Robotics
Practical 4: Probabilistic Motion

Andrew Davison
a.davison@imperial.ac.uk

1 Introduction

This week we will revisit robot motion and but now with a probabilistic viewpoint, understanding how to model and reason about uncertainty. This practical lays the groundwork for Practical 5 where we will bring this together with probabilistic sensing to implement a full algorithm for Monte Carlo Localisation, a probabilistic localisation filter that allows a robot to localise accurately and without drift within a mapped area.

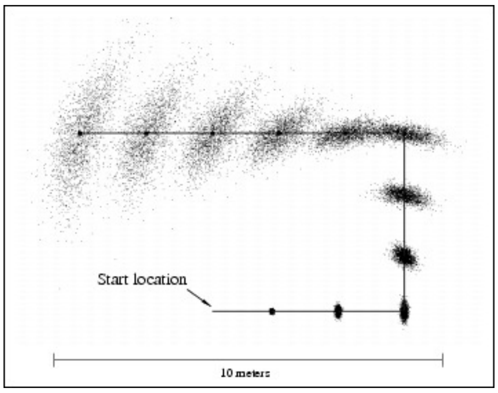
This practical is one of three this term that will be ASSESSED. There are 30 marks to be gained for completing the objectives defined for today's practical, out of a total of 100 for the coursework mark for Robotics over the whole term. You should work in your groups on the exercises. Assessment will take place in your groups via short videoconference demonstrations and discussion of your robots and other results with me or one of the lab assistants DURING THE LIVE PRACTICAL SESSION NEXT WEEK, on Friday 18th February.

2 Representing Uncertain Motion with Particles (15 Marks)

Start by downloading and running this new scene file:
`https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/practical4.ttt`
This looks initially similar to the what we had in Practical 2; the robot, not using sensors, follows a series of steps of motion consisting of turns and forward movements. The robot should be quite well calibrated and move accurately on the floor. Also you will see that we have added a feature (which some groups implemented themselves last week) where the robot slows down as it approaches the end of a step, making it more accurate than stopping suddenly.

In Practical 2 we saw clearly that even after careful calibration, a robot will never do the same thing every time, because of small factors such as the bumpiness of the floor which are difficult to model. If it makes an estimate of its motion based on odometry, therefore, this estimate will always have uncertainty relative to its true motion.

In this week's lecture I explained that in probablistic robotics, one way to 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.



2.1 Particle Motion Prediction and Display

Your first aim this week is to add code to the program to represent the uncertain motion of the moving robot via a particle distribution, drawn in real-time using within the CoppeliaSim simulator. The particles should move once per movement step of the robot; so when the robot reaches the end of one step we will see the particles jump to represent the new uncertain position of the robot.

You will need to demonstrate this program to us next week. Program your robot to follow a **full 1m square trajectory as in Practical 2, 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.

So, in your demonstration we should see, happening at the same time, your robot driving on the floor to complete a square motion, and then also a real-time particle display which updates after each movement step. Stick to the coordinate system and square motion we used in Practical 2, where the x coordinate is forward and y is left at the start of the motion, and your robot should make left turns.

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, as explained in lectures:

- 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.

You will need to find suitable standard deviations for the Gaussian distributions you use to sample e, f and g from. The best way to do this is to try some initial values, and then run the program for the square trajectory and observe th particle spreading which occurs. e, f and g represent different things, so each should have a different standard deviation, though all should have zero mean as long as your robot is well calibrated. You can adjust each one by trial and error until it looks right.

In principle, the amount your particles spread out should agree with the amount of uncertainty you think your robot has in its motion. So if your 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 by looking at your graphics display (you can judge by eye the amount of spreading relative to the size of the 50cm floor tiles).

Note that it usually makes sense to err on the side of a bit too much uncertainty in the way that the particles spread out because when the robot is out in the real world over a wide range of conditions it may be less precise than during controlled experiments. We would expect that the particles spreading out to around 5–10cm standard deviation over the whole square motion would be reasonable.

The following sections have further details.

2.1.1 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 arrays of length `N`, and we have included simple code for this in the starting program. A suitable initial value for `N` 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/ N`. The weights become important next week when we incorporate sonar measurements.

