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

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

July 6th, 2014

Contents

1 Introduction **3**

1.1 MCL Process Overview 3

2 Building the 3D Map **4**

2.1 Taking the Scans 4

2.2 Filling In the Gaps 4

2.3 Creating a Database of Images from the Map 4

3 Computing Features from the Map **4**

3.1 Types of Features 5

3.2 Storing the Features 5

4 MCL Implementation **5**

4.1 Actor 5

4.2 Assigning Weights 5

4.3 Determing the Location 6

5 Results **6**

6 Conclusions **6**

Abstract

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 research group attempts to implement the Monte Carlo Localization algorithm using a 3D model of the 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. This paper will outline the overall processes that our research group has undertaken to accomplish this task and an analysis of our resulting program.

1 Introduction

1.1 MCL Process Overview

This section is for those unfamiliar with the Monte Carlo Localization algorithm.
The overall process of having an actor[2] localize itself in an environment via the MCL algorithm is comprised of many steps.

Pre-Localization Attempt:

1. A map of the environment needs to be imported or constructed.
2. Some set of quantifiable features about the map must be chosen. This enables a comparison between sensor readings from the actor and expected sensor readings from different places in the map.

During Localization Attempt:

1. An actor is placed in the environment, and some amount of guesses as to where it could be are randomly generated according to some distribution.[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 in the program. The more heavily weighted a particle is, the higher probability it will be sampled from our distribution.
4. A new set of particles are sampled from this distibution, as well as 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 program. Each particle has its perspective updated according to the same commands, plus some statistical error based off of the uncertainty in the robot's actual movements relative to the movement commands. 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 the localization is stopped.

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

2 Building the 3D Map

We used a 3-dimensional model as our map of the robot's environment. The models tested were fully textured, high-quality laser scans of a room or series of rooms and hallways. The 3D model was important because we intended on using cameras as our sensors for the actor in our environment. The 3D map was built using a high quality camera and range scanner donated by Matterport. The camera interfaces with an iPad and scans its surroundings from multiple vantage points, eventually stitching together everything it sees into one map.

2.1 Taking the Scans

Our team took 42 scans of the space we work in, a roughly 3500 sqft floor consisting of 7-8 rooms and a large amount of non-static objects, such as chairs, robots, and whiteboards. 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 .obj file format for viewing on their site. We were then able to use this object file to determine features about our map.

2.2 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.

2.3 Creating a Database of Images from the Map

Our goal is to use computer vision algorithms that compare 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 that sits above and parallel to the x-y plane of our map.
3. Define the scale of a grid imposed on this plane.
4. Go to each grid location and render images from 8 perspectives, rotating from 0 to 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 do the matching and weighing between places in our 3D model and the incoming image feed from the actor in the environment.

Once the database of images was computed, the next step was computing another database of computer-vision derived features from these images.

3 Computing Features from the Map

Once we have a database of images, we must go through each image and compute a set of general feature data about each picture. This is done before localization because computing features is fairly computationally intensive, and doing so during runtime slows localization down. During the boot phase of localization, we load all features into a map.

3.1 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. Using OpenCV, we can detect these points, describe them, and eventually compare two keypoints.

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.

3.2 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.

4 MCL Implementation

4.1 Actor

The actor in our MCL implementation satisfies three constraints: communicate with the Robot Operating System (ROS), move autonomously, and publish image data from a camera. During our testing, we used several robots, including the Parrot AR Drone, the iCreate Robot, and a virtual robot. The actor also should be able to navigate in its environment. 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.

4.2 Assigning Weights

Each possible location in the environment is described by a discrete particle. The particle contains a weight associated with the probability that the actor is currently at that particle in the real world. When the robot moves, every particle moves as well. During each iteration, the program reevaluates by weighing every particle sampled from the distribution. Using the stored features, computer vision algorithms determine the similarity between the actor image data and the image data at each particle.

There are several possible ways two images can be compared to each other. Issues with scaling, lighting, and photo quality can make comparing images difficult. In its current implementation, our algorithm only uses computed SURF features. OpenCV has methods to match descriptors in two images. Using the k nearest neighbors matching algorithm with $k = 2$ and the ratio test[4] we get some set of matched keypoints. We want to get some kind of score for how "good" the matches are. Each match (stored as a `DMatch` in OpenCV) contains a member (`distance`) that indicates the goodness of a single match. A weighting for each pair of images can be determined by

$$\texttt{weight} = \sum_i \frac{1}{\texttt{matches}[i].\texttt{distance} + 0.8}$$

where `matches` is a vector of `DMatches`. This formula has several benefits. First, it rewards pairs of images that have many good matches. Recall that the higher the similarity, the higher the probability that the two images are the same. If two images have many matches (as two similar images ought to),

[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/`.

`matches.size()` will be high and there will be many terms in the summation. Secondly, it rewards matches that are *good*. If a match is a good match, it will have a low `distance`.

Another formula that does well is the following:

$$\texttt{weight} = \texttt{matches.size}() - 10 \times \frac{\sum_i \texttt{matches}[i]\texttt{.distance}}{\texttt{matches.size}() + 1}$$

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.

4.3 Determing the Location

Once we have

5 Results

What a dumb robot.

6 Conclusions

What a dumb robot.