Lab 2 - The Extended Braitenberg Architecture

Return to main labs page: **Main Labs Page (https://canvas.sussex.ac.uk/courses/34985/pages/main-labs-page-3)**

The *Sandbox* framework code, and all other code for labs 1, 2 and 3 is here:

> **AS_autumn_2025_exercises.zip (https://canvas.sussex.ac.uk/courses/34985/files/6130167?wrap=1)** ⭳ **(https://canvas.sussex.ac.uk/courses/34985/files/6130167/download?download_frd=1)**

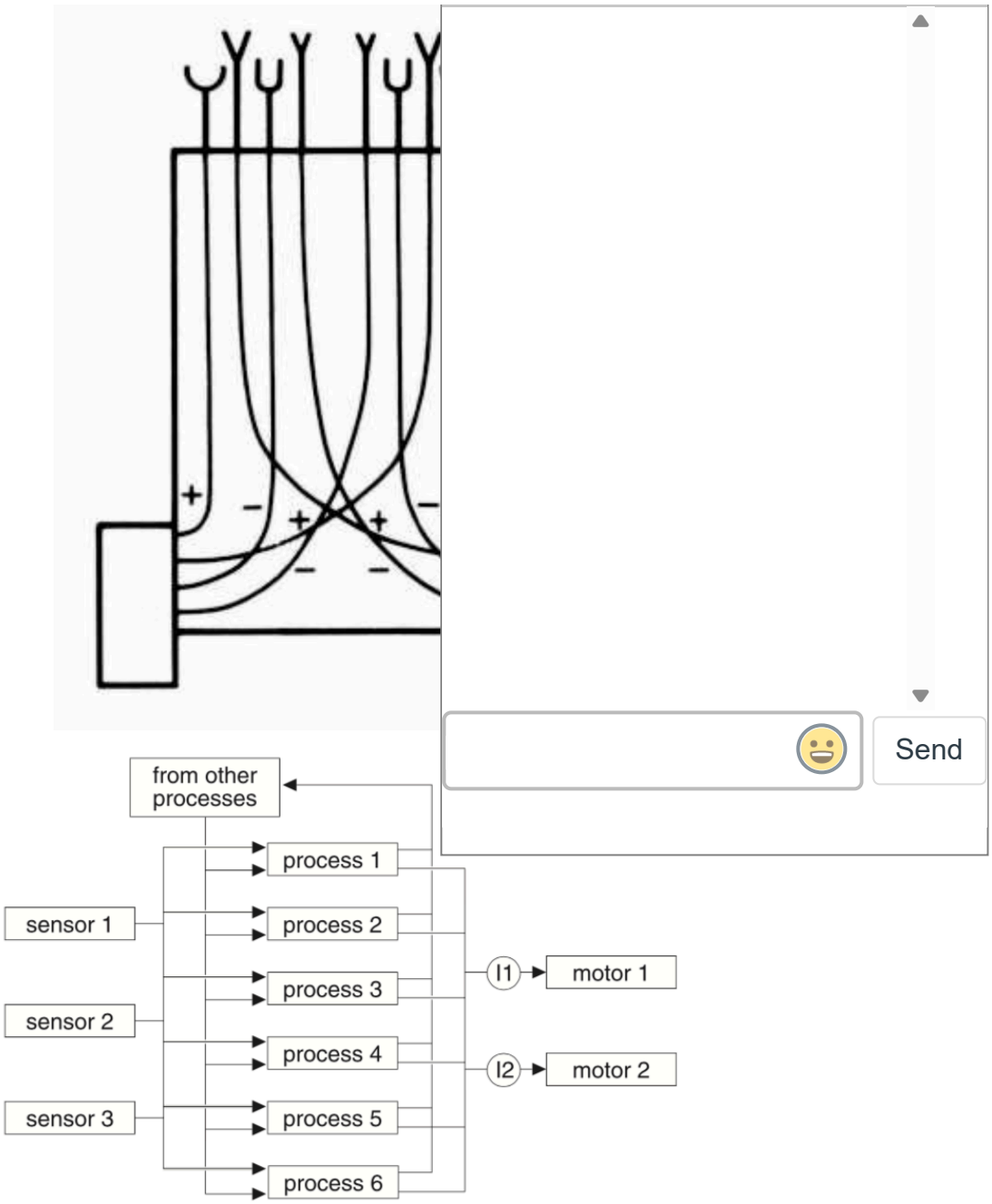


Figure 1. Braitenberg's vehicle 3c [1] (left), which combines multiple sensory modalities and both inhibitory and excitatory (i.e. positive and negative) connections. The extended braitenberg architecture [2] (right) takes Braitenberg's idea further, having multiple processes connected to multiple sensors, and to each other (which could be simple sensorimotor connections, like Braitenberg's, or could be more complex, e.g. like neural networks).

