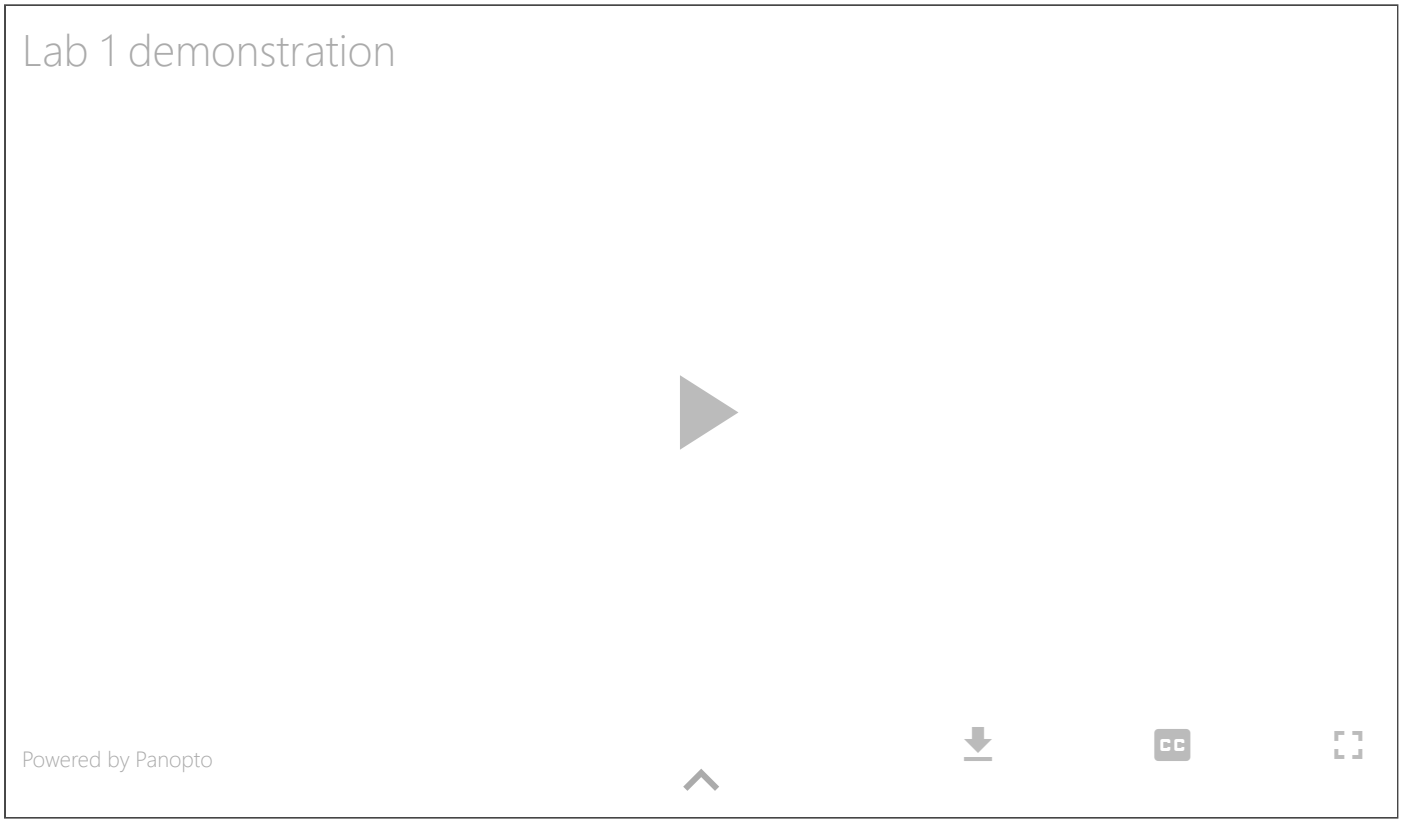
Lab 1: Adapting Braitenberg's vehicles in Sandbox

Navigate 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)**

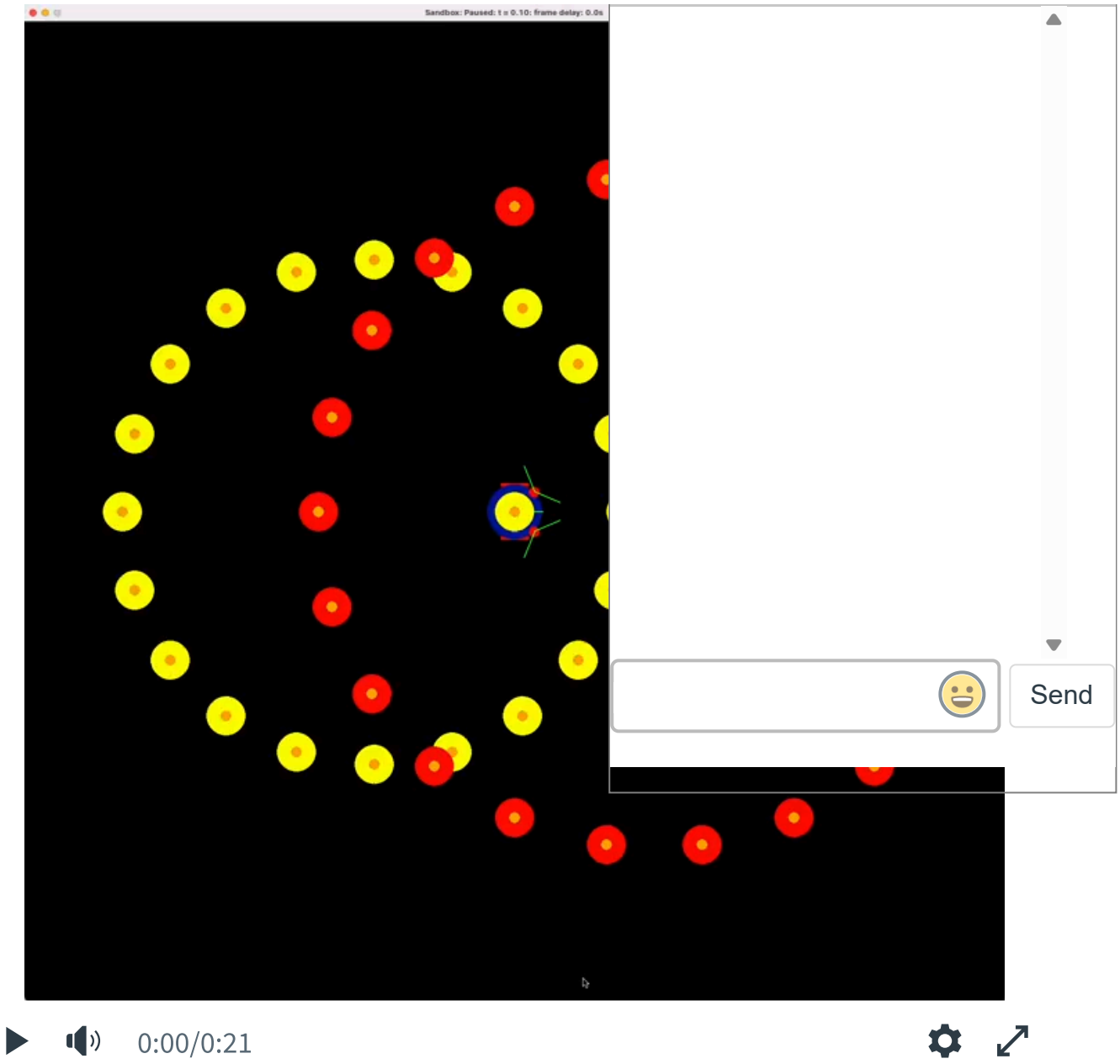
Here is a short demonstration of how to run the code:



Please note that the demonstration is not fully up-to-date on the directory structure for this year, which is shown in Figure 2, so the first minute or two of the recording show an old Canvas page and directory structure. However, to run the code for this lab, you should still open the `lab1_braitenberg` folder (even though that is in a slightly different place now) and run the `lab1.py` file, and everything else in the recording should be accurate.

A brief introduction to Sandbox

Module chat



Recording 1: Multiple robots in Sandbox

In this lab, you will use my simulation framework, *Sandbox*, for the first time (unless you have met it before, on the IAM module). *Sandbox* is the latest version of a simulation framework for agent-based modelling which I created for teaching the Adaptive Systems module at Sussex, in 2020. In 2020, it was loosely known as *the simulation with no name*. In 2021 it became *SituSim* (*An abbreviation for Situated and Embodied Agent Simulation*). In 2022, I renamed it again, to *Sandbox*. The reason for this new name is that I want to make it clear that we use this simulation framework as a *sandbox for experimenting* with adaptive systems - in most labs, we will work with simulations of mobile agents, which will often be modelled on robots, but that does not mean that our objective is to become good robot programmers. Our true objective is to use the various agents and systems in Sandbox as vehicles for exploring and learning about processes of adaptation.

As *Sandbox* is developed for the purposes of experimentation, it is programmed in an object-oriented and highly customisable style. It is relatively easy to set up simulations of agents and environments for our experiments, and it is also *relatively* easy to extend the existing code to add

new types of agents and environmental features. In each of our lab classes, you will learn something new about *Sandbox* and the many ways in which it can be used and extended.

Sandbox is always under development. If you find anything in the code which you think might be an error or bug, please let me know - either you have found a mistake in my code, and you will be giving me valuable help in fixing it, or there is something which I have not explained sufficiently well, and you will be helping me to improve my explanation, which will also be very welcome help.

Simulating Braitenberg's vehicles

"... follow me not through a world of real brains but through a **toy world** that we will create together."

(Braitenberg, 1984)

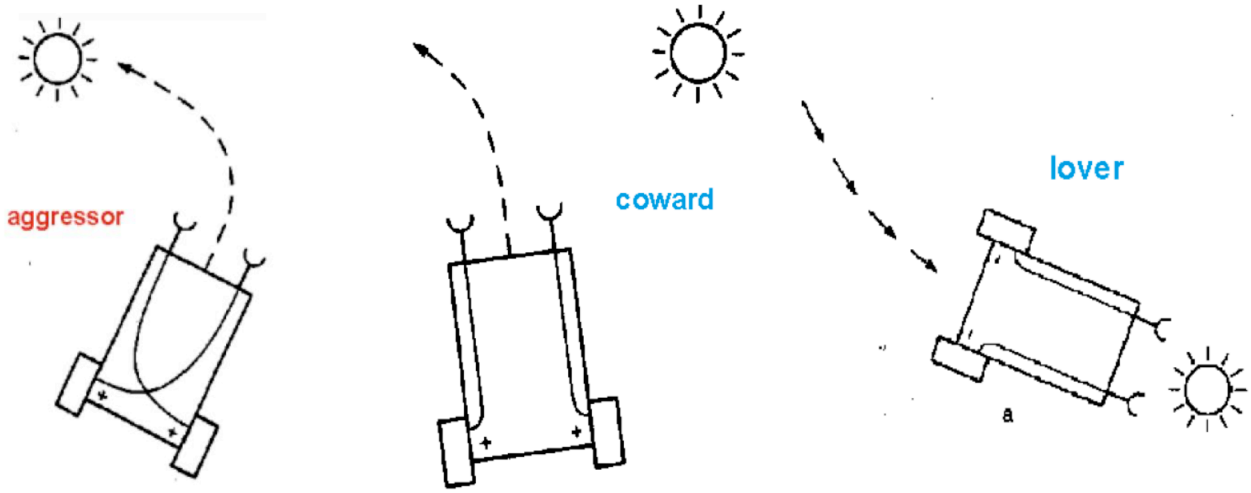


Figure 1. Those of Braitenberg's vehicles which you will implement in this lab.

The thought experiments

In his book, *Vehicles: Experiments in Synthetic Psychology* (1984), Braitenberg leads his reader through a series of related thought experiments, which demonstrate the potential for outwardly interesting behaviours to arise from the interaction of simple elements which are connected in a sensorimotor loop (**Figure 1**).

