Chapter 6
Braitenberg Vehicles

6.1 A brief Introduction

Valentino Braitenberg was a neuroscientist, a cybernetician and a musician. In his book 'Vehicles: Experiments in Synthetic Psychology, he explains how when investigating various structure in the animal brain, he realized that these structures could be interpreted as "pieces of computing machinery". He then goes on to create a number of *vehicles* to represent the animals, together with simple interconnexions between sensors and motors. Then, to describe the behaviour of these vehicles, he uses terms from psychology, such as *fear, aggression, love*. In total he discusses 14 vehicles, though we shall look only at two or three of these. The diagrams in his book show connexions between sensors and motor actuators, hooked up using individual discrete units of processing reminiscent of neurons. Of course, he was a neuroscientist! The history of computation shows developments of models of the brain and computing algorithms and structures have moved forward in lockstep. There are therefore strong links today between computing, neuroanatomy, neurophysiology and psychology.

We shall take a neural-central approach to discussing these vehicles using a novel neural-circuit computing paradigm we have invented. Of course, it is possible to use a wholly *procedural* approach to coding these vehicles, and we shall start with this. But first, let's have a look at some of Braitenberg's vehicles.

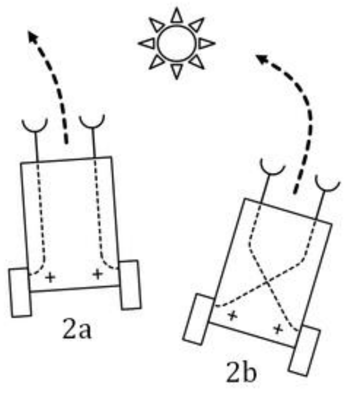


Figure 6.1 Vehicles 2a and 2b.

6.2 Some Vehicles

Let's start by looking at the structure of vehicle 2 which comes in two flavours vehicle 2a and vehicle 2b shown in Fig. 6.1. Each has two sensors (eyes) and two actuators (motors). The connexions are shown as dotted lines and the '+' symbol indicates that a signal coming down that wire from eye to motor has an *excitatory* effect on that motor; the larger the signal the faster the motor moves. Also we must understand how the light sensors work; simply the closer the sensor to a light, the larger the output signal produced by the light.

So for vehicle 2a, the right eye is closer to the light than the left eye so it outputs a larger signal. The right motor therefore receives a larger signal than the left, so the right motor turns faster (since this signal is *excitatory*). This makes the robot veer away from the light as shown in the diagram. Braitenberg attributed this vehicle with *fear*.

It's easy to understand why vehicle 2b veers towards the light; here the left eye gets more light and sends out a larger signal, but because of the ‘crossed’ connexions, the right motor receives this and turns faster than the left motor. Braitenberg called this *aggression* since the vehicle turns towards the light and speeds up, since both eyes receive more light as it is approached. Eventually the vehicle will collide with the light and abruptly halt as its eyes can’t see backwards.

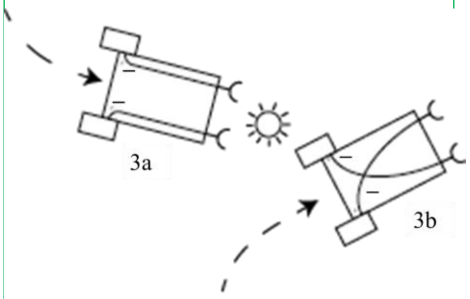


Figure 6.2 Vehicles type 3a and 3b. Note the ***inhibitory*** *connexions.*

Let’s have a look at vehicle 3 shown in Fig.6.2. The connexions resemble vehicle 2, but the behaviour is totally different. That’s because the connexions are *inhibitory*. What does that mean? Well if the signal into a motor is made larger, then the motor rotates slower, so the opposite of an *excitatory* connexion.

Consider first vehicle 3b. When it approach the light from the left, the left eye receives more light and generates a larger output signal than the right eye. This is passed to the right motor which slows down, since the signal is *inhibitory*. The diagram shows vehicle 3b having progressed beyond the point light, that’s because it has the attribute of an *explorer*, once it has moved to, and beyond, that light, then it moves off to find another one (to avoid). Note we are assuming that both motors have some drive to keep them moving in the absence of any light, this *tonic* signal must be present if the *inhibitory* connexion can work effectively.

