Robotics
Practical 4: Monte Carlo Localisation

Andrew Davison
ajd@doc.ic.ac.uk

1 Introduction

This week we will follow on from last week's work to complete the components of a full algorithm for Monte Carlo Localisation, a probabilistic localisation filter. For this practical your robot will need wheels and a forward-looking sonar, and no other sensors will be used.

This practical will be ASSESSED. There are **40 marks** to be gained for completing the objectives defined for today's practical, out of a total of 100 for the coursework mark for Robotics over the whole term. Assessment will take place via a short demonstration and discussion of your robots and other results to me or my lab assistants AT THE START OF NEXT THURSDAY'S PRACTICAL SESSION (February 23rd). No submission of reports or other materials is required. We will assign marks based on our judgement of whether each objective has been successfully achieved.

2 Parts of the Algorithm

As explained in lectures, the main parts of one step of MCL are:

1. Motion Prediction based on Odometry
2. Measurement Update based on Sonar
3. Normalisation
4. Resampling

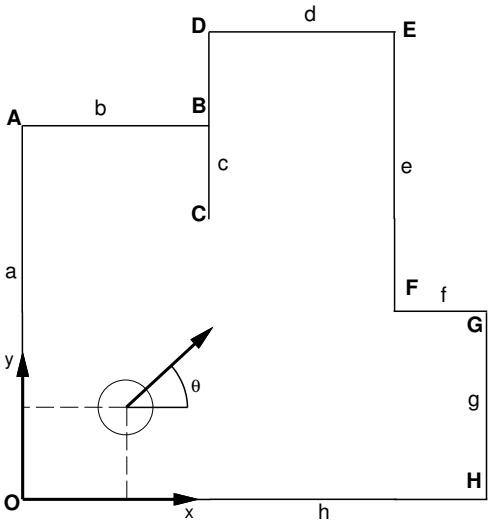
You should refer to the lecture slides for the details of how to implement these steps. The paper 'Monte Carlo Localization: Efficient Position Estimation for Mobile Robots' by Fox, Burgard, Dellaert and Thrun which is available from the Additional Handouts section of the course website gives more detail on MCL and is well worth reading. We covered step 1 last week. This week you will complete steps 2, 3 and 4. You should continue on from the code you wrote for last week and various things on last week's practical sheet will still be useful.

When the algorithm is complete, your robot should be able to move around the mapped environment in small steps of 20cm, pausing after each movement. During motion, the particle distribution should be updated to represent the uncertain motion measured by odometry. After motion, the robot should make a sonar measurement of the distance to the wall in front, and use this to adjust the particle weights based on a likelihood function. Then, it should normalise and resample the particle distribution and be ready to complete another motion step. By repeating these steps, hopefully the robot will be able to keep track of its location accurately (at least up to the limits of what is possible with only one sonar sensor).

2.1 The Environment and Map

The robots will run an enclosed area we have built in the lab using foamboard 'walls' 30cm high. The walls are moveable but the locations they should be in are marked with tape on the floor which has been put down with careful measurement.

Below you are provided with a 'map' of the course to use in MCL which consists of the coordinates of the endpoints of each of the straight walls.



In the figure, each vertex which forms the endpoint of a wall is marked with a capital letter. Each wall is marked with a small letter. A world coordinate frame W is defined as shown with the origin in the bottom-left corner. The orientation θ of the robot is defined as shown. The 9 points marked **O**, **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H** have the following coordinates in the world coordinate frame (all in centimetre units):

Point	x	y
O	0	0
A	0	168
B	84	168
C	84	126
D	84	210
E	168	210
F	168	84
G	210	84
H	210	0

The map consists of a list of 8 straight line segments a, b, c, d, e, f, g, h which join these points up. This data, and example code for displaying the map and current particle set via our Pi to web interface, is in this file which you can download: `http://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/particleDataStructures.py`. Note that if you wish to use different code and data structures that is no problem.

2.2 Web Interface

Our Raspberry Pi Web Interface, introduced in last week's practical, should be used. Particles can be displayed with colour indicating their weight, and this will happen if you draw them as in the file

`particleDataStructures.py` linked above. Note that this file also takes care of a coordinate transform such that you can keep your particle and wall definitions in centimetre coordinates but they'll be drawn exactly as in the figure above.

You should have already discovered our Python file editor built into the web interface — click on edit next to a file name to try this out.

2.3 Using Odometry

For your robots with the standard differential drive configuration, it is sensible to define the centre of rotation of the robot (i.e. the point centrally between the two wheels) to be its 'centre' — meaning that when you talk about the (x, y) world coordinates of the robot, you refer to the location of this point. The advantage of this is that when the robot rotates on the spot, this point should not move and it will be much easier to update your pose estimates.

2.4 Using the Sonar

The sonar is the crucial outward-looking sensor which is used to make contact with the mapped world. I am assuming that you will mount it looking straight forward in your robot's driving direction, although in principle this need not be the case. The sonar should be mounted horizontally and at a height ideally above 10cm and below 25cm to be sure that it 'sees' all the walls of the environment. The environment is small enough that the sonar's theoretical maximum range of 255cm will never be exceeded — though probably you have already discovered that the sonar sometimes doesn't work very well beyond a depth of around 100cm

An important point to consider is that your sonar will probably not be mounted exactly above the rotation centre of your robot relative to which you define its location. If your sonar is t cm in front of the robot centre, you can correct for this by just adding t to all of the sonar readings.

