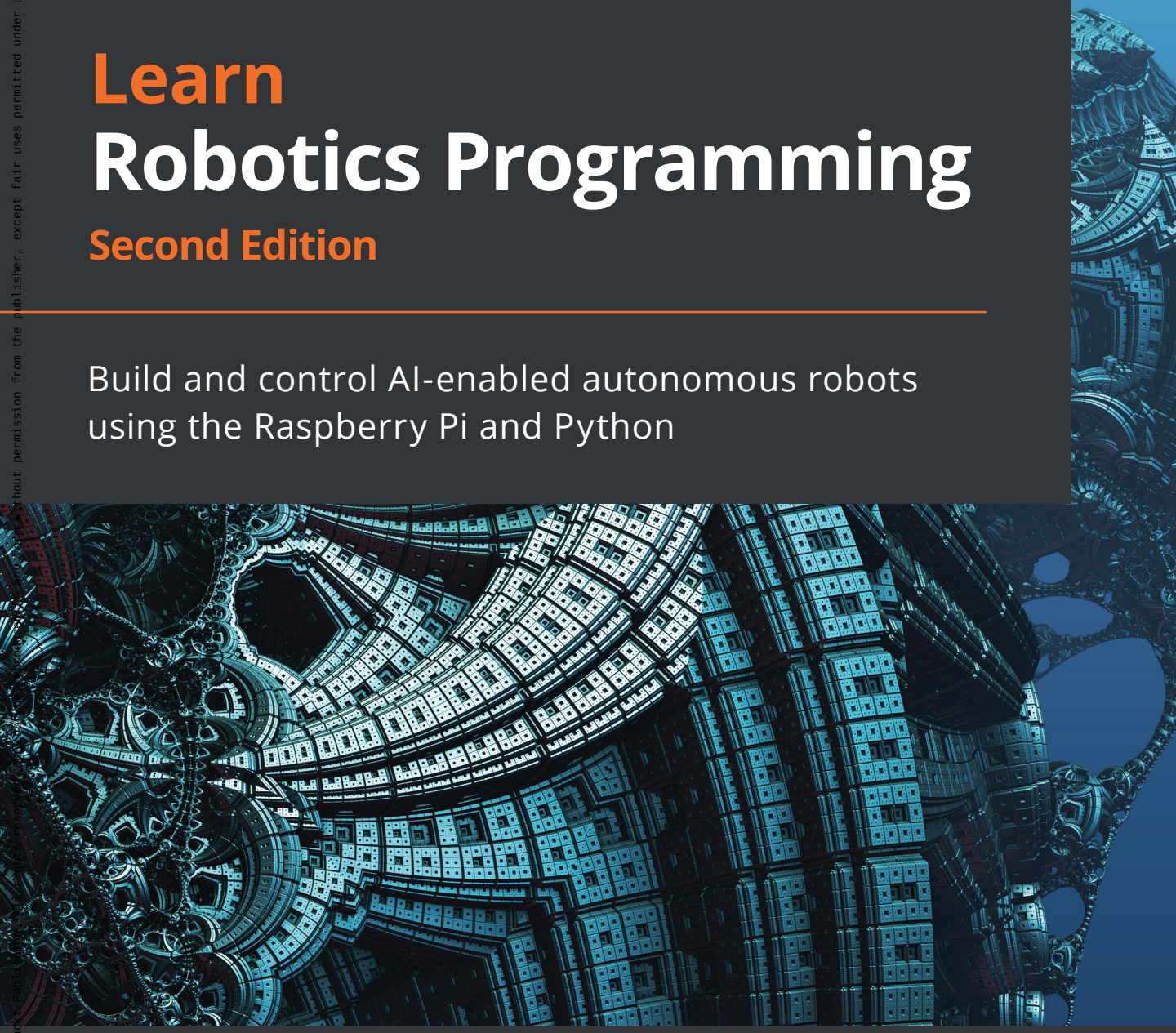
Copyright 2021. Packt Publishing. All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical reviews or other non-commercial uses permitted by copyright law. For permission requests, write to the publisher, except fair uses permitted under U.S. or applicable copyright law.

Learn Robotics Programming

Second Edition

Build and control AI-enabled autonomous robots using the Raspberry Pi and Python



Danny Staple

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/9/2023 11:36 AM via
AN: 2731479 ; Danny Staple.; Learn Robotics Programming : Build and Control AI-enabled Autonomous Robots Using the Raspberry Pi and Python
Account: ns335141

Learn Robotics Programming

Second Edition

Build and control AI-enabled autonomous robots using the Raspberry Pi and Python

Danny Staple

Packt

BIRMINGHAM—MUMBAI

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Learn Robotics Programming

Second Edition

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Wilson D'souza
Publishing Product Manager: Rahul Nair
Senior Editor: Rahul Dsouza
Content Development Editor: Nihar Kapadia
Technical Editor: Sarvesh Jaywant
Copy Editor: Safis Editing
Project Coordinator: Neil D'mello
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Aparna Bhagat

First published: November 2018

Second edition: February 2021

Production reference: 1140121

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83921-880-4

www.packt.com

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Contributors

About the author

Danny Staple builds robots and gadgets as a hobby, makes videos about robots, and attends community events such as Pi Wars and Arduino Day. He has worked as a professional software engineer since 2000, and he became a Python programmer in 2009 with a strong focus on DevOps and automation. Danny has worked with embedded systems, including embedded Linux systems, throughout the majority of his career. He mentors at CoderDojo Ham, where he shows kids how to code, and also used to run LEGO Robotics clubs.

The robots he has built with his children include TankBot, SkittleBot, Bangers N Bash (a lunchbox robot), Big Ole Yellow (more tank tracks), ArmBot, and SpiderBot.

I would like to thank David Anderson for being a great person to bounce ideas off and for his motivational energy. I would like to thank Ben Nuttall and Dave Jones (@waveform80) for GPIOZero, and for helping me out countless times on Twitter. Dave Jones kickstarted my journey into computer vision in a restaurant in Cardiff and is the author of the PiCamera library. Finally, I would like to thank my children, Helena and Jonathan, for their support and patience, even occasionally reviewing diagrams for me.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

About the reviewers

Leo White is a professional software engineer and graduate of the University of Kent, whose interests include electronics, 3D printing, and robotics. He first started programming on the Commodore 64, then later wrote several applications for the Acorn Archimedes, and currently programs set-top boxes for his day job. Utilizing the Raspberry Pi as a foundation, he has mechanized children's toys and driven robot arms, blogging about his experiences and processes along the way, as well as given presentations at Raspberry Jams and entered a variety of robots into the Pi Wars competition.

