Worksheet 2
Webots Robot Odometry - 1

Learning Outcome 2

Book Chapter 1

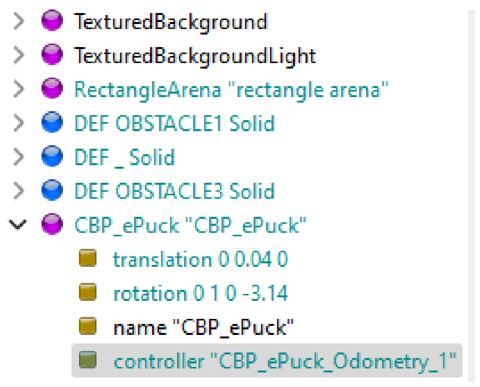
Purpose.

To investigate aspects of odometry using a simulated ePuck robot in Webots.

1. Initialization.

(a) Download and unzip the Webots assets folder.

(b) Fire up webots and select **File > Open World.** Navigate to the **worlds** folder and select **CBP_ePuck_Odometry_1.wbt**. On the top left panel, expand the node with this name and left-click on the **controller** tab. The diagram on the right should help.


TexturedBackground
TexturedBackgroundLight
RectangleArena "rectangle arena"
DEF OBSTACLE1 Solid
DEF _ Solid
DEF OBSTACLE3 Solid
CBP_ePuck "CBP_ePuck"
translation 0 0.04 0
rotation 0 1 0 -3.14
name "CBP_ePuck"
controller "CBP_ePuck_Odometry_1"

(c) In the box underneath, hit **Select…** which will bring up a list of the available controllers. Choose **CBP_ePuck_Odometry_1.c** Then hit **Edit** and the c-code for this controller should appear on the right.

2. Investigating the Controller _Odometry_1

The controller drives the robot in a straight line for the desired distance 0.5m and completes this in 10.0 seconds, so it's speed is 0.05 m/s. The code calculates the *perceived* distance moved by the wheels, and also logs the gps distance which is the *actual* distance moved. We expect the perceived and actual distances to be different, due to wheel slippage etc. The code creates an Octave file which can be used to plot the trajectory.

(a) Check the code. Look for the function which returns the left wheel angle, and the line that converts this to the *perceived* distance. Check out the condition which stops the robot. Does this use actual or perceived distance?

(b) Hit the compile button . You will be asked to reset or reload the world. Choose **Reset**.

(c) Run the simulation . If you are working on your own machine, you may need to disable your anti-virus software. When the robot has stopped, hit pause and then reset the robot to its initial location . The robot should draw you a nice straight line.

(d) Navigate to the **controllers** folder and into your controller sub-folder. You will see an Octave file **Robot.m** Open this with Octave and run it (find the file in Octave's top-left window, right click on it and select **Run**).

(e) You will see a plot of distance (m) against time (secs). The red line is the perceived distance, and the blue is the actual distance. You will see the perceived is larger than the actual. What does this tell you?

(f) Estimate the % error in the final position. You can get these on the Octave command line by typing **gpsX(end)** and **perX(end)**.

(g) Plan a mini-investigation of what changes the error. Here are some suggestions.

(i) Keep the speed constant but change the distance the robot travels. **N.B.** Your code may exceed the max motor velocities, Webots will tell you in its console.

(ii) Keep the distance constant but change the speed.

Can you observe any patterns? Can you explain these?

2. ePuck with Simulated Encoders

Here we shall simulate encoders on the ePuck. The ePuck doesn't have encoders since it uses *stepper motors*, which we shall see later. So why simulate? Well, it gives us another way to understand encoders, in addition to the Parallax hardware.

(a) Let's start with some basic calculations. Parameters for the ePuck are given in the box below. Use these to calculate:

(i) The distance moved (in mm) for one step
(ii) The number of steps needed to travel a distance 250 mm.

ePuck parameters

wheel radius = 20.5mm
axle length = 53.0 mm
steps / rev = 1000

(b) Open the world **CBP_ePuck_Odometry_10.wbt** and make sure that the controller **CBP_ePuck_Odometry_10a.c** is selected and open in the editor window. Look for the code which does the following:

(i) Sets the desired distance to 250mm. You will see this as 0.25 (in metres).
(ii) Calculates the number of steps **nL** needed.
(iii) Sets values for **omegaL** and **omegaR**. These are equal. Why?
(iv) Simulates the encoder **ISR** returning a value of **count**.

(c) Clear the console (right click), compile the controller and run. Some useful data is printed on the console.

(i) How many steps have the motors taken? Does this agree with your calculation in (a)?
(ii) How far does the robot think it has travelled?
(iii) How far has it actually travelled?

(d) So the robot has not gone far enough. We need to get it to take more steps and to correct its speed. Here's how to find the new value for **nL**, where I have used my perceived and actual distances.

```
nL = (int)((float)nL*0.249882/0.242587);
```

Here's how to correct the motor speeds

```
omegaL = omegaL*0.249882/0.242587;
omegaR = omegaL;
```

Make sure you understand how these calculations work. Now insert these lines of code at the point given by the comments and run the robot again. Check the actual distance and the new number of steps.

(e) Navigate to the subfolder containing your controller and open up the Octave file **Robot.m**. Run this and it will give you a nice plot of perceived and actual distance. Note that the actual distance is correct now, and that this is achieved in the desired time of 25 secs.