Constructive Artificial Intelligence
Practical session 7

by Lola Cañamero and Matthew Lewis
m.lewis4@herts.ac.uk

In this practical, you will work towards these goals:

a) Continue to learn about the Webots robot simulator that you will be using for your 2nd piece of course work, set this week. We will practice the use of sensors in the simulator (intialisation and reading). Think of the similarities and differences between sensors in a real robot and sensors in the simulator.

b) Program sample exercises and "building blocks" that you can use as a guide for your coursework. You will do an exercise in which you implement a simple behaviour.

c) A second advanced exercise, will start you implementing a two-resource problem, as seen in lecture 7. If this cannot be completed this week, we will continue to work on it next week.

This practical assumes you have gone through the document **Part 1: Introduction** from last week's practical (practical 6). It will use the code, environment and ePuck robot used in that document.

1 Example: Line Following in C

This example works towards goals (a) and (b) above: the use of sensors in Webots, and the example line following behaviour, which has similarities with the wall following required for your course work.

Reminder: When you revert/reset your World after running, it will revert back to the last saved state, so make sure you save regularly or you risk losing your work.

1. Recall that you created a project folder on your U: drive. Open that .wbt file from Webots.

   First, we need a line to follow. You'll need to make an image file, call it **line.png**, and put it in the **worlds/textures** (which you'll need to create) in your project folder. The height and width of the image file need to be a **power of 2 pixels** (e.g. 256, 1024, 4096; for a 2m×2m arena, try a 1024×1024 pixel image).

   Change the floorTextureUrl property of your RectangleArena to point to your file: "textures/line.png". Also change the floorTileSize to equal the floor Size (so the floor is one big tile).

2. Last time, you added the ground sensor module to the ePuck robot. Check in the Scene Tree that this is still present in your ePuck. If not, then add it (see last week's handout).

3. First, some preliminaries. Add the following includes:

```
#include <webots/device.h>
#include <webots/distance_sensor.h>
```

   And add the following constants to your code:

```
#define TIME_STEP 64
#define GS_COUNT 3
#define BLACK_MAX 500
```

4. As with the LEDs last week, the code will need to do some initialisation. First, add global static arrays to hold the ground sensor tags and the values read by the sensors:

```
static WbDeviceTag ground_sensors[GS_COUNT];
static double gs_vals[GS_COUNT];
```

   Then initialise the array, and the sensors, by adding the following to the initialisation function:

```
  // initialise ground sensors
  int n_dev = wb_robot_get_number_of_devices(); // all devices
  for (i=0; i<n_dev; i++) {
    WbDeviceTag dev = wb_robot_get_device_by_index(i);
    // Check to see if device is a ground sensor
    const char *nm = wb_device_get_name(dev);
    WbNodeType type = wb_device_get_type(dev);
    if(type==WB_NODE_DISTANCE_SENSOR && nm[0]=='g' && nm[1]=='s')
{
      int idx = nm[2]-'0';
      if (idx>=0 && idx<GS_COUNT){ // we expect 3 ground sensors
        printf("Initialising %s\n", nm);
        ground_sensors[idx] = wb_robot_get_device(nm);
        wb_distance_sensor_enable(ground_sensors[idx],TIME_STEP);
      }
    }
  }
```

   To learn about the capabilities of the ePuck, it is a useful exercise to print out the names of all the devices on the robot.

5. Now we can read the ground sensor values. For future flexibility, we will add this as a function:

```
void read_sensors()
{
  int i;
  for (i=0; i<GS_COUNT; i++)
    gs_vals[i]=wb_distance_sensor_get_value(ground_sensors[i]);
}
```

And add a function call in your main loop: `read_sensors();`

Print the sensor values, so you can see their values. Just like real sensors, sensors in Webots are noisy (i.e. the values returned by the sensors fluctuate a lot, even if nothing changes).

6. We can use the values from the ground sensor to set the wheel speeds. Add the following function:

```
void line_follow_motor_values(double *left, double *right)
{
  if (gs_vals[0] < BLACK_MAX) {
    *left = MAX_SPEED * 0.1;
    *right = MAX_SPEED * 0.7;
  }
  else if (gs_vals[2] < BLACK_MAX) {
    *left = MAX_SPEED * 0.7;
    *right = MAX_SPEED * 0.1;
  }
  else {
    *left = MAX_SPEED * 0.5;
    *right = MAX_SPEED * 0.5;
  }
}
```

Replace the code to set the variables left and right in the main loop with a call to the new function, and make sure the call to set the motor speed is still present:

```
    double left, right;
    line_follow_motor_values(&left, &right);
    ...
    wb_differential_wheels_set_speed(left, right);
```

7. Now that you have a line-following robot, set the viewpoint (you can think of it as the "camera") to follow the robot: move the viewpoint close to the robot and set the *follow* property to "e-puck" (the name of the robot).
   If you're having trouble moving the viewpoint to a suitable position, try the values: orientation (x,y,z,a) = (1, 0, 0, −0.3) and position (x,y,z) = (0, 0.5, 0.7).

2 Exercise: Object avoidance

This example works towards goals (a) and (b) above: the use of sensors in Webots, and the example object avoidance behaviour, useful for your course work.

Exercise: Using the infra-red proximity sensors of the ePuck, implement an object avoiding function.

Hints:

- I suggest you start from the line-following code (on StudyNet).
- There are 8 proximity sensors, named “ps0”, … , “ps7”).
- You will need to call enable() on the distance sensors during initialisation.
- I suggest add a function avoid_motor_values() to the class to calculate left & right and call this instead of line_follow_motor_values.
- Print out the values of the proximity sensors to find an appropriate value at which to turn away from an object.

3 Exercise: Homeostatic variables and resources

This exercise works towards goal (c) above. We will continue with this next week.

As a step towards implementing a two-resource problem, similar to that seen in last week’s lecture, this exercise involves implementing a homeostatically controlled robot.

Add an essential variable (e.g. “energy”) on which the survival of the robot depends. The energy variable should decay over time (e.g. an energy variable could decay depending on the speed of the motors). If the essential variable reaches its fatal limit, the robot “dies”.

Add a “resource” to the environment that will allow the robot to “recharge” its essential variable (e.g. simulated “food” or “drink” using light, or a coloured patch on the wall or ground).

Add a behaviour that allows the robot to consume the resource to improve its survival.

Make the variable homeostatically controlled, so it drives the robot’s motivation to actively look for and consume the resource when the essential variable is too low (you are free to define what “too low” means, e.g. by using a threshold or some equivalent mechanism).