Ramkumar Gandhinathan is a roboticist and a researcher by profession. He started building robots in sixth grade and has been in the robotics field for over 15 years through personal and professional connections. He has built over 80+ robots of different types. With an overall professional experience of 7 years (4 years full-time and 3 years part-time/internships) in the robotics industry, he has 5 years of ROS experience. As a part of his professional career, he has built over 15 industrial robot solutions using ROS. He is also fascinated with building drones and is a drone pilot by practice. His research interests and passions are in the field of SLAM, motion planning, sensor fusion, multi-robot communication, and systems integration.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Table of Contents

Preface

Section 1: The Basics – Preparing for Robotics

1

Introduction to Robotics

What does robot mean? 4

Exploring advanced and impressive robots 6

The Mars rovers 8

Discovering robots in the home 9

The washing machine 9

Other household robots 10

Exploring robots in industry 11

Robot arms 12

Warehouse robots 13

Competitive, educational, and hobby robots 13

Summary 16

Assessment 17

Further reading 18

2

Exploring Robot Building Blocks – Code and Electronics

Technical requirements 20

Looking at what's inside a robot 20

Exploring types of robot components 23

Types of motors 24

Other types of actuators 26

Status indicators – displays, lights, and sounds 26

Types of sensors 27

Exploring controllers and I/O 29

I/O pins 29

Controllers 31

Choosing a Raspberry Pi 33

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Planning components and code structure 34
Planning the physical robot 37
Summary 40
Exercise 40
Further reading 40

3
Exploring the Raspberry Pi

Technical requirements 42
Exploring the Raspberry Pi's capabilities 42
Speed and power 42
Connectivity and networking 43
Picking the Raspberry Pi 3A+ 43
Choosing the connections 44
Raspberry Pi HATs 46
What is Raspberry Pi OS? 46
Preparing an SD card with Raspberry Pi OS 47
Summary 50
Assessment 50
Further reading 50

4
Preparing a Headless Raspberry Pi for a Robot

Technical requirements 52
What is a headless system, and why is it useful in a robot? 52
Setting up Wi-Fi on the Raspberry Pi and enabling SSH 54
Finding your Pi on the network 56
Setting up Bonjour for Microsoft Windows 56
Testing the setup 57
Troubleshooting 58
Using PuTTY or SSH to connect to your Raspberry Pi 59
Configuring Raspberry Pi OS 61
Renaming your Pi 62
Securing your Pi (a little bit) 64
Rebooting and reconnecting 65
Updating the software on your Raspberry Pi 67
Shutting down your Raspberry Pi 69
Summary 69
Assessment 69
Further reading 70

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5

Backing Up the Code with Git and SD Card Copies

Technical requirements 72
Understanding how code can be broken or lost 72
SD card data loss and corruption 72
Changes to the code or configuration 73
Strategy 1 – Keeping the code on a PC and uploading it 73
Strategy 2 – Using Git to go back in time 76
Strategy 3 – Making SD card backups 79
Windows 79
Mac 81
Linux 84
Summary 85
Assessment 86
Further reading 86

Section 2: Building an Autonomous Robot – Connecting Sensors and Motors to a Raspberry Pi

6

Building Robot Basics – Wheels, Power, and Wiring

Technical requirements 90
Choosing a robot chassis kit 90
Size 91
Wheel count 91
Wheels and motors 93
Simplicity 94
Cost 94
Conclusion 94
Choosing a motor controller board 95
Integration level 95
Pin usage 96
Size 96
Soldering 97
Power input 98
Connectors 98
Conclusion 98
Powering the robot 99
Test fitting the robot 102
Assembling the base 105
Attaching the encoder wheels 107
Fitting the motor brackets 107
Adding the castor wheel 110
Putting the wheels on 111
Bringing the wires up 111
Fitting the Raspberry Pi 112
Adding the batteries 113
The completed robot base 116

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Connecting the motors to the Raspberry Pi 116
Wiring the Motor HAT in 119
Independent power 121
Summary 123
Exercises 124
Further reading 124

7
Drive and Turn – Moving Motors with Python
Technical requirements 126
Writing code to test your motors 126
Preparing libraries 127
Test – finding the Motor HAT 127
Test – demonstrating that the motors move 130
Troubleshooting 131
Understanding how the code works 131
Steering a robot 133
Types of steering 133
Steering the robot we are building 137
Making a Robot object – code for our experiments to talk to the robot 138
Why make this object? 139
What do we put in the robot object? 140
Writing a script to follow a predetermined path 145
Summary 148
Exercises 148
Further reading 149

8
Programming Distance Sensors with Python
Technical requirements 152
Choosing between optical and ultrasonic sensors 152
Optical sensors 153
Ultrasonic sensors 154
Logic levels and shifting 154
Why use two sensors? 157
Attaching and reading an ultrasonic sensor 158
Securing the sensors to the robot 158
Adding a power switch 160
Wiring the distance sensors 163
Installing Python libraries to communicate with the sensor 166
Reading an ultrasonic distance sensor 166
Troubleshooting 169
Avoiding walls – writing a script to avoid obstacles 170
Adding the sensors to the robot class 171
Making the obstacle avoid behaviors 172
Summary 179
Exercises 179
Further reading 180

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9
Programming RGB Strips in Python

Technical requirements 182
What is an RGB strip? 182
Comparing light strip technologies 183
RGB values 184
Attaching the light strip to the Raspberry Pi 185
Attaching the LED strip to the robot 186
Making a robot display the code object 187
Making an LED interface 187
Adding LEDs to the Robot object 189
Testing one LED 191
Making a rainbow display with the LEDs 194
Colour systems 194
Making a rainbow on the LEDs 196
Using the light strip for debugging the avoid behavior 199
Adding basic LEDs to the avoid behavior 199
Adding rainbows 202
Summary 203
Exercises 203
Further reading 204

10
Using Python to Control Servo Motors

Technical requirements 206
What are servo motors? 206
Looking inside a servo 208
Sending input positions to a servo motor 208
Positioning a servo motor with the Raspberry Pi 210
Writing code for turning a servo 211
Troubleshooting 215
Controlling DC motors and servo motors 215
Calibrating your servos 216
Adding a pan and tilt mechanism 217
Building the kit 219
Attaching the pan and tilt mechanism to the robot 223
Creating pan and tilt code 224
Making the servo object 224
Adding the servo to the robot class 227
Circling the pan and tilt head 228
Running it 231
Troubleshooting 231
Building a scanning sonar 232
Attaching the sensor 233
Installing the library 236
Behavior code 236
Summary 239
Exercises 240
Further reading 240

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11
Programming Encoders with Python

Technical requirements 242
Measuring the distance traveled with encoders 242
Where machines use encoders 242
Types of encoders 243
Encoding absolute or relative position 244
Encoding direction and speed 246
The encoders we are using 247
Attaching encoders to the robot 248
Preparing the encoders 249
Lifting the Raspberry Pi 250
Fitting the encoders onto the chassis 250
Wiring the encoders to the Raspberry Pi 251
Detecting the distance traveled in Python 253
Introducing logging 253
Simple counting 254
Adding encoders to the Robot object 257
Turning ticks into millimeters 260
Driving in a straight line 262
Correcting veer with a PID 263
Creating a Python PID controller object 265
Writing code to go in a straight line 266
Troubleshooting this behavior 269
Driving a specific distance 270
Refactoring unit conversions into the EncoderCounter class 270
Setting the constants 272
Creating the drive distance behavior 272
Making a specific turn 275
Writing the drive_arc function 281
Summary 282
Exercises 283
Further reading 283

12
IMU Programming with Python

Technical requirements 286
Learning more about IMUs 287
Suggested IMU models 287
Soldering – attaching headers to the IMU 288
Making a solder joint 289
Attaching the IMU to the robot 291
Physical placement 291
Wiring the IMU to the Raspberry Pi 294
Reading the temperature 295
Installing the software 295
Troubleshooting 297
Reading the temperature register 297
Troubleshooting 302
Simplifying the VPython command line 303
Reading the gyroscope in Python 303
Understanding the gyroscope 304
Adding the gyroscope to the interface 307

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Plotting the gyroscope 308

Reading an accelerometer in Python 310
Understanding the accelerometer 310
Adding the accelerometer to the interface 311
Displaying the accelerometer as a vector 311

