Autonomous Localization Using 3D Models, Particle Filtering, and the Monte Carlo Localization Algorithm

John Allard, Alex Rich
2014 Summer Computer Science REU, Harvey Mudd College*

December 15, 2014

Contents

*Funded By The National Science Foundation and mentored by Professor Zachary Dodds

Abstract

Our research group attempts to implement the Monte Carlo Localization algorithm using a 3D model of an environment and a stream of images from a robot in that environment. This entails the use of various computer vision algorithms to find and compare features between the image-feed and our 3D model. The objectives of this project are to be able to localize a general actor in any 3D-modeled environment quickly and accurately. This paper outlines the process that our research group has undertaken to accomplish this task and an analysis of our resulting program.

1 Introduction

1.1 Project Overview

The Monte Carlo Localization (MCL) algorithm has been used in the past[1] to successfully localize robots using two dimensional maps of an environment and a stream of range sensor information that measures the distance to the nearest plane in front of the robot. Our goal was to build off of these successes and implement a similar algorithm using fully-textured 3D models of an environment and color images from the robot as sensor data. The end goal of this project was to be able to place a robot anywhere inside of a mapped environment, have our program quickly and reliably localize the robot, then continue to localize the robot as it moves around the environment while accomplishing its programmed goals (fetching coffee, cleaning the floor, etc).

We chose to implement the MCL algorithm using 3D models and a live image-feed from the robot because seems to be the general direction in which the field of robotics is heading. 3D models of an environment might be difficult to obtain, but they are constantly getting easier and cheaper to create. Our 3D map making camera was donated by a company called Matterport and was able to model a single floor of a large building in about an hour with little human involvement. We decided to use a camera instead of a range sensor because cameras are cheap, portable, and common and the information present in an image ought to be enough to localize. Localizing by vision is certainly something humans can do.

1.2 About

This project was undertaken by Alex Rich and John Allard during the Summer of 2014 at Harvey Mudd College, under the mentorship of Professor Zach Dodds. Our project was part of the Harvey Mudd Computer Science REU, with funding that originated from the National Science Foundation. The website/blog that organizes everything related to this project (including our code repository, code documentation, and research artifacts) is hosted at `https://jhallard.github.io/3DLocalization`.

2 Monte Carlo Localization Process Overview

The overall process of having an actor[2] localize itself in an environment via the MCL algorithm is comprised of several steps. These steps can be broken into two parts : steps taken before the localization attemp, and steps taken during localization.

2.1 Pre-Localization

These are the requirements that must be satisfied to begin the MCL algorithm.

[1] Dieter Fox, et al. Carnegie Mellon University, University of Bonn.

[2] An actor is any device that has sensors and can move around an environment, e.g. a robot.

1. A map of the environment needs to be imported or constructed.
2. Some type of quantifiable features present in the map that can be detected with a sensor on the robot must be chosen (image features, range data, etc). This enables a comparison between sensor readings from the actor and the expected sensor readings from different places in the map.
3. This step is optional. Features from the map are computed from many different reference points and are stored for use during the localization attempt. This significantly narrows the set of locations that the robot can be determined to be at compared to generating feature data at run-time, but it reduces the computational resources needed significantly and allows one to simply look up the data in a map or container as opposed to computing it in real-time. Pre-computing features also significantly reduces the complexity of the localization source code.

2.2 During Localization

The MCL algorithm iterates through a series of steps to determine the location of an actor. Each step can be implemented in a variety of ways.

1. An actor is placed somewhere in the environment, and some number of guesses as to where it might be are randomly generated.[3] These 'guesses' are called particles, and each particle is a data structure that contains information about its own current perspective in the environment (ex [x, y] coordinated) and the sensor data it would expect to read from that perspective (ex. $[d] \in \mathbb{R}$) distance in meters to the nearest plane).
2. The program compares the current sensor readings from the actor to the expected sensor readings for each particle, and assigns a weight to each particle based on how strongly the readings correspond to one another.
3. A distribution is created according to the grouping and weighting of particles active in the program. The more heavily weighted a particle is, the higher probability it or one of its close neighbors will be sampled from our distribution during the next step.
4. A new set of particles are sampled from this distribution, as well as a small amount are sampled from a uniform distribution across the map. After this step the total number of particles in the program is the same as in the last step.
5. The actor is moved to a new point in the environment via some movement command from the actor control program. Each particle has its perspective updated according to the same commands, plus some statistical error from the uncertainty in the actor's movements. The expected sensor readings for each particle are also updated to correspond to its new perspective of the environment.
6. Steps 2-4 are repeated until we receive an `end-program` command or we otherwise lose contact with the actor. As long as there is a connection our program will attempt to compute the actors location and relay this data to the actor. The general idea is as we weigh and resample the particles, the locations of the particles should converge on the actor's location in the environment.

