```
        print ("Humidity = %3.1f %%" % hdc1080.readHumidity())
        print ("-----------------")


        blynkUpdate(temperature, humidity)

        time.sleep(3.0)
```

This code updates your Blynk app every three seconds.

WARNING

If you update your Blynk app more than once a second, you may be disconnected from the server. Try to keep your request sends to less than ten values per second to be a good Blynk citizen.

4. **The last thing you need to do before you run the code is to replace the `'xxxx'` with your Blynk authorization code, which will look something like this: 445730794c1c4c8ea7852a31555f44444.**

Before:

```
BLYNK_AUTH = 'xxxx'
```

After:

```
BLYNK_AUTH = '445730794c1c4c8ea7852a31555f44444'
```

Note: You must use the authorization code you received by email (or by cutting and pasting from the app) otherwise, you will not connect to your app. The example code shown here will not work.

Breaking down the code

This code is very similar to the HDC1080 code from earlier in this chapter with the exception of the `blynkUpdate` code.

```
def blynkUpdate(temperature, humidity):
    print ("Updating Blynk")
    try:
```

Why do we have a 'try' here? Because sometimes the requests library will throw an error if the Internet is being funky.

Next, we set up the required `http` header for the requests library:

```
        put_header={"Content-Type": "application/json"}
```

THE REQUESTS LIBRARY

The python requests library is one of most useful libraries for communicating across the Internet using `http` requests.

It is designed to be used by humans to interact with `http` requests without exposing the complexity of the requests. Very Pythonic.

The requests library allows you to send HTTP/1.1 requests using Python. It also allows you to access the response data of the requests using Python.

The following code sets the number of digits to the right of the decimal point to 1 so we won't have long numbers of relatively meaningless digits because of the accuracy of the HDC1080:

```
        val = temperature
        put_body = json.dumps(["{0:0.1f}".format(val)])
```

The following code does the actual transfer of the data to the Blynk server in the form of an `http` request:

```
        r = requests.put(BLYNK_URL+BLYNK_AUTH+'/update/V0', data=put_body,
   headers=put_header)
```

Here we print out any exception to the terminal screen. This also helps you figure out if you have set the Blynk authentication code incorrectly:

```
  except Exception as e:
                print ("exception in updateBlynk")
                print (e)
                return 0
```

Next, let's run the program:

```
Sudo python3 myTemperature.py
```

You will see this type of output on your terminal screen:

```
-----------------
Temperature = 22.6 C
Humidity = 36.8 %
-----------------
```

```
Updating Blynk
------------------
Temperature = 22.5 C
Humidity = 36.8 %
------------------
Updating Blynk
```

Hit the Run key at the top-right of your Blynk app on the phone, and then watch your data start to come in. If you don't get data in a few seconds, check your authentication code and make sure you have started your app by hitting the Start button in the upper-right corner of the app.

Your results should look like Figure 3-18.

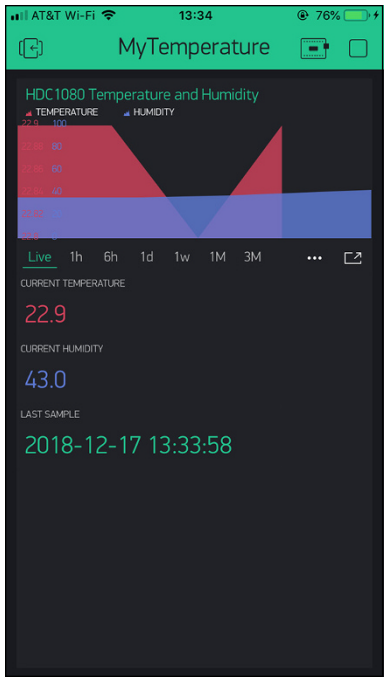

FIGURE 3-18: The MyTemperature app's Live view.

Note that the Live display on the graph can look a little funky, but your other displays will start filling in and looking really good.

Where to Go from Here

In this chapter, you have learned a lot about how to connect to the real world Python on your Raspberry Pi. We suggest these other interesting things to do, building upon your new expertise:

- Add more I2C sensors to your Raspberry Pi. There are hundreds of them.
- Try adding some digital sensors to your Pi, like PIR detectors to detect warm bodies like humans in front of your Pi.
- Add an I2C compass and accelerometer to your Pi.
- Build larger and more complex dashboards and show off your project with your friend (yes, you can do this by sharing your authentication code with your friends).
- Add a motor to make things move. Oh wait, we are doing that in the next chapter!

IN THIS CHAPTER

- How to make things move with Python
- Understanding DC motors and software
- Using a servo motor
- Making a stepper motor step

Chapter 4

Making Things Move with Python

Making things move around with Python is undeniably cool. With motors, physical computing goes to a whole new level.

Robots, microwaves, refrigerators, and electric cars all use electric motors to move around, blow air, pump coolant, and take you 60mph wherever you want to go. Electric motors are everywhere!

At its simplest, an electric motor is a machine that converts electrical energy into mechanical energy. In this chapter, we talk about DC (direct current) motors. Direct current is a single fixed voltage, like 9V or 5V or 3.3V. Alternating current (AC), on the other hand, is what you get out of your house outlets.

Interestingly, electric motors consume more than half of the electric energy produced in the United States.

Exploring Electric Motors

An electric motor is all about magnetism. All motors use magnets to create motion. All magnets have a north and a south pole. North to north and south to south repel each other whereas north and south attract. Clever people have figured out how to use this fact to create motion. We are all familiar with permanent magnets, like the ones you use to hang things on the front of your refrigerator. However, you can also create magnets by running a current around a coiled wire, which creates a magnetic field. By periodically reversing the current through this electromagnet, you can create force, which then becomes motion. There are many ways to build motors, but this is the fundamental basis of all of them.

In this chapter, we are going to talk about three common types of motors used in small projects and robots. They are:

- Small DC motors
- Servo motors
- Stepper motors

Small DC motors

A DC motor has two wires, power and ground. When you supply power (putting 5V on the power line, for example), the motor will start spinning. Reverse the power and ground wires, and the motor will spin in the opposite direction. You control the speed of a DC motor by using pulse width modulation (PWM), a technique that we saw in Chapter 1 of this minibook for controlling the brightness of an LED. If the power is cycled at 50 percent (half on/half off) then the motor will spin at one-half the speed. These DC motors are inexpensive and great for driving wheels. Sometimes you will put an "encoder" on the motor shaft so you read into a computer how far the shaft has turned, giving the computer some feedback that can be useful.

Use a DC motor anytime you want something to be spun at a RPM (revolutions per minute), such as a fan or a car wheel. (See Figure 4-1.)

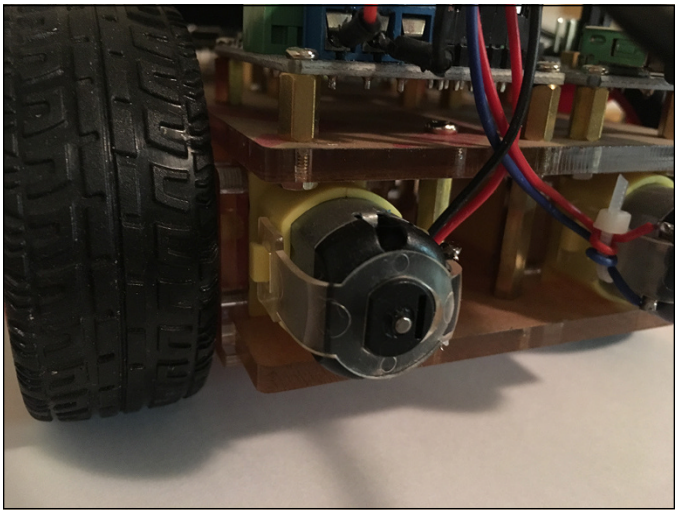

FIGURE 4-1: A DC motor on a small robot.

Servo motors

Servo motors are a generally a combination of three things: a DC motor, a simple control circuit, and a gearing set. Sometimes you will find a potentiometer (which is a variable resistor) that will give position feedback like the "encoder" in the DC motor discussed earlier. The servo motor is commanded to go to a specific position by using our friend PWM again. However, in this case, a specific pulse wave will hold the motor in a specific position and will resist a load or a force trying to move the motor. The maximum amount of force that the servo motor can exert against an external force or load is called the *torque rating* of that servo motor. Servos are continuously powered and generally only have about a 180-degree range of motion.

A classic place you find servo motors is in model RC airplanes and in some types of robot arms. When you want to move an object and hold it at a specific position, a servo motor is often the answer.

Use a servo motor for fast, high torque and for pretty accurate rotation to a specific position with a limited range of degrees. Good uses include rudder control on an RC boat or flaps on an RC airplane as well as robotic arms.

Stepper motors

A *stepper motor* is kind of like a servo motor that uses a different way to move the shaft. Whereas a servo motor uses a DC motor, a stepper motor uses multiple-toothed electromagnets surrounding a central-toothed shaft.

Stepper motors use an external controller (you will use the Raspberry Pi to do this) that will sequence the electromagnets surrounding that central shaft to make the central shaft turn in "steps," hence the name *stepper* motor. The design of a stepper motor provides a steady holding torque even when not powered up. Contrast that to the servo motor, which has to be powered up to supply torque. As long as the load is within the limits of the servo motor torque, then there are no positional errors.

Stepper motors are for slow, precise rotation. They are superior to servo motors in those types of applications. 3D printers are a great example of the use of stepper motors. Stepper motors also don't require a feedback system to determine where they are positioned as servo motors do. (See Figure 4-2.)

FIGURE 4-2: Sun-tracking solar panels using a stepper motor.

Controlling Motors with a Computer

Now we get to have some fun with Python in controlling motors. Next up, we go through all three types of motors and show you how to control them with Python. You will learn how to control motors both through GPIO (general purpose input-output) pins and through an I2C controller.

Python and DC Motors

There are lots of different ways of driving DC motors from a Raspberry Pi. There are dozens of robot controllers and motor controller boards that will work for this project. The one we will be using is an I2C controlled board (tying in with

our projects from the last chapter), which gives us control over two motors, their individual direction and their individual speed. Pretty cool.

Here is the parts list:

- **Pi2Grover Grove interface board:** Try store.switchdoc.com or Amazon.com.
- **Grove I2C motor drive:** Available at `www.seeedstudio.com` or Amazon.com (comes with a Grove cable).
- **Two small DC motors:** Try `https://www.adafruit.com/product/711` or Amazon.com.

For more on the Pi2Grover board, refer to Chapters 1–3 of this minibook. Let's spend some time on the Grove I2C motor drive because it has a rather unique way of doing things.

Grove I2C motor drive

The Grove I2C motor drive (see Figure 4-3) is capable of driving two motors at the same time, all controlled by our old friend, the I2C bus from the Raspberry Pi.

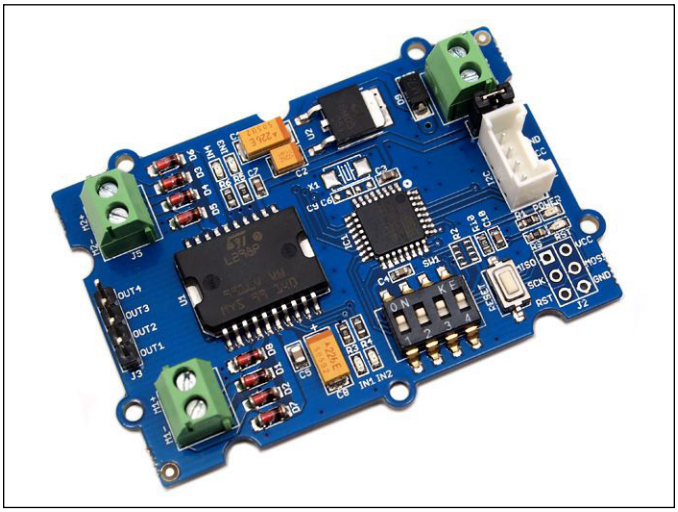

FIGURE 4-3: The Grove I2C motor drive.

It can drive up to 2A for each motor, but we are using motors a lot smaller than that. It can optionally handle 6V to 15V motors, but again, we are using small motors, so we will just use the Raspberry Pi power supply. If you are using bigger motors or want to use an external power supply, this board can do that for you.

The reason we are using this board is that it uses an I2C interface to control the motors, which gets us up and running quickly.

Figure 4-4 shows what is on the I2C motor drive board.

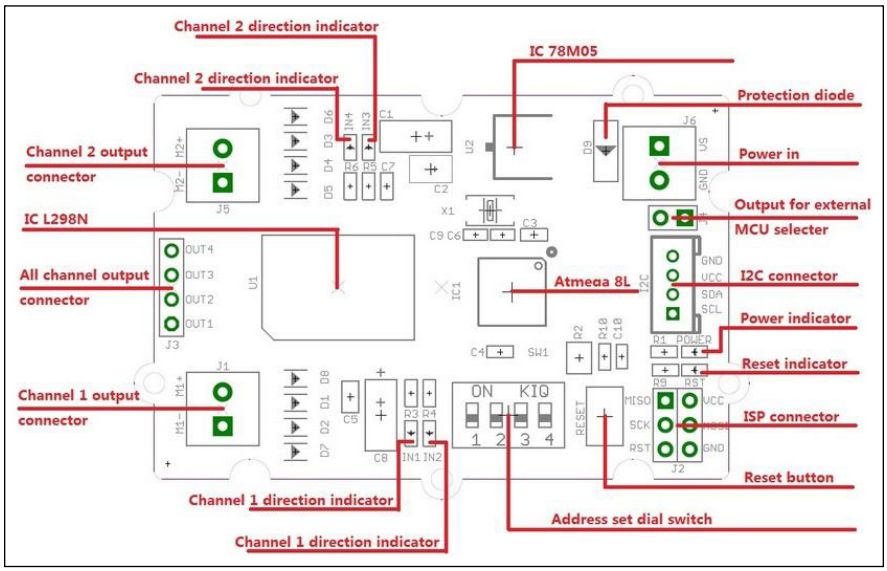

FIGURE 4-4: Annotated diagram of the I2C motor drive board.

There are a couple of interesting things about this diagram: First of all, there is another computer on this board! It is an Atmega 8L and is another small computer that actually emulates and I2C interface, processes commands coming in from your Raspberry Pi and then controls the motors. Yet another example of how even boards for little computers have little computers on them. Computers are everywhere! You can see the two motor connections on the left side of the board and also what the LEDs mean in the middle of the board.

You can also find the software for the onboard computer on the Grove I2C motor drive on the `www.seeedstudio.com` product page. You could change the programming if you want or at least understand how you can make a little computer look like an I2C device.

Let's wire it up and start our engines! Turn off the power on your Raspberry Pi before you hook this up:

1. **Loosen the set of two screw terminals on the end of the and insert the bare end of the wires on the motors (see Figure 4-5) into the holes and tighten the screws. (See Figure 4-6.)**

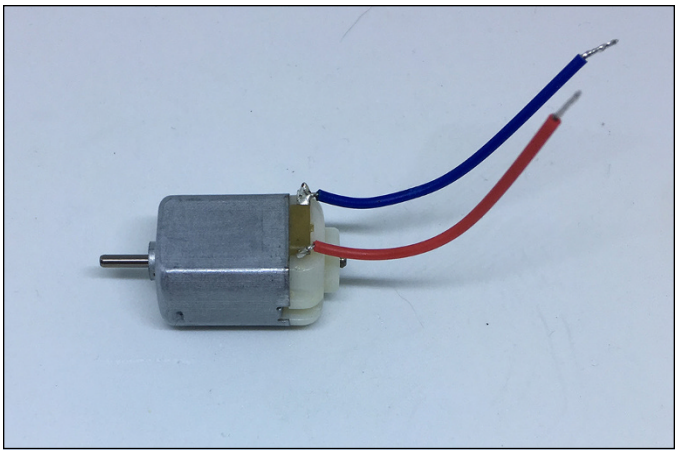

FIGURE 4-5: The Adafruit DC motor.

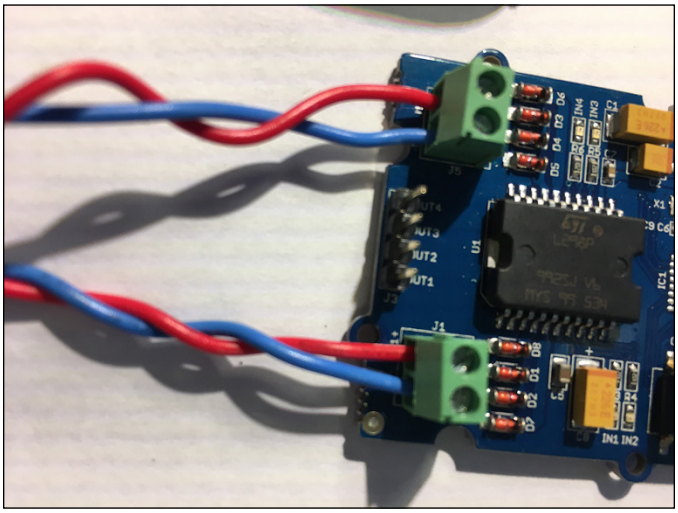

FIGURE 4-6: The wires in the I2C motor drive screw terminals.

Note that it really doesn't matter which color goes in which hole with a DC motor. It will just rotate in the opposite direction. Just match them both as in Figure 4-6. We added some length to the wires, but that is optional. Figure 4-7 shows the installed motor on your Grove I2C motor drive board.

2. Plug a Grove cable into the Grove connector on the Grove I2 motor drive board and then into an I2C Grove connector on the Pi2Grover board.

WARNING

Make sure you have the jumper in place on the board as indicated in Figure 4-4 (called "Output for external MCU selector" in the diagram). It comes that way in the package, but make sure it is still in place. The board won't operate as wired if you don't have that jumper installed.

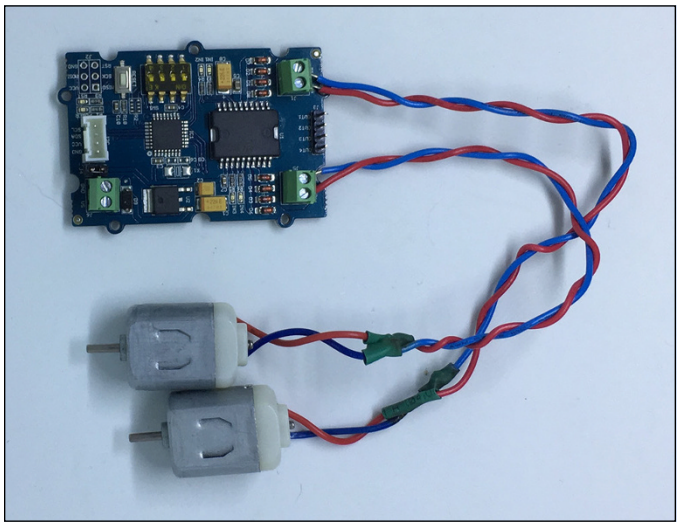

FIGURE 4-7: Motors installed on the motor drive.

TIP

If your board and motor don't respond after hookup, try hitting the Reset button on the board. Figure 4-8 shows the assembled motor and board.

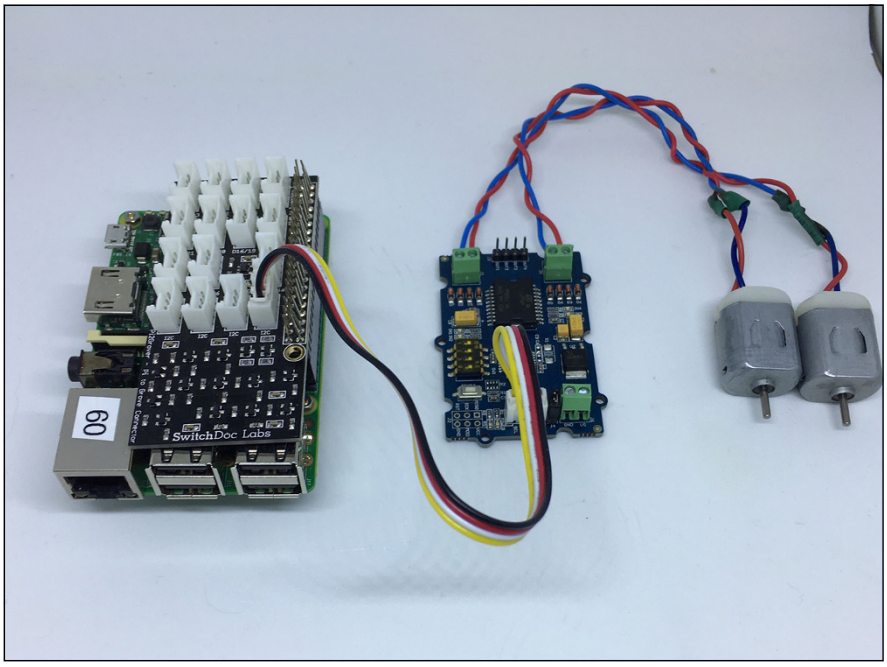

FIGURE 4-8: The DC motor setup.

