6

Insect-Like Behaviors

So you gotta let me know
Should I stay or should I go?
From the song, *Should I stay or should I go*, Mick Jones (The Clash), 1982.

Opposite page: Ladybug
Photo courtesy of Jon Sullivan (www.pdphoto.org)

Designing robot behaviors is a challenging, yet fun process. There isn't a formal methodology or a technique that one can follow. It involves creativity, the ability to recognize the strengths and limitations of the physical robot, the kind of environment the robot will be carrying out the behavior, and of course the knowledge of available paradigms for programming robot behaviors. Creativity is essential to the design of robot behaviors. You have already seen how even a simple robot like the Scribbler can be programmed to carry out a diverse range of behaviors. We have also spent a considerable effort so far in exploring the range of possible functions a robot can perform. Where a robot is placed when it is running can play an important role in exhibiting a programmed behavior successfully. In this chapter, we take a different look at robot behaviors.

Braitenberg Vehicles

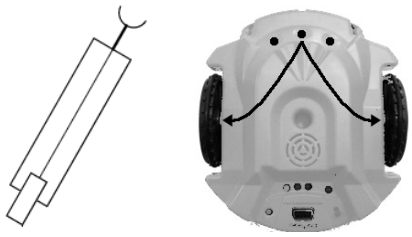
In 1984, Valentino Braitenberg wrote a book titled: *Vehicles: Experiments in Synthetic Psychology* (MIT Press). In it, he describes several thought experiments that are centered around the creation of simple vehicles with very simple control mechanisms that exhibit seemingly complex behavior. The purpose of the thought experiments was to illustrate some fundamental insights into the internal structure of animal (and human) brains. Each experiment involves a descripion of a simple vehicle that is endowed with a small suite of sensors (much like our Scribbler robot) and how the sensors can be connected to the motors of these imaginary vehicles in ways that parallel neurological connections in animals. He shows how the resulting vehicles are capable of complex behaviors which can be described as: fear, aggression, love, logic, free will, etc.

One central theme underlying Braitenberg's experiments is the demonstration of what he calls the *Law of uphill analysis and downhill invention*: It is much more difficult to guess the internal structure of an entity just by observing its behavior than it is to actually create the structure that leads to the behavior. That is, trying to postulate the internal structure purely by observing certain behavior is an uphill (harder) task whereas trying to create an entity that exhibits a certain behavior is a downhill (easy) task. While all of Braitenberg's

vehicles were imaginary and not really designed to be actually fabricated people have found it a fun and intellectually interesting exercise to create them. Personal robots like Scribblers make perfect platforms to do this and in what follows we will describe some of Braitenberg's (and Braitenberg-type) vehicles and design robot behaviors based on them.

Vehicle 1: Alive

Vehicle#1: Alive



The first vehicle Braitenberg describes has one sensor and one motor. The value transmitted by the sensor directly feeds into the motor. If the value being reported by the sensor is a varying quantity (say light), the vehicle will move at a speed proportional to the amount of quantity being detected by the sensor.

A schematic of the vehicle is shown above on the left. In order to design this vehicle using the Scribbler, you can use the center light sensor and connect what it reports directly to both motors of the robot. This is shown on the right. That is, the same light reading is directly controlling both the motors by the same amount. As you have already seen, there are a many different ways to specify motor movement commands to the Scribbler. Suppose the value obtained from the center light sensor is `C`, you can control both motors using this value by using the command:

```
motors(C, C)
```

Alternately, you can also use the `forward` command:

```
forward(C)
```

Now that we know how the internal structure of this vehicle looks, we can start to write a program that will implement it. But, before we get there, we need to sort out a small issue of compatibility: light sensors report values in

the range 0..5000 whereas motors and movement commands take values in the range -1.0 to 1.0. In this example, we are only concerned with movements that range from a complete stop to full speed forward, so the values range from 0.0 to 1.0. We have to computationally normalize, or map the light sensor values in this range. A first attempt at this would be to write a function called `normalize` that operates as follows:

```
def normalize(v):
   # normalizes v to range 0.0 to 1.0
```

Once we have this function, we can write the behavior for the vehicle as follows:

```
def main():
    # Braitenberg vehicle#1: Alive

    while True:
        L = getLight("center")
        forward(normalize(L))

main()
```

Normalizing Sensor Values

It is time now to think about the task of the `normalize` function. Given a value received from a light sensor, it has to transform it to a proportional value between 0.0 and 1.0 so that the brighter the light, the higher the value (i.e. closer to 1.0). Vehicle#1 moves in proportion to the amount of light it receives. This is a good time to revisit the `senses` function of Myro to look at the values reported by the light sensors. Go ahead and do this.