The power of this algorithm comes from a few key characteristics.

[3]The uniformity of this distribution depends on the user's knowledge of the actor's initial location.

- The algorithm is general and leaves a large amount of the implementation details for the user to decide. As mentioned before, this algorithm has been used in the past with both laser and sonar range data and a 2D map of an environment, in our case it was used with images from the robot and a 3D model of the environment.

- This algorithm takes into account the movement of the actor in the environment in addition to the weighting of the particles with respect to the current perspective. Over time, impossible paths will cause incorrect particles to be removed from the current set of guesses, leaving room for better guesses to take their place. An example of this would be a particle on the outer x-boundary of a map, then the actor reporting a movement in the x direction. This would move the particle off of the map, and we would know the particle was an invalid guess. Overall, this results in the gradual merging of particles in space to the correct location.

3 Building the 3D Map

Our research team used 3D models as maps of the actor's environment for the localization process. These models were fully textured, high-quality scans of a room or series of rooms and hallways. The maps were built using a Matterport 3D imaging camera donated to our research group by Matterport Inc. This camera is attached to the top of a tripod and placed somewhere in an enclosed environment, it then uses an internal motor to rotate 360 degrees and take a scan of the space using both an RGB camera and laser-range sensors. Subsequent scans are merged with the existing model, allowing one to walk the camera around a space and progressively build a larger and more detailed model.

Figure 1: 3D model of the Sprague environment.

The main environment that we wished to localize an actor in was the 2nd floor of the Sprague building. We took 42 scans of this space (See Figure 1), a roughly 3500-square-foot floor consisting of 8 rooms. The Matterport software automatically converts the data to a Wavefront object file format for viewing on their site. We were then able to download this file

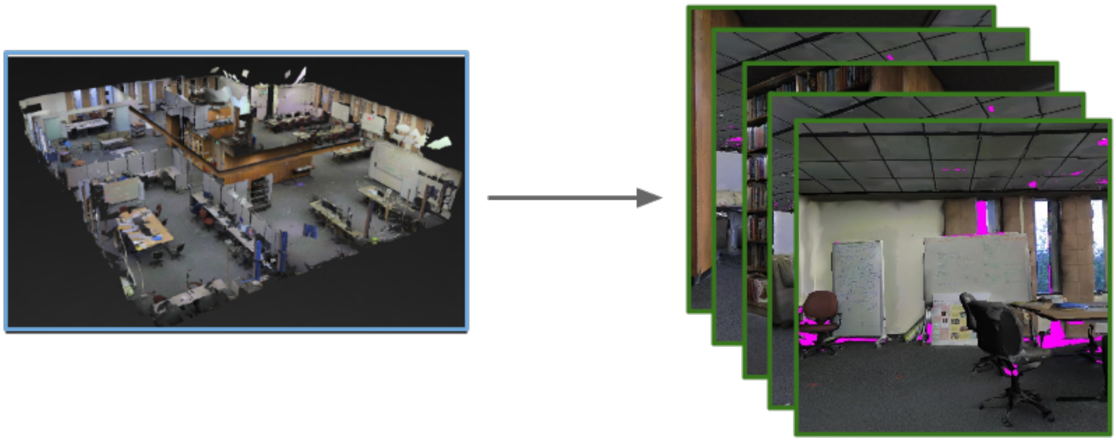

Figure 2: Turning a 3D model into 2D images.

to use it in our own 3D model viewers. Our model had a few gaps in places that were obstructed from the view of our camera. To improve the overall quality of our map we used the Meshlab software. This allowed us to remove disconnected pieces, remove unreferenced vertices, and smoothen out some jagged areas. on top of this, we rendered the background color pink to allow us to single out features on these areas and remove them from the program.

4 Creating a Database of Images from the Map

4.1 Overview

Our goal is to use computer vision algorithms that perform comparisons between 2D images from the actor and different places in our map. This means that we need to represent different perspectives within our three dimensional map in two dimensions for our feature matching algorithms to work. To do this, we reduce our single 3D map to a large number of 2D images from different perspectives within the map. This transformation allows a direct comparison between the 2D images taken in from the actor and the different particle locations in the map.