3 Objectives

3.1 Sonar Likelihood and Measurement Update (16 Marks)

Based on the information in the lecture notes this week, implement an update function which every time the sonar makes a measurement loops through all of your particles and updates each one's weight by multiplying it by an appropriate likelihood function which corresponds to how well that particle agrees with the sonar reading.

The likelihood function should be implemented as a function
`def calculate_likelihood(x, y, theta, z):`
which reads in the (x, y, θ) position estimate of a particular particle as well as the sonar measurement z and returns a single likelihood value.

This function you define will have several parts. First it needs to find out which wall the sonar beam would hit if the robot is at position (x, y, θ), and then the expected depth measurement m that should be recorded. The geometry I gave in the lecture should allow you to do this. Then, it needs to look at the difference between m and the actual measurement z and calculate a likelihood value using a Gaussian model (with a constant added to make it more robust) as I explained in the lecture. Use

standard deviation set according to what you learned about the sonar in last week's calibration exercise — I would probably use around 2–3cm to be a bit conservative.

The function could be made more sophisticated by also checking whether the incidence angle is going to be too big to get a sensible sonar reading. If too many particles say that this is the case, probably the best thing is to skip the update step entirely on this step. This is probably not needed this week because the trajectory for the robot to follow is designed so that it will usually be looking at walls face-on.

Remember that the absolute scale of the likelihood function is unimportant since we will later on be normalising all the particle weights to add up to 1.

To demonstrate the your likelihood function in the assessment, we will ask you to show us your Python code for the function (on screen is fine) and explain how it works. I do not think it is easy to observe how well the function works by looking at the robot because it will not do much on its own without the other parts of MCL below!

3.2 Normalising and Resampling (16 Marks)

Following the lecture notes, implement the final two steps of MCL which take a particle set whose weights have been adjusted by the measurement step, normalise it so that all weights add up to 1 and then resample it to generate a new particle set.

The normalisation part is straightforward. You need to add up all the weights in the unnormalised set, and then divide the weight of each by this total.

In the genetic algorithm interpretation of MCL, resampling is where strong, successful particles get to reproduce and weak ones die out! Given a set of N particles with varying but normalised weights as the starting point, resampling generates N new particles whose weights are all equal to $1/N$ but whose spatial distribution now represents the previous weighted distribution.

Technically, what happens is that each new particle is a copy (in terms of (x, y, θ)) of one of the old particles. For each new particle, the old particle to copy is selected randomly by sampling from the old set according to their weights. i.e., if one of the old normalised particles had weight 0.2, then a fraction 0.2 of the new particles on average should be copies of this one. Obviously we will get many copies of the strong particles, and few or in many cases no copies of the weak particles.

The random choosing of particles to copy proportionally to their weights can be simply achieved by building a cumulative weight array and then generating random numbers between 0 and 1. i.e. if you have N particles, construct an array where the value of entry n is $\sum_{i=1}^{n} w_i$: the probability of this or any previous particle. Then to make a new copied particle, generate a random number between 0 and 1 and see where it intersects with the values of this array. e.g. if particle $n = 15$ has high weight $w_n = 0.30$, it may cover the range 0.15 to 0.45 in the cumulative probability array. Every time a random number in this range is chosen, a copy is made of this particle. You should use a temporary array space to generate the new particle set, then copy it back to overwrite the main arrays once resampling is completed. Finally now, you are at the end of the algorithm and can go back to robot motion and the step 1 (motion prediction) of MCL.

To assess this part, I would like to see the whole MCL algorithm running on your robots in the test course. Normally we can assess this section and the next part (waypoint navigation) together. I want watch the evolution of the particle set on a remote display using the display functions in the Pi to web interface which we introduced last week and see the correct behaviour. An ideal way to prove that your measurement function, normalising and resampling are working during development would be to artificially worsen your odometry model and motion prediction so that on its own it doesn't work well at estimating distance and you get a large particle spread. When moving towards a wall with the sonar

and full MCL enabled, however, you should still be able to achieve accurate distance estimates and a well clustered particle set in the movement direction.

3.3 Waypoint-Based Navigation While Localising (8 marks)

Last week you implemented a navigation capability for your robot to move to a sequence of pre-programmed waypoints.

The proof of good localisation is if it can be used to achieve accurate navigation. You can make a point estimate of the current position and orientation of the robot by taking the mean of all of the particles (note that this means to take individually the means of the x, y and θ components of the particles — and watch out for angular wrap-around in the angular part):

$$\bar{\mathbf{x}} = \sum_{i=1}^{N} w_i \mathbf{x}_i \ .$$

Based on this estimate you can then control the robot to move towards a target location. You should now control your robots to accurately navigate through a sequence of waypoints in the mapped area.

Here is the list of waypoints. These are (x, y) coordinates in cm that your robots should pass through in sequence, and we will mark out their locations on the floor of the real course. At the start, your robot should be placed first precisely with its centre at waypoint 1 and oriented so that it faces forwards along the x axis. Your program should then be started and the robot should drive to waypoints 2, 3, 4. . . and eventually to waypoint 9 which is back in the same place as waypoint 1. I will judge how accurately each robot passes through the waypoints (which I will mark out on the map).

1. (84, 30)
2. (180, 30)
3. (180, 54)
4. (138, 54)
5. (138, 168)
6. (114, 168)
7. (114, 84)
8. (84, 84)
9. (84, 30)

You will see if you plot these out that a robot that is localising well will not need to do any obstacle avoidance to follow this sequence of landmarks. Just moving in straight lines between them and turning on the spot to point to the next landmark will be enough. Your robots should move and follow this path while performing continuous Monte Carlo Localisation.