After examining the values returned by the light sensors you may notice that they report small values (less than 50) for bright light and larger values (as large as 3000) for darkness. In a way, you can say that the light sensor is really a darkness sensor; the darker it is the higher the values reported by it. The light sensors are capable of reporting values between 0 and 5000. Now,

we can certainly calibrate or normalize using these values using the following definition of `normalize`:

```
def normalize(v):
   # Normalize v (in the range 0..5000) to 0..1.0, inversely

   return 1.0 - v/5000.0
```

That is, we divide the value of the light sensor by its maximum value and then subtract that from 1.0 (for inverse proportionality). Thus a brighter light value, say a value of 35, will get normalized as:

```
1.0 - 35.0/5000.0 = 0.9929
```

If 0.9929 is sent to the motors (as in the above program), the robot would move full speed forward. Let us also compute the speed of the robot when it is in total darkness. When you place a finger on the center sensor, you will get values in the 2000-3000 range. For 3000, the normalization will be:

```
1.0 - 3000.0/5000.0 = 0.40
```

The robot will still be moving, although at nearly half the speed. Most likely, you will be operating the robot in a room where there is sufficient ambient light. You will notice that under ambient daylight conditions, the values reported by the light sensors are in the 150-250 range. Using the above normalization you will get:

```
1.0 - 200.0/5000.0 = 0.9599
```

That is almost full speed ahead. In order to experience the true behavior of the above vehicle, we have to use a normalization scheme that takes into account the ambient light conditions (they will vary from room to room). Further, let us assume that in ambient light conditions, we will watch the robot respond to a light source that we will control. A flashlight will work nicely. So, to make the robot appropriately sensitive to the flashlight under ambient light conditions you can write a better version of normalize as follows:

```
def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient
```

That is, the darkest condition is represented by the ambient light value (`Ambient`) and then normalization is done with respect to that value. You can either set the ambient value by hand, or, a better way is to have the robot sense its ambient light at the time the program is initiated. This is the same version of normalize that you saw in the previous chapter. Now you know how we arrived at it. The complete program for Vehicle#1 is shown below:

```
# Braitenberg Vehicle#1: Alive
from myro import *
initialize("com"+ask("What port?"))

Ambient = getLight("center")

def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient

def main():
    # Braitenberg vehicle#1: Alive

    while True:
        L = getLight("center")
        forward(normalize(L))
```

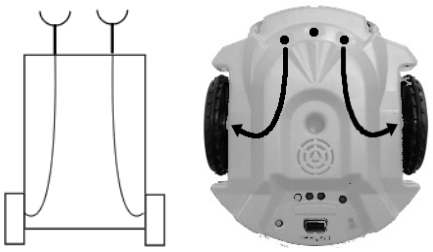
Do This: Implement the program above and observe the robot's behavior. Does it respond as described above?

You may have also noticed by now that the three light sensors are not necessarily closely matched. That is, they do not report exactly the same values under the same conditions. When writing robot programs that use multiple light sensors, it is a good idea to average the values returned by all the light sensors to represent the ambient value. Modify the program above to

use the average of all three values as the ambient value. There shouldn't be a noticeable difference in the robot's behavior. However, this is something you may want to use in later programs.

Vehicle 2: Coward and Aggressive

Vehicle 2a: Coward



The next set of vehicles use two sensors. Each sensor directly drives one motor. Thus the speed of the individual motor is directly proportional to the quantity being sensed by it sensor. There are two possible ways to connect the sensors. In the first case, Vehicle2a, the sensor on each side connects to the motor on the same side. In the other case, Vehicle2b, the connections are interchanged. That is, the left sensor connects to the right motor and the right sensor connects to the left motor. Let us design the control program for Vehicle 2a first:

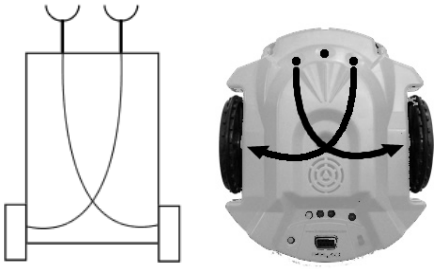
```
# Vraitenberg Vehicle#2a
from myro import *
initialize("com"+ask("What port?"))

Ambient = sum(getLight())/3.0

def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient

def main():
    # Braitenberg vehicle#2a: Coward

    while True:
        L = getLight("left")
        R = getLight("right")
        motors(normalize(L), normalize(R))
```

The structure of the above program is very similar to that of Vehicle1. We have modified the setting of the ambient light value to that of an average of the three light values. Also, we use the `motors` command, to drive the left and right motors proportional to the left and right light sensor values (after normalizing).

