

Robotic Localization and Mapping

CS 182 Final Project

Ivan Cisneros and John Holland

December 20, 2016

1 Introduction

Robot localization and mapping is an on-line filtering problem within the field of Artificial Intelligence that utilizes sensor information to approximate the current state of the robot, given an initial state estimate and a sequence of observations [4]. A popular algorithm is particle filter localization (aka Monte Carlo localization), which is a method for sifting through a set of states in a Hidden Markov Model to find the best state estimate. Given a set of observations, a particle filter estimates the posterior density of the state variables.

We chose to use this method in order to simulate a robot localizing itself in Harvard engineering buildings. We implemented a particle self-localization procedure, where a robot has knowledge of the map layout but no knowledge of its location within this layout.

We also implemented the start of a SLAM procedure, where the robot has no knowledge of the map layout, but uses its sensors and odometry information to construct a perceived map.

2 Background and Related Work

Particle filtering is a common technique used for Robot localization. For example, University of Washington has an entire lab dedicated to particle filter techniques for state estimation of robots (Robotics and State Estimation Lab). They introduce new techniques for particle filter efficiency by reducing the number of particles based on the confidence of the estimation. [3]

Because of the potential value in autonomous robotic mapping, there has been a lot of work on improving SLAM algorithms. One of the most cited papers on SLAM is Montemerlo's 2002 paper where he introduces the FastSLAM algorithm. In the FastSLAM algorithm, a particle is no longer just a robot pose, but a robot pose and the locations of several landmarks. In FastSLAM, landmarks are extracted from the environment. Particles are then reweighted based on their belief in their own position and belief in the location of previously observed landmarks. [1].

3 Problem Specification

The primary goal of this project was to implement a Monte Carlo localization based procedure that allows our simulated robot to estimate its location within any floor plan. Given an image of the map layout, the procedure would allow the robot to estimate its position on the map based on information that it gathers from moving around and sensing its environment.

Additionally, we wanted to extend this functionality to include mapping. If the robot were denied access to the map layout should be able to use SLAM procedure in order to generate a perceived map of the walls and obstacles that it comes across.

4 Approach

To avoid the complications of working with actual hardware, we developed a simulation system that simulates the robot, its sensors, and the environment that it is exploring. Our simulation system takes a png of a floorplan and converts this into a binary valued numpy grid where each grid element maps to a pixel. This map matrix is then utilized for calculating and displaying the robots position and sensor readings, as well as the particles and their locations.



overlays a png of the floorplan being used for the simulation with colored dots representing the robot (big blue dot) and the particles (small red dots). Upon initialization the ConvBinMap class generates a binary occupancy grid where 1 = obstacle (and is represented by black), and 0 = no obstacle (and is visually represented by white).

In this simulation, a user can pilot a simulated holonomic robot via keyboard input (see Appendix A) and move it around the building so that it can acquire the necessary observation data. The robot only has access to readings from a noisy simulated odometer and laser range finder, while the simulation framework keeps track of the robot's true state.

4.2 Particle Filter Implementation

The mechanics described here were implemented in ParticleFilter.py

Given a map of the environment, the goal of this algorithm is for the robot to determine its position within the environment [2]. Our particle filter consists of random initialization and 2 major steps. Each time the robot moves, each particle goes through a state transition model by sampling from a Gaussian representing the probability of an actual movement given an odometry reading. Then, particles are weighted by comparing sensor data to the theoretical perfect reading given the floorplan and a particle's state. Particles are resampled based on this weighting.

The particle filter class implemented the following algorithm along with some supplementary methods.

Algorithm 1 The particle filter algorithm.

```

procedure PARTICLEFILTER(particleList, movement, sensorReading)
  for  $particle_t$  in particleList do
     $particle_t = \text{motion\_update}(\text{movement}, particle_{t-1})$ 
     $weight_t = \text{sensor\_update}(\text{sensorReading}, particle_t)$ 
     $\text{newParticleList} \leftarrow [particle_t, weight]$ 
  end for
  return newParticleList
end procedure

```

Because we are using large environments, we needed to distribute a large amount of particles to cover as much of the area as possible. In order to visualize the improvement in localization as the robot moves about, it is necessary to display these particles as well, but as one can imagine displaying thousands of particles slows down performance significantly. Thus, we elected to only display the first 100 particles, and only in every 10 frames of the sequence. This allows us to keep a smoother visual representation without using too much RAM or slowing down the process too much.