4.2 Process

The method our group used is outlined below.

1. Establish a bounding box around our map.
2. Define a plane, bounded by the box defined above, that sits above and parallel to the x-y plane of our map at some constant z height.
3. Define the a grid of a certain density on this plane (e.g. `[0.2 x 0.2]` grid squares).
4. Go to each grid intersection and render images from `n` perspectives, where `n` is some number that cleanly divides 360. We used `n = 8`, rendering pictures from angles of 0, 45, 90, . . . , 315, 360 degrees.
5. Name the images according to their location in the map and store them for later use.

This process allows us to turn our 3D model into a catalogued database of 2D images from a large amount of places within our model. From here we can use existing computer vision related algorithms (SURF, SIFT, ORB, etc.) to detect keypoints and describe features needed to match images with one another.

5 Computing Features from the Map

After generating a database of images and labeling them according to their perspective in the environment, we compute various types of image features for each image. This is done before localization because computing features is a computationally intensive process, and doing so during runtime significantly slows down the localization process. The information catalogued during this stage is loaded during the boot phase of the localization program.

5.1 Types of Features

There are a variety of features that can be used when describing an image. Our algorithm uses SURF features as well as grayscale and black and white images.

SURF (Speeded Up Robust Features) is a feature detection algorithm similar to SIFT (Scale-Invariant Feature Transform). Using OpenCV, we detect these points, describe their features, and save the descriptions.

The grayscale image is a highly coarse image that simply splits an image into a grid, then computes the average intensity inside each grid square. From this image, we can compute the black and white "above below" image. This is created by determining if a square is either higher or lower than the average intensity of the grayscale image. If a square is higher it is colored white, otherwise it is colored black. Both the grayscale and the above below images can be represented by small matrices that can be used in quick comparison operations.

5.2 Storing the Features

The precomputed features are stored in a yaml file. OpenCV has built in storage method, which makes this a simple process. Because each perspective has its own file for storing keypoints in addition to its own file for grayscale and black and white images—the filename contains the meta information about each description. The program is able to load all features and know their corresponding location and orientation by first looking inside the folder to see what perspectives are available, then load into the map.

6 Implementation

6.1 Overview

Our implementation of the Monte Carlo Localization algorithm is contained inside of a program called `3DLocalization`.[4] This program is written entirely in C++, using various open-source libraries such as OpenCV, ROS, and Boost to aid in development. All of the data that was precomputed as described in the previous sections is used by this program. Some of the main goals when designing this program are as follows

- Make the program relatively light-weight as to not redirect computational resources from the main robot-control program.
- Be able to run in the background with no user input needed after the initial starting of the program.
- Impose as few restrictions on the characteristics of the actor as possible (e.g. dimensions of freedom, movement timing, wheeled vs airborne).
- Make the program robust to environmental changes (e.g. chairs being moved, lights being dimmed).

[4] See `https://github.com/jhallard/3DLocalization/Localization` for the 3D localization algorithm and `https://github.com/aarich/ROS-Controllers` for sample actor controllers.

6.2 Actor Specifications

One of the main objectives for our project was to be able to localize a *general* actor in a *general* environment. With this in mind, we tried to limit the number of requirements we impose on the movement and control of the actor. The robot must implement the following functionality to maintain the generality of our program.

- Send the 3DLocalization program images from the robot's camera. The rate at which it publishes these images is not specified, but generally faster is better.
- Send the 3DLocalization program movement commands after it performs a move. These movement commands are relative to the robot's local coordinate system.
- Subscribe to a single data feed from our program. This data feed tells the robot the best guess for its location.
- Make decisions on movement. The robot must implement some kind of navigation algorithm

6.3 Program Flowchart

Figure 3 is a visual overview of our implementation of the MCL algorithm.