Now, let's power the Raspberry Pi up and start writing some Python! After power-up, you should see all five LEDs shining on the Grove I2C motor drive board. If not, shut the Pi down again and check your wiring.

Python DC motor software

DC motors are often used for robot wheels, and so the words *forward* and *backward* should start to give you some ideas for later in the book when we are building a robot car.

The software we are going to use for the Python DC motor experiment should have a familiar feel because it is very similar to the I2C software we dealt with in Chapters 2 and 3.

The use of Python libraries are key to being productive in writing Python applications. We will be using the `SDL_Pi_HDC1080_Python3` available on Github.com.

To set up the software, follow these steps:

1. **Create a directory in your main directory by entering:**

```
cd
mkdir dcMotor
cd dcMotor
```

Now you are in the `dcMotor` directory.

2. **Before looking at the Python code for running your motors, install the library on the Raspberry Pi. You do this by "cloning" the library located up at github.com by using the following command in your terminal window:**

```
git clone https://github.com/switchdoclabs/SDL_Pi_GroveI2CMotorDrive.git
```

The `git clone` clones the git repository located at the address and copies it to your Raspberry Pi. If you enter `ls` in the terminal window, you will see the following output:

```
pi@RPi3-60:~/dcMotor $ ls
SDL_Pi_GroveI2CMotorDrive
pi@RPi3-60:~/I2CTemperature $
```

3. **Using nano (or your favorite text editor), open up a file called `dcmotorTest.py` and enter the following code:**

```
import sys
sys.path.append("./SDL_Pi_GroveI2CMotorDrive")


import SDL_Pi_GroveI2CMotorDrive
import time

#"0b1010" defines the output polarity
#"10" means the M+ is "positive" while the M- is "negative"

MOTOR_FORWARD =  0b1010
MOTOR_BACKWARD =  0b0101


try:

    m= SDL_Pi_GroveI2CMotorDrive.motor_drive()

    #FORWARD
    print("Forward")
      #defines the speed of motor 1 and motor 2;)
    m.MotorSpeedSetAB(100,100)
      m.MotorDirectionSet(MOTOR_FORWARD)
    time.sleep(2)

    #BACK
    print("Back")
    m.MotorSpeedSetAB(100,100)
    #0b0101  Rotating in the opposite direction
      m.MotorDirectionSet(MOTOR_BACKWARD)
    time.sleep(2)

    #STOP
    print("Stop")
    m.MotorSpeedSetAB(0,0)
    time.sleep(1)

    #Increase speed
    for i in range (100):
        print("Speed:",i)
        m.MotorSpeedSetAB(i,i)
        time.sleep(.02)
```

```
        print("Stop")
        m.MotorSpeedSetAB(0,0)

    except IOError:
        print("Unable to find the I2C motor drive")
        print("Hit Reset Button on I2C Motor Drive and Try Again")
```

This program runs both motors forward at full speed (100), then backwards at full speed, stops the motors, and then runs them backward at a slow speed ramping up to full speed, and then stops the motors entirely.

The key aspects of this software are calls to the `SDL_Pi_GroveI2CMotorDrive` library. The library supports the following functions:

- `MotorSpeedSetAB(` *`MotorSpeedA`*`,` *`MotorSpeedB`*`)`: Motor speed for the A motor (M1) and the B motor (M2). Range 0–100.
- `MotorDirectionSet(`*`Direction`*`)`: Forward or backward —constants set in the program. `MOTOR_FORWARD = 0b1010, MOTOR_BACKWARD = 0b0101`

One interesting piece of the `SDL_Pi_GroveI2CMotorDrive` library is that it uses one of the many I2C libraries available, `smbus`. In the library, you send the command to the I2C board as a `block write` consisting of the I2C address, a command byte, and then the arguments. Here is the call for setting the motor direction:

```
    #Set motor direction
    def MotorDirectionSet(self,Direction):
        bus.write_i2c_block_data(self.I2CMotorDriveAdd, self.DirectionSet,
[Direction,0])
        time.sleep(.02)
```

Time to run the DC motors now. Type this into your terminal window:

```
sudo python3 testMotor.py
```

You will be rewarded by seeing the LEDs change and seeing your motors go through a sequence that you have programmed in Python. You should be able to make your own sequences very easily from this example. Now on to a Servo motor.


TIP

All these motors take power from the Raspberry Pi when running, so disconnect the DC motors when you are ready to move to the next section. Shut down the Pi first!

Python and running a servo motor

Servo motors are a different beast than a DC motor. A servo motor has a DC motor inside, but it also has a controller circuit that allows us to position the DC motor to a specific position and then hold the motor there. A great use of a servo motor is on a robot arm where you want to arm to go to a specific position and wait there for further orders.

You control servo motors by using PWM (pulse width modulation). Although you can buy boards that will do PWM (and support bigger servo motors!) under control of your computer, for this small servo we will be using the built-in PWM capability of the GPIO (general purpose input-output) pins of the Raspberry Pi.

Here is the parts list:

- **Pi2Grover Grove interface board:** Look for it at store.switchdoc.com or amazon.com.
- **SG90 micro servo motor:** Try finding it on ebay.com or amazon.com. These are inexpensive so you may end up buying two or more for under $10.
- **A package of Grove male patch cables:** Grove-connector-to-male-pin cables: Available at store.switchdoc.com or amazon.com.

The Pi2Grover has been described before in Chapters 1, 2, and 3 of this minibook.

The SG90 micro servo motor (see Figure 4-9) is a small, inexpensive servo motor available from many sources. It has an operating voltage of 3.0V – 7.0V with about a current draw of about 40mA (40 milliamps – a milliamp is 1/1000th of an amp of current) at maximum so the 5V on the Raspberry Pi is good to operate this servo. It can turn about 90 degrees in each direction, for a total of 180 degrees. There are three control wires on most servo motors and the SG90 is no exception. The three wires are:

- Yellow – PWM control signal
- Red – Power (5V, in our case)
- Brown – Ground

The Grove-male-pin-to-Grove-connector patch cables (see Figure 4-10) are used to make a connection between the three pins on the servo to a Grove connector, and then you can plug it into your Raspberry Pi.

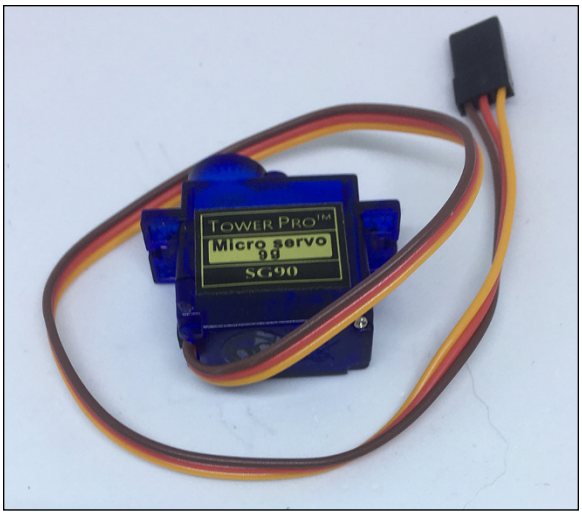

FIGURE 4-9: The SG90 micro servo with wires.

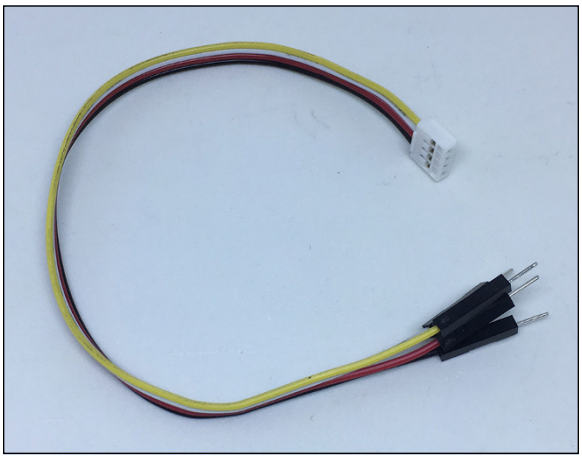

FIGURE 4-10: Grove male-pin-to Grove-connector patch cable.

TECHNICAL STUFF

HOW MUCH CURRENT CAN THE RASPBERRY PI SUPPLY TO THE 5V PINS?

Unfortunately, there isn't a simple answer to this question, because it depends on what you have connected to the Raspberry Pi 3 and what kind of a USB power supply you are using. 250mA is a pretty good number for a rule of thumb, but if you have a beefy 5V USB power supply (say 2.5A) you can go a lot higher, up to 1000mA or more. The Raspberry Pi 3 has a 2.5A fuse on the 5V power supply. It is a resettable fuse so if you pop it, just let it cool down and it will work again. Very good if you make a mistake.

Now let's connect the wires and make a servo motor rock!

1. **Shut down your Raspberry Pi and remove power.**
2. **Plug the Grove patch cable into your SB90 servo motor following this wire chart in Table 4-1. (See Figure 4-11.)**

TABLE 4-1 Servo Motor to Patch Cable Wiring

SG90 Servo	Grove Patch Cable	Function
Yellow Wire	Yellow Wire	PWM Signal
Red Wire	Red Wire	Power
Brown Wire	Black wire	Ground

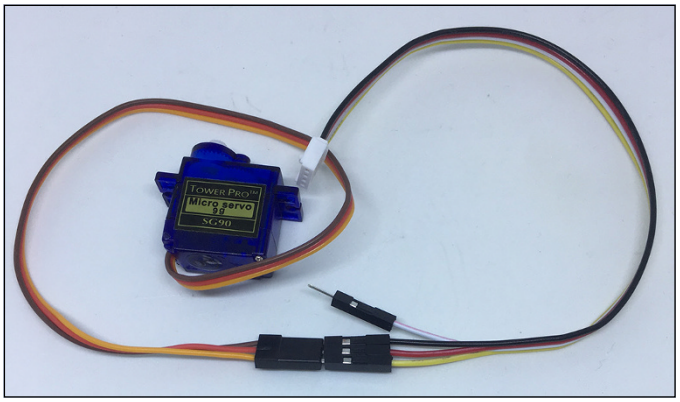

FIGURE 4-11: Servo motor correctly wired to patch cable.

WARNING

Check your wiring carefully. You can damage your Pi and motor if you reverse these wires.

3. **Plug the end of the Grove cable in the Pi2Grover Grove connector marked D4/D5.**
4. **Put a piece of electrical tape or blue tape over the white exposed pin on the Grove patch cable to keep it from shorting anything out. Also put one of the supplied rocker arms on the servo motor gear so we can see more easily its range of motion. (See Figure 4-12.)**

Now let's look at the Python software.

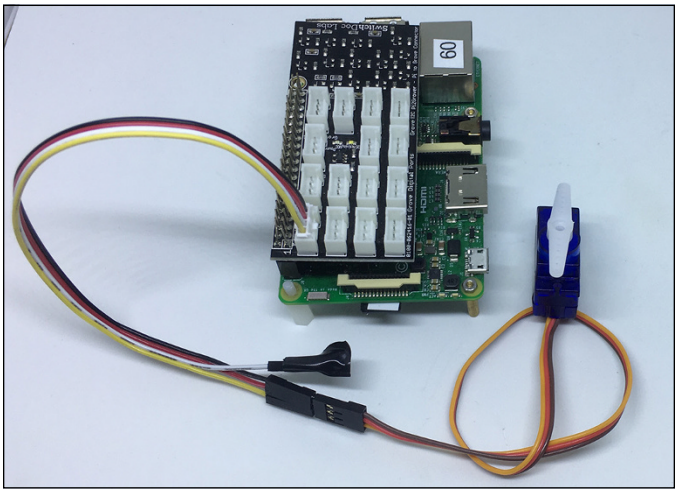

FIGURE 4-12: Fully connected Pi and servo motor.

Python servo software

We are not going to use a higher-level servo library (and there are many available for the Raspberry Pi in Python); instead, we are going to show you how to control a servo motor directly by using GPIO and the PWM function of the `RPi GPIO` built-in library. Okay, okay. We *are* using a library (`RPi.GPIO`), but we're not adding layers of API (application programming interface) calls like we normally would do. We're getting down and dirty with the GPIO pins. To do so, follow these steps:

1. **Create a directory in your main directory by entering:**

```
cd
mkdir Servo
cd Servo
```

2. **Using nano (or your favorite text editor), open up a file called `servoTest.py` and enter the following Python code:**

```
import RPi.GPIO as GPIO
import time


GPIO.setmode(GPIO.BCM)
ServoPin = 4

GPIO.setup(ServoPin, GPIO.OUT)

p = GPIO.PWM(ServoPin, 50)
```

```
p.start(7.5)

try:
    while True:
        p.ChangeDutyCycle(7.5)  # turn towards 90 degree
        print ("90 degrees")
        time.sleep(1) # sleep 1 second
        print ("0 degrees")
        p.ChangeDutyCycle(2.5)  # turn towards 0 degree
        time.sleep(1) # sleep 1 second
        print ("180 degrees")
        p.ChangeDutyCycle(12.5) # turn towards 180 degree
        time.sleep(1) # sleep 1 second


except KeyboardInterrupt:
    p.stop()
    GPIO.cleanup()
```

Breaking down the code

This is where we set the pin number to the D4/5 Grove connector on the Pi2Grover board:

```
import RPi.GPIO as GPIO
import time


GPIO.setmode(GPIO.BCM)
ServoPin = 4
```

Next, we set the GPIO pin to "Output":

```
GPIO.setup(ServoPin, GPIO.OUT)
```

This command sets an object `p` to the `ServoPin` (4) and at a frequency of 50Hz. 50Hz is a good number for this type of servo motor:

```
p = GPIO.PWM(ServoPin, 50)
```

The following line starts the servo motor at 7.5 percent duty cycle. Remember how PWM works on a servo? It goes from one end of the servo turn to the by going from according to the duty cycle. The 7.5 and the number 2.5 and 12.5 are determined by

the servo type. We just looked at the numbers and empirically determined these numbers for the SG90 servo. They are similar for most small servos:

```
p.start(7.5)
```

Now we move the servo through its entire range:

```
try:
    while True:
        p.ChangeDutyCycle(7.5)  # turn towards 90 degree
        print ("90 degrees")
        time.sleep(1) # sleep 1 second
        print ("0 degrees")
        p.ChangeDutyCycle(2.5)  # turn towards 0 degree
        time.sleep(1) # sleep 1 second
        print ("180 degrees")
        p.ChangeDutyCycle(12.5) # turn towards 180 degree
        time.sleep(1) # sleep 1 second
```


Notice how we put the entire body of this program in try: except: clause? We did this so when your control-c out of the program, the program will shut down the servo motor and do some cleanup on the GPIO (release all pins back to the operating system). This is generally a good thing to do when you are dealing with GPIO pins directly.

```
except KeyboardInterrupt:
    p.stop()
    GPIO.cleanup()
```

Now it is time to run the program. Type the following into a terminal window:

```
sudo python3 servoTest.py
```

You should be rewarded by your screen printing out the following liens and your servo happily obeying your programmed orders. Change things out and try different angles and sequences. You can't hurt the servo by trying things.

```
90 degrees
0 degrees
180 degrees
90 degrees
0 degrees
180 degrees
90 degrees
0 degrees
```

Now we have a servo motor working on our Raspberry Pi. Remember at the beginning when we told you a servo motor can move an robot arm, a flap on an RC airplane or a rudder on an RC boat? Watching the servo motor go thorough its programmed sequence should spark your thoughts of what to do with a servo motor. You can see why you use a DC motor for wheels and a servo to do things in a non-rotating manner.

Experiment and build your own magical projects! Now we will step right down the line to our last major motor, a stepper motor.

Python and making a stepper motor step

Stepper motors are yet a different beast than DC motors or servo motors. They are used for accurate positioning of items with a digital interface. You can accurately position things with a servo motor too, but it requires more electronics and definitely needs what is called positional feedback.

A stepper motor gets around this by accurately moving from one "step" to another under command of software. The motor is constructed to use two motor coils to advance the motor one step by sending a specific sequence to the motors. You will be implementing this "stepping" sequence in the Python software controlling the stepper motor.

A stepper motor typically has two coils used to move the motor from one step to another (see Figure 4-13).

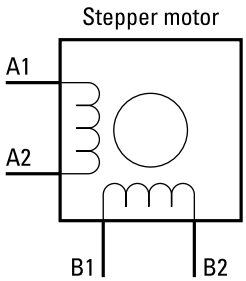

FIGURE 4-13: A diagram of a stepper motor.

Table 4-2 shows the sequence for stepping the stepper motor forward one step, and Table 4-3 shows the stepping sequence for moving the motor backward one step. This pattern of steps will be very obvious in our Python software.

TABLE 4-2

Forward Stepping the Stepper

Coil_A_1 (Pin 12)	Coil_A_2 (Pin 20)	Coil_B_1 (Pin 13)	Coil_B_2 (Pin 21)
1	0	1	0
0	1	1	0
0	1	0	1
1	0	0	1

TABLE 4-3

Backward Stepping the Stepper

Coil_A_1 (Pin 12)	Coil_A_2 (Pin 20)	Coil_B_1 (Pin 13)	Coil_B_2 (Pin 21)
1	0	0	1
0	1	0	1
0	1	1	0
1	0	1	0

In Figure 4-14, this digital sequence is graphically portrayed from a logic analyzer connected to the Raspberry Pi GPIO pins used to drive the stepper motor.

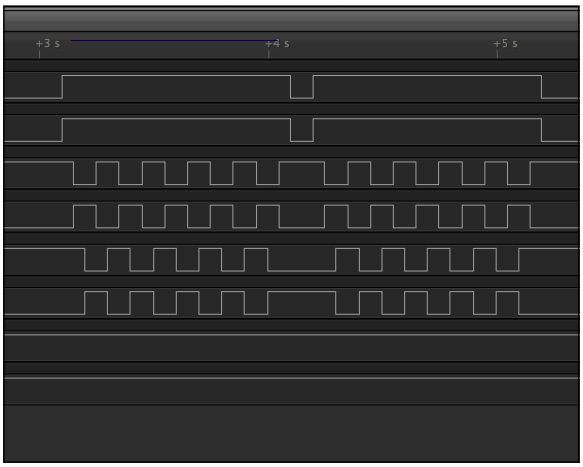

FIGURE 4-14: Logic analyzer showing the motor stepping sequence.

Well, now you know all you need to know about stepper motors to build your first project.

FEEDBACK: WHAT A USEFUL THING!

Feedback occurs when you route the output of a system back into the inputs of a system as a loop. Sounds complicated? It can be, but the basics are simple. Say you publish and article and ask for comments. People supply comments (hopefully nice ones) and you change the article based on some of those comments. That is feedback!

You use feedback in electrical circuits to achieve a better and more accurate positioning of a servo motor, for example. By reading an encoder on the shaft of the servo (an encoder gives you an electrical signal proportional to the position of the shaft), your software can tweak the PWM duty cycle to get to a more accurate position depending on the feedback.

There are two types of feedback: negative and positive feedback. In our example above, we think of positive feedback as being good comments or at least constructive criticism on our article, whereas negative feedback is just comments telling us how bad the article is.

However, in electronics, you tend to like negative feedback and not like positive feedback, at least in general. Negative feedback is what we use to get closer to the position on the shaft we want to be at, for example. Positive feedback tends to make differences get larger. Ever hear a speaker wail when you put a microphone too close the speaker? That's positive feedback.

Here is the parts list:

- **Pi2Grover Grove interface board:** Try store.switchdoc.com or amazon.com.
- **28BYJ-48 ULN2003 5 V stepper motor:** Look for these at eBay.com or `https://amzn.to/2BuNDV1`. These are inexpensive, so you may end up buying five for $12. Make sure you get the ones with the driver boards (such as the ones at the preceding Amazon.com link).
- **A package of Grove female patch cables, Grove-connector-to-female-pins:** Available at store.switchdoc.com or amazon.com.

