Robotics

Week 3 Practical: Sensors

Andrew Davison
a.davison@imperial.ac.uk

1 Introduction

In this practical we will use a depth sensor (sometimes called a range sensor, or proximity sensor) with reactive and feedback control to implement some simple but powerful robot behaviours.

This week's practical is UNASSESSED. There are several tasks laid out in the following sections, and it will be very useful for your understanding of the examinable material in the course to attempt them, but we will not have a formal assessment session and achievement in this practical will not count towards your overall coursework mark for Robotics.

2 Depth Sensing

Download the CoppeliaSim starting file for this practical from:
`https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/practical3.ttt`,
where you will see the same differential drive robot that we used in Practical 2, but now placed in a 5×5m arena surrounded by walls and containing some obstacles. The floor in this arena is a randomly generated height map as we used last week.

This week we will be using the outward-looking depth sensor mounted at the top of the robot, called 'turretSensor' in the scene hierarchy. This is a single beam sensor which measures the distance to the closest obstace in a straight line. In its 'perfect' simulated form it gives a very precise measurement, (and is therefore even better than the very accurate measurements from a real laser range-finder). We will be adding some simulated noise to its measurements in this and later practicals which make it more like a sonar sensor in performance.

The robot's depth sensor is horizontally mounted on a rotating turret which can be rotated using the turretMotor joint. The origin of the sensor is above the centre of the robot.

If you run the program, you will see the robot start to drive forward. Two graphs will pop up: one which shows the measurements recorded by the depth sensor, which is facing forwards, and one which shows the robot's X/Y trajectory like last week. The program doesn't do anything sensible yet, and the robot will crash into the wall.

Note that I added print statements and changed the variable `UPDATE_GRAPHS_EVERY` to 1 to make it easier to see what is going on initially; these can slow your programs down. Once you understand things, in particular change `UPDATE_GRAPHS_EVERY` back to 20 (which updates the graphs less often) or remove the graphs altogether, and comment out print statements and your program will run faster.

3 Objectives

3.1 Reactive 'Random Bounce' Control

First, we will use the depth sensor as a simple way to detect when the robot is getting too close to an obstacle, by continuously checking the measurement it obtains against a minimum safe distance of 25cm. Here the sensor is acting very much like a physical 'bump' detector, but it will usually detect a bump before the robot actually hits an obstacle (though not always if the angle to an obstacle is oblique and the side of the robot crashes).

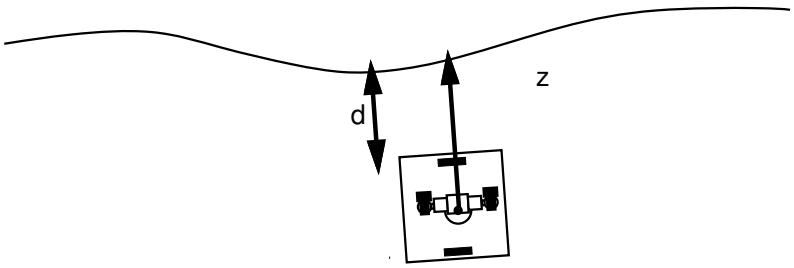
Write a simple program which will drive your robot straight forward at a fixed speed until the forward-looking depth sensor detects a 'bump', which means a measurement of less than 25cm. When this happens the robot should stop and make a rotation on the spot through a random angle. It would make sense for this angle to be greater than 90 degrees (either left or right) so that the robot is pointing away from the obstacle afterwards. The robot should then resume moving forwards and repeat. Overall, the behaviour will be a 'random bounce' pattern, where the robot moves in straight lines in between collisions where it 'bounces' in a random direction. This behaviour is the way that the earliest robot vacuum cleaners worked. If you run the behaviour for a long time, the robot should do a reasonable job of finding its way into most parts of the scene, though with a lot of randomness and repetition — you can check this by looking at the trajectory graph.

Have a look at the current code in the starting file for this practical, where we have set up a suitable state machine in `sysCall_init()` with a suitable series of steps in `stepsList` to implement random bounce. Note that the 'repeat' step now does not have the same meaning as we used last week in the robot motion practical of resetting the robot to the origin and generating a new random floor; now we just use repeat to mean to go back to the beginning of the steps list.

In `sysCall_actuation()` (remember that this is the callback function which is called once every tick of the main simulation), you will need to fill in code to actually make it work. The first part of the code here checks whether we have started a new step, and should be used to set up suitable actuation commands and targets. The second part handles the current ongoing step and should be used to check whether that step has completed, in which case `stepCompletedFlag` should be set to true so that we move onto the next step.

You shouldn't have to add much code to get random bounce to work.

3.2 Wall Following



Next, implement a wall following behaviour which allows the robot to use continuous proportional control to follow smoothly along a wall at a desired distance such as 25cm. We would expect this behaviour to allow a robot to follow along a wall which is straight or has smooth turns when starting from a position which is about the right distance from the wall.

Assuming for now that the sensor points left and that we want to drive along a wall at distance 25cm,

then when the sensor measures exactly 25cm we want to stay at that distance and therefore should set both motors to the same speed to drive straight forwards. If the measured distance is more than 25cm, we want to turn towards the left, and therefore the right wheel needs to speed up while the left wheel slows down. For proportional control, we should set the difference between the speeds of the wheels to be proportional to the negative 'error' between the measured distance and the desired distance of 25cm. The average speed of the two wheels should stay at a pre-chosen constant value (so we are always increasing the left wheel speed by the same amount that we decrease the right wheel speed and vice versa).

The robot's turret should be rotated such that the sensor is looking sideways at a fixed angle relative to the robot's forward direction. You can use position control to set the angle of the turret motor to a new fixed angle using the `sim.setJointTargetPosition` command. You could rotate it to the left by 90 degrees, or try something a bit less like 70 degrees which might help to get smooth wall following behaviour.

If z is the sensor measurement at each step, and d is the desired distance, v_C is a constant base velocity and K_p is a proportional gain constant, you can achieve smooth and symmetric wall-following behaviour by setting desired velocity of the left and right wheels as follows:

$$v_R = v_C + \frac{1}{2}K_p(z - d)$$

$$v_L = v_C - \frac{1}{2}K_p(z - d)$$

With a well-chosen gain value, this simple law should give smooth behaviour because the robot will make rapid turns when there is a big difference between the measured and desired distance, and gradual turns when it is almost at the right distance. The robot should be able to cope with walls which are somewhat curved as well as straight ones; though probably not sharp turns such as right angles.

4 Combined Behaviour

If you have time, you could try putting the two behaviours together into a single combined pattern which is similar to some later robot vacuum cleaners, where the robot will intersperse some periods of wall following between random bounces. There are several ways you can try this, but one idea would be to have this pattern:

1. With the depth sensor pointing forwards, drive forward until a bump, and stop.
2. Rotate the robot about 90 degrees in a known direction, e.g. right.
3. Rotate the turret to face left (i.e. back towards the obstacle), and start wall following.
4. Stop after some predetermined distance or time has been covered.
5. Make a random rotation to the right to hopefully point away from the obstacle.
6. Return the turret to point forwards, and return to step 1. to drive forwards again.

You could randomise the wall following so that it doesn't happen on every bump, or randomise the period spent wall following.