Designing Intelligent Agents  
Spring 2024  
Worksheet 1

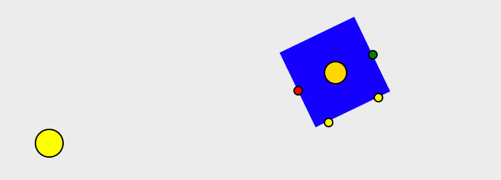
Colin Johnson  
[Colin.Johnson@nottingham.ac.uk](mailto:Colin.Johnson@nottingham.ac.uk)

# Introduction

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 is 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 other objects in their environment without crashing.

# Getting Started

**Download the file** called *simpleBot1.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 (there is an *.ipynb* version). 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.

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).

# Behaviours

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

* Creates a window to display the scene (the *initialise* function)
* Creates a number of lights and robots (the *createObjects* function).
  + This consists of two lists. *agents* is a list of things that move (e.g. robots), and *passiveObjects* is a list of things that don’t move (e.g. lights, charging stations, barriers)
* Runs an endless loop (using the *moveIt* function) that:
  + Adjusts the speed of the motors, perhaps using information from those sensors. This uses the *thinkAndAct* method in the *Brain* class. In the code that you have been given, this doesn’t do much: the motor speeds are both set to zero by the function .
  + Updates the state of the robot. At the moment, all update does is to call the *move* method in the *Bot* class. The maths here is from Dudek and Jenkin (2011), *Computational Principles of Mobile Robotics,* Cambridge University Press.
  + Waits 50 milliseconds, then repeats (this uses the tkinter *after* function)

This is implemented using the *tkinter* GUI library in Python. You do not need to understand the details of the code to begin the exercises, but you might want to spend a few minutes having a look through it before starting the activities. In particular, you don’t need to understand the mathematical details of how the code draws the robot (in *draw*) and moves it (in *move*).

To start with, **experiment with the parameters** passed to the *createObjects* function in *main*. **See what happens when you change** *noOfBots* and *noOfLights* and then run the code. Having experimented with this, you should set the number of bots and number of lamps to 1.

## Thinking and Acting

Now, let’s turn our attention to the *thinkAndAct* method in the *Brain* class. This gets called in every iteration, between sensing and moving. It gets a list of parameters from the *Bot* that it belongs to, i.e:

* *LightL, LightR*: the total value of light on (respectively) the left and right light sensors
* *x,y*: the horizontal and vertical coordinates of the bot, measured from the top-left corner
* *sr,sl*: the current speed of the left and right wheels, respectively

It returns four values:

* *speedLeft, speedRight*: the new speed of the left and right wheels respectively
* x,y: the new position of the bot, as if you had picked it up and put it down. Usually these are set to *None*, meaning that the position does not change, because we usually move them by adjusting the speed of the wheels. But, occasionally we will need to do this.

For the rest of the exercises in this section, you will only need to modify the *thinkAndAct* method in the *Brain* class and not change any other code.

Let’s get the robot to move:

* Get the bot to move! **Start by setting the variables** *speedLeft* and *speedRight* to the same constant value (a good range of values to try is between 1.0 and 5.0). What happens? (if you lose the bots off the edge of the window, then click in the window and they will all return to that point).
* What happens when you **set those variables to different values**? What values do you need **to set them to in order to get the robot to rotate on the spot?**
* What happens if you **set those variables randomly** each time *thinkAndAct* is called?
* Stop the robots going outside of the window. The simplest way to do this is to implement a *toroidal geometry*, that is, a robot going off the edge will reappear on the opposite side (e.g. if it goes off the right edge, it appears in the corresponding position on the left edge). To do this, you will need to set the *newX* and *newY* values to appropriate values when the robot gets close to the edges (i.e. when *x* or *y* is too small or too large).
* (harder; if you cannot do this, move on for now, as the next exercises are easier) Can you **get the robot to move a certain distance (perhaps a random amount within a certain range), then make a random turn, then move again**, etc? This will need you to count the number of moves, then make the turn—so, you will need to store this count as an attribute of the object (i.e. a “self” variable). Alternatively, you could use the *time* library to do this in a different way. This kind of “exploratory” behaviour is important for many intelligent agents, which need a kind of “background” exploration when they are not focused on a task, so as to accumulate knowledge of their environment.