Do This: Implement the control program for Vehicle 2a as shown above. Observe the robot's behaviors by shining the flashlight directly in front of the robot and in each of the left and right sensors.

Next, write the control program for Vehicle2b as shown here. This requires one simple change from the program of Vehicle2a: switch the parameters of the `motors` command to reflect the interchanged connections. Again observe the behaviors by shining the flashlight directly ahead of the robot and also a little to each side.

Vehicle 2b: Aggressive



You will notice that the robots behave the same way when the light is placed directly ahead of them: they are both attracted to light and hence move towards the light source. However, Vehicle 2a will move away from the light if the light source is on a side. Since the nearer sensor will get excited more, moving the corresponding motor faster, and thereby turning the robot away. In the case of Vehicle 2b, however, it will always turn towards the light source and move towards it. Braitenberg calls these behaviors *coward* (2a) and *aggressive* (2b).

Controlling Robot Responses

It is often necessary, when designing and testing robot behaviors, to properly set up the robot's environment and the orientation of the robot in it. In simple cases this is easily achieved by first placing the robot in the desired

orientation and then loading and executing the program. However prior to the robot's actual behavior, the robot may need to perform some preliminary observations (for example, sensing ambient light), it becomes necessary to re-orient the robot properly before starting the execution of the actual behavior. This can be easily accomplished by including some simple interactive commands in the robot's program. The resulting program structure is shown below:

```
# import myro library and establish connection with the robot
# define all functions here (like, normalize, etc.)
# set values of ambient conditions

def main():
    # Description of the behavior...

    # Give user the opportunity to set up the robot
    askQuestion("Press OK to begin...", ["OK"])

    # Write your robot's behavior commands here
```

Do This: Modify the programs for Vehicles 1, 2a, and 2b to include the `askquestion` command above.

We have introduced a few basic programming patterns above that can be used in many robot programming situations. The thing to remember is that, at any point in the execution of a robot's program, you can also program appropriate interjections to perform various experimental or control functions. We will see several other examples later on.

Other Normalizations

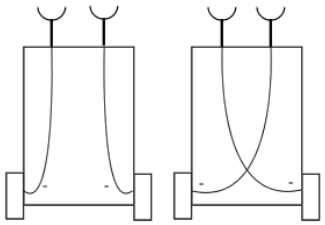
All the normalizations of light sensor values shown above were used to normalize the values in the range 0.0..1.0 in direct proportion to the amount of light being sensed. That is, the darker it is, the closer the normalized values are to 0.0 and the brighter it gets, the closer the normalized values get to 1.0. This is just one way that one can relate the quantity being sensed to the amount of speed applied to the robot's motors. You can imagine other

relationships. The most obvious of course is an inverse relationship: the darker it is the closer to 1.0 and vice versa. Braitenberg calls this *inhibitory* (as opposed to *excitatory*) relationship: the more of a quantity being sensed, the slower the robot's motors turn. As in Vehicles 2a and 2b above, there is choice of two kinds of connections: straight and crossed. These are shown below (a plus (+) sign next to a connector indicates an excitatory connection and a minus sign (-) represents an inhibitory connection):

Writing the `normalize` function for an inhibitory connection is quite straightforward:

```
def normalize(v):
    if v > Ambient:
        v = Ambient

    return v/Ambient
```

Vehicles 3a (Love) and 3b (Explorer)



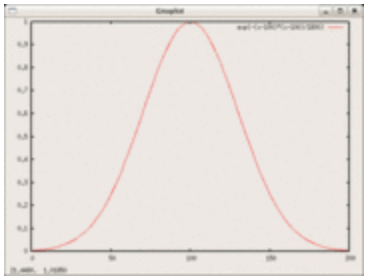
Braitenberg describes the behavior of the resulting vehicles as *love* (Vehicle 3a) and *explorer* (Vehicle 3b). That is, if you were to observe the behavior of the two vehicles, you are likely to notice that Vehicle 3a will come to rest facing the light source (in its vicinity) whereas vehicle 3b will come to rest turned away from the source and may wander away depending on the presence of other light sources.

