Using 3D Models and Computer Vision Algorithms to Implement Monte Carlo Localization

John Allard, Alex Rich

2014 Summer Computer Science REU, Harvey Mudd College*

August 13, 2014

Contents

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 2D maps of an environment and a stream of range sensor information. Our goal was to build off of these successes and implement a similar algorithm using a 3D map 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 in attempt of 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 it appears that this will be a common goal for robots to be able to perform in the near future. 3D models of an environment might be difficult to curently obtain, but they are constantly getting easier and cheaper to create. Our camera was donated by a new start-up company called Matterport, as companies like this grow we should see a proliferation of 3D-imaging cameras accompanied by a general decrease in price and increase in scan-quality. The reason we decided to use a camera as a sensor device instead of a range sensor is because cameras have continued following the trend in recent years of becoming cheaper and more portable, and the extra information present in an image compared to a set of range sensor data could make the localization process easier and more robust. The combination of these two trends suggests that the need for research related to localizing a robot using 3D models and image data is going to grow in the upcoming years, which is why we decided to undertake on this project.

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 Fondation. The website/blog that organizes everything related to this project (including our code repository, code documentation, and research artifacts) is hosted here :
`https://jhallard.github.io/3DLocalization`

---

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

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 many steps. A general outline of these steps is given below.

2.1 Pre-Localization

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

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 different 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' we will call particles, and each particle is a data structure that contains information about its own current perspective in the environment and the sensor data it would expect to read from that perspective.

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 distibution, 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.

---

[2]An actor is any device that has sensors and can move around an environment, e.g. a robot.
[3]The uniformity of this distribution depends on the user's knowledge of the actor's initial location.

6. Steps 2-4 are repeated until we recieve 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 characterstics.

- The algorithm is very general and leaves a large amount of the implementation details for the user to decide. As was stated 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 is applicabile to virtually any type of sensor data, as long as this data can be compared to the expected sensor readings that various places in a map.

- Secondly, from the mobility of the particles. When the actor moves, the particles move, and thus they store the probability that the actor has traveled in the path that the particle has taken. 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. This results in the gradual merging of particles in space to the correct location.

3 Building the 3D Map

Our research team used 3 dimensional models to serve as maps of the actor's environment for the localization process. These models were fully textured, high-quality laser scans of a room or series of rooms and hallways. The 3D map was built using a high quality Matterport camera and range scanner donated by Matterport Inc. This camera is placed on 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 RGB and laser-range data. After the next scan of the space is taken, the software internal to the camera automatically merges this new scan with the old scans, allowing one to continuously build a larger and more detailed model of an environment.

3.1 Taking the Scans

Our team took scans of various places on the Harvey Mudd campus, but the main environment that we wished to localize ourselves in was the 2nd floor of the Sprague building, where we work. We took 42 scans of this space, a roughly 3500 sqft floor consisting of 7-8 rooms and a large amount of non-static objects, such as chairs, laptops, and people. The Matterport software allows about 100 scans for a single model, and the software bundled with the camera automatically merges these scans into one textured mesh. The software also automatically converts the data to a `Wavefront .obj` file format for viewing on their site. We were then able to download this file to use it in our own 3D model viewers.

3.2 Filling In the Gaps

Our model was inevitably left with gaps from places that were obstructed from the view of our camera. To help 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.

Figure 1: 3D model of the Sprague environment.

3.3 Creating a Database of Images from the Map

Our goal is to use computer vision algorithms that perform comparisons between 2D images, which meant that we had to represent our 3D map in 2D for our feature matching algorithms to work properly. To represent our 3D map as 2D information, we decided to render images from thousands of different perspectives within our map. The following steps were taken to accomplish this.

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. Our group used `n = 8`, rendering pictures from an angle of `[0, 45, 90, ..., 315, 360]` degrees in the x-y plane at each grid intersection.

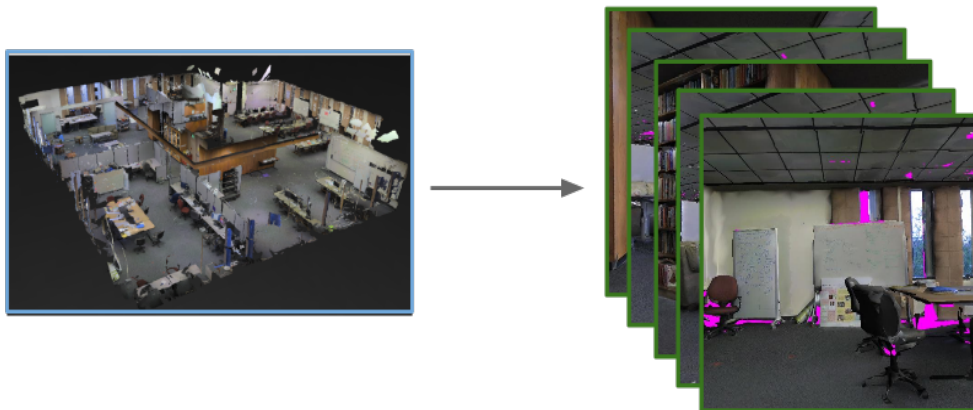5. Name the images according to their location in the map and store them for later use.