Now vehicle 3a; as it moves towards the light, the signal from the eyes get stronger and so they increasingly *inhibit* the motors, slowing them down. So the vehicle approaches the light, slows down and stops. Braitenberg gave the attribute of *love* to this robot. When the vehicle is not head-on (say it’s off to the left) then the right eye gets more light and inhibits the right motor more, making it slower. So the vehicle turns towards the light.

6.3 Procedural coding of Behaviour

Note this is not our preferred way of doing things, we strongly advocate the neural approach, however thinking procedurally will get us going, and will provide a platform for comparison.

We shall assume that we have some code to read the signal from the eyes; here is how we would do it using an Arduino, where LDR_L and LDR_R are the Arduino pins where the left and right eye sensors are connected.

```
eyeL = (float)analogRead(LDR_L);
eyeR = (float)analogRead(LDR_R);
```

We also need some code to drive the motors. This could look like this.

```
driveServos(driveL,driveR);
```

Now we have to decide how we are going to connect the eyes to the motors. First we must realize that we may have signal of different sizes; our typical LDR-based eyes have an output in the range of 100 (dark) to 620 (light), and our servo-motors require a drive in the range of 10 (slow) to 60 (fast). So we have to do some mathematical *mapping* of the sensor values onto the motor values. This is illustrated, conceptually in Fig.6.3. There many ways to achieve such a mapping; we shall have a look at just two.

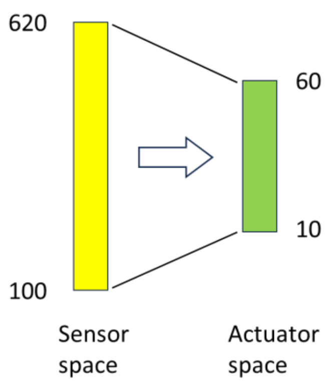


Figure 6.3 Mapping from sensor space (range of values) to actuator space (different range of values)

6.3.1 Linear Interpolation

A linear mapping preserves sizes of *intervals* of values between the two spaces. So if a sensor interval 100 – 200 (size 100) is mapped onto an actuator interval of 10 – 20 (size 10), then another sensor interval 300 - 400 (size 100) could map onto the actuator interval 30 – 40 (size 10). So equal intervals in one space have corresponding intervals in the other space which are equal. There is no *distortion* in the mapping, it is *linear*.

The technique is shown in Fig.6.4. There's lots going on here, but our goal is to input a sensor reading s and to find the corresponding actuator value a. The lower and upper sensor values s_L and s_H and the corresponding actuator values a_L and a_H are shown; these are obtained by experiment.

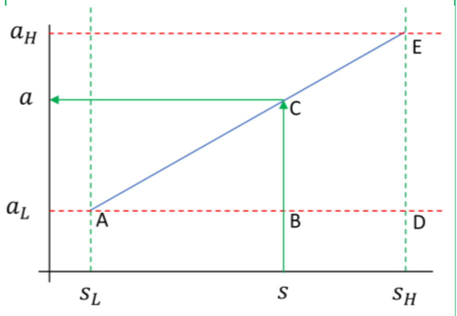


Figure 6.4 Linear mapping from sensor space (horizontal axis) to actuator space (vertical axis)

We need to get a mathematical expression for the mapping which we can implement in our computer code. Look at Fig.6.4 where you will see two *similar* triangles ABC and ADE. Since these are similar, the ratios of their lengths to their heights must be the same, so we have,

$$\frac{BC}{AB} = \frac{DE}{AD}$$

which gives us,

$$\frac{a - a_L}{s - s_L} = \frac{a_H - a_L}{s_H - s_L}$$

which we can solve for a in terms of s,

$$a = a_L + (s - s_L)\frac{(a_H - a_L)}{(s_H - s_L)}$$

Remember all the symbols with subscripts are known values, so when we plug in our measured eye signal s out pops the associated actuator value a. Here's some code that does this mapping, where we use the word **drive** for the actuator (motor) drive signal and the word **eye** for the eye sensor value.

```
driveOutput = _driveLow + (_eyeInput - _eyeLow)*(_driveHigh - _driveLow)/(_eyeHigh - _eyeLow);
```

