4
MAKING YOUR ROBOT MOVE

AT THIS STAGE, YOU HAVE A SWEET-LOOKING RASPBERRY PI ROBOT THAT DOESN'T DO ANYTHING... YET! TO UNLOCK THE CAPABILITIES OF ALL THE HARDWARE YOU JUST WIRED UP, YOU'LL HAVE TO SINK YOUR TEETH INTO SOME MORE PROGRAMMING.

www.itbook.store/books/9781593279202

In this chapter, I'll show you how to use the Python programming language to make your robot move. We'll cover basic movement, making your robot remote-controlled, and varying its motor speed.

THE PARTS LIST

Most of this chapter will be about coding the robot, but to enable remote control you'll need a couple of parts later:

- Nintendo Wii remote
- Bluetooth dongle if you're using a Pi older than a Model 3 or Zero W

UNDERSTANDING THE H-BRIDGE

Most single-motor controllers are based around an electronics concept called an *H-bridge*. The L293D motor driver chip we're using contains two H-bridges, permitting you to control the two motors of your robot through a single chip.

An H-bridge is an electronic circuit that allows a voltage to be applied across a load, usually a motor, in either direction. For the purposes of robotics, this means that an H-bridge circuit can drive a motor both *forward* and *backward*.

A single H-bridge is made of four electronic switches, built from transistors, arranged like S1, S2, S3, and S4 in Figure 4-1. By manipulating these electronic switches, an H-bridge controls the forward and backward voltage flow of a single motor.

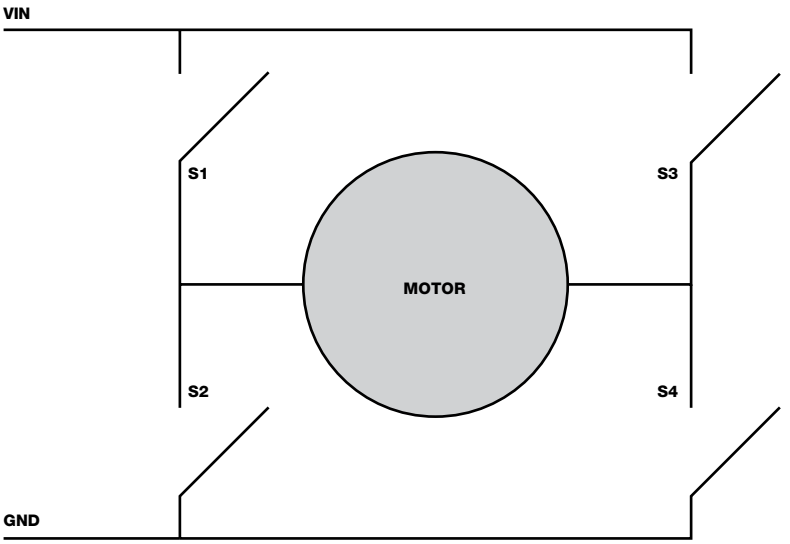

FIGURE 4-1
A single H-bridge circuit

www.itbook.store/books/9781593279202

When all the switches are open, no voltage is applied to the motor and it doesn't move. When only S1 and S4 are closed, there is a flow of current in one direction through the motor, making it spin. When only S3 and S2 are closed, a current flows in the opposite direction, making the motor spin the other way.

The design of the L293D means that we can't close S1 and S2 at the same time. This is fortunate, as doing so would short-circuit the power, causing damage! The same is true of S3 and S4.

The L293D abstracts this one step further and requires only two inputs for one motor (four inputs for a pair of motors, like you wired up in Chapter 3). The behavior of the motor depends on which inputs are high and which are low (1 or 0, respectively). Table 4-1 summarizes the different input options for the control of one motor.

INPUT 1	INPUT 2	MOTOR BEHAVIOR
0	0	Motor off
0	1	Motor rotates in one direction
1	0	Motor rotates in other direction
1	1	Motor off

TABLE 4-1
Motor Behavior Based on Inputs

We'll use the GPIO Zero Python library to interface with the Pi's GPIO pins and motor controller. There are several functions in the library for controlling basic movement, so you won't have to worry about turning specific GPIO pins on and off yourself.

FIRST MOVEMENT

Now for the most exciting step of your robotics journey yet: moving your robot! You'll eventually make your robot entirely remote-controlled and even able to follow your instructions, but before that let's master some basic motor functionality. You'll start by programming your robot to move along a predefined route.

Programming Your Robot with a Predefined Route

Boot up your Raspberry Pi on your robot and log in over SSH. While your robot is stationary and being programmed, it is best to disconnect your batteries and power your Pi from a micro USB cable connected to a wall outlet. This will save your batteries for when they are really needed.

www.itbook.store/books/9781593279202

From the terminal, navigate from your home directory into the folder you are using to store your code. For me, I'll navigate into my *robot* projects folder like so:

```
pi@raspberrypi:~ $ cd robot
```

Next, create a new Python program and edit it in the Nano text editor with the following command; I have called my program *first_move.py*:

```
pi@raspberrypi:~/robot $ nano first_move.py
```

Now you need to come up with a predefined route to program! With the DC motors we're using, you *can't* rotate them a specific distance or number of steps, but you *can* power them on and off for a certain amount of time. This means that any path will be a rough approximation of where you want your robot to go rather than a precise plan.