4.3 SLAM Implementation

Our original plan was to use the FastSLAM algorithm, as outlined in [1] and [4], to generate a map while the robot localized itself within it. However, FastSLAM was designed to extract landmarks,

as opposed to filling in an occupancy grid, which was the approach we took for our particle filter. Therefore, we attempted to design our own approach to SLAM.

The robot's map would be an array of probabilities that a certain pixel contains an obstacle. Probabilities were initialized based on the ratio of filled locations to total locations in the true map. We also made the assumption that the starting location was known with a high degree of accuracy. Once the map was initialized, the SLAM algorithm would consist of two steps, only the first of which was successfully implemented. The first step was updating the map based on the robot's supposed location (as determined by the particle filter) and sensor readings. The second step would be updating the particle filter based on the robot's current knowledge of the map. Below is the algorithm for step 1.

Algorithm 2 The SLAM algorithm map update.

```

procedure UPDATEMAP(ScanAngles, ScanRanges)
   $\alpha \leftarrow \text{LearningRate}$ 
  for ScanAngle do
     $\sigma_1 \leftarrow \text{standard deviation of position along ScanAngle}$ 
    for Pixel along scan line do
       $\sigma_2 \leftarrow \text{sensor noise}$ 
       $\sigma_{Tot} \leftarrow \sqrt{\sigma_2^2 + \sigma_1^2}$ 
       $\text{NewProbability} = P(\text{PixelOccupied} | (\text{PixelRange} - \text{ScanRange}), \sigma_{Tot})$ 
       $\text{UpdatedPixel} = \alpha * \text{NewProbability} + (1 - \alpha) * \text{OldProbability}$ 
    end for
  end for
end procedure

```

5 Experiments

We experimented with varying the number of particles that were displayed and seeing how this affected the runtime. We conducted this test on two maps: Maxwell Dworkin ground floor (MD_0), and a subsection of Maxwell Dworking (MD_MINI), and for the two versions of our implementation: for the particle filter, and for the SLAM. The times listed are the per particle runtimes for a single time step in the scripts sequence.

Results for the particle filter version:

Num of Particles	MD_0	MD_MINI
100	0.00313s	0.00330s
1000	0.00310s	0.00316s

Results for the SLAM version:

Num of Particles	MD_0	MD_MINI
100	0.00342s	0.00358s
1000	0.00350s	0.00333s

From this we gather that the number of particles nor the size of the map affects the per particle runtime very much. Additionally, the extra computation introduced by the SLAM feature does not significantly impact runtime. The one issue is that the total runtime is linearly dependent with the number of particles (1000 particles take longer to update than 100 do), and so there are lengthy loop times when more than a 1000 particles are being used.

6 Results

6.1 Particle Filter

To demonstrate the functionality of our particle filter, we initially tested our Robot.py script with a partial version of the Maxwell Dworkin floorplan:

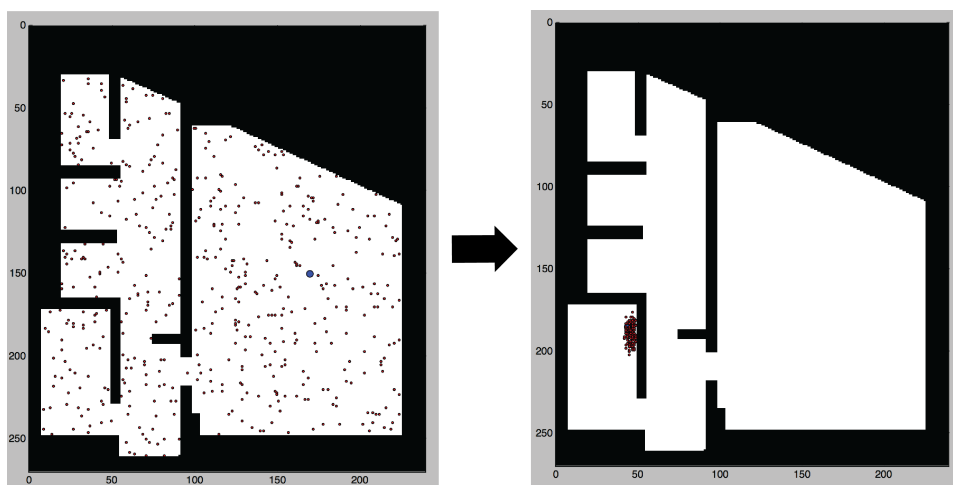


Figure 2: The first step and 10th step of the sequence. The robot was piloted towards the small room and the particles converged on its location.