6.3.2 Logarithmic Mapping

We don't have to use a linear mapping, we can choose any function we like depending on our needs, or indeed no function at all, but use a load of data points. However one situation we often encounter makes use of a *logarithmic* mapping shown in Fig.6.5. This function is a curve which flattens out for high values. Look at the two triangles on the figure, they both have the same horizontal extent, i.e., same range of sensor values. But the corresponding actuator range for the lower sensor values is much larger than the range for the higher sensor values. This means that when the light is not so bright, the motors respond with a larger speed change then when the light is bright. You can see how this could be useful for a Braitenberg vehicle; when it is further from the target light, the vehicle will turn faster than when it is closer to the light, so it will be able to located the target light more easily.

The maths to establish this mapping is quite straightforward and is presented in the box below.

Assume the desired function has the following form where *A* and *B* are two coefficients which we must find

$$a = f(s) = A\log(s) + B$$

Inserting lower and upper values,

$$f(s_L) = a_L = A\log(s_L) + B$$
$$f(s_H) = a_H = A\log(s_H) + B$$

Subtract and solve for A

$$A = \frac{a_H - a_L}{\log(s_H) - \log(s_L)}$$

Add and solve for *B*

$$B = \frac{1}{2}[a_H - a_L - A(\log(s_H) + \log(s_L))]$$

And that's how we do that. We can calculate *A* and *B* and use these in the first expression to obtain our mapping.

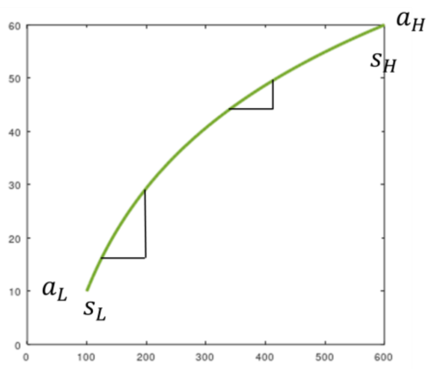


Figure 6.5 Logarithmic mapping which is more sensitive at lower light levels.

Here's some code to implement the log mapping,

```
A = (_driveHigh - _driveLow)/(log(_eyeHigh) - log(_eyeLow));
B = (_driveLow + _driveHigh) - A*(log(_eyeLow)+log(_eyeHigh));
B = B/2;

driveOutput = A*log(_eyeInput) + B;
```

6.3.3 Putting it all together.

Now we can combine the code snippets from the start of this section with an interpolation routine and make the eye-motor connexions. Let's start by looking at the code. Lines 4&5 we read the eye signals. Then, lines 7&8 we assign these to intermediate variables, the inputs to the interpolators. We connect left to left and right to right. So we are coding Vehicle 2a. Then we do the interpolations, lines 10&11 and finally use the outputs to drive the servos, line 13.

```
3
4       eyeL = (float)analogRead(LDR_L);
5       eyeR = (float)analogRead(LDR_R);
6
7       interpInL = eyeL;
8       interpInR = eyeR;
9
10      driveL = interpolate(interpInL,eyeLow,eyeHigh,driveLow,driveHigh);
11      driveR = interpolate(interpInR,eyeLow,eyeHigh,driveLow,driveHigh);
12
13      driveServos(driveL,driveR);
```

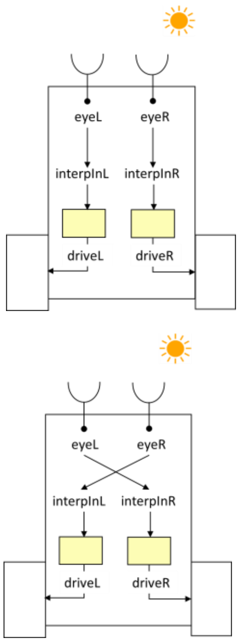


Figure 6.6 Connexions for vehicle 2a (top) and vehicle 2b (bottom). Yellow blocks are the interpolators.