The Pi2Grover has been described before in Chapters 1, 2, and 3 of this minibook.

The 28BYJ-48 stepper motor is a 5V stepper motor that has 5.625 x 1/64 degrees per step (approximately 0.822 degrees per step) and comes with a driver board

with a ULN2003 motor drive chip, and the best of all, four LEDs that show what you are doing with the motor from your software. (See Figure 4-15.)

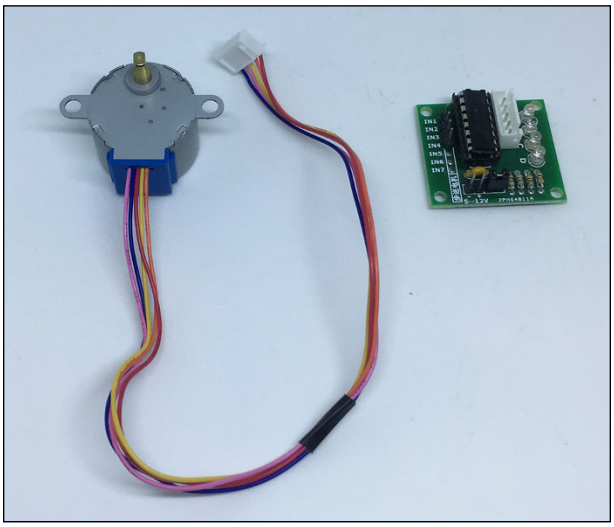

FIGURE 4-15: The 28BYJ-48 stepper motor and UNL2003 driver board.

The female-Grove patch cables are used to connect the stepper motor drive board to the Raspberry Pi. (See Figure 4-16.)

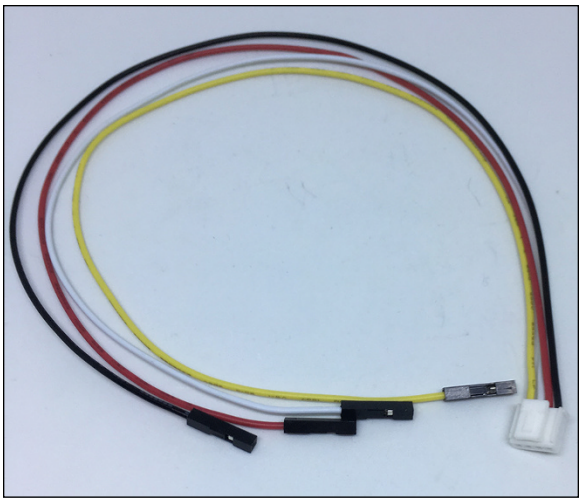

FIGURE 4-16: A Grove-connector-to-female-pin-header patch cable.

Time to build your stepper motor project! Just follow these steps:

1. **Shut down your Raspberry Pi and remove power.**
2. **Take a Grove female patch cord and connect it to the UNL2003 driver board, as in the wire chart in Table 4-4.**

   Note we put a wire tie on the cable to keep things neat and tidy.

TABLE 4-4 First Grove Female Patch Cord to UNL2003 Driver Board

Grove Patch Cable	UNL2003 Driver Board	Function
Yellow Wire	IN1	Coil A_1
White Wire	IN2	Coil B_1
Red Wire	+	Power
Black wire	–	Ground

Look very carefully at your red and black wire on the Grove patch cord to make sure it is plugged in, as shown in Figure 4-17.

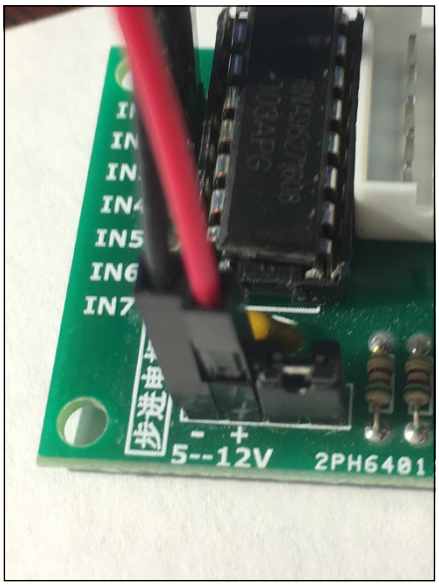

FIGURE 4-17: Close-up of power connections on the UNL2003 driver board.

3. **Take a second Grove female patch cord and connect it as in the wire chart in Table 4-5.**

TABLE 4-5

Second Grove Female Patch Cord to UNL2003 Driver Board

Grove Patch Cable	UNL2003 Driver Board	Function
Yellow Wire	IN3	Coil A_2
White Wire	IN4	Coil B_3
Red Wire	No Connect	
Black wire	No Connect	

Use a wire tie or a piece of tape to keep the unused red and black wires up and out of the way, as shown in Figure 4-18.

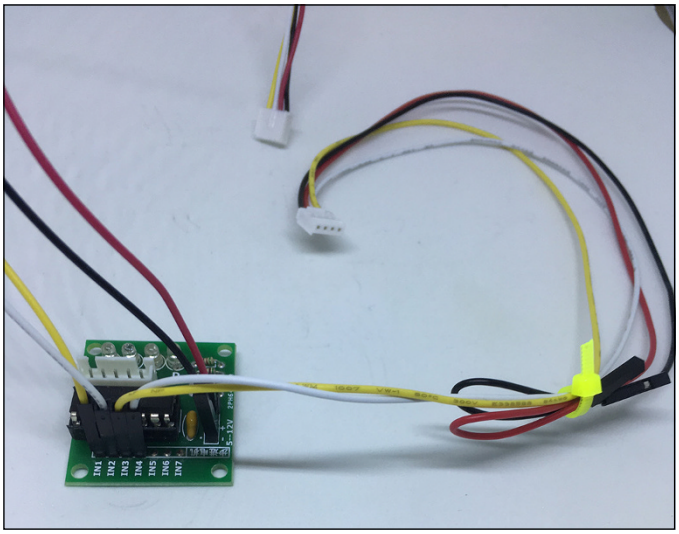

FIGURE 4-18: Second Grove patch cable attached.

Now, check all your connections again and make sure it looks like those in Figure 4-19.

4. **Plug your 28BYJ-4 stepper motor cable into the UNL2003 driver board connector. It is keyed and only goes in one way. (See Figure 4-20.)**

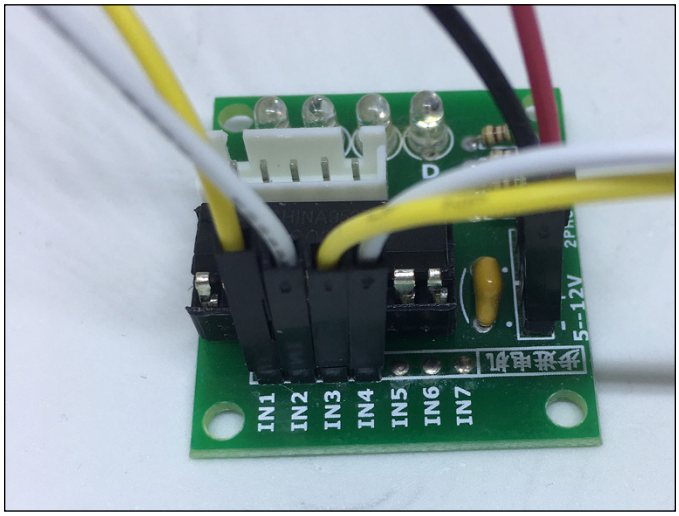

FIGURE 4-19: All patch wires installed on UNL2003 driver board.

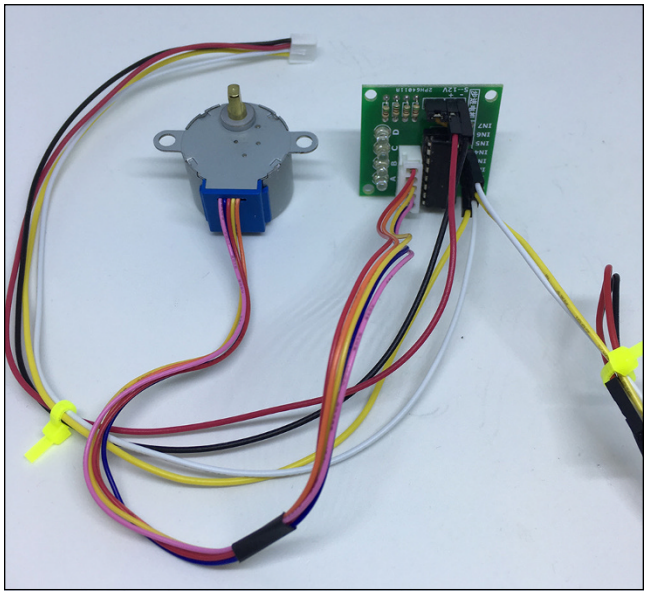

FIGURE 4-20: Stepper motor and driver board connected.

5. **Plug the first Grove patch cable (the one that has all four wires connected to the UNL2003 driver board) into the Pi2Grover Grove connector marked D12/13 and the second Grove patch cable (the one that only has the yellow and white wires connected) into the Pi2Grove Grove connector marked D20/21.**

   Wiring is now complete. The full wired system is shown in Figure 4-21.

6. **Put a cardboard arrow on your stepper motor shaft so you can really see it move. (See Figure 4-22.)**

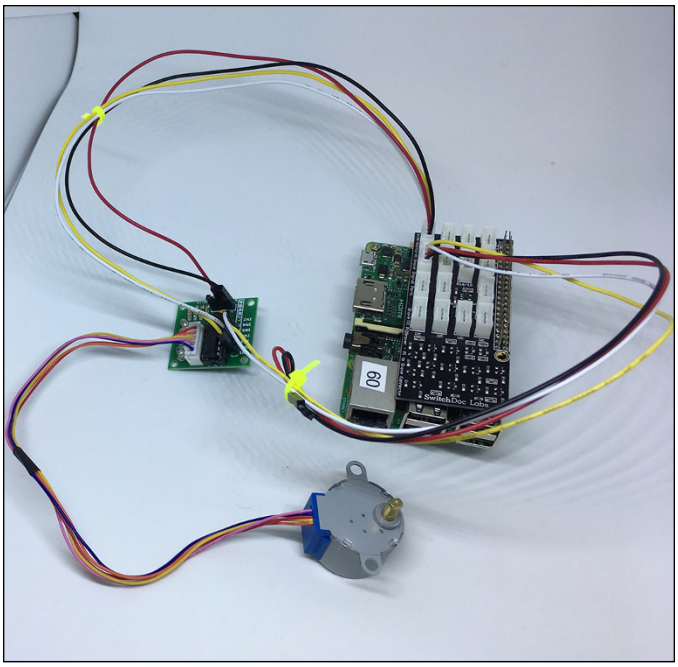

FIGURE 4-21: Fully wired Raspberry Pi and stepper motor project.

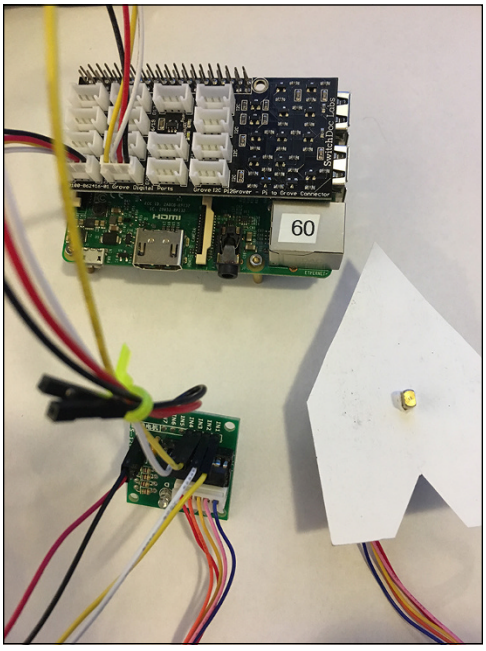

FIGURE 4-22: Stepper motor, ready to step.

Python stepper software

Similar to what we did with the servo motor, we are not going to use a higher level stepper library (and there are many available for the Raspberry Pi in Python — not all work with all stepper motors!), and instead we are going to show you how to control a stepper motor directly by using GPIO pins. Just follow these steps:

1. **Create a directory in your main directory by entering:**

```
cd
mkdir Stepper
cd Servo
```

2. **Using nano (or your favorite text editor), open up a file called `stepperTest.py` and enter the following Python code:**

```
import sys

import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
coil_A_1_pin = 12
coil_B_1_pin = 13
coil_A_2_pin = 20
coil_B_2_pin = 21

GPIO.setup(coil_A_1_pin, GPIO.OUT)
GPIO.setup(coil_A_2_pin, GPIO.OUT)
GPIO.setup(coil_B_1_pin, GPIO.OUT)
GPIO.setup(coil_B_2_pin, GPIO.OUT)


def forward(delay, steps):
  for i in range(0, steps):
    setStep(1, 0, 1, 0)
    time.sleep(delay)
    setStep(0, 1, 1, 0)
    time.sleep(delay)
    setStep(0, 1, 0, 1)
    time.sleep(delay)
    setStep(1, 0, 0, 1)
    time.sleep(delay)
```

```
def backwards(delay, steps):
   for i in range(0, steps):
    setStep(1, 0, 0, 1)
    time.sleep(delay)
    setStep(0, 1, 0, 1)
    time.sleep(delay)
    setStep(0, 1, 1, 0)
    time.sleep(delay)
    setStep(1, 0, 1, 0)
    time.sleep(delay)


def setStep(w1, w2, w3, w4):
  GPIO.output(coil_A_1_pin, w1)
  GPIO.output(coil_A_2_pin, w2)
  GPIO.output(coil_B_1_pin, w3)
  GPIO.output(coil_B_2_pin, w4)

while True:

  try:

    # Delay between steps (milliseconds)
    delay = 10
    # How many Steps forward
    steps = 50
    forward(int(delay) / 1000.0, int(steps))
    # How many Steps backwards
    steps = 50
    backwards(int(delay) / 1000.0, int(steps))

  except KeyboardInterrupt:
    # shut off all coils
    setStep(0,0,0,0)
    sys.exit()
```


Breaking down the code

The `stepperTest.py` code is pretty simple. We turn GPIO outputs to 1 and 0, according to the stepper motor sequences shown in Tables 4-2 and 4-3. You can see the exact sequences in the code in the forward and backwards functions.

Now let's run the code and start stepping away. We make sure we shut off all the outputs when you hit Ctrl-C to stop the program.

Time to run! Power up your Pi and open a terminal window. Note that all four of the LEDs turn on when you power up, but these will be turned off when you run your Python program. (See Figure 4-23.)

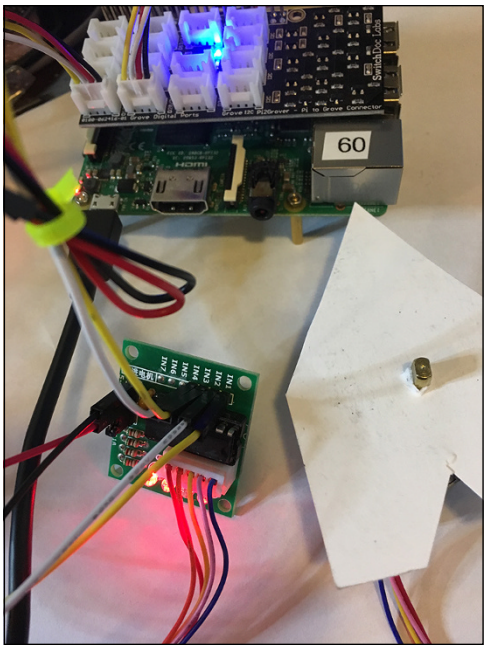

FIGURE 4-23: The Raspberry Pi running the stepper motor.

You will see the stepper motor turn 50 steps to the left and then 50 steps to the right. Try changing those variables in the program above to move your stepper motor to other positions. Do you see how these motors can be used in 3D printers and robots to accurately position printing heads, bed height, and robot arms?

7

Building Robots with Python

Contents at a Glance

IN THIS CHAPTER

- Understanding what a robot is
- Types of robots
- Knowing the parts of a robot

Chapter 1
Introduction to Robotics

Robots. That's a name that has been bandied about for a hundred years. It comes from a Czech word, *robota*, which means "involuntary labor." It was first used in the 1920 play "R.U.R. – Rossum's Universal Robots" by Karel Capek, but it was really Capek's brother Josef who coined the word. Did you know there was a robot in Frank Baum's land of Oz in 1907? He didn't call it a robot, but it's definitely a robot.

Robots are everywhere in today's modern world. Your house is filled with them. How is that possible? To understand what we are talking about, you need to understand the definition of *robot* more in computer-science terms than in Hollywood's.

A Robot Is Not Always like a Human

Two things to know about robots:

- Robots have only two features, a computer and an actuator.
- Robots are dumb; they are not people.

Robots have computers of some kind; think of it as the machine's brain. These brains can vary from IBM's Watson to a small eight-bit processor with a few thousand bytes of RAM. (You will see this with Baxter the Coffee-Making

Robot — yes, Baxter has a computer inside, about 16 of them, to be clear. But more on that in a moment.) The computers don't even have to be in the robot itself. You can have a network connection to the computer controlling the robot.

So, what is an actuator? It is something that physically affects the world outside. Under this definition, an sophisticated IOT (Internet of Things) device connected to a sensor and a database is not really a robot, whereas a computer controlling a toaster (that pops up the toasted bread) is a robot. As with any definition, you can argue corner cases all day. But this is a good working definition that shows just how varied the "body type" of a robot can be.

Not Every Robot Has Arms or Wheels

The classic conception of a robot tends to be something that looks at least vaguely human. The amazing robots that help us assemble cars in factories, for example, have giant arms that pick up car doors, weld metal, place windshields, and do many other assembly line tasks. Smaller arms are all over manufacturing lines helping to produce small goods as well as large ones.

One of the relatively new categories of robots is called *Cobots* (*cooperative robots*). These are robots that are designed to work closely alongside humans in manufacturing lines. Robots in car lines will hurt you if you get in their way. Cobots will stop if they encounter you when moving. These robots can work right next to people, doing tasks to make the people more efficient. Baxter the Coffee-Making Robot is an example of a Cobot.

Robots don't always have arms (remember our toaster?). Robots can look like microwaves. They can look like cars (yes, self-driving cars, but also like your current car). Modern-day cars are filled with computers doing robotic things. For example, there is a computer that measures you pushing down the gas pedal and then adjusts the fuel-and-air mixture to get you just the increase that you are looking for. It's a drive-by-wire system, just like airplanes. No cables physically connect your pedal to a mechanical gas pump or valve. In the new 2019 BMW X3s, there is a button you can push to change the way your car behaves from "Comfort" mode to "Sport" mode. It changes the way the wheel feels (and how much feedback you get from the road), the way your gas pedal responds, and even the way the suspension reacts to road conditions. (See Figure 1-1.) In a modern car, there are over 20 computers doing all sorts of things all over the vehicle and talking to each other constantly.

FIGURE 1-1: Inputs for the BWM Robot Driving System.

People think of robots as being in assembly lines or those fabulous robotic dogs made by Boston Dynamic. We'll choose a few other types of examples of robots. All three of the following examples are robots under our definition.

The Wilkinson Bread-Making Robot

Wilkinson Baking is located in Walla Walla, Washington (`www.wilkinsonbaking.com`). They have invented a robotic bread-making system that may actually bring bread-baking back to the local stores and away from giant bread plants. (See Figure 1-2.)

This type of disruption has happened before. Remember how you used to have to send film away to big plants to get it developed? Then suddenly, machines (yes, robots) came out that allowed mom-and-pop stores to get back into the film-development and picture-printing business. Yes, the proliferation of digital cameras took most of that business way, but you can see the point.

Wilkinson has developed quite a bread-making robot that should be practical for locally mass-producing bread. By the way, one of the claims they make is that their bread only requires one-sixth of the fossil fuel needed to get the product to the consumer.

FIGURE 1-2: A robot making bread.

Baxter the Coffee-Making Robot

Baxter is a general-purpose Cobot. Manufactured by Rethink Robotics, he is a fairly old Cobot, introduced in 2011, but he is good for helping students to learn robotics because of his really fabulous set of sensors and cameras (one in each arm!) and his two arms, each with a different attachment (a gripper on the left and a suction cup on the right). This allows some very sophisticated projects to be done with the robot. (See Figure 1-3.)

