Introduction

This is a worksheet for the first COMP3071 lab.

The work in this session is designed to explore the idea of reactive intelligence. That is, systems that display some kind of meaningful behaviour by interacting with their environment, despite not having any memory or learning capabilities. We might think of this as pre-programmed "instinct" rather than "intelligence". Whether this is "true intelligence" is a complex debate, and is perhaps irresolvable and perhaps irrelevant to practical concerns. Instead, think of this as the starting point of looking at behaviour, starting from the simplest. Some of the ideas in this worksheet are taken from the first few chapters of Braitenberg's book Vehicles: Experiments in Synthetic Psychology. A copy of these chapters are on the Moodle as reading for the first week of the module.

In this session we will work with a simple simulation of a mobile robot. That robot is driven by two wheels in a differential drive format. That means that the robot has two wheels, each of which can be driven forwards or backwards at a range of speeds. If you run both motors forward at the same speed, it will move in a straight line forwards. If you run them backwards, then it will move backwards. If they run at different speeds, it will turn. This is a common means of controlling a robot, used for example in most robot vacuum cleaners. The model is somewhat simplified. For example, robots can run over each other and the lamps without crashing.

Getting Started

Download the file called simpleBot.py from the module Moodle page, and make sure that you can run it. You can run this from the command line, or import the file into an IDE, or into a system such as Jupyter notebooks. When you run it, a window should appear, containing two "lamps" (yellow circles) and two "robots" (blue squares). Here is an example of a lamp and a robot.

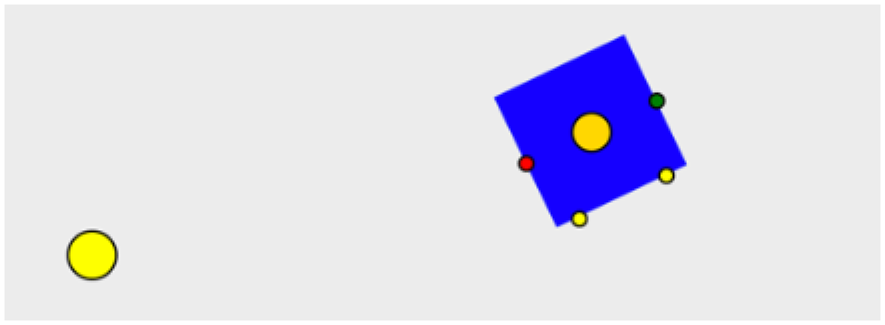

Figure 1 Window displaying the robots and obstacles

The gold-colour circle in the middle of the square represents the centre of the robot, with the green and right circles representing its wheels. The two smaller yellow circles at the "front" of the robot represent light sensors (the choice of colours is deliberately the same as the lamps, to remind us that they detect the signals from the light sensors).

Behaviors

The code that you have been provided with does the following:

1. Initializes a window to display the scene using the "`initialize`" function.
2. Creates a specified number of lights and robots using the "`register`" function.
3. Runs an infinite loop using the "`moveIt`" function that:
   a. Calculates the light sensed by each robot using the "`senseLight`" method in the "`Bot`" class.
   b. Adjusts the speed of the motors based on the information from the light sensors, using the "`transferFunction`" method in the "Bot" class. The provided code sets the motor speeds to zero.
   c. Moves the robot a small step using the "move" method in the "Bot" class. This movement is based on the mathematical principles outlined in Dudek and Jenkin's (2011) book "`Computational Principles of Mobile Robotics`" (Cambridge University Press).
   d. Waits for `50` milliseconds before repeating the above steps using the tkinter "`after`" function.

Instruction

1. This simulation is implemented using the `tkinter` GUI library in Python. While you are not required to have a thorough understanding of the code to complete the following exercises, it is recommended to review the code before beginning the activities.
2. Experiment with the values in the register function. Notice the changes in the simulation when you modify the number of bots and lamps using the variables `noOfBots` and `noOfLights`. Once you have completed these experiments, set the number of bots and lamps to 1 for the following tasks.
3. Next, focus on the `transferFunction` method in the Bot class. This method is called in every iteration, between sensing and moving. This task goal is to get the robot to move.
   a. To start, experiment with setting the variables `self.vl` and `self.vr` to the same constant value within a range of `1.0` to `5.0` to see how the robot moves.
   b. Next, try setting the variables to different values to see the effect on the robot's movement. Determine the values needed to make the robot rotate on the spot.
   c. Experiment with randomly setting the variables each time the `transferFunction` is called to see the effect on the robot's movement.
   d. For more advanced challenges, try to get the robot to move a certain distance, make a random turn, and repeat this process. This will need you to count the number of moves, then make the turn—so. You can accomplish this by storing the count as an attribute of the robot object, or by using the time library to track the duration of the robot's movement. This type of "`exploratory`" behavior is crucial for many

intelligent agents, as it allows them to gather knowledge about their environment through "background" exploration when not actively engaged in a specific task.