In other variations on sensor value normalizations, Braitenberg suggests using non-monotonic mathematical functions. That is, if you look at the excitatory and inhibitory normalizations, they can be described as monotonic: more light, faster motor speed; or more light, slower motor speed. But consider other kinds of relationships for normalizations. Observe the function shown on the next page. That is, the relationship is increasing in proportion to sensory input but only up to a certain point and after that it decreases. Incorporating such relationships in vehicles will lead to more complex

behavior (Braitenberg describes them as vehicles having *instincts*). The following defines a normalization function, based on the curve shown:

$$f(x) = e^{-(x-100)^2/1800}$$

A Non-Monotonic Function



The above function is based on the simpler function:

$$f(x) = e^{-x^2}$$

which in the first definition is stretched to span the range 0..200 for values of x with 100 being the point where it reports the maximum value (i.e. 1.0). Mathematically this function is also known as the *bell curve* or a *Gaussian Curve* in general. A bell curve is defined in terms of a *mean (π)* and *standard deviation (σ)* as shown below:

$$f(x) = e^{-(x-\pi)^2/2*\sigma^2}$$

Thus, in the normalization function we are using 100 as mean and 30 as standard deviation. You can easily scale the curve for the range of sensor values you desire using the following `normalize` function.

```
def normalize(v):
        mean = Ambient/2.0
        stddev = Ambient/6.0
        if v >= Ambient:
                v = Ambient
        return exp(-(v - mean)**2 / 2*(stddev**2))
```

`exp(x)` is a Python function that computes the value of e^x. It is available in the Python `math` library. We will delve into the math library in more detail in the next chapter. In order to use the `exp` function as shown above you have to import the `math` library:

```
from math import *
```

There are of course several other possibilities that one could try: a step function; or a threshold; and any other mathematical combinations. The key idea is that there is a clear mapping of the range of sensor values to motor values in the range 0.0..1.0.

Robots using these normalizations and other variations are likely to exhibit very interesting and sometimes unpredictable behaviors. Observers unaware of the internal mapping mechanisms will have a hard time describing precisely the robot's behavior and will tend to use anthropomorphic terms (like, *love*, *hate*, *instincts*, etc.) to describe the behavior of robots. This is what an *uphill analysis* means.

Multiple Sensors

Adding several sensors enriches the design space for robot behaviors. As a designer, you now have a choice of different types of mappings: excitatory, inhibitory, or more complex; and connections: straight or crossed. Suddenly the resulting robot behavior will seem complex. On the Scribbler, for instance, in addition to light sensors, you also have the stall sensor, and the IR sensors. With the exception of light sensors, all of these other sensors are digital or threshold sensors that are either ON or OFF (i.e. they report values that are either 0 or 1 indicating the presence or absence or the thing they are sensing). In a way you can think that the digital sensors are already normalized, but it is still possible to invert the relationship if need be. You can design several interesting behaviors by combining two or more sensors and deciding whether to connect them straight or crossed.

Do This: In your design of vehicles 2a and 2b substitute the obstacle sensor in place of the light sensors. Describe the behavior of the resulting vehicles. Try the same for Vehicles 3a and 3b. Next, combine the behavior of the resulting vehicles with the light sensors. Try out all combinations of connections, as well as inhibitory and excitatory mappings. Which vehicles exhibit the most interesting behaviors?

More Vehicles

Here are descriptions of several vehicles that are in the spirit of Braitenberg's designs and also exhibit interesting behaviors. Using the concepts and programming techniques from above, try to implement these on the Scribbler robot. Once completed, you should invite some friends to observe the behaviors of these creatures and record their reactions.

Timid

Timid is capable of moving forward in a straight line. It has one threshold light sensor, pointing up. When the light sensor detects light, the creature moves forward, otherwise, it stays still. The threshold of the light sensor should be set to ambient light. That way, when the creature can "see" the light, it will move. When it enters a shadow (which can be cast by a hand or another object) it stops. If whatever is casting the shadow is moved, the creature will move again. Therefore, timid is a shadow seeker.

Indecisive

Indecisive is similar to Timid, except, it never stops: its motors are always running, either in forward direction, or in reverse direction, controlled by the threshold light sensor. When the light sensor detects light, it moves forward, otherwise, it moves backwards. When you run this creature, you will notice that it tends to oscillate back and forth at shadow edges. Thus, Indecisive is a shadow edge seeker.

Paranoid

Paranoid is capable of turning. This is accomplished by moving the right motor forward and moving the left motor in reverse direction at the same time. It has a single threshold light sensor. When the sensor detects light, it moves forward. When the sensor enters a shadow, it reverses the direction of its left motor, thus turning right. Soon the sensor will swing around, out of the

shadow. When that happens, it resumes its forward motion. Paranoid, is a shadow fearing creature.