Baxter has around 16 different computers inside: A main computer (actually a Dell PC, believe it or not) strapped in his torso and individual computers controlling all the joints, cameras, and sensors into both arms. Interestingly enough, Baxter's brain for the coffee-making program (all in Python, by the way) is located across the room and is connected to Baxter by a network connection. It uses a distributed ROS (robot operating system) to be controlled and provide information and images to the user and the controlling computer. Take a look at this video: `https://youtu.be/zVL8760H768`

Three students were called upon to teach Baxter to make coffee in their senior robotics class, and after three months and thousands of lines of code, they succeeded. The team had to use computer vision techniques to recognize when the coffee was done, connect Baxter up to the Amazon AWS Cloud so they could use Alexa ("Alexa, tell Baxter the Robot to make coffee"), and use a variety of pick-and-place code to select a Keurig cup and to deliver a full cup of coffee to the customer's table.

Interestingly, now that he can do this, Baxter appears much smarter, but Rule #2 still applies ("Robots are dumb"). He can make coffee, but he can't make toaster strudel. Yet. Next semester.

FIGURE 1-3: Baxter making coffee.

The Griffin Bluetooth-enabled toaster

You really didn't think we were going to finish these examples of robots without using a toaster did you? The Griffin Bluetooth-enabled toaster was presented at CES (Consumer Electronic Show) as part of suite of connected kitchen appliances. It allows you to program (by app) your desired level of toasty crispness, and it can send your phone a notification when your toast is done. This is great, but on the other hand, it needs about three minutes to make toast. Also, it looks like it hasn't yet gone into production, which is a great disappointment to John personally, because he wants a connected toaster.

Here is a link to a YouTube video for the toaster: `https://www.youtube.com/watch?v=Z7h8-f-k8C8`.

Before we leave the world of robot toasters, one more toaster: Toasteroid, funded by a Kickstarter campaign takes robotic toast to a whole new level. (See Figure 1-4.) It can print messages on your toast! From a smartphone! Funded in 2016 (but sadly still not available), the company claims a retail price of $85, which tells you how inexpensive a lot of this robotic technology has become.

FIGURE 1-4: The Toasteroid Internet-connected toaster.

Enough examples. Let's now move into to what makes a robot and how we program them in Python!

Understanding the Main Parts of a Robot

You can break all the robots in generally four types of components:

- Computers
- Motors and actuators
- Communications
- Sensors

Computers

Computers are totally ubiquitous in robotics. You have a computer controlling the cameras, another computer controlling each joint in the arms, and another computer interpreting the images coming in from the cameras and reading the sensors monitoring the environment around the robot.

These computers are mostly small computers called *embedded systems*. They may only have one purpose (such as monitoring the amount of current being used for a motor) or they may be coordinating an entire arm, telling the other computers what to do.

The higher-level computers in the system are used for planning and receiving orders from other robots (or the assembly line itself) and people. And, yes, most of these higher-level computers are programmed in Python.

Motors and actuators

Motors and actuators are what differentiate robots from computers. These motors are all sorts of types and sizes (Refer to Book 5 for an introduction to motors). *Actuator* is a term with a definition that's a bit broader than *motor*'s. For example, an actuator could be memory wire, which is a type of metal that can be heated up to perform an action and later, when it cools down, it goes back to its original size. It is kind of like muscle fiber, and it has a lot of uses in the robotic industry. As we did in Book 5, you will be making motors move using Python in the following chapters.

Communications

Robots need to communicate. Not just verbally or even onscreen but in digital format that other computers and robots understand to coordinate actions and react to the environment. And robots use communications to offload complex tasks (such as, for example, computer vision interpretation) to other computers, sometimes even up in the cloud. And yes, many of these communication devices (BlueTooth, TCP/IP networks, WiFi) use computers inside the communication devices.

Sensors

We'll admit it. We love sensors. We are always on the lookout for the latest and greatest sensors. Electricity, temperature, humidity, electronic gyroscopes, pressure, touch sensors, people sensors, cameras, and image-processing sensors are becoming more inexpensive and pervasive every day. And many of these sensors and functions are programmed using Python for data processing and hardware drivers.

Unfortunately, in modern robotics and embedded systems, these components all end up somewhat mixed together. A motor in a robot will have a computer, communications, and sensors all together.

Programming Robots

Robots are programmed in many different types of languages. Some robots can be programmed by moving the arm to a specific set of locations and other robots are programmed in more traditional ways. We have found that programming by moving the arms will kind of get you in the ballpark and provide a structure for your programming to implement a particular function.

The most popular programming language in the world for programming robots at a high level (above the motor drives and such things) is Python. Hands down. Baxter is programmed via an API (application programming interface) provided for Python. Python is used to call many robotic functions and image processing as well as to provide the movement planning and coordination between robots. Although many robot manufacturers will have their own proprietary software, almost all of them will provide a method for working with Python.

IN THIS CHAPTER

- Building a robot
- Understanding the components
- Learning to program the components

Chapter 2
Building Your First Python Robot

In this chapter, we open up a robot to take a look inside, and we show you how to talk to all the parts with Python. We tell you how to program a robot after you build it. Why build the robot first? For two reasons: First, a kit-based robot is inexpensive compared to buying a prebuilt one. Second, by building a robot you get to know how a robot works and how you can use Python to control it.

We have chosen a robot that is based on our friend the Raspberry Pi. You can get robots that are based on many other computers, including the Arduino, but with those smaller computers, you can't do the kind of processing or artificial intelligence that you can on the Raspberry Pi.

After all, this is *Python All-In-One For Dummies.* Wouldn't you like to be able to use some of the tools you learned earlier in this book?

Introducing the Mars Rover PiCar-B

When we were deciding which robot to build in this book, we looked at and assembled four different small robot cars. All of them were similarly priced, and each of them had some drawbacks. However, after careful consideration,

we chose to use the Adeept PiCar-B. (See Figure 2-1.) We did this for several reasons:

- The assembly manual was clear with lots of pictures and diagrams.
- The supplied software was compatible with Python 3 (and the Stretch version of the Raspberry Pi operating system).
- The PiCar-B required no soldering.
- It had a reasonable price and good availability.

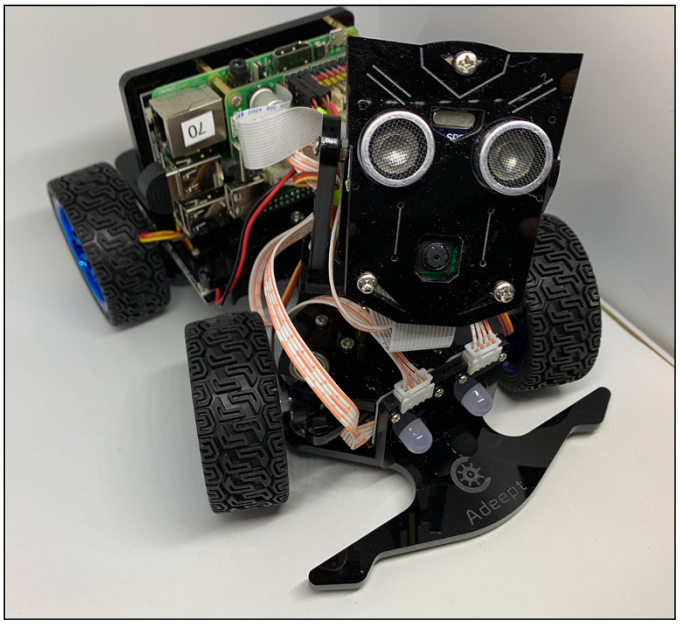

FIGURE 2-1: The assembled PiCar-B robot.

TIP

A radio control car can be considered a robot too. Why did we choose *this* car? It is because you can easily get inside the software and add more software to make the robot do what you want it to do. Yes, it does have a user interface (see the next chapter) on an additional PC. However, we are far more interested in putting our own Python software inside the robot than we are in playing with the joystick.

What you need for the build

There are three things you need in order to build the robot used in this chapter, in addition to some basic tools (although the kit comes with Allen wrenches and screwdrivers) and some plastic wire ties to get the wires bundled together after assembly:

- **Raspberry Pi 3B+:** Yes, you could get by with a smaller Raspberry Pi (like the Raspberry Pi Zero), but we recommend you get the fast one so you can do more sophisticated processing onboard the robot. The price you pay for a faster Pi (like the 3B+) is in power consumption and battery life. For our purposes here, it is a good tradeoff.

  Among other places, you can buy the Raspberry Pi 3B+ at

  - Amazon.com (make sure you buy the 3B+)
  - Newark.com
  - Adafruit.com
  - Shop.switchdoc.com

- **Adeept Raspberry Pi PiCar-B:** The Adeept Raspberry Pi PiCar-B is not quite as available as the Raspberry Pi. When you buy this, make sure you are buying the PiCar-B and *not* the PiCar-A. They add *Mars Rover* to the name of this product in their catalog, so look for the "Adeept Mars Rover PiCar-B."

  You can buy the PiCar-B at these places:

  - Amazon.com (`https://amzn.to/2B7mtop`)
  - eBay.com
  - Adeept.com
  - Shop.switchdoc.com

- **18650 LiPo batteries:** The PiCar-B requires two 18650 3.7V LiPo 5000mAh batteries. You can also power the robot by turning off the power switch (or removing the batteries) and supplying power for the Raspberry Pi from the micro USB plug, which then powers both the robot and the Raspberry Pi. The power for both the robot and the Raspberry Pi are connected together.

  The package we chose had two sets of batteries and an included wall charger. You can buy these kinds of batteries all over the place, including

  - Amazon.com (`https://amzn.to/2TgPsx1`)
  - Many, many other places.

Understanding the robot components

Now it is time to look at the components in the PiCar-B. We're not going to focus on the mechanical structure of the robots but rather on each of the active components. We'll also talk about the Python software used to communicate with each device used in the Python system test software later on in this chapter and in our own robotic software in Chapter 3.

We will be giving small Python code snippets to show you how each of the sensors and motors are controlled. For more of the code and description, see the "Running Tests on Your Rover in Python" section, later in this chapter.

Controller board

This motor controller is designed to interface the Raspberry Pi to the sensors and motors on the PiCar-B. (See Figure 2-2.) The main two chips on the board are a PCA9685, an I2C device used to control up to 16 servo motors (of which three are used, so there's lots of room for expansion) and an L289P, which is used to provide power to the main drive motor. The rest of the board is used to connect up the GPIO (general purpose input-output) pins from the Raspberry Pi to the various sensors and devices. It also has a 5.0V power supply that supplies the Raspberry Pi and motors from the LiPo batteries. (See Figure 2-3.)

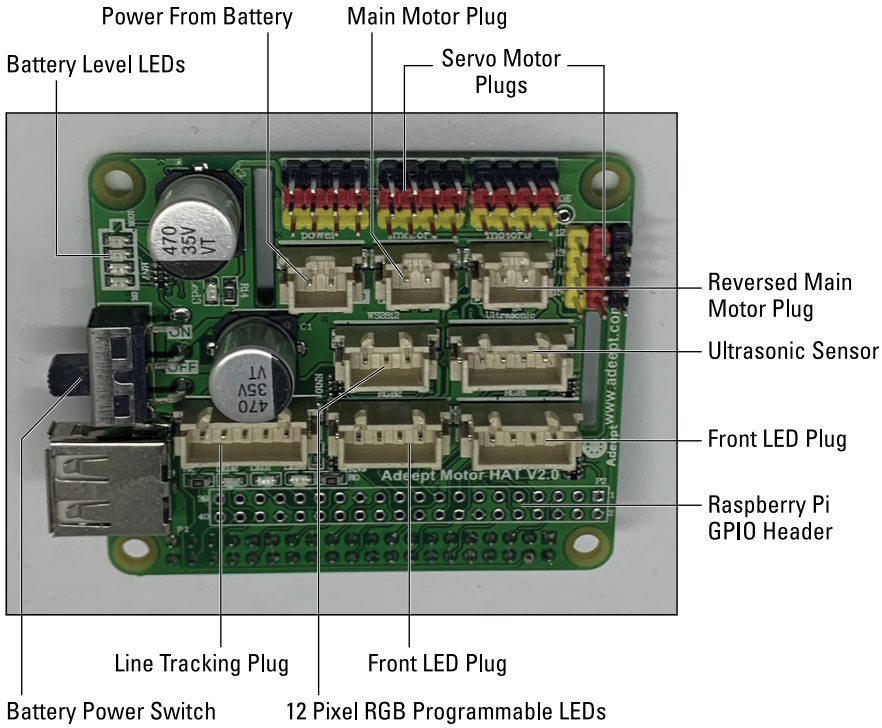

FIGURE 2-2: The PiCar-B motor controller board.

Servo motors

Servo motors are a generally a combination of three things: a DC motor, a simple control circuit and a gearing set. Sometimes you will find a potentiometer (which is a variable resistor) that will give positional feedback.

You control the position by using pulse width modulation (PWM), a technique that we saw in Chapter 1 for controlling the brightness of an LED.

The SG90 micro servo motor supplied with the robot is a small, inexpensive servo motor available from many sources. (See Figure 2-3.) It has an operating voltage of 3.0V–7.0V with a current draw of about 40mA (40 milliamps; a *milliamp* is 1/1000th of an amp of current) at maximum so the 5V on the Raspberry Pi is good to operate this servo. It can turn about 90 degrees in each direction for a total of 180 degrees. There are three control wires on most servo motors and the SG90 is no exception. The three wires are:

- **Yellow:** PWM control signal
- **Red:** Power (5V, in our case)
- **Brown:** Ground

FIGURE 2-3: The SG90 micro servo motor.

Servos are continuously powered and generally only have about a 180–270 degree range of motion.

All three of the servo motors are SC-90 9g micro servos.

These servos are controlled from the PCA9685 I2C chip, so it doesn't use any GPIO lines from the Raspberry Pi. It is on the I2C bus of the Raspberry Pi. (See Chapter 3 for more information about the I2C bus.)

To control a servo motor, we just have to set the value to the position we want the servo moved on the appropriate PWM line in the PCA9685:

```
print ("--------------------")
print ("Servo Test - Head Left")
print ("--------------------")
pwm.set_pwm(HEAD_TURN_SERVO, 0, calValues.look_left_max)
time.sleep(1.0)
```

The number we are passing into the servo motor for position (`calValues.look_left_max`) is empirically determined and is set by looking at the range of the servo motor as you command it to the left and right. See "Calibrating Your Servos," later in this chapter.

Drive motor

The main drive motor is a DC motor with two wires: power and ground. (See Figure 2-4.) When you supply power (by putting 5V on the power line, for example), the motor will start spinning. Reverse the power and ground wires, and the motor spins in the opposite direction. You control the speed of a DC motor by using pulse width modulation (PWM), a technique for controlling the brightness of an LED (refer to Chapter 1 of this minibook). If the power is cycled at 50 percent (half on/half off) then the motor will spin at one-half the speed. Sometimes you will put an "encoder" on the motor shaft, which allows you to read into a computer how far the shaft has turned, giving the computer some feedback that can be useful.

This motor uses six GPIO lines from the Raspberry Pi to control speed and direction.

The intricacies of controlling this motor are well hidden from the user. Here is the Python code to move the car forward:

```
motor.motor_left(MOTOR_START, forward,left_spd*spd_ad)
motor.motor_right(MOTOR_START,backward,right_spd*spd_ad)
```


TIP

Why are we turning both a left and right motor on when there is only one drive motor in the PiCar-B? The reason is that you can't be sure which way your motor is wired; it may be wired one direction or it may be reversed. (Ours was reversed.) You have two motor plugs on the controller board. If the forward command causes the robot to move backward, you just move the motor to the other motor

connection and everything works. Writing the preceding code (using both motor controllers) allows the software to work with either kind of motor.

FIGURE 2-4: The main drive motor.

RGB LED

The front of the robot has two single RGB LEDs, one on each side. (See Figure 2-5.) These big LEDs actually have three LEDs inside the housing. The LEDs are red, blue, and green and are individually controlled by GPIO lines (running software PWM code that allows us to mix the R, G and B LEDs).

The RGB LEDs each use three GPIO lines from the Raspberry Pi:

```
print ()
print ("--------------------")
print ("Left Front LED Test - Red ")
print ("--------------------")

led.side_on(led.left_R)
time.sleep(1.0)
led.side_off(led.left_R)
```

You turn the LEDs on individually and can (using other software) actually set the brightness of each LED.

FIGURE 2-5:
A single RGB LED.

Pixel RGB programmable LEDs

There are 12 programmable RGB LEDs on the robot, two sets of three on the bottom of the robot and two sets of three pointing to the rear. (See Figure 2-6.) These 12 LEDs are connected in serial like Christmas lights. And like some Christmas lights, if one goes out, then all the rest of the string goes too. That is because they are controlled by a single serial data stream that is sent through all the lights by the Raspberry Pi. This serial stream is very precisely timed, which requires some pretty special software on the Raspberry Pi to make it work.

The Pixel string uses a single GPIO pin coming from the Raspberry Pi:

```
print ()
print ("--------------------")
print ("12 RGB Pixel LED Test - On ")
print ("--------------------")

rainbowCycle(strip, wait_ms=20, iterations=3)
```

This command cycles a rainbow of colors around all 12 LEDs on the back and bottom of the robot. The driver for these RGB Pixels is very complicated, but we'll provide code to easily control the LEDs and we'll give you some examples of how to use them for other purposes. We really do love these LEDs and use them in many projects.

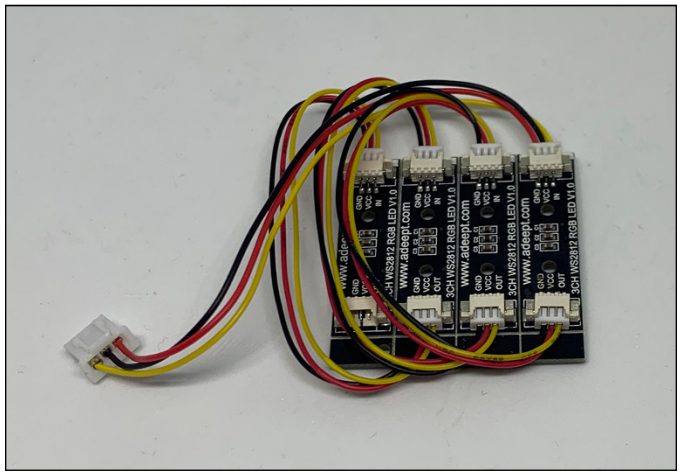

FIGURE 2-6: The 12 programmable RGB LEDs.

TECHNICAL STUFF

PIXEL RGB STRINGS ON THE RASPBERRY PI

The Raspberry Pi has a complex, multifaceted operating system based on Linux. It is a multitasking preemptive operating system, which means virtually any task (and all user tasks) can be interrupted (meaning stopped) and thus our serial stream to the Pixel LEDs stopped and corrupted to some degree.

The library we are using solves the real-time control problem by using the PWM and DMA hardware on the Raspberry Pi's processor. The PWM (pulse-width modulation) module can generate a signal with a specific duty cycle; for example, to drive a servo or dim an LED. The DMA (direct memory access) module can transfer bytes of memory between parts of the processor without using the CPU. By using DMA to send a specific sequence of bytes to the PWM module, the Pixel data signal can be generated without being interrupted by the Raspberry Pi's operating system.

Because the Arduino type of processors don't really have an operating system, it is pretty easy to generate these signals on an Arduino compared to an Raspberry Pi. Processors like the ESP8266 and the ESP32 do have tasks running in the background (like WiFi) and so require special drivers to compensate for that to avoid data corruption and flickering. Note that although this works well on the Raspberry Pi 3B+, the LEDs do not always work well with the older and smaller Raspberry Pis (A+, 3B, Pi Zero, or Pi Zero W, for example).

Pi camera

The camera used in the PiCar-B is the classic Raspberry Pi camera, version 2.1. (See Figure 2-7.) It has a Sony 8 megapixel sensor and can support pictures up to 3280 x 2464 at 15 frames per second.

It has good color response and is well supported in Python and with OpenCV (OpenCV is the Open Source Computer Vision package we use later in Chapter 4). The Pi camera talks to the Raspberry Pi via a parallel data ribbon cable directly into the Raspberry Pi board.