4. Up to this point, the robot has ignored the light. The `transferFunction` method has two parameters, which represent the left and right sensor values. Now, we will make the robot respond to these values:

   a. Make the movement of both the right and left motors (`self.vl` and `self.vr` values) dependent on the light level. This means that each movement should be proportional to the sum of the two light levels. This will cause the robot's speed depends on how close it gets to the light, but it will not change its direction. This behavior is described in Chapter 1 of Vehicles (specifically, the implementation used here is that of vehicle 2c, but the ideas are the same as vehicle 1).
   b. Ensure the robot moves towards the light. To do this, the robot needs to move in the direction of the lowest light input value. This can be achieved by changing two lines of code in `transferFunction`. If you need help, look at vehicle 2b in Vehicles.
   c. Make the robot afraid of the light. This means that the robot should move away from the light (this is the opposite of the previous exercise).

5. You may have noticed that the robot randomly moves at high speed. To address this, you can set a speed limit by ensuring that the values of `self.vl` and `self.vr` are within a certain range (for example, between `-10.0` and `10.0`). However, this does not solve the issue where the robot stuck at the upper or lower bound. Implementing this effectively can be challenging and is a topic that is related to control theory, which is discussed in Chapter 4 of Vehicles.
6. I hope that everyone in the class should be able to reach this stage. The next two exercises are a bit more involved, but I hope that most of you will be able to tackle at least one of them during class time.

Introduce an idea of battery level.

The idea of introducing a battery level attribute in the robot is to mimic the real-life scenario where the robot's battery drains with usage. This attribute will help to simulate the battery life of the robot and will add an extra layer of complexity to the robot's behavior. The robot's movement is subjected to its battery level, and it will be forced to seek a charger to recharge when the battery level below certain threshold. This task introduce the idea of energy management in the robot's system and provide a better understanding of the real-world challenges that robots face.

1. Now, we will make the robot respond to its battery life as follow:

a. Introduce an attribute called "`self.batteryLevel`" to the robot, which decreases in value each time the robot makes a move. For now, stick to a simple model where each call to the "`move`" method decrements the battery life by one point (later, you could do something more sophisticated).
b. Introduce a "`checkBattery`" method in the "`Bot`" class and make sure it is called at the end of each iteration of the "`moveIt`" function.
c. If the battery runs out, the robot stops.
d. Introduce a "`seek charger`" mode when the battery level falls below a threshold. In "`seek charger`" mode, the robot is attracted to an object of class "`Charger`" (make a duplicate of the "`Lamp`" class for this).
e. When the robot is over a charger, it stops, and the battery level increases.
f. Depart when the battery is fully charged.

2. This is a simple form of the subsumption architecture. That is, the agent can exist in different "modes", and when a condition is triggered (in this case, the battery being below threshold), a different behaviour happens until it reaches a condition where it can move out of that mode (note that the condition for going into a mode and coming out of it can be different, as in this example).Introduce a second form of passive object

Introduce a second form of passive object; call it a "heater". Duplicate the code for the Lamp class andchange the name/colour. You will then need to duplicate the senseLight method in the Bot class, call it senseHeat, and then the transferFunction will have heat parameters as well as light ones. Now, experiment with a robot that is attracted by light but dislikes heat, etc

This idea can be implemented by

a. Create a new class called "`Heater`" that behaves similar to the "`Lamp`" class, but has a different name and color.
b. Duplicate the "`senseLight`" method in the "Bot" class, and rename it to "`senseHeat`".
c. Update the "`transferFunction`" method in the "`Bot`" class to take into account both light and heat parameters, by using the new "`senseHeat`" method.
d. Test the robot's behavior, by observing how it responds to the presence of both light and heat sources in its environment.
e. The robot should be attracted to light but dislikes heat.

Extension

Here are a number of extensions, for those of you who have time and the programming knowledge. I am
not expecting you to start these in the class, and completing them all could be several days of work! So,
don't try to do them all. Some of these could be the beginnings of a project for the coursework.

1. Vacuum cleaner: have the robot pick up "dirt" that is placed on the floor. We will discuss this extension in more detail during the next session when we cover reactive vs mapping behaviors.
2. Customize the setup: allow the user to place robots, lamps, walls, etc. in any location they choose.
3. Prevent the robots from going outside the window. One way to do this is to implement a toroidal geometry, meaning that a robot going off the edge in one direction will reappear on the opposite side. Another option is to have the robot "bounce" off the edge with various levels of realism, from a physically unrealistic reversal of position to a strong repulsive field at the edge.
4. Introduce some level of noise - for example, the sensors may have some degree of inaccuracy and the robot's movements may be a bit "slippery". Consider how much noise is necessary before the behaviors disappear and whether the behaviors disappear suddenly or "degrade gracefully".
5. Proximity sensors: use sensors that activate when the robots hit a "wall" or each other instead of light sensors.
6. Experiment with changing the velocity based on the light levels (for example, instead of `self.vl = lightL`, use `self.vl += lightL/100.0`)
7. Read chapter 3 of Vehicles and experiment with negative connections.
8. Refactor the code to create a superclass of "Bots" and "Objects" with various subclasses that realize different behaviors.
9. Consider experiments where robots interact with each other. For example, have a robot emit a light source instead of using fixed light sources. We will explore this idea in a future session.