Introduction

Braitenberg described a vehicle, 3c (**Figure 1**), which had multiple pairs of sensors, which respond to different kinds of stimuli [1]. The behaviour of vehicle 3c is based on the combined actions of multiple sensorimotor loops which run in parallel, through the vehicle's various sensors, and being added together in the vehicle's motors.

The Extended Braitenberg Architecture [2] (EBA, **Figure 1**) works in a similar way, with multiple processes connecting an agent or robot's sensors to its motors. It is more sophisticated and

flexible than vehicle 3c, because its processes can be of various types, of arbitrary complexity, and can also communicate with each other.

You may find the EBA a useful template to follow if you program robot controllers in your coursework assignments. In this lab, we will focus on vehicle 3c, as learning how to make this simple vehicle work should help you if you choose to go on to the more challenging task of designing more complex EBAs, for more interesting behaviours.

Programming controllers in Sandbox

```python
def generalised_braitenberg(dt, inputs, params, state=[]) -> List[float]:

    ### set left motor speed
    # weighted connection from left sensor to left motor command
    left_speed_command = inputs[4] * params[0]
    # weighted connection from right sensor to left motor command
    left_speed_command += inputs[5] * params[1]
    # add bias term to left motor command
    left_speed_command += params[2]
    ### set right motor speed
    # weighted connection from left sensor to right motor command
    right_speed_command = inputs[4] * params[3]
    # weighted connection from right sensor to right motor command
    right_speed_command += inputs[5] * params[4]
    # add bias term to right motor command
    right_speed_command += params[5]

    # return motor speed commands to robot's controller
    return [left_speed_command, right_speed_command], None
```

Figure 2. An example of a step_fun function, from lab 1.

```python
def part1controller(dt, inputs, params, state):

    # replace these lines with your code for setting the motor speed commands
    left_speed_command = 0
    right_speed_command = 0

    return [left_speed_command, right_speed_command]
```

Figure 3. An empty `step_func` function, for you to complete.

There are various ways to implement controllers for the agents in Sandbox. The one you will use in this lab is to write a `step_fun` function which works as a kind of plugin for a `Controller` object. In lab 1, you used a `step_fun` which I wrote for you, `generalised_braitenberg`, which is in the `controllers.py` file for that lab (see **Figure 2**). For this lab, I have provided you with some empty `step_fun` functions, which you will complete, by adding code which uses the sensory inputs to the controller to generate appropriate commands for the motor.

Any `step_fun` functions we write for the `Controller` class must have the same input parameters and the same outputs as shown in **Figures 2** and **3**. In short, the role of the function is to take the sensory inputs from the controller, `inputs`, and use them to generate appropriate commands for the motors, `left_speed_command` and `right_speed_command`. For this lab, you don't need to worry about the `dt` or `state` input parameters, but you may want to make use of `params`. `params` is a list of parameters which can have any length. In `generalised_braitenberg`, I used a list of 6 parameters, for the 4 weights and 2 biases in the neural network. When you are combining different behaviours in your controllers you may want to have a weight parameter for each behaviour, so that you can easily adjust how strongly each of them influences the robot's behaviour. In **Figure 5**, you can see that I have included 3 parameters for this purpose, but you can have more or less parameters, to mee the requirments of you own design.

Figure 4. How the inputs and outputs of a `step_fun` map to the simulated robot. Inputs 4 and 5 map to the first pair of sensors, as in Lab 1. The second pair of sensors maps to inputs 6 and 7.

For this robot, the inputs in our `step_fun` functions will map to the robot's sensors as follows:

- `inputs[4]` → left yellow light sensor
- `inputs[5]` → right yellow light sensor
- `inputs[6]` → left red light sensor
- `inputs[7]` → right red light sensor

```
50  # set up controller - uncomment only one of these lines, for the part of the lab which you are working on
51  # controller = Controller(step_fun=part1controller, inputs_n=4, commands_n=2, params=[2, 0.1, 10])
52  # controller = Controller(step_fun=part2controller, inputs_n=4, commands_n=2, params=[2, 0.1, 10])
53  controller = Controller(step_fun=part3controller, inputs_n=4, commands_n=2, params=[2, 0.1, 10])
```

Figure 5. Selecting the controller for each part of the lab. The only difference between the 3 lines is the name of the step_fun function which is to be used, so to change the behaviour of the robot, all we need to do is

change that name, but for convenience we can switch between the 3 options by simply commenting/uncommenting these lines.

In the `controllers.py` file for this lab, there are 3 empty `step_fun` functions for you to complete: one for each part of the lab. You can select which one to use by uncommenting the corresponding line, as shown in **Figure 5**.