This, That, or the Other

The `if`-statement introduced earlier in Chapter 5 is a way of making simple decisions (also called one-way decisions). That is, you can conditionally control the execution of a set of commands based on a single condition. The if-statement in Python is quite versatile and can be used to make two-way or even multi-way decisions. Here is how you would use it to choose among two sets of commands:

```
if <condition>:
   <this>
else:
   <that>
```

That is, if the `<condition>` is true it will do the commands specified in `<this>`. If, however, the `<condition>` is false, it will do `<that>`. Similarly, you can extend the `if`-statement to help specify multiple options:

```
if <condition-1>:
   <this>
elif <condition-2>:
   <that>
elif <condition-3>:
   <something else>
...
else:
   <other>
```

Notice the use of the word `elif` (yes, it is spelled that way!) to designate "else if". Thus, depending upon whichever condition is true, the corresponding `<this>`, `<that>`, or `<something else>` will be carried out. If all else fails, the `<other>` will be carried out.

Simple Reactive Behaviors

Using the three light sensors the robot can detect varying light conditions in its environment. Let us write a robot program that makes it detect and orient towards bright light. Recall from Chapter 5 that light sensors report low values in bright light conditions and high values in low light. To accomplish this task, we only need to look at the values reported by left and right light sensors. The following describes the robot's behavior:

```
do for a given amount of time
   if left light is brighter than right light
      turn left
   else
      turn right
```

Thus, by making use of the if-else statement, we can refine the above into the following:

```
while timeRemaining(30):
   if left light is brighter that right light:
      turnLeft(1.0)
   else:
      turnRight(1.0)
```

The only thing remaining in the commands above is to write the condition to detect the difference between the two light sensors. This can be done using the expression:

```
getLight('left') < getLight('right')
```

Do This: Write a complete program that implements the above behavior and test it on your robot.

You may have noticed that even in uniform lighting conditions sensors tend to report different values. It is generally a good idea to threshold the difference when making the decision above. Say we set the threshold to a difference of at least 50. That is, if the left and right sensors differ by at least 50 then turn

towards the brighter sensor. What happens if the difference is less than the threshold? Let us decide that in that case the robot will stay still. This behavior can be captured by the following:

```
thresh = 50

while timeRemaining(30):
   # Get sensor values for left and right light sensors
   L = getLight('left')
   R = getLight('right')

   # decide how to act based on sensors values
   if (L - R) > thresh:
      # left is seeing less light than right so turn right
      turnRight(1.0)
   elif (R - L) > thresh:
      # right is seeing less light than left, so turn left
      turnLeft(1.0)
   else:
      # the difference is less than the threshold, stay put
      stop()
```

Notice how we have used the variable `thresh` to represent the threshold value. This is good programming practice. Since the performance of sensors varies under different light conditions, this allows you to adjust the threshold by simply changing that one value. By using the name `thresh` instead of a fixed value, say 50, you only have to make such changes in one place of your program.

In the statements above, there is a pattern that you will find recurring in many programs that define robot behaviors using simple decisions:

```
while timeRemaing(<seconds>):
   <sense>
   <decide and then act>
```

Such behaviors are called *reactive* behaviors. That is, a robot is reacting to the change in its environment by deciding how to act based on its sensor values. A wide range of robot behaviors can be written using this program structure.

Below, we present descriptions of several interesting, yet simple automated robot behaviors. Feel free to implement some of them on your robot.

Simple Reactive Behaviors

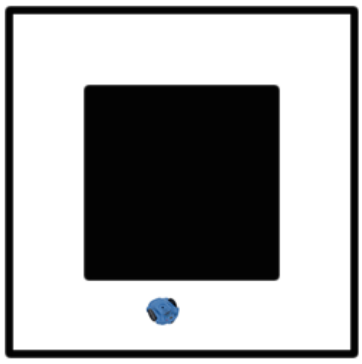
Most of the behaviors described below require selection among alternatives using conditional expressions and `if`-statements.

Refrigerator Detective: As a child did you ever wonder if that refrigerator light was always on? Or did it shut off when you closed the door? Well, here is a way to find out. Build a refrigerator detective robot that sits inside the fridge and tells you if the light is on or off!

Burglar Alarm Robot: Design a robot that watches your dorm door. As soon as the door opens, it sounds an alarm (beeps).

Wall Detector: Write a robot program that goes straight and then stops when it detects a wall in front. You will be using the IR sensors for this task.

Hallway Cruiser: Imagine your robot in an environment that has a walled corridor going around a 2 ft by 2 ft square box (see picture on right). Write a program that will enable the robot to go around this box. One strategy you can use is to have the robot go forward in a straight line until it bumps into a wall. After a bump it will proceed to make a 90 degree turn (you may need to have it go backwards a little to enable turning room) and then continue again in a straight line.