So far, the robot has ignored the light. Note that two of the parameters to the *thinkAndAct* method, represent the left- and right-hand light sensor values. So, we can get it to respond to these:

* Make the movement of both the right-hand and left-hand motors (i.e. the *speedLeft* and *speedRight* values) **depend on the light level**, by adding code to the *thinkAndAct* class. That is, each it is proportional to the sum of the two light levels. The robot’s speed should then depend on how close it gets to the light, but it’s direction will not change. This is the behaviour described in Chapter 1 of *Vehicles* (actually, the implementation used here is that of vehicle 2c, but the ideas are the same as vehicle 1).
* Make the robot **move towards the light**. How will you do this? Think about it—it needs to move in the direction *towards* the lowest light input value. It is possible to do this by changing two lines of code in *thinkAndAct.* If you are stuck, look at vehicle 2b in *Vehicles*.
* Make the robot **afraid of the light**. That is—make it move away from the light by changing the code in *thinkAndAct* (this is, quite literally, the opposite of the previous exercise). I am deliberately using emotional terminology here—this is how people might well describe this behaviour. Braitenberg calls this one a “coward” and the former one “aggressive”, for example. This shows how a simple, reactive system can give rise to behaviour that we might attribute human- or animal- like behaviour to. If you are stuck, look at vehicle 2a in *Vehicles*.

You will probably have noticed that the robot sometimes runs off at high speed (why is this?). You might find it useful to **put a speed limit** on the robot—that is, when *speedLeft* and *speedRight* change, **make sure that they are bounded by a certain value** (say, 10.0). Make sure that these bounds **apply in the negative direction** as well (the speed of a motor never gets below, say, -10.0). This works okay—but, in practice the robots can easily get stuck with the values saturated at -10.0 or 10.0. Think about some better ways to do this, and **implement them**. To do this well is really difficult, and leads into a whole other topic (control theory etc.). There is a sketch of some related ideas in Chapter 4 of *Vehicles*.

I hope that everyone in the class should be able to reach this stage.

## More Complex Exercises

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 the class time. You might need to change more parts of the code now, particular in the *Bot* class.

**Introduce an idea of battery level**. That is, add an attribute to the *Brain* class called *self.batteryLevel*, say. This decreases each time it makes a move; for now, stick to a simple model where each call to the *update* method decrements the battery life by one point, with the lowest value being zero (later, you could do something more sophisticated). Add code at the beginning of *move* so that if *self.batteryLevel* is zero, then the robot stops moving.

Now, make it so that when the battery level falls below a threshold, it switches mode into a “seek charger” mode. In this mode, it is attracted to an object of class *Charger* (make a duplicate of the *Lamp* class for this; perhaps set it to be a different colour). When it is over a charger, it stops, and the battery level increases, departing when it is fully charged. 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**; call it a “heater”. Duplicate the code for the *Lamp* class and change the name/colour. You will then need to duplicate the *senseLight* method in the *Bot* class, call it *senseHeat*, and then the *thinkAndAct* will have heat parameters as well as light ones. Now, experiment with a robot that is attracted by light but dislikes heat, etc

# Extensions

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.

* Vacuum cleaner: have “dirt” on the floor, which is picked up by the robot. (We will go into this one in more detail in the next sessions when we look at reactive vs. mapping behaviours).
* Customise the setup—allow the user to place robots/lamps/walls/etc. where they like.
* Experiment with other ways of handling robots coming to the edge of the window. For example, “bounce” the robot off the edge: there are various levels of sophistication from a physically-unrealistic reversal of position, to some kind of strong repulsive field at the edge, to some idea of a barrier where the robot turns around until it is free to move again. You should be able to do this from within *thinkAndAct*, but you might need to modify the *move* method.
* Introduce some idea of noise—the sensors have some degree of inaccuracy; the movement of the motors is a bit “slippy”. How much noise does there need to be before the behaviours go away? Do the behaviours go away suddenly, or do they “degrade gracefully”?
* Proximity sensors: rather than light sensors, there are sensors that activate when the robots hit a “wall” or each other.
* A whole other set of experiments can be done where the light levels (etc.) *change* the velocity rather than *setting* it (i.e. rather than having statement like *speedRight* = *lightL*, you have statements like *speedRight += lightL/100.0*)
* Read chapter 3 of *Vehicles* and experiment with negative connections by adding code to the *Brain* (and perhaps also the *Bot*) class.
* Refactor the code so that there is a superclass of “Bots” and “Objects” and various subclasses that realise the different behaviours.

A whole other set of experiments can be done where robots interact with each other. At its simplest, consider having a robot that emits a light source, rather than the light sources being fixed. We will explore this in a future session.