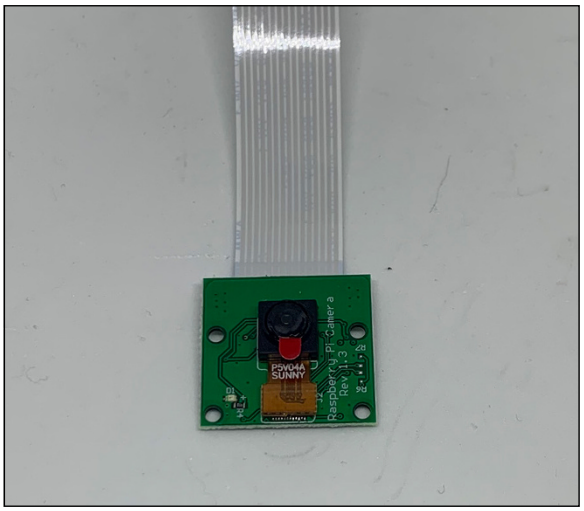

FIGURE 2-7: Raspberry Pi camera and cable.

The following code opens up a small window on the GUI of the Raspberry Pi, waits 20 seconds, moves the window and resizes it, waits 2 seconds, moves it again and then closes the window:

```
            print ()
            print ("--------------------")
            print ("Open Video Window")
            print ("--------------------")



            camera.resolution = (1024, 768)
     camera.start_preview(fullscreen=False,window=
(100,100,256,192))
            time.sleep(20)
            camera.preview.window=(200,200,256,192)
            time.sleep(2)
```

```
camera.preview.window=(0,0,512,384)
time.sleep(2)
camera.close()
```

TIP

If you are using VNC to see the GUI of the Raspberry Pi (as we are doing), you need to open up the options of the VNC server in the upper-right corner, go into Options, then down into Troubleshooting and click the Enable Direct Capture Mode checkbox. See the "Assembling the Robot" section, later in this chapter.

Ultrasonic sensor

The ultrasonic detector used in the PiCar-B is a non-contact distance measurement module that works at 40KHz. (See Figure 2-8.)

When provided a pulse trigger signal with more than 10uS through signal pin, the unit issues 8 cycles of 40kHz cycle level and detects the echo. The pulse width of the echo signal is proportional to the measured distance. Simple, yet effective.

FIGURE 2-8: An ultrasonic distance sensor.

The formula used is: Distance = Echo signal high time * Sound speed (340M/S)/2.

In the following code, the call to `ultra.checkdisk()` calls the software that sets the transmit GPIO bit and then waits for the returning echo, marking the time received. It then calculates the time of transit and reports the distance in meters, which we convert to centimeters.

```
print ()
print ("--------------------")
print ("Ultrasonic Distance Test")
print ("--------------------")
```

```
            average_dist = 0.0
            for i in range(0,10):
                distance = ultra.checkdist()
                average_dist = average_distance + distance
                print ("Distance = {:6.3f}cm ".format( distance*100))
                time.sleep(1.0)

            average_distance = average_distance / 10

        print ("average_dist={:6.3f}cm".format(average_dist*100))
```

Assembling the Robot

The PiCar-B comes with an assembly manual complete with blow out pictures of how things go together. (See Figure 2-9.)

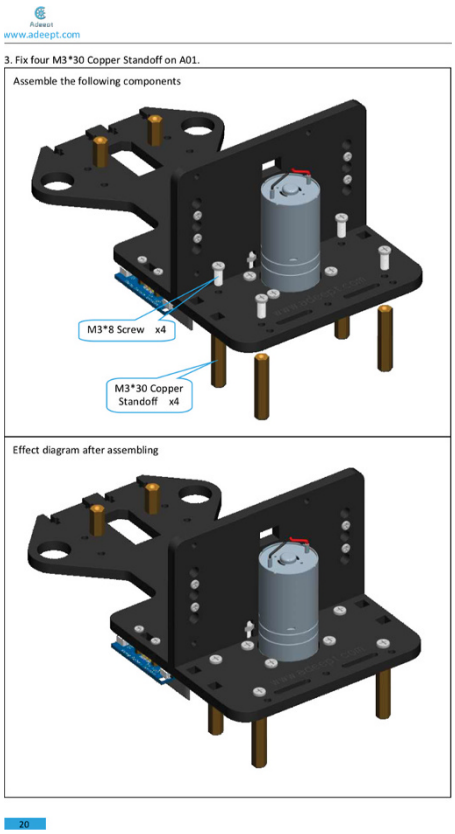

FIGURE 2-9: An example of the assembly manual diagrams.

It took us about four hours to put the robot together and get to the point to begin testing.

Note: Do not go beyond Chapter 2 in the PiCar-B assembly manual. Chapter 3 starts installing all the Adeept software on the Raspberry Pi and you will want to test the robot using the software in this book to better understand the parts of the robot before continuing.

So, go build your robot and then meet us back here to start testing your new robot. Calibrating your servos is the first step! Then we run a full system test. Figure 2-10 shows the assembled robot.

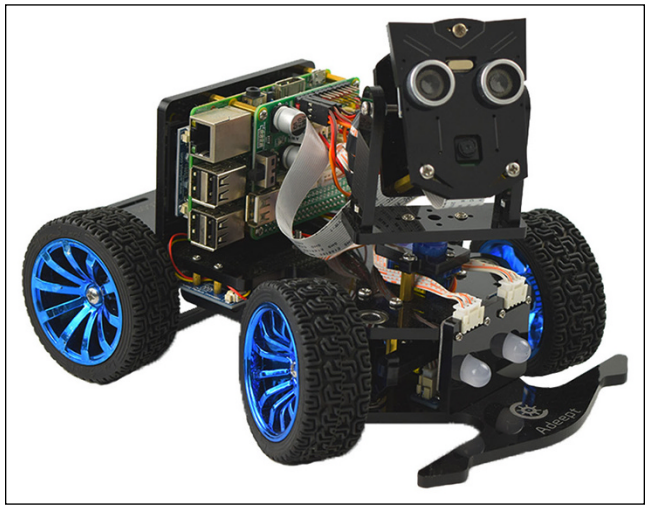

FIGURE 2-10: The assembled PiCar-B showing wiring.

TIP

Here are a few helpful tips for building the robot:

- There is a power switch on the left side of the motor drive board (viewed from the front of the robot). It is not mentioned in the manual. It shuts off the power from the batteries.
- Make sure you use the longer of the supplied ribbon cables for the Pi camera. The short one will not quite reach. Change it before you install the camera into the robot.
- Route the wires as shown in Figure 2-10 so the motors won't bind. Use some plastic wire ties to hold things in place, allowing for room for the servos and housing to turn. There is a wire wrap included that will fit some of the wires.

WARNING

PROPERLY TURNING OFF YOUR RASPBERRY PI

Unlike most other computers, the Raspberry Pi does not have an on/off switch. However, like many other computers, just pulling the plug on a Raspberry Pi can have dire consequences, in this case, corrupting the SDCard that the Raspberry Pi uses for program and data storage. Before you shut down the Raspberry Pi, enter the following in a terminal window: **`sudo halt`**.

This safely shuts down the Raspberry Pi. When you run this command, after a bit, you'll see the "ACT" light (the green one) blink 10 times (at 0.5 second intervals). When it stops blinking, the green light will turn off. At this point, it is safe to remove the power or pull the plug.

The red power LED will remain on as long as there is power applied to the Raspberry Pi.

- Pay close attention to the orientation of the plastic parts and servos during assembly. Almost all of them have an asymmetrical top and bottom and need to be oriented correctly to be assembled.
- Don't drop tiny screws in the carpet. They're hard to find for sure!

Calibrating your servos

Now that you have assembled the robot, it is time to start testing things. The first thing to do is to calibrate your servo motors. Why do they need calibration? Because although the instructions have you leave the servos centered during assembly, they will not necessarily be completely in the right place.

The `calibrateServo.py` program runs each of the three servos from one end to the other. By watching the motors as they turn, you can write down the max, min, and center of each servo from the display on the terminal window. Then you place these values in the `calValues.py` program for the rest of the programs to access. The values in the `calValues.py` are right for our robot and will probably be pretty close for yours, but you should run the program to be sure.

The `calibrateServo` code is as follows:

```
#!/usr/bin/python3

# calibrate servos
```

```
import time

import Adafruit_PCA9685

import calValues

#import the settings for servos


pwm = Adafruit_PCA9685.PCA9685()
pwm.set_pwm_freq(60)


#servo mapping
# pmw 0 head tilt
HEAD_TILT_SERVO = 0
# pwm 1 head turn
HEAD_TURN_SERVO = 1
# pwm 2 wheels turn
WHEELS_TURN_SERVO = 2


if __name__ == '__main__':

    print("--------------------")
    print("calibrate wheel turn")
    print("--------------------")

    for i in range(calValues.turn_right_max,
            calValues.turn_left_max,10):
        pwm.set_pwm(WHEELS_TURN_SERVO,0, i)
        print("servoValue = ", i)
        time.sleep(0.5)


    print("--------------------")
    print("calibrate head turn")
    print("--------------------")

    for i in range(calValues.look_right_max,
            calValues.look_left_max,10):
        pwm.set_pwm(HEAD_TURN_SERVO,0, i)
        print("servoValue = ", i)
        time.sleep(0.5)


    print("--------------------")
    print("calibrate head up/down")
    print("--------------------")
```

```
    for i in range(calValues.look_up_max,
            calValues.look_down_max,10):
        pwm.set_pwm(HEAD_TILT_SERVO,0, i)
        print("servoValue = ", i)
        time.sleep(0.5)
```

The code is pretty straight forward, but let me talk about one of the servo program loops.

```
    for i in range(calValues.look_right_max,
            calValues.look_left_max,10):
        pwm.set_pwm(HEAD_TURN_SERVO,0, i)
        print("servoValue = ", i)
        time.sleep(0.5)
```

This loop steps through the servo range as given in `calValues.py` from the right to the left, turning the head in steps of 10. This gives you a pretty good idea where the right, left, and center should be (taking into account the robot frame too!) and you can add those values to `calValues.py`.

The `calValues.py` file holds the calibration values for the servo motors. You replace the values in this program with your own values from `calibrateServos.py`:

```
# Servo calibration values

# head
look_up_max     = 150
look_down_max   = 420
look_tilt_middle = 330

# head turn
look_right_max   = 200
look_left_max     = 450
look_turn_middle = 310

# wheels
turn_right_max  = 180
turn_left_max   = 460
turn_middle     = 320

# turn_speed
look_turn_speed  = 5

# motor speed

left_spd   = 100         #Speed of the car
right_spd  = 100         #Speed of the car
```

Running tests on your rover in Python

Okay, now you have the PiCar-B assembled, time to run some tests. If you have installed the Adeept software on your Raspberry Pi, you have to disable the auto startup of their software.

To do so, change the following line in the `~/.config/autostart/car.desktop` file:

```
Exec=sudo python3 /home/Adeept_PiCar-B/server/server.py
```

To

```
#Exec=sudo python3 /home/Adeept_PiCar-B/server/server.py
```

And then reboot your pi with: `sudo reboot`.

Of course, we told you not to install the software, but if you got too excited and did then you need to do the above or else you will have conflicts in the software.

Here is a video of what the following Python test software does on the PiCar-B robot: `https://youtu.be/UvxRBJ-tFw8`.

That is what you will be running very shortly.

Installing software for the CarPi-B Python test

First of all, go to this book's support page on `www.dummies.com` (refer to the Introduction) and download the software for Book 7, Chapter 2.

Go into the Testing directory and then complete these instructions, which are necessary to get the 12 programmable RGB LEDs to work on the Raspberry Pi:

1. **Install some developer libraries, which allow us to compile the software. This is installed using the normal Raspberry Pi installer, as follows:**

   ```
   sudo apt-get install build-essential python3-dev git scons swig
   ```

2. **Download the neopixel code from github using the `clone` command, which copies all the source code to your local computer:**

   ```
   git clone https://github.com/jgarff/rpi_ws281x.git
   ```

3. **Change to that directory and run `scons` to compile the software:**

```
cd rpi_ws281x
scons
```

4. **Change to the `python` directory and install the Python module from there:**

```
cd python
```

5. **Install the Python 3 library file using:**

```
sudo python3 setup.py install
```

The PiCar-B Python test code

The file is approximately 370 lines long, so we don't provide a full listing here. The important parts of this file have been discussed along with the individual components and sensors above. There are a number of other libraries and files in the Testing directory.

If you haven't already, go to this book's support page at `www.dummies.com` (refer to the Introduction) and download the software for Book 7, Chapter 2.

Run the test software by typing:

```
sudo python3 PiCar-B-Test.py
```

You should immediately see your car start doing the testing sequence, as shown at `https://youtu.be/UvxRBJ-tFw8`.

Pi camera video testing

To prepare for this test, you must be running in a GUI on your Raspberry Pi. If you are using VNC to display the GUI on another computer, you must do enable a VNC option. Open up the options of the VNC server in the upper-right corner, go into Options, then down into Troubleshooting and click the Enable Direct Capture Mode checkbox. (See Figure 2-11.)

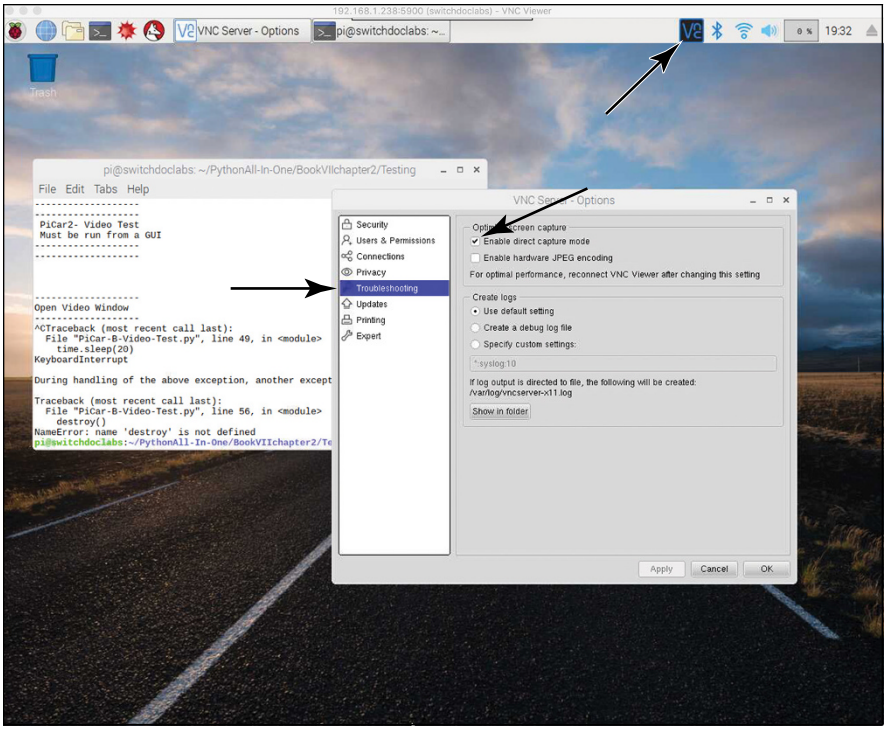

FIGURE 2-11: Setting the VNC viewer option.

This test software (`PiCar-B-Video-Test.py`) follows:

```
#!/usr/bin/python3
DEBUG = True
VIDEOTEST = True
# runs through a video tests for the PiCar-B

import RPi.GPIO as GPIO
import motor
import ultra
import socket
import time
import threading
import turn
import led
import os
import picamera
from picamera.array import PiRGBArray
import cv2
```

```
import calValues


if __name__ == '__main__':

    camera = picamera.PiCamera()              #Camera initialization
    camera.resolution = (640, 480)
    camera.framerate = 7
    rawCapture = PiRGBArray(camera, size=(640, 480))


    try:

        print ("--------------------")
        print ("--------------------")
        print (" PiCar2- Video Test")
        print (" Must be run from a GUI")
        print ("--------------------")
        print ("--------------------")
        print ()
        print ()

        if (VIDEOTEST):

            print ()
            print ("--------------------")
            print ("Open Video Window")
            print ("--------------------")



            camera.resolution = (1024, 768)
            camera.start_preview(fullscreen=False,
                    window=(100,100,256,192))
            time.sleep(20)
            camera.preview.window=(200,200,256,192)
            time.sleep(2)
            camera.preview.window=(0,0,512,384)
            time.sleep(2)
            camera.close()
    except KeyboardInterrupt:
        destroy()
```

This code opens up a small window on the GUI of the Raspberry Pi, waits 20 seconds, moves the window and resizes it, waits 2 seconds, moves it again, and then closes the window.

Now you are done building and testing your robot. Next, we are going to add some Python brains and have some fun with the robot.

IN THIS CHAPTER

- Learning how to move and sense with your robot
- Understanding autonomous vehicles
- Playing and using the Adeept software

Chapter 3

Programming Your Robot Rover in Python

Okay, let's review where you are now. You have a basic understanding of robots and (more importantly) of the major components of robots and how they work. You understand that these components can be controlled with Python and that they can work together to accomplish robotic tasks. That's really a lot of information. Next, we show you how to string these things together to make a very, very simple "robotic brain" to make our robot move by itself. This won't quite be a fully self-driving car, but after doing this you will have some sense of how those cars are programmed.

Building a Simple High-Level Python Interface

Let's first make a short Python wrapping function that allows us to build much more complicated programs while hiding the complexity of the robot hardware.

Our high-level interface is a `python` class file called `RobotInterface.py`. The code length is beyond what we want to show in this book, so let us describe a couple of functions and then document the rest.

The motorForward function

The `motorForward` function is typical of the motor functions located within the `RobotInterface` class:

```
        def motorForward(self, speed, delay):
            motor.motor_left(self.MOTOR_START, self.forward,speed)
            motor.motor_right(self.MOTOR_START, self.backward,speed)
            time.sleep(delay)
            motor.motor_left(self.MOTOR_STOP, self.forward,speed)
            motor.motor_right(self.MOTOR_STOP, self.backward,speed)
```

This function drives the robot forward for the number of seconds passed into the function in the `delay` argument. This means that when you call this function, you must use an actual number in the `delay` argument.

It basically starts the motors running forward, waits a delay and then shuts them off.

The wheelsLeft function

```
        def wheelsLeft(self):
    pwm.set_pwm(self.WHEELS_TURN_SERVO, 0,
            calValues.turn_left_max)
     time.sleep(0.05)
```

The `wheelsLeft` function sets the `WHEELS_TURN_SERVO` to the leftmost position of the wheels and then delays 50ms. Why the delay? You will see these delays scattered through the `RobotInterface` class file. These are to keep multiple servo commands back-to-back from exceeding the current capacity of the power supply. By delaying the next servo command 50ms, the high current transient caused by the moving the first servo has a chance to die away before the next servo command is executed.

The wheelsPercent function

This function allows the user to set the servo to a percent of the total range of the servo motor. It goes from the full left (0) to full right (100) for the wheels; 50 would be approximately in the middle. It may differ a little from the middle if you have an asymmetric range of motion of your servo. If you do, then use the `wheelsMiddle()` function to center your wheels.

This code calculates the total range of motion of the servo motor and then multiplies it by the percentage requested. It then sets the servo motor to the requested range:

```
        def wheelsPercent(self,percent):
            adder = (calValues.turn_left_max –
                 calValues.turn_right_max)*(percent/100.0)
            pwm.set_pwm(self.WHEELS_TURN_SERVO, 0,
                 int(calValues.turn_right_max + adder))
            time.sleep(0.05)
```

Making a Single Move with Python

First of all, go to this book's support page at `www.dummies.com` (refer to the Introduction) and download the software for Book 7, Chapter 3.

In the following code, we move the robot a short distance ahead with a `motorForward` command and then back to its original position with a `motorBackward()` command. Hopefully, you're starting to see the magic of this approach.

The "Single Move" code:

```
#!/usr/bin/python3
# Robot Interface Test

import RobotInterface
import time

RI = RobotInterface.RobotInterface()

print ("Short Move Test")

RI.wheelsMiddle()

RI.motorForward(100,1.0)
time.sleep(1.0)
RI.motorBackward(100,1.0)
```

First, we import the `RobotInterface` class library and also the time library (for `sleep()`). Note the simplicity of this. The complexity underlying robot interface libraries are hidden by this class:

```
import RobotInterface
import time
```

Then we initialize the `RobotInterface` module and assign the module to the variable `RI`:

```
RI = RobotInterface.RobotInterface()

print ("Short Move Test")
```

We center the wheels with the `wheelsMiddle()` function command:

```
RI.wheelsMiddle()
```

Now comes the good part. We drive the robot forward for one second, pause a second, and then run it backwards for one second to the robot's original position:

```
RI.motorForward(100,1.0)
time.sleep(1.0)
RI.motorBackward(100,1.0)
```

Pretty simple, right? Call the file `singleMove.py`.

