91 | |
| **laser-mcl-front-add** | **1** | 1.34 | 0.5 | 11.73 | 0.23 | 0.42 | 0.29 | 0.64 | 7 | 10.34 | 0.31 | 1.92 | 3.15 | 3.31 |
| | **2** | 4.51 | 9.79 | 2.27 | 0.1 | 0.47 | 0.38 | 0.16 | 6.52 | 2.41 | 7.07 | 2.82 | 3.32 | |
| | **3** | 1.09 | 0.46 | 6.79 | 3.24 | 6.1 | 4.25 | 0.32 | 7.56 | 6.06 | 0.19 | 1.92 | 3.45 | |
| | **4** | 1.79 | 0.45 | 0.63 | 1.18 | 0.42 | 0.14 | 0.74 | 9.58 | 5.08 | 0.58 | 1.7 | 2.03 | |
| | **5** | 3.48 | 17.52 | 0.67 | 1.3 | 0.42 | 0.21 | 0.93 | 12.65 | 10.91 | 0.46 | 1.92 | 4.59 | |
| **laser-mcl-sides-double** | **1** | 1.28 | 0.56 | 0.65 | 1.08 | 0.52 | 0.89 | 0.84 | 7.16 | 11.27 | 9.03 | 1.53 | 3.17 | 4.14 |
| | **2** | 1.99 | 0.41 | 0.78 | 1.21 | 0.41 | 0.86 | 0.76 | 8.62 | 5.92 | 2.18 | 1.57 | 2.25 | |
| | **3** | 3.3 | 15.33 | 0.86 | 1.1 | 0.38 | 0.4 | 0.83 | 12.26 | 22.8 | 2.11 | 1.09 | 5.5 | |
| | **4** | 4.17 | 14.32 | 9.5 | 5.04 | 0.54 | 0.67 | 0.44 | 7.97 | 10.16 | 1.29 | 1.85 | 5.09 | |
| | **5** | 1.83 | 0.37 | 9.1 | 13.84 | 0.45 | 4.23 | 0.32 | 12.45 | 7.03 | 0.41 | 1.72 | 4.71 | |
| **laser-mcl-sides-add** | **1** | 13.73 | 16.52 | 0.74 | 1.23 | 0.37 | 0.45 | 0.97 | 5.19 | 5.19 | 4.66 | 1.47 | 4.59 | 4.96 |
| | **2** | 5.01 | 19.93 | 2.28 | 0.27 | 0.52 | 3.64 | 6.25 | 0.39 | 0.32 | 8.09 | 1.67 | 4.4 | |
| | **3** | 3.31 | 15.32 | 9.1 | 5.09 | 0.52 | 0.68 | 6.41 | 3.4 | 6.33 | 10.02 | 1.34 | 5.59 | |
| | **4** | 5.5 | 15.51 | 0.63 | 1.15 | 0.4 | 0.63 | 0.34 | 18.07 | 14.4 | 3.64 | 1.27 | 5.59 | |
| | **5** | 2.94 | 16.32 | 12.38 | 0.14 | 0.43 | 0.46 | 10.25 | 2.72 | 3 | 0.39 | 1.8 | 4.62 | |

| Implementation | Run | Point | | | | | | | | | | | Avg Dist/Run | Avg Dist/Imp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | |
| **both-mcl-front-double** | **1** | 3.05 | 1.05 | 3.74 | 5.5 | 0.47 | 0.22 | 0.29 | 16.53 | 12.95 | 7.75 | 1.39 | 4.81 | 4.27 |
| | **2** | 13.95 | 1.09 | 0.7 | 0.95 | 0.16 | 0.38 | 0.72 | 17.63 | 3.48 | 2.73 | 1.72 | 3.96 | |
| | **3** | 2.13 | 0.64 | 1.01 | 1.24 | 0.28 | 0.12 | 0.28 | 32.83 | 0.07 | 0.52 | 1.44 | 3.69 | |
| | **4** | 5.12 | 17.54 | 0.79 | 1.14 | 0.32 | 0.27 | 0.53 | 15.43 | 1.86 | 7.22 | 1.33 | 4.69 | |
| | **5** | 3.94 | 14.14 | 0.73 | 1.11 | 0.38 | 0.27 | 0.71 | 15.43 | 7.91 | 0.52 | 1.35 | 4.23 | |
| **both-mcl-front-add** | **1** | 2.75 | 14.43 | 10.63 | 0.06 | 0.38 | 0.17 | 0.46 | 14.84 | 7.31 | 0.3 | 1.85 | 4.83 | 5.18 |
| | **2** | 1.45 | 0.49 | 4.03 | 3.04 | 1.31 | 4.78 | 0.19 | 16.81 | 17.76 | 13.16 | 1.87 | 5.9 | |