Measuring Device: You have calibrated your robot with regards to how far it travels in a given amount of time. You can use that to design a robot that

measures space. Write a program that enables a robot to measure the width of a hallway.

Follower: Write a robot program to exhibit the following behavior: The robot prefers to stay near a wall (in front). If it does not have a wall in front of it, it moves forward until it finds it. Test your program first by placing the robot in a play pen. Ensure that your program behaves as described. Next, place it on a floor and hold a blank piece of paper in front of it (close enough so the robot can detect it). Now, slowly move the paper away from the robot. What happens?

Designing Reactive Behaviors

Most of the robot behaviors that are implemented using the Braitenberg style rely on a few simple things: selecting one or more sensors; choosing the kind of wiring (straight or crossed); and selecting normalization functions for each sensor. While you can guess the behavior that may result from these designs the only way to confirm this is by actually watching the robot carry out the behavior. You also saw how, using if-statements you can design simple, yet interesting robot. In this section we will design additional reactive behaviors.

Light Following

To begin, it will be fairly straightforward to extend the behavior of the light orienting behavior from above into one that results in a light follower robot. That is, with a flashlight you will be able to guide the robot to follow you around. Again, you have to start by observing the range of values reported by the robot under various lighting conditions. If a flashlight is going to be the bright light source, you will observe that the light sensors report very low values when a light is shining directly on them (typically in the 0..50 range). Thus, deciding which way to go (forward, turn left, or turn right) can be decided based on the sensor readings from the three light sensors. The structure of the program appears as follows:

```
# Light follower

from myro import *
initialize(ask("What port?"))

# program settings...

thresh = 50
fwdSpeed = 0.8
cruiseSpeed = 0.5
turnSpeed = 0.7     # left turn, -0.7 will be right turn

def main():
    while True:
       # get light sensor values for left, center, and right
       L, C, R = getLight()

       # decide how to act based on sensor values
       if C < thresh:
          # bright light from straight ahead, go forward
          move(fwdSpeed, 0)
       elif L < thresh:
          # bright light at left, turn left
          move(cruiseSpeed, turnSpeed)
       elif R < thresh:
          # bright light on right side, turn right
          move(cruiseSpeed, -turnSpeed)
       else:
          # no bright light, move forward slowly (or stop?)
          move(cruiseSpeed/2, 0)
main()
```

Notice that, in the program above, we have decided to set values for light threshold (`thresh`) as well as movements to specific values. Also, in all cases, we are using the `move` command to specify robot movement. This is because the `move` command allows us to blend translation and rotation movement. Additionally, notice that regardless of the sensor values, the robot is always moving forward some amount even while turning. This is essential since the robot has to follow the light and not just orient towards it. In the case where there is no bright light present, the robot is still moving forward (at half the cruise speed).

Do This: Implement the light following program as described above and observe the robot's behavior. Try adjusting the value settings (for threshold as well as motor speeds) and note the changes in the robot's behaviors. Also, do you observe that this behavior is similar to any of the Braitenberg vehicles described above? Which one?

In the design of the light following robot above, we used a threshold value for detecting the presence of bright light. Sometimes it is more interesting to use differential thresholds for sensor values. That is, is the light sensor's value different from the ambient light by a certain threshold amount? You can use the `senses` function again observe the differences from ambient light and modify the program above to use the differential instead of the fixed threshold.

Here is another idea. Get several of your classmates together in a room with their robots, all running the same program. Make sure the room has plenty of floor space and a large window with a curtain. Draw close the curtains so the outside light is temporarily blocked. Place the robots all over the room and start the program. The robots will scurry around, cruising in the direction of their initial orientation. Now, slowly draw the curtains open to let in more light. What happens?

Avoiding Obstacles

Obstacles in the path of a robot can be detected using the IR sensors in front of the robot. Then, based on the values obtained, the robot can decide to turn away from an approaching obstacle using the following algorithm:

```
if obstacle straight ahead, turn (left or right?)
if obstacle on left, turn right
if obstacle on right, turn left
otherwise cruise
```

This can be implemented using the program below:

```
# Avoiding Obstacles

from myro import *
initialize(ask("What port?"))

# program settings...

cruiseSpeed = 0.6
turnSpeed = 0.5     # this is a left turn, -0.5 will be right
turn

def main():
    while True:
       # get sensor values for left and right IR sensors
       L, R = getIR()
       L = 1 - L
       R = 1 - R

       # decide how to act based on sensors values
       if L and R:
          # obstacle straight ahead, turn (randomly)
          move(0, turnSpeed)
       elif L:
          # obstacle on left, turn right
          move(cruiseSpeed, -turnSpeed)
       elif R:
          # obstacle on right, turn left
          move(cruiseSpeed, turnSpeed)
       else:
          # no obstacles
          move(cruiseSpeed, 0)
main()
```