Working with the magnetometer 313
Understanding the magnetometer 313
Adding the magnetometer interface 315
Displaying magnetometer readings 315

Summary 317
Exercises 317
Further reading 318

Section 3: Hearing and Seeing – Giving a Robot Intelligent Sensors

13

Robot Vision – Using a Pi Camera and OpenCV

Technical requirements 322

Setting up the Raspberry Pi camera 322
Attaching the camera to the pan-and-tilt mechanism 323
Wiring in the camera 327

Setting up computer vision software 330
Setting up the Pi Camera software 330
Getting a picture from the Raspberry Pi 331
Installing OpenCV and support libraries 331

Building a Raspberry Pi camera stream app 332
Designing the OpenCV camera server 333
Writing the CameraStream object 334
Writing the image server main app 336
Building a template 338
Running the server 338
Troubleshooting 339

Running background tasks when streaming 340

Following colored objects with Python 347
Turning a picture into information 348
Enhancing the PID controller 351
Writing the behavior components 352
Running the behavior 359
Troubleshooting 361

Tracking faces with Python 362
Finding objects in an image 362
Planning our behavior 366
Writing face-tracking code 367
Running the face-tracking behavior 371
Troubleshooting 372

Summary 372
Exercises 373
Further reading 373

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

14

Line-Following with a Camera in Python

Technical requirements 376
Introduction to line following 376
What is line following? 376
Usage in industry 377
Types of line following 378
Making a line-follower test track 379
Getting the test track materials in place 379
Making a line 380
Line-following computer vision pipeline 381
Camera line-tracking algorithms 381
The pipeline 382
Trying computer vision with test images 384
Why use test images? 384
Capturing test images 384
Writing Python to find the edges of the line 386
Locating the line from the edges 390
Trying test pictures without a clear line 391
Line following with the PID algorithm 393
Creating the behavior flow diagram 394
Adding time to our PID controller 395
Writing the initial behavior 396
Tuning the PID 403
Troubleshooting 404
Finding a line again 404
Summary 405
Exercises 406
Further reading 406

15

Voice Communication with a Robot Using Mycroft

Technical requirements 408
Introducing Mycroft – understanding voice agent terminology 409
Speech to text 409
Wake words 409
Utterances 409
Intent 410
Dialog 410
Vocabulary 410
Skills 410
Limitations of listening for speech on a robot 411
Adding sound input and output to the Raspberry Pi 411
Physical installation 412
Installing a voice agent on a Raspberry Pi 413
Installing the ReSpeaker software 414
Getting Mycroft to talk to the sound card 417
Starting to use Mycroft 418
Troubleshooting 420

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Programming a Flask API 421
Overview of Mycroft controlling the robot 421
Starting a behavior remotely 422
Programming the Flask control API server 425
Troubleshooting 427
Programming a voice agent with Mycroft on the Raspberry Pi 428
Building the intent 428
Troubleshooting 435
Adding another intent 436
Summary 438
Exercises 438
Further reading 439

16
Diving Deeper with the IMU

Technical requirements 442
Programming a virtual robot 442
Modeling the robot in VPython 442
Detecting rotation with the gyroscope 447
Calibrating the gyroscope 448
Rotating the virtual robot with the gyroscope 450
Detecting pitch and roll with the accelerometer 454
Getting pitch and roll from the accelerometer vector 454
Smoothing the accelerometer 458
Fusing accelerometer and gyroscope data 459
Detecting a heading with the magnetometer 463
Calibrating the magnetometer 463
Getting a rough heading from the magnetometer 470
Combining sensors for orientation 473
Driving a robot from IMU data 480
Summary 482
Exercises 482
Further reading 483

17
Controlling the Robot with a Phone and Python

Technical requirements 486
When speech control won't work – why we need to drive 486
Menu modes – choosing your robot's behavior 487
Managing robot modes 489
Troubleshooting 491
The web service 491
The template 492
Running it 494
Troubleshooting 496

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Choosing a controller — how we are going to drive the robot, and why 497
Design and overview 498
Preparing the Raspberry Pi for remote driving—get the basic driving system going 500
Enhancing the image app core 501
Writing the manual drive behavior 502
The template (web page) 505
The style sheet 509
Creating the code for the sliders 512
Running this 516
Troubleshooting 517
Making the robot fully phone-operable 518
Making menu modes compatible with Flask behaviors 518
Loading video services 519
Styling the menu 522
Making the menu start when the Pi starts 524
Adding lights to the menu server 524
Using systemd to automatically start the robot 525
Summary 528
Exercises 529
Further reading 530

Section 4: Taking Robotics Further

18

Taking Your Robot Programming Skills Further

Online robot building communities – forums and social media 534
YouTube channels to get to know 535
Technical questions – where to get help 536
Meeting robot builders – competitions, makerspaces, and meetups 537
Makerspaces 537
Maker Faires, Raspberry Jams, and Dojos 538
Competitions 538
Suggestions for further skills – 3D printing, soldering, PCB, and CNC 539
Design skills 539
Skills for shaping and building 541
Electronics skills 543
Finding more information on computer vision 545
Books 545
Online courses 545
Social media 546
Extending to machine learning 546
Robot Operating System 547
Summary 548
Further reading 548

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

19

Planning Your Next Robot Project – Putting It All Together

Technical requirements 550
Visualizing your next robot 550
Making a block diagram 552
Choosing the parts 553
Planning the code for the robot 555
Letting the world know 557
Summary 558

Other Books You May Enjoy

Index

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Preface

Learn Robotics Programming is about building and programming a robot with smart behavior. It covers the skills required to make and build a gadget from parts, including how to choose them. These parts include sensors, motors, cameras, microphones, speakers, lights, and a Raspberry Pi.

This book continues with how to write code to make those parts do something interesting. The book uses Python, together with a little bit of HTML/CSS and JavaScript.

The technology used is intended to include things that are available and affordable and the code is shown to demonstrate concepts, so they can be used and combined to create even more interesting code and robots.

The topics combine aspects of being a programmer with aspects of being a robot maker, with a number of specialist topics such as computer vision and voice assistants thrown in.

Who this book is for

This book is intended for someone with a little programming experience, or someone more experienced but looking to apply their skills to a hardware project. You do not need to be an expert-level programmer, but do have to have written some lines of code and be comfortable with looping, conditionals, and functions. Knowledge of object-oriented- (class- and object-) based programming isn't necessary but is introduced in the book.

The book does not require a specialist workshop, although there will be a little soldering and bolting things together. This will be introduced later in the book.

You do not need to have any experience at all with electronics or making things, but hopefully, you'll have a healthy interest in learning more, since some very basic concepts are introduced throughout the book. Being keen to build a robot, get it to do stuff, and find out what to do with it next is probably the most important aspect of the book.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

What this book covers

Chapter 1, *Introduction to Robotics*, introduces what robots are, and finds examples in the home and industry, along with the kinds of robots beginners build.

Chapter 2, *Exploring Robot Building Blocks – Code and Electronics*, looks at the components of a robot. We will start making choices about the robot's parts and see block diagrams for both systems and code.

Chapter 3, *Exploring the Raspberry Pi*, introduces the Raspberry Pi and its connections and the Raspbian Linux operating system we'll use on it, and also covers the preparation of an SD card for use in a robot.

Chapter 4, *Preparing a Headless Raspberry Pi for a Robot*, shows you how to set up an untethered Raspberry Pi and communicate with it wirelessly.