To start, let's keep things simple and make your robot drive around in a square, with a route like the one shown in Figure 4-2.

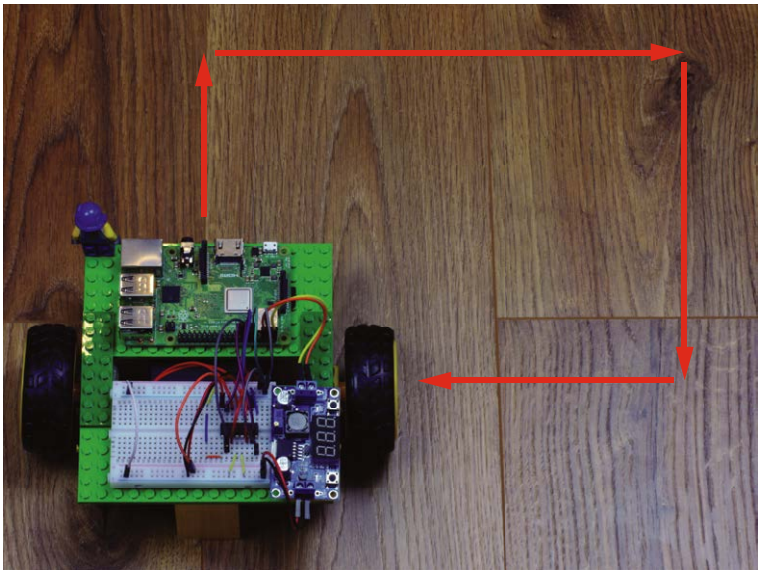

FIGURE 4-2
The robot's planned route

www.itbook.store/books/9781593279202

In your *first_move.py* file, enter the code in Listing 4-1 to program a square route.

LISTING 4-1
Programming your robot to move in a square

```
import gpiozero
import time

❶ robot = gpiozero.Robot(left=(17,18), right=(27,22))

❷ for i in range(4):
    ❸ robot.forward()
    ❹ time.sleep(0.5)
    ❺ robot.right()
    ❻ time.sleep(0.25)
```

The program starts by importing familiar Python libraries: `gpiozero` and `time`. Then you create a variable called `robot` ❶, to which you assign a `Robot` object from the GPIO Zero library.

Objects in Python are a way of holding variables (pieces of information) and functions (predefined sets of instructions that perform tasks) in a single entity. This means that when we assign an object to a variable, that variable then has a range of predefined things that it knows and can do. An object gets these capabilities from its *class*. Each class has its own functions (called *methods*) and variables (called *attributes*). These are advanced features of Python and you don't have to worry about them too much at this stage. Just know that we're using some predefined classes from Python libraries, like GPIO Zero, to make it easier for us.

The GPIO Zero library has an inbuilt `Robot` class that features a variety of functions for moving a two-wheeled robot in different directions. Notice the two sets of values in the parentheses assigned to `left` and `right` ❶. These represent the input pins of the L293D you have wired up. If you followed my exact wiring from Chapter 3, then the four GPIO pins should be: 17, 18 and 27, 22.

This program also uses a new type of loop called a `for` loop ❷. In Chapter 2, while making LEDs flash on and off and getting inputs from buttons, you used a `while` loop. A `while` loop keeps repeating its contents indefinitely while a certain condition is met, but a `for` loop repeats a block of code a *fixed* number of times. The syntax of this loop, `for i in range(4):`, means "do the following four times."

The `for` loop commands your robot to start going forward ❸ and then wait for half a second ❹ to allow some time for the robot to move. The result is that both motors move in a single direction (forward) for half a second.

www.itbook.store/books/9781593279202

You then instruct your robot to turn right ❺ and wait for a quarter of a second as this happens ❻. By telling the robot to turn right, you replace the forward command issued half a second ago with a new command for the motors.

Once this has been executed once, the "go forward, then turn right" process starts again and continues for a total of *four* times. You are trying to make your robot go in a square, and squares have four sides, hence the specific repetition.

Once you've finished writing your program, exit Nano by pressing CTRL-X and save your work like usual. Next, we'll run the program to make the robot move!

The GPIO Zero Robot class has commands for all directions and basic functionality, summarized in Table 4-2.

TABLE 4-2
The `Robot` Class Commands

COMMAND	FUNCTIONALITY
`robot.forward()`	Run both motors forward.
`robot.backward()`	Run both motors backward.
`robot.left()`	Run the right motor forward and the left motor backward.
`robot.right()`	Run the left motor forward and the right motor backward.
`robot.reverse()`	Reverse the robot's current motor directions. For example: if going forward, go backward. If going left, go right. This is *not* the same as going backward!
`robot.stop()`	Stop both motors.