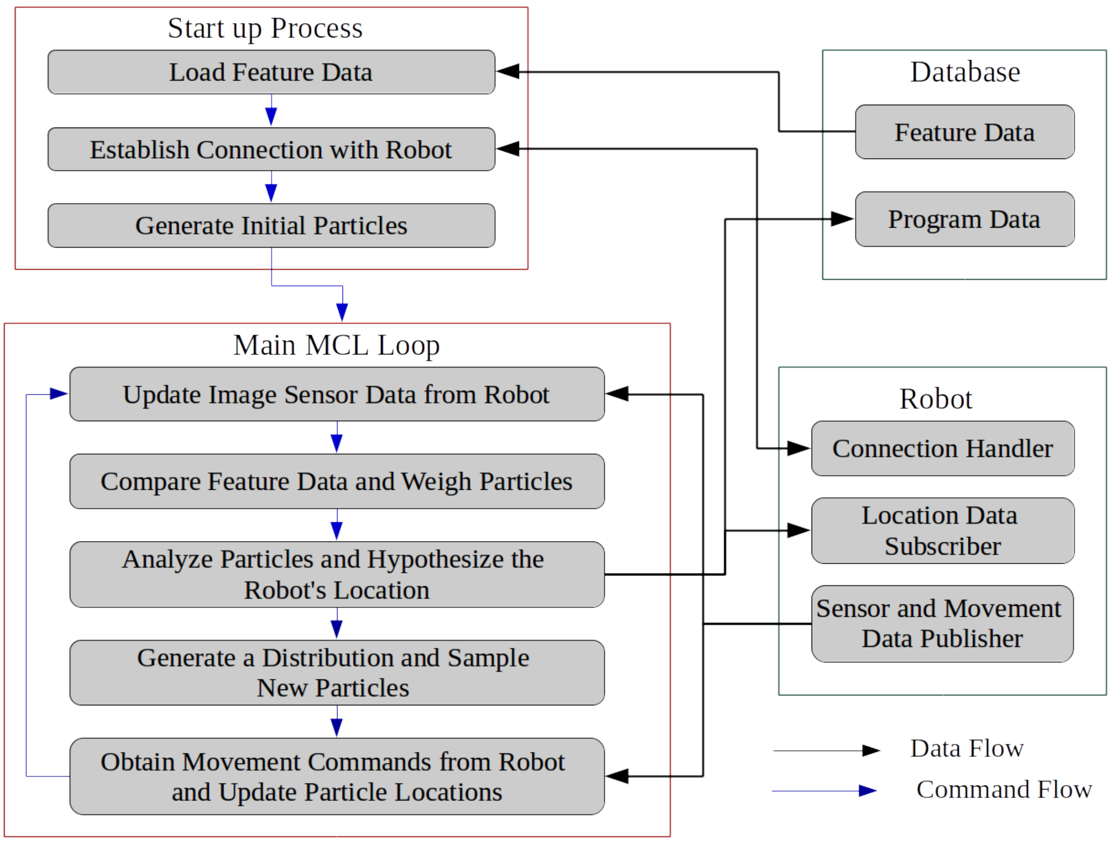

Figure 3: Data and control flow of the 3DLocalization program

The 3DLocalization program consists of a start-up process and a loop that continues until the program is told to end. During the various localization steps, the program must also communicate with two outside entities. The first is the robot, for the reasons listed in the above section. The second is the database. During the start-up process, we read in the map and feature data from the database. During the main localization loop, we document various aspects of our program by writing data to the database.

6.4 Particle Generation

The MCL algorithm is a kind of particle filter. At any one time during the process, there is a set of active particles that is a subset of all possible particles. There can be anywhere between 200 to 600 active particles at a time, with fewer active particles resulting in a faster program. During each iteration, a distribution of particles is created, from which the next set of active particles are sampled.

6.4.1 Generating a Distribution

The distribution is created such that when the next set of particles are sampled

6.4.2 Sampling from the Distribution

6.5 Weighing the Particles

Each particle in our list of active particles contains a weight associated with it that represents the probability that the actor is currently at the particle's location in the map. This weight is determined by comparing the precomputed feature data for the perspective associated with the particle to the feature data computed in real-time from the actor's image feed. The weight assigned to the particle proportionally affects the probability of another particle being sampled from near that perspective in the environment during the next loop iteration of the algorithm.

Before a particle can be weighed, it must have its image features matched with the features from the actors image. This entails going through the features for both images and seeing which features match best to each other. To begin, we will define what information is needed to start the feature matching process.

