# Robotics
## Practical 2: Accurate Robot Motion

Andrew Davison
a.davison@imperial.ac.uk

## 1 Introduction

In this practical you will be programming and modifying a wheeled robot to move accurately based on the differential drive design, and then investigating its motion experimentally.

This practical is one of three this term that will be ASSESSED. There are 35 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 live or 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 5th February.

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 whether you can clearly explain the work you have done, and tell you your mark immediately. All members of a group will receive the same mark by default.

(Note that a dummy coursework exercise has been created on CATE. You don't need to do anything about this yet, but at the end of term we will ask you to use this to register the members of your group and we will assign your final coursework marks there.)

## 2 Wheeled Robots and Accurate Motion

In this practical we will be investigating how wheeled robots move and estimate their motion based on odometry, which means measuring how their wheels turn. We will be testing robots with the differential drive wheel configuration, and running them in a simulation which includes a random bumpy floor which is a model of the conditions that real robots face, whether driving outdoors for instance on gravel, or indoors on carpet. We will learn how to calibrate robots for repeatable motion, but also that even the best calibrated robots are affected by external unmodelled factors which mean that their motion is uncertain. This uncertainty can be quantified. This motivates the work in the coming weeks on using outward-looking sensors, and especially probabilistic methods for localisation and navigation.

## 3 CoppeliaSim Setup and Starting Scene File

I hope that you have all had the chance to complete the first week's exercise to familiarise yourself with CoppeliaSim, and have looked at some of the more general documentation on CoppeliaSim and Lua. To remind you, support is always available during the live practical sessions on MS Teams, or at any other time in the week via Piazza.

We have provided a starting scene file for this practical, which is available at `https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/practical2.ttt`
(this file is also available from the schedule part of the course website). Download it and open it in CoppeliaSim.

In this scene you will see a simple differential drive robot, similar to bubbleRob from the CoppeliaSim built-in tutorials. The robot has two wheels attached to the robot body using rotational joints which act as motors. It also has two 'sliders' at the front and back which are passive smooth balls that slide over the floor and keep the robot balanced. Note that the robot has been designed such that the horizontal centre of the robot is exactly between the wheels, so that the robot is able to turn on the spot without translating. The robot also has a disk-shaped 'turret' and depth sensor mounted on the top, but we will not be using these in this practical.

When you run the scene, the robot drives forward some distance, then stops and makes a rotation on the spot, and then is returned to its starting position at the origin of coordinates to repeat the motion.

You will see the trajectory the centre of the robot takes each time plotted in the graph that pops up in terms of its x and y coordinates plotted against time. Press the 'X/Y graph' button to see this in the more natural form of y plotted against x. The scale here is in metres.

In a second graph, we plot the joint angles of the robot's joint motors against time (the angular unit here is radians).

Stop and run the program several times, and you will notice that the robot makes slightly different movements each time. This is because each time the program runs, a randomly generated bumpy floor is added to the scene to simulate the real surfaces that robots would have to run on.

## 4 Robot Programming, Simulation and the CoppeliaSim API

As you saw in Practical 1, CoppeliaSim is fundamentally a world simulator, where a scene that we set up will evolve through time under simulated physics. We write a program to control a robot as an 'unthreaded child script' attached to a scene object, which defines callback functions which are called by the main simulation. The code in our child script implements the 'intelligence' of a robotic agent.

Our goal is to use CoppeliaSim to learn about real world robotics, so we need to carefully think about what aspects of a simulation should be accessible to the program controlling a real robot.

Remember our definition of a robot in Lecture 1: it is a physically embodied AI agent which can sense and act, and must join those two aspects via computation serving as intelligence.

How can a robot sense the world? Via receiving numerical values from its on-board sensors.

And how can a robot act on the world? Via sending numerical values to its actuators, such as motors.

So while the CoppeliaSim API allows almost any aspect of the simulation to be altered from a script, the things that we allow our programs serving as a robot's intelligence to do should be limited to operations which match up to a real robot's sensing and actuation: reading the values of sensors and sending commands to actuators.

Some of the things which we shouldn't do, because they would not be possible in the real world, are for instance directly reading the coordinates of an object from the API (instead of trying to estimate its position from sensor readings); or 'teleporting' by simply setting the location of a robot or object to a desired position, rather than trying to reach that position via actuation. The golden rule is to imagine that our agent program could be seamlessly ported between the simulation or a real-world robot and work in either case.

## 5 Callback Programming in CoppeliaSim and using a State Machine

In this practical we want to make our robot move in controlled steps, which might take a few seconds each.