As in the case of the light follower, observe that we begin by setting values for movements. Additionally, we have flipped the values of the IR sensors so that the conditions in the `if`-statements look more natural. Recall that the IR sensors report a 1 value in the absence of any obstacle and a 0 in the presence of one. By flipping them (using `1 - value`) the value is 1 for an obstacle

present and 0 otherwise. These values make it more natural to write the conditions in the program above. Remember in Python, a 0 is equivalent to `False` and a 1 is equivalent to `True`. Read the program above carefully and make sure you understand these subtleties. Other than that, the program structure is very similar to the light follower program.

Another way to write a similar robot behavior is to use the value of the stall sensor. Recall that the stall sensor detects if the robots has bumped against something. Thus, you can write a behavior that doesn't necessarily avoid obstacles, but navigates itself around by bumping into things. This is very similar to a person entering a dark room and then trying to feel their way by touching or bumping slowly into things. In the case of the robot, there is no way to tell if the bump was on its left or right. Nevertheless, if you use the program (shown below) you will observe fairly robust behavior from the robot.

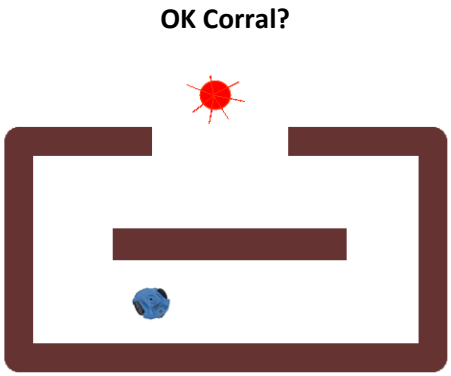
```
# Avoiding Obstacles by bumping

from myro import *
initialize(ask("What port?"))

# program settings...

cruiseSpeed = 1.0
turnSpeed = 0.5     # this is a left turn, -0.5 will be right
turn

def main():
    while True:

        if getStall():
           # I am stalled, turn (randomly?)
           move(0, turnSpeed)
        else:
           # I am not stalled, cruise on
            move(cruiseSpeed, 0)

main()
```

At times, you may notice that the robot gets stuck even when trying to turn. One remedy for this is to stop the robot, back up a little, and then turn.

Do This: Implement the program above, observe the robot behavior. Next, modify the program as suggested above (when stalled stop, backup, then turn).

Maze Solver: Create a simple maze for your robot. Place the robot at one end of the maze and use the obstacle avoidance programs from above (both versions). Does you robot solve the maze? If not, note if your maze is right handed or left handed (i.e. every turn is a right turn or left turn in the maze), or both. Modify the obstacle avoidance programs to solve the right-handed, left-handed mazes. How would you enable the robot to solve a maze that has both right and left turns?

Corral Exiting



Given that a simple obstacle avoidance program can enable a robot to solve simple mazes, we can also design more interesting behaviors on top of that. Imagine a corral: an enclosed area with maze like partitions and an entrance, with a light source at the entrance (see picture on right). Given the robot's position, can we design a behavior that will enable the robot to exit the corral?

One can design a solution for the specific corral shown here: follow a wall (any wall) until it sees bright light then switch to light seeking. Can the Scribbler be designed to follow a wall? Remember the Fluke dongle has left and right obstacle sensors that are pointing to its sides. Another approach will be to combine the obstacle avoidance behavior from above with the light

seeker behavior. That is, in the absence of any bright light, the robot moves around the corral avoiding obstacles and when it sees a bright light, it heads towards it. The hard part here will be to detect that it has exited the corral and needs to stop.

Summary

Braitenberg uses very simple ideas to enable people to think about the way animal and human brains and bodies are wired. For example, in humans, the optic nerves (as do some others) have crossed connections inside the brain. That is, the nerves from the left eye are connected to the right side of the brain and vice versa. Actually they cross over and some information from either side is also represented on the same side (that is there are straight as well as crossed connections). However, it is still a puzzle among scientists as to why this is the case and what, if any, are the advantages or disadvantages of this scheme. Similarly, observing the behaviors of Vehicles 2a and 2b one can easily see in them parallels in the behavior of several animals, like flies orienting towards light/heat sources. Simple robot behaviors can provide deep insights into complex behavior: that the observation and analysis of something is an uphill task if one doesn't know the internal structure. And, by constructing simple internal structures one can arrive at seemingly complex behaviors. These seemingly complex behaviors have also been shown to influence group behavior in insects (see the picture of article on next page). That is, robots that do not look anything like insects, and not too different in size than the Scribbler, can be used to influence insect behavior in many situations.

