Robotics
Week 4 Practical: Probabilistic Motion and Sensing

Andrew Davison
ajd@doc.ic.ac.uk

1 Introduction

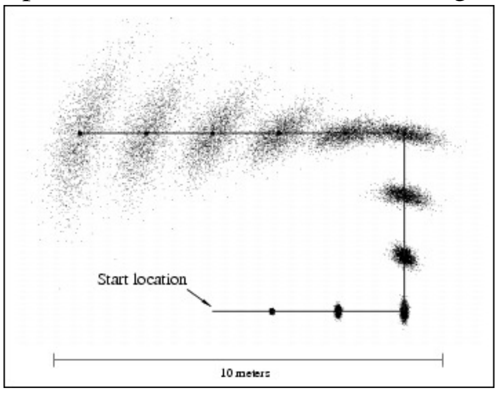
This week we will revisit robot motion and sensing but now with a probabilistic viewpoint, understanding how to model and reason about uncertainty. This practical lays the groundwork for next week's session where these components will be brought together into a full algorithm for Monte Carlo Localisation, a probabilistic localisation filter.

This practical will be ASSESSED. There are **30 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 one of the lab assistants AT THE START OF THE PRACTICAL SESSION ON THURSDAY NEXT WEEK, 8th NOVEMBER. No submission of reports, code or other materials is required. We will assign marks based on our judgement of whether each objective has been successfully achieved, and give you feedback and tell you your marks on the spot. At the end of term I will create a dummy coursework on CATE which will be used to fill in your total coursework mark.

2 Objectives

2.1 Representing and Displaying Uncertain Motion with a Particle Set (18 Marks)

In the first practical we saw clearly that even after careful calibration, a robot's estimate of its motion based on odometry measurement will always have uncertainty. In today's lecture I explained how in probablistic robotics, one way represent this uncertain estimate and how it changes over time is via a particle distribution. The figure below shows an example particle distribution evolving over time to represent the growing uncertainty in the position of a robot navigating using only odometry, though with a different scale and motion pattern from what we will be using today.



Write a program which represents the uncertain motion of your moving robot via a particle distribution to achieve a similar result, and displays the results in the form of a cloud of particles drawn in real-time using our Pi web interface (see below).

You will need to demonstrate this program to us next week. Program your robot to follow a **full 40cm square trajectory as in Practical 1** using position control, though **make your robot move in 10cm steps, stopping after each movement or turn**. We want to see the particle distribution on screen, updating in real-time after each movement. The particles should move after every robot movement step, and spread out gradually to represent its growing uncertainty during this motion. No outward looking sensors are to be used in this part so the uncertainty should always get bigger after each step.

Note that you should clear the screen each time you redisplay the particles, so that we just see the new state of the particles rather than showing the whole history as in the figure above.

In preparation for next week your robot should run on the lab **carpet** rather than paper, and you may need to alter your tuning and calibration parameters slightly to get accurate motion on carpet.

At the start of motion, the set of particles should all be initialised to the origin $(x = 0, y = 0, \theta = 0)$. Each time the robot stops, you should update the position of the particles using the equations below:

- After a straight-line period of motion of distance D:

$$\begin{pmatrix} x_{new} \\ y_{new} \\ \theta_{new} \end{pmatrix} = \begin{pmatrix} x + (D + e)\cos\theta \\ y + (D + e)\sin\theta \\ \theta + f \end{pmatrix}$$

- After a pure rotation of angle angle α:

$$\begin{pmatrix} x_{new} \\ y_{new} \\ \theta_{new} \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta + \alpha + g \end{pmatrix}$$

Here e, f and g are noise terms, which are generated for each particle separately by sampling random numbers with zero mean and an appropriate standard deviation from a Gaussian distribution. Adding a small *different* random amount onto the position of each particle like this will cause spreading like you see in the diagram.

The best way to adjust the standard deviations of the Gaussians for each of e, f and g which you sample is by running the whole program for the square trajectory and observing the total amount of particle spreading which occurs. In principle, this should agree approximately with the amount of uncertainty you think your robot has in its motion. So if you robot can complete the square trajectory and get back to its starting point with around a 3cm standard deviation total error, look for this amount of spread in the particles at the end of the motion. Note that it usually makes sense to err on the side of too much uncertainty in the way that the particles spread out because when the robot is out in the real world and especially running on carpet it may be less precise than during the experiment we did on paper two weeks ago. We would expect that the particles spreading out to around 5cm standard deviation over the whole square motion would be reasonable for most robots. The following sections have further details.

2.1.1 Web Interface and Displaying Output

You may have already been using the web interface with your Raspberry Pi Robots which makes it much easier to interact with programs running on the Pi from any device on the college network (PC, tablet, etc.). We are providing a web server, implemented in node.js, which you should install on your Raspberry Pis. You can then connect to this webserver from your PC or device. Note that we have also created an easy web-based way to query your Pi's current IP address which provides immediate click-through to your Pi's web interface when you provide the MAC address.