Here is a video of what you should see when you run this code on your robot in a terminal window: `https://youtu.be/UT0PG7z2ccE`.

The job isn't finished until when? Oh yes, when the documentation is finished. Off to document the `RobotInterface` class functions.

Functions of the RobotInterface Class

In this section, we document the functions of the `RobotInterface` class. We show you the Robot Interface Test program, and then it is off to the races in building robot software! The `RobotInterface` class is derived from both original software by the author and also from internal drivers from the PiCar-B Adeept software.

Front LED functions

The following functions control the two LEDs on the front of the robot.

set_Front_LED_On()

This function sets the front LED to `On`:

```
set_Front_LED_On(colorLED)
```

Remember that these two front LEDs are tricolor with red, green, and blue LEDs, each of which is individually controllable. The parameter, `colorLED`, controls the side of the robot to turn on and the color to turn on. You set the color and select the side by using the following constants from the `RobotInterface` class:

```
RobotInterface.left_R
RobotInterface.left_G
RobotInterface.left_B

RobotInterface.right_R
RobotInterface.right_G
RobotInterface.right_B
```

For example, `RobotInterface.left_R` turns on the red LED on the Robot Left side. *Robot Left* refers to the left side of the robot as viewed from the rear of the robot. You can make multiple calls to this program to turn on all three of the LEDs. Turning on an already on LED does not hurt anything and is ignored.

A more sophisticated driver could be written driving the LEDs GPIOs with PWM (pulse-width modulation) allowing even greater color mixing. Note that unless you are using hardware PWM pins on the Raspberry Pi, you will see flickering of the LEDs when using this technique because of the Raspberry Pi multitasking operating system. You could, however, write drivers for the PCA9685 servo driver board (on the PiCar-B) that currently drives the servo motors to fix the flickering problem.

set_Front_LED_Off()

This function sets the Front LED to `Off`:

```
set_Front_LED_Off(colorLED)
```

Remember that these two front LEDs are tricolor with red, green, and blue LEDs, each of which is individually controllable.

The parameter, `colorLED`, controls the side of the robot to turn on and the color to turn on. You set the color and select the side by using the following constants from the `RobotInterface` class:

```
RobotInterface.left_R
RobotInterface.left_G
RobotInterface.left_B

RobotInterface.right_R
RobotInterface.right_G
RobotInterface.right_B
```

For example, `RobotInterface.left_R` turns on the red LED on the Robot Left side. *Robot Left* refers to the left side of the robot as viewed from the rear of the robot. You can make multiple calls to this program to turn on all three of the LEDs. Turning on an already on LED does not hurt anything and is ignored.

Pixel strip functions

There are 12 LEDs on the robot strung together as a single 12 LED strip. These RGB LEDs are called Pixels and are controlled by a single serial line that runs through all 12 of the LEDs. They are controlled by a fairly sophisticated and touchy serial sequence of precisely timed pulses from the Raspberry Pi. The Raspberry Pi (again because of the operating system) cannot generate these pulses accurately enough using Python and GPIO signals. Therefore a complex driver using the DMA (direct memory access) interface on the Raspberry Pi has been created by a "jgarff" (rpi_ws281x — very clever coding) and we are using that driver to generate those signals. Our `RobotInterface` software hides all this complexity from the user.

rainbowCycle()

This call starts a rainbow cycle that uses all 12 of the Pixel LEDs and runs through many colors:

```
rainbowCycle(wait_ms = 20, iterations = 3)
```

The parameter `wait_ms` sets the delay (in milliseconds) between each color change. It defaults to 20ms. `Interations` sets the number of full color cycles to perform before returning, and it defaults to 3.

colorWipe()

This function sets all 12 Pixel LEDs to the same color:

```
colorWipe(color)
```

For example, `colorWipe(color(0,0,0))` sets all the Pixels to `Off`. This is a handy way to set all 12 pixels to the same color. The parameter, `color`, specifies the color to be used using the `color()` function. (See the `color()` function later in this chapter.)

theaterChaseRainbow()

This function starts a 40-second-long pattern of chasing LEDs on all 12 of the LEDs:

```
theaterChaseRainbow(wait_ms = 50)
```

The `wait_ms` parameter sets the delay between each of the movements in milliseconds. It defaults to 50 milliseconds.

setPixelColor()

This function sets an individual pixel (numbered from 0 through 11) to a specific color. Brightness affects all the pixels and uses the last brightness value set:

```
setPixelColor(pixel, color, brightness)
```

The `pixel` parameter sets the specific pixel to set. Pixels are numbered 0–11.

The `color` parameter specifies the color to be used using the `color()` function.

The `brightness` parameter sets the brightness (0–255) for *all* the pixel string.

Color()

This function is a helper function that converts the R, G and B values into a single 24 bit integer used by the internal Pixel driver:

```
Color(red, green, blue, white = 0)
```

The parameters `red`, `green`, and `blue` are integers and range from 0 (off) to 255 (fully on). The `white=0` parameter is for RGBW Pixel LEDs. The Pixels on the robot are RGB LEDs.

allLEDSOff()

This function turns all the LEDs on the robot off, both the front two LEDs and the 12 LED Pixel string:

```
allLEDSOff()
```

Ultrasonic distance sensor function

The ultrasonic distance sensor functions by sending out a pulse of high frequency sound and then counting the time before it bounces back to the receiver. Because we know the speed of sound, we can calculate the distance in front of the sensor. It's not a perfect method (we would rather be using a laser!), but it is pretty good, and it makes for a good starting distance sensor.

fetchUltraDistance()

This function does an immediate measurement from the ultrasonic distance sensor in the head of the robot and returns the distance in centimeters (cm):

```
fetchUltraDistance()
```

Main motor functions

The main motor on our robot is what drives the back wheels and makes our robot move. The motor functions are used to tell how fast and how long to run the main motor.

motorForward()

This function drives the robot forward at `speed` for the `delay` number of seconds before the shutting off the motors:

```
motorForward(speed, delay)
```

The parameter `speed` sets the duty cycle of the PWM GPIO pin driving the interface for the motor. It goes from 0 (off) to 100 (fast).

The parameter `delay` tells the driver how long to run the motor in seconds.

motorBackward()

This function drives the robot backwards at `speed` for the `delay` number of seconds before the shutting off the motors:

```
motorBackward(speed, delay)
```

The parameter `speed` sets the duty cycle of the PWM GPIO pin driving the interface for the motor. It goes from 0 (off) to 100 (fast).

The parameter `delay` tells the driver how long to run the motor in seconds.

stopMotor()

This function immediately stops the main motor:

```
stopMotor()
```

This is really only useful if, for example, you were driving the motor in another thread. You can think of a thread as another program running at the same time as your main program. This is a sophisticated programming technique that has some huge benefits to writing robot code.

Servo functions

This group of functions control the three servos on the robot: head-turning, head-tilting, and front wheels.

headTurnLeft()

This function turns the robot head all the way to the left:

```
headTurnLeft()
```

"All the way to the left" is defined in the `calValues.py` file. Refer to Chapter 2 in this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

headTurnRight()

This function turns the robot head all the way to the right:

```
headTurnRight()
```

"All the way to the right" is defined in the `calValues.py` file. Refer to Chapter 2 in this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

headTurnMiddle()

This function turns the robot head towards the front:

```
headTurnMiddle()
```

"The middle" is defined in the `calValues.py` file. Refer to Chapter 2 in this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

headTurnPercent()

This function turns the head from 0 (all the way to the left) to 100 (all the way to the right):

```
headTurnPercent(percent)
```

This is useful for more precisely aiming the head. Again, "all the way to the left" and "all the way to the right" " are defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

The parameter `percent` has values 0–100 and represents the linear percent from left to right. Note that the value 50 may not be quite in the middle because your servos may not be set exactly in the middle of their range.

headTiltDown()

This function tilts the robot head all the way down:

```
headTiltDown()
```

"All the way down" is defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

headTiltUp()

This function tilts the robot head all the way up:

```
headTiltUp()
```

"All the way up" is defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

headTiltMiddle()

This function tilts the robot head towards the front:

```
headTiltMiddle()
```

"The middle" is defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

headTiltPercent()

This function turns the head from 0 (all the way down) to 100 (all the way to the up):

```
headTiltPercent(percent)
```

This is useful for more precisely aiming the head. Again, "all the way down and "all the way up"" are defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

The parameter `percent` has values 0–100 and represents the linear percent from down to up. Note that the value 50 may not be quite in the middle because your servos may not be set exactly in the middle of their range as set by the servo calibration process and because of the way your robot was physically built.

wheelsLeft()

This function turns the robot front wheels all the way to the left:

```
wheelsLeft()
```

"All the way to the left" is defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

wheelsRight()

This function turns the robot front wheels all the way to the right:

```
wheelsRight()
```

"All the way to the right" is defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

wheelsMiddle()

This function turns the robot front wheels to the middle:

```
wheelsMiddle()
```

"Middle" is defined in the `calValues.py` file. Refer to Chapter 2 of this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

wheelsPercent()

This function turns the head from 0 (all the way to the left) to 100 (all the way to the right):

```
wheelsPercent(percent)
```

This is useful for more precisely setting the direction of the robot front wheels. Again, "all the way to the left and "all the way to the right"" are defined in the `calValues.py` file. Refer to Chapter 2 in this minibook for information on the calibration values and how to set them using the `calibrateServos.py` program.

The parameter `percent` has values 0–100 and represents the linear percent from down to up. Note that the value 50 may not be quite in the middle because of how your servos may not be set exactly in the middle of their range as set by the servo calibration process and how your robot was physically built.

General servo function

We have included general functions to control all the servos at once. Calling this function moves all servos to the center position.

centerAllServos()

This function puts all the servos on the robot to the center of their ranges as defined in the `calValues.py` file:

```
centerAllServos()
```

The Python Robot Interface Test

Now that we have defined our robot API (applications programming interface), let's run the system test using the `RobotInterface` Python class. This program is useful for two reasons. First, it tests all our functions in the `RobotInterface` class. Second, it shows how to use each of the functions in a Python program.

The code for `RITest.py`:

```
#!/usr/bin/python3
# Robot Interface Test

import RobotInterface
```

```
import time


RI = RobotInterface.RobotInterface()

print ("Robot Interface Test")

print ("LED tests")
RI.set_Front_LED_On(RI.left_R)
time.sleep(0.1)
RI.set_Front_LED_On(RI.left_G)
time.sleep(0.1)
RI.set_Front_LED_On(RI.left_B)
time.sleep(1.0)
RI.set_Front_LED_On(RI.right_R)
time.sleep(0.1)
RI.set_Front_LED_On(RI.right_G)
time.sleep(0.1)
RI.set_Front_LED_On(RI.right_B)
time.sleep(1.0)

RI.set_Front_LED_Off(RI.left_R)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.left_G)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.left_B)
time.sleep(1.0)
RI.set_Front_LED_Off(RI.right_R)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.right_G)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.right_B)
time.sleep(1.0)

RI.rainbowCycle(20, 1)
time.sleep(0.5)

# Runs for 40 seconds
#RI.theaterChaseRainbow(50)
#time.sleep(0.5)

print ("RI.Color(0,0,0)=", RI.Color(0,0,0))
RI.colorWipe(RI.Color(0,0,0))
time.sleep(1.0)

for pixel in range (0,12):
    RI.setPixelColor(pixel,RI.Color(100,200,50),50)
    time.sleep(0.5)
```

```
print ("Servo Tests")
RI.headTurnLeft()
time.sleep(1.0)
RI.headTurnRight()
time.sleep(1.0)
RI.headTurnMiddle()
time.sleep(1.0)

RI.headTiltDown()
time.sleep(1.0)
RI.headTiltUp()
time.sleep(1.0)
RI.headTiltMiddle()
time.sleep(1.0)

RI.wheelsLeft()
time.sleep(1.0)
RI.wheelsRight()
time.sleep(1.0)
RI.wheelsMiddle()
time.sleep(1.0)

print("servo scan tests")
for percent in range (0,100):
    RI.headTurnPercent(percent)
for percent in range (0,100):
    RI.headTiltPercent(percent)
for percent in range (0,100):
    RI.wheelsPercent(percent)


print("motor test")
RI.motorForward(100,1.0)
time.sleep(1.0)
RI.motorBackward(100,1.0)

print("ultrasonic test")

print ("distance in cm=", RI.fetchUltraDistance())

print("general function test")

RI.allLEDSOff()
RI.centerAllServos()
```

Note: We have commented out the test code for `RI.theaterChaseRainbow()` because it runs for 40 seconds.

ROS: THE ROBOT OPERATING SYSTEM

We have written a fairly simple interface class for the PiCar-B robot. This allows us to control the robot from a Python program. If we had more room in this book (actually another whole book could be written about the use of ROS for a robot like ours), we would connect up our robot to the ROS (Robot Operating System). ROS is a system specifically designed for controlling robots in a distributed system. Even though it is called the *Robot Operating System*, it really isn't an operating system.

ROS is what is called *middleware*. Middleware is software that is designed to manage the complexity of writing software in a complex and heterogenous (meaning lots of different types of robots and sensors) environment. ROS allows us to treat very different robots in a very similar manner.

ROS operates using what is called a publish-subscribe system. It works like a newspaper. A newspaper publishes stories, but only the people that subscribe to the newspaper see those stories. You might have a subscriber that only wants a subscription to the comics. Or the front page.

ROS works like that. A robot like ours may publish the current value of the ultrasonic sensor or the current camera image (or even a video stream) and other computers or robots on the network could subscribe to the video stream and see what your robot is seeing. And your robot could subscribe to other sensors (such as a temperate sensor located in the middle of the room) or even look at what the other robots are seeing. The power of this technique is that now you can make your robot part of an ecosystem consisting of computers, sensors, and even people making use of your data and contributing information to your robot.

We could build a ROS interface on our robot and then we could control it remotely and feed sensor data to other computers.

In many ways, ROS really rocks. Find out more about ROS at `http://www.ros.org/`.

Run the program by typing **`sudo python3 RITest.py`** into a terminal window. Note that you *have* to use `sudo` because the Pixel LEDs require root permission (granted by `sudo`) to correctly run.

```
Robot Interface Test
LED tests
RI.Color(0,0,0)= 0
Servo Tests
servo scan tests
motor test
```

```
ultrasonic test
distance in cm= 16.87312126159668
general function test
```

Here's a link to the video of the `RobotInterface` class test: `https://youtu.be/1vi-UGao0oI`

Coordinating Motor Movements with Sensors

The ability to modify and coordinate motor movements with sensor movements is key to movement in the environment. Sensors give information to be acted upon as well as feedback from our motions. Think of the act of catching a baseball with a glove. Your sensors? Eyes and the sense of touch. Your eyes see the ball and then move your hand and arm to intercept the ball. That's coordinating your movement with a sensor. The feedback? Knowing you have caught the ball in your mitt by the feeling of it hitting your gloved hands. Of course, you are also updating your internal learning system to become better at catching the ball.

PiCar-B has two sensors that read information from the outside world. The ultrasonic sensor can detect what is in front of the robot while the camera can photograph the world, and then the robot can analyze what it is seeing. The first thing to remember, however, is that robot vision is hard. Very hard. We touch upon using the camera images for analysis in the next chapter of this book. Chapter 4 talks about using artificial intelligence in robots, and we will be building an example of how to do this using machine learning.

For our example, we will focus on the simpler sensor, the ultrasonic distance sensor.

Here is an example of code that will move the robot forward or backwards depending on the distance from the object in front of the robot. Here is the Python code for `simpleFeedback.py`:

```
#!/usr/bin/python3
# Robot Interface Test

import RobotInterface
import time

DEBUG = True
```

```
RI = RobotInterface.RobotInterface()

print ("Simple Feedback Test")

RI.centerAllServos()
RI.allLEDSOff()

# Ignore distances greater than one meter
DISTANCE_TO_IGNORE = 1000.0
# Close to 10cm with short moves
DISTANCE_TO_MOVE_TO = 10.0
# How many times before the robot gives up
REPEAT_MOVE = 10

def bothFrontLEDSOn(color):
    RI.allLEDSOff()
    if (color == "RED"):
        RI.set_Front_LED_On(RI.right_R)
        RI.set_Front_LED_On(RI.left_R)
        return
    if (color == "GREEN"):
        RI.set_Front_LED_On(RI.right_G)
        RI.set_Front_LED_On(RI.left_G)
        return
    if (color == "BLUE"):
        RI.set_Front_LED_On(RI.right_B)
        RI.set_Front_LED_On(RI.left_B)
        return




try:
    Quit = False
    moveCount = 0
    bothFrontLEDSOn("BLUE")
    while (Quit == False):
        current_distance = RI.fetchUltraDistance()
        if (current_distance >= DISTANCE_TO_IGNORE):
            bothFrontLEDSOn("BLUE")
            if (DEBUG):
                print("distance too far ={:6.2f}cm"
                        .format(current_distance))
        else:
            if (current_distance <= 10.0):
                # reset moveCount
                # the Robot is close enough
```

```
                bothFrontLEDSOn("GREEN")
                moveCount = 0
                if (DEBUG):
                    print("distance close enough ={:6.2f}cm"
                            .format(current_distance))

                time.sleep(5.0)
                # back up and do it again
                RI.motorBackward(100,1.0)
            else:
                if (DEBUG):
                    print("moving forward ={:6.2f}cm"
                            .format(current_distance))
                # Short step forward
                bothFrontLEDSOn("RED")
                RI.motorForward(90,0.50)
                moveCount = moveCount + 1

        # Now check for stopping our program
        time.sleep(1.0)
        if (moveCount > REPEAT_MOVE):
            Quit = True


except KeyboardInterrupt:
    print("program interrupted")

print ("program finished")
```

This is a great example of the use of feedback in robotics. The robot first checks to see if it is less than one meter (1000cm) from the wall. If it is, it slowly starts to advance towards the wall in short little steps. When it is closer than 10cm to the wall, it stops, then waits five seconds and backs up to do it again.

It also gives up if it takes more than 10 moves to get to the wall, if somehow we have moved further than 1000cm away from the wall, or if the user has hit Ctrl-C to interrupt the program.

Note how we use the LEDs to give feedback to surrounding people as to what the robot is doing. This sort of visual feedback is an important part of making human–robot interaction more efficient, understandable, and safer.

The main structure of the program is contained within a Python `while` loop. As long as we haven't interrupted the program (or one of the other quit criteria hasn't been satisfied) our little robot will keep working until the battery goes dead.

Copy the code into `simpleFeedback.py` and give it a try by executing `sudo python3 simpleFeedback.py`. Here are the printed results:

```
Simple Feedback Test
moving forward = 55.67cm
moving forward = 44.48cm
moving forward = 34.22cm
moving forward = 26.50cm
moving forward = 17.53cm
distance close enough = 9.67cm
moving forward = 66.64cm
moving forward = 54.25cm
moving forward = 43.55cm
moving forward = 36.27cm
moving forward = 28.44cm
moving forward = 21.08cm
moving forward = 13.55cm
distance close enough = 6.30cm
moving forward = 64.51cm
moving forward = 52.89cm
moving forward = 43.75cm
moving forward = 33.95cm
moving forward = 26.79cm
^Cprogram interrupted
program finished
```

And you can see the feedback video here: `https://youtu.be/mzZIMxch5k4`.

Play with this code. Try different things and different constants to get different results.

Making a Python Brain for Our Robot

Now we are going to create a simple self-driving car. In a sense, we are going to apply the results above to create an autonomous vehicle that is not very smart but illustrates the use of feedback in decision-making.

This Python brain we are writing is nothing more than a combination of our code for sensing the wall (from earlier in this chapter) and for generating a random walk based on the information. After running the code for a while, we saw where the robot would get stuck and added code to detect back out of stuck positions.

Note: Make sure you have fully charged up batteries to run this code. When the batteries dip a bit your motor speed dramatically decreases. How to fix this? Use bigger batteries.