In CoppeliaSim, the physical simulation of the scene runs at a certain fixed rate (with each step corresponding to a time-step dt), and our programs to control a robot are implemented as 'unthreaded child scripts'. Within a child script, we define callback functions which will be called by the simulation. In particular, the function `sysCall_init()` in a child script is called just once when the simulation is started, and `sysCall_actuation()` is called once per time-step.

Now that we want to program a robot to move in steps, such as 'move forward 1m' or 'rotate left 90°', each of which will usually be much longer than a single simulation time-step, we have to think about how to implement that with the callback style of programming. What we will **not** be able to do is define and call a sequence of standard imperative programming style functions such as `driveForward(distance)` and `rotateLeft(angle)`.

Instead, we need to use `sysCall_init()` to set up variables which represent a state machine keeping track of which step the robot is on, and the status within those steps, and then update these variables within `sysCall_actuation()` to keep track of the progress between states.

This is what we do in the child script we have provided for this practical. In `sysCall_init()` we define a variable `stepCounter=0` and an array `stepList` which defines the sequence of actions we want the robot to take, each of which is to drive or rotate by some amount. We also define variables `motorAngleTargetL` and `motorAngleTargetR` which will serve as the motor joint target angles for a particular step.

Now have a look at `sysCall_actuation()`. At the high level, we have a loop that looks like this:

```
if (targetForCurrentStep == achieved or currentStep == 0) then
    -- transition to next step
    -- set new targets
end
-- set control commands to continue current step
```

## 6 Some Points about Programming in Lua

There is not a huge amount of programming to do this week, but you should be continuing to get familiar with Lua.

A reminder that if you are new to programming in Lua, there is a good introduction at: `https://www.lua.org/pil/contents.html`, though the best way to learn in general will by trying to understand and modify our sample programs.

The syntax for 'for', 'if', and so on is a bit different from other languages, and the 'end' keyword is used to mark the end of these statements, but this should be easy to get used to.

There is only one type of numeric variable in Lua, which can handle both integers and floating point values.

Variable names that haven't been assigned a value will by default have the value 'nil'. So a common error when you run a script might be something like 'attempt to compare number with nil', which probably means that one of the variables in an 'if' statement hasn't been properly declared.

Arrays in Lua can be implemented using 'tables', where the index can be a variable of any type (e.g. a number or a string).

When indexing loops and arrays in Lua, it is usual to use the convention to start with index 1 (rather than 0 as in most programming languages). Standard libraries in Lua are written in this way.

## 7 Practical Assessed Objectives

### 7.1 Understand and Explain Our Robot Program (4 Marks)

Have a look through the child script which is attached to myRobot to understand how it works, and be ready to explain this to us briefly in the assessment.

At the top of the script there are some helper functions
`gaussian()`, `resetBase()`, `createRandomBumpyFloor()` we have defined for use in this practical which are not really part of the robot's main control functionality and you will not need to modify these. In `createRandomBumpyFloor()` you can see that a random height field is generated for the robot to run on. We have set this up to be invisible in the simulation, but if you want to see what it looks like you can comment out the line under the comment 'Make the floor invisible'.

In `sysCall_init()`, which is called by the simulation just one time at start-up, we set up some data structures. Pay attention to `stepList`, which is a Lua table where we store the list of action steps we want the robot to take. We use the variable `stepCounter` to keep track of which step we are on. Variables `motorAngleTargetL` and `motorAngleTargetR` will hold target joint angles (in radians) for the robot's left and right motors to try to achieve on each particular step. We are using a special 'stop' step to make sure the robot is stationary before each new movement step, which helps accuracy.

`sysCall_actuation()` is the callback function which the simulation calls once per 'tick' time dt of the simulation. This function monitors which step the robot is currently in, and checks whether the motor joint targets for that step have been achieved (the details of this check have been separated out into function `isCurrentTargetAchieved()`). If they have, the program moves onto the next step and sets new targets.

Something special happens at a 'repeat' step, which we use for the purposes of the experiments we want to do in this practical and is not a realistic physical robot action. Having marked the robot's position with a 'dummy' marker for reference, the robot is teleported back to the origin, a new random bumpy floor is generated, and `stepCounter` is reset to zero.

Finally in this function, we call `setJointTargetVelocity()` commands to actually make the robot move as required in the current step.

### 7.2 Distance and Rotation Calibration for Accurate Driving in a 1m Square (10 Marks)

You will see that when you run the current program, the distance that the robot moves and the amount it rotates do not correspond to the desired movement steps of driving forward one metre and rotating left by 90 degrees.