[For additional information on the Controller class, see the Sandbox API documentation, here **Sandbox_API.zip (https://canvas.sussex.ac.uk/courses/34985/files/6130171?wrap=1)** **(https://canvas.sussex.ac.uk/courses/34985/files/6130171/download?download_frd=1)**]

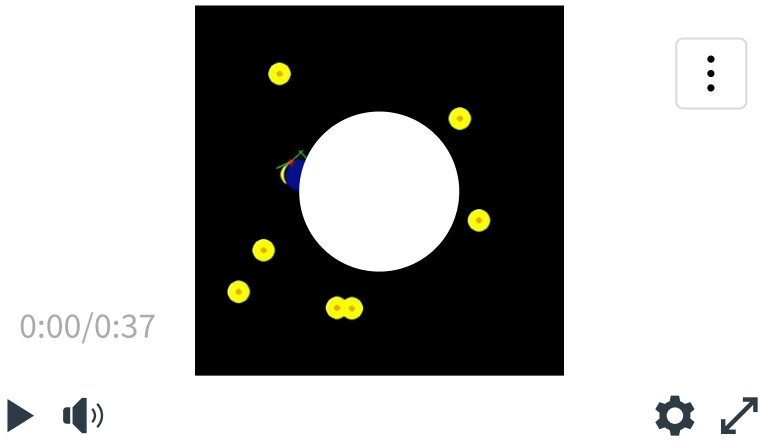
Selecting environments

```
30  # for parts 1 and 2, set this variable to True, so that all lights are yellow
31  # for part 3, set this variable to False, so that lights are red/yellow
32  parts1_2 = False
```

Figure 6. Selecting the correct environmental features for the different parts of today's lab, using the parts1_2 variable.

In parts 1 and 2 of the lab, we will only have yellow lights in the environment. In part 3, we will have equal numbers of yellow and red lights in the environment. We can switch between these two options using the `parts1_2` variable (**Figure 6**). For parts 1 and 2, this variable should be set to `True`. For part 3, it should be set to `False`.

Part 1: Parallel behaviours



In this exercise, you will program a controller which only uses one of the `Robot's` sensor pairs to make a robot search for lights and drive towards them, but to turn away when it gets very close to them. We can break this behaviour down into three parts:

1. Turn to find lights when none can be seen.
2. Drive towards lights when they are seen.
3. Turn away from lights which are very close.

It might seem natural to write controller code which selects between these three components of behaviour using program logic, but the idea here is to make them all run in parallel.

Program three blocks of code: one which makes the robot drive towards lights (of course, you can re-use existing code), one which makes the robot turn slowly in the absence of light, and one which makes the robot turn sharply in response to very close lights. Each of these blocks should produce a command for each motor, and the commands to the motors should be combined using simple addition.

$$left\_speed\_command = left\_light\_seeking + left\_slow\_turning + left\_avoid\_$$
$$right\_speed\_command = right\_light\_seeking + right\_slow\_turning + right\_av$$

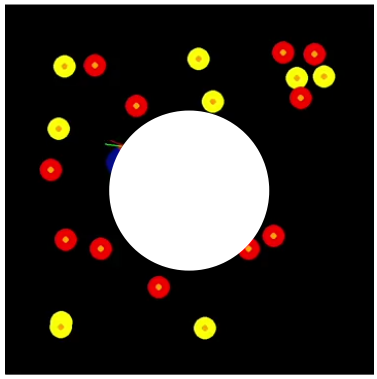
Because we *add* the motor speeds from the three parts together, rather than choosing one or the other, they effectively act in parallel, until they combine in the robot's motors. You can use logic *in* the individual parts of the controller, but you should not use logic to select *between* them.

If you don't know how to approach this problem, then my recommendation is to begin by programming all of the elements separately, e.g. you might start with a light-seeking controller, which you already know something about. When you have the different elements of the controller working on their own, then you can think about how to make them work together.

Part 2: Parallel behaviours with multiple sensors

Copy and modify your controller from part 1 to use the `Robot's` second sensor pair in the block of code which is used to avoid "crashing" into lights.

Part 3: Seeking and avoiding at the same time



In this exercise, we use the same `Robot` again, but this time we have two different kinds of light. This time, the robot should seek yellow lights and avoid red lights. To achieve this, you can use parallel behaviour as before, where one behaviour uses one pair of sensors to seek yellow lights, and the other behaviour uses the other pair to avoid the red lights. It is up to you which sensors you use to make the robot turn when it sees no lights - any combination which works is fine.

Bibliography

[1] Braitenberg, V. (1984). *Vehicles, experiments in synthetic psychology*. Cambridge, Mass: MIT Press.

[2] Pfeifer, R., & Scheier, C. (2001). *Understanding intelligence*. MIT press.