Running Your Program: Make Your Robot Move

Before you execute your program, ensure your robot is disconnected from the wall power outlet and the batteries are connected and turned on. You should also place your robot on a relatively large, flat surface clear of obstacles and hazards. Rough surfaces, like carpets, may cause your robot to become stuck or struggle to move. Try to avoid this, as struggling motors draw more current, and when their movement is completely blocked (or *stalled*) you might even damage your electronics! The flatter the surface, the better your robot will run.

www.itbook.store/books/9781593279202

It is also a good idea to be in a position to "catch" your robot in case either it or something/someone is in peril. It may try to go down the stairs, for example, or the cat may be in the way.

To run your program, wirelessly access the terminal of your Pi using SSH and enter:

```
pi@raspberrypi:~/robot $ python3 first_move.py
```

Your robot should burst into life and start to move. If all has gone well, it will move on a square-based path and then come to a stop, and your program will end by itself. If you need to stop your robot at any point, press CTRL-C on your keyboard to kill the motors immediately.

TROUBLESHOOTING GUIDE: ROBOT NOT WORKING PROPERLY?

If your robot isn't functioning as it should be, don't worry. Usually malfunctions fall into some common categories and should be easy to fix! The following quick guide will help you resolve most issues you might have.

Robot Moving Erratically

The most common problem after you execute the *first_move.py* program is that your robot moves, but not in the right pattern. Instead of going forward, it goes backward; or instead of turning right, it turns left. You may even find that it just spins on the spot!

This behavior can be easily fixed. As we discussed, DC motors have two terminals with no particular polarity. This means that if you change the direction of current flowing through the motor, the motor spins the other way. Consequently, if one or both of your motors is going in the opposite direction of your commands, you can swap the wires connected to the output pins of your motor controller to reverse this. For example, swap the wires connected to Output 1 with Output 2 of your L293D. Refer to Chapter 3 for guidance and relevant diagrams.

(continued)

www.itbook.store/books/9781593279202

Motors Not Moving

If your program successfully executes, but your robot's wheels don't move or only one motor starts to move, then you could have an issue related to your wiring. Go back to the previous chapter and check that you've connected everything as per the instructions. Ensure the connections to the motors are solid and that none of the wires have become loose. If you're convinced that you've wired everything correctly, check whether your batteries are charged and that they can provide enough power for your specific motors.

If your Raspberry Pi crashes when the motors start to turn, you most likely have a power issue. Check how you have set up your buck converter. If you are using a different converter than mine, you may run into problems. Go back a chapter for guidance and recommendations.

Robot Moving Very Slowly

A slow robot is usually a sign that not enough power is being provided to the motors. Check the voltage requirements of your motors and make sure you're supplying them with what they need. Often motors will accept a range of voltages—for example, from 3 V to 9 V. If your motors do, try a higher voltage that stays within the recommended range. Bear in mind that if you change your batteries and any of the voltages, you'll need to check and reset your buck converter to ensure that you don't feed more than 5.1 V into your Raspberry Pi.

Alternatively, the motors themselves may just have a slow, geared RPM. If that's the case, while your robot may be slow, it will probably have a lot of torque, which is a fair trade-off.

Robot Not Following the Programmed Path

If your robot successfully executes the program and starts to move at a suitable speed, but doesn't follow the exact path you had planned, don't fret! Every motor is different and will need adjustments for the program to work the way you want. For example, 0.25 seconds may not be enough time for the motors to make your robot turn approximately 90 degrees. Edit the program and play around with the `sleep()` and `robot()` statements inside the `for` loop to adjust.

www.itbook.store/books/9781593279202

MAKING YOUR ROBOT REMOTE-CONTROLLED

Making a robot come to life and move is an exciting first step in robotics, and the natural next step is to make your robot remote-controlled. This means it will no longer be limited to a predefined path, so you'll be able to control it in real time!

The aim of this project is to program your robot so you can use a wireless controller to guide it. You'll be able to instantaneously change your robot's movements without going back into your code.

The Wiimote Wireless Controller

In order to control your robot with a wireless controller, first you'll need one! The perfect remote for our robot is a Nintendo Wii remote, also known as a *Wiimote*, like the one in Figure 4-3.

FIGURE 4-3
My much-loved Nintendo Wiimote

A Wiimote is a pretty nifty little Bluetooth controller with a set of buttons and some sensors that are able to detect movement. The Wiimote was originally created for the Nintendo Wii games console, but fortunately there's an open source Python library, called `cwiid`, that allows Linux computers, like your Raspberry Pi, to connect and communicate with Wiimotes. We'll use `cwiid` to manipulate the data from a Wiimote to control your robot's motors.

If you don't have a Wiimote already, you'll need to get your hands on one. These are widely available online, both new and used. I recommend picking up a cheap used one on a site like eBay or from a secondhand shop—mine cost me less than $15.

WARNING

To guarantee compatibility with your Raspberry Pi, make sure that your Wiimote is a Nintendo-branded official model. Over the years a considerable number of third-party Wiimotes have become available to buy. Though usually cheaper than an official Wiimote, these aren't guaranteed to work with the `cwiid` library.

www.itbook.store/books/9781593279202

You'll use Bluetooth to pair your Wiimote with the Raspberry Pi on your robot. *Bluetooth* is a wireless radio technology that many modern devices, like smartphones, use to communicate and transfer data over short distances. The latest Raspberry Pi models, like the Pi Zero W and Raspberry Pi 3 Model B+, come with Bluetooth capabilities built in. All models prior to the Raspberry Pi 3 Model B, like the original Raspberry Pi and Pi 2, do *not*, and consequently you'll need to get a Bluetooth USB adapter (or *dongle*), like the one pictured in Figure 4-4, to connect to a Wiimote.