Chapter 5, *Backing Up the Code with Git and SD Card Copies*, shows how code can be lost or broken, then ways to protect your work and keep a history of it.

Chapter 6, *Building Robot Basics – Wheels, Power, and Wiring*, introduces the trade-offs for buying and test fitting a robot base, then assembling it.

Chapter 7, *Drive and Turn – Moving Motors with Python*, shows you how to write code to move a robot, laying down the foundations for the code in subsequent chapters.

Chapter 8, *Programming Distance Sensors with Python*, adds sensors and code to make a robot that autonomously avoids walls and obstacles.

Chapter 9, *Programming RGB Strips in Python*, adds multicolored lights to your robot. Explore this additional output to use for debugging or fun on the robot.

Chapter 10, *Using Python to Control Servo Motors*, shows how to use these motors to position a sensor head, and where they could be used in arms or legs on other robots.

Chapter 11, *Programming Encoders with Python*, demonstrates how odometry/tacho wheels can be read in your code, letting your robot drive in a straight line, make an accurate turn, or record how far it's driven. This chapter also introduces the PID controller.

Chapter 12, *IMU Programming with Python*, introduces the **Inertial Measurement Unit** (**IMU**), a set of sensors to measure temperature, acceleration, turning speeds, and magnetic fields. This chapter also gives you an introduction to soldering and VPython.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Chapter 13, *Robot Vision – Using a Pi Camera and OpenCV*, shows how to get data from a camera and use computer vision to make movements based on what the robot sees. This chapter also streams processed video to a browser.

Chapter 14, *Line-Following with a Camera in Python*, demonstrates how to make line-following behavior with the Raspberry Pi camera.

Chapter 15, *Voice Communication with a Robot Using Mycroft*, builds a voice control agent to link with your robot, letting you talk to control it and receive voice feedback.

Chapter 16, *Diving Deeper with the IMU*, takes the sensors we learned about in *Chapter 12*, *IMU Programming with Python*, and combines them to provide data about the orientation of the robot, building behavior that responds to the compass direction.

Chapter 17, *Controlling the Robot with a Phone and Python*, builds a menu system and a gaming-style control pad for your robot from your smartphone, letting you drive while seeing what the robot sees.

Chapter 18, *Taking Your Robot Programming Skills Further*, looks at the wider world of robotics, what communities there are, how to get in touch with other robot builders and makers, potential development areas, and where to compete with a robot.

Chapter 19, *Planning Your Next Robot Project – Putting It All Together*, is the final chapter, where we summarize what you have seen in the book, while encouraging you to plan the construction of your next robot.

To get the most out of this book

Before you begin with this book, you need to have programmed a little in a text programming language. I am assuming some familiarity with variables, conditional statements, looping, and functions.

You will need a computer, running macOS, Linux, or Windows, an internet connection, and Wi-Fi.

In terms of manual skills, I assume that you can use a screwdriver, that you can deal with occasional fiddly operations, and that you won't be too scared off by the possibility of soldering things.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Code examples have been tested on Python 3 with Raspbian Buster and Picroft Buster Keaton. The book will show you how to install these when needed. The book will show you how to choose and find robot parts when needed too.

Software/hardware covered in the book	OS requirements
Python 3	Raspbian Buster
Picroft/Mycroft	Picroft Buster Keaton
OpenCV	Raspbian Buster/Python 3
VPython	Raspbian Buster/Python 3
Flask	Python 3

Please read the appropriate chapters with trade-offs and recommendations before buying robot hardware.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

After reading the book, please come and join the #piwars community on Twitter for lots of robot discussion.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Code in Action videos for this book can be viewed at `http://bit.ly/3bu5wHp`.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781839218804_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This sets one LED at `led_number` to the specified `color`."

A block of code is set as follows:

```
cyan_rgb = [int(c * 255) for c in cyan]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
right_distance = self.robot.right_distance_sensor.distance
  # Display this
            self.display_state(left_distance, right_distance)
```