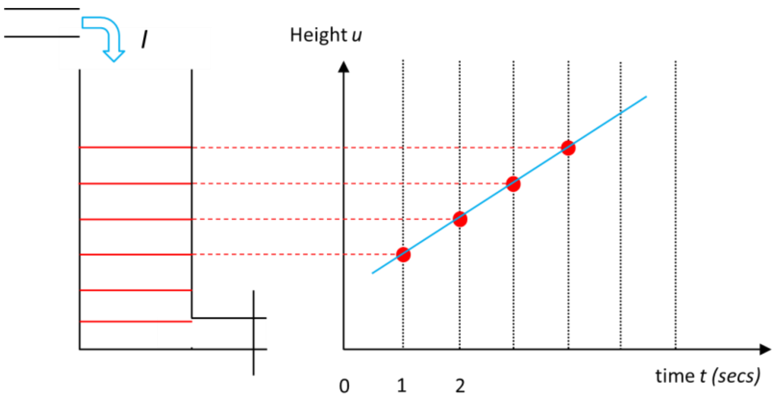
We can show this diagrammatically in Fig.6.6 where the structure of both vehicles 2a and 2b are shown. You can see the code variables, the yellow blocks are the interpolators and the arrows show the data flow for this procedural architecture.

6.4 Neural Circuits

Braitenberg clearly had neurons as functional units in mind, so that's what we shall look at here. We take the simplest model of a neuron which correctly captures its behaviour; it's desirable to keep things simple since ultimately we shall need to code our neuron model on the Arduino which has limited memory resource. The model is called the 'leaky integrator' model.

6.4.1 The 'Leaky Integrator' model

A neuron has an input which we can think of as a flow, and it accumulates that flow over time, which produces its *state* variable. Let's take the analogy of water flowing into (and out of) a bucket with a leak. The diagram below shows water entering the bucket (with the leak plugged) at a constant rate *I*.

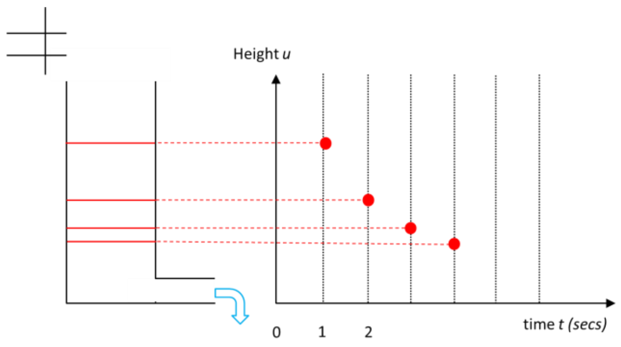


The height of the water is shown steadily increasing, the height at various times is indicated. We can write this mathematically, in each time interval Δt the height increases by the same amount Δu (we use the symbol u for the neuron state). The speed of height change is just

$$\frac{\Delta u}{\Delta t} = I$$

Where the left hand side means the increase in height per second, and the I on the right is the *cause* of the height increase, i.e. the water flow rate into the bucket.

Now suppose the bucket is nearly full so we turn off the input, and also open the leak so that water flows out, and the height starts to decrease. The diagram below details this.



The height falls following an *exponential* curve, the amount leaving reduces with time. This is a pressure effect; at first there is a lot of pressure on the water at the bottom (where it is leaving) but as the height decreases so does the pressure, and hence the flow gets smaller. We can write the change in height as

$$\frac{\Delta u}{\Delta t} = -u$$

Look at the right hand side, the *cause* of the change; it is negative which means the height is decreasing, and the rate of decrease is proportional to u which is the pressure effect.

Now we can combine the two expressions where water is flowing in and leaking out,

$$\frac{\Delta u}{\Delta t} = I - u$$

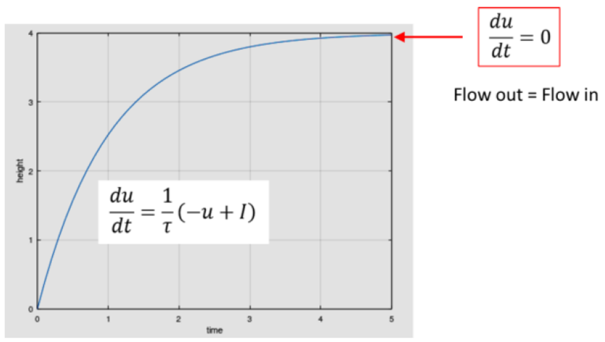
Of course, we have not included tons of physics in this model such as pipe size, resistance and gravity, but that would muddy the water. But there is one complication which we cannot side-step. In general the rate of change may depend on u in a more complex way, perhaps we need to use u^2 or other more fun dependencies. In this case, the model only works where Δt becomes very (infinitesimally) small. To remind us of this, we replace the symbol