2.1.2 Displaying Output

When you run our new scene file, you will see at the origin a small point with coordinate axes. This is a 'dummy', or in fact 100 dummies drawn on top of each other at the moment, which is what we will use in CoppeliaSim to display particles. The nice point about dummies is that they can conveniently represent both the position and angular orientation of a particle. Look for these commands in the program:

```
sim.setObjectPosition(dummyArray[i], -1, {0,0,0})
sim.setObjectOrientation(dummyArray[i], -1, {0,0,0})
```

The `sim.setObjectOrientation(dummyArray[i], -1, {0,0,0})` command sets the orientation of a dummy, and we can use this to display the orientation of a particle. The three numerical arguments (currently zero) are the Euler angles describing the dummy's orientation. The last one of these is its horizontal rotation (around the z axis) and matches up to the `theta` orientation of our robot.

In the starting program we have given you, these dummies don't move from the origin. You need to write code to update the particle coordinates and move the dummies after every movement step.

2.1.3 Generating Random Numbers

Gaussian-distributed random numbers for use in the particle movement can be easily generated in Lua using the `gaussian(mean,variance)` function we have included at the top of the scene file (if you are interested, this function implements the 'Box-Muller Transform' method for generating Gaussian random numbers from a uniform random generator). Note that this function uses variance as its argument, where variance is the square of standard deviation.

2.2 Waypoint Navigation (15 Marks)

Before working on this second part of the practical, save a working scene file for the first part under a different filename so that you have it to show in the assessment — in this part you will be making some changes to the structure of the program.

In this part of the practical, your aim is to implement waypoint navigation, such that the robot can follow commands which take the form not of incremental motion commands such as forward and turn, but instead a sequence of position coordinates (waypoints) in world coordinates which we would like the robot to reach in turn. You should change your program so that the robot can follow a series of steps like this:

```
stepList[1] = {"set_waypoint", 1, 0}
stepList[2] = {"turn"}
stepList[3] = {"stop"}
stepList[4] = {"forward"}
stepList[5] = {"stop"}
stepList[6] = {"set_waypoint", 1, 1}
stepList[7] = {"turn"}
stepList[8] = {"stop"}
stepList[9] = {"forward"}
stepList[10] = {"stop"}
stepList[11] = {"set_waypoint", 0, 1}
stepList[12] = {"turn"}
stepList[13] = {"stop"}
stepList[14] = {"forward"}
stepList[15] = {"stop"}
stepList[16] = {"set_waypoint", 0, 0}
stepList[17] = {"turn"}
stepList[18] = {"stop"}
stepList[19] = {"forward"}
stepList[20] = {"stop"}
stepList[21] = {"repeat"}
```

Here the coordinates in the `set_waypoint` steps are in world (x, y) coordinates in metre units, and one of these steps is called the robot should set new targets for motion from its current position, consisting of first a turn on the spot to point to the desired position and then then a forward motion to reach it. Then it should actually execute those movement steps during the `turn` and `forward` steps. We looked at the formulae you need for waypoint navigation in the lecture.

Ideally your robot should always make the shortest turn possible, which will be less than 180 degrees left or right. Refer back to the lecture notes 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.

In order to calculate the motion needed to get to a waypoint, a robot needs to have an estimate of its current location, and we will use particles to get this estimate. You should adapt your code from the previous section so that whenever the robot makes a turn or forward movement, the particle positions and orientations are updated appropriately to reflect that motion.

One important detail is that now that your robot could move through variable step sizes, the amount of uncertainty that we add to the particles should depend on the distance moved. The correct way to do this is to scale the **variance** of the Gaussian distributions we sample from to be proportional to the linear or angular distance moved — variance is additive, so for instance two 1m steps will cause the same spread as one 2m step. So if you have found good values for the variances of e, f and g for the 1m and 90 degree motions in the first section, you can scale those variances dynamically according to the size each motion the robot needs to make.

For path planning, we need a point estimate of the position of the robot. 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 accurate. The mean position of the particles will follow the desired motion of the robot, rather than the drifting motion it actually has due to its uncertain motion; and therefore the movements calculate to move to waypoints will also not be very accurate. So you might wonder what the point of using particles for waypoint navigation is. The answer is that with this machinery in place, you will be ready for next week, where we will be using sonar measurements to update the particle distribution to better estimates than can be achieved with odometry only, and waypoint navigation using this method will work much better.

During the assessment of this part, we will also ask you to change some of the waypoints to different values and run your program again, to make sure that your robot can make the correct motions to reach any (W_x, W_y) waypoint. We know that the accuracy of the actual movement of the robot will not be very high, and that is OK; we don't want you to work on improving that through further calibration in this practical. We will just be checking that the robot makes sensible motion calculations relative to its estimated location.