The "Robot Brain" code:

```
#!/usr/bin/python3
# Robot Brsin

import RobotInterface
import time

from random import randint

DEBUG = True

RI = RobotInterface.RobotInterface()

print ("Simple Robot Brain")

RI.centerAllServos()
RI.allLEDSOff()

# Close to 20cm
CLOSE_DISTANCE = 20.0
# How many times before the robot gives up
REPEAT_TURN = 10


def bothFrontLEDSOn(color):
    RI.allLEDSOff()
    if (color == "RED"):
        RI.set_Front_LED_On(RI.right_R)
        RI.set_Front_LED_On(RI.left_R)
        return
    if (color == "GREEN"):
        RI.set_Front_LED_On(RI.right_G)
        RI.set_Front_LED_On(RI.left_G)
        return
    if (color == "BLUE"):
        RI.set_Front_LED_On(RI.right_B)
        RI.set_Front_LED_On(RI.left_B)
        return

STUCKBAND = 2.0
# check for stuck car by distance not changing
def checkForStuckCar(cd,p1,p2):
```

```
    if (abs(p1-cd) < STUCKBAND):
        if (abs(p2-cd) < STUCKBAND):
            return True
    return False



try:
    Quit = False
    turnCount = 0
    bothFrontLEDSOn("BLUE")

    previous2distance = 0
    previous1distance = 0

    while (Quit == False):
        current_distance = RI.fetchUltraDistance()
        if (current_distance >= CLOSE_DISTANCE ):
            bothFrontLEDSOn("BLUE")
            if (DEBUG):
                print("Continue straight ={:6.2f}cm"
                        .format(current_distance))
            if (current_distance > 300):
                # verify distance
                current_distance = RI.fetchUltraDistance()
                if (current_distance > 300):
                    # move faster
                    RI.motorForward(90,1.0)
            else:
                RI.motorForward(90,0.50)
            turnCount = 0

        else:
            if (DEBUG):
                print("distance close enough so turn ={:6.2f}cm"
                        .format(current_distance))
            bothFrontLEDSOn("RED")
            # now determine which way to turn
            # turn = 0 turn left
            # turn = 1 turn right
            turn = randint(0,1)

            if (turn == 0): # turn left
                # we turn the wheels right since
                # we are backing up
                RI.wheelsRight()
            else:
```

```
                # turn right

                # we turn the wheels left since
                # we are backing up
                RI.wheelsLeft()

            time.sleep(0.5)
            RI.motorBackward(100,1.00)
            time.sleep(0.5)
            RI.wheelsMiddle()
            turnCount = turnCount+1
            print("Turn Count =", turnCount)

        # check for stuck car
        if (checkForStuckCar(current_distance,
            previous1distance, previous2distance)):
            # we are stuck.  Try back up and try Random turn
            bothFrontLEDSOn("RED")
            if (DEBUG):
                print("Stuck - Recovering ={:6.2f}cm"
                        .format(current_distance))
            RI.wheelsMiddle()
            RI.motorBackward(100,1.00)

            # now determine which way to turn
            # turn = 0 turn left
            # turn = 1 turn right
            turn = randint(0,1)


            if (turn == 0): # turn left
                # we turn the wheels right since
                # we are backing up
                RI.wheelsRight()
            else:
                # turn right

                # we turn the wheels left since
                # we are backing up
                RI.wheelsLeft()
            time.sleep(0.5)
            RI.motorBackward(100,2.00)
            time.sleep(0.5)
            RI.wheelsMiddle()


        # load state for distances
        previous2distance = previous1distance
        previous1distance = current_distance
```

```
        # Now check for stopping our program
        time.sleep(0.1)
        if (turnCount > REPEAT_TURN-1):
            bothFrontLEDSOn("RED")
            if (DEBUG):
                print("too many turns in a row")
            Quit = True


except KeyboardInterrupt:
    print("program interrupted")

print ("program finished")
```

This seems to be a much more complex program than our ultrasonic sensor program earlier in this chapter, but it is really not.

We took the same structure of the program (the `while` loop) and added several features.

First, we added a clause to speed up the car when we were far away from an obstacle (over 300cm):

```
        if (current_distance >= CLOSE_DISTANCE ):
            bothFrontLEDSOn("BLUE")
            if (DEBUG):
                print("Continue straight ={:6.2f}cm"
                        .format(current_distance))
            if (current_distance > 300):
                # verify distance
                current_distance = RI.fetchUltraDistance()
                if (current_distance > 300):
                    # move faster
                    RI.motorForward(90,1.0)
            else:
                RI.motorForward(90,0.50)
            turnCount = 0
```

We continued to move in short little hops as the robot gets closer to the wall. When the robot gets within about 10cm of the wall, the robot decides to turn its front wheels in a random direction and backs up to try a new direction:

```
            if (DEBUG):
                print("distance close enough so turn ={:6.2f}cm"
                        .format(current_distance))
            bothFrontLEDSOn("RED")
            # now determine which way to turn
            # turn = 0 turn left
```

```
            # turn = 1 turn right
            turn = randint(0,1)

            if (turn == 0): # turn left
                # we turn the wheels right since
                # we are backing up
                RI.wheelsRight()
            else:
                # turn right

                # we turn the wheels left since
                # we are backing up
                RI.wheelsLeft()

            time.sleep(0.5)
            RI.motorBackward(100,1.00)
            time.sleep(0.5)
            RI.wheelsMiddle()
            turnCount = turnCount+1
            print("Turn Count =", turnCount)
```

We ran the robot for quite a while with just this logic, and we would see it get stuck if part of the robot was blocked, but the ultrasonic sensor was still picking up greater than 10cm distance.

To fix this, we added a running record of the past two ultrasonic distance readings, and if you had three readings +/- 2.0cm, then the robot would decide it was stuck and back up, turn randomly, and proceed again to wandering. Worked like a champ:

```
        if (checkForStuckCar(current_distance,
            previous1distance, previous2distance)):
            # we are stuck.  Try back up and try Random turn
            bothFrontLEDSOn("RED")
            if (DEBUG):
                print("Stuck - Recovering ={:6.2f}cm"
                        .format(current_distance))
            RI.wheelsMiddle()
            RI.motorBackward(100,1.00)

            # now determine which way to turn
            # turn = 0 turn left
            # turn = 1 turn right
            turn = randint(0,1)
```

```
            if (turn == 0): # turn left
                # we turn the wheels right since
                # we are backing up
                RI.wheelsRight()
            else:
                # turn right

                # we turn the wheels left since
                # we are backing up
                RI.wheelsLeft()
            time.sleep(0.5)
            RI.motorBackward(100,2.00)
            time.sleep(0.5)
            RI.wheelsMiddle()
```

We set the robot down in a room that has furniture and a complex set of walls and let it loose. Here are the results from the console:

```
Simple Robot Brain
Continue straight =115.44cm
Continue straight =108.21cm
Continue straight =101.67cm
Continue straight = 95.67cm
Continue straight = 88.13cm
Continue straight = 79.85cm
Continue straight = 70.58cm
Continue straight = 63.89cm
Continue straight = 54.36cm
Continue straight = 44.65cm
Continue straight = 36.88cm
Continue straight = 28.32cm
Continue straight = 21.10cm
distance close enough so turn = 11.33cm
Turn Count = 1
Continue straight = 33.75cm
Continue straight = 25.12cm
distance close enough so turn = 18.20cm
Turn Count = 1
Continue straight = 40.51cm
Continue straight = 33.45cm
Continue straight = 24.73cm
distance close enough so turn = 14.83cm
Turn Count = 1
Continue straight = 35.72cm
Continue straight = 26.13cm
distance close enough so turn = 18.56cm
```

```
Turn Count = 1
Continue straight = 43.63cm
Continue straight = 37.74cm
Continue straight = 27.33cm
Continue straight = 84.01cm
```

You can see the robot drive towards a wall and then turn several times to find a way out and then continue on. in the video here: `https://youtu.be/U7_FJzRbsRw`.

A Better Robot Brain Architecture

If you look at the *robotBrain.py* software from an software architectural perspective, one thing jumps out. The main part of the program is a single `while` loop that polls the sensor (the ultrasonic sensor) and then does one thing at a time (moves, turns, and so on) and then polls it again. This leads to the somewhat jerky behavior of the robot (move a little, sense, move a little, sense, and so on). Although this is the simplest architecture we could use for our example, there are better, albeit more complicated, ways of doing this that are beyond the scope of our project today.

These better architectures are based on what are called *threads*. You can think of threads as separate programs that run at the same time and communicate to each other by using things called *semaphores* and *data queues*. Semaphores and data queues are simply methods by which a thread can communicate with other threads in a safe manner. Because both threads are running at the same time, you have to be careful how they talk and exchange information. This is not complicated, if you follow the rules.

A better architecture for our robot brain would be like this:

- **Motor thread:** This thread controls the motors. It makes them run on command and can stop the motors anytime.
- **Sensor thread:** This thread reads the ultrasonic sensor (and any other sensors you may have) periodically so you always have the current distance available.
- **Head thread:** This thread controls the head servos using commands from the Command thread.
- **Command thread:** This is the brains of software. It takes current information from the Sensor thread and sends commands to the motors and head to their respective thread.

This architecture leads to a much smoother operation of the robot. You can have the motors running while you are taking sensor values and sending commands simultaneously. This is the architecture that is used in the Adeept software `server.py` file included with the PiCar-B.

Overview of the Included Adeept Software

The Adeept software supplied with the robot (see Figure 3-1) is primarily a client/server model in which the client is a control panel on another computer and the server runs on the Raspberry Pi on PiCar-B. The control panel allows you to control the robot remotely and has a lot of interesting features, such as object tracking using OpenCV and a radarlike ultrasonic mapping capability. You can also see the video coming from the robot and use that to navigate manually.

FIGURE 3-1: Adeept remote control software.

It is pretty complicated to install, however, so pay close attention to the instructions.

The software is definitely fun to use, but it does not require any real programming. The software is all open source, so you can look inside to see how they are doing things. Especially check out `server.py` under the server directory and look at the way they use threading to get smooth motion out of the robot.

Where to Go from Here?

You now have a small robot that can display quite complex behavior based on the ultrasonic sensor built in. You can add a lot to this robot in terms of adding sensors to the Raspberry Pi. (How about a laser distance finder? Bumper sensors? Light conditions?) Refer back to Book 6 and combine some of the motors and sensors you used there with this robot. Because we choose the PiCar-B, you can plug the Pi2Grover on top of the motor controller so you can use all the Grove devices you have accumulated.

The sky is the limit!

IN THIS CHAPTER

- Understanding the use of AI in robotics
- How AI helps in robotics
- Machine learning in our robot

Chapter 4
Using Artificial Intelligence in Robotics

"Artificial Intelligence (AI) is the theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages."

—DICTIONARY.COM

So, AI is meant to replace people? Well, not really. Modern AI looks to enhance machine intelligence at certain tasks that are normally done by people. Even saying the words "machine intelligence" is somewhat of a misnomer because it is hard to claim that machines have intelligence at all, at least as we think of it in people.

Instead of the philosophical debate, let's focus on how to use some modern AI techniques in a real robot example.

REMEMBER

For a better overview of AI and some of the philosophy involved, check out Book 4.

So, what AI technique can we use in our robotic car? Turns out there is a Pi camera on the car, and computer vision is really hard, so let's do something with that.

TIP

Making robots see is easy, but making them understand what they are seeing is exceptionally hard. If you want to study up on computer vision using Python, check out *Computer Vision Projects with OpenCV and Python 3* by Matthew Rever.

This Chapter's Project: Going to the Dogs

In this chapter, we show you how to build a machine-learning neural network and train it to recognize cats versus dogs. (This is a skill all robots should have.)

We will train the network using a 1,000-image subset of the 25,000 element database of pictures of cats and dogs from the Kaggle cats and dog database using TensorFlow.

TensorFlow is a Python package that is also designed to support neural networks based on matrices and flow graphs. It is similar to NumPy, but it differs in one major respect: TensorFlow is designed for use in machine learning and AI applications, and so it has libraries and functions designed for those applications.

TIP

If you need to, refer back to Book 4 as it has extensive information and examples about using TensorFlow in Python and on the Raspberry Pi.

Setting Up the Project

For the Windows, Linux, and the Raspberry Pi check out this official TensorFlow link: `https://www.tensorflow.org/install/pip`. Download TensorFlow and install according to the directions.

Download the truncated list of the Cats and Dogs database here: `https://github.com/switchdoclabs/CatsAndDogsTruncatedData`. It is about 65mb and is included with our software at dummies.com.

TIP

For more experimentation, download the full Cats and Dogs dataset from this link: `https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data`. Another source of the full data is: `https://www.microsoft.com/en-us/download/details.aspx?id=54765`.

Unzip the main folder and then unzip the `test.zip` and the `train.zip` subfolders. `Test` contains test images that have not been classified, and `train` contains our training data that we will use to train our neural network.

Run the following command in your program directory to download the data:

```
git clone https://github.com/switchdoclabs/CatsAndDogsTruncatedData.git
```

Now that we have our data ready, let's go train that network.

Machine Learning Using TensorFlow

Our goal in this section is to fully train our machine learning neural network on the difference between cats and dogs, validate the test data and then save the trained neural network so we can actually use it on our robot. Then the real fun will begin!

TIP

When you run the following program, if you see `ImportError: No module named 'seaborn'`, type `sudo pip3 install seaborn`.

We starting using a pretty simple two-layer neural network for our cats and dogs machine learning network. There are many more complex networks available and may give better results, but we were hoping this would be good enough for our needs.

There is also the option of using a much larger dataset (our training dataset has 1,000 cats and 1,000 dogs but over 25,000 images are available in the full dataset).

Using a simple two-layer neural network on the cats and dog dataset did not really work very well because we achieved only about a 51 percent detection rate (50 percent is as good as just guessing randomly) so we needed to go to a more complex neural network that works better on complex images.

You can use CNN (convolutional neural networks) in place of simple neural networks, data augmentation (increasing the training samples by rotating, shifting, and zooming that pictures) and a variety of other techniques that are beyond the scope of this book. We are going to use a pretty standard six-layer CNN instead of our simple neural network.

We changed the model layers in our program to use the following six-level convolutional layer model. You just have to love how easy Keras and TensorFlow makes it to dramatically change the neural network.

CONVOLUTIONAL NEURAL NETWORKS

CNNs work by scanning images and analyzing them chunk by chunk, say at a 5x5 window that moves by a stride length of two pixels each time until it spans the entire message. It's like looking at an image using a microscope; you see only a small part of the picture at any one time, but eventually you see the whole picture. Every time we loop through the data is called an *epoch*.

Going to a CNN network on a Raspberry Pi increased the single epoch time to 1,000 seconds from the 10-seconds epoch we had on the simple two-layer network. And it has a CPU utilization of 352 percent, which means it is using 3.5 cores on the Raspberry Pi, a machine that only has a total of 4. This amounts to a pretty high utilization of the Raspberry Pi 3B+. The little board is using almost 3.8W up from about 1.5W normally.

You can see the complexity of our new network by looking at the `model.summary()` results:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 75, 75, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 75, 75, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 37, 37, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 37, 37, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 18, 18, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 20736)             0
_________________________________________________________________
dense (Dense)                (None, 64)                1327168
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 130
=================================================================
Total params: 1,355,938
Trainable params: 1,355,938
Non-trainable params: 0
```

The code

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.examples.tutorials.mnist import input_data
from PIL import Image
from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.layers import *

# load data

img_width = 150
img_height = 150
train_data_dir = 'data/train'
valid_data_dir = 'data/validation'

datagen = ImageDataGenerator(rescale = 1./255)

train_generator = datagen.flow_from_directory(
                directory=train_data_dir,
                target_size=(img_width,img_height),
                classes=['dogs','cats'],
                class_mode='binary',
                batch_size=16)

validation_generator = datagen.flow_from_directory(directory=valid_data_dir,
                target_size=(img_width,img_height),
                classes=['dogs','cats'],
                class_mode='binary',
                batch_size=32)

# build model


model = tf.keras.Sequential()



model.add(Conv2D(32, (3, 3), input_shape=(150, 150, 3), padding='same',
   activation='relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(2, activation='softmax'))

model.compile(optimizer=tf.train.AdamOptimizer(),
                              loss='sparse_categorical_crossentropy',
                              metrics=['accuracy'])

print (model.summary())

# train model

print('starting training....')
history = model.fit_generator(generator=train_generator,
        steps_per_epoch=2048 // 16,epochs=20,
        validation_data=validation_generator,validation_steps=832//16)

print('training finished!!')


# save coefficients

model.save("CatsVersusDogs.trained")

# Get training and test loss histories
training_loss = history.history['loss']
accuracy = history.history['acc']
# Create count of the number of epochs
epoch_count = range(1, len(training_loss) + 1)

# Visualize loss history
plt.figure(0)
plt.plot(epoch_count, training_loss, 'r--')
plt.plot(epoch_count, accuracy, 'b--')
plt.legend(['Training Loss', 'Accuracy'])
plt.xlabel('Epoch')
plt.ylabel('History')
plt.grid(True)
plt.show(block=True);
```

Save the code into a file called `CatsVersusDogs.py`.

Examining the code

The code has 93 lines. It's pretty amazing what we can do with so few lines of code.

TIP

If you want to learn more about neural networks and machine learning, refer back to Book 4.

First, we import all the libraries:

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.examples.tutorials.mnist import input_data
from PIL import Image
from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.layers import *
```

Next, we massage the data into tensors (matrices) for training through our neural network. The cat and dog images are stored under separate directories under `data`:

```
# load data

img_width = 150
img_height = 150
train_data_dir = 'data/train'
valid_data_dir = 'data/validation'

datagen = ImageDataGenerator(rescale = 1./255)

train_generator = datagen.flow_from_directory(
                directory=train_data_dir,
                target_size=(img_width,img_height),
                classes=['dogs','cats'],
                class_mode='binary',
                batch_size=16)
```

```
validation_generator = datagen.flow_from_directory(directory=valid_data_dir,
                target_size=(img_width,img_height),
                classes=['dogs','cats'],
                class_mode='binary',
                batch_size=32)
```

Now we build the neural network that forms the basis of the machine-learning model. It is a six-layer neural network:

```
# build model


model = tf.keras.Sequential()
```

The first layer is a 2D convolutional neural network that starts to extract features from the photograph. We are using the RELU (rectified linear unit) as the neuron activation function. This is quite common:

```
model.add(Conv2D(32, (3, 3), input_shape=(150, 150, 3), padding='same',
   activation='relu'))
```

This next layer, `MaxPooling2D`, simplifies the features found in the previous layer:

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Next, another two layers of convolutional neural networks are added, each followed by a pooling layer:

```
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

One of the problems with neural networks is called "overfitting" — when the machine matches the data too closely such that the network won't match any new, slightly different pictures. Dropout layers help here by randomly setting input units to zero. We are removing 25 percent of the input units in this layer:

```
model.add(Dropout(0.25))
```

Now we flatten the data into a one-dimensional array and then use a final densely connected 64-neuron layer:

```
model.add(Flatten())
model.add(Dense(64, activation='relu'))
```

Next, drop out another 50 percent of the input units to help with overfitting:

```
model.add(Dropout(0.5))
```

And finally, our output layer "Cat or Dog":

```
model.add(Dense(2, activation='softmax'))

model.compile(optimizer=tf.train.AdamOptimizer(),
                              loss='sparse_categorical_crossentropy',
                              metrics=['accuracy'])

print (model.summary())

# train model
```

This training on 1,000 pictures takes about five hours on a Raspberry Pi 3B+:

```
print('starting training....')
history = model.fit_generator(generator=train_generator,
        steps_per_epoch=2048 // 16,epochs=20,
        validation_data=validation_generator,validation_steps=832//16)

print('training finished!!')
```

This next line of code saves the coefficients for use later. This saves five hours per run!

```
# save coefficients

model.save("CatsVersusDogs.trained")
```

Finally, we generate a graph using MatPlotLib that shows how the accuracy improves with each epoch:

```
# Get training and test loss histories
training_loss = history.history['loss']
accuracy = history.history['acc']
```

```
# Create count of the number of epochs
epoch_count = range(1, len(training_loss) + 1)

# Visualize loss history
plt.figure(0)
plt.plot(epoch_count, training_loss, 'r--')
plt.plot(epoch_count, accuracy, 'b--')
plt.legend(['Training Loss', 'Accuracy'])
plt.xlabel('Epoch')
plt.ylabel('History')
plt.grid(True)
plt.show(block=True);
```

The results

WARNING

Install the `h5py` library before running this program. Otherwise the `save` statement will not work.

```
sudo apt-get install python-h5py
```

Showtime! Run the following command in your terminal window:

```
python3 CatsVersusDogs.py
```

It takes about five hours on a Raspberry Pi 3B+ to generate 20 epochs of training and save the results into our `CatsVersusDogs.training` file. A snapshot of the last epoch is as follows:

```
Epoch 20/20
128/128 [==============================] - 894s 7s/step - loss: 0.0996 - acc:
   0.9609 - val_loss: 1.1069 - val_acc: 0.7356
training finished!!
```

TIP

You can safely ignore warnings from TensorFlow, such as:

```
/usr/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: compiletime
   version 3.4 of module 'tensorflow.python.framework.fast_tensor_util' does not
   match runtime version 3.5
  return f(*args, **kwds)
/usr/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: builtins.type
   size changed, may indicate binary incompatibility. Expected 432, got 412
  return f(*args, **kwds)
```

This will be fixed in an upcoming version.