FIGURE 4-4
A $3 Raspberry Pi–compatible Bluetooth dongle

These are available for less than $5 online; just search for "Raspberry Pi compatible Bluetooth dongle." Before you proceed, make sure you have plugged the dongle into one of the USB ports of your Pi.

Installing and Enabling Bluetooth

Before you start to write the next Python script, you'll need to make sure that Bluetooth is installed on your Pi and that the `cwiid` library is set up. Power your Raspberry Pi from a wall outlet and then, from the terminal, run this command:

```
pi@raspberrypi:~ $ sudo apt-get update
```

And then run this one:

```
pi@raspberrypi:~ $ sudo apt-get install bluetooth
```

If you have Bluetooth installed already, you should see a dialogue that states `bluetooth is already the newest version`. If you don't get this message, go through the Bluetooth installation process.

www.itbook.store/books/9781593279202

Next, you'll need to download and install the `cwiid` library for Python 3. We'll grab this code from *GitHub*, a website where programmers and developers share their software.

Run the following command in the home folder of your Pi:

```
pi@raspberrypi:~ $ git clone https://github.com/azzra/
python3-wiimote
```

You should now have the source code of the `cwiid` library downloaded to your Raspberry Pi, stored in a new folder called *python3-wiimote*. Before we can get to our next Python program, the source code must first be *compiled*, a process that makes and readies software for use on a device.

You also need to install four other software packages before you can proceed. Enter the following command to install all four at once:

```
pi@raspberrypi:~ $ sudo apt-get install bison flex automake
libbluetooth-dev
```

If you're prompted to agree to continue, press Y (which is the default). Once this command has finished executing, change into the newly downloaded directory containing your Wiimote source code:

```
pi@raspberrypi:~ $ cd python3-wiimote
```

Next, you must prepare to compile the library by entering each of the following commands, one after the other. This is all part of the compilation process—you don't have to worry about the specifics of each command! The first two commands won't output anything, but the rest of them will. I'll show the start of each output here:

```
pi@raspberrypi:~/python3-wiimote $ aclocal
```

```
pi@raspberrypi:~/python3-wiimote $ autoconf
```

```
pi@raspberrypi:~/python3-wiimote $ ./configure
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
--snip--
```


www.itbook.store/books/9781593279202

```
pi@raspberrypi:~/python3-wiimote $ make
make  -C libcwiid
make[1]: Entering directory '/home/pi/python3-wiimote/libcwiid'
--snip--
```

And then finally, to install the `cwiid` library, enter:

```
pi@raspberrypi:~/python3-wiimote $ sudo make install
make install -C libcwiid
make[1]: Entering directory '/home/pi/python3-wiimote/libcwiid'
install -D cwiid.h /usr/local/include/cwiid.h
--snip--
```

After that, `cwiid` should work in Python 3! Now you can navigate out of the *python3-wiimote* directory and back to where you have all of your other code.

NOTE

If you have trouble with the Python 3 `cwiid` installation, check out the book's website to see whether the process has been updated: https://nostarch.com/raspirobots/.

Programming Remote Control Functionality

Now create and open a new Python program to store the Wiimote code. I have called mine *remote_control.py*:

```
pi@raspberrypi:~/robot $ nano remote_control.py
```

In general, before you start to code, it is important to first plan what exactly you want to do. In our case, we want to think about how we want the Wiimote to control the robot exactly. Let's make a plan.

The Wiimote has 11 digital buttons, which is more than we'll need for this simple project. Interestingly for us, 4 of those buttons belong to the D-pad—the four-way directional control buttons at the top of your Wiimote, shown in Figure 4-5.

FIGURE 4-5
The four-way D-pad of a Wiimote

www.itbook.store/books/9781593279202

That's perfect for our purposes: we can use up to make the robot go forward, right to make the robot go right, down to make the robot go backward, and left to make the robot go left. This is very similar to the program we wrote earlier, except that now we read our inputs from the Wiimote rather than them being programmed in.

We also need something to make the robot stop. The "B" trigger button on the underside of the Wiimote would be well suited to this. Let's write some code in Nano that executes the plan we've made; see Listing 4-2. I have saved this program as *remote_control.py*.

LISTING 4-2
Programming your robot to respond to the D-pad of your Wiimote

```
import gpiozero
import cwiid

❶ robot = gpiozero.Robot(left=(17,18), right=(27,22))

print("Press and hold the 1+2 buttons on your Wiimote simultaneously")
❷ wii = cwiid.Wiimote()
print("Connection established")
❸ wii.rpt_mode = cwiid.RPT_BTN

while True:
 ❹ buttons = wii.state["buttons"]

 ❺ if (buttons & cwiid.BTN_LEFT):
        robot.left()
    if (buttons & cwiid.BTN_RIGHT):
        robot.right()
    if (buttons & cwiid.BTN_UP):
        robot.forward()
    if (buttons & cwiid.BTN_DOWN):
        robot.backward()
    if (buttons & cwiid.BTN_B):
        robot.stop()
```

As before, you start by importing `gpiozero` as well as the new `cwiid` library. A `Robot` object is then set up ❶.

