INCORPORATING HISTOGRAMS OF ORIENTED GRADIENTS INTO MONTE CARLO LOCALIZATION

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

Michael K. Norris

June 2016

COMMITTEE MEMBERSHIP

TITLE:	Incorporating Histograms of Oriented Gradients into Monte Carlo Localization
AUTHOR:	Michael K. Norris
DATE SUBMITTED:	June 2016
COMMITTEE CHAIR:	John Seng, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Bellardo, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Lynne Slivovsky, Ph.D. Professor of Electrical Engineering

ABSTRACT

Incorporating Histograms of Oriented Gradients into Monte Carlo Localization

Michael K. Norris

This work presents improvements to Monte Carlo Localization (MCL) for a mobile robot using computer vision. Solutions to the localization problem aim to provide fine resolution on location approximation, and also be resistant to changes in the environment. One such environment change is the kidnapped/teleported robot problem, where a robot is suddenly transported to a new location and must re-localize. The standard method of "Augmented MCL" uses particle filtering combined with addition of random particles under certain conditions to solve the kidnapped robot problem. This solution is robust, but not always fast. This work combines Histogram of Oriented Gradients (HOG) computer vision with particle filtering to speed up the localization process.

The major slowdown in Augmented MCL is the conditional addition of random particles, which depends on the ratio of a short term and long term average of particle weights. This ratio does not change quickly when a robot is kidnapped, leading the robot to believe it is in the wrong location for a period of time. This work replaces this average-based conditional with a comparison of the HOG image directly in front of the robot with a cached version. This resulted in a speedup ranging from from 25.3% to 80.7% (depending on parameters used) in localization time over the baseline Augmented MCL.

ACKNOWLEDGMENTS

Thanks to:

- Dr. John Seng, for all of his insightful comments and great suggestions.
- Dr. John Bellardo, for showing me how much error checking matters in network coding.
- Dr. Lynne Slivovsky, for mentoring the project that got me my first major internship.
- My mother, Naomi Norris, for supporting me through college and beyond.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Localization is a widely studied problem in robotics, with a vast swath of literature researching the topic. It is defined as the estimation of a robot's pose (location and orientation) when the robot is placed into an unknown location in a known map. Many algorithms and improvements upon those algorithms exist to solve this problem, but localization is essentially accomplished by comparing incoming sensor data to a map represented in the robot's memory.

Across many fields, the most logical technology for positioning information is GPS. GPS is critical in the mobile industry, global clock synchronization, military applications, and in branches of robotics. However, GPS is only accurate down to a few meters [23]. This is the difference between being obstructed by obstacles and navigating a path successfully. GPS is also subject to environmental interference, and cannot be used in applications without satellites (such as Mars rovers).

To work with GPS, the most common method of navigation is through straight line distance data, gathered through a sensor. By analyzing this data and comparing it to a known map in the robot's memory, it can perform one of two types of localization: *local* or *global*. Local techniques rely on knowing some sort of initial position of the robot, and cannot recover if position data is lost. For example, if the robot relies on tracking wheel revolutions and is temporarily halted (or completely displaced, as in the kidnapped/teleported robot problem [16–18]), the position estimate will be off. Global techniques have no such restriction, and can localize a robot when placed arbitrarily in a given map.

Monte Carlo Localization, or MCL, achieves global localization by passing sensor data through a *particle filter*. Particle filters are methods used approximate a number of problems in signal processing. Particle filtering has been shown by previous work to outperform other filtering methods when used with MCL [21]. A simplified look of the algorithm can be found in Figure 1.1. At a basic level, the robot first generates a large number of *particles* (shown as the initial set in 1.1) around its in-memory map of the environment, which are hypothesized poses (position-orientation pairs) that the robot could be located at. This pose is shown as an orange point with a direction at the top of Figure 1.1. It then assigns a weight to each particle according to how well sensor readings match up to the particle's readings, obtained by ray tracing from the particle to the nearest object in the in-memory map. Finally, it randomly *resamples*, or chooses, particles from that weighted set based on the particle's weight. The higher the weight, the more likely a particle is to be resampled. A colored version of this is shown on the bottom of Figure 1.1, where the larger circles have higher weights. As the image shows, particles can be chosen multiple times. The algorithm repeats this process continuously to refine the robot's true position.

Classic "Adaptive MCL" does handle the teleported robot problem, which is essentially a repeated global localization after initially determining a location. However, due to the nature of the algorithm, it takes the robot some time to realize that it has been moved. This can be problematic in a dynamic environment where the robot may be displaced frequently such as a sand dune or a windy location (in the case of an aerial robot). Shortening the time it takes to localize would improve performance of certain tasks of the robot such as sampling minerals, aerial imaging, and other tasks.

Improvements to localization algorithms can be made by incorporating different types of sensor data besides simple straight line distance. These can take the form of WiFi signal strength, radio signal strength, floor and wall texture data [23], light intensity [12], or computer vision [20]. The work of this thesis is to incorporate

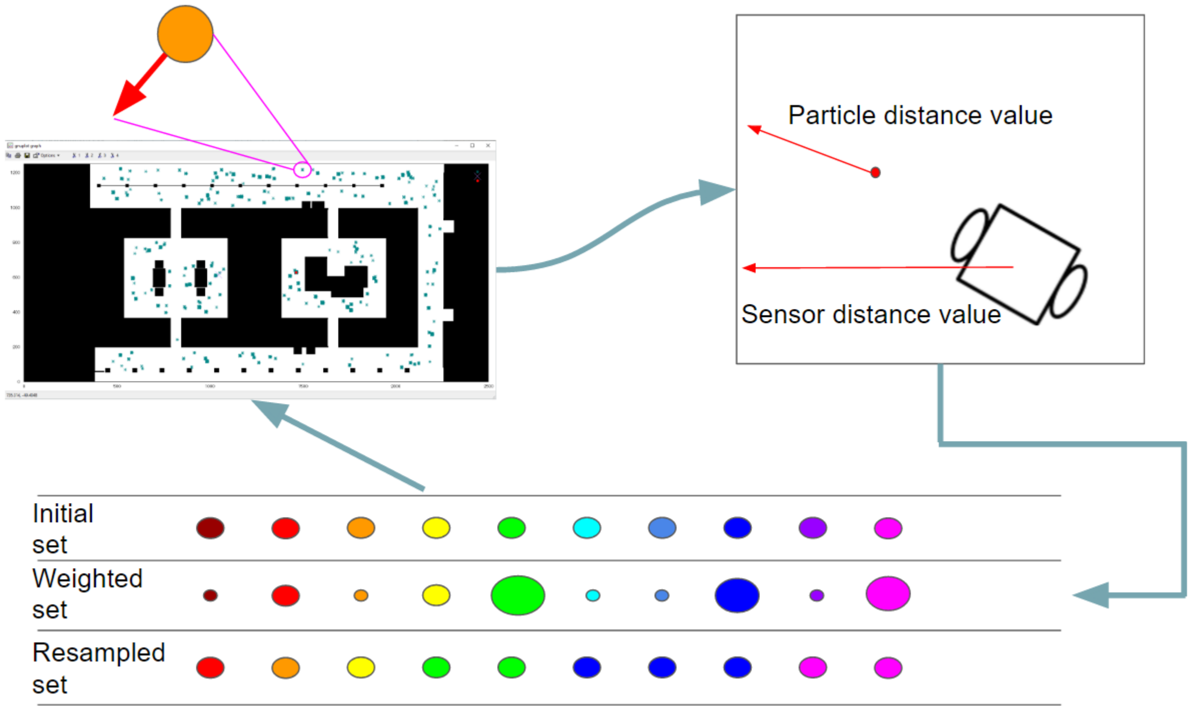


Figure 1.1: Simplified depiction of Monte Carlo's particle filtering.

computer vision specifically into MCL to improve the recovery time in the case of the teleported robot problem.

This work is organized as follows. Chapter 2 discusses a variety of related work in the field of robot localization. Chapter 3 describes background in both particle filter localization and histograms of oriented gradients. Chapter 4 discusses design of the modified MCL and computer vision algorithm using histograms of oriented gradients, then goes into implementation details of the system including the simulation environment that these results were obtained in. Chapter 5 examines the results obtained from experiments with both the baseline implementation and the improved version. Chapter 6 details possible ideas for future work in localization using computer vision. Chapter 7 concludes this work, and discusses enhancements.

Chapter 2

RELATED WORK

A great deal of previous research has gone into solving the localization problem efficiently. Localization in particular involves refinement of a set of probabilistic locations. This can apply to both the global or local (also known as *position tracking*) localization problem. The basis for more modern localization solutions lies in the Monte Carlo methods, which are a broad range of algorithms that use some form of random sampling to produce a result. This section briefly lists some of the historical contributions to localization, then goes into a technical background of the most current approach.

Kalman [22] was one of the first to describe a continual refinement of probabilistic locations. The strategy he employed became known as the Kalman filter approach to localization. The iterative improvement of probabilistic states set the basis for modern implementations. However, Kalman filters can only solve the position tracking problem; not the global problem [17].

Smith and Cheeseman [31] focused on establishing a system for associating one reference frame with another, given different degrees of error. Their work gave a way of representing the amount of certainty a robot has of its position by chaining comparisons of reference frames.

Burgard et al. [11] created what is known today as Markov localization. The name of this method refers to the Markov Property. This states that the system is memoryless, and only depends on the present state to determine future states; past states have no effect. This yields a much simpler implementation strategy than Smith and Cheeseman's reference frame comparison. The Markov solution employs a set of states of likely locations that are continually refined, and either a topological or a grid map of the area to be explored. The topological map is less precise, as positions are only distinguished by different levels in the topography. The grid type separates the map into many small cells, giving a smaller resolution for the possible locations. It solves the global localization problem successfully.

The most modern strategies are variations on Monte Carlo localization, established by Fox et al. [16–18, 32–34]. It solves the global localization problem, but it also outperforms Markov localization in both memory usage and accuracy. Because Markov localization uses discrete blocks as locations (each grid cell must be represented in memory), an Markov implementation must decide between a small block size (which increases memory usage), or a larger block size (which decreases accuracy). MCL avoids this by using a set of non-discrete locations called particles. Particles include the estimated position and orientation of the robot, as well as the probability. Chapter 3.1 describes how the algorithm works in detail.

Many modern MCL variations exist. Elinas and Little [14] introduced a localization approach called σMCL that uses in-memory maps made of 3D landmarks. Zickler et al. [35] use wireless signal strength along with a model that maps strengths to real-world locations. Of important note is that Zickler's approach did not require landmarks, as many computer vision approaches do. Röwekämper et al. [29] use a precise motion capture system (essentially high-precision cameras that feed into a control system) to show that very precise localization can be performed with modern hardware. Their results returned precision of roughly one to two centimeters in the worst case. Gil et al. [19] use SIFT (scale-invariant feature transform) features to identify objects in an environment and perform successful localization within a few iterations. It was not readily apparent how long each iteration of the algorithm took though.

Chapter 3

BACKGROUND

Chapter 1 briefly mentioned that Monte Carlo uses a particle filter to perform localization. Figures 3.1, 3.2, and 3.3 give a visual for the process occurring. At a high level, particles are initially randomly spread through the robot's internal map of the environment in valid locations. In these figures, this is the non-black area. In this work, the robot's map was created as a bitmap from the robot's simulation environment, discussed further in Chapter 4.2.2. As the robot moves and takes sensor values, it will resample particles from the old set based on how closely they match the sensor values. The particles form into groups, then down to a single point. If the localization implementation is correct, this will be on top of the robot.

Some works [23] have sought to increase accuracy in generating particles and giving a final estimation of the robot's position. Others [12] have simply explored the possibility of localizing using certain methods without comparing to a baseline MCL implementation. The goal of this thesis is to improve the speed of global localization after a robot has been kidnapped. In particular, this work incorporates histograms of oriented gradients to localize faster than a baseline implementation of MCL. This chapter gives an overview of both particle filtering localization and computer vision (specifically histograms of oriented gradients).

3.1 Monte Carlo Localization

A *basic* breakdown of the algorithm has been difficult to find in other literature, even though the pseudocode in Figure 3.4 can be found in many places. As a note: the computer vision portion of this work covered in Chapter 3.2 is intended to optimize

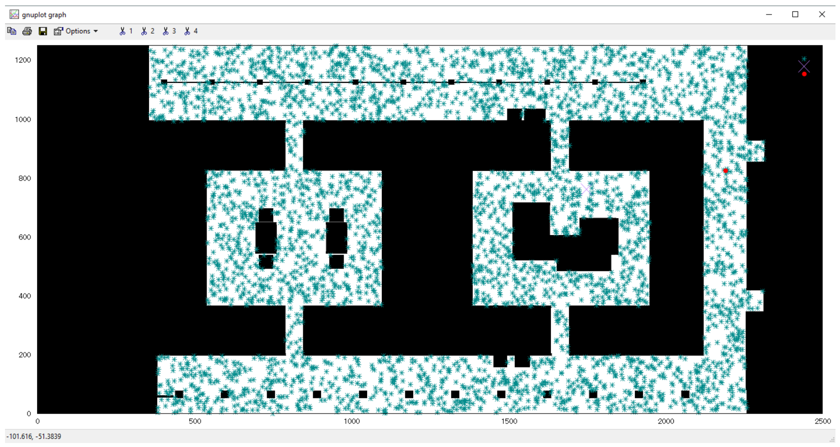


Figure 3.1: Initial particle spread

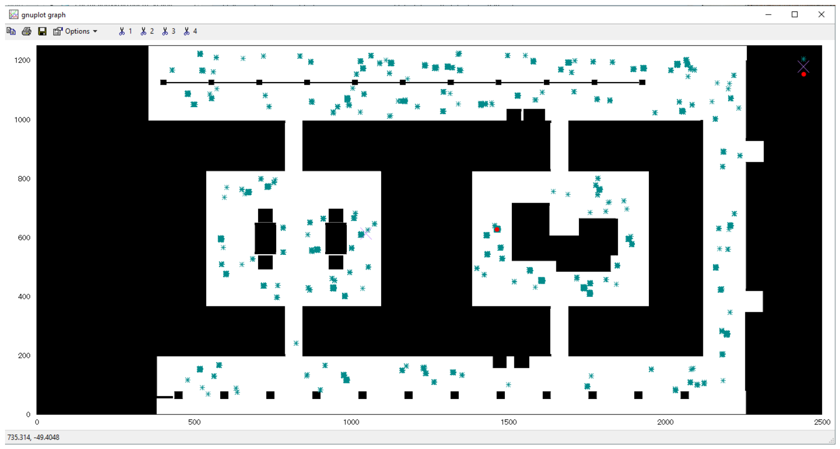


Figure 3.2: Particles trimmed down to more likely locations

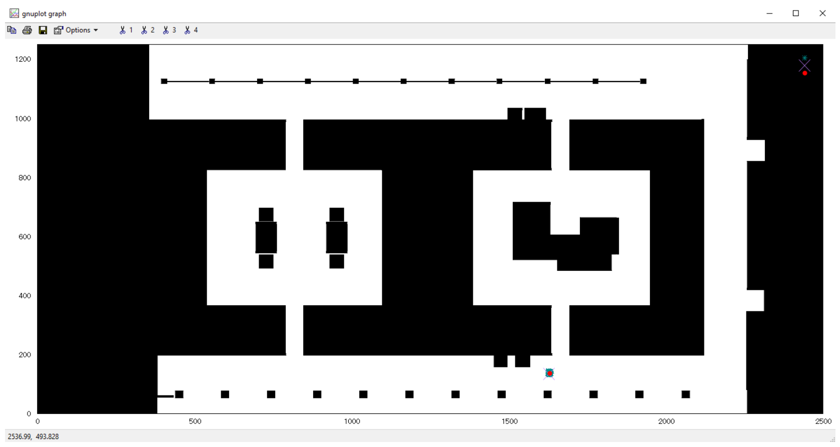
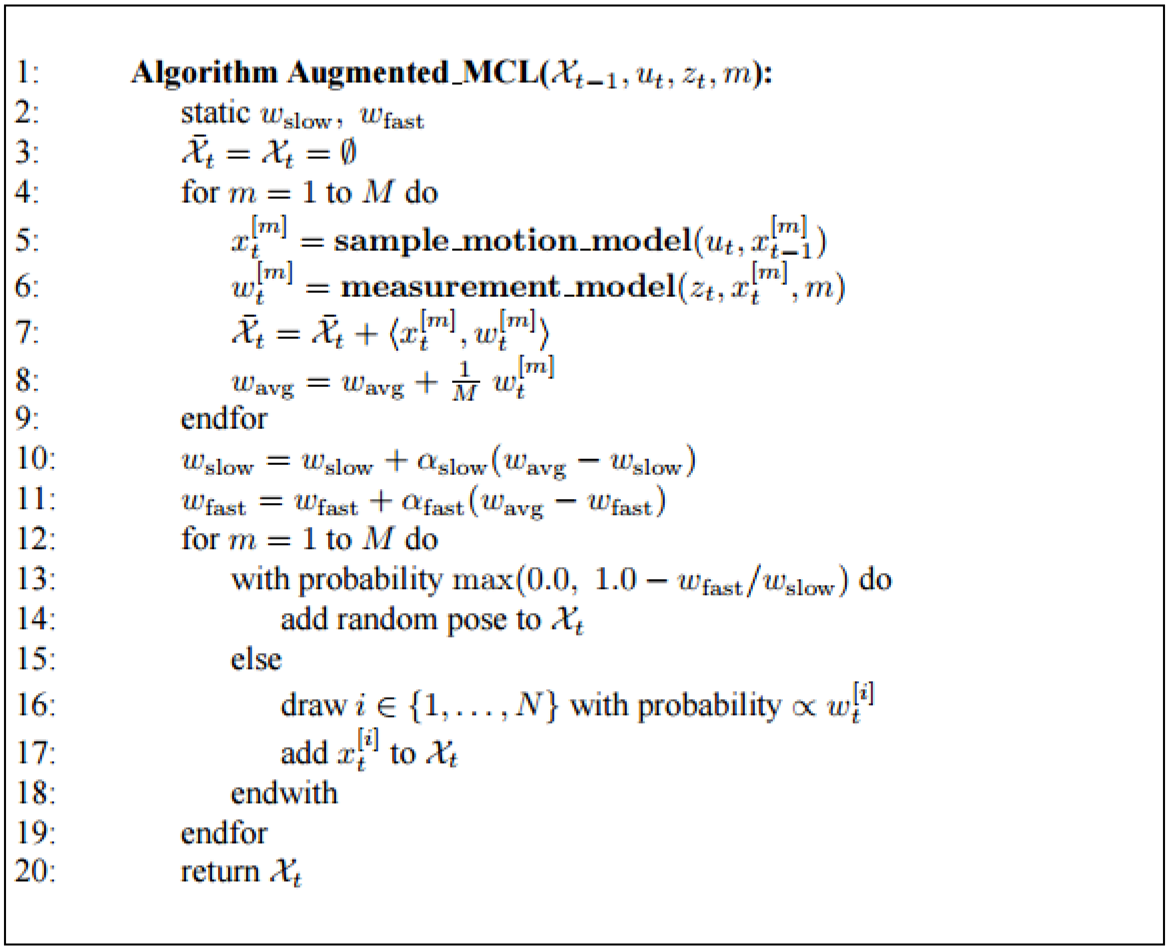


Figure 3.3: Particles localized on a point



```
1:  Algorithm Augmented_MCL($\mathcal{X}_{t-1}, u_t, z_t, m$):
2:      static $w_{\text{slow}}, w_{\text{fast}}$
3:      $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
4:      for $m = 1$ to $M$ do
5:          $x_t^{[m]} =$ sample_motion_model($u_t, x_{t-1}^{[m]}$)
6:          $w_t^{[m]} =$ measurement_model($z_t, x_t^{[m]}, m$)
7:          $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
8:          $w_{\text{avg}} = w_{\text{avg}} + \frac{1}{M} w_t^{[m]}$
9:      endfor
10:     $w_{\text{slow}} = w_{\text{slow}} + \alpha_{\text{slow}}(w_{\text{avg}} - w_{\text{slow}})$
11:     $w_{\text{fast}} = w_{\text{fast}} + \alpha_{\text{fast}}(w_{\text{avg}} - w_{\text{fast}})$
12:     for $m = 1$ to $M$ do
13:         with probability $\max(0.0,\ 1.0 - w_{\text{fast}}/w_{\text{slow}})$ do
14:             add random pose to $\mathcal{X}_t$
15:         else
16:             draw $i \in \{1, \ldots, N\}$ with probability $\propto w_t^{[i]}$
17:             add $x_t^{[i]}$ to $\mathcal{X}_t$
18:         endwith
19:     endfor
20:     return $\mathcal{X}_t$
```

Figure 3.4: Random sampling of weighted particles. [32]

essentially a single conditional in the classic MCL algorithm, seen in Figure 3.4 on line 13. The following steps have been summarized from various sources [16,17,23,30] to give a perhaps more easily understood description of the algorithm. Monte Carlo localization performs the following:

1. Initialize the set of particles randomly around the in-memory map of the environment.
2. Apply the robot's most recent movement to each particle.
3. Randomly sample a new set of particles from the previous set based on each particle's probability.
4. Take distance sensor readings.
5. Assign new weights to the sample set of particles based on sensor readings. This consists of:
    (a) A Gaussian calculation of sensor readings and particle position estimation.
    (b) Normalization of each probability.
6. Conditionally add a small number (comparatively to the set size) of particles with an average probability and random pose to the set.
7. Go to Step 2.