The relevant chapters are here (you don't need to read past Vehicle 3):

> **Braitenberg_vehicles_1-28.pdf (https://canvas.sussex.ac.uk/courses/34985/files/6130109?wrap=1) (https://canvas.sussex.ac.uk/courses/34985/files/6130109/download?download_frd=1)**

Braitenberg arrived at his vehicles by route of years of the study of real brains and behaviour, but the mechanisms and behaviours of his vehicles are described only in words and the most rudimentary of diagrams. While their simplicity may actually make them easier to communicate than models which are described by systems of mathematical equations, it is still reasonable to

ask whether they can *actually* work. We may also question whether the chasm which divides them, in terms of level of detail, from even the simplest of agents in the real world, actually renders Braitenberg's vehicles implausible as even the most general descriptions of simple sensorimotor behaviours.

By simulating Braitenberg's vehicles, in an *extended* thought experiment, which a small step closer to reality than the account in his book, we can enhance our ability to judge the credibility of his vehicles, while also exploring our own hypotheses about sensorimotor behaviour in the toy world. **More importantly**, in later labs, we will implement controllers for these vehicles which add processes of self-adaptation, so this lab will lay some of the groundwork for experiments which we are ultimately much more interested in.

The labs directory structure

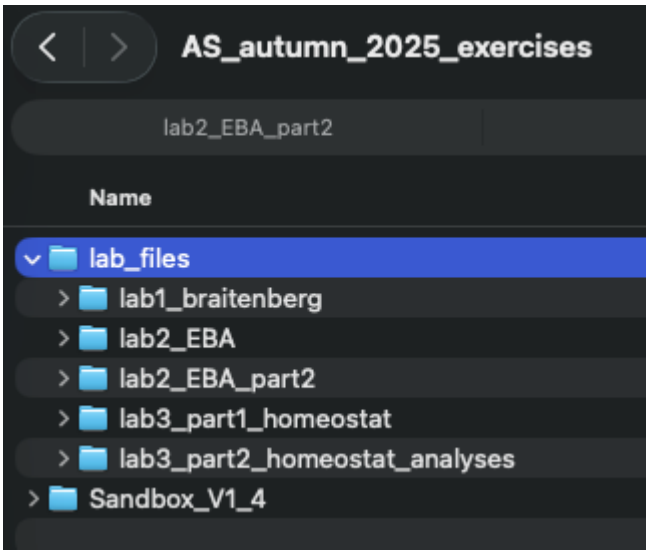


Figure 2. The directory structure for our lab code

It is important to understand and stick to the directory structure shown in **Figure 2**, or some parts of the code might not run correctly. This is because the main scripts for lab classes use relative paths to find *Sandbox*, so if they are moved then they won't be able to find the code they need.

The main simulation framework code is all in `Sandbox_V1_4`. We won't make ever changes to the files in that folder unless we really have to (e.g. if there is a bug I need to fix, or something that I must add). `lab_files` is where the folders for *all labs* (including ones which I have not yet provided the code for) must be, if the code is going to run without errors.

params.yaml lab1.py

```yaml
# controller parameters
b_l: 1
b_r: 1
w_ll: 0
w_rl: 0
w_lr: 0
w_rr: 0
# left motor noise parameters
white: 0
brown: 1
spike: 0
# simulation parameters
n_runs: 1
duration: 200
screen_width: 800
animate: "true"
animation_frame_delay: 10
plot_data: 0
```

Figure 3. The parameters file.

The params file

In later labs, you will start to learn more about how my code works, and you will also start to write your own code. For this lab, I don't want you to be distracted by studying or writing code, and so you will be able to do all of the exercises just by manipulating the parameter file shown in **Figure 3**. The first block of parameters will affect the controllers, and therefore the behaviours, of our simulated vehicles. The next block will control the noise in the simulation. The last block will affect the simulation itself. I'll explain that block here, and the other blocks later on this page.

- `n_runs` controls how many the simulation will run every time you run the `lab1.py` Python file. I recommend leaving this set to 1 at first. Once you start to get your controllers working, see what happens when you increase this number, e.g. to 8. Set this to positive integer values only.
- `duration` controls how many units of simulated time each simulation runs for. You can adjust it to make the simulation longer or shorter as required. Set this to positive integer values only.
- `screen_width` controls the size of the simulation window. 800 is good for my laptop, but might be a bit small for the screen you are working with, so you might want to make this bigger. Set this to positive integer values only.
- `animate` determines whether the simulation animation runs or not. If you set this parameter to any value other than `1` then the simulation will run, and you will still see plots at the end, but

you won't see any animations. In this class, you might leave this set to `1` all the time, but in later labs (and possibly coursework assignments) you probably won't want to watch every single animation.

- `animation_frame_delay` can be used to slow down or speed up the animation of a simulation (you can also do this when the simulation is running, as described below).
- `plot_data` determines whether or not the plots shown in **Figure 6** are drawn. They are only drawn if this parameter is set to `1`. There will be times when you want to actually study these plots, but often it will be convenient to disable them, by setting this parameter to any value that is not `1`.

Animation controls

The *Sandbox* animation has some useful controls. When a simulation runs and is animated, with the PyGame animation window selected (click on it to make sure), you can use these keys to control what happens and how quickly it runs:

- **SPACE KEY**: When the animation appears, *it will be in a paused state*. To un-pause or pause the animation, press the space key.
- As you will see in later labs, we can easily run and animate a batch of simulation runs. We have the following controls available to either exit a simulation or just animations:
  - **ESC KEY**: Press this key to exit a simulation immediately. Any simulation runs in the current batch which are waiting to run will still run.
  - **1 or LEFT ARROW**: Press the 1 or left arrow key (1 might not work on some Chinese keyboards) to quit the animation for the current simulation. Unlike when you press Esc, this will not exit the simulation - that will run in the background. Any simulation runs in the current batch which are waiting to run will still run and be animated as usual.
  - **0 or RIGHT ARROW**: Press the 0 or right arrow key (0 might not work on some Chinese keyboards) to quit the current animation and skip the animations for all remaining simulation runs. Any simulation runs in the current batch which are waiting to run will still run, but will not be animated.
- **DOWN ARROW**: Press to slow down simulation. If you would like to temporarily slow a simulation down to watch an agent's behaviour carefully, then press this button repeatedly to slow things down (you have to press it repeatedly, unfortunately - you can't just press and hold).
- **UP ARROW**: Press to speed up simulation. The simulation will not run any faster than the default speed - this control is only useful if you have slowed the simulation down and would like to speed it up again.