In the next section of code ❷, you set up the Wiimote. As with the `Robot` object, we assign the `Wiimote` object to a variable called `wii`. When this code runs and execution reaches this line, there will be a pairing handshake between the Raspberry Pi and Wiimote. The user will need to *press and hold* buttons 1 and 2 on the Wiimote at the same time to put the Wiimote in a Bluetooth-discoverable mode. We add a `print()` statement here to tell the user when to press the buttons.

If the pairing is successful, the code prints a positive message for the user. We then turn on the Wiimote's reporting mode ❸,

www.itbook.store/books/9781593279202

which permits Python to read the values of the different buttons and functions.

After this, we use an infinite `while` loop to tell the robot what to do when each button is pressed. First, the loop reads the current status of the Wiimote ❹, meaning it checks what buttons have been pressed. This information is then stored in a variable called `buttons`.

Finally, we start the last chunk of the program ❺: a variety of `if` statements and conditions that allocate an action to each button. To look at one example, the first `if` statement ensures that if the left button of the D-pad has been pressed, the robot is instructed to turn left. Over the next lines, the same sort of logic is applied: if the right button of the D-pad has been pressed, the robot is instructed to turn right, and so on.

As usual, once you have finished writing your program, exit Nano and save your work.

Running Your Program: Remote-Control Your Robot

Place your robot on a large surface and have your Wiimote handy. If your Pi requires a Bluetooth dongle, don't forget to plug it into one of the USB ports. To run your program, use an SSH terminal to enter:

```
pi@raspberrypi:~/robot $ python3 remote_control.py
```

Soon after program execution, a prompt will appear in the terminal asking you to press and hold the 1 and 2 buttons on your Wiimote simultaneously. You should hold these buttons until you get a success message, which can take up to 10 seconds. The Bluetooth handshake process can be fussy, so try to press them as soon as the program instructs you to do so.

If the pairing was successful, another message stating `Connection established` will appear. Alternatively, if the pairing was unsuccessful, an error message saying that `No Wiimotes were found` will be displayed, and your program will crash. If this is the case, and you are using an official Nintendo-branded Wiimote, then you most likely were not fast enough pressing the 1 and 2 buttons! Rerun the program with the same command and try again.

With your Wiimote now successfully connected, you should be able to make your robot dash around in any direction you want at the touch of a button! Remember that you can stop both motors at any point by pressing B on the underside of your Wiimote. As usual, you can kill the program by pressing CTRL-C.

www.itbook.store/books/9781593279202

VARYING THE MOTOR SPEED

Up until now your robot has been able to go at two speeds: 0 mph, or top speed! You might have noticed that this isn't the most convenient. Traveling at full speed makes precise maneuvers almost impossible, and you probably crashed into things a few times. Fortunately, it doesn't always have to be this way. Let's give your robot some control over its speed.

In this project, we'll build upon the previous example and create a remote control robot with variable motor speed. To do this I'll introduce a technique called *pulse-width modulation (PWM)*, and I'll explain how to use it inside the Python GPIO Zero library. We'll also put a special sensor called an *accelerometer* in your Wiimote to good use to create a much improved version of the remote control program!

Understanding How PWM Works

The Raspberry Pi is capable of providing *digital* outputs but not *analog* outputs. A digital signal can be either on or off, and nothing in between. An analog output, in contrast, is one that can be set at no voltage, full voltage, or anything in between. On the Raspberry Pi, at any given time a GPIO pin is either on or off, which is no voltage or full voltage. By this logic, motors connected to a Pi's GPIO can only either stop moving or go full speed.

That means that it is impossible to set a Pi's GPIO pin to "half voltage" for half the motor speed, for example. Fortunately, the PWM technique allows us to approximate this behavior.

To understand PWM, first take a look at the graph in Figure 4-6. It depicts the state of a digital output changing from low to high. This is what happens when you turn on one of your Pi's GPIO pins: it goes from 0 V to 3.3 V.

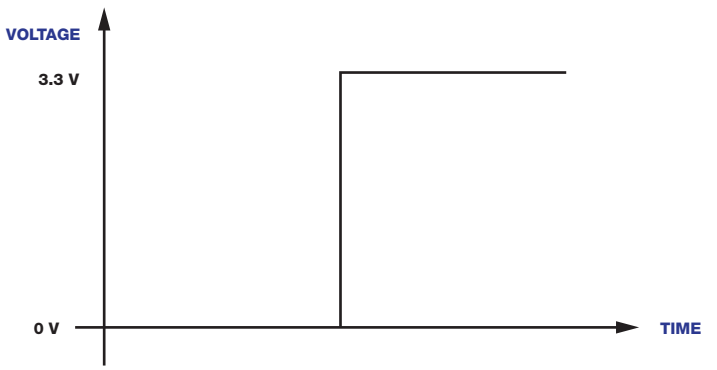

FIGURE 4-6
A state change from low (0 V) to high (3.3 V)

www.itbook.store/books/9781593279202

PWM works by turning a GPIO pin on and off so quickly that the device (in our case, a motor) "notices" only the *average* voltage at any given time. This means that the state is somewhere in between 0 V and 3.3 V. This average voltage depends on the *duty cycle*, which is simply the amount of time the signal is on, versus the amount of time a signal is off in a given period. It is given as a percentage: 25 percent means the signal was high for 25 percent of the time and low for 75 percent of the time; 50 percent means the signal was high for 50 percent of the time and low for the other 50 percent, and so on.