After the five-hour-long training, we achieved an accuracy of 96 percent! Pretty good. Figure 4-1 shows how the accuracy improved during training.

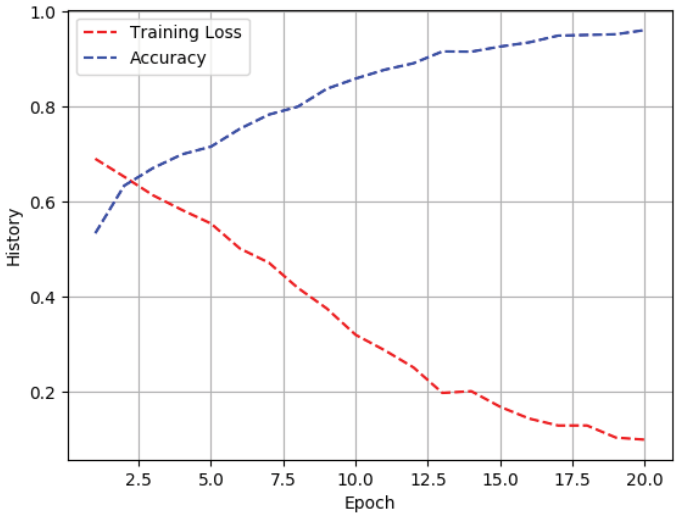

FIGURE 4-1: Cats and dogs recognition accuracy per epoch.

By the shape of the curve, we might have been able to get a little better by running more epochs, but this is good enough for our example.

Testing the Trained Network

Time to do a test on an external cat/dog image.

We're going to use the trained data from our neural network training session above to do a couple of predictions on new pictures to the neural network.

We chose the cat picture (see Figure 4-2) because it is a low-contrast picture, whereas the dog is pretty much a standard dog picture (see Figure 4-3).

FIGURE 4-2: Panther the Cat on salmon.

FIGURE 4-3: Winston the Dog.

The code

```
#import libraries
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
from PIL import Image
```

```
print("import complete")
# load model

img_width = 150
img_height = 150

class_names = ["Dog", "Cat"]

model = tf.keras.models.load_model(
     "CatsVersusDogs.trained",compile=True)
print (model.summary())

# do cat single image
imageName = "Cat150x150.jpeg"
testImg = Image.open(imageName)
testImg.load()
data = np.asarray( testImg, dtype="float" )

data = np.expand_dims(data, axis=0)
singlePrediction = model.predict(data, steps=1)

NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)
print(NumberElement)
print(Element)
print(singlePrediction)

print ("Our Network has concluded that the file '"
        +imageName+"' is a "+class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")

# do dog single image
imageName = "Dog150x150.JPG"
testImg = Image.open(imageName)
testImg.load()
data = np.asarray( testImg, dtype="float" )

data = np.expand_dims(data, axis=0)
singlePrediction = model.predict(data, steps=1)

NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)
print(NumberElement)
print(Element)
print(singlePrediction)

print ("Our Network has concluded that the file '"
        +imageName+"' is a "+class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")
```

Explaining the code

This code uses the training that we generated by training with the cats and dogs dataset earlier in this chapter.

First, we import our libraries:

```
#import libraries
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
from PIL import Image

print("import complete")
# load model

img_width = 150
img_height = 150
```

We set up the class name array so we have classification names for our images:

```
class_names = ["Dog", "Cat"]
```

Here is the place where we load the training data that we generated earlier for the neural network machine learning model:

```
model = tf.keras.models.load_model(
       "CatsVersusDogs.trained",compile=True)
print (model.summary())
```

Now, we test a single cat image:

```
# do cat single image
imageName = "Cat150x150.jpeg"
testImg = Image.open(imageName)
testImg.load()
```

Convert to a NumPy tensor:

```
data = np.asarray( testImg, dtype="float" )
```

Expand the dimension, since this function looks for an array of images:

```
data = np.expand_dims(data, axis=0)
```

Now, we do the predication based on our image:

```
singlePrediction = model.predict(data, steps=1)
```

We print out the raw data:

```
NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)
print(NumberElement)
print(Element)
```

```
print(singlePrediction)
```

Interpret the prediction:

```
print ("Our Network has concluded that the file '"
       +imageName+"' is a "+class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")
```

Next, we do the same with a single dog image:

```
# do dog single image
imageName = "Dog150x150.JPG"
testImg = Image.open(imageName)
testImg.load()
data = np.asarray( testImg, dtype="float" )

data = np.expand_dims(data, axis=0)
singlePrediction = model.predict(data, steps=1)

NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)
print(NumberElement)
print(Element)
print(singlePrediction)

print ("Our Network has concluded that the file '"
       +imageName+"' is a "+class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")
```

The results

Save the code into `singleTestImage.py` and run using `sudo python3 singleTestImage.py`.

Here are the results:

```
import complete
WARNING:tensorflow:No training configuration found in save file: the model
   was *not* compiled. Compile it manually.
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 75, 75, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 75, 75, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 37, 37, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 37, 37, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 18, 18, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 20736)             0
_________________________________________________________________
dense (Dense)                (None, 64)                1327168
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 130
=================================================================
Total params: 1,355,938
Trainable params: 1,355,938
Non-trainable params: 0
_________________________________________________________________
None
1
1.0
[[ 0.  1.]]
Our Network has concluded that the file 'Cat150x150.jpeg' is a Cat
100% Confidence Level
0
1.0
[[ 1.  0.]]
Our Network has concluded that the file 'Dog150x150.JPG' is a Dog
100% Confidence Level
```

Well, this network worked very well. It identified both the cat and dog as their respective species. See the 100 percent confidence intervals? It is actually like

99.99 percent or something like that and is just rounded to 100 percent by the formatting.

Now that we have the trained model and have tested it with some real images, it is time to put it into our robot and use the Pi camera for some dog and cat investigation.

WARNING

A real limitation of the way we built this neural network is that it really is only looking at images of cats and dogs and determining whether the image is a cat or dog. However, the network classifies *everything* as either a cat or dog. If we were to build a more comprehensive network, we would have to train it to differentiate between a cat, a dog, or a dress, for example. (See Figure 4-4.) We ran one more test of the network using the much maligned dress picture from Book 4.

FIGURE 4-4: A picture of a dress?

And, as expected, the network got it wrong:

```
Our Network has concluded that the file 'Dress150x150.JPG' is a Cat
```

There is a lot more to the practice of teaching a machine to learn in a general manner.

TIP

If you get an error such as: `undefined symbol: cblas_sgemm` from your `import numpy`, try running your program with `sudo`.

Taking Cats and Dogs to Our Robot

Time to add a new experience to last chapter's robot. We are going to install the trained cats and dogs neural network on the PiCar-B robot and use the onboard LEDs to display whether the onboard Pi camera is looking at a cat or a dog.

As mentioned earlier, our neural network classifies just about everything as a cat or a dog. So, what we are going to do is trigger the neural network classification when the ultrasonic sensor changes, such as when a dog or cat may walk in front of the robot.

Our test setup, because John's cat would not cooperate, is shown in Figure 4-5. It has the robot staring at the screen (triggering the ultrasonic sensor) showing a PowerPoint presentation of various cats and dogs.

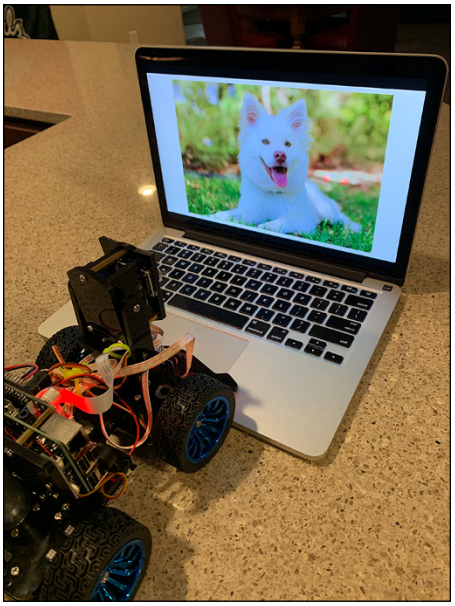

FIGURE 4-5: Robot vision neural network test setup.

The code

```
#!/usr/bin/python3
#using a neural network with the Robot

#import libraries
import numpy as np
import tensorflow as tf
```

```
from tensorflow.python.framework import ops
from PIL import Image


import RobotInterface
import time
import picamera


print("import complete")
RI = RobotInterface.RobotInterface()

# load neural network model

img_width = 150
img_height = 150


class_names = ["Dog", "Cat"]
model = tf.keras.models.load_model("CatsVersusDogs.trained",compile=True)



RI.centerAllServos()
RI.allLEDSOff()

# Ignore distances greater than one meter
DISTANCE_TO_IGNORE = 1000.0
# How many times before the robot gives up
DETECT_DISTANCE = 60

def bothFrontLEDSOn(color):
    RI.allLEDSOff()
    if (color == "RED"):
        RI.set_Front_LED_On(RI.right_R)
        RI.set_Front_LED_On(RI.left_R)
        return
    if (color == "GREEN"):
        RI.set_Front_LED_On(RI.right_G)
        RI.set_Front_LED_On(RI.left_G)
        return
    if (color == "BLUE"):
        RI.set_Front_LED_On(RI.right_B)
        RI.set_Front_LED_On(RI.left_B)
        return
```

```
def checkImageForCat(testImg):

    # check dog single image
    data = np.asarray( testImg, dtype="float" )

    data = np.expand_dims(data, axis=0)
    singlePrediction = model.predict(data, steps=1)

    print ("single Prediction =", singlePrediction)
    NumberElement = singlePrediction.argmax()
    Element = np.amax(singlePrediction)

    print ("Our Network has concluded that the file '"
        +imageName+"' is a "+class_names[NumberElement])

    return class_names[NumberElement]

try:
    print("starting sensing")
    Quit = False
    trigger_count = 0
    bothFrontLEDSOn("RED")

    #RI.headTiltPercent(70)
    camera = picamera.PiCamera()
    camera.resolution = (1024, 1024)
    camera.start_preview(fullscreen=False,
           window=(150,150,100,100))
    # Camera warm-up time
    time.sleep(2)

    while (Quit == False):

        current_distance = RI.fetchUltraDistance()
        print ("current_distance = ", current_distance)
        if (current_distance < DETECT_DISTANCE):
            trigger_count = trigger_count + 1
            print("classifying image")


            camera.capture('FrontView.jpg')
            imageName = "FrontView.jpg"
            testImg = Image.open(imageName)
            new_image = testImg.resize((150, 150))
            new_image.save("FrontView150x150.jpg")
```

```
            if (checkImageForCat(new_image) == "Cat"):
                bothFrontLEDSOn("GREEN")
            else:
                bothFrontLEDSOn("BLUE")


            time.sleep(2.0)
            bothFrontLEDSOn("RED")
            time.sleep(7.0)

except KeyboardInterrupt:
        print("program interrupted")

print ("program finished")
```


How it works

Most of the preceding code is pretty straightforward and similar to the robot brain software earlier in the chapter. One part does deserve to be talked about, however, and that is our classifier function:

```
def checkImageForCat(testImg):

    # check dog single image
    data = np.asarray( testImg, dtype="float" )

    data = np.expand_dims(data, axis=0)
    singlePrediction = model.predict(data, steps=1)

    print ("single Prediction =", singlePrediction)
    NumberElement = singlePrediction.argmax()
    Element = np.amax(singlePrediction)

    print ("Our Network has concluded that the file '"
        +imageName+"' is a "+class_names[NumberElement])

    return class_names[NumberElement]
```

This function takes the incoming test vision (taken from the Pi camera and then resized into an 150 x 150 pixel image — the format required by our neural network).

The results

Save the program into a file called `robotVision.py` and run the program — `sudo python3 robotVision.py` — to get your machine to start looking for cats. You

really should run this program on the Raspberry Pi GUI so you can see the small camera preview on the screen.

Here is some results from our test setup in Figure 4-5:

```
current_distance =  20.05481719970703
classifying image
single Prediction = [[ 0.  1.]]
Our Network has concluded that the file 'FrontView.jpg' is a Cat
100.00% Confidence Level
current_distance =  20.038604736328125
classifying image
single Prediction = [[ 1.  0.]]
Our Network has concluded that the file 'FrontView.jpg' is a Dog
100.00% Confidence Level
current_distance =  19.977807998657227
```

Overall, the results were pretty good. We found variations in recognition due to lighting, which wasn't a big surprise. However, the network consistently identified one picture as a dog that was actually a cat. (See Figure 4-6.)

We suspect it is the folded ears, but one of the really hard things about figuring out what is wrong with a neural network machine learning model like this is there is no way to really know how the machine made its decision. Looking at the network you can look at 1.3 *million* parameters and weights and you can't tell where it is going wrong. After a while, you get a feel for what works and what doesn't in machine learning, but as for figuring out what this network is doing exactly wrong, you are just out of luck.

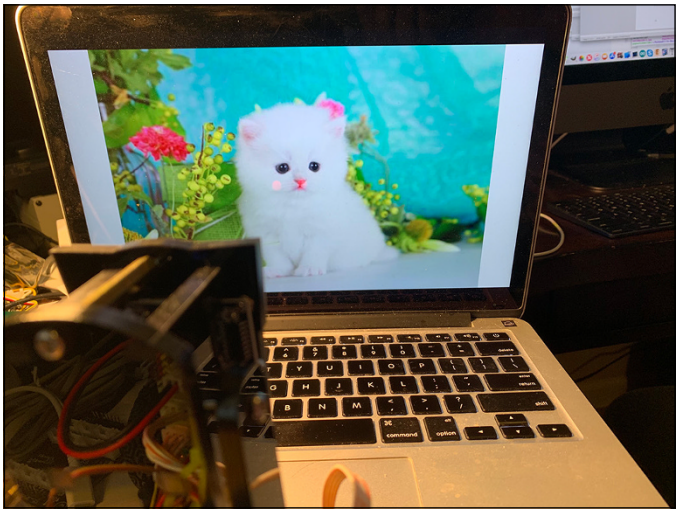

FIGURE 4-6: The cat who is apparently a dog.

We have a nefarious plan here for a variant of this neural network in the future. We're going to use it on a new project, the Raspberry Pi based MouseAir, that will launch toy mice when the camera spots a cat but not when it spots a dog. You can see the Pi camera at the top-right corner of the Figure 4-7. That should definitely be fun.

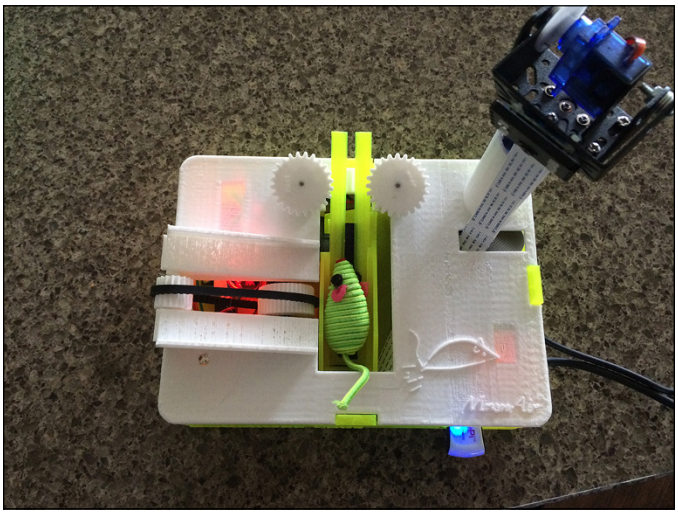

FIGURE 4-7: MouseAir, an AI mouse-launching cat toy.

Other Things You Can Do with AI Techniques and the Robot

As part of an advanced Python brain on the PiCar-B, there are a number of things you could do with neural networks and other AI techniques. Here are a few examples:

Cat/Not Cat

Retrain our Cat/Dog network to focus only on cats; it lumps everything else into the "Not Cat" category. You can kind of do that with the current network because this training set tends to somewhat classify anything that doesn't look like a cat as a dog. It doesn't do that all the time though. Remember the dress picture in Figure 4-4 (classified as a cat by our network)?

Santa/Not Santa

Take the streaming video from our Raspberry Pi and grab a frame once in a while and check with a neural network (trained on Santa images) to determine whether Santa is here for Christmas. Impossible! No, wait. It's been done in a very cool series of articles here: `https://www.pyimagesearch.com/2017/12/18/keras-deep-learning-raspberry-pi/`.

Follow the ball

You can program the Raspberry Pi using a library called OpenCV (Open-Source Computer Vision) to detect a ball in your camera stream and give you the image coordinates. Then you can drive towards it. You can swivel the head and change the tilt to look for it too. A version of this project is part of the Adeept software included with the PiCar-B. You can see the source code inside of `server.py` under the server directory.

Using Alexa to control your robot

You can use the power of the Amazon Alexa program to order your robot to do things. If you'd like to see how to connect up your Raspberry Pi to Alexa to do just that kind of project, check out the ebook *Voice Time: Connecting Your Raspberry Pi to the Amazon Alexa* by John Shovic.

AI and the Future of Robotics

In our opinion, artificial intelligence is the future of robotics. As the hardware becomes less and less expensive and the software techniques more easily accessible, there will be more and more uses for robots, not only manufacturing but also in the home and on the road.

We expect to see new AI in many new consumer products in the coming years. The only question we have is "How long before we have AI in my toaster?" We'll buy one when they come out.

Index

Special Characters

Numbers

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

John Shovic has been working with software and electronics since he talked his school into letting him use their IBM 1130 computer for the whole summer of 1973. That really launched him into his lifelong love affair with software. Dr. Shovic has founded multiple companies: Advance Hardware Architectures; TriGeo Network Security; Blue Water Technologies; InstiComm, LLC; and bankCDA. He has also served as a professor of computer science at Eastern Washington University, Washington State University, and the University of Idaho. Dr. Shovic has given over 70 invited talks and has published over 50 papers on a variety of topics on Arduinos, Raspberry Pis, iBeacons, HIPAA, GLB, computer security, computer forensics, embedded systems, and others. Currently John is proud to be serving as a computer science faculty member, specializing in robotics and artificial intelligence, at the University of Idaho, Coeur d'Alene, Idaho, and is surrounded by a bunch of students that are as excited about technology and computers as he is.

Alan Simpson is the author of over 100 computer books on database, programming, and web development. His books have been published throughout the world in over a dozen languages, and have sold millions of copies. Alan left the writing world a few years ago to get out of the ivory tower and into the real working world, first as a developer, and now as a manager, of the Apps and DBA team in his county government's IT department. Alan has been called a "master communicator" throughout his extensive career, and his online courses and YouTube videos continue to get rave reviews from his many students and followers.

Author's Acknowledgments

John Shovic: I would like thank the wonderful staff at Wiley and give full credit to my co-author, Alan. Thanks to my literary agent Carol Jelen for encouraging me to pursue writing this book. I would like to specifically thank the University of Idaho for their support and suggestions, especially to Dr. Bob Rinker and Dean Larry Stauffer. Great people, great university. Also, no book like this would be complete without my thanking my fabulous students, especially Amanda, Adrian, and Doug, who inspire me every day and who had to listen to the chapters of this book as I wrote them. And of course, my cat Panther, who spent most of this book curled up on my lap.

Alan Simpson: Many thanks to Steve Hayes and everyone else at Wiley for offering me this great opportunity. Thanks to Christopher Morris, my intrepid editor. Thanks to my literary agents Carol Jelen and Margot Maley Hutchinson of Waterside Productions. And thanks to my wife Susan for her patience while I was working at all kinds of crazy hours.

Dedication

John Shovic: To my wife, Laurie. She has supported me in so many ways including making sure my socks match in the morning. Thank you!

Alan Simpson: To Susan, Ashley, and Alec.

Publisher's Acknowledgments

Executive Editor: Steve Hayes
Project Editor: Christopher Morris
Copy Editor: Christopher Morris
Technical Editor: Russ Mullen
Production Editor: Vasanth Koilraj
Cover Image: © matejmo/iStock.com