MCL begins by initializing the set of particles around the space to be explored (Step 1). These have random poses (both position and orientation) around the space to be explored. This means that most initial particles will be nowhere near the robot's initial pose.

In Step 2, the robot moves some distance, and then applies this transformation in movement to each particle. This means that each particle moves forward in their own

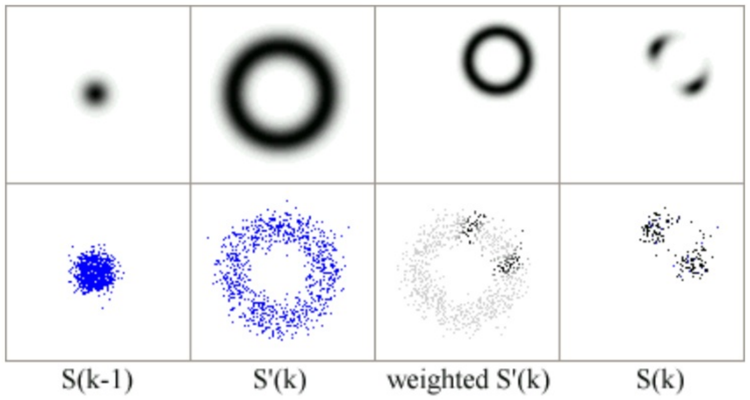


Figure 3.5: Random sampling of weighted particles. [33]

pose by a certain amount. To give a sense of scale, this "motion model" step can last for many seconds; it is not restricted to be a tiny movement forward. A small amount of Gaussian random noise is added to the movement of the particle to account for inexact motor calibration and resistance between the wheels and the environment.

Step 3 utilizes random sampling aspect of the Monte Carlo methods. Given a set of particles (which each have a certain weight), randomly sample a set of new particles based on the weight. These probabilities are initially uniform, but are quickly weighted towards particles with similar readings to the robot's location. This can be seen in Figure 3.5. Particles are first spread uniformly as seen in "S(k-1)"; after weights are calculated, the particles with the higher weights (matching the sensors most closely as in "weighted S'(k)") survive the resampling and are displayed in "S(k)".

Sensor readings are then taken at this new location in Step 4. These take the form of laser or sonar distance readings. An easy way to think of a "reading" is a straight beam that extends from the robot to the first obstacle. Laser scanners may take thousands of readings on each pass [28], but an accurate position can still be attained with orders of magnitude fewer scans. Indeed, the next step is faster with

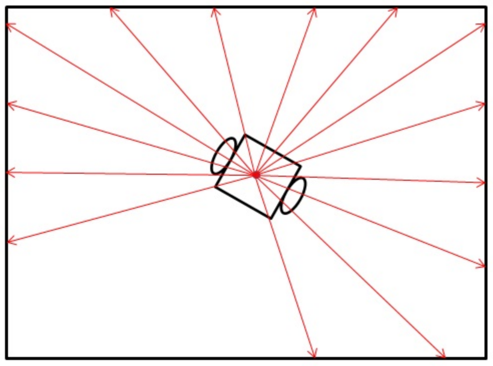


Figure 3.6: Simple laser scan visualization.

fewer scans and it is thus less optimal to use the thousands that typical hardware will provide. A simple visualization can be seen in Figure 3.6.

Step 5 compares the particle position with the sensor readings, and get a measure of how accurate each particle is. To be clear, each particle has a set of readings obtained by ray tracing the particle's position to the nearest wall of the in-memory map, and the below accuracy calculation is done on each beam for each particle. The full probability equation for a single beam is shown by formula 3.1. This consists of three probabilities added together. The coefficients a_{hit}, a_{rand}, and a_{max} are non-negative weighting parameters that sum to one. These are determined empirically based on sensor accuracy.

$$p(s^k|x) = a_{hit}p_{hit}(s^k|x) + a_{rand}p_{rand}(s^k|x) + a_{max}p_{max}(s^k|x) \tag{3.1}$$

The first (titled p_{hit} in 3.1) is given by Gaussian distribution seen in formula 3.2. In the Gaussian distribution, the mean μ represents the sensor reading, x represents the particle's reading, and σ represents the standard deviation. The standard deviation value depends on the noise associated with the sensor, and is also determined empirically. It is interesting to note that varying the standard deviation directly controls how "aggressive" the resampling process is. If this value is set too small, only the highest weight particles will be resampled and the likelihood of localizing to the wrong point is high. Too low, and most particles will be resampled and no progress is made toward localizing the robot.

$$p_{hit}(s^k|x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\left(\frac{(x-\mu)^2}{2\sigma^2}\right)} \tag{3.2}$$

The two remaining probabilities, p_{rand} and p_{max}, represent the probability of noise in the sensor reading and the sensor erroneously reading a maximum value respectively. Both are fairly small, as p_{rand} can be represented by a small percentage of p_{hit} and p_{max} is negligible in simulation.

The final weight that a particle has consists of first multiplying the probabilities of each beam associated with that particle, obtained above. Then, the weights of all particles are normalized to sum to one, to enable Step 3 (resampling of particles). The formal definition for this equation can be seen in 3.3, where K is the number of beams for each particle.

$$p(s|x) = \prod_{k=1}^{K} p(s^k|x) \tag{3.3}$$

Step 6 adds a few random particles into the set based on a condition. According to Fox [17], this step allows the robot to recover if it becomes wildly off-course but the particles all happen to be at another location. Because new particles are sampled from the old set, if the old set is all at a single location, the robot cannot correct for a complete change in pose (i.e. if it fell off a cliff and had to re-localize). However, with a small number of random samples mixed in, the random samples have a chance of being more accurate than the large grouping at the wrong location, and will have a higher probability to be sampled on the next iteration. This allows the robot to recover its position.

At the end, the robot loops back around to Step 2, as the particles are continuously refined towards the robot's true location.

Monte Carlo localization can also include adaptive sampling. When particles are grouped closely together after iterations of resampling, the number of particles can be trimmed by orders of magnitude to improve performance. This actually does not result in a loss in accuracy in practice. This is because when the robot's location is known with a high probability, the problem turns into position tracking localization (the local problem, rather than the global one). This requires comparatively very few particles to retain an accurate location. Even when the robot loses course catastrophically, Step 6 of the algorithm will simply mix in many more randomly posed particles across the exploration space, which will ensure the robot's recovery [17, 18, 33].

The condition in Step 6 is the important part, and the key piece that this work seeks to optimize using histograms of oriented gradients. Particles are added with the chance seen on line 13 in Figure 3.4. The alpha values are empirically determined decay rates, and are required to be $0 \leq a_{\text{slow}} \ll a_{\text{fast}}$ [32]. The w_{fast} will decrease much more quickly than w_{slow} when the robot is kidnapped, introducing random particles [32]. The problem with this approach is that if a_{slow} is too large (as in closer to a_{fast}), particles will be introduced randomly when they shouldn't be. If a_{slow} is too small, then random particles will never be added or will be added too slowly to localize effectively. The baseline implementation in this work used 0.001 for a_{slow} and 0.1 for a_{fast}.

3.2 Histograms of Oriented Gradients

A *visual descriptor* is a description of the content of an image, video, or other visual media. These descriptors are meant to characterize shape, color, or other distinguishing elements in the above [15]. One such shape visual descriptor is the *histogram of oriented gradients* (HOG) introduced by Dalal and Triggs [13]. HOG has been shown to perform well in object detection [15, 26] even in black-and-white images. On a pos-

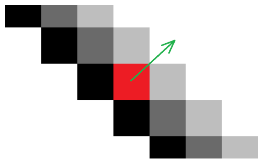


Figure 3.7: Idealized gradient for one pixel. Gradient vector in green.

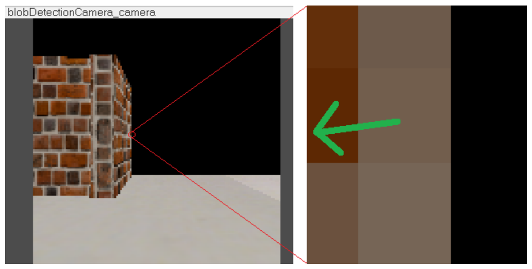


Figure 3.8: V-REP gradient for one pixel. Gradient vector in green.

sibly constrained system, only sending a third of the data to encode a black-and-white image compared to a more expensive RGB image is a bonus. There are other features detectors such as SIFT and GIST, but they perform fairly similarly to HOG [10]. In short, "HOG decomposes an image into small squared cells, computes an histogram of oriented gradients in each cell, normalizes the result using a block-wise pattern, and return a descriptor for each cell." [7] The following explanation of the algorithm uses information from [13] and [26] heavily; to avoid overciting, both are listed here.

The HOG computation begins by dividing the image up into cells. Dalal and Triggs found 8x8 pixels for each cell to be optimal in their experiments. In each cell, the *gradient vector* for each pixel is calculated. This is simply the rate of change of a function that points in the direction of greatest increase in function value. Figure 3.7 shows an idealized, zoomed-in edge in an image, while Figure 3.8 illustrates the effect in action on a wall edge from the V-REP robotics simulator used in this work. In the context of computer graphics, the gradient vector of the pixel in red is simply calculated by the difference in intensity of the pixel above minus the pixel below, and the pixel to the right minus the pixel to the left. If both parts of the vector

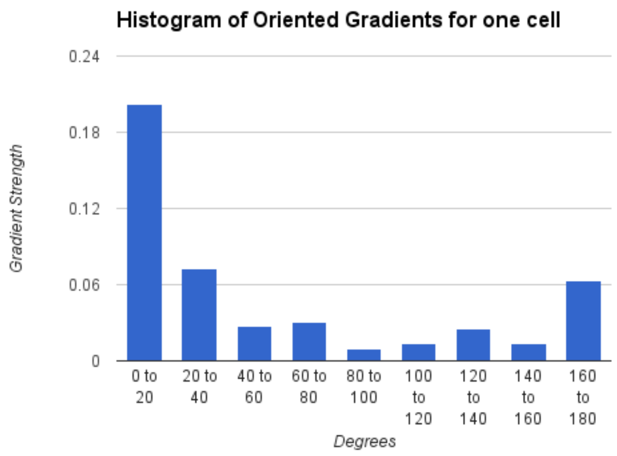


Figure 3.9: Example histogram with vectors sorted into bins.

are positive as in Figure 3.7, the gradient vector will point up and to the right, perpendicular to the edge. In Figure 3.8, the black area on the right is much darker than the pixel on the left, so the vector will point away from the black. The top pixel is also slightly darker than the bottom, so the gradient vector points slightly downward. This shows that the gradient is indeed pointing in the direction of greatest increase in pixel values.

Each gradient vector magnitude is then binned into a histogram based on the vector angle. Figure 3.9 shows only up to 180because the original implementation by Dalal and Triggs used unsigned gradients. This means that a vector pointing in an arbitrary direction will be put into the same bin as a vector pointing in the exact opposite direction. The rationale here is that it doesn't matter if an edge is trending dark to light or light to dark; the only concern is that it exists. The gradients in each bin of the histogram are added together to form the Y axis of the histogram, meaning that very strong vectors (perpendicular to very sharp edges in the image) will effect the bin magnitude more. To prevent a strong gradient vector right on the edge of a bin from only affecting a single bin, the algorithm splits gradient vectors between the closest two bins. Thus a vector at angle 120would get split its magnitude

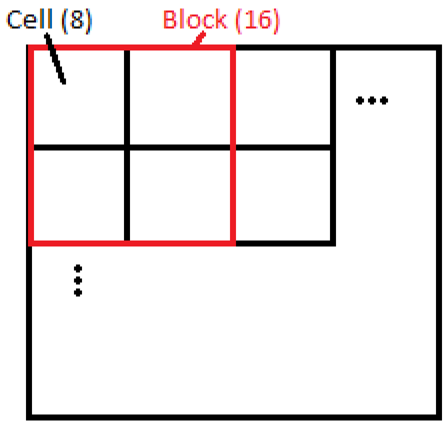


Figure 3.10: Cells vs blocks in an image.

evenly between 110and 130 A vector at 125would be split 3/4 in the 130bin, and only 1/4 in the 110bin. Dalal and Triggs found that 9 bins worked optimally. In the end, this histogram only has the same number of values as bins. This reduces the number of values needed from 64 gradient vectors each with their own angle to only 9 sums of magnitudes of the gradient vectors in that bin. Programmatically, this can be thought of as an array (or similar data structure) of 9 float values.

It is important to note that due to the nature of gradient vectors, HOG is immune to contrast changes of the source image. This is because even though the vector values themselves will change (as contrast increases, both vector values will increase also), the ratio between them will not. The angle is thus the same, and the gradient magnitude will be binned into the same container of the histogram. This is important in localization outside, where images of a location may change in brightness depending on the time of day.

After calculating the histogram for each cell, blocks are formed in groups of cells. Figure 3.10 shows a 2x2 grouping that Dalal and Triggs used. This is referred to as *local contrast normalization* [13]. The histograms of each cell in a block are concatenated into one vector, and then divided by the magnitude of the vector. This means that each block is normalized by a different value depending on the gradient strengths of the cells that comprise it. The next block to process is only *one* cell width to the right, including the rightmost cells of the first block and two new cells. This overlap means that cells appear multiple times (non corner edge cells twice, inner cells four times), but are normalized by different cells depending on the block that normalized it. The goal of this can be thought of as capturing contrast changes in a small area to assess gradient strengths instead of trying to increase contrast over the entire image. As Dalal and Triggs stated, "This may seem redundant but good normalization is critical and including overlap significantly improves the performance."

The result is the "vector of all components of the normalized cell responses from all of the blocks" [13]. In other words, each block will return a 36 element (9 bins, 4 cells per block) vector of values. The size in bytes of the vector for the entire image is a concatenation of these block vectors, and is given by the formula in 3.4.

$$size = numBlocks * numCellsPerBlock * sizeof(float) * numBinsPerCell \quad (3.4)$$

Dalal and Triggs set out with the explicit intent to identify people in images by training a support vector machine classifier with these HOG results. In the context of robot localization, this is not strictly necessary. If a camera-equipped robot knows that an image does not match up with some cached form of the HOG result vector, it knows it is probably in the wrong location.

Chapter 4

IMPLEMENTATION

This chapter discusses the proposed modifications to MCL with computer vision, and then the implementation of these changes in the V-REP robotics simulator [3]. As a broad overview: both laser scanner and camera image data are collected from sensors controlled by Lua scripts mounted on a mobile robot inside the simulator. These are passed to a C++ client, which then processes laser scanner data for classic MCL and passes the image data to OpenCV [5] to compute histograms of oriented gradients for each cell in the image. A visualization of the gradients computed on each cell can be seen in Figure 4.1 (credit to [9] for method of producing a visual HOG on each cell), which also gives an overview of the bigger process. The captured data is compared against a cached version to determine whether the robot should add randomly posed particles. The client also pipes data to gnuplot [4] to give a visual representation of the locations of each particle.

4.1 Modifying Monte Carlo Localization

As mentioned in Section 3.1, the conditional in line 13 of 3.4 is the target for improvement. The goal is to mix in random particles in a faster way if the robot realizes that it has been displaced.

The image HOG captured from the robot can be compared to a cached histogram of the closest location and orientation of the robot's hypothesized location. If the two histograms are different above some empirically determined threshold, and the histograms continue to be above this threshold as the robot moves (checking the cache

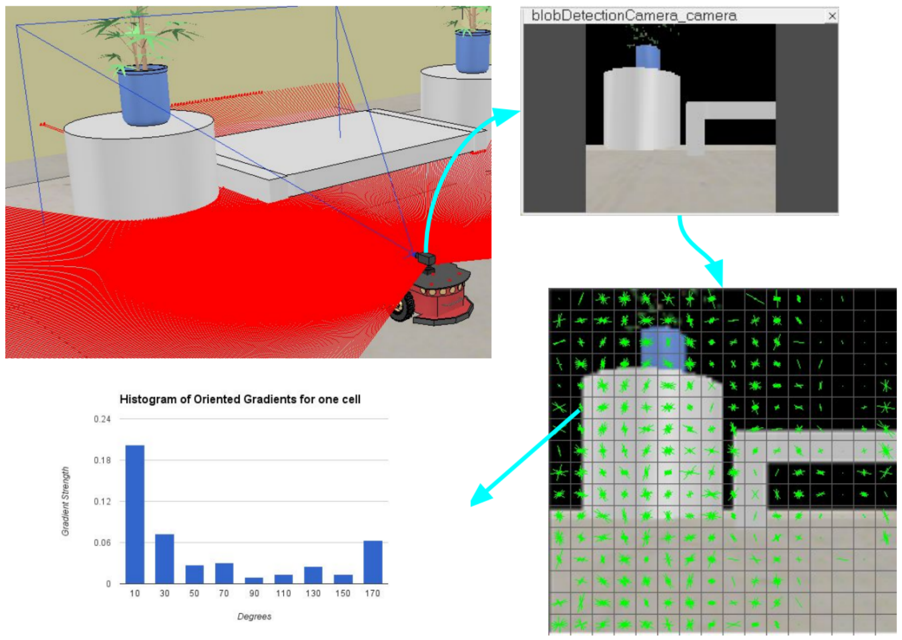

blobDetectionCamera_camera
Histogram of Oriented Gradients for one cell
Gradient Strength
Degrees
0
0.06
0.12
0.18
0.24
10
30
50
70
90
110
130
150
170

Figure 4.1: Overview of the modified MCL process.

in different areas as the robot itself moves to different areas), then the robot knows to mix in random particles.

Thus the implementation strategy is as follows: the remote API client (the primary controlling system for the robot) will first move a floating camera around the simulator, capturing images and computing histograms at various distances and orientations. The result is a series of location-orientations that map to a HOG for the image at that location-orientation. The robot can then do a fast lookup to check whether its hypothesized position's histogram matches with what it sees in front of it. If it does, then no random particle inclusion is necessary. If it does not match for a period of time, then it will mix in random particles in the same way classic Monte Carlo Localization does.

4.2 Implementation Environment

Implementation of this work simulates the robot movement and sensor data collection inside the V-REP robotics simulator, while a V-REP *remote API client* handles localization using scanner data, image processing using OpenCV, and output to gnuplot for particle visualization on a map.

4.2.1 OpenCV

OpenCV is a computer vision framework with interfaces available in C, C++, Python and Java. The C++ interface was used for this project for use with the V-REP remote API client. It implements a histogram of oriented gradients class that can perform the computation to return the result vector described in Chapter 3.2 and general object detection.

OpenCV's `HOG.compute()` function calculates a histogram of oriented gradients for an image. This image is given as a Mat object with dimensions 128x128 in this work, of type CV_8UC1. The image captured from V-REP is black and white, and thus only requires one byte per pixel as opposed to an RGB triplet.

As opposed to the ray tracing which is done on readings for every particle, the image processing comparison is done only after initial localization onto a point. The call to OpenCV's `HOG.compute()` is thus only made once per MCL iteration.