The duty cycle affects the output voltage proportionally, as shown in Figure 4-7. For example, for the Raspberry Pi, pulse-width modulating a GPIO pin at a 50 percent duty cycle would give a voltage of 50 percent: 3.3 V / 2 = 1.65 V.

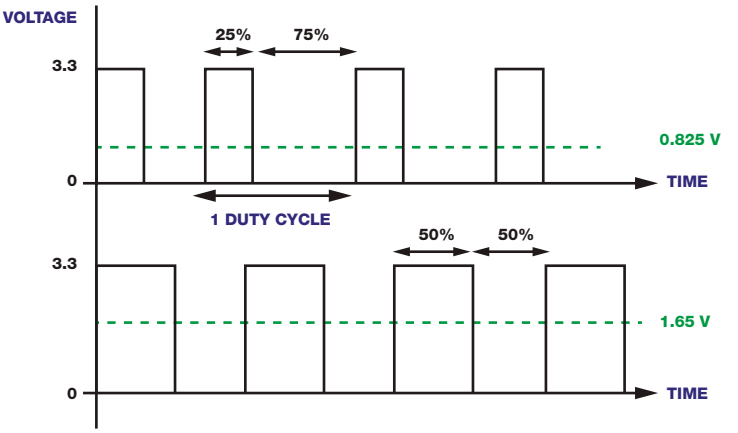

FIGURE 4-7
Two different PWM voltage traces: a duty cycle of 25 percent (top) and a duty cycle of 50 percent (bottom)

While PWM is not a perfect approximation of an analog signal, for most cases it works well, especially at this level. Digitally encoding analog signal levels will allow you to control the exact speed of your robot's movement.

The GPIO Zero Python library authors have made it easy to vary motor speed using PWM, so you don't need to know the exact mechanics behind it. All you need to do is provide a value between 0 and 1 in the parentheses of each motor command to represent a value between 0 percent and 100 percent, as follows:

```
robot.forward(0.25)
time.sleep(1)
robot.left(0.5)
```


www.itbook.store/books/9781593279202

```
time.sleep(1)
robot.backward()
time.sleep(1)
```

This program would command your robot to move forward for 1 second at 25 percent of its full speed, turn left at 50 percent of its full speed for another second, and then go backward at full speed for a final second. If you don't provide a value, Python assumes that the robot should move at full speed, just the same as if you were to enter a 1.

Understanding the Accelerometer

Before we improve upon the remote control program in the previous project, let's learn about the accelerometer in your Wiimote and how we can use it.

Previously, you used the D-pad of the Wiimote to provide control. These four buttons are digital and can only detect being pressed on or off. This isn't ideal for controlling both speed and direction at once.

Inside each Wiimote, however, there is a sensor called an *accelerometer* that can detect and measure the acceleration the Wiimote is undergoing at any point. This means that moving a Wiimote in the air provides sensory data in all three axes: in all three axes: *x*, *y*, and *z*. In this way, the accelerometer can track the direction of movement, and the speed of that direction. See Figure 4-8 for a diagram.

FIGURE 4-8
The axes of motion the Wiimote's accelerometer can detect

This kind of analog data is ideal for a variable-motor-speed remote control program. For example, the more you pitch the Wiimote in the *x* direction, the faster your robot could move forward.

NOTE
If your robot has been zipping around too fast in the previous examples, feel free to go back and adjust the speed in the last two projects using this method!

www.itbook.store/books/9781593279202

Looking at the Data

Before we rework the robot's program, it would be incredibly helpful to see the raw data that the accelerometer from the Wiimote outputs. Once we have an idea of what that output looks like, we can think about how to manipulate that data to correspond to the robot's movement.

Power the Pi on your robot from a wall outlet, open a new file in Nano and call it *accel_test.py*, and then enter the code in Listing 4-3—this script uses the `cwiid` library too, so if you haven't installed that, see the instructions in "Installing and Enabling Bluetooth" on page 88.

LISTING 4-3
The code to print raw accelerometer data

```
import cwiid
import time

❶ print("Press and hold the 1+2 buttons on your Wiimote simultaneously")
wii = cwiid.Wiimote()
print("Connection established")
❷ wii.rpt_mode = cwiid.RPT_BTN | cwiid.RPT_ACC

while True:
 ❸ print(wii.state['acc'])
   time.sleep(0.01)
```

This simple program prints the Wiimote's accelerometer data to the terminal every 0.01 seconds.

The `print()` statement denotes the start of the Wiimote setup ❶. The three following lines are the same as in the prior project, with the exception of the final line in that code block ❷, with which we're not just turning on a Wiimote's reporting mode like before, but also permitting Python to read values from both the buttons *and* the accelerometer. If you haven't come across it before, the keyboard character in the middle of this line is called a *vertical bar* or a *pipe*. It is likely to be located on the same key as the backslash on your keyboard.