You can copy programs to the Pi through this interface, run them and see text output in the browser. Also, it enables real-time 2D graphics output from your Pis via special formatted Python print statements. You will still write programs in Python in the usual way, and can still copy them to your Pi and execute them over scp/ssh as before.

For full instructions on how to install the web server, access it from a PC or device, copy and run Python programs and use the special drawing commands, see: `http://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/webviewer.txt`

2.1.2 The Particle Set

The set of particles, each of which has values for $\mathbf{x}_i = (x_i, y_i, \theta_i)$ and weight w_i, can be stored in pre-allocated Python arrays of length `NUMBER_OF_PARTICLES`. A suitable initial value for `NUMBER_OF_PARTICLES` this week is 100. In this week's practical, the weights w_i of the particles will not be important and should just be set to 1/ `NUMBER_OF_PARTICLES`. The weights become important next week when we incorporate sonar measurements.

The web interface provides a simple way to display a set of points on screen to represent the positions of a particle set. You should experiment to find a suitable scale so that the particle set stays within the bound of the interface window at all times.

2.1.3 Generating Random Numbers

Random numbers for use in the particle movement can be easily generated in Python using the `random` module. See `http://docs.python.org/2/library/random.html` for full documentation. In particular, this week you will need to generate random numbers sampled from a Gaussian distribution. The function `random.gauss(mu, sigma)` generates a random number sampled from a Gaussian distribution with mean `mu` and standard deviation `sigma`.

2.2 Waypoint Navigation (6 Marks)

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:

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

Based on this estimate you can then control the robot to move towards a target location. Note that this week, this mean estimate is just calculated from odometry so is not very interesting — as the particles spread out as uncertainty increases, their mean will not change — but with this machinery in place you will be set up for next week when the particle distribution will also be affected by sonar measurements.

Provide your robot with the capability to perform position-based navigation via simple path planning through a set of waypoints. A waypoint is an (W_x, W_y) position relative to the world coordinate frame W through which the should pass. Write a function `navigateToWaypoint(float X, float`

`Y)` which, given the robot's current estimated location (x, y, θ), drives it to the waypoint at (W_x, W_x) specified in metre units. Most straightforwardly this will be achieved by first a turn on the spot through the appropriate angle and then a straight forward motion of the right distance. Refer back to the lecture notes from week 2 on position-based path planning, and please use the same 2D coordinate frame convention defined in the figures in that lecture such that the robot starts at $(x, y, \theta) = (0, 0, 0)$ with its forward direction aligned with the x-axis, the y axis points left and positive θ representing a left turn.

Prove the operation of your path planning function to use by demonstrating a short interactive program where a user is asked to enter (W_x, W_y) coordinates in metres which the robot will then move to before asking for more coordinates. (Again, bear in mind that at this stage your robot is navigating only based on odometry so there will be inevitable drift in its position over long motions; but with this machinery in place we will be ready to implement a full MCL localisation and navigation system next week.)

2.3 Sonar Investigation (6 Marks)

The sonar is the crucial outward-looking sensor which we will use to make contact with the mapped world in Monte Carlo Localisation next week.

This week we will make some initial quantitive investigation of its properties. We will attempt to calibrate the sonar by comparing the values it returns with *ground truth* obtained from measurements with a ruler or tape measure (there are some around the lab and I should have some spares). Set up your sonar sensor with a simple program to read continuously and report depth values to the screen (look at the example program `ultrasonic_example.py` on your Raspberry Pis to see how to do this).

2.3.1 Sonar Calibration Questions to Answer

On paper, prepare brief answers to these questions which we will ask you about at the assessment:

1. When placed facing and perpendicular to a smooth surface such as a wall, what are the minimum and maximum depths that the sensor can reliably measure?
2. Move the sonar so that it faces the wall at a non-orthogonal incidence angle. What is the maximum angular deviation from perpendicular to the wall at which it will still give sensible readings?
3. Do your sonar depth measurements have any systematic (non-zero mean) errors? To test this, set up the sensor at a range of hand-measured depths (20cm, 40cm, 60cm, 80cm, 100cm) from a wall and record depth readings. Are they consistently above or below what they should be?
4. What is the the accuracy of the sonar sensor and does it depend on depth? At each of two chosen hand-measured depths (40cm and 100cm), make 10 separate depth measurements (each time picking up and replacing the sensor) and record the values. Do you observe the same level of scatter in each case?
5. In a range of general conditions for robot navigation, what fraction of the time do you think your sonar gives garbage readings very far from ground truth?

Acknowledgments

Thanks to Lukas Platinsky, Jan Jachnik, Robert Lukierski, Jacek Zienkiewicz and Duncan White for huge help in preparing the software, hardware and documentation for this practical.