4.2.2 V-REP Robotics Simulator

The V-REP robotics simulator by Coppelia [3] is a free (with paid options for additional functionality), open source simulator that includes a host of example plugins, great documentation, hundreds of models and a complex physics engine. This was chosen over a physical robot due to cost (additional sensors in a simulator are free),

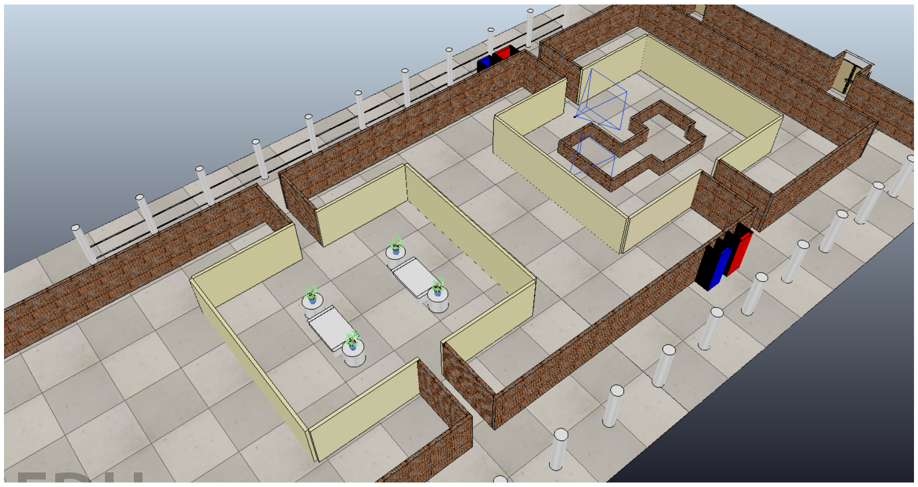


Figure 4.2: Simulated environment of the Cal Poly CSC offices

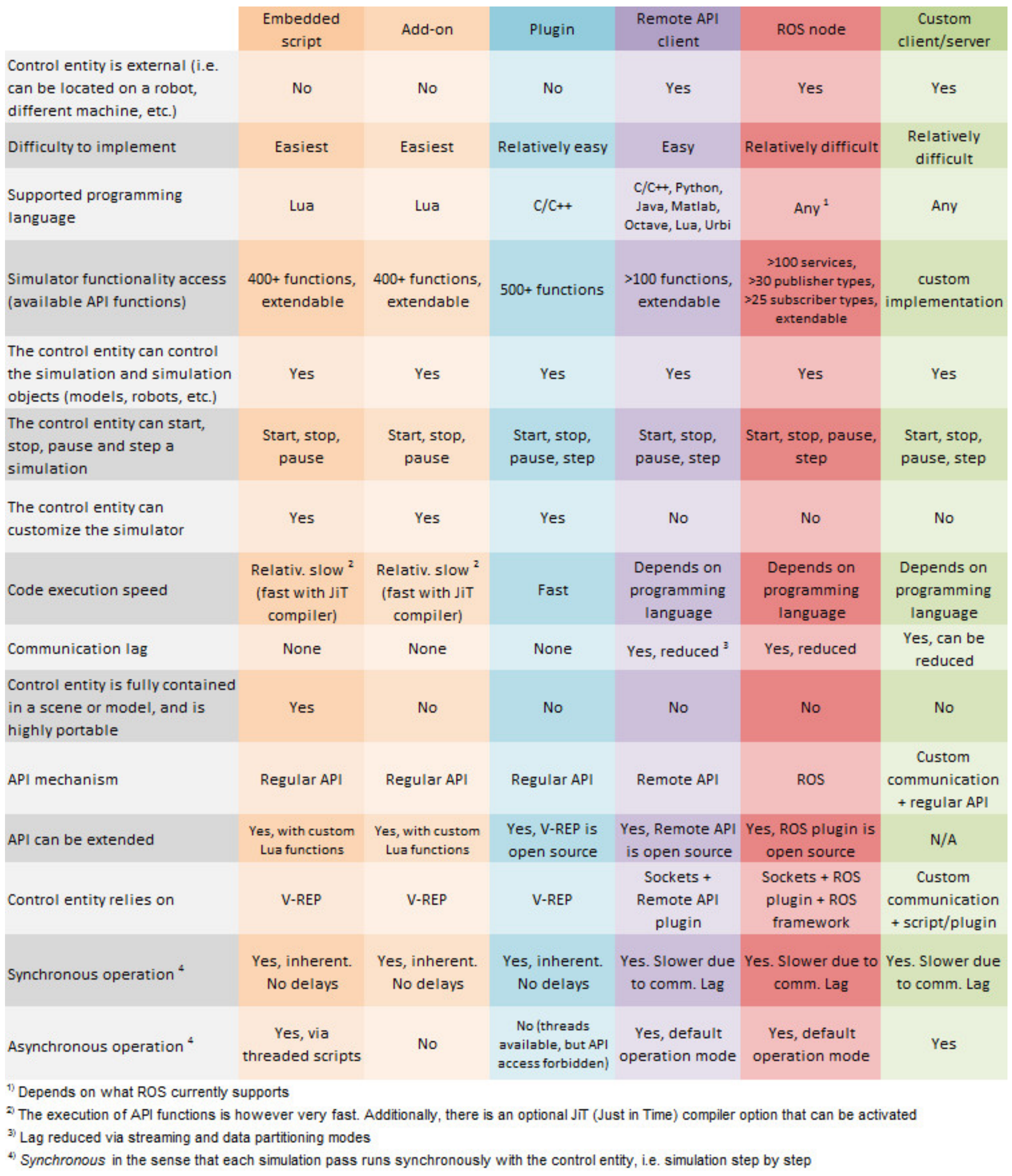
time (additional sensors in a simulator don't take a week to ship), logistics of checking out a physical laser scanner, and securing physical locations to test.

Figure 4.2 shows the environment created for the robot to localize in: a to-scale recreation of the Cal Poly Computer Science professors' offices. The environment is 50 meters by 25 meters, closely matching the real area. Finer details such as individual office doors and windows were considered not strictly necessary and thus omitted. Figure B.1 shows the map used by the robot to localize and plot particles via gnuplot. This was generated by taking the positions of each object in the V-REP environment via V-REP's `simxGetObjectPosition`, made available to the remote API client by V-REP's framework.

The Pioneer P3DX robot was chosen to navigate the simulated environment as it was a robot included with V-REP. It was equipped with the V-REP's simulated version of the Hokuyo URG 04LX UG01 laser scanner to take distance measurements, and V-REP's "blob detection camera" that (contrary to the name) was used to simply capture a 128x128 pixel black-and-white image of the view in front of the robot without additional blob processing. This camera had a range of 2 meters and a field of view of 60 degrees. Color images from the camera were deemed not strictly necessary as HOG functions just as well without it, so black and white images were captured. This image was delivered to the remote API from V-REP's `simxGetVisionSensorImage`, the V-REP remote API client function for getting camera image data. This returns a series of bytes where each byte represented the brightness of a pixel from 0 to 255. No image header information is attached to this byte stream.

There are several means of communicating commands to robots inside V-REP. These include embedded scripts, add-ons, plugins, remote API clients, ROS nodes, and custom client/server setups. Figure 4.3 shows the different advantages and disadvantages of each. The Remote API client was chosen for use in this project, because it is the most similar to code running on a real robot if this were to be used in the real world.

The problem eventually encountered with the remote API was that V-REP takes some time ranging in the hundreds of milliseconds to send data between the simulator and a remote API client. V-REP requires `simxGetVisionSensorImage` to be called in non-blocking mode of operation, so attempting to access the image data directly after this call will produce a crash. To take images from V-REP, `Sleep` calls in between calls to `simxGetVisionSensorImage` were inserted. It is recommended to use an embedded Lua script (which can be adapted for real-world use) for future use of V-REP for repeated camera image capture which will not incur this communication time. The laser scanner did not experience this issue because it transmitted much less data (one float value for each reading compared to thousands for each HOG).



	Embedded script	Add-on	Plugin	Remote API client	ROS node	Custom client/server
Control entity is external (i.e. can be located on a robot, different machine, etc.)	No	No	No	Yes	Yes	Yes
Difficulty to implement	Easiest	Easiest	Relatively easy	Easy	Relatively difficult	Relatively difficult
Supported programming language	Lua	Lua	C/C++	C/C++, Python, Java, Matlab, Octave, Lua, Urbi	Any [1]	Any
Simulator functionality access (available API functions)	400+ functions, extendable	400+ functions, extendable	500+ functions	>100 functions, extendable	>100 services, >30 publisher types, >25 subscriber types, extendable	custom implementation
The control entity can control the simulation and simulation objects (models, robots, etc.)	Yes	Yes	Yes	Yes	Yes	Yes
The control entity can start, stop, pause and step a simulation	Start, stop, pause	Start, stop, pause	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step
The control entity can customize the simulator	Yes	Yes	Yes	No	No	No
Code execution speed	Relativ. slow [2] (fast with JiT compiler)	Relativ. slow [2] (fast with JiT compiler)	Fast	Depends on programming language	Depends on programming language	Depends on programming language
Communication lag	None	None	None	Yes, reduced [3]	Yes, reduced	Yes, can be reduced
Control entity is fully contained in a scene or model, and is highly portable	Yes	No	No	No	No	No
API mechanism	Regular API	Regular API	Regular API	Remote API	ROS	Custom communication + regular API
API can be extended	Yes, with custom Lua functions	Yes, with custom Lua functions	Yes, V-REP is open source	Yes, Remote API is open source	Yes, ROS plugin is open source	N/A
Control entity relies on	V-REP	V-REP	V-REP	Sockets + Remote API plugin	Sockets + ROS plugin + ROS framework	Custom communication + script/plugin
Synchronous operation [4]	Yes, inherent. No delays	Yes, inherent. No delays	Yes, inherent. No delays	Yes. Slower due to comm. Lag	Yes. Slower due to comm. Lag	Yes. Slower due to comm. Lag
Asynchronous operation [4]	Yes, via threaded scripts	No	No (threads available, but API access forbidden)	Yes, default operation mode	Yes, default operation mode	Yes

[1] Depends on what ROS currently supports
[2] The execution of API functions is however very fast. Additionally, there is an optional JiT (Just in Time) compiler option that can be activated
[3] Lag reduced via streaming and data partitioning modes
[4] *Synchronous* in the sense that each simulation pass runs synchronously with the control entity, i.e. simulation step by step

Figure 4.3: Means of communication around V-REP [3]

4.2.3 Qt Creator and MinGW

Qt Creator [6] is an IDE meant for development with Qt, a cross-platform application development framework. Qt Creator allows a user to choose a specific version of Qt (Desktop Qt 5.5.0 in this work) and compiler (MinGW 32-bit in this work) to use. At the time of starting this work, V-REP samples all used Qt Creator make files to build, so Qt Creator was chosen as the primary remote API client development system for this work. C++ was chosen as the main implementation language for the same reason.

The Qt project file used for including sources and libraries has a very easy-to-use syntax that reads like a simple coding language. For example, including files in the path is done by `INCLUDEPATH += "path/to/file"`.

4.2.4 Position Tracking

The position tracking portion of this work is referred to as the "motion model" in 3.4, and has been a well-solved problem for quite some time (this work uses the descriptions of trigonometry from Lucas [25] in this subsection, but similar descriptions exist in a variety of resources). Position tracking is performed on each particle; each particle is moved forward in their respective directions by the same amount that the robot moved, plus or minus a random measure of error. This error is generated from the Box-Muller transform [8], a pseudo-random number sampling method.

When the robot is going straight, distance traveled is given by the classic distance formula (distance equals rate times time, where the rate is given in meters per second). Time passed was determined using `std::high_resolution_clock`. The rate is a defined constant for wheel speed. When the robot backs up and turns (which happens when obstacles are within a certain threshold distance), the wheels will move

$$\begin{aligned}\bar{s} &= (SR + SL) \;/\; 2\\ \theta &= (SR + SL) \;/\; b + \theta_0\\ \mathrm{x} &= \bar{s}\cos(\theta) + x_0\\ \mathrm{y} &= \bar{s}\sin(\theta) + y_0\end{aligned}$$

Figure 4.4: Position tracking formulas.

at different velocities and the reckoning becomes slightly trickier. To find the distance traveled in x and y to apply to each particle, approximations [25] can be used as seen in Figure 4.4. Both *SR* and *SL* below are the arc length of the turn for the right and left sides respectively, b is the width of the robot, θ_0 is the initial angle of the particle, and x_0 and y_0 are the coordinates for the initial position.

4.2.5 Gnuplot Particle Visualization

As previously stated, data was piped to gnuplot for graphing. Figures B.2 and B.3 show some images generated from gnuplot. Gnuplot supports command-line style input, which is necessary for this application. More than one graphing library was tried and discarded over the course of this work. Of note, Qt's own C++ graphing library called through QWidget did not work because it required that a QApplication be started, which V-REP forbade at the time of implementation. The final result was using `popen` on gnuplot as shown in directly below and calling `fprintf` to write gnuplot commands to the new process.

```
FILE *pipe = popen("gnuplot -persist", "w");
```

Appendix D contains the gnuplot commands used for graphing. Newline characters denote the actual end of each command. It is important to note that entering a newline before the first long command is complete results in error. The <p_x> and <p_y> actually represent many values; they were replaced by newline-separated float values for each particle position x and y. The <a_x> and <a_y> represent the robot's actual location, and are a single set of values. Lastly, the <m_x> and <m_y> are again a single set of two float values, and are the robot's most likely location, represented by the average of the highest weight particles. Again, this average will only be single point after successful localization. The "map.png" line below allows the particles to be plotted on top of the graphic shown in B.1. This implementation used 4000 particles total.

4.2.6 Remote API Client

Extensive discussion has been given about replacing the single line in the pseudocode to optimize the algorithm. This section goes over the implementation of the remote API client that will implement this more holistically, incorporating the points at which V-REP calls, position tracking, comparison of scan data, and plotting are made. Referring to the broken-down steps in Chapter 3.1 may be helpful for understanding how the pieces of MCL fit together in this section. *Robot* in the following section refers to the simulated robot inside V-REP with accompanying sensors, and *remote API client* refers to the controlling code that performs MCL and processes images based on the sensor data from the robot.

Before entering the main program loop, a call to `simxStart` is made using `argv` parameters that are passed in when V-REP begins simulating. Several successive calls are made to obtain *handles* to V-REP objects such as the robot and camera objects. These are used later when plotting the robot's real location and setting wheel speeds. Particles are initialized with random position and orientation around the area. An array of readings for each particle is computed by 2D ray tracing of white locations to the first dark pixel in the map seen in Figure B.1. Pixel resolution to simulated-environment distance is 1 pixel to 2 square centimeters. The control loop is then entered, and the first of two main branches is chosen: if this is an initial run where a

```
HOGDescriptor d(Size(res[0], res[1]), Size(16, 16), Size(8, 8), Size(8,
8), GRADIENT_BINS, 1, -1, HOGDescriptor::L2Hys, 0.2, false,
cv::HOGDescriptor::DEFAULT_NLEVELS);
```

Figure 4.5: HOGDescriptor object used in these experiments (`GRADIENT_BINS` was varied).

cached HOG map must be computed, the program will generate one. Otherwise, it will read the cached map from disk and proceed localizing.

Generating the HOG map was done by moving a camera around the simulator to a grid of points and orientations at those points, delivering the image to the remote API client, processing that with OpenCV, and storing that value for this location on disk. Certain areas were skipped when moving the camera around that the robot would not be able to enter, marked in black on the map. Additional `Sleep` calls were required in between moving the camera, setting the orientation and collecting the image because of the V-REP-mandated non-blocking modes. The OpenCV HOGDescriptor in Figure 4.5 was created. Some notable parameters are the resolution (128x128), block size (16x16), cell size (8x8) and step size (8x8) which are the same values as seen in Chapter 3.2. This object then called the `HOGDescriptor.compute()` function which returned a vector of descriptors to be cached.

After generating the cached HOGmap, the remote API client begins the motion model described in Section 4.2.4. The remote API client then retrieves the current scan data. This is initially sent from the Lua script controlling the laser scanner inside the simulator. The laser scanner Lua script by default sends absolute positions relative to the laser scanner; it was modified to send raw distance measurements as float values. This particular scanner sends several hundred values on each pass [28], but only 50 total values were used. As discussed in Chapter 3.1, localization can be performed accurately with many fewer readings. A small amount of noise (also from the Box-Muller transform [8]) was added to the distance readings to reflect interference that would be observed in a real-world environment. The standard deviation in Equation 3.2 for this work was 3.0 to 6.0 for all tests. Weights were then calculated for each particle after the readings were converted to double values instead of floats. The p_{hit} value is obtained for each particle by multiplying the equation in 3.2 for each reading (which can be seen in equation 3.3), which becomes a value fairly close to 0 on the order of 10^{-60}. On the experimental platform, single precision floats are only accurate to roughly 10^{-38} [2] while double precision is accurate to roughly 10^{-324}, making it more than adequate for the task. The final weight assigned to the particle is then obtained by dividing the value obtained from equation 3.3 for one particle by the total from all particles, which gives a weight much closer to `1 / number_of_particles` (within one or two magnitudes). This is termed the "normalized weight" and is used for the resampling process. The sum of the normalized weights of every particle is 1.0.

The remote API client can declare itself initially localized in two ways: measuring how far apart every particle is to ensure a single grouping, or simply waiting a certain number of iterations of the algorithm and assuming localization occurred successfully. Both were tested in this work, though measuring each particle's distance from one another added approximately 100ms overhead to each iteration of the algorithm. The remote API client then captures an image from the current position from the camera mounted on the robot. The histogram of oriented gradients is computed on this image, and then compared to the cached HOG map location that the remote API client believes the robot to be. Thus the second central conditional of the modified MCL arises: if this observed HOG value remains different than the cached versions for a period of time, then the remote API client will mix in particles with random position and pose.

In the other case, particles are resampled from the previous set of supposed "good" particles. This means that the new set of particles is chosen uniformly at random

Algorithm 2 Resampling

```
1: $//Normalizing$
2: $running\_total \leftarrow 0$
3: $zones \leftarrow$ **empty dictionary**
4: **for** m = 1 to M **do**
5:     $X_m.weight \leftarrow X_m.weight/total\_weight$
6:     $running\_total \leftarrow running\_total + X_m.weight$
7:     $zones[m] \leftarrow < running\_total, X_m >$
8: **end for**
9: ...
10: $//Resampling$
11: **for** m = 1 to M **do**
12:     $sector \leftarrow$ rand(0.0, 1.0)
13:     $X_m \leftarrow zones.$find($sector$)
14: **end for**
```

Figure 4.6: Resampling algorithm pseudocode.

from the old set; the normalized weights of each particle in the old set are essentially the likelihood that they are going to be resampled. This resampling can be thought of as a circle where each particle weight represents one single sector. All of these sectors together form a single complete circle. The higher the weight, the larger the sector will be that represents that particle. The uniform random float from 0 to 1 will act like a spinning arm mounted at the center of the circle, selecting a sector on every iteration of the resampling process. Because the normalized weights conveniently add to 1.0, the resampling algorithm used can be seen in Figure 4.6.

X is the set of particles, *total_weight* was obtained during Equation 3.3, and *zones* was implemented using a C++ `std::map` in conjunction with the `std::map::upper_bound` function to find the correct sector.

In either case (resampling of old particles or inclusion of new particles with random position and pose), the 2D ray tracing is performed again to obtain the updated readings for each particle. At the end of this process, each particle is plotted by piping commands to gnuplot, and the algorithm repeats from the top of the main control loop.

Chapter 5

RESULTS

The goal of this thesis is to improve Monte Carlo localization in terms of speed, even when confronted with the teleported robot problem after initial localization. Thus is is most logical to compare the work of this thesis to a baseline MCL in a dynamic environment and assess accuracy, memory usage, and speed at which the particles can converge toward the robot. Unit testing of various key functions (Gaussian probability density function, ray tracing) is, of course, important. However, judging the work as a whole compared to a baseline is the primary method of validation.

The first section of this validation covers additional memory and disk usage in the computer vision method. The second section displays the results of using different numbers of bins in each histogram. The third section measures the time it takes for particles to converge to a certain threshold of accuracy; this is arguably the most important, as it determines whether the change to computer vision improved the algorithm. The final section of this chapter analyzes and draws conclusions from each prior section.

All experiments took place on a desktop machine with an Intel 2500k CPU, 8 GB of RAM, an Nvidia GTX 970, and a 256 GB SSD running an installation of Windows 10.

5.1 Memory, Disk Usage, and Computation Time

As discussed in Chapter 3, the output of the histogram computation is a vector of float values given by the formula 3.4. As discussed in Chapter 4.2.2, images on the robot were taken as 128x128 pixel images. Thus the overall size of the vector is given by the formula in equation 5.1. Note: the number of bins in 5.1 is left as a variable as it was a variable value in these experiments that ranged from 5 to 18. The size of a float on the simulation platform was 4 bytes and has been substituted into the equation.

$$size = numBlocks * numCellsPerBlock * sizeof(float) * numBinsPerCell \tag{5.1}$$

$$size = (128/8 - 1) * (128/8 - 1) * 4 * numBinsPerCell \tag{5.2}$$

$$size = 900 * numBinsPerCell \tag{5.3}$$

The bin sizes tested were 5, 7, 9, and 18. Computing the HOG for an image took the the OpenCV `HOG.compute()` function alone approximately 120 miliseconds to complete. Compared to the rest of the algorithm (particularly the 2D ray tracing for every particle), this is a fairly cheap computation. The baseline algorithm took roughly 2 seconds to complete an entire iteration, while the computer vision modifications took roughly 3 seconds to complete an iteration. This additional second is due to the `Sleep` call required to get valid data from the simulated camera into the remote API client. As discussed at the end of Chapter 4.2.2, a Lua plugin inside the simulator would not have this overhead.