An infinite `while` loop prints the status of the accelerometer ❸. The next line waits for 0.01 seconds between each *iteration* of the `while` loop so that the outputted data is more manageable. In programming, each time a loop goes round and executes again is called an iteration.

You can run this program with the command:

```
pi@raspberrypi:~/robot $ python3 accel_test.py
```


www.itbook.store/books/9781593279202

After you pair your Wiimote, accelerometer data should start printing to the terminal. The following output is some of the data that I saw in my terminal:

```
(147, 123, 136)
(151, 116, 136)
(130, 113, 140)
(130, 113, 140)
(130, 113, 140)
```

Each line of data is delivered as three values in parentheses, representing the x-, y-, and z-axes, respectively, which change as you move the Wiimote in the different axes. Experiment with different movements and watch as the figures go up and down. Exit the program by pressing CTRL-C.

With this raw data, we can put some thought into the next part of the programming process, namely answering the question: How can you translate those three figures into instructions for your robot? The best way to approach this problem is logically and in small steps.

Figuring Out the Remote Movement Control

First, consider the movement of your two-wheeled robot. Because it moves around only on the floor, and doesn't fly up and down, its movement can be expressed in two dimensions: *x* and *y*, as shown in Figure 4-9. We can disregard the z-axis data, which simplifies the problem substantially.

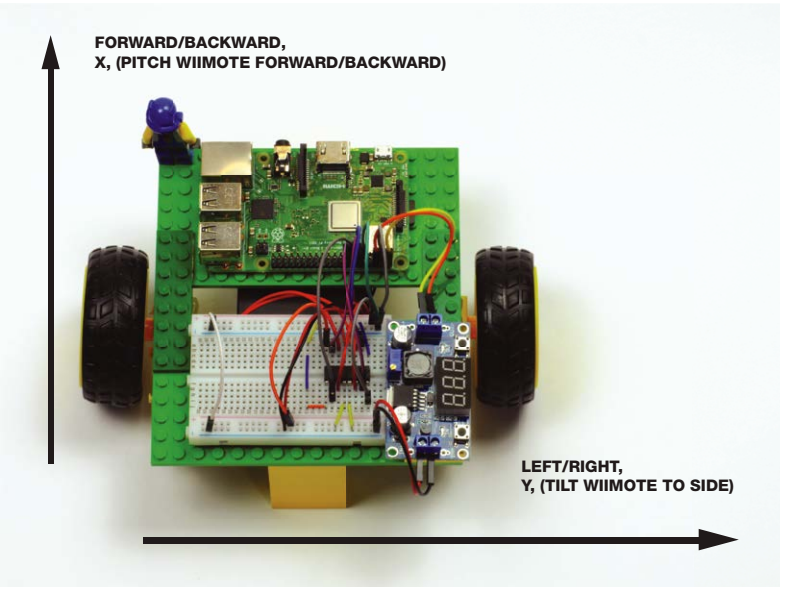

FIGURE 4-9
Only two axes of control are required for your two-wheeled robot.

www.itbook.store/books/9781593279202

Second, consider how you wish to hold the Wiimote when controlling your robot. I have decided to hold it horizontally, with the 1 and 2 buttons close to my right hand, as shown in Figure 4-10. This is the most common orientation for traditional Wii-based racing games and is ideal for controlling your robot.

FIGURE 4-10
How to hold the Wiimote in this project

When you're holding the Wiimote in this way, pitching it backward and forward controls the *x* values. Tilting it side-to-side controls the *y* values.

When you printed your accelerometer data, you may have noticed that the outputted numbers tended to be between 95 and 145. You can run the test program again to observe this. This is because the lowest *x* value is 95, when the Wiimote is pitched all the way back. This highest value is 145, when it's pitched entirely forward.

For the y-axes, left to right, the lowest value is 95 and the highest is 145. The difference between 145 and 95 is 50, and this gives us the usable range of data in each axis. See Figure 4-11 for an illustration of how the Wiimote's values change.

So far in this chapter, you've controlled your robot's movement by instructing it to go forward, backward, left, or right at full speed. We want to change this to vary the speed according to the accelerometer. Luckily, the `Robot` class from the GPIO Zero Python library has another way of turning the motors on and setting their speed that suits our needs.

www.itbook.store/books/9781593279202

FIGURE 4-11
The Wiimote's extreme accelerometer values

The `Robot` class has an *attribute*—a variable that is part of a class—called `value`. At any given time, `value` represents the motion of the robot's motors as a pair of numeric values between –1 and 1. The first value in the pair is for the left motor's speed, while the second value is for the right motor's speed. For example, (–1, –1) represents full speed backward, whereas (0.5, 0.5) represents half speed forward. A value of (1, –1) would represent turning full speed right. By setting the `value` attribute, you can manipulate your robot in any direction you wish. This is going to come in super-handy in the upcoming program!