The Sensorimotor loop in action

In **Figures 4** to **7**, I have illustrated how we can move from from the agent-environment coupled system, to the sensorimotor loop, to the outputs from the simulation and how they correspond to the different elements in the sensorimotor loop, and finally to the behaviour of the agent which is embedded in the loop.

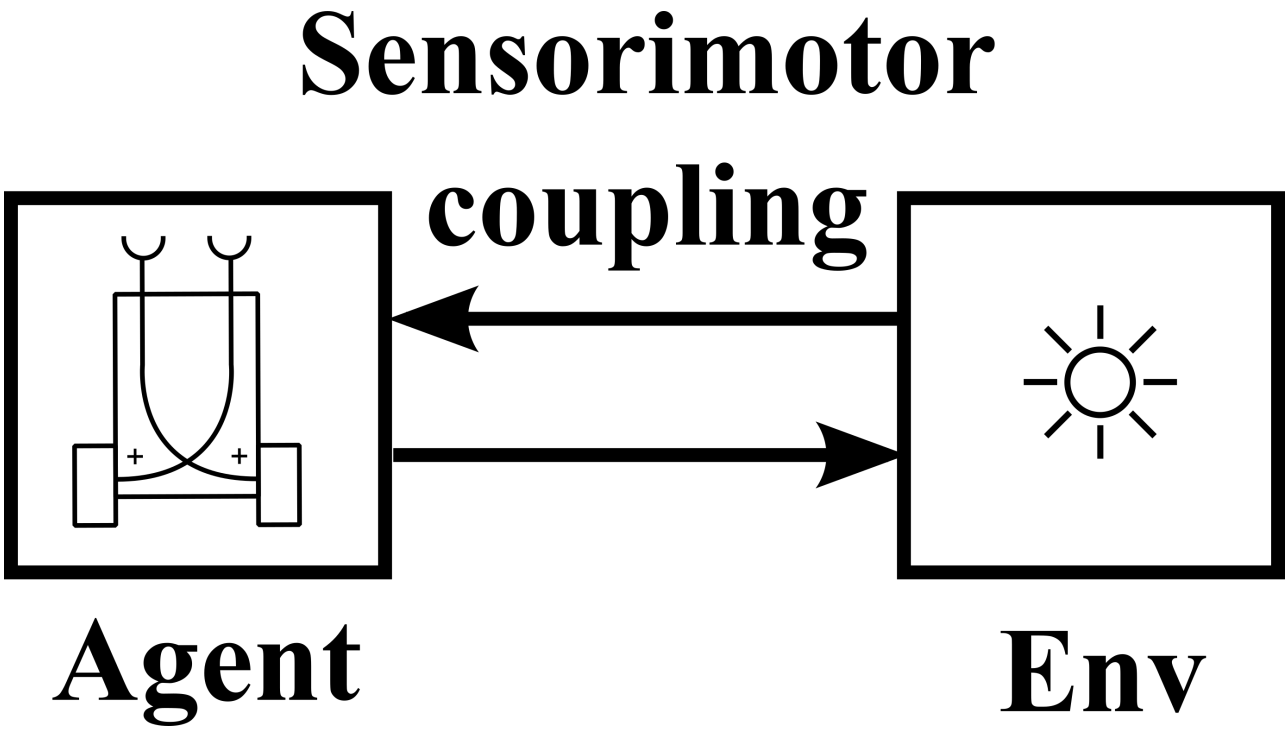


Figure 4. One of Braitenberg's vehicles coupled to its environment through its sensors and motors.