In this chapter, we have attempted to give you a flavor for the idea of synthetic psychology. At the same time you have also learned how to program internal structures in a robot brain and learned several techniques for robot control.

Story from *Science* Magazine, January 10, 2008

BEHAVIOR

Robot Cockroach Tests Insect Decision-Making Behavior

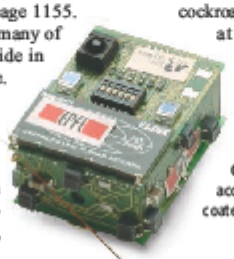
Science-fiction writers have long envisioned societies in which the boundaries between humans and lifelike droids blur and man and machine freely intermingle. José Halloy has taken the first steps toward creating that world, at least for insects. His tiny, autonomous robots lack legs, wings, and antennae, but they nonetheless pass muster with cockroaches. Indeed, these wheeled machines are so well accepted by the household pests that the robots become part of the insects' collective decision-making process, Halloy, a theoretical biologist at the Free University of Brussels, Belgium, and his colleagues report on page 1155. The robots persuaded many of their insect "peers" to hide in an unconventional place.

Halloy's innovative approach puts theories of collective behavior among insects into practice. "We can manipulate these behaviors very easily in a model, but doing so in experiments is often challenging," explains ethologist Jerome Buhl of the University of Sydney, Australia. Others have used remote-controlled robots to study animal

level of darkness of the shelter and the number and activity of its fellow roaches. Halloy's group then used this model to program robots designed by him and Francesco Mondada and other engineers at the École Polytechnique Fédérale de Lausanne, Switzerland.

The roaches usually ran away from the robots but not if the machines smelled like the insects. For the experiments, Halloy and Sempo covered the robots with a filter paper containing the pheromone equivalent of one cockroach.

Halloy initially programmed the robots to have the same darkness preference as the cockroaches, and they joined the cockroaches at whatever shelter the majority chose to rest in. Next, Halloy programmed the robots to prefer the lighter shelter. About 60% of the time, the robots tipped the group's preference



Can't we be friends? Cockroaches seem to accept this robot as one of their own once it's coated with pheromone.

in favor of the light shelter. "This is a true example of automated leadership," says David Sumpter of Uppsala

Background

All the numbered vehicles described here were developed in a set of thought experiments designed by Valentino Braitenberg in his book, *Vehicles: Experiments in Synthetic Psychology*, MIT Press, 1984.

Some of the other vehicles described here were designed by David Hogg, Fred Martin, and Mitchel Resnick of the MIT Media Laboratory. Hogg et al

used specialized electronic LEGO bricks to build these vehicles. For more details, see their paper titled, *Braitenberg Creatures*.

To read more about robots influencing insect behavior see the November 16, 2007 issue of *Science* magazine. The primary article that is discussed in the picture above is by Halloy *et al*, *Social Integration of Robots into Groups of Cockroaches to Control Self-Organized Choices, Science*, November 16, 2007. Volume 318, pp 1155-1158.

Myro Review

There were no new Myro features introduced in this chapter.

Python Review

The if-statement in Python has the following forms:

```
if <condition>:
   <this>

if <condition>:
   <this>
else:
   <that>

if <condition-1>:
   <this>
elif <condition-2>:
   <that>
elif <condition-3>:
   <something else>
...
...
else:
   <other>
```

The conditions can be any expression that results in a True, False, 1, or 0 value. Review Chapter 4 for details on writing conditional expressions.

Exercises

1. An even better way of averaging the ambient light conditions for purposes of normalization is to have the robot sample ambient light all around it. That is, turn around a full circle and sample the different light sensor values. The ambient value can then be set to the average of all the light values. Write a function called, `setAmbient` that rotates the robot for a full circle (or you could use time), samples light sensor values as it rotates, and then returns the average of all light values. Change the line:

```
Ambient = sum(getLight())/3.0
```

to the line:

```
Ambient = setAmbient()
```

Try out all of the earlier behaviors described in this chapter to see how this new mechanism affects the robot's behavior.

2. Design and implement a program that exhibits the corral exiting behavior described in this chapter.

3. Implement the refrigerator detective behavior described in this chapter.

4. Implement the Burglar alarm robot described in this chapter.

5. Implement the hallway cruiser behavior described in this chapter.

6. In addition to movements try to integrate music/sound output in your robot behaviors and observe how the addition of sounds amplifies the perception of the robot’s personality.