Programming Your Robot for Variable Speed

Now that we've broken down this problem and found a neat and efficient final approach to the program, we can start coding! Use Nano to create a new program called *remote_control_accel.py* and input the code shown in Listing 4-4.

```
import gpiozero
import cwiid

robot = gpiozero.Robot(left=(17,18), right=(27,22))

print("Press and hold the 1+2 buttons on your Wiimote simultaneously")
wii = cwiid.Wiimote()
print("Connection established")
wii.rpt_mode = cwiid.RPT_BTN | cwiid.RPT_ACC
```

LISTING 4-4
Programming your robot to respond to the motion of your Wiimote

www.itbook.store/books/9781593279202

```
while True:
    ❶ x = (wii.state["acc"][cwiid.X] - 95) - 25
       y = (wii.state["acc"][cwiid.Y] - 95) - 25

    ❷ if x < -25:
              x = -25
       if y < -25:
              y = -25
       if x > 25:
              x = 25
       if y > 25:
              y = 25

    ❸ forward_value = (float(x)/50)*2
       turn_value = (float(y)/50)*2

    ❹ if (turn_value < 0.3) and (turn_value > -0.3):
              robot.value = (forward_value, forward_value)
       else:
              robot.value = (-turn_value, turn_value)
```

NOTE

Python (and many other programming languages) can deal with numbers in different ways. The main two number types in Python are called integers *and* floats*. Integers are whole numbers that have no decimal point. Floats (floating-point real values) have decimal points and can represent both the integer and fractional part of a number. For example, 8 is an integer, whereas 8.12383 or 8.0 is a float. In the* remote_control_accel.py *program, we need to use floats, as the movement of your robot will be governed by two numbers in between –1 and 1.*

The program shares the same Wiimote setup process as the accelerometer test program. Then we set up a `while` loop to keep running our code. The first statement ❶ reads the *x* value from the accelerometer and then stores it in a variable called `x`. Within the variable, the value undergoes two arithmetic operations. First, 95 is subtracted; this limits the data to a value between 0 and 50, rather than between 95 and 145, so that it fits within the usable range we discovered earlier. Second, we subtract a further 25. This ensures the range of data will be between –25 and +25. Exactly the same process then happens for the *y* value, and the result is then stored in a variable called `y`.

We need to do this because the `value` attribute of the `Robot` class accepts negative values for backward movement and positive values for forward movement. This manipulation balances the accelerometer data on either side of 0, making it clear which values are for reverse and which are for forward movement.

The four `if` statements ❷ eliminate the chance for errors later in the program. In the unlikely event that the Wiimote's accelerometer outputs data that is not within the –25 to +25 range, the `if` statements catch this occurrence and then round up or down to the relevant extremity.

Next, the final x-axis value for the robot is determined and stored in a variable called `forward_value` ❸. This calculation divides the `x` variable value by 50, providing a new proportional number

www.itbook.store/books/9781593279202

between –0.5 and 0.5. This result is then multiplied by 2 to get a value between –1 and 1. The same process is repeated to get the final y-axis value, which is then stored in a similar variable called `turn_value`.

The line at ❹ starts an `if/else` clause. If the `turn_value` is less than 0.3 or greater than –0.3, `robot.value` is set to be the `forward_value`. So, if the Wiimote is tilted by *less than 30 percent* to either side, the program will assume that you want the robot to move forward/backward. This means that your robot won't turn in the wrong direction at the slightest tilt of your Wiimote. The forward/backward speed of your robot is then set according to the pitch of your Wiimote. For example, if your Wiimote is pitched all the way forward, it will set `robot.value` to (1, 1) and your robot will accelerate forward.

Alternatively, if the Wiimote is tilted by more than 30 percent to either side, the program will assume that you want the robot to turn left or right on the spot. The program then turns the robot based on the angle of your Wiimote tilt. For example, if you have the Wiimote tilted all the way to the right, your robot will spin very quickly to the right; but if you have it tilted only slightly to the right, the robot will turn more slowly and in a more controlled manner.

As usual, after you have finished your program, exit Nano and save your work.

Running Your Program: Remote-Control Your Robot with PWM

Disconnect your robot from your wall outlet, and ensure that it is powered by its batteries. Then place it on a large surface and have your Wiimote in hand and in a horizontal orientation. To run your program, enter:

```
pi@raspberrypi:~/robot $ python3 remote_control_accel.py
```

After you have gone through the familiar Bluetooth handshake process, your robot should come to life and start to move as you change the orientation of your Wiimote. Experiment with driving it around at different speeds and practice maneuvering!

Challenge Yourself: Refine your Remote-Controlled Robot

When you have a feel for the behavior of your remote-controlled robot, take another look at the code and refine it as you see fit. For

www.itbook.store/books/9781593279202

example, you could try to make the steering more sensitive, limit the speed of your robot, or even make your robot move in a predefined pattern when you press a button. The possibilities are endless!

SUMMARY

This chapter has taken you from having a robot-shaped paperweight to a fully functional Wiimote-controlled little machine! We have covered a wide range of concepts from H-bridges to PWM to accelerometers. Over the process you have written three programs, each more advanced than the last.

In the next chapter, I'll guide you through making your robot a little bit more intelligent so that it can automatically avoid obstacles!

www.itbook.store/books/9781593279202