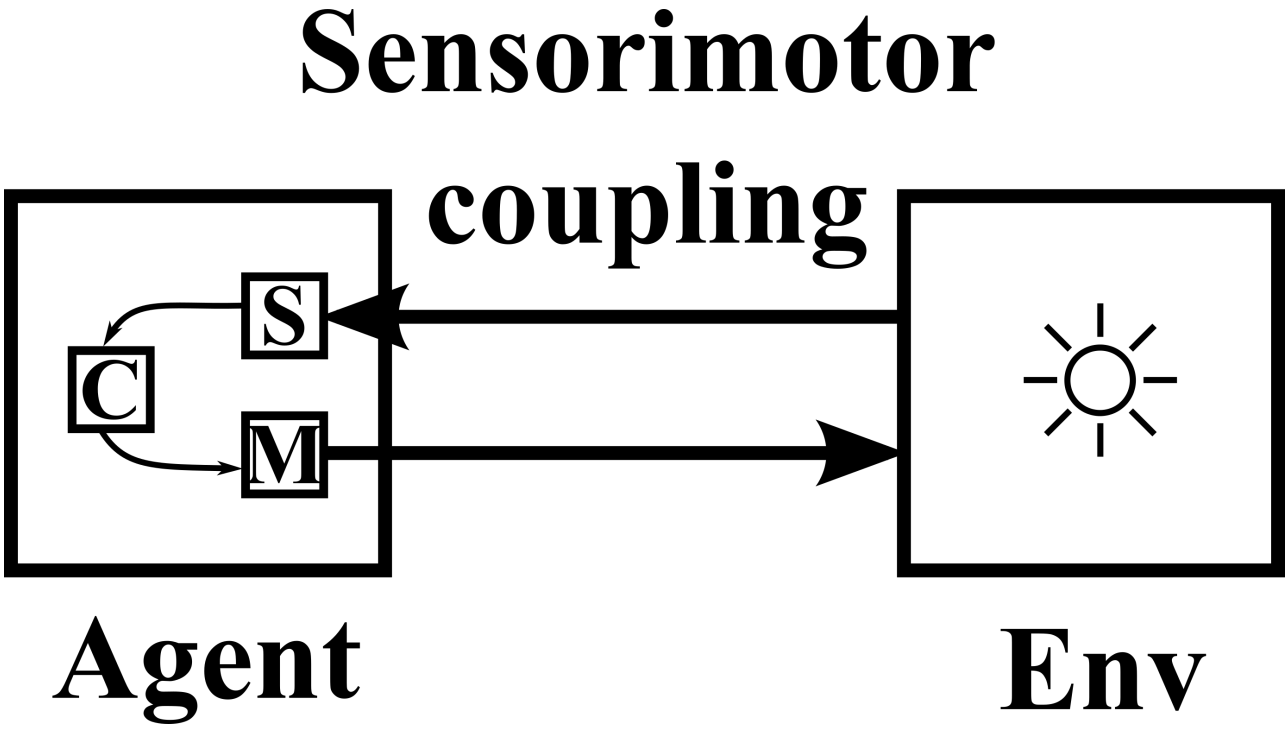


Figure 5. The agent-environment coupled system drawn as a sensorimotor loop, and split into its 4 main elements: the agent's sensors, controller, motors, and its environment.

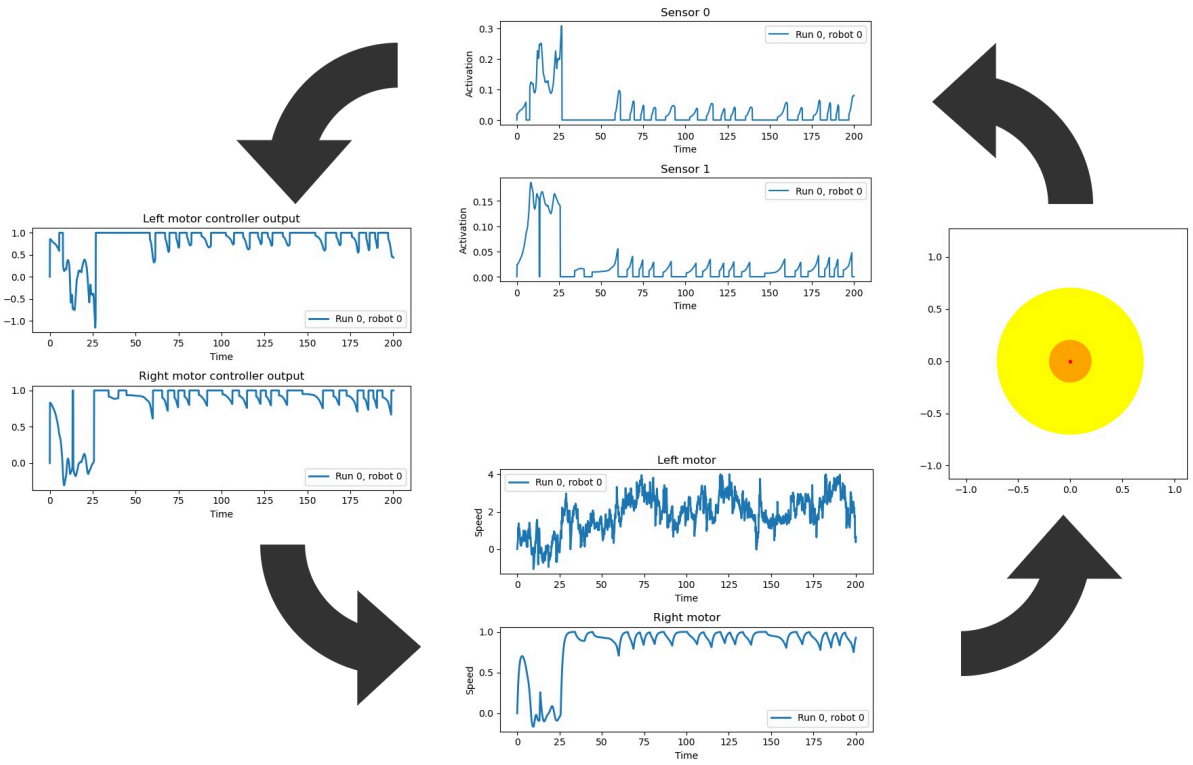


Figure 6. The sensorimotor loop, as it appears in the outputs from our simulation. One of the things I would like you to learn from our labs is how to interpret and explain plots like these.

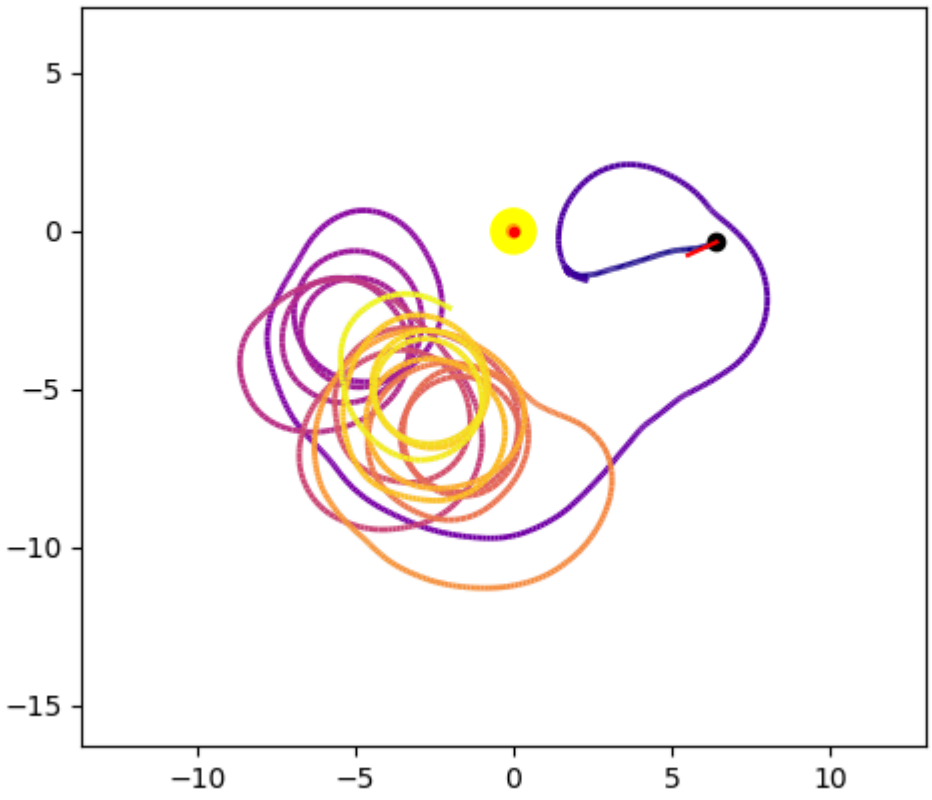


Figure 7. The robot's external behaviour, or trajectory through the xy-plane, corresponding to the plots shown in Figure 6.