First, we have two vectors of SURF features, one for the robot ($\boldsymbol{F_r}$) and one for a specific particle ($\boldsymbol{F_p}$). Each vector contains features `f` detected in its respective image. There must exist a distance function f that determines the similarity between any two features.

$$f([\mathtt{f_{pj}}, \mathtt{f_{rk}}]) = \mathtt{distance}$$

We use an OpenCV `FlannMatcher` object to generate an initial matrix of matched features between the two images. The `FlannMatcher` object has a built-in function `KnnMatch` that return a matrix of matching pairs. Each row of this matrix M is a matching vector consisting of `n` columns, defines as follows.

$$M = [\mathtt{f_j},\ \mathtt{f_{k1}},\ \mathtt{f_{k2}},\ \mathtt{...}\ ,\mathtt{f_{kn}}]\ \mathtt{such\ that}$$

$$\begin{aligned}
&f([\mathtt{f_j}, \mathtt{f_{k1}}]) < f([\mathtt{f_j}, \mathtt{f_x}])\ \forall \mathtt{f_x} \in \boldsymbol{F_r} \\
&f([\mathtt{f_j}, \mathtt{f_{k2}}]) < f([\mathtt{f_j}, \mathtt{f_x}])\ \forall \mathtt{f_x} \in [\boldsymbol{F_r} - \mathtt{f_{k1}}] \ldots \\
&f([\mathtt{f_j}, \mathtt{f_{kn}}]) < f([\mathtt{f_j}, \mathtt{f_x}])\ \forall \mathtt{f_x} \in [\boldsymbol{F_r} - \mathtt{f_{k1}} - \mathtt{f_{k2}} - \ldots \mathtt{f_{kn-1}}]
\end{aligned}$$

Thus a matching vector of length `n` contains a feature and the `n` features that are most similar to it. The *KnnMatch* function of order `n` return a matrix with a matching vector of order `n` for each feature in both the robot and particle feature vectors. For our purposes, we ran the *KnnMatch* function with `n = 2`. This gave us the two best matches for each feature.

$$Flann.KnnMatch(\boldsymbol{F_r},\ \boldsymbol{F_p},\ 2) \rightarrow \boldsymbol{M} = [\boldsymbol{m}_0, \boldsymbol{m}_1, ..., \boldsymbol{m}_\mathrm{n}]$$

$$\texttt{where } \boldsymbol{m_i} = [\texttt{f}_\texttt{j}\texttt{, f}_\texttt{k1}\texttt{, f}_\texttt{k2}]$$

The *Flann* function will give us a matrix of feature matches, but these matches still need to be refined. the ratio test[5] we refine the matrix of top-two matches into a smaller vector of best matches. The ratio test is described below.

INSERT MATH HERE SHOWING THE RATIO TEST -
Now we need to get some kind of score for how well the matches correspond to each other, and use this score to determine how closely the image from the actor and the image from the particle's perspective match with each other. Each pair of matching keypoints has a 'distance' score that can be calculated. This value tells us roughly how similar that pair of keypoints are to one another. Using the vector of matching keypoint pairs, the overall weight for a particle is determined as follows

$$\texttt{Given a vector of n matching feature pairs}$$
$$\boldsymbol{M} = [\boldsymbol{m}_0, \boldsymbol{m}_1, ..., \boldsymbol{m}_\mathrm{n}] \texttt{ where } \boldsymbol{m_i} = [\texttt{f}_\texttt{pi}, \texttt{f}_\texttt{ri}]$$
$$f(\boldsymbol{M}_i) \rightarrow \texttt{distance} \in \texttt{R}^+$$
$$\texttt{Particle Weight w} = \sum_{i=0}^{n} \frac{1}{f(\boldsymbol{M}_i) + 0.8}$$

In summary, this algorithm iterates through all $\texttt{n}$ pairs of keypoint matches in our vector of matches (**M**) and sum up a value that gets larger when the distance between the keypoints in an individual pair are small and smaller when that distance in large. The added 0.8 in the denominator is to stop a feature pair with a very low distance from having a disproportionally high influence on the overall weighting of the particle. For example,

$$\texttt{If } f(\boldsymbol{M}_i) \rightarrow \texttt{distance} = \texttt{ 0.001 .}$$

Without the added contant (0.8) in the denominator

$$\frac{1}{f(\boldsymbol{M}_i)} = 1000$$

1000 is much too high of a value for a single match to contribute to the sum, so adding the extra contstant in the denominator makes it so even the best key point matches only add a value slightly greater than one to our sum. This algorithm possesses a few traits that are beneficial for the accuracy and speed of our program.

- The value of the $\texttt{w}$ variable increases as the number of keypoint matches increases. This is desired, an image pair with many feature matches has a higher chance of being them being similar images than an image pair with few feature matches.
- Keypoint pairs with a high distance (are not very alike) will contribute a small amount to the sum, while keypoints pairs with low distance (are quite similar) will contribute a larger amount to the sum. This will cause images with similar keypoints to be weighted higher than images with non-similar keypoints.