Any command-line input or output is written as follows:

```
>>> r.leds.show()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **4** for **Other USB Microphone** and try the sound test."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Section 1: The Basics – Preparing for Robotics

In this section, we will learn what a robot is with examples, get an idea of what is in a robot, and get a Raspberry Pi ready for robot experiments.

This part of the book comprises the following chapters:

- *Chapter 1*, *Introduction to Robotics*
- *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*
- *Chapter 3*, *Exploring the Raspberry Pi*
- *Chapter 4*, *Preparing a Headless Raspberry Pi for a Robot*
- *Chapter 5*, *Backing Up the Code with Git and SD Card Copies*

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

1
Introduction to Robotics

Throughout this book, we will build a robot and create programs for it that give the robot behaviors that make it appear intelligent and able to make decisions. We will write code to use sensors to observe the robot's surroundings and build real-world examples of advanced topics, including vision, speech recognition, and talking.

You will see how the simple build techniques, when combined with a little bit of code, will result in a machine that feels like some kind of pet. You will also see how to debug it when things go wrong, which they will. You'll find out how to give the robot ways to indicate problems back to you, along with selecting the behavior you would like to demonstrate. We will connect a joypad to it, give it voice control, and finally show you how to plan a further robot build.

Before we start building a robot, it's worth spending a little time on what a robot is. We can explore some types of robots, along with basic principles that distinguish robots from other machines. You'll think a little about where the line between robot and non-robot machines is located, and then perhaps muddy that line a little bit with the somewhat fuzzy truth. We will then look at a number of robots built in the hobbyist and amateur robotics scenes.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this chapter, we will be covering the following topics:

- What does *robot* mean?
- Exploring advanced and impressive robots
- Discovering robots in the home
- Exploring robots in industry
- Competitive, educational, and hobby robots

What does robot mean?

A **robot** is a machine that makes autonomous decisions based on input from sensors. A software agent is a program that automatically processes input and produces output. Perhaps a robot is best described as an autonomous software agent with sensors and moving outputs, or it could be described as an electromechanical platform with software running on it. Either way, a robot requires electronics, mechanical parts, and code.

The word *robot* conjures up images of fantastic sci-fi creations, devices with legendary strength and intelligence. These often follow the human body plan, making them an **android**, a human-like robot. They're often given a personality and behave like a person who is, in some simple way, naïve:

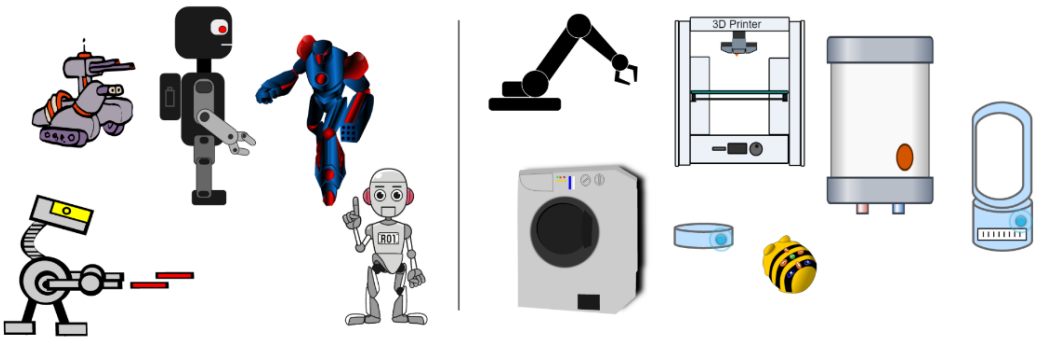


Figure 1.1 – Science fiction and real-world robots. Images used are from the public domain OpenClipArt library

The word *robot* comes from science fiction (also known as sci-fi). The word is derived from the Czech word for *slave* and was first used in the 1921 Karel Capek play, *Rossum's Universal Robots*. The science fiction author Isaac Asimov coined the word *robotics* as he explored intelligent robot behavior.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Most real robots in our homes and industries are not cutting-edge and eye-catching. Most do not stand on two legs, or indeed any legs at all. Some are on wheels, and some are not mobile but still have moving parts and sensors.

Robots such as modern washing machines, autonomous vacuum cleaners, fully self-regulating boilers, and air sampling fans have infiltrated our homes and are part of everyday life. They aren't threatening and have become just another machine around us. The 3D printer, robot arm, and learning toys are a bit more exciting, though:

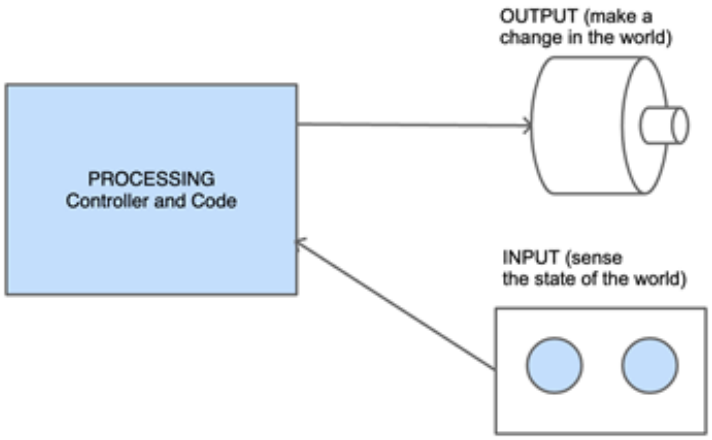


Figure 1.2 – The robot, simplified and deconstructed

At their core, robots can all be simplified down to **outputs** such as a motor, **inputs** such as a **sensor**, and a **controller** for processing or running code. So, a basic robot would look something like this:

- It has inputs and sensors to measure and sample properties of its environment.
- It has outputs such as motors, lights, sounds, valves, or heaters to alter its environment.
- It uses data from its inputs to make autonomous decisions about how it controls its outputs.

Now, we will go ahead and look at some advanced robots in the next section.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exploring advanced and impressive robots

Now that you have an overview of robots in general, I'll introduce some specific examples that represent the most remarkable robots around, and what they are capable of. Except for the Mars robots, human and animal forms have been favored by these robot makers for their adaptability, contrasting with robots designed for industrial use and intended for single repeated use.

Figure 1.3 shows the similarities between these robots and humans/animals:

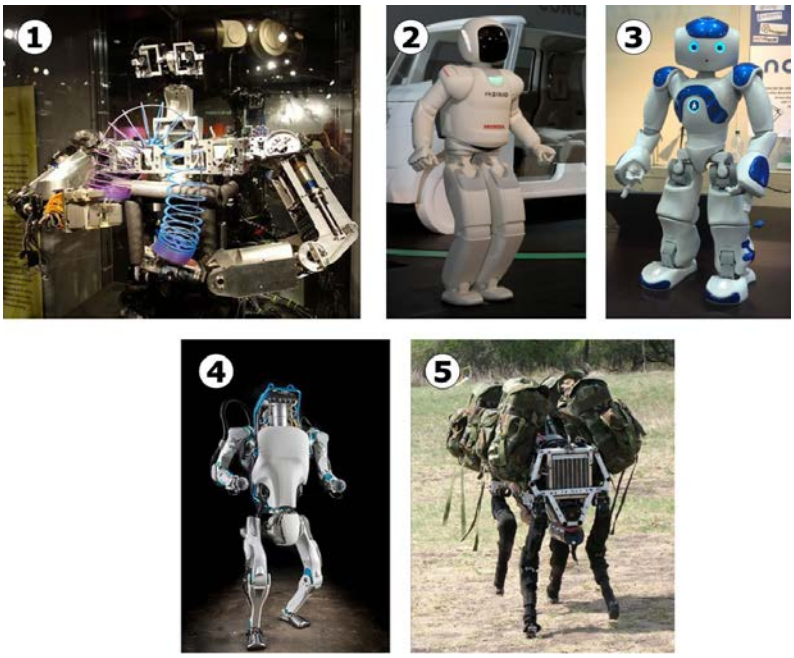