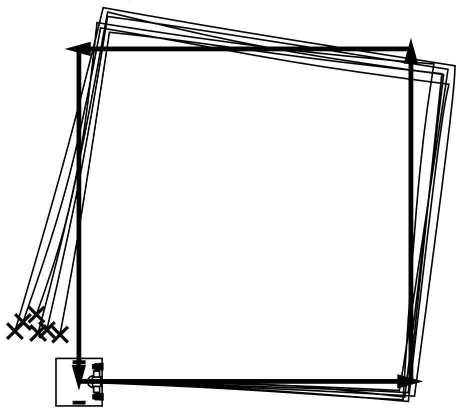
You should adjust and possibly add calibration constants to the program to **calibrate** the robot's movement to be as accurate as possible. The best way to achieve this is by trial and error, by making use of both the 3D view of the robot as it moves and the trajectory graph it plots. We remind you that the chessboard floor tiles in the main simulation view are each 0.5m squares.

Once you have a reasonable initial calibration, add extra steps to the robot's plan to make it execute a full square 1m trajectory consisting of four 1m forward motions separated by 90° left turns (including a final turn at the end so that the robot finishes facing in the original direction. You can then refine your calibration further.

It is easier to calibrate rotation by seeing how it affects the whole square trajectory than by looking at how it rotates one time. The aim is that the centre of the robot, the point between its wheels, should ideally exactly trace out the square. Ideally the robot will return to the origin after the full square. Due to the random bumpy floor, however, when run multiple times the robot will not move in the same way every time. Your goal in calibration is to ensure that the **average** motion of the robot is as close as possible to the ideal 1m square.

## 7.3 Experiment to Measure Robot Accuracy (8 Marks)

Once your robot is calibrated, carry out an experiment to demonstrate its performance. Run your program with the 'repeat' step to draw a dummy at the final position and teleport the robot back to the origin a total of 10 times. On the XY trajectory graph view you will see 10 trajectories appear on top of each other, looking something like this (though note that in this picture we can still see some systematic error which can be improved by calibration).



When the robot has completed 10 squares, pause your simulation and capture a **screengrab** of the XY trajectory graph to record your experiment (though before you complete and record this experiment, read the next section and make sure you have also written code to calculate and print the covariance matrix).

You will see clearly here that due to the unknown floor the robot is very unlikely to follow exactly the same path every time! What is the approximate scatter range of the different outcomes? Hopefully your calibration means that there is little *systematic error* (meaning that the final locations are consistently different from the ideal result with an error in the same direction)? Is there much *scatter*, such that the points are quite spread out from each other? SAVE THE SCREENGRAB TO SHOW AT THE ASSESSMENT.

## 7.4 Calculating a Covariance Matrix (5 Marks)

Add some code to your program to calculate and print out the *covariance matrix*, which is an explicit measure of the scatter range, or uncertainty in the motion of the previous section, once the robot has completed the square trajectory 10 times. Save the 10 sets of $(x_i, y_i)$ final coordinates (where the dummies are drawn) and use them the calculate the statistics here.

With $N = 10$ individual final location measurements $(x_i, y_i)$, the covariance matrix is a matrix of four values as follows:

$$\mathtt{P} = \left[ \begin{array}{cc} \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 & \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}) \\ \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})(x_i - \bar{x}) & \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2 \end{array} \right] ,$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N}(x_i)$ and $\bar{y} = \frac{1}{N}\sum_{i=1}^{N}(y_i)$ are the coordinates of the mean final location. **Use metre units.** Note that this and all covariance matrices are symmetric. As a sanity check, the square roots of the two values on the diagonal should represent the standard deviations of the scatter in the $x$ and $y$ directions respectively — i.e. we would expect a few centimetres after a 1m square motion. We will check to see that the covariances you calculate tie up with your 10 trajectory screengrabs.

### 7.5 Experiment with Modifying Your Robot to Make it More Accurate (8 Marks)

Make a copy of your main scene file to a new file where you can try some new things without changing the main calibrated robot scene file that you will show us in the other sections.

In this new file, try some simple ways to modify your robot with the aim of making it move more accurately, such that it could complete the square trajectory with less 'scatter'. Your modified robot should run on the same randomly generated bumpy floor as in the rest of the practical.

You can use the GUI tools in CoppeliaSim to change the robot's design. Some good ideas could be to change the size of the wheels, the spacing of the wheels, or the mass or friction properties of parts of the robot.

Some other possibilities involve not just changing the robot's physical design, but something about the way its motion is controlled. For instance, you could change the speed of its motion, or give it a different speed profile such that it accelerates more gradually rather than going straight to full speed.

We will discuss these modifications with you in the assessment, and we do not expect that you have spent a long time optimising an accurate design or any extensive experiments. Instead we would just like to see some interesting example concepts, and to discuss with you whether it seems like the robot is moving more accurately.