Another formula that does well is the following:

$$\texttt{w} = \texttt{n} - 10 \times \frac{\sum_i^n f(\boldsymbol{M}_i)}{\texttt{n} + 1}$$

This also rewards images that have many good matches with the first term, then penalizes pairs of images where the average match distance is high.

[5] See Lowe, David G. "Distinctive Image Features from Scale-Invariant Keypoints." *International Journal of Computer Vision* (2004). pg 19-20. `http://www.cs.ubc.ca/~lowe/keypoints/`.

6.6 Determining the Location

The list of weighted particles has been computed and can now be used to locate the actor. There are a few ways the location can be determined: Top Match, Weighted Average, or a combination of the two. The Top Match is simply the perspective that has the highest weight. This is a fairly good estimate, however it disregards much of the information that the MCL algorithm provides, such as the weights of all other perspectives.

The second option is the Weighted Average, which looks at the top n particles (we used $n = 20$). It finds the weighted average position of these particles and the most common orientation. This tends to be accurate, but can have problems when, for example, there is a region in the middle of a room that a actor cannot possibly be, it is possible to return a location inside this region. To combat this issue, we use the Top Match when the Weighted Average is at an impossible location.

6.7 Visualization

For the user to see how the localization program is working, various visualization assistants were constructed to accompany the program. A user is able to see what the robot sees, a top-down view of the map showing the location and relative weights of each particle, the Top Match and Weighted Average chosen by the program, and a three dimensional visualizer showing all the particles within the actual map.

7 Results

We can analyze how our program performs for each of the initial objectives.

7.1 Runtime Speed

Depending on the type of feature matching being performed and the number of particles currently active, our algorithm takes less than one second to determine a guess for the location of the actor. When the number of particles surpass 700, this time begins to noticeably take longer than a second. We consider the algorithm to be working in real time because it usually takes longer for the robot to make its move than the algorithm takes in each iteration.

7.2 Accuracy

The localization efforts tend to be accurate without precision. In most runs, the best guess for the robot was correct (it was in the right room, often looking the right direction), however was within a few meters of the precise location.

7.3 Versatility

Our algorithm is versatile to the extent that a map exists. It works with any robot that communicates with ROS. As long as a map of an environment exists in .obj file format, the program is able to localize a stream of image data to the best of its ability.

7.4 Ease of Use

Through a series of well-documented commands, our localization program is fairly easy to use. The most difficult aspect of running the program is setting up the libraries. OpenGL, OpenCV, and ROS are all required.

8 Issues and Future Work

8.1 Dynamic Environments

In our implementation we assume a static environment. This is not always the case, as there are often people inhabiting the same space as robots. Also, objects like chairs, coffee cups, and even tables are not guaranteed to remain in the same position for any period of time. Future work will involve locating objects that are deemed non-static and ignoring them in the weighing process. Moving objects, such as a person walking, ought to also be ignored.

8.2 Wall Detection

If the MCL program knew where walls are in the environment, smarter weighing and weeding procedures can be used. Knowledge of walls might allow a robot to decide which wall it is looking at within the map. Wall awareness also allows for the weeding-out of particles that pass through a wall during their movement process. Since a robot cannot pass through walls, the particle that moved through a wall is an impossible location, and ought to be removed from possible locations.

8.3 Navigation

It is certainly possible to implement a smarter navigation algorithm in which a robot knows where walls, doors, and tables are. In our current implementation the robot merely travels in a straight line from its location to the destination, however this can cause it to bump into tables or crash into walls. A smart navigation algorithm would allow the robot to pass through a door and, if it were a drone, to fly over tables.

8.4 Weighing

When each particle's image data is compared with that of the robot, there are many different ways to arrive at a similarity score. Currently we use only the number of feature matches and the "goodness" of each match. There are, however ways to incorporate algorithms such as RANSAC (Random Sample Consensus) to consider how well the matches align with each other. In our attempts at using this strategy, however, we found it was difficult to find one formula that will consistently weigh pairs of images with different match quantities. In addition, there may be quick image comparison techniques available to relate the black and white and gray scale images.

9 Conclusion

We have successfully implemented a 3D Localization tool for a robot using only a camera as sensor data. While there are features that could be added in the future, our current system is a proof of concept.