Figure 1.3 – A selection of human and animal-like robots. [Image credits: Image 1: This image can be found at https://commons.wikimedia.org/wiki/File:Cog,_1993-2004,_view_2_-_MIT_Museum_-_DSC03737.JPG, and is in the public domain; Image 2: This image can be found at https://commons.wikimedia.org/wiki/File:Honda_ASIMO_(ver._2011)_2011_Tokyo_Motor_Show.jpg, by Morio, under CC BY-SA 3.0, at `https://creativecommons.org/licenses/by-sa/3.0/deed.en`; Image 3: This image can be found at https://commons.wikimedia.org/wiki/File:Nao_Robot_(Robocup_2016).jpg and is in the public domain; Image 4: This image can be found at https://commons.wikimedia.org/wiki/File:Atlas_from_boston_dynamics.jpg, by https://www.kansascity.com/news/business/technology/917xpi/picture62197987/ALTERNATES/FREE_640/atlas%20from%20boston%20dynamics, under CC BY-SA 4.0, at `https://creativecommons.org/licenses/by-sa/4.0/deed.en`; Image 5: This image can be found at https://commons.wikimedia.org/wiki/Commons:Licensing#Material_in_the_public_domain and is in the public domain

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

What these robots have in common is that they try to emulate humans and animals in the following ways:

1. Robot 1 is Cog from the Massachusetts Institute of Technology. Cog was an attempt to be human-like in its movements and sensors.
2. Robot 2 is the Honda ASIMO, which walks and talks a little like a human. ASIMO's two cameras perform object avoidance, as well as gestures and face recognition, and have a laser distance sensor to sense the floor. It follows marks on the floor with infrared sensors. ASIMO accepts voice commands in English and Japanese.
3. Robot 3 is the Nao robot from Softbank Robotics. This cute 58 cm tall robot was designed as a learning and play robot for users to program. It has sensors to detect its motion, including if it is falling, and ultrasonic distance sensors to avoid bumps. Nao uses speakers and a microphone for voice processing. It has multiple cameras to perform similar feats to the ASIMO.
4. Robot 4 is Atlas from Boston Dynamics. This robot is fast on two legs and has natural-looking movement. It has a **laser radar** (**LIDAR**) array, which it uses to sense what is around it so as to plan its movement and avoid collisions.
5. Robot 5 is the Boston Dynamics BigDog, a four-legged robot, or quadruped. It can walk and run. It's one of the most stable four-legged robots, staying upright when being pushed, shoved, and walking in icy conditions.

You'll add some features like these in the robot you'll build. We'll use distance sensors to avoid obstacles, using ultrasonic sensors in the same way as Nao, and discussing laser distance sensors like ASIMO. We'll explore a camera for visual processing, line sensors to follow marks on the floor, and voice processing to work with spoken commands. We'll build a pan and tilt mechanism for a camera like the head of Cog.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Mars rovers

The **Mars rover robots** are designed to work on a different planet, where there is no chance of human intervention if it breaks. They are robust by design. Updated software can only be sent to a Mars rover via a remote link as it is not practical to send up a person with a screen and keyboard. The Mars rover is **headless by design**:

Figure 1.4 – NASA's Curiosity rover at Glen Etive, Mars (Image Credit: NASA/JPL-Caltech/MSSS; https://mars.nasa.gov/resources/24670/curiosity-at-glen-etive/?site=msl)

Mars rovers depend on wheels instead of legs, since stabilizing a robot on wheels is far simpler than doing it for one that uses legs, and there is less that can go wrong. Each wheel on the Mars rovers has its own motor. The wheels are arranged to provide maximum grip and stability to tackle Mars's rocky terrain and lower gravity.

The Curiosity rover was deposited on Mars with its sensitive camera folded up. After landing, the camera was unfolded and positioned with servo motors. The camera is pointed using a **pan and tilt** mechanism. It needs to take in as much of the Mars landscape as it can, sending back footage and pictures to NASA for analysis.

Like the Mars robots, the robot you'll build in this book uses motor-driven wheels. Our robot is also designed to run without a keyboard and mouse, being **headless by design**. As we expand the capabilities of our robot, we'll also use servo motors to drive a pan and tilt mechanism.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Discovering robots in the home

Many robots have already infiltrated our homes. They are overlooked as robots because, at first glance, they appear ordinary and mundane. However, they are more sophisticated than they appear.

The washing machine

Let's start with the washing machine. It is used every day in some homes, with a constant stream of clothes to wash, spin, and dry. But how is this a robot?

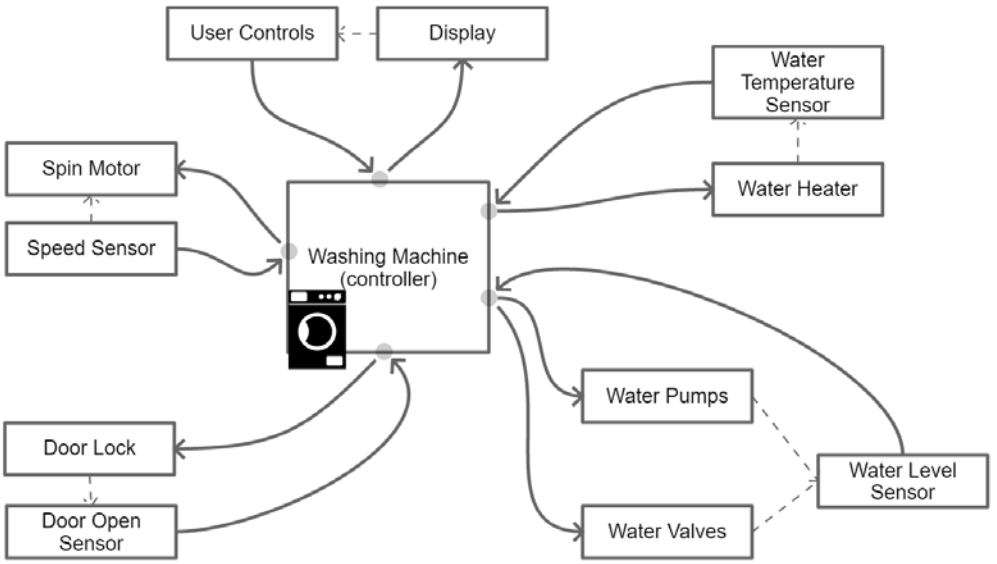


Figure 1.5 – Components of a washing machine

Figure 1.5 shows a washing machine as a block diagram. There's a central controller connected to the display with controls to select a program. The lines going out of the controller are outputs. The connections coming into the controller are data from sensors. The dashed lines from outputs to the sensors show a closed loop of output actions in the real world, causing sensor changes. This is feedback, an essential concept in robotics.

The washing machine uses the display and buttons to let the user choose the settings and see the status. After the start button is pressed, the controller checks the door sensor and will sensibly refuse to start if the door is open. Once the door is closed, and the start button is pressed, it will output to lock the door. After this, it uses heaters, valves, and pumps to fill the drum with heated water, using sensor feedback to regulate the water level and temperature.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Each process could be represented by a set of statements like these, which simultaneously fill the drum and keep it heated:

```
start water pump
turn on the water heater
while water is not filled and water is not hot enough:
  if water filled then
    stop water pump
  if the water is hot enough then
    turn off heater
  else
    turn on the water heater
```

Note the `else` there, which is in case the water temperature drops below the correct temperature slightly. The washing machine then starts the drum spinning sequence – slow turns, fast spins, sensing the speed to meet the criteria. It will drain the drum, spin the clothes dry, release the door lock, and stop.

This washing machine is, in every respect, a robot. A washing machine has sensors and outputs to affect its environment. Processing allows it to follow a program and use sensors with feedback to reach and maintain conditions. A washing machine repair person may be more of a roboticist than I.

Other household robots

A gas central heating boiler has sensors, pumps, and valves. The boiler uses feedback mechanisms to maintain the temperature of the house, water flow through heating, gas flow, and ensuring that the pilot light stays lit. The boiler is automatic and has many robot-like features, but it is stationary and could not readily be adapted to other purposes. The same could be said for other home appliances such as smart fans and printers.

Smart fans use sensors to detect room temperature, humidity, and air quality, and then output through the fan speed and heating elements.

Other machines in the home, like a microwave, for example, have only timer-based operation, they do not make decisions, and are too simple to be regarded as robots.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Perhaps the most obvious home robot is a robot vacuum cleaner, as shown in *Figure 1.6*:

Figure 1.6 – A robotic vacuum cleaner – PicaBot (Image credit: Handitec [Public Domain - https://commons.wikimedia.org/wiki/File:PicaBot.jpg])

This wheeled mobile robot is like the one we will build here, but prettier. They are packed with sensors to detect walls, bag levels, and barrier zones, and avoid collisions. They most represent the type of robot we are looking at. This robot is autonomous, mobile, and could be reprogrammed to different behaviors.

As we build our robot, we will explore how to use its sensors to detect things and react to them, forming the same feedback loops we saw in the washing machine.

Exploring robots in industry

Another place where robots are commonly seen is in industry. The first useful robots were used in factories, and have been there for a long time.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Robot arms

Robot arms range from tiny delicate robots for turning eggs, to colossal monsters moving shipping containers. Robot arms tend to use stepper and servo motors. We will look at servo motors in the pan and tilt mechanism used in this book. Most industrial robot arms (for example, ABB welding robots) follow a predetermined pattern of moves, and do not possess any decision making. However, for a more sensor-based and smart system, take a look at the impressive **Baxter** from Rethink Robotics in *Figure 1.7*. Baxter is a collaborative robot designed to work alongside humans:

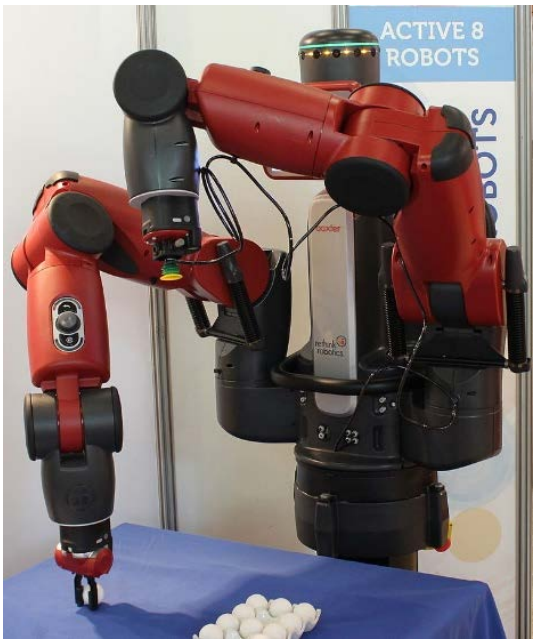


Figure 1.7 – The Rethink Robotics Baxter Robot (Image credit: Baxter at Innorobo by © Xavier Caré / Wikimedia Commons [CC-BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)])

Many robot arms are unsafe to work next to and could result in accidents, requiring cages or warning markings around them. Not so with Baxter; it can sense a human and work around or pause for safety. In the preceding image, these sensors are seen around the *head*. The arm sensors and soft joints also allow Baxter to sense and react to collisions.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Baxter has a training and repeat mechanism for workers to adapt it to a task. It uses sensors to detect joint positions when being trained or playing back motions. Our robot will use encoder sensors to precisely control wheel movements.

Warehouse robots

Another common type of robot used in industry is those that move items around a factory floor or warehouse:

Figure 1.8 – Warehouse robot systems: Stingray system by TGWmechanics [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], and Intellicart by Mukeshhrs [public domain]

Figure 1.8 picture 1 shows robotic crane systems for shifting pallets in storage complexes. They receive instructions to move goods within shelving systems.

Smaller item-moving robots, like Intellicart in *Figure 1.8* picture 2, employ line sensors, by following lines on the floor, magnetically sensing wires underneath the floor, or by following marker beacons like ASIMO. Our robot will follow lines such as these. These line-following carts frequently use wheels because these are simple to maintain and can form stable platforms.

Competitive, educational, and hobby robots

The most fun robots are those created by amateur robot builders. This is an extremely innovative space.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Robotics always had a home in education, with academic builders using them for learning and experimentation platforms. Many commercial ventures have started in this setting. University robots are often group efforts, with access to hi-tech equipment to create them:

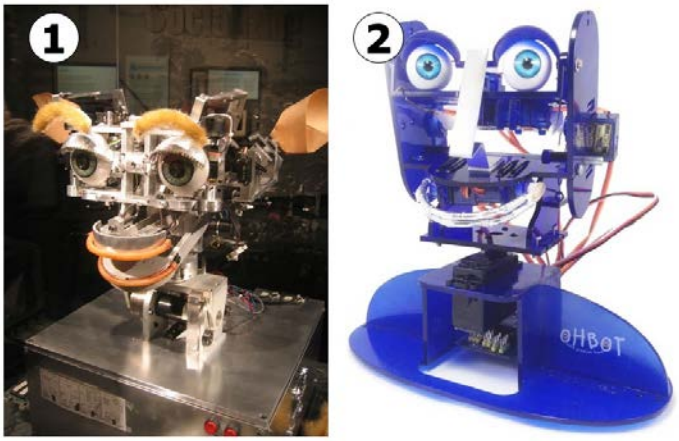


Figure 1.9 – Kismet [Jared C Benedict CC BY-SA 2.5 https://creativecommons.org/licenses/by-sa/2.5] and OhBot [AndroidFountain [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)]]

Kismet (*Figure 1.9* picture 1) was created at MIT in the late 90s. Several hobbyist robots are derived from it. It was groundbreaking at the time, using motors to drive face movements mimicking human expressions. OhBot, a low-priced hobbyist kit using servo motors, is based on Kismet. OhBot (*Figure 1.9* picture 2) links with a Raspberry Pi, using voice recognition and camera processing to make a convincing face.

Hobby robotics is strongly linked with the open source software/hardware community, making use of sites such as GitHub (`https://github.com`) for sharing designs, and code, leading to further ideas. Hobbyist robots can be created from kits available on the internet, with modifications and additions. The kits cover a wide range of complexity, from simple three-wheeled bases to drone kits and hexapods. They come with or without the electronics included. An investigation of kits will be covered in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*. I used a hexapod kit to build SpiderBot (*Figure 1.10*) to explore the walking motion:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

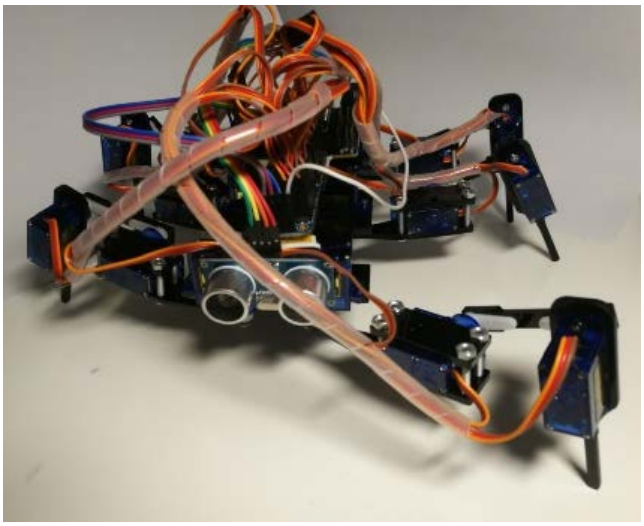