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

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

6

vision related algorithms (SURF, SIFT, ORB, etc.) to do the keypoint detection and feature description needed to match images with one another.

4   Computing Features from the Map

After generating a database of images and labeling them according to their perspective in the environment, we had to go through and compute various types of image features from each one. 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.

4.1   Types of Features

There are a variety of features that can be used when describing an image. In its current state, the project uses extracted SURF features as well as grayscale and black and white images.

SURF (Speeded Up Robust Features) is a feature detection algorithm that detects similar features as SIFT (Scale-Invariant Feature Transform). SURF detects interesting keypoints in an image in a manner that is a bit quicker than some competing algorithms such as SIFT, but it also lacks some nice properties like scale invariance. Using OpenCV, we can detect these points, describe them as features, save these descriptions, and eventually compare these features with ones computed from the robot during the localization process.

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 finding the average intensity of the grayscale, then determining if a square is either higher or lower than this. If a square is higher, it is colored white, otherwise it is colored black.

4.2   Storing the Features

The precomputed features are stored in a yaml file. OpenCV has built in storage methods, 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.

5 Implementation

5.1 Overview

Our implementation of the Monte Carlo Localization algorithm is contained inside of a program called `3DLocalization`, which relies directly on the pre-computation performed by the programs discussed in sections 3 and 4 of this document. Below is a visual overview of the 3DLocalization program.
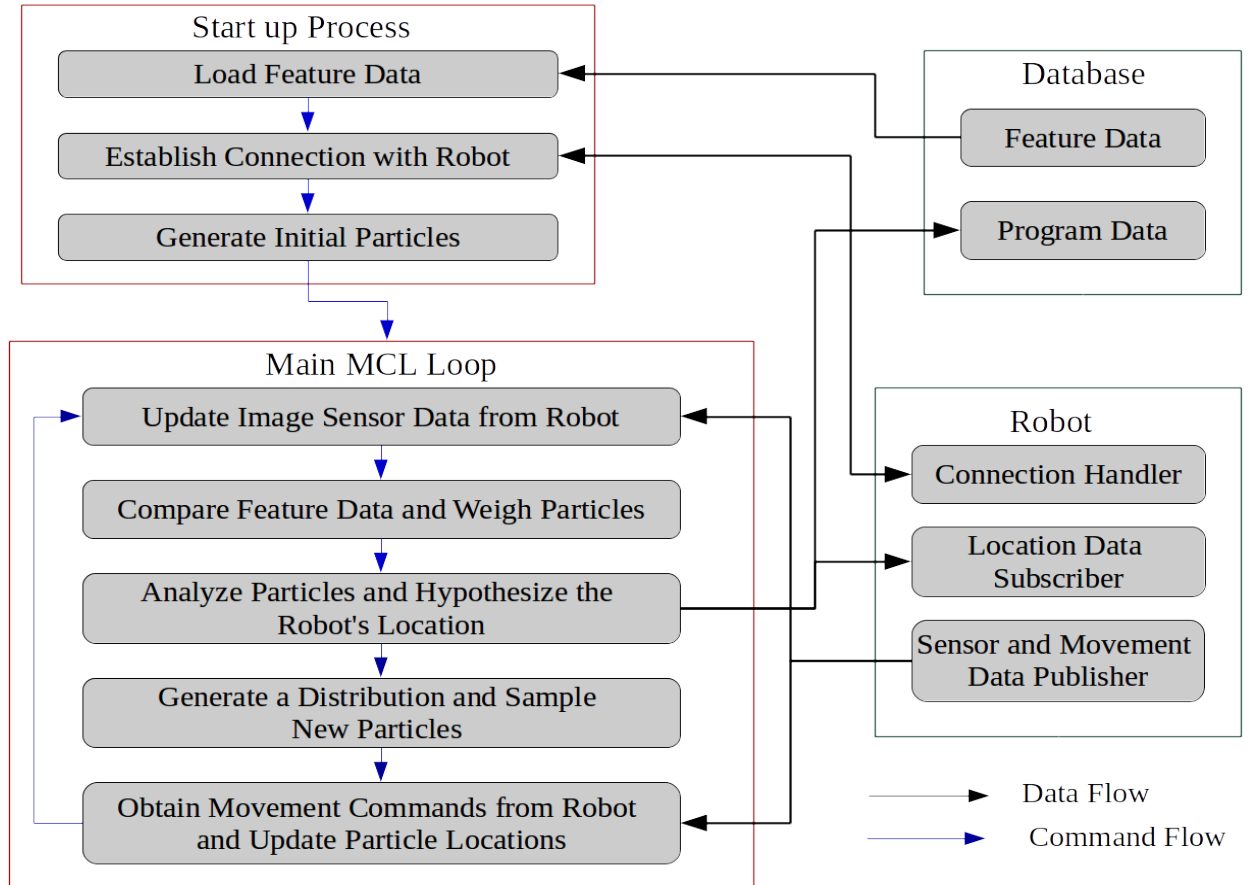
Figure 3: Data and control flow of the 3DLocalization program

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 environment changes (e.g. chairs being moved, lights being dimmed).