The other two parameters varied in these experiments were the number of points at which the floating camera took images in the simulator, and the number of images taken at those points in different orientations. The points at which the camera took images ranged from 10 by 10 on each axis (each point covering a range of 5 by 2.5 meters) to 40 on the X and 30 on the Y (each point covering a range of 1.25 to 1.67 meters). Both 4 (0, 90, 180, and 270 degrees) and 8 (0, 45, 90, 135, 180, 225, and 270 degrees) orientations were tested. 2 orientations was tested with both 0/180 and 90/270 degrees, but failed to localize regardless of other parameters.

5.2 Comparing Histogram Binning

Dalal and Triggs [13] have shown that 9 bins works well for histograms of oriented gradients in object detection. Lower bin values (5, 7) were tested in this work in an attempt to reduce the memory footprint. A much larger bin value (18) was also tested to see if any benefits are gained.

The goal of these tests was to find the lowest number of bins that could still give a clear threshold when the robot was kidnapped. Using different numbers of histogram bins affects how well the remote API client can discern if the area captured in front of it matches up to the cached location. These runs used the maximum of other parameters tested (8 orientations and 40 by 30 points for the histogram cache), making the histogram bins the only possible "weak variable". Using fewer bins logically reduces the ability to distinguish images because the gradient vectors are being sorted into more of the same bins. Figure 5.1 shows that only using 5 bins does not provide a clear horizontal boundary for when the robot has been teleported. Right before the teleportation (red vertical line), the histogram difference jumps to approximately 200 even though the robot is still on track. Even though one high histogram difference is observed later, it doesn't occur for a long enough period to distinguish teleportation from error. The robot ultimately fails to localize in this run because no consistent threshold can be obtained. Figure 5.2 shows a clearer difference when the teleportation happens. Values rise after the teleportation to around 700 before the robot localizes. Appendix C contains data from all four bin sizes.

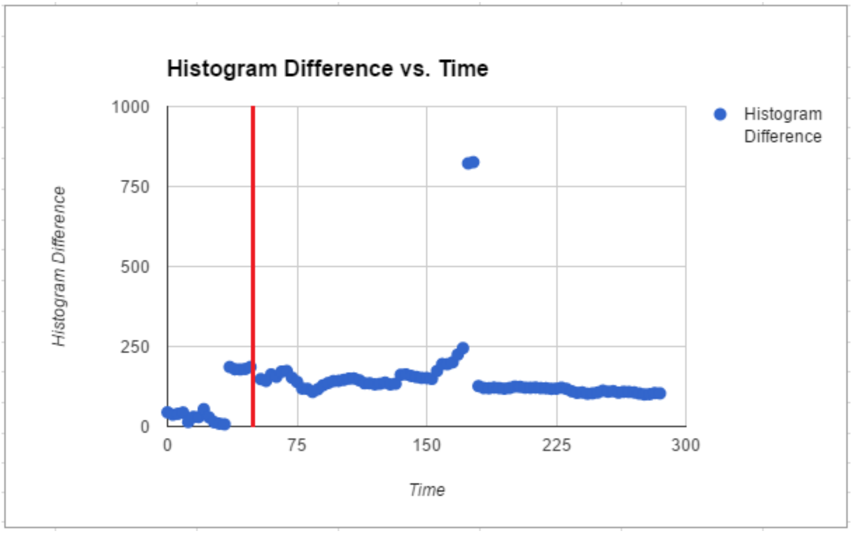


Figure 5.1: Comparing the difference in output gradient vector of the captured image against the cached image with 5 bins.

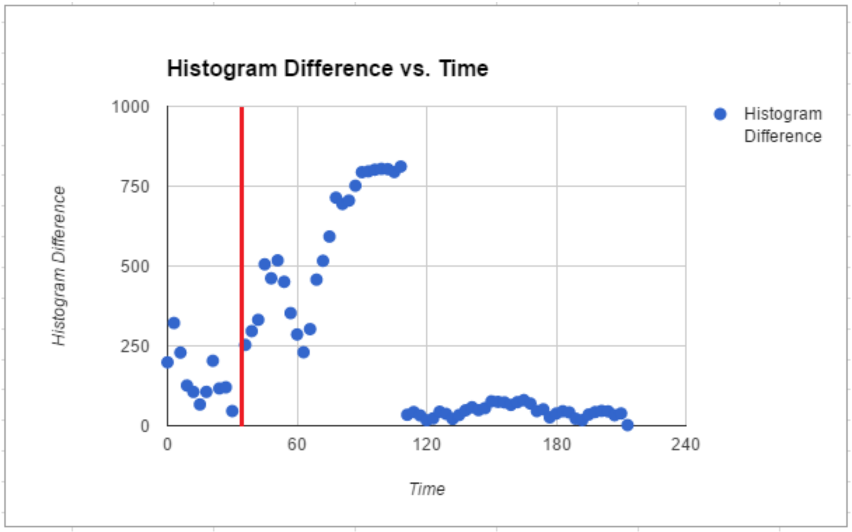


Figure 5.2: Comparing the difference in output gradient vector of the captured image against the cached image with 18 bins.

5.3 Convergence After Kidnapping

The goal of changing the parameters covered in this section is to minimize memory usage (in terms of bins, points at which images were taken, and orientations) while still localizing successfully. Runs of variations on every parameter can be found in Appendix A.

The most minimal and still consistently successful configuration was using 9 histogram bins, 1.67 x 1.25m areas for the cached camera locations, and 4 orientations at each area. The number of points at which the camera captured image had a great deal to do with the camera's field of vision. The camera had a perspective angle of 60 degrees and a visual depth of 2 meters. Objects beyond this simply appeared black in the image. If the points at which the cached image HOGs were created is too far apart (i.e. there is a possibility of the robot ending up in a gap where no images were captured), then the likelihood of the robot-mounted camera matching up to the hypothesized location is much lower.

As mentioned in Appendix A, the red line in the figures in this section Figure 5.3 shows the time taken for the baseline implementation to localize after teleportation. After teleportation, the distance from the actual location to the hypothesized location rises to above 400. It takes the remote API client over 80 seconds to successfully include random particles and localize. Immediately after including random particles, the robot's hypothesized location often stays at the original location for a short period of time for the baseline. This is because randomly posed particles are introduced at a moving ratio of the entire set of particles. As a note: the distance from the actual stays at the same value so long in this run because the teleported robot and the previously most likely location are moving parallel to one another.

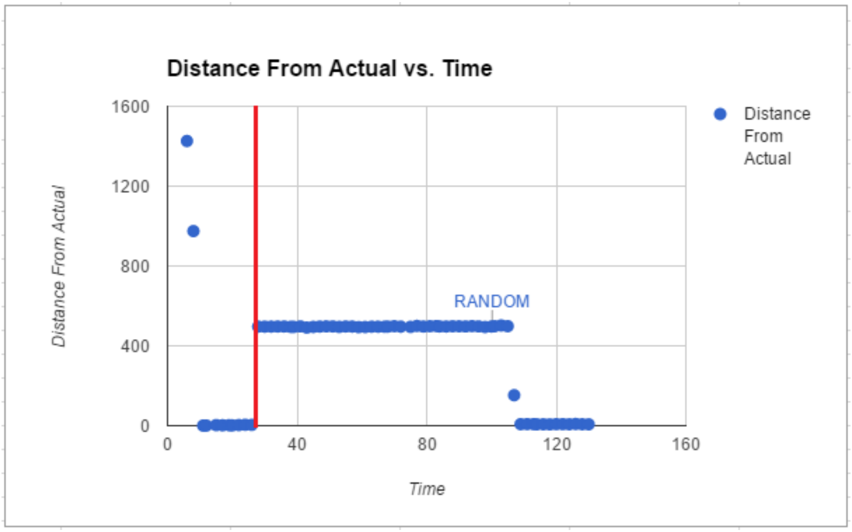


Figure 5.3: Time taken for baseline MCL to localize.

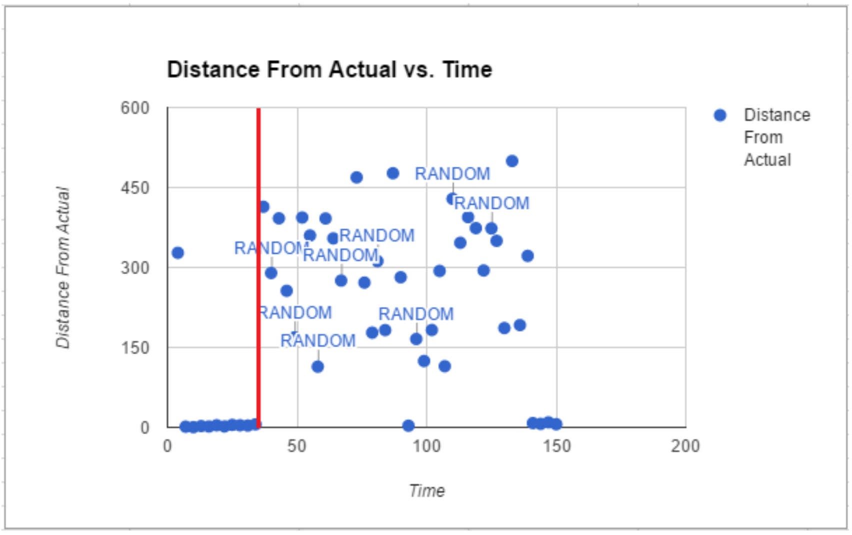


Figure 5.4: 2.5 x 1.25m areas, 8 orientations, 9 bins (Inconsistent)

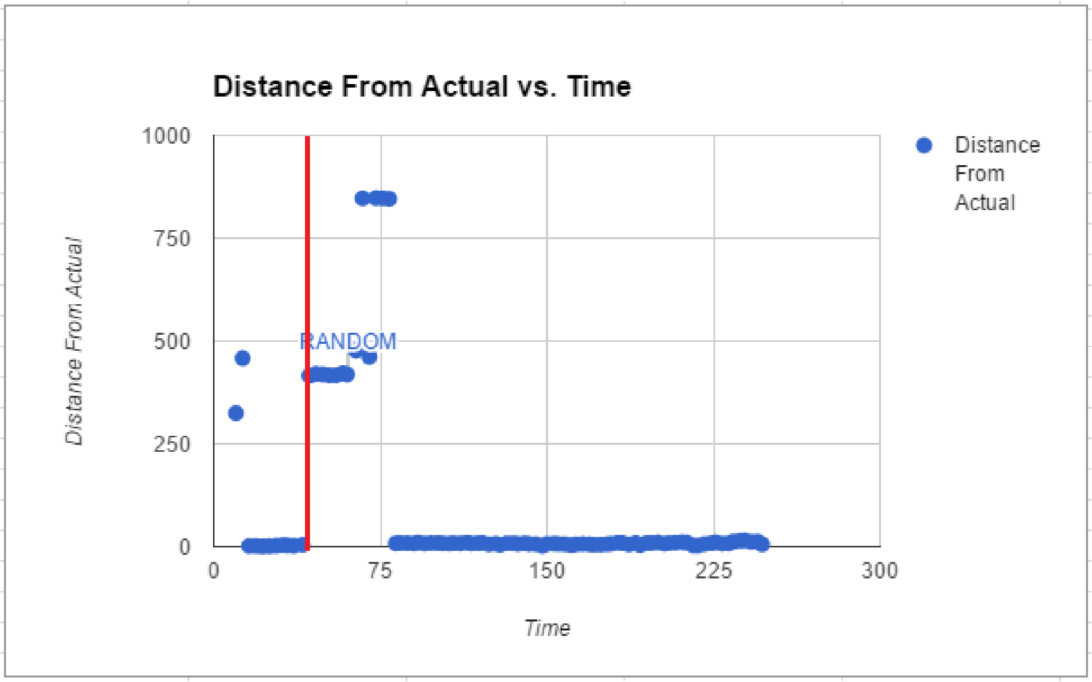


Figure 5.5: 1.25 x 1.25m areas, 4 orientations, 9 bins

Figure 5.4 shows one example of inconsistent localization. By the end of this particular run the robot eventually localizes, but takes a comparatively long time of mixing in random particles and is not guaranteed to actually localize successfully for a long period of time. This is discussed further in Section 5.4.

Figure 5.5 shows one example of a working localization with the HOG modifications to the algorithm. The robot is teleported at roughly 40 seconds into simulation, and re-localizes at just over 80. This run is approximately twice as fast as the baseline. The spike in distance right around 75 seconds shows that a group of particles farther away briefly had a higher weight than the real position. As the robot continued to move, the resampling of particles favored the random particles that appeared where the robot had teleported to, as the distance measurements there matched up much more closely. Marginally better and worse runs can be found in Appendix A, which also contains the most minimal but still successful memory usage configuration in Figure A.12.

5.4 Analysis of Results

Up to now, the term *successful*, *inconsistent*, and *failed* have been mentioned without strict definitions. These will now be defined as follows:

Successful: If the mean of the distance between the robot's actual location and the most likely location (based on particle weight) is below 30cm for 60 seconds, the localization was performed successfully. Note: this 60 second window of deciding if the localization was performed successfully is not included in the localization speeds in this section. These speeds record the time taken for particles to converge to the correct location. The 60 second measure is only meant to verify that more particles are not generated incorrectly after convergence.

Inconsistent: If the robot generates random particles after it successfully localizes *or* continuously generates random particles (showing that it at least knows it got off-track), this is labeled inconsistent.

Failed: If the robot never recognizes that it was teleported, then the localization completely failed.

Modifying MCL with computer vision did improve the localization speed over the baseline. Table 5.1 displays the time taken to localize for the baseline and every other variation of parameters. Times marked with an asterisk denote inconsistent localization, as the robot may have localized correctly once but loses its position soon after. A simpler look can be found in Figures 5.6 and 5.7. In Figure 5.6, the bin size and orientations are fixed to 9 and 4 respectively, and the different numbers of areas tested are varied along the X axis (with the baseline on the left). On the Y axis, the time to localize is recorded. Empty columns signify a failed or inconsistent localization. In Figure 5.7, all varied bin sizes and orientations are listed for the 30

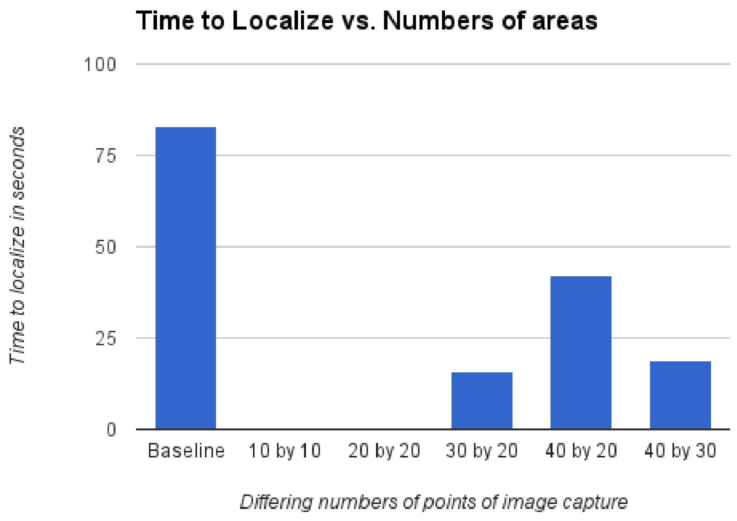


Figure 5.6: Varying areas for 4 orientations, 9 bins.

by 20 points of image caching. The Y axis similarly records the time to localize. The most notable results for each varied parameter are listed as bullet points below.

- Using either 9 or 18 bins did not seem to affect localization speed or success rate, but below 9 bins led to inconsistent localization.
- The number of points at which the camera captured images had to be at least more than 30 on the X and 20 on the Y. Everything at or above this localized successfully. Values below this localized inconsistently or completely failed.
- Using either 4 or 8 orientations did not seem to affect localization speed or success rate. Using only 2 orientations failed to localize.

In regards to the first bullet about histogram bins: the inconsistent localization below 9 bins is due to the difference in the captured histogram and the cached histogram not being consistent enough to find a good threshold. That threshold was determined empirically, and ranged from 100 to 150 for 9 bins and 200 to 250 for 18 bins. Chapter 5.2 covered this in more detail.

Table 5.1: Time taken to localize for various parameters

Baseline			83 seconds
10 by 10	4 orientations	9 bins	32 seconds*
10 by 10	4 orientations	18 bins	FAILED
10 by 10	8 orientations	9 bins	FAILED
10 by 10	8 orientations	18 bins	FAILED
20 by 20	4 orientations	9 bins	FAILED
20 by 20	4 orientations	18 bins	FAILED
20 by 20	8 orientations	9 bins	107 seconds*
20 by 20	8 orientations	18 bins	48 seconds*
30 by 20	4 orientations	5 bins	FAILED
30 by 20	4 orientations	7 bins	55 seconds*
30 by 20	4 orientations	9 bins	16 seconds
30 by 20	4 orientations	18 bins	31 seconds
30 by 20	8 orientations	9 bins	56 seconds
30 by 20	8 orientations	18 bins	27 seconds
40 by 20	4 orientations	9 bins	42 seconds
40 by 20	4 orientations	18 bins	35 seconds
40 by 20	8 orientations	9 bins	62 seconds
40 by 20	8 orientations	18 bins	27 seconds
40 by 30	4 orientations	9 bins	19 seconds
40 by 30	4 orientations	18 bins	42 seconds
40 by 30	8 orientations	9 bins	35 seconds
40 by 30	8 orientations	18 bins	16 seconds

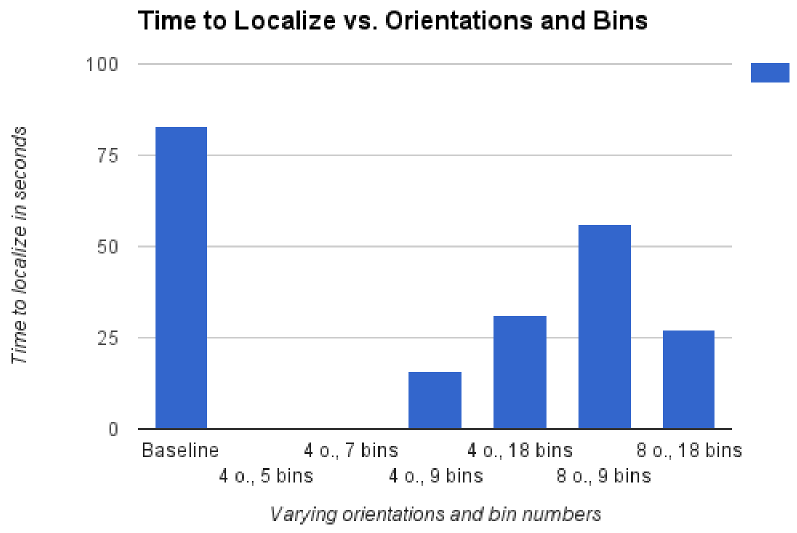


Figure 5.7: Varying orientations and bins for 30 by 20 areas.

The second bullet point regarding areas has quite a bit to do with the camera and simulated environment in this experiment. The camera has a range of 2 meters, and the 20 by 20 points has an area of 2.5 by 1.67 meters, which just barely exceeds the camera range. This means that a gap can exist between the image capture points. To test whether the hypothesis that the camera range directly impacted how many image capture points were needed, an additional test was run with 25 points on the X by 20 points on the Y, to provide a 2 by 1.67 meter area for each image capture using 9 histogram bins and 8 orientations. Figure 5.8 displays this data. The robot was indeed able to localize.