This was done with 500 particles. We then experimented with applying the same amount of particles to the complete Maxwell Dworking floorplan, and managed similar results in fewer steps (8 steps):

An issue that we noticed with our particle filter implementation was that in order to accurately estimate the position of the robot within larger spaces we required a huge amount of particles (hundreds of thousands) due to the amount of possible grid spaces that the robot could occupy, which severely impaired runtime. Thus, in order to run the script at all, we elected to sacrifice some accuracy in order to have speedier loop cycles.

Another issue that we found was that our resampling method was too biased towards the higher weighted particles, and so the particles would converge on a small region with only a



Figure 3: The first step and 10th step of the sequence. The robot was piloted towards the small room and the particles converged on its location.

few iteration steps. The good thing about this was that because the weighting method was very precise, they very often converged on the correct region.

6.2 SLAM

Below is the robot's map of the subsection of Maxwell Dworkin ground floor.

Because we were unable to accomplish part 2 (updating the particle filter only using information from the robot's known map), we gave the robot access to the original particle filter. While our method maps the space with some success, there are a few issues with the algorithm we came up with. First, we take the standard deviation of the particles to be a confidence. However, when running the algorithm with a small number of particles, they would sometimes all share the same state despite not being accurate. Second, there is no provision for smoothing or updating pixels that do not lie along a sensor line, which leads slow mapping and the choppy pattern seen above.

7 Discussion

In summary, we implemented two robotic localization and mapping algorithms: a particle filter localization strategy, and a SLAM strategy. Both algorithms were heavily modified from their normal implementations, because we were working with simulated noisy data and a simulated robot, and thus had to devise ways to vary the information so that we could generate "uncertainty" and be able to generate probabilistic inferences based on the assumption of uncertainty.

We found that our particle filtering implementation was accurate, but weighted the particles too