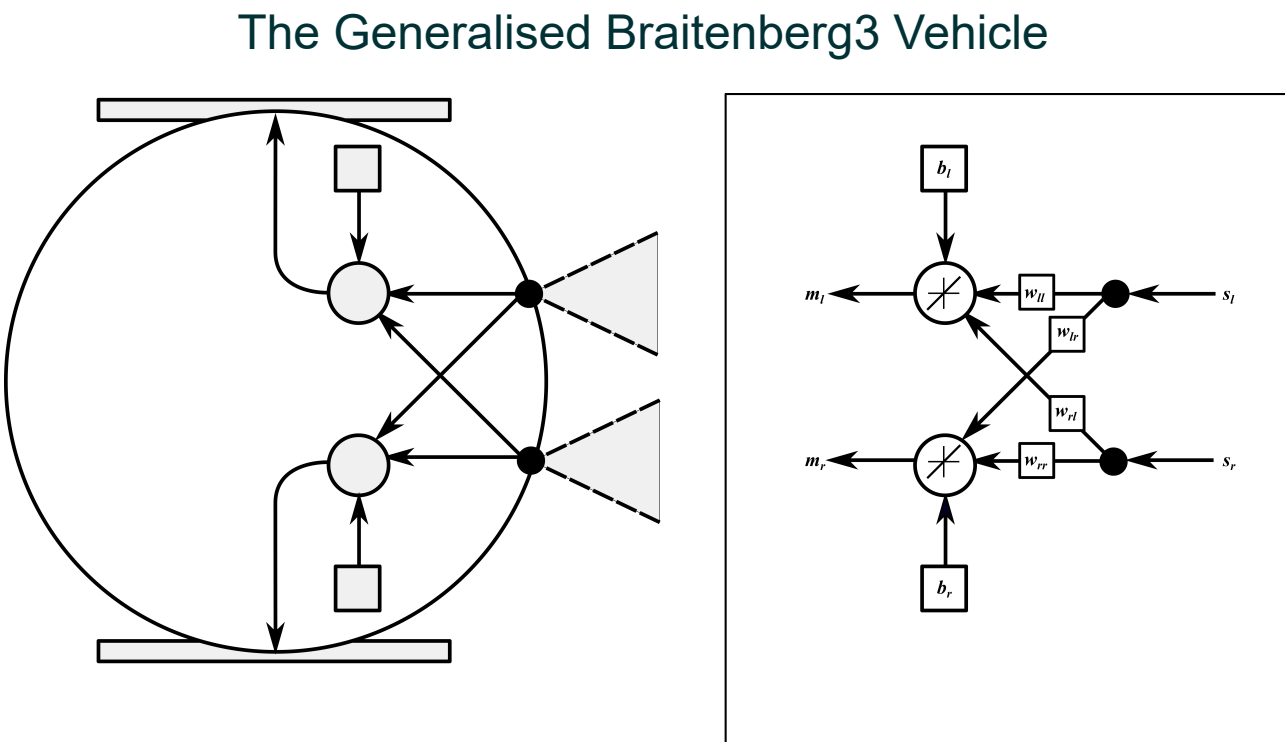


Figure 8. The neural network control architecture which generalises Braitenberg's vehicles 1 to 3b.

From vehicle 1 up to vehicle 3b, we can implement Braitenberg's vehicles [1] with the robot and simple neural network shown in **Figure 8**. I think you can probably guess this, but just to be clear, the parameters shown in this diagram correspond to the parameters in the file shown in **Figure 3** like so:

- $b_l \rightarrow$ `b_l`
- $b_r \rightarrow$ `b_r`
- $w_{ll} \rightarrow$ `w_ll`
- $w_{rl} \rightarrow$ `w_rl`
- $w_{lr} \rightarrow$ `w_lr`
- $w_{rr} \rightarrow$ `w_rr`

As I have already programmed the neural network, you only need to experiment with the parameters to see how they affect behaviour. You can set these parameters to positive, zero or negative values.

Exercises

In the exercises, you will start to get to know *Sandbox*, and the differential drive. You will also learn how to adapt the behaviour of the simulated robot by setting its parameters to appropriate values. This robot, or Braitenberg vehicle, is hardwired - it has no way to adapt itself (no *internal* processes of adaptation), so you will act as its designer and an *external* process of adaptation.