The reason that two orientations simply failed is that the robot will travel in one direction for quite some time, and then travel in a direction at a right angle to the first for quite some time in this mode of exploration. The wheel code only backed up to change direction when an object was too close to the front of the robot. Because the Cal Poly Computer Science offices have a fairly rectangular architecture, the robot will end up going in one direction, then backing up and moving perpendicular

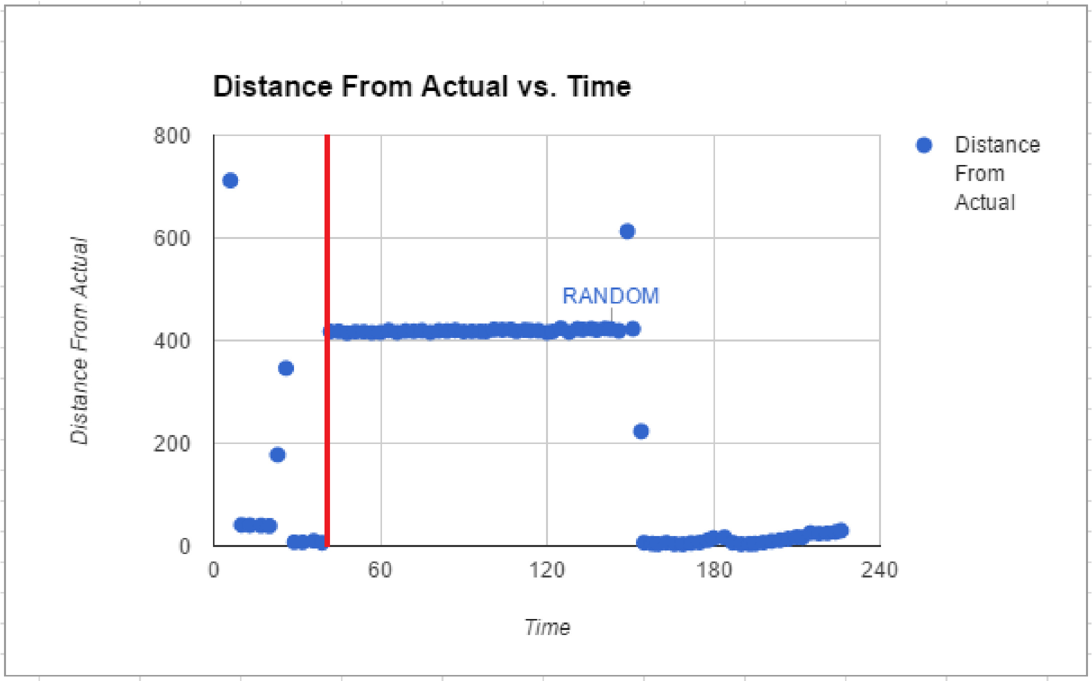


Figure 5.8: 2.0 x 1.67m areas, 8 orientations, 9 bins

to the first direction. Even if the first direction matches to a 2-orientation cached direction, the second will not. In such a rectangular environment, a 4-orientation cache is sufficient.

Table 5.1 showed that the modified version of MCL using histograms of oriented gradients does outperform the baseline in terms of pure runtime. To determine the number of algorithm iterations for both the baseline and the modified version to localize, 300 runs of each version were performed. Figure 5.9 shows how long it took each algorithm to localize in terms of purely algorithm iterations, to offset the additional `Sleep()` required by the simulated environment in the histograms of oriented gradients approach. The HOG approach uses only 12.96 of iterations on average to recover from the kidnapped robot problem, while the baseline averaged 84.36.

In summary, the modified MCL with greater than or equal to 30 by 20 points of image capture, greater than or equal to 4 orientations at each point, and at least 9

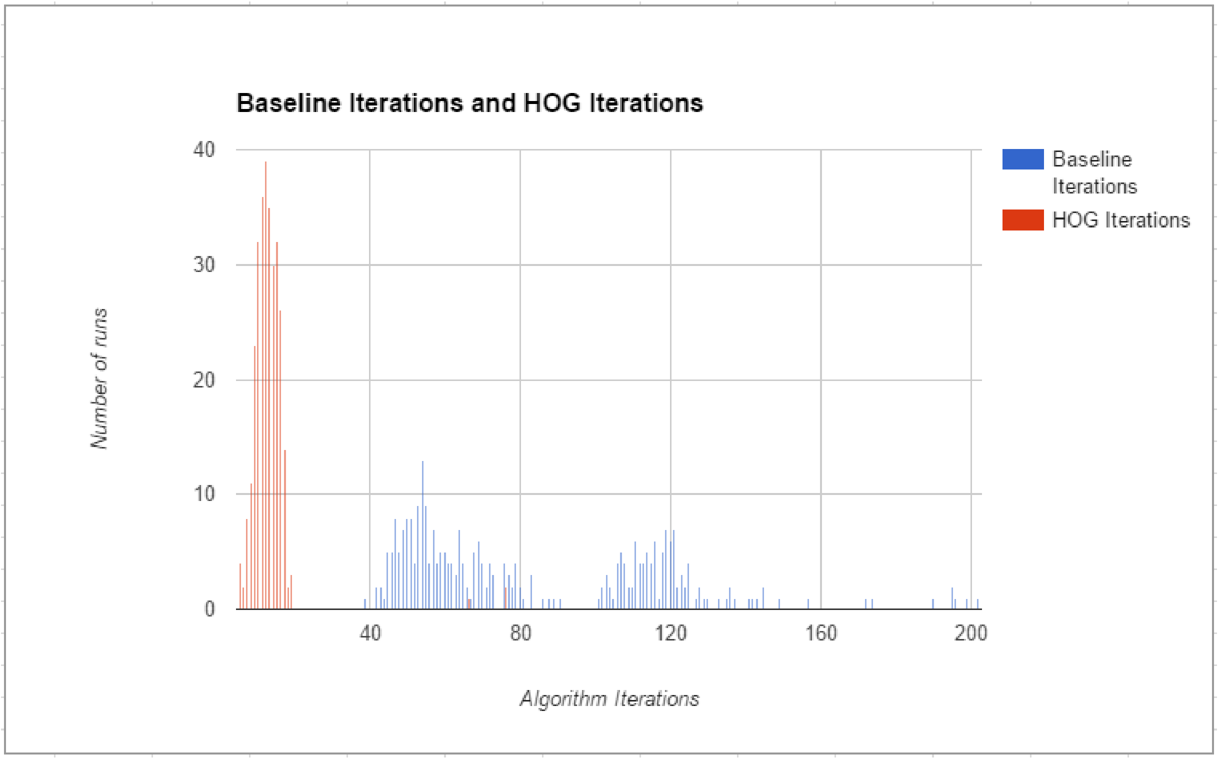


Figure 5.9: Algorithm iterations made by the HOG approach in green, and the baseline in blue.

bins performed from 25.3% faster (62 seconds) to 80.7% faster (16 seconds) than the baseline. This wide range is due in part to the random resampling portion of the algorithm. More likely particles based on ray tracing are indeed weighted heavier and are more likely to be selected, but Monte Carlo is still subject to a chance of selecting lower weight particles to survive.

Chapter 6

FUTURE WORK

While this work did compare against the baseline Monte Carlo algorithm and showed improvement, more analysis can be done when comparing to other localization algorithms with computer vision. Even though training a classifier for object detection takes some time, it would be worthwhile to compare this to the fast comparison approach in this work. The most relevant characteristics to study are how object detection affects the runtime of an iteration of the algorithm, memory usage, and if the overall speed of localization is quicker. The environment that this work was tested in had a large number of uniform structures (many pillars, fences, brick walls, etc.) which may inhibit an object detection approach. Other studies [14, 19, 29] seemed to use smaller environments with fairly distinctive objects. Performing object detection and differentiating between similar objects is a good objective for future work.

Of course, there is always room to incorporate additional sensors as discussed in Chapter 1. Localization is by no means a solved problem, and different types of sensor data may speed up the process or increase particle generation accuracy as in [23].

Chapter 7

CONCLUSION

This thesis presented a version of Monte Carlo Localization that incorporated image processing using histograms of oriented gradients into the localization process. Certain parameter configurations failed to localize while others outperformed the baseline MCL inside the V-REP simulator. The most minimal setup that still localized reliably required 30 by 20 cached locations, 4 orientations and 9 histogram bins. Fewer cached locations produced failed localizations to the wrong location, as well as using only 2 orientations and fewer than 9 histogram bins.

One of the limiting factors of using the remote API client with V-REP is the communication time mentioned in Chapter 4.2.2. The Windows `Sleep` call was made both when capturing an image (1 seconds) and when moving the camera (another 1 second) to generate the cached HOG map. This renders the cheap HOG computation fairly moot and the `Sleep` become the primary source of overhead. However, despite the 1 second image capture overhead when actually localizing, the aforementioned configurations with histograms of oriented gradients outperformed the baseline implementation. Implementing a Lua plugin inside the V-REP simulator to negate this need for a `Sleep` would return even quicker results.

The other limiting factor was the camera range. However, even with a camera range that could capture images all the way down the long hallways of the simulated environment, the histogram of oriented gradients produced on that image will not match what a robot captures partway down the hallway. A fair number of image points will still need to be taken to perform this localization strategy even with a long-distance camera.

Other work has used various forms of computer vision in localization, but these generally have to do with recognizing location based on the shape of certain objects. This work explored a simple, very cheap comparison of images instead. Hopefully this strategy can be used when developing future solutions.

BIBLIOGRAPHY

[1] Cal Poly Github. `http://www.github.com/CalPoly`.

[2] *IEEE standard for binary floating-point arithmetic.* Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[3] Coppelia robotics. `http://www.coppeliarobotics.com/`, 2016.

[4] gnuplot homepage. `http://www.gnuplot.info/`, 2016.

[5] Opencv (open source computer vision. `http://opencv.org/`, 2016.

[6] Qt. `https://www.qt.io/developers/`, 2016.

[7] T. authors of VLFeat. Basic hog computation, 2016.

[8] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Ann. Math. Statist.*, 29(2):610–611, 06 1958.

[9] J. Brauer. Hog descriptor computation and visualization, 2016.

[10] M. Brown and S. Süsstrunk. Multi-spectral sift for scene category recognition. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 177–184, June 2011.

[11] W. Burgard, D. Fox, D. Hennig, and T. Schmidt. Estimating the absolute position of a mobile robot using position probability grids. *Proc. of the Fourteenth National Conference on Artificial Intelligence*, pages 896–901, 1996.

[12] F. Cozman and E. Krotkov. Robot localization using a computer vision sextant. 1:106–111, May 1995.

[13] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[14] P. Elinas and J. J. Little. σMCL: Monte-Carlo localization for mobile robots with stereo vision. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.

[15] A. Fierro-Radilla, K. Perez-Daniel, M. Nakano-Miyatakea, H. Perez-Meana, and J. Benois-Pineau. An effective visual descriptor based on color and shape features for image retrieval. In *Human-Inspired Computing and Its Applications*, volume 8856, pages 336–348. Springer, 2014.

[16] D. Fox. Adapting the sample size in particle filters through kld-sampling. *International Journal of Robotics Research*, 22:2003, 2003.

[17] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *IN PROC. OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (AAAI*, pages 343–349, 1999.

[18] D. Fox, W. Burgard, and S. Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.

[19] A. Gil, Ó. Reinoso, M. A. Vicente, C. F. Peris, and L. Payá. Monte carlo localization using SIFT features. In *Pattern Recognition and Image Analysis, Second Iberian Conference, IbPRIA 2005, Estoril, Portugal, June 7-9, 2005, Proceedings, Part I*, pages 623–630, 2005.

[20] T. Goedemé, M. Nuttin, T. Tuytelaars, L. V. Gool, K. U. Leuven, K. U. Leuven, and E. T. H. Zürich. Markerless computer vision based localization using automatically generated topological maps. 2004.

[21] J. S. Gutmann and D. Fox. An experimental comparison of localization methods continued. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 1, pages 454–459 vol.1, 2002.

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

[23] S. Krapil. Adapting Monte Carlo Localization to Utilize Floor and Wall Texture Data. Master's thesis, California Polytechnic State University, San Luis Obispo, 2014.

[24] R. Kummerle, R. Triebel, P. Pfaf, and W. Burgard. Monte carlo localization in outdoor terrains using multilevel surface maps. *Journal of Field Robotics*, 25:346–359, June 2008.

[25] G. Lucas. A tutorial and elementary trajectory model for the differential steering system of robot wheel actuators. `http://rossum.sourceforge.net/papers/DiffSteer/#d3`, 2001.

[26] C. McCormick. Hog person detector tutorial. May 2013.

[27] A. Milstein. Dynamic maps in monte carlo localization. In *Advances in Artificial Intelligence*, pages 1–12. Springer, 2005.

[28] Mori, Maeda, and Yamamoto. Scanning laser range finder URG-04LX-UG01 (Simple-URG), 2009.

[29] J. Röwekämper, C. Sprunk, G. D. Tipaldi, C. Stachniss, P. Pfaff, and W. Burgard. On the position accuracy of mobile robot localization based on particle filters combined with scan matching. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3158–3164, 2012.

[30] M. P. Schlachtman. Using Monocular Vision and Image Correlation to Accomplish Autonomous Localization. Master's thesis, California Polytechnic State University, San Luis Obispo, 2010.

[31] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainly. *Int. J. Rob. Res.*, 5(4):56–68, Dec. 1986.

[32] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics.* MIT press, 2005.

[33] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Monte carlo localization for mobile robots. *Robotics and Automation*, 2:1322 – 1328, May 1999.

[34] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust monte carlo localization for mobile robots. *Artificial Intelligence*, 128(1-2):99–141, May 2001.

[35] S. Zickler and M. Veloso. Rss-based relative localization and tethering for moving robots in unknown environments. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 5466–5471. IEEE, May 2010.

Appendix A

GRAPHS WITH VARIED PARAMETERS

The graphs below show the distance from the most likely location of the robot (highest weighted particle) to the actual location vs time in seconds. When the Y value is low for a long time, the robot has a good estimate of where it is, as the highest weight particle matches the true position very closely. The vertical red lines delineate when the robot was teleported to another location. Varied parameters include the area of the grid that the camera images were taken at, the number of orientations that the camera pointed in, and the number of bins in each histogram. Figures tagged with (Inconsistent) shows that the incorporation of random particles did not always relocalize successfully. Figures tagged with (Failed) mean that the histogram difference threshold was high and no random particles could be added without adding at the wrong time.

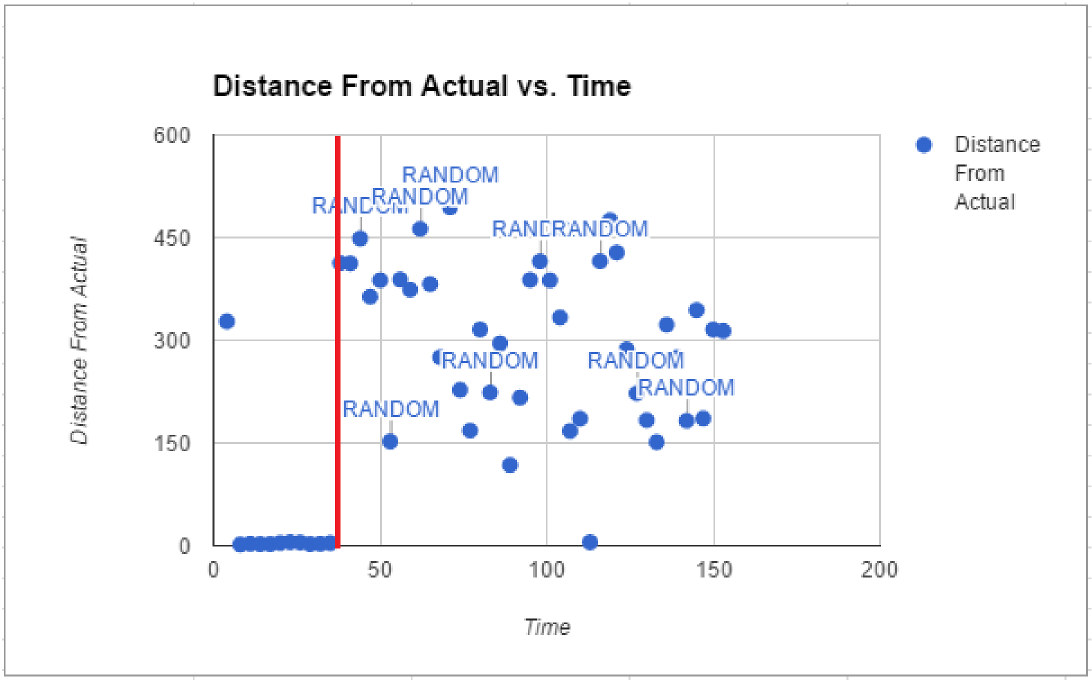


Figure A.1: 5 x 2.5m areas, 4 orientations, 18 bins (Inconsistent)

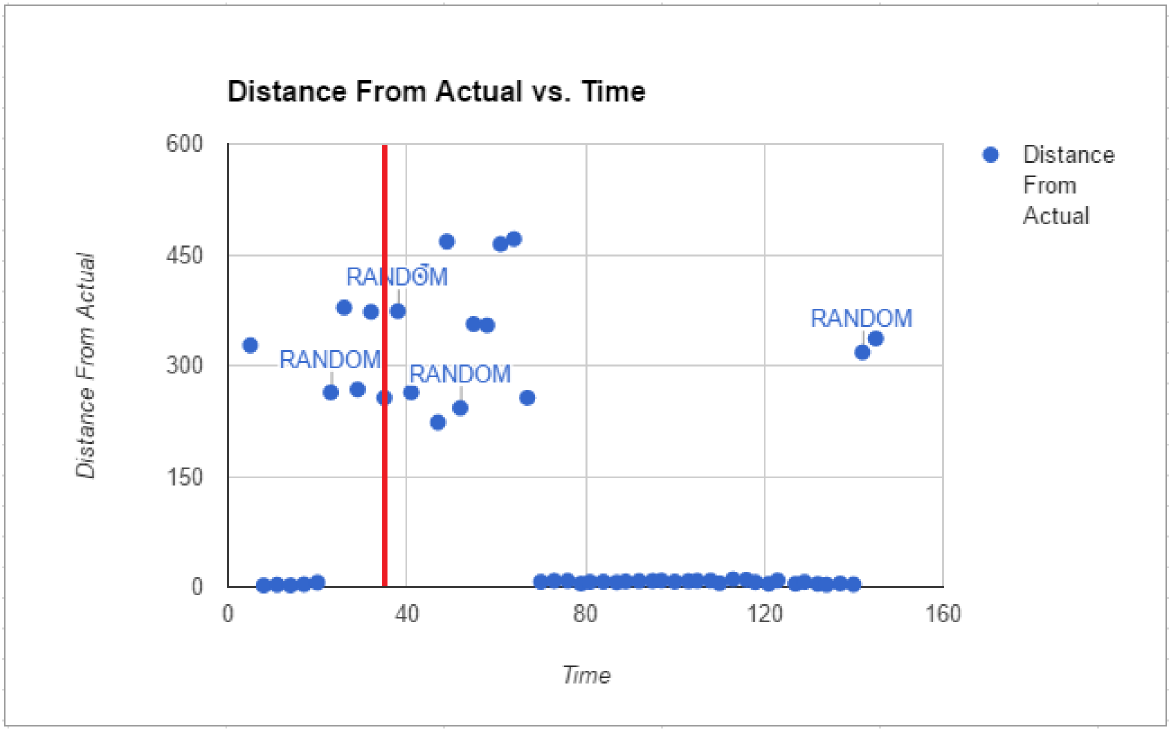


Figure A.2: 5 x 2.5m areas, 4 orientations, 9 bins (Inconsistent)

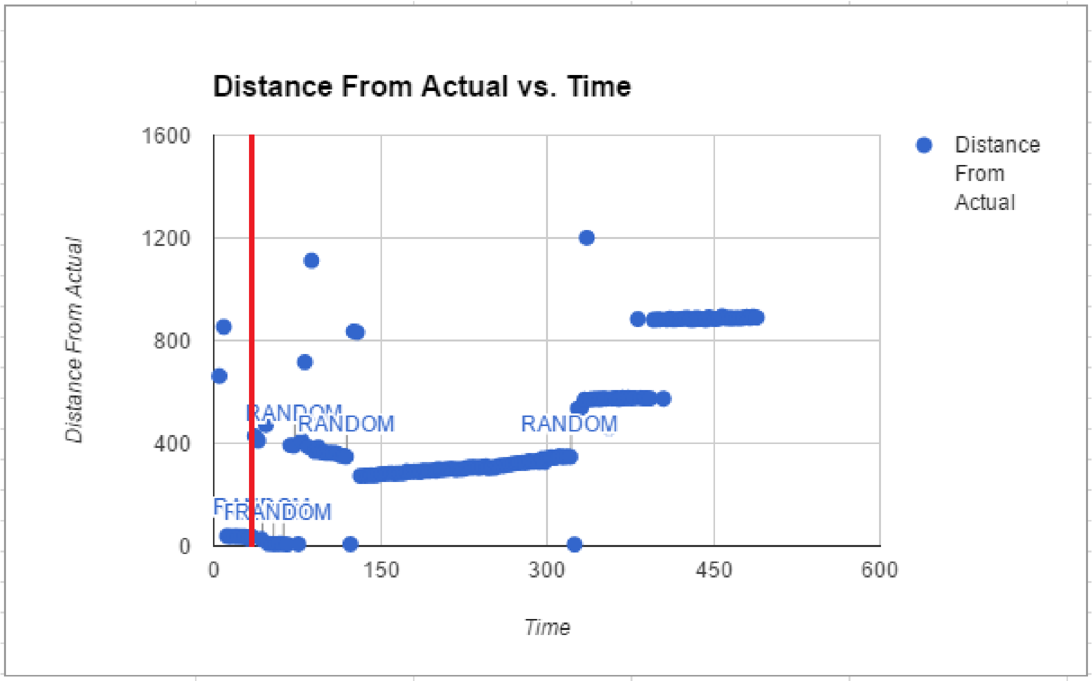


Figure A.3: 5 x 2.5m areas, 8 orientations, 18 bins (Inconsistent)