Δt with dt where the little d reminds us that the change is infinitesimal. The equation for our leaky integrator neuron is now

$$\frac{du}{dt} = \frac{1}{\tau}(-u + I)$$

Here we have snook in a parameter τ (pronounced 'tauw') which tells us how fast the neuron responds; a large τ means the response is slow (small pipe diameter) and a small τ means a fast response (large pipe so water flows quickly out). The units of τ are seconds.

The graph below shows a solution of the above expression.



Initially the bucket is empty, so it fills up quickly since the rate of exit due to the leak is small. As the bucket fills up, the pressure increases so the leak flow out gets larger, and after a large time to rate of leak equals the rate of input flow, so the height does not change. At this point the bucket is in equilibrium so that

$$\frac{du}{dt} = 0 \qquad u = I$$

In the above example $I = 4$ so the equilibrium height is $u = 4$.

6.4.2 Adding two signals.

A circuit which can add two signals is shown in Fig.6.7, since there are now two inputs, we can extend the above expression,

$$\frac{du_1}{dt} = \frac{1}{\tau}(-u_1 + I_1 + I_2)$$

At equilibrium the above expression tells us

$$u_1 = I_1 + I_2$$

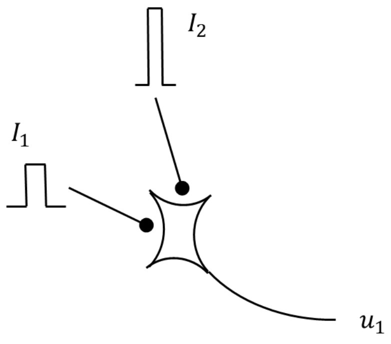


Figure 6.7Addition of two input signals I1 and I2.

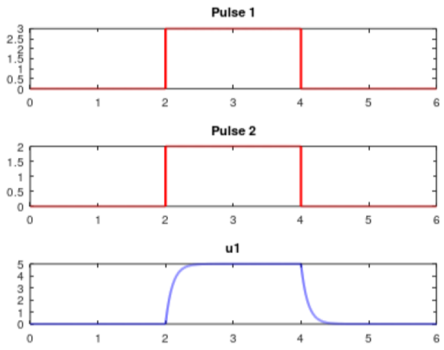


Figure 6.8 Addition of two input signals.

Showing that the circuit clearly adds the input signals, an example is given in Fig.6.8.

A circuit to subtract two signals is shown in Fig.6.9. Here one signal comes in as an *inhibitory* input, the larger the input the more it reduces the neuron's state. You could think of this as a little pump which is pumping water out of our bucket analogy. The equation for this case is

$$\frac{du_1}{dt} = \frac{1}{\tau}(-u_1 + I_1 - I_2)$$

with the equilibrium solution

$$u_1 = I_1 - I_2$$

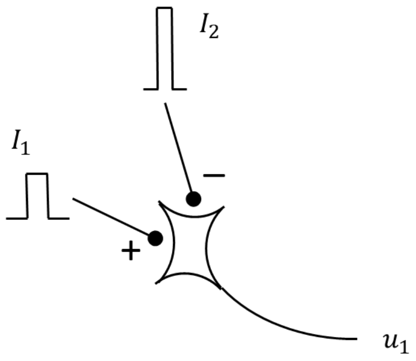


Figure 6.9 Neural circuit for subtraction .