1. Getting to know the differential drive

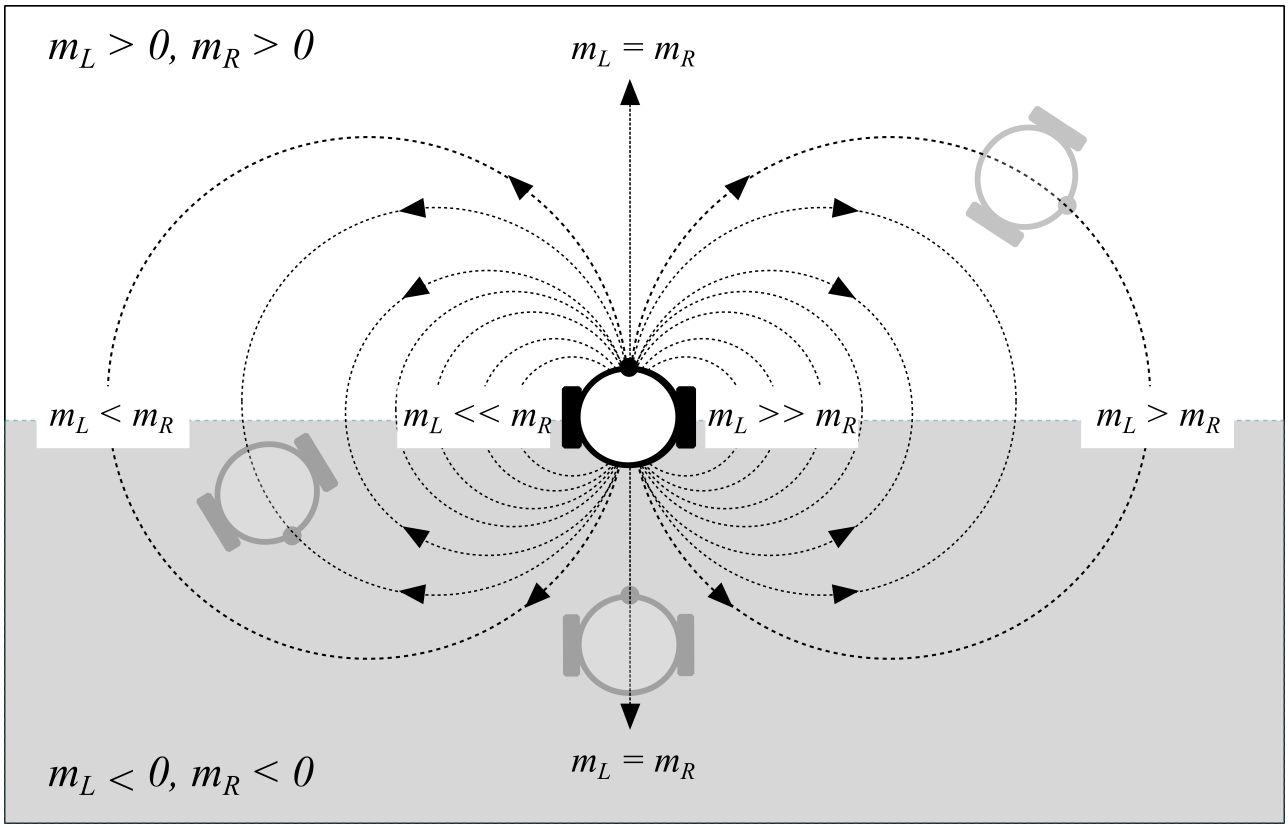


Figure 9. The differential drive.

We can use the bias parameters in our neural controller to set the robot's motor speeds to constant values, which may result in the kinds of trajectories shown in **Figure 9**. In the parameters file, try setting b_l and b_r (`b_l` and `b_r` in the file) to the following pairs of values, and run the simulation to see what behaviour each pair of values leads to:

- [0, 0]
- [1, 1]
- [-4, -4]
- [-2, 2]
- [2, -2]
- [0.5, 5]
- [5, 0.5]
- [-5, -0.5]
- [-1, 3]

For each pair of bias parameter values, identify which region of the diagram in **Figure 9** the resultant behaviour corresponds to. Some behaviours may not correspond to the trajectories shown in the diagram at all, as it only deals with certain cases. If you have a behaviour which does not match the diagram, can you work out why?

2. Hating and fearing lights

In this exercise, your task is to find good parameters for implementing both the light-seeking (Braitenberg's *aggressor*) and light-avoiding (Braitenberg's *coward*) vehicles. Note that I have written *good* rather than correct - most good solutions to these problems will look similar to each

other, but there is no single set of parameters for either vehicle which we would necessarily all agree to be optimal.

I recommend beginning with the light-seeking vehicle. Once you have parameters which work for that vehicle, it is relatively to modify them to also implement the light-avoiding vehicle. One more clue that I will add is that a connection with a zero-valued weight is effectively removed. In other words, if you look at the connections of the one of Braitenberg's vehicles which you are trying to implement for this exercise, you might be able to see which subset of the weights should be *non*-zero.

3. Loving the light

Implement a vehicle which is attracted to the light, but which slows down as it gets closer to the light, and - unlike the simple (aggressive) light-seeking vehicle - stops moving just before it reaches the light. Note: the majority of students find this exercise very difficult - please spend at least a few minutes thinking about how to solve this puzzle, but if you are completely stuck, please don't hesitate to ask for help. You are not being tested in these lab classes - I will often deliberately write your exercises with the expectation that Haoyue and I will need to come and talk to you from time to time - especially in practical classes (but in lectures too), I prefer to teach through conversation rather than purely through giving instructions.

A small clue: the most straightforward implementation of this vehicle will require you to use the bias parameters in the neural network.

4. Relying on only one "eye"

In this very challenging exercise, you need to set the parameters of the controller in such a way as to make the robot be able to find its way towards the light ***using only one of its sensors***.

I have taught a lesson similar to this one on many different occasions, and I have rarely seen a student solve this problem on their own - in fact, I was also completely unable to solve this problem on my own when I was a student. Once you are told it, you will probably soon realise that it is very simple, but it is also highly counter-intuitive, i.e. very difficult to think up on your own.

5. The difference noise can make

In the real world, noise is difficult to avoid for systems in general, and certainly for sensorimotor ones. The meaning of the word "noise" is a little vague, but it generally refers to some source of interference in a signal or a system, or alternatively just to some unspecified source of variation in the properties of a system or collection of systems.

From the point of view of experimenting with systems, noise can be good for *testing* how robust or stable the system's behaviour really is. Any system which will fail as soon as it encounters noise is probably not worth much, so it is good to see what kinds of noise, and what *scales* of noise, a system can tolerate. [I mentioned in the EvoRobotics lecture that noise is often used to aid in crossing the *reality gap*, essentially for this reason of robustness - a system evolved in very

challenging noise conditions in simulation can be robust to noise in general, in both simulated and real environments]

In today's lab, there are three kinds of noise which you can turn on and off independently: white noise (like static), brown noise (a random walk in one dimension), and spike noise (the occasional spike in either the positive or negative direction). To turn any of these on, set their corresponding parameters (**Figure 3**) to `1`, and to turn them off set the parameters to any other values.

As you turn the noise types on, look at how your changes affect the plots shown in **Figures 6** and **7**.