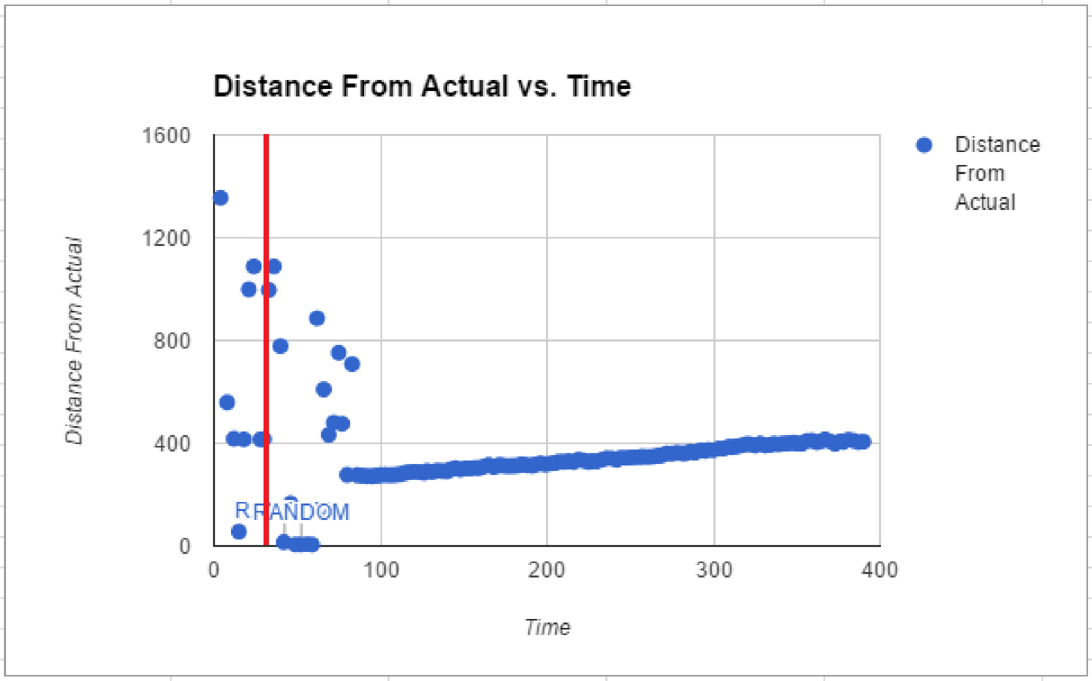


Figure A.4: 5 x 2.5m areas, 8 orientations, 9 bins (Inconsistent)

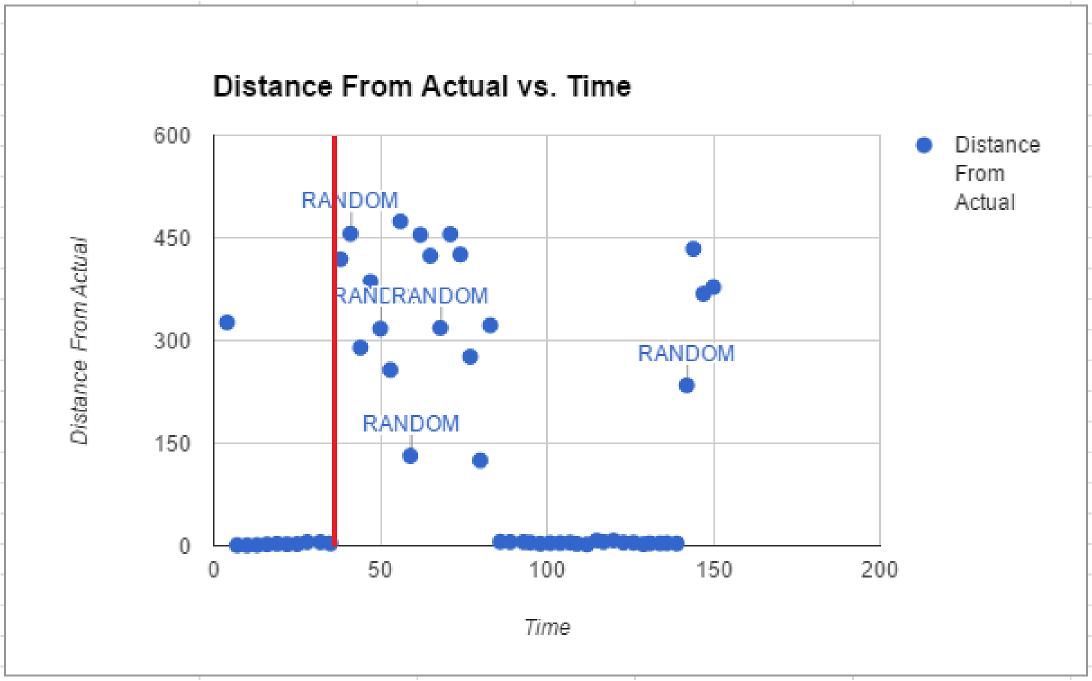


Figure A.5: 2.5 x 1.25m areas, 4 orientations, 18 bins (Inconsistent)

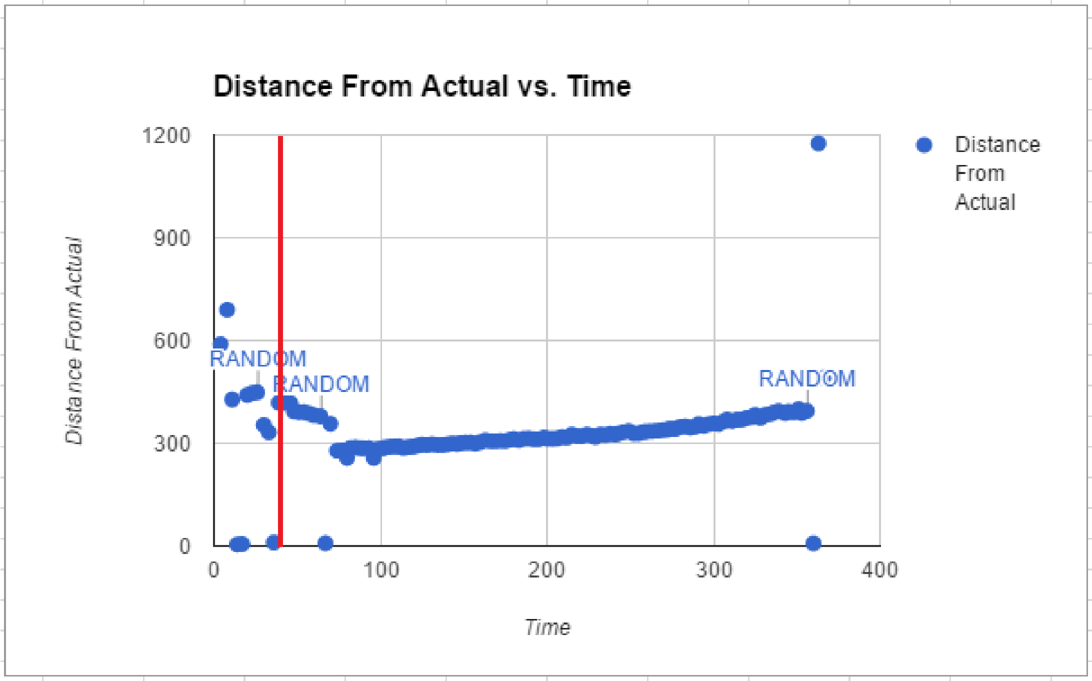


Figure A.6: 2.5 x 1.25m areas, 4 orientations, 9 bins (Inconsistent)

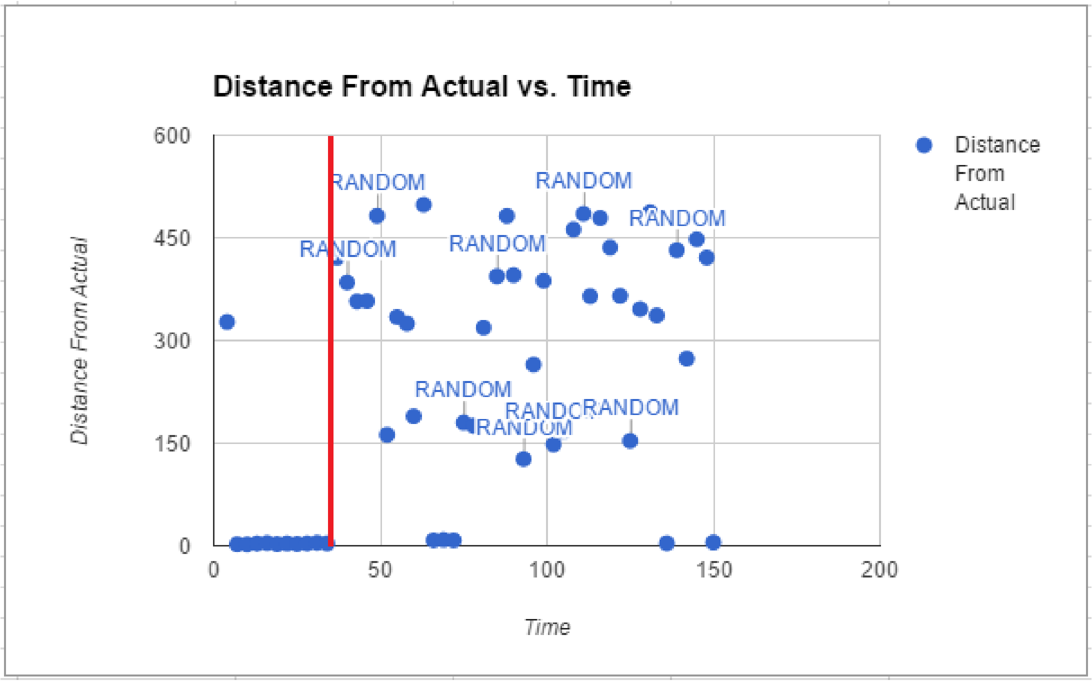


Figure A.7: 2.5 x 1.25m areas, 8 orientations, 18 bins (Inconsistent)

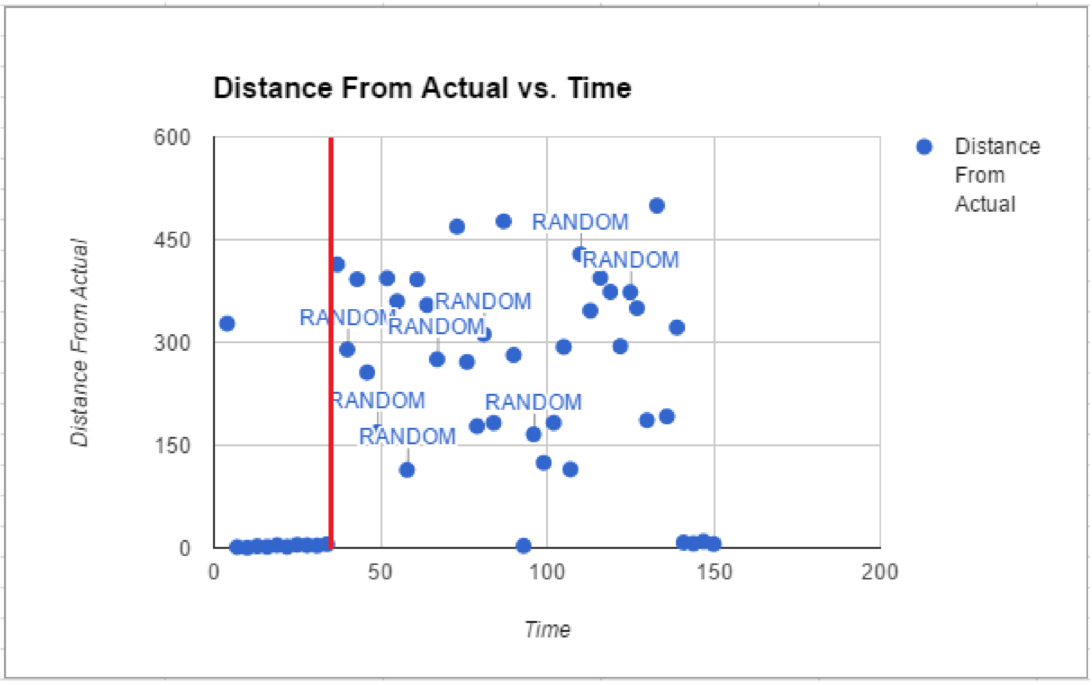


Figure A.8: 2.5 x 1.25m areas, 8 orientations, 9 bins (Inconsistent)

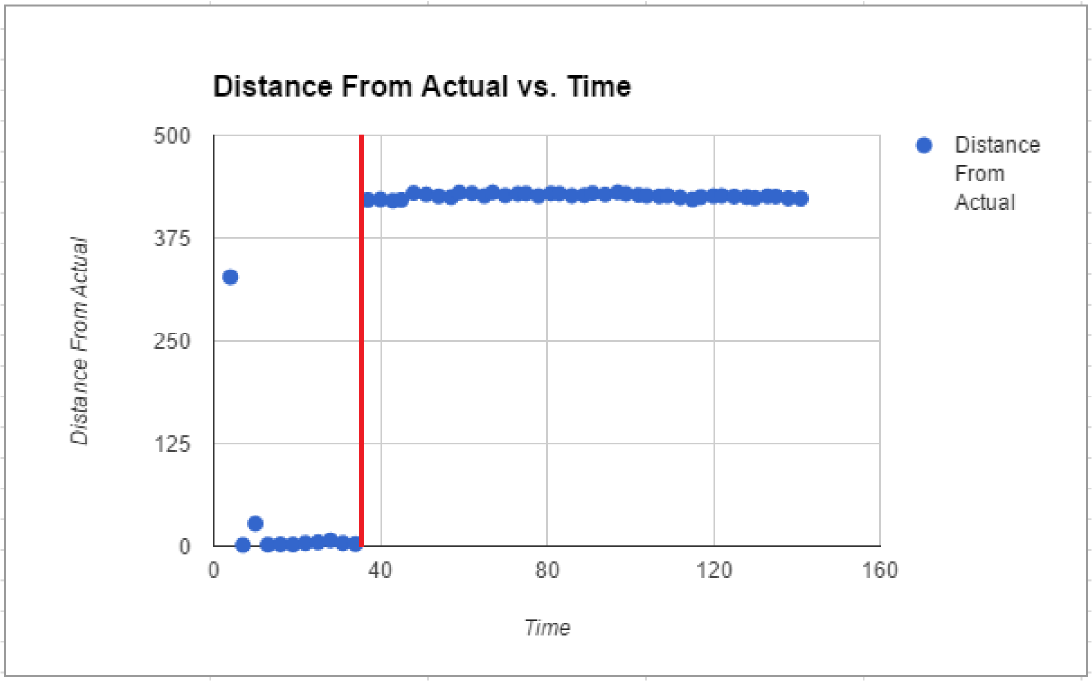


Figure A.9: 1.67 x 1.25m areas, 8 orientations, 5 bins (Failed)

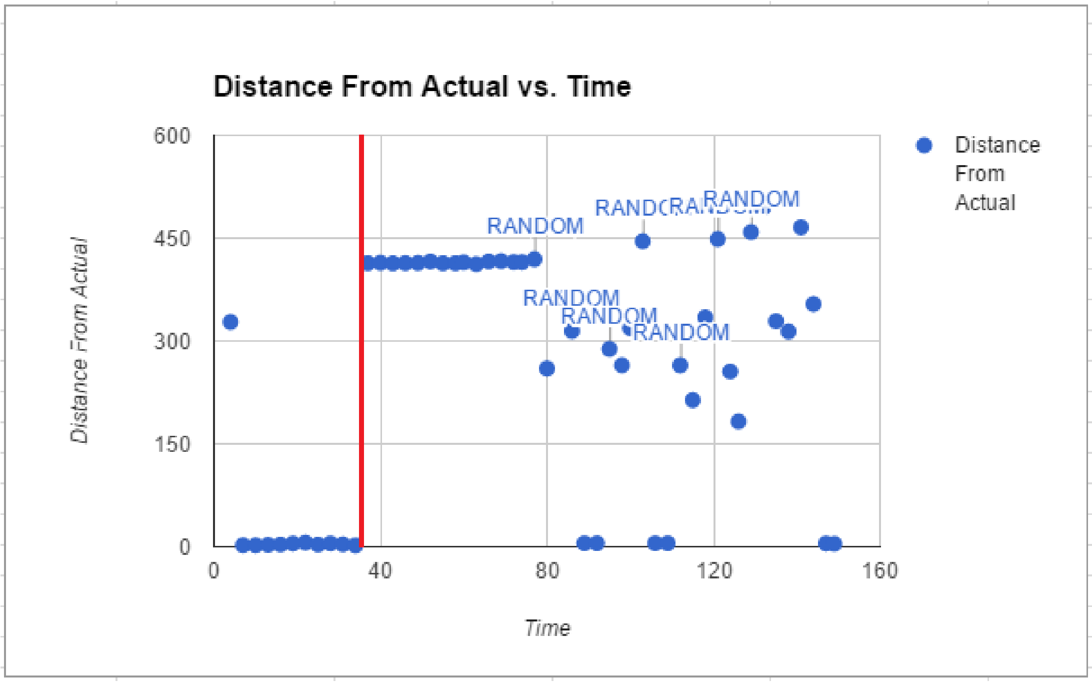


Figure A.10: 1.67 x 1.25m areas, 8 orientations, 7 bins (Inconsistent)