These goals are be tough to achieve, and are some what contradictory in their implementation. For instance, making the program robust to enironment changes entails adding more particles to the program which in turn makes it less light-weight.

5.2 Actor Specifications

- **Overview**

  One of the main goals 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. In summary, the actor must send us data from a camera and let us know its local movement coordinates, as well as listen for its global coordinates from our localization progran. All other control aspects of the actor are up to the user to decide. We have used this program successfully with both the ARDrone quad-copters and the ICreate robots, two very different robots. In our implementation, simple point to point navigation was sufficient, however a future goal is to implement the ability to make smart paths allowing the robot to go through doors and around or over tables.

- **Communication**

  The following functionality must be implemented by the robot

    - Publish data via two `ros::Publisher` objects. The first is for sending image data to our program, the second is to send its current movement commands to our program.

    - Subscribe to our data feed via a `ros::Subscriber` object. We will use this data feed to send localization data to the actor.

    - The control code for the actor must be developed separately, the user chooses how to move the robot, we only care about localizing it in the given environment.

5.3 Particle Generation

- **Generating a Distribution**

- **Sampling from the Distribution**

5.4 Weighing the Particles

Each particle in our particle vector contains a weight associated with it that represents the probability that the actor is currently at that particle's location in the real world. 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 will proportionally affect 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 eachother. To begin, we will define what information is needed to start the feature matching process.

Two vectors of SURF features

F_{robot}, $F_{particle}$ where $F = $ [f_1, f_2, ... , f_n]

A distance function that compares two features

$$f([f_{pj}, f_{rk}]) \rightarrow \texttt{distance} \in R^+$$

Our group decided to 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 is a matching vector consisting of `n` columns, defines as follows.

$$\boldsymbol{m} = [\mathtt{f_j},\ \mathtt{f_{k1}},\ \mathtt{f_{k2}},\ \ldots\ ,\mathtt{f_{kn}}]\ \text{such that}$$

$$f([\mathtt{f_j}, \mathtt{f_{k1}}]) < f([\mathtt{f_j}, \mathtt{f_x}])\ \forall \mathtt{f_x} \in \boldsymbol{F_r}$$

$$f([\mathtt{f_j}, \mathtt{f_{k2}}]) < f([\mathtt{f_j}, \mathtt{f_x}])\ \forall \mathtt{f_x} \in [\boldsymbol{F_r} - \mathtt{f_{k1}}]\ \ldots$$

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

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.

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

$$\text{Where}\ \boldsymbol{m_i} = [\mathtt{f_j},\ \mathtt{f_{k1}},\ \mathtt{f_{k2}}]$$

The *Flann* function will give us a matrix of feature matches, but these matches still need to be refined. the ratio test[4] 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

$$\text{Given a vector of n matching feature pairs}$$

$$\boldsymbol{M} = [\boldsymbol{m}_0, \boldsymbol{m}_1, ..., \boldsymbol{m}_\mathtt{n}]\ \text{where}\ \boldsymbol{m_i} = [\mathtt{f_{pi}}, \mathtt{f_{ri}}]$$

$$f(\boldsymbol{M}_i) \rightarrow \texttt{distance} \in \mathtt{R}^+$$

$$\text{Particle Weight}\ \mathtt{w} = \sum_{i=0}^{n} \frac{1}{f(\boldsymbol{M}_i) + 0.8}$$

In summary, this algorithm iterates through all `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,

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

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.

---

[4]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/`.

- The value of the `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 very 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 very similar keypoints to be weighted higher than images with non-similar keypoints.

Another formula that does well is the following:

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

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.

5.5   Determing the Location

The list of weighted particles has been computed and can now help 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 sums up all of the locations, weighted by their probaility, and finds the average. Once the average position is found, the locating algorithm only looks at nearby particles to determine the actor's orientation. This is also good, but sometimes does not make sense, for example if there is a square 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.

5.6   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.

6   Results

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

6.1   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 noticably 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.

6.2 Accuracy

6.3 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.

6.4 Ease of Use

7 Issues

7.1 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 nonstatic and ignoring them in the weighing process. Moving objects, such as a person walking, ought to also be ignored.

7.2 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 throuh a door and, if it were a drone, to fly over tables.

7.3 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.

8 Conclusions

What a dumb robot.