6.4.3 Multiplication by Shunting Inhibition

Biological neurons achieve multiplication by a slightly different method, shown in Fig.6.10. Here the input $I_1$is fed into the neuron, but the second input I_2 acts on the incoming signal I_1 directly before it reaches the neuron body. This is called a *shunting* connexion and achieves multiplication by the following expression,

$$\frac{du_1}{dt} = \frac{1}{\tau}(-u_1 + I_1 * I_2)$$

with equilibrium solution

$$u_1 = I_1 * I_2$$

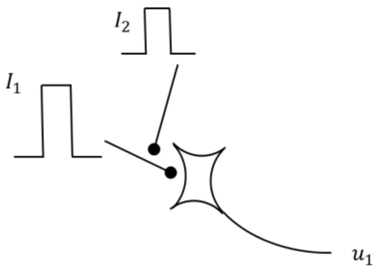


Figure 6.10 Neural multiplication by shunting inhibition.

6.4.4 The Sigmoid Output Function

So far, we have considered how a neuron processes its inputs by addition, subtraction or multiplication. Following this processing, the neuron passes its value of $u_1$through an output function, and it's this signal which is then passed on to other neurons. The idea is shown in Fig.6.11where the neuron area to the left of the dotted line does the processing, and the right area calculates the output

$$o_i = f(u_i)$$

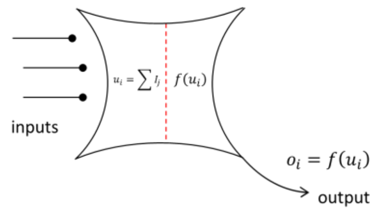


Figure 6.11 Neuron showing inputs which are processed followed by the output function.

The most common form of the output function is the *sigmoid* which is defined as follows and drawn in Fig.6.12.

$$f(u) = \frac{1}{1 + e^{-(u-C)/D}}$$

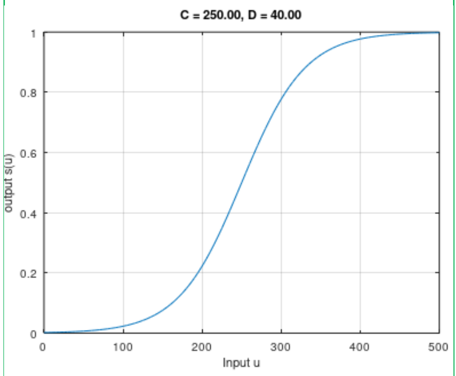
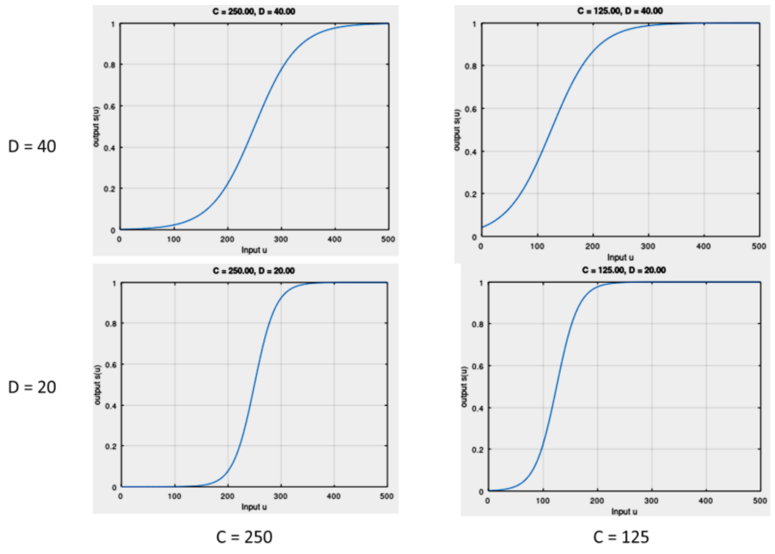


Figure 6.12 Example of a sigmoid curve showing that the output it limited to the range 0.0 - 1.0.

Here *A* and *B* are parameters which define the shape of the curve. First let's have a general think about the sigmoid. We have chosen a rather large input range of 0.0 – 500.0, and the sigmoid curve maps this to an output range of 0.0 – 1.0. In fact, the sigmoid created using the above expression will map *any* input range into an output range of 0.0 – 1.0. This is very useful, since it makes using the output signal rather easy as we shall see.