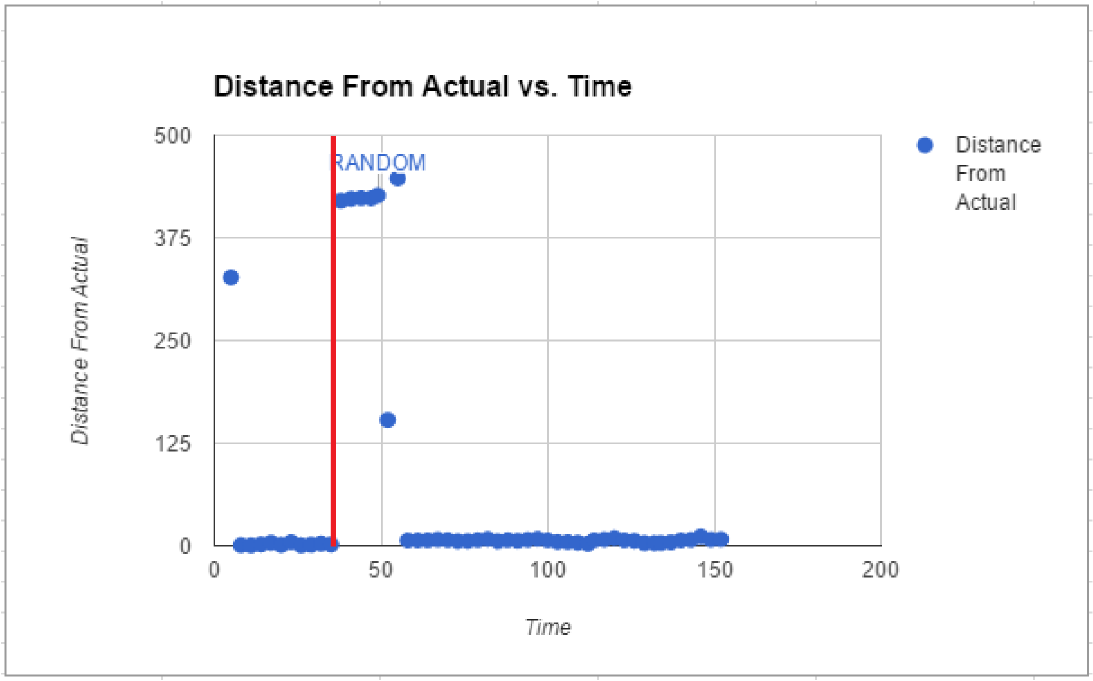


Figure A.11: 1.67 x 1.25m areas, 4 orientations, 18 bins

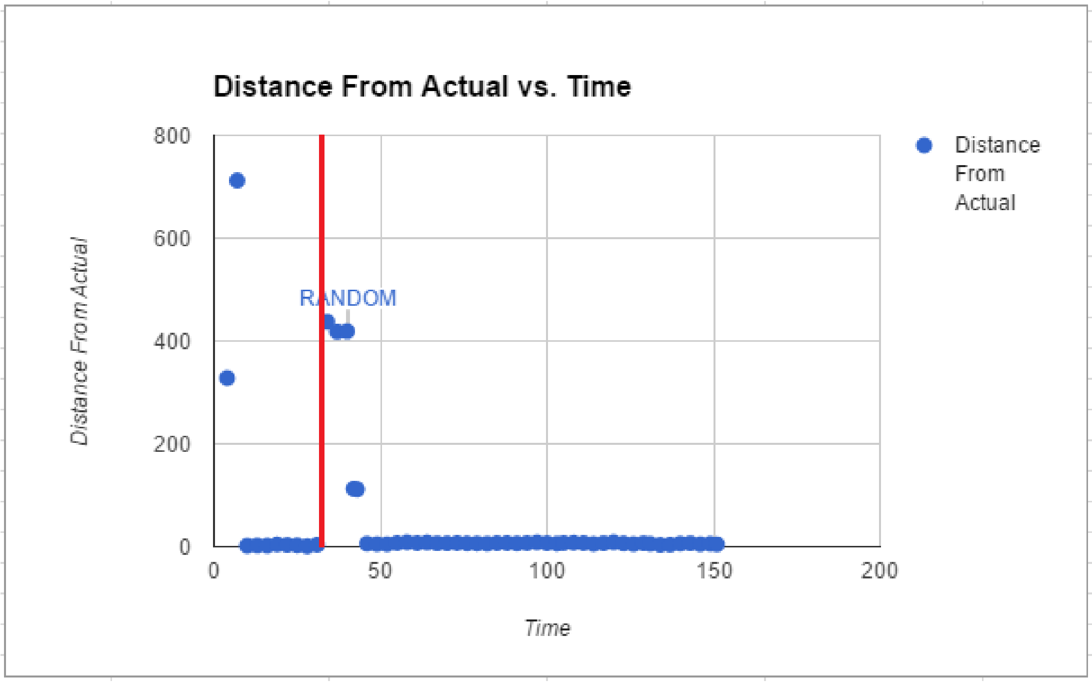


Figure A.12: 1.67 x 1.25m areas, 4 orientations, 9 bins

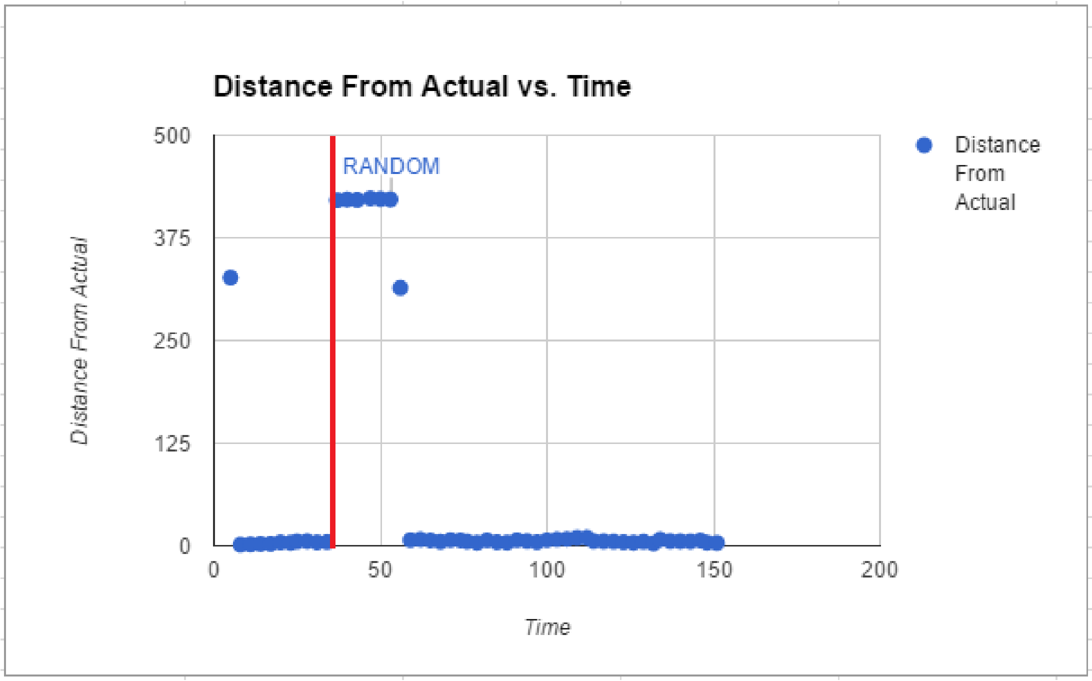


Figure A.13: 1.67 x 1.25m areas, 8 orientations, 18 bins

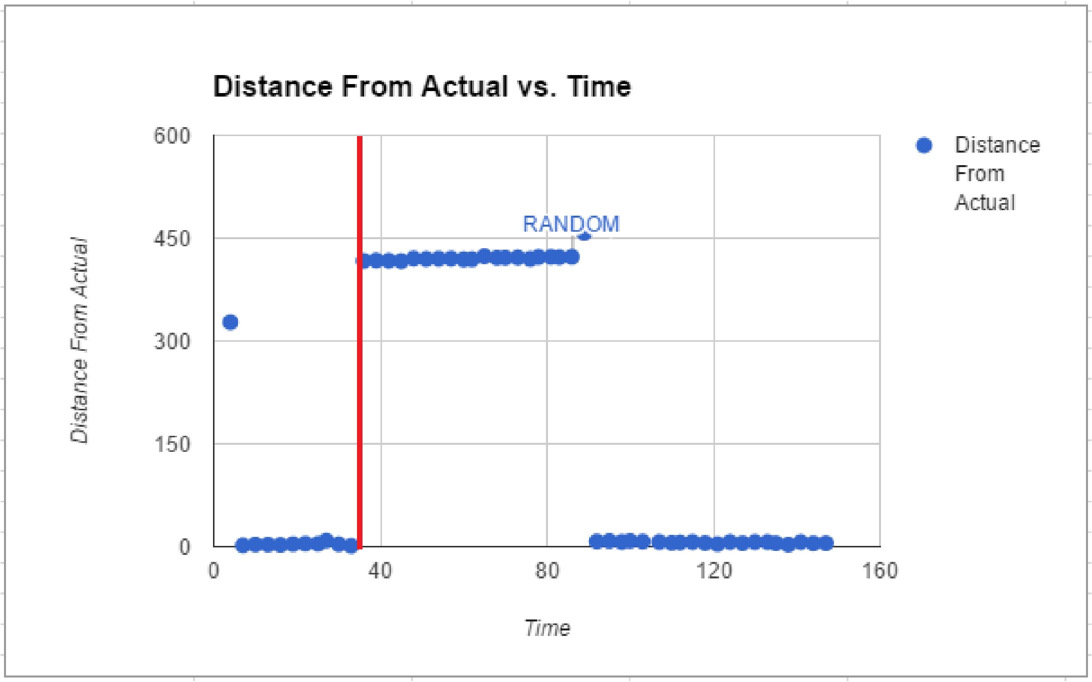


Figure A.14: 1.67 x 1.25m areas, 8 orientations, 9 bins

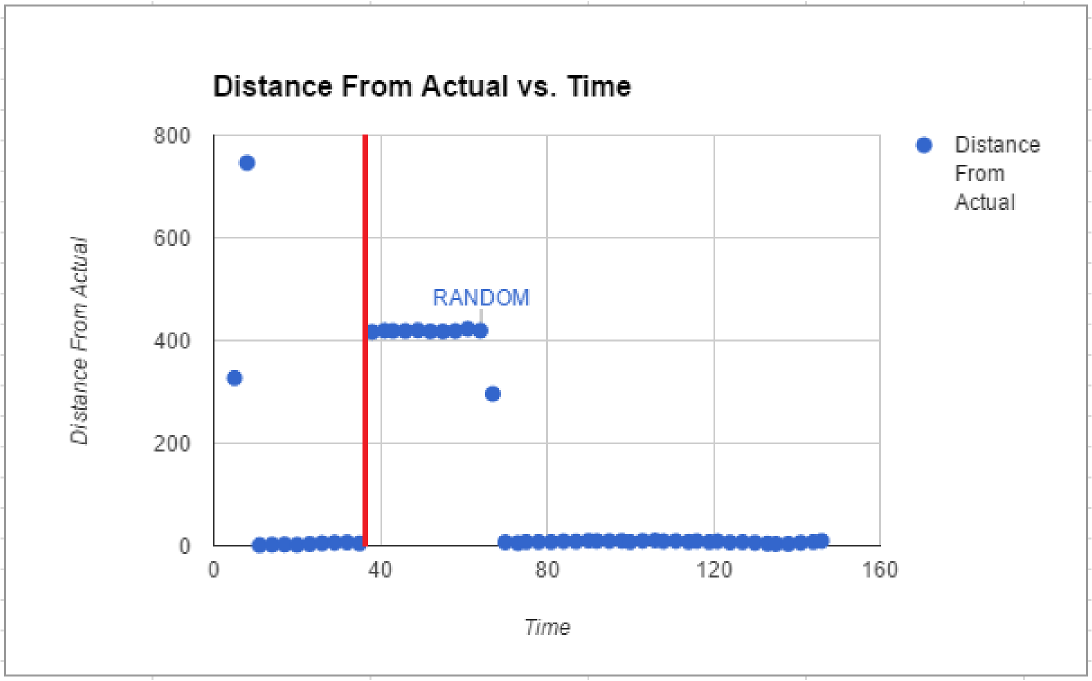


Figure A.15: 1.25 x 1.25m areas, 4 orientations, 18 bins

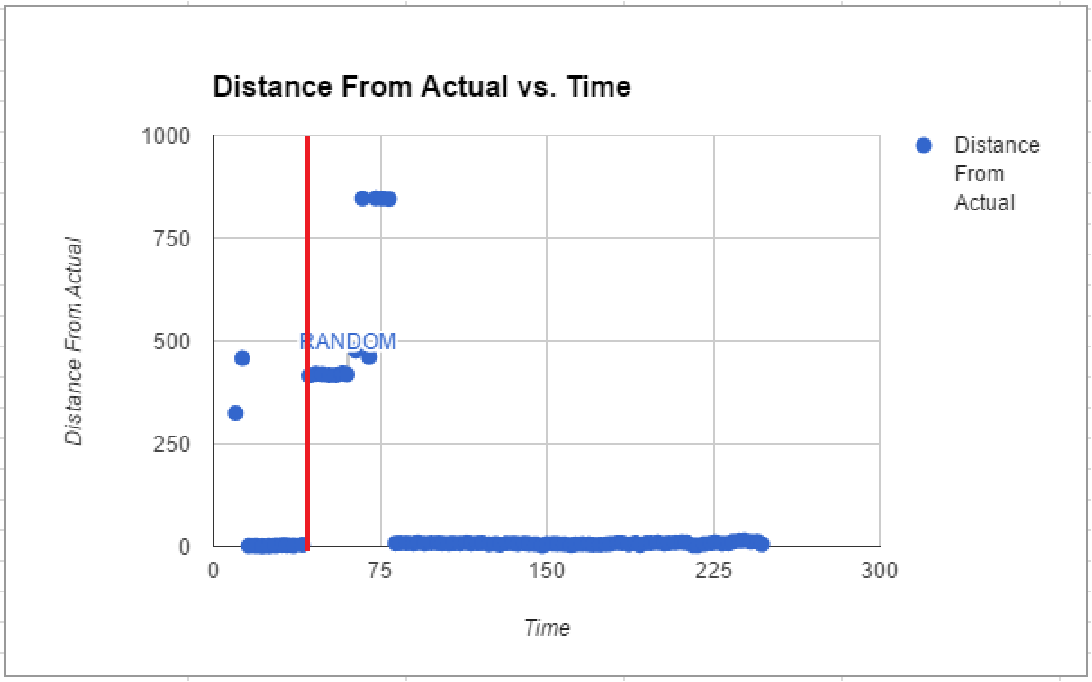


Figure A.16: 1.25 x 1.25m areas, 4 orientations, 9 bins

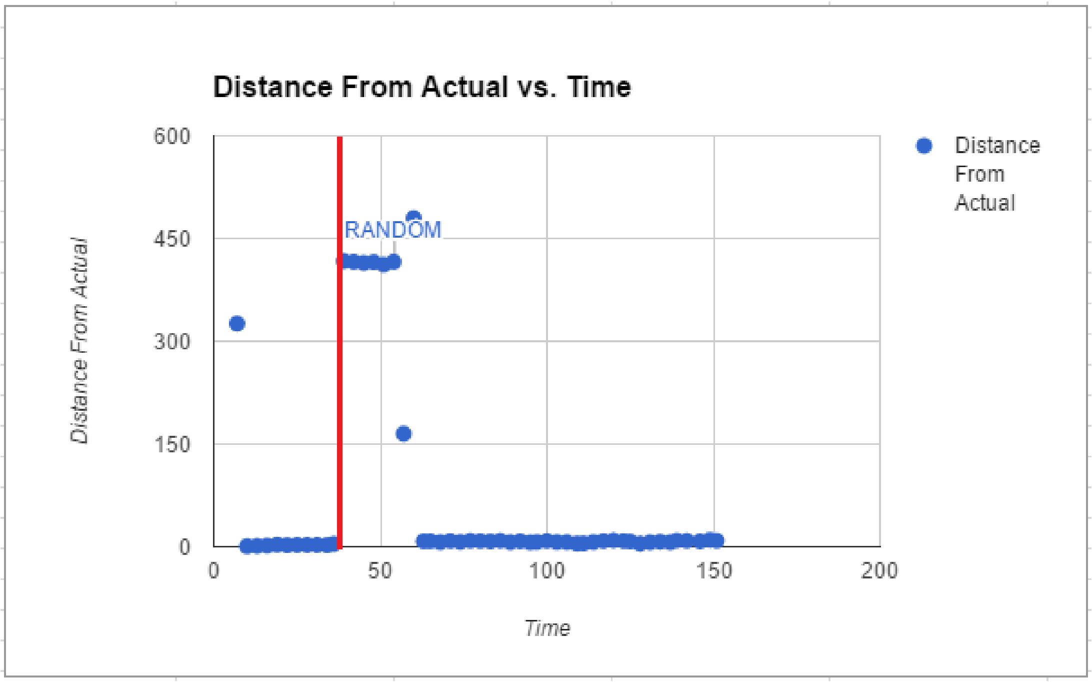


Figure A.17: 1.25 x 1.25m areas, 8 orientations, 18 bins

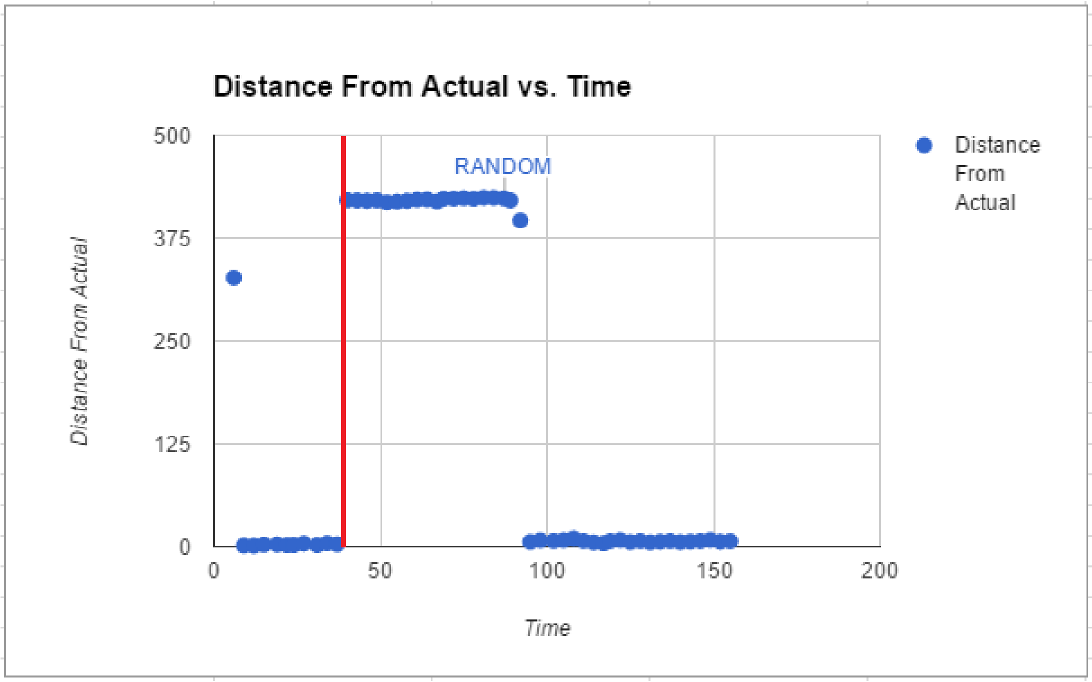


Figure A.18: 1.25 x 1.25m areas, 8 orientations, 9 bins

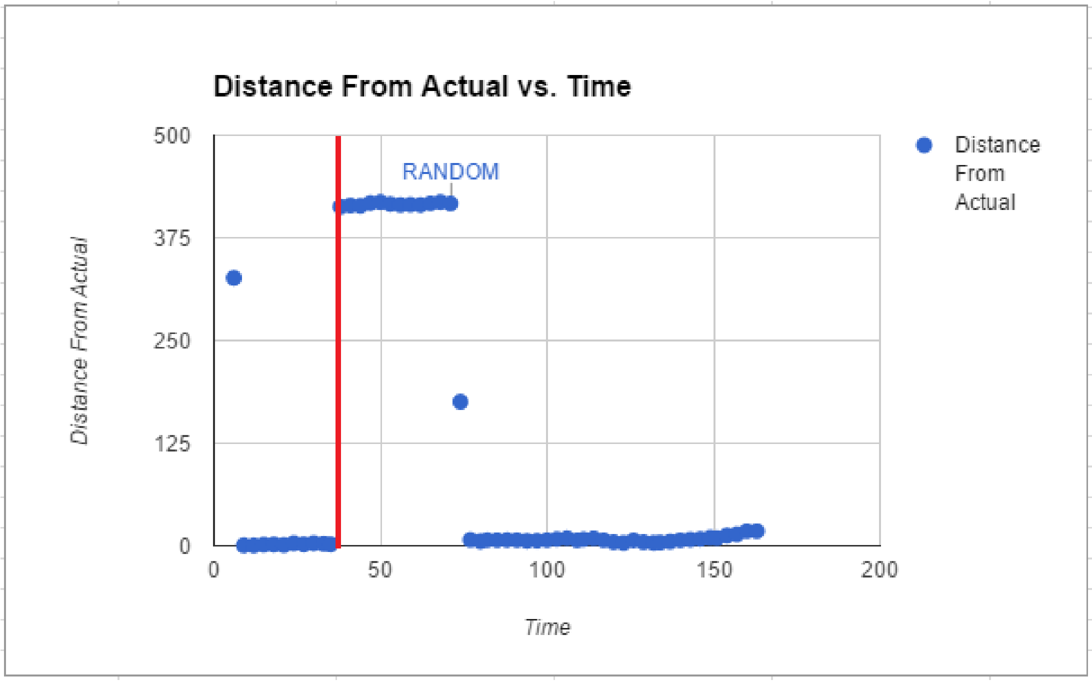


Figure A.19: 1.25 x 0.83m areas, 4 orientations, 18 bins

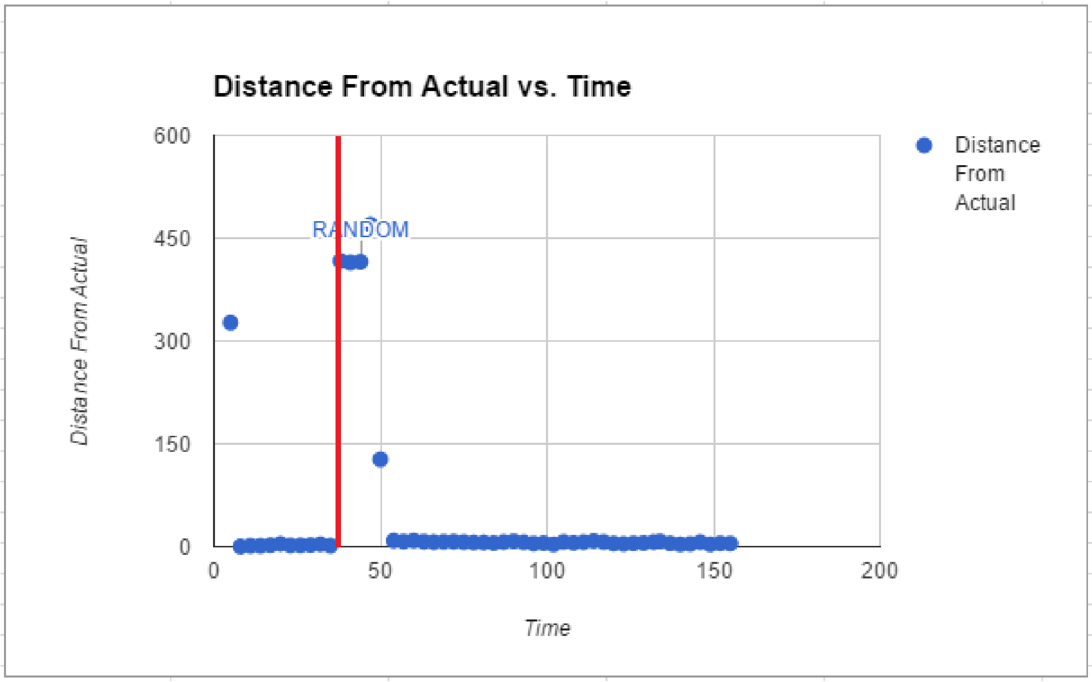


Figure A.20: 1.25 x 0.83m areas, 4 orientations, 9 bins

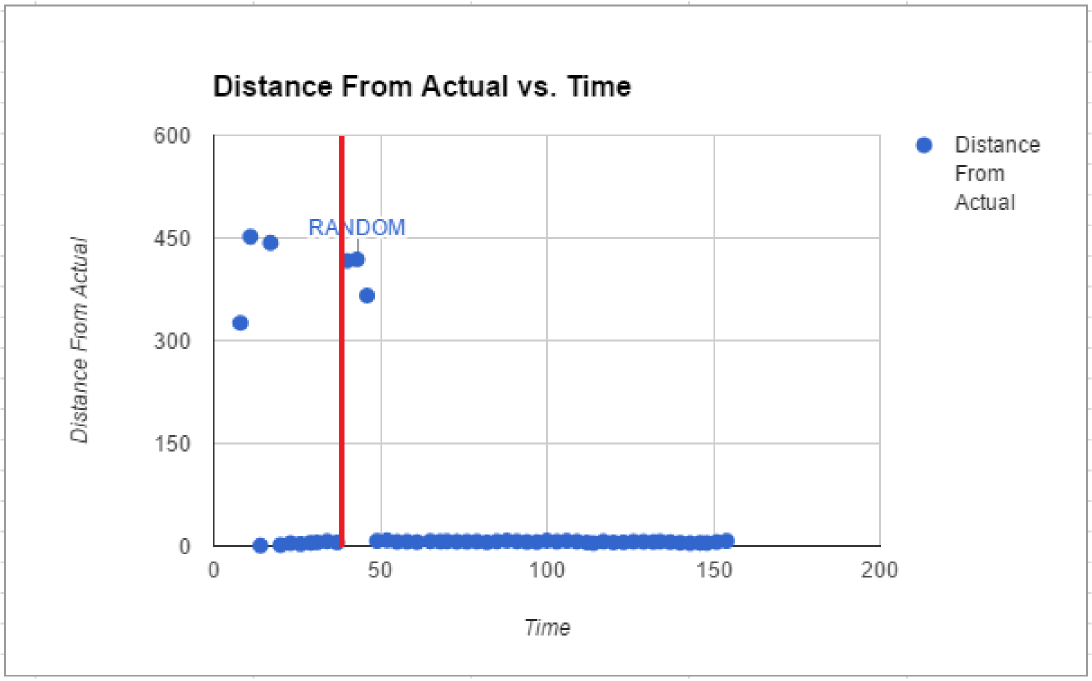


Figure A.21: 1.25 x 0.83m areas, 8 orientations, 18 bins

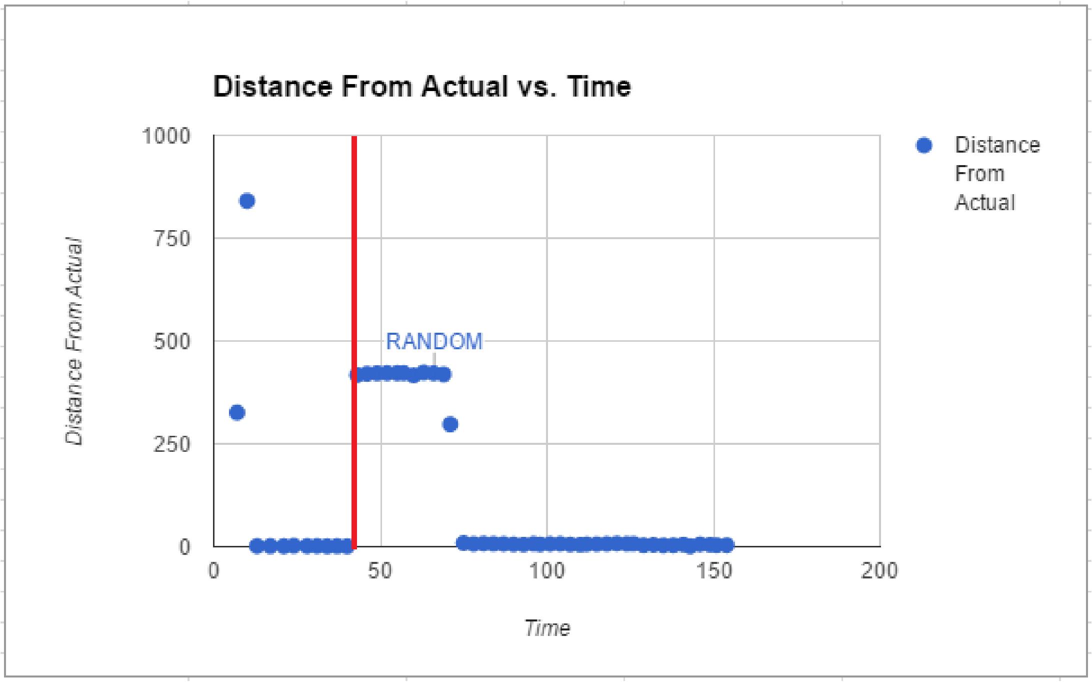


Figure A.22: 1.25 x 0.83m areas, 8 orientations, 9 bins

Appendix B

PLOTTING PARTICLES

This section shows three images of robot's internal bitmap of the environment that was also graphed in gnuplot. Figure B.1 simply shows the bitmap without particles, the most likely particle, or the robot's actual location. In B.2 and B.3, the robot's real position is marked with a purple X, the most likely location is marked as a red dot, and particles are marked in green. These were taken directly from gnuplot while running a simulation of the V-REP environment directed by the remote API client. B.3 shows the robot's true position farther away from the particles, as it was just kidnapped to another location for testing the kidnapped robot problem.

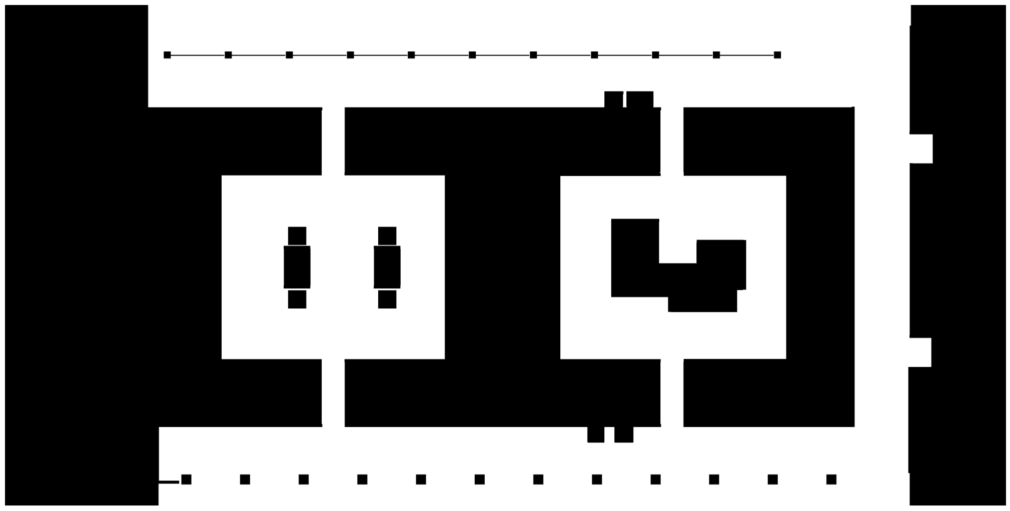


Figure B.1: The starting graph without any particles plotted.

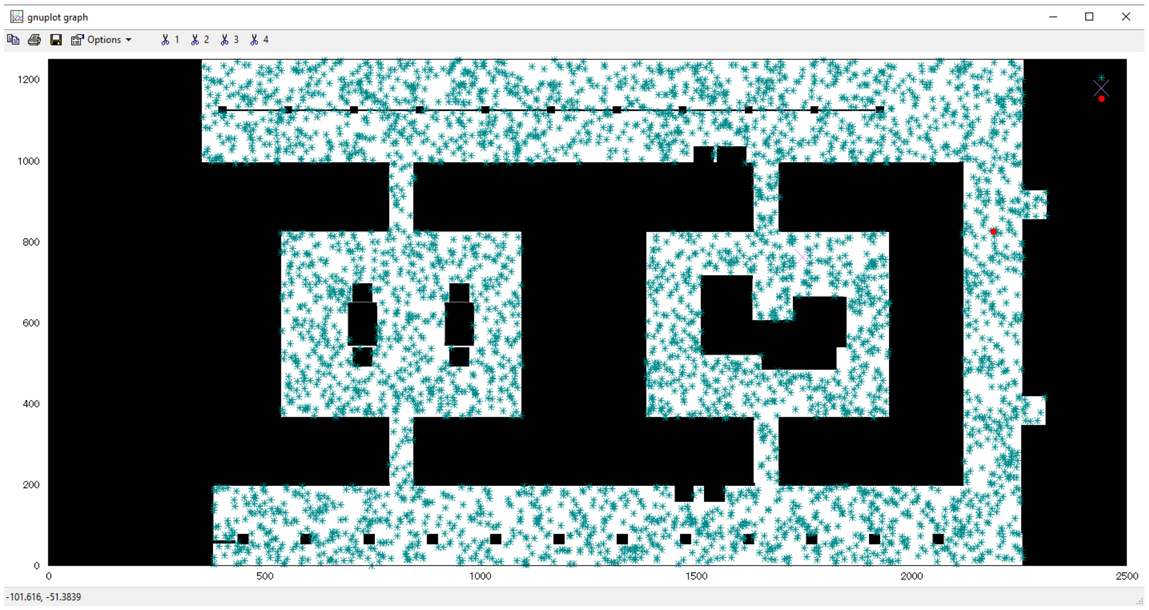