| | **3** | 12.07 | 14.47 | 12.55 | 4.13 | 3.05 | 4.27 | 0.24 | 12.49 | 7.6 | 4.15 | 1.44 | 6.95 | |
| | **4** | 3.05 | 15.55 | 3.87 | 3.4 | 1.69 | 3.46 | 4.14 | 11.23 | 1.97 | 3.29 | 0.36 | 4.73 | |
| | **5** | 3.15 | 16.68 | 2.7 | 2.87 | 0.39 | 0.2 | 0.99 | 6.78 | 2.7 | 0.37 | 1.48 | 3.48 | |
| **both-mcl-sides-double** | **1** | 2.03 | 15.84 | 1.05 | 0.75 | 0.5 | 0.22 | 1.04 | 8.46 | 0.48 | 14.94 | 0.63 | 4.18 | 3.39 |
| | **2** | 2.08 | 0.93 | 0.87 | 1.37 | 0.43 | 0.45 | 0.7 | 7.56 | 8.18 | 0.5 | 1.69 | 2.25 | |
| | **3** | 2.48 | 16.17 | 1.46 | 1.12 | 0.27 | 0.28 | 0.59 | 15.55 | 9.88 | 6.53 | 1.43 | 5.07 | |
| | **4** | 1.98 | 0.66 | 3.69 | 6.48 | 0.4 | 0.4 | 0.83 | 15.84 | 9.12 | 1.01 | 1.76 | 3.83 | |
| | **5** | 4.13 | 0.94 | 1.31 | 0.55 | 0.14 | 0.04 | 0.69 | 0.42 | 7.5 | 0.6 | 1.65 | 1.63 | |
| **both-mcl-sides-add** | **1** | 1.19 | 14.81 | 1.07 | 0.89 | 0.56 | 0.39 | 0.68 | 17.49 | 9.95 | 0.39 | 0.14 | 4.32 | 5.47 |
| | **2** | 12.02 | 17.72 | 0.1 | 1.19 | 0.42 | 0.4 | 0.32 | 16.21 | 4.68 | 13.71 | 1.87 | 6.24 | |
| | **3** | 14.11 | 15.84 | 1.44 | 0.21 | 0.4 | 3.27 | 0.79 | 14.72 | 7.22 | 0.25 | 1.93 | 5.47 | |
| | **4** | 3.19 | 16.06 | 5.28 | 4.03 | 2.36 | 3.49 | 17.18 | 3.21 | 0.62 | 4.03 | 1.7 | 5.56 | |
| | **5** | 2.65 | 16.96 | 4.83 | 18.3 | 1.35 | 5.06 | 0.21 | 0.23 | 11.22 | 0.63 | 1.62 | 5.73 | |
| **imu-mcl-double** | **1** | 2.05 | 0.41 | 0.84 | 1.12 | 0.42 | 0.69 | 0.92 | 16.07 | 7.55 | 0.28 | 2.04 | 2.94 | 4.09 |
| | **2** | 2.75 | 16.94 | 0.71 | 1.13 | 0.46 | 0.25 | 0.69 | 15.37 | 0.65 | 6.48 | 1.93 | 4.31 | |
| | **3** | 3.54 | 16.17 | 1.14 | 1.17 | 0.46 | 0.26 | 0.82 | 12.04 | 8.32 | 0.29 | 1.97 | 4.2 | |
| | **4** | 5.83 | 17.62 | 5.78 | 0.27 | 0.31 | 0.36 | 0.62 | 8.76 | 1.89 | 3.49 | 1.53 | 4.22 | |
| | **5** | 2.11 | 0.87 | 0.82 | 1.16 | 10.5 | 3.52 | 1.12 | 8.92 | 7.77 | 15.63 | 0.19 | 4.78 | |
| **imu-mcl-double** | **1** | 1.96 | 14.1 | 1.53 | 1.26 | 0.98 | 3.5 | 0.65 | 9.03 | 6.19 | 0.47 | 1.47 | 3.74 | 4.27 |
| | **2** | 4.4 | 17.39 | 1.46 | 0.07 | 0.4 | 4.3 | 0.62 | 0.5 | 6.72 | 0.3 | 1.99 | 3.47 | |
| | **3** | 2.22 | 16.07 | 0.51 | 0.92 | 0.32 | 3.25 | 0.61 | 7.79 | 4.25 | 1.37 | 1.61 | 3.54 | |
| | **4** | 2.11 | 16.99 | 1.26 | 1.01 | 0.35 | 5.37 | 4.86 | 15.12 | 2.74 | 13.12 | 1.66 | 5.87 | |
| | **5** | 1.82 | 15.09 | 10.41 | 0.26 | 0.34 | 0.29 | 1.21 | 14.32 | 5.7 | 0.43 | 1.97 | 4.71 | |