1. Try each of the noise types one at a time with your light-seeker (aggressor) controller.
   - Which of the noise types do/don't cause a lot of trouble? Can you explain why?
2. Try different combinations of noise types.
3. Try noise with all of your controllers.
   - Which of your controllers are/aren't robust to noise?
     - Can you explain why?
     - Can you think of ways to make your controllers more robust?

One last thing that I'll point out here: noise can be introduced anywhere in the sensorimotor loop - it won't necessarily have the same effect wherever we put it, but due to the loop being closed its effect will propagate through the whole system. Here, I chose to inject the noise into one of the robot's motors, but I will often generate the noise in the robot's sensors or controller instead.

6. Multiple lights

The light in the simulation is an instance of the `LightSource` class, created on line 53, and simultaneously placed in a list. Try adding more lights to this list, and see how you can use them to control the behaviour of the robot. The parameters to the `LightSource` constructor are `x`, `y`, and `brightness`.

Analysis and reflections

Analysing the system's behaviour

Analysis of a system's behaviour often involves looking for meaningful patterns. For this lab, the easiest place to look for patterns is in the trajectories the robot follows towards or away from the light. One other useful tool for finding patterns in the behaviours of dynamical systems is the phase portrait. In the phase portrait I have included for this lab, I plot the following time series on the x- and y-axes:

- $x = (s_L - s_R)$
- $y = \dot{x} = \frac{dx}{dt}$

where s_L is the output or activation of the robot's left sensor over the full duration of the simulation, and s_R is the same for the robot's right sensor. This means that x is the difference

between the activations of the two sensors over time, and that y is the rate of change of that difference.

There are two things I'd like you to consider regarding the phase portrait:

- Why do you think I chose these variables to plot?
- What similarities or differences do you see between the phase portraits for different behaviours, e.g. *cowards* and *aggressors*?
    - How can they be explained?

The other thing I want you to note is the difference between the "horrible" and "useful" phase portraits (**Figure 10**). When phase portraits plot variables which change very quickly, or even instantaneously, it is often necessary to smooth the data first. In this case, the "horrible" phase plot is not very bad (they can be much worse than this one), but the "useful" phase portrait, which is based on smoothed sensor data, is still better, as it makes the patterns more clear.

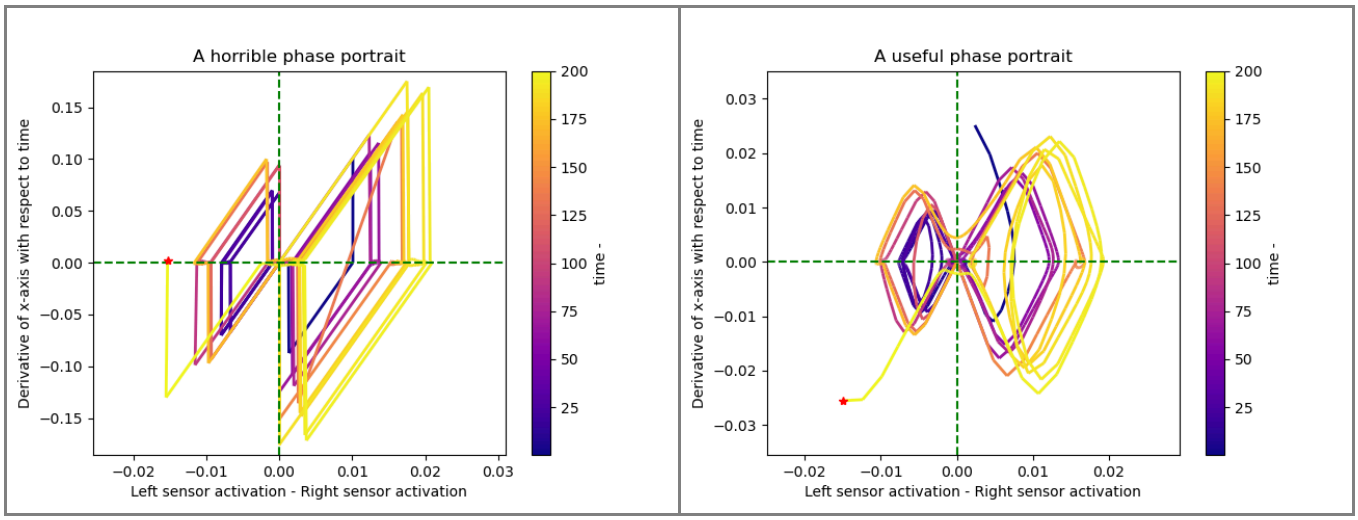


Figure 10. Phase portraits

Reflecting on your results

As mentioned above, part of our reason for this lab is that it will help us get to more interesting labs later. However, the lab has value in its own right, and as such is worth reflecting on. In your second assignment for this module, you will be expected to collect some results, analyse and then discuss them, so you can consider this your first opportunity to practice the discussion part. Unlike in your coursework assignments (which you ***must*** complete on your own), you are welcomed and strongly encouraged to discuss your lab class results with each other, as well as with me & Haoyue. You can discuss in person in classes, or on the padlet (at the bottom of this page), if you prefer.

Questions worth asking and discussing with each other and the teaching team include:

- What, if anything, did you have to add to your controllers which Braitenberg did not describe?
- Apart from any omissions already noted when answering the above, did Braitenberg make any mistakes in describing his vehicles and their behaviours?
- To what extent did your results confirm or contradict your first impression of Braitenberg's thought experiments?

- Are Braitenberg's vehicles models, or do they have some other status? And what about our simulated robots?
- Which part of the simulated world dominates in determining the robot's behaviour: its controller, its body, or its environment?
- How did the parameters constrain the kinds of controller you were able to implement?
  - How many additional parameters do you estimate you would need to implement more interesting behaviours?
- How do you think the complexity of the vehicle's behaviour relates to the number of parameters?
  - Do you think that the relationship between the number of parameters and the complexity of a system's behaviour will be the same for all systems?

References

[1] Braitenberg, V. (1986). *Vehicles: Experiments in synthetic psychology*. MIT press, Cambridge, MA.