Figure B.2: Initial particles across the field.

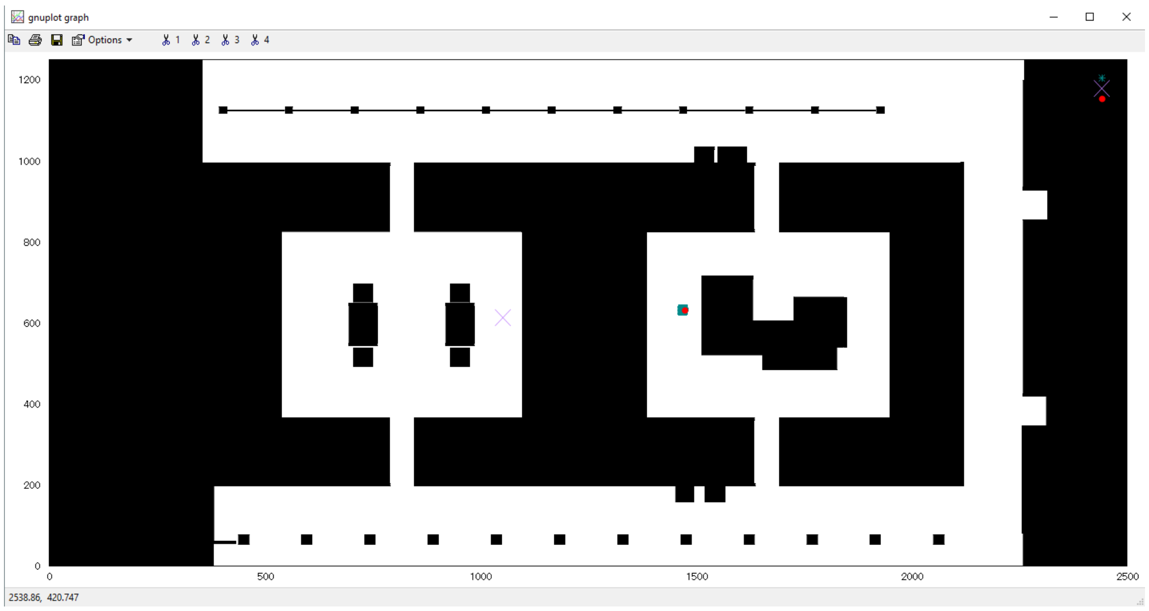


Figure B.3: Teleporting the robot after it has localized.

Appendix C

HISTOGRAM BIN COMPARISON

These figures compare the varying histogram bin sizes tested in this experiment. Each graph shows the difference between the cached histogram and the histogram directly in front of the robot versus time. The remote API client will use more memory linearly as histogram bins increase, so minimizing the number of bins used is advantageous. The 9 and 18 bin runs rise to a very noticeable difference (above 600) for a long period of time, clearly showing that the robot has been teleported. The 7 bin run is much less clear showing the teleportation, and the 5 bin run does not show the teleportation at all.

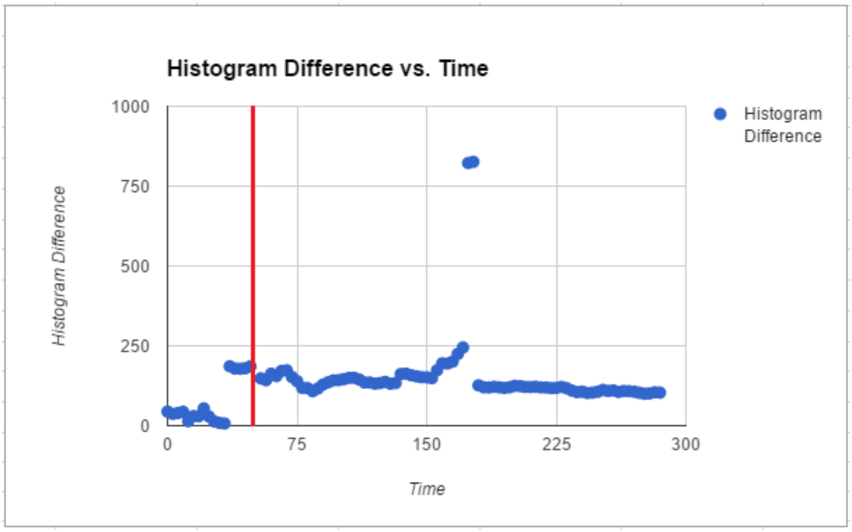


Figure C.1: Comparing the difference in output gradient vector of the captured image against the cached image with 5 bins.

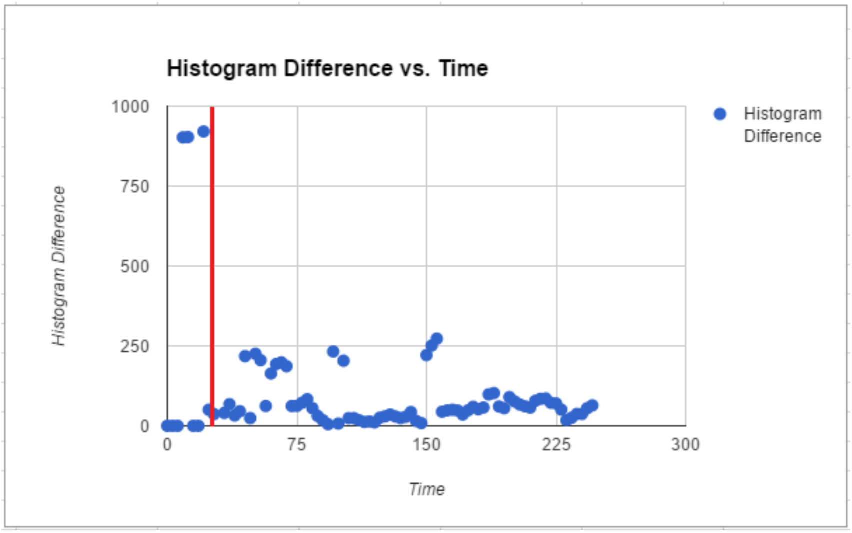


Figure C.2: Comparing the difference in output gradient vector of the captured image against the cached image with 7 bins.

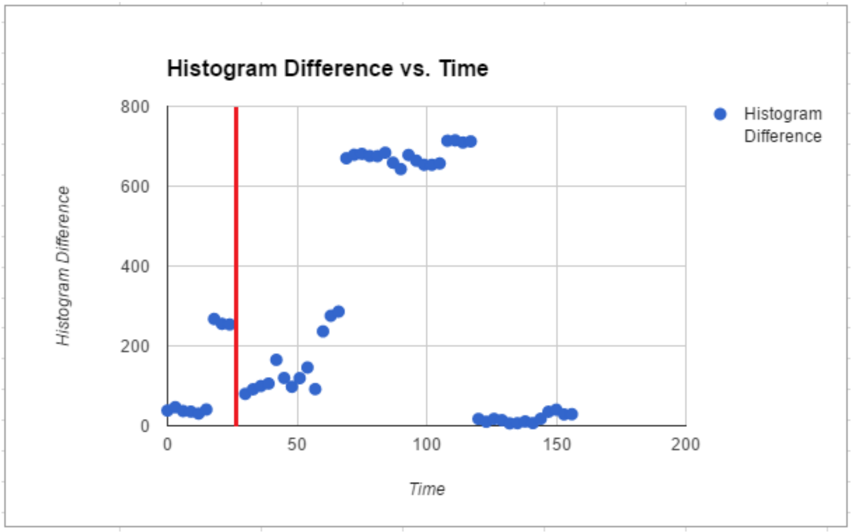


Figure C.3: Comparing the difference in output gradient vector of the captured image against the cached image with 9 bins.

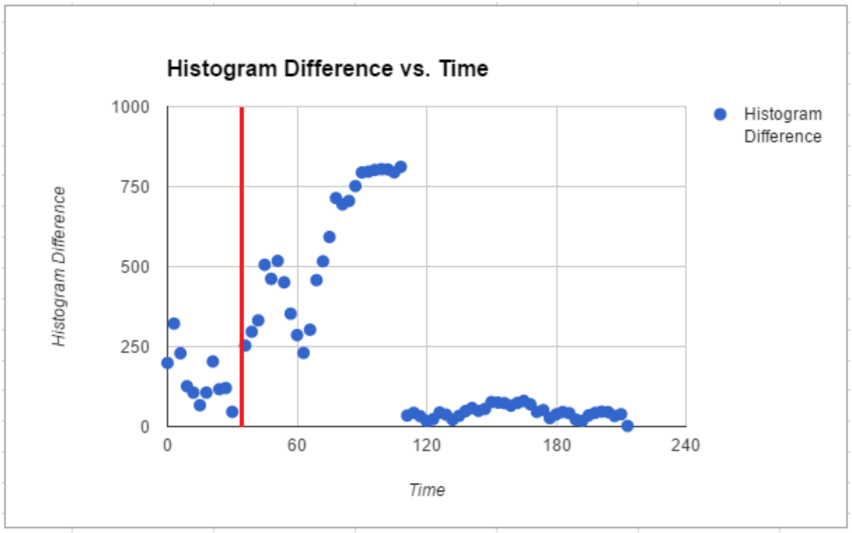


Figure C.4: Comparing the difference in output gradient vector of the captured image against the cached image with 18 bins.

Appendix D

GNUPLOT COMMANDS

These commands have been made available in this appendix because no examples were readily available online that performed the exact functionality desired at the time of writing. These may be useful for any future use of gnuplot for real-time graphing.

```
plot \"map.png\" binary filetype=png w rgbimage, '-'
using 1:2 with points pointtype 3 ps 1 lc rgb '#00008B8B', '-'
using 1:2 with points pointtype 2 ps 3 lc rgb 'purple', '-'
using 1:2 with points pointtype 82 ps 1 lc rgb 'red'\n
<p_x> <p_y>\n
e\n
<a_x> <a_y>\n
e\n
<m_x> <m_y>\n
e\n
```