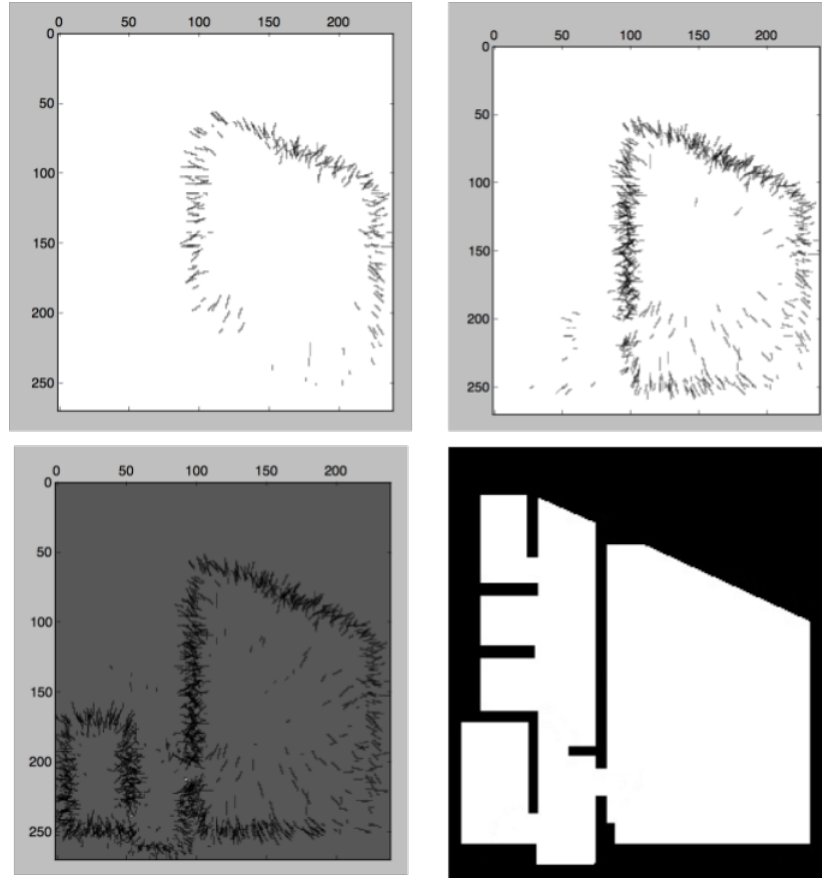


Figure 4: The robot's construction of this subsection of Maxwell Dworkin after a short, medium, and long time exploring. The fourth image is the input map.

heavily, and so they converged too quickly (though precisely).

Implementing SLAM proved to be much more difficult than implementing Monte Carlo localization alone, because we also had to create a way for the sensor to sweep around the robot, and correctly place its observations on an occupancy grid, in addition to updating the particle filter algorithm. Despite this, we obtained some promising results.

A Appendix 1 - System Description

This project was written in Python 2.7.

The following instructions assume a Linux type operating system. If you don't have pip installed, installing using:

```
sudo easy_install pip
```

A.1 Dependencies

The following python packages need to be installed to run the project scripts:

- Numpy
- Scipy
- Matplotlib

All three can be installed using the pip command:

```
pip install --user numpy scipy matplotlib
```

And for map generation we used a maintained version of PIL:

```
pip install Pillow
```

A.2 Running the code

The script relies on a few classes that create the simulated robot, the simulated odometry and sensors, the visualization, the particles and particle filter, and the movement.

Make sure that the following classes are all in the same directory:

- conv_to_bin_mat.py
- Odometry.py
- Sensor.py
- visualization.py
- TrueMap.py
- Particlefilter.py
- RobotMap.py
- Robot.py
- Slam.py

Additionally, the directory 'Binary Maps' should be in the same directory, as the above mentioned files. This is where the maps are stored.

To run the particle filter implementation, use command:

```
python Robot.py
```

Pilot the robot around the space and watch as the particles converge on the location of the robot
To run the SLAM implementation, use command:

```
python Slam.py
```

Pilot the robot around the space in order to map out the area. Moving the robot more improves its confidence in its map, and generates a more detailed and accurate representation.

Both simulation scripts open up a Python window that displays the true state of the simulation (the map, the robot's true location, and the generated particles). The Slam.py script opens up a second window, which displays the robot's generated map. Both simulation scripts rely on user input in order to move the robot. If you are not currently in a terminal window, navigate to the window so that you can input keyboard commands. To move the robot input one of the following keystrokes followed by pressing the 'enter' key.

- 'w' to move the robot upwards (north)
- 'a' to move the robot left (west)
- 's' to move the robot downwards (south)
- 'd' to move the robot right (east)
- 'aw' or 'wa' to move the robot northwest
- 'as' or 'sa' to move the robot southwest
- 'sd' or 'ds' to move the robot southeast
- 'wd' or 'dw' to move the robot northeast

The Python window/s should pop up after every input command, but you can continue to input commands into the terminal as long as you don't click on any other window.

B Appendix 2 - Group Makeup

For a complete set of the code used in this project, please see the publicly available github repository found here: https://github.com/icisneros/cs182_finalproject_ICJH

John Holland: Sensor simulation classes (Odometry.py, Sensor.py), Particle Filter (ParticleFilter.py), SLAM Algorithm (RobotMap.py), Poster, Report (30%)

Ivan Cisneros: Visualization environment (conv_to_bin_mat.py, visualization.py, TrueMap.py), Particle Filter (ParticleFilter.py), Robot Agent (Robot.py, Slam.py), Poster, Report(70%)

References

- [1] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In *Aaai/iaai*, pages 593–598, 2002.
- [2] Ioannis M Rekleitis. A particle filter tutorial for mobile robot localization. *Centre for Intelligent Machines, McGill University, Tech. Rep. TR-CIM-04-02*, 2004.
- [3] Washington University Robotics and State Estimation Lab. Particle filters - computer science engineering.
- [4] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.