Now we need to understand the effect of the parameters *C* and *D*. Well, *C* defines the input value where the output is at its mid-point 0.5. In Fig.12, we have *C = 250* which is half the input range 0 – 500, so an input of 250 gives an output of 0.5. That's also useful. What about *D*? Well, this defines the *slope* or *gradient* of the curve; a small value of *D* means the curve rises quickly and a larger value means it rises slowly. The diagrams below show curves for two values of *C* and *D*.



6.5 Arduino Neural Circuit Controller

Let's consider one vehicle, type 2b. You will remember that the left eye excites the right motor and vice versa to obtain the desired behaviour. We can construct a neural circuit using just four neurons, two connected to the eyes, and two to drive the motors. The circuit is shown in Fig.6.13.

Neurons u_0 and u_1 are driven by signals from the left and right eye respectively, they also receive an ‘offset’ O which is the background ambient light level. You can see these neurons subtract the ambient light level from that of the point source. The red squiggles at the outputs of these neurons show that we use a sigmoid to generate the outputs.

The outputs are then fed into motor neurons u_2 and u_3 but before these output signals arrive, they are both multiplied by parameter D which is the amount of drive we use to drive the motors. Again the outputs are passes through sigmoids.

You will also note that the raw signals from the eyes are passed through a function. This converts the electrical signals from the LDR sensors into *luminance* values, so the eye neurons receive information about the light levels.

Now let’s fill in some important details by looking at some snippets from the Arduino code. The code is organized in a sequential manner, from eyes to motors. The input signals are **inL** and **inR** and these are passed through an exponential function to generate the luminance values, the two constants have been found by a separate calibration experiment.

```
inLexp = 12.683*exp(inL*0.0064);
inRexp = 12.683*exp(inR*0.0064);
```

Next comes the retinal neural layer. One neuron for each eye

```
dudt[0] = (-u[0] + inLexp - offsetL) / tau;
dudt[1] = (-u[1] + inRexp - offsetR) / tau;
u[0] += dudt[0] * deltaT;
u[1] += dudt[1] * deltaT;
outL1 = 1.0 / (1 + exp(-(u[0] - A) / B));
outR1 = 1.0 / (1 + exp(-(u[1] - A) / B));
```

You can see that the offsets are subtracted from the luminance inputs. The computed values of **u[0]** and **u[1]** are then passed through sigmoid function, where A = 100 and B = 25. The value of A was chosen from a measured maximum **u[ ]** value of 200. The value of B was chosen from experience. The outputs of the sigmoids lie in the range 0.0 – 1.0.

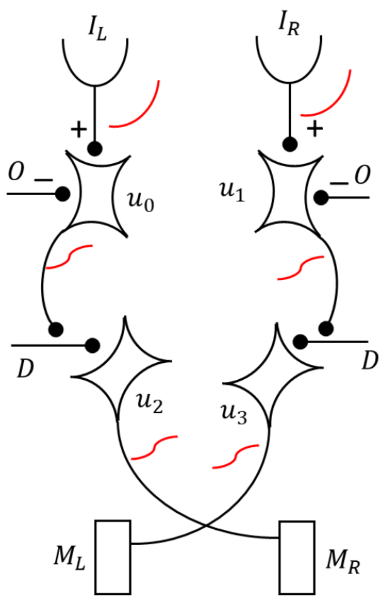


Figure 6.13 Neural circuit for Braitenberg vehicle 2b.

The outputs from the sigmoids are then passed into the motor neurons,

```
dudt[2] = (-u[2] + maxDrive * outL1) / tau;
dudt[3] = (-u[3] + maxDrive * outR1) / tau;
u[2] += dudt[2] * deltaT;
u[3] += dudt[3] * deltaT;

outL2 = maxDrive / (1 + exp(-(u[2] - C) / D));
outR2 = maxDrive / (1 + exp(-(u[3] - C) / D));
```

First you see the incoming values are multiplied by **maxDrive** which is typically set to 60 for the Parallax robot. Then comes another sigmoid where C is set to 25 (about half of maxDrive) and D is set to around 16. Note that the calculation of the sigmoids has **maxDrive** in the numerator rather than 1.0, in other words the outputs of the sigmoids lie in the range 0 – **maxDrive** which will make the robots happy.

Note that identical code will work with the Webots simulation. That’s the beauty of using the C-language!