Figure 1.10 – Spiderbot, made by me, based on a kit. The controller is an esp8266 + Adafruit 16 Servo Controller

Skittlebot was my Pi Wars 2018 entry, built using toy hacking, repurposing a remote control excavator toy into a robot platform. **Pi Wars** is an autonomous robotics challenge for Raspberry Pi-based robots, with both manual and autonomous challenges. There were entries with decorative cases and resourceful engineering. **Skittlebot** (*Figure 1.11*) uses three distance sensors to avoid walls, and we will investigate this kind of sensor in *Chapter 8*, *Programming Distance Sensors with Python*. Skittlebot uses a camera to find colored objects, as we will see in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*:

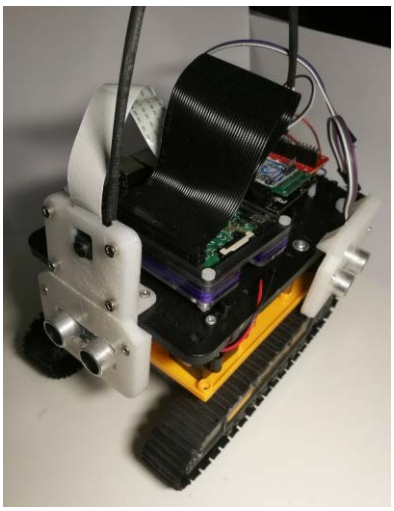


Figure 1.11 – Skittlebot – My PiWars 2018 Robot, based on a toy

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Some hobbyist robots are built from scratch, using 3D printing, laser cutting, vacuum forming, woodwork, CNC, and other techniques to construct the chassis and parts:

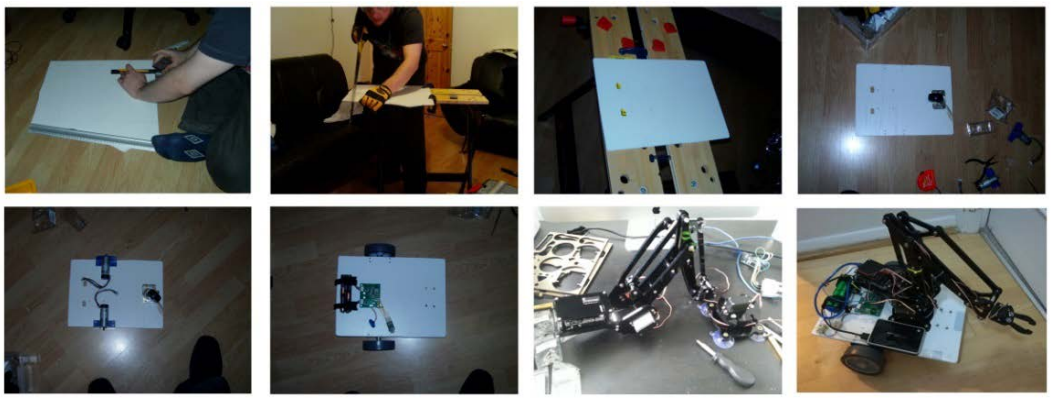


Figure 1.12 – Building ArmBot

I built the robot in *Figure 1.12* from scratch, for the London robotics group *The Aurorans*, in 2009. The robot was known as EeeBot in 2009 since it was intended to be driven by an Eee PC laptop. The Aurorans were a community that met to discuss robotics. The robot was later given a Raspberry Pi, and a robot arm kit (the uArm) seemed to fit, earning it the name **ArmBot**.

In the current market, there are many chassis kits, and a beginner will not need to measure and cut materials in this way to make a functioning robot. These are built to experiment on, and to inspire other robot builders and kids to code. Toward the end of the book, we will cover some of the communities where robots are being built and shared, along with starting points on using construction techniques to make them from scratch.

The television series *Robot Wars* is a well-known competitive robot event with impressive construction and engineering skills. There is no autonomous behavior in Robot Wars, though; they are manually driven like remote control cars. Washing machines, although less exciting, are smarter, so they could be more strictly considered robots.

Summary

In this chapter, we have looked at what the word *robot* means, and the facts and fiction associated with robots. We have defined what a real robot is. You have seen what a machine needs to do in order to be considered a robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've investigated the robots seen in the home and in industry. You've been shown some designed to amaze or travel to other planets. We've also looked at hobbyist and education robots, and how some of these are just built for fun. You've seen some block diagrams of real-world devices that may not have been considered robots. You've also spotted how our homes may already have several robots present.

I hope this chapter has you thinking about what earns the title of *robot*. A washing machine can be fully automatic, starting at some time later, following a program, with some advanced machines saving water by detecting the quality of the water coming out from the clothes as a metric for how clean they are. A machine called a robot, however, could be simply a remote-controlled device, such as telepresence robots or Robot Wars robots. Undoubtedly, all have sophisticated engineering, requiring many similar skills to make them.

While some robots are clearly robots, such as the Honda ASIMO and Baxter, some others are far harder to draw the line at. If the broad concept of a *decision-making, electro-mechanical machine* fits these cases, it would exclude the remote-controlled type. If the concept of *machines that are mobile* is applied, then a toy RC car would be included, while a fully autonomous smart machine that is stationary is excluded. A machine could be made to look robot-like with anthropic (human-like) characteristics, but simply being mechanical, moving an arm up and down – is this a robot? It isn't running a program or reacting to an environment.

Now that we have explored what robots are, let's move on to the next chapter, in which we'll look at how to plan a robot so we can build it.

Assessment

Look around your home. There will be other automatic machines with many of the features of robots in them. Take a common household machine (other than a washing machine), and look at its inputs and outputs. Use these to make a diagram showing them going in or out of a controller. Think about how they move if they move around the house.

Consider further what feedback loops may be present in this system. What is it monitoring? How is it responding to that information?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

Refer to the following links:

- Honda ASIMO: `http://asimo.honda.com/`.
- Baxter at Rethink Robotics: `https://www.rethinkrobotics.com/baxter/`.
- Kismet at MIT: `http://www.ai.mit.edu/projects/humanoid-robotics-group/kismet/kismet.html`.
- The OhBot: `http://www.ohbot.co.uk/`.
- The Mars Science Laboratory at NASA: `https://mars.nasa.gov/msl/`.
- For building a robot arm like the one used in ArmBot, take a look at MeArm: `https://github.com/mimeindustries/MeArm`.
- For more information about my ArmBot design, visit `https://www.youtube.com/watch?v=xY6Oc4_jdmU`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2

Exploring Robot Building Blocks – Code and Electronics

In this chapter, we'll take a robot apart to see its parts and systems. We'll explore the components of a robot, both the software (code, commands, and libraries) and the hardware, and how they go together. When starting to make a robot, it's valuable to think about the parts you want and how they relate to one another. I recommend sketching a plan of your robot—a block diagram as a guide to the connected code and parts, which we will explore in this chapter as well.

In this chapter, we will be covering the following topics:

- Looking at what's inside a robot
- Exploring types of robot components
- Exploring controllers and I/O
- Planning components and code structure
- Planning the physical robot

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

For this chapter, you will require the following:

- Simple drawing materials, such as a pen and paper
- Optional – diagram software such as Draw.io (free at `https://app.diagrams.net`) or Inkscape (free at `https://inkscape.org`)

Looking at what's inside a robot

We can start by looking at a robot as a physical system. In *Figure 2.1*, we can see a simple hobby robot:

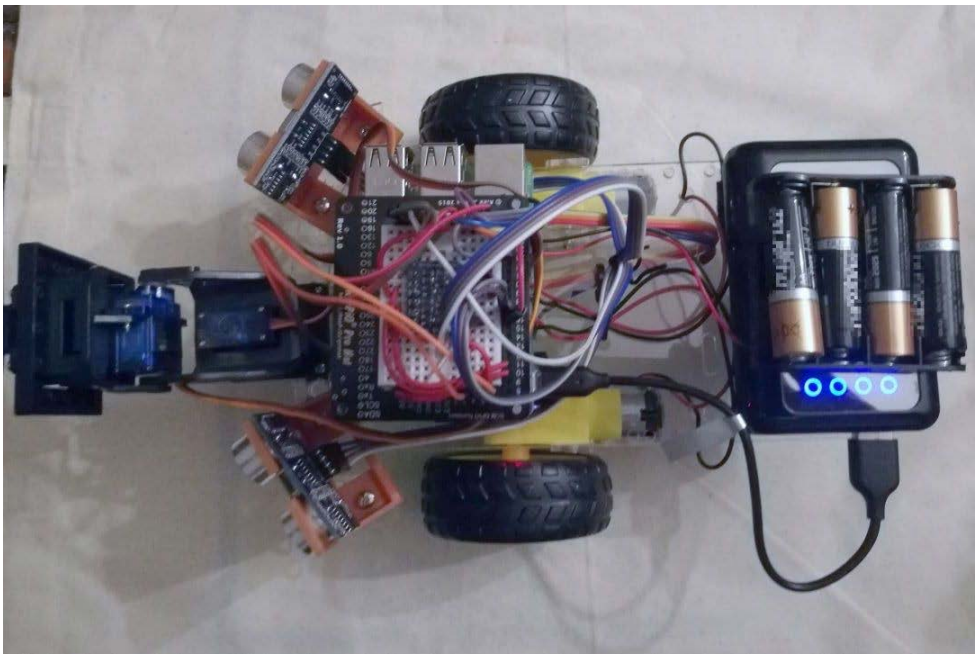


Figure 2.1 – An assembled hobby robot

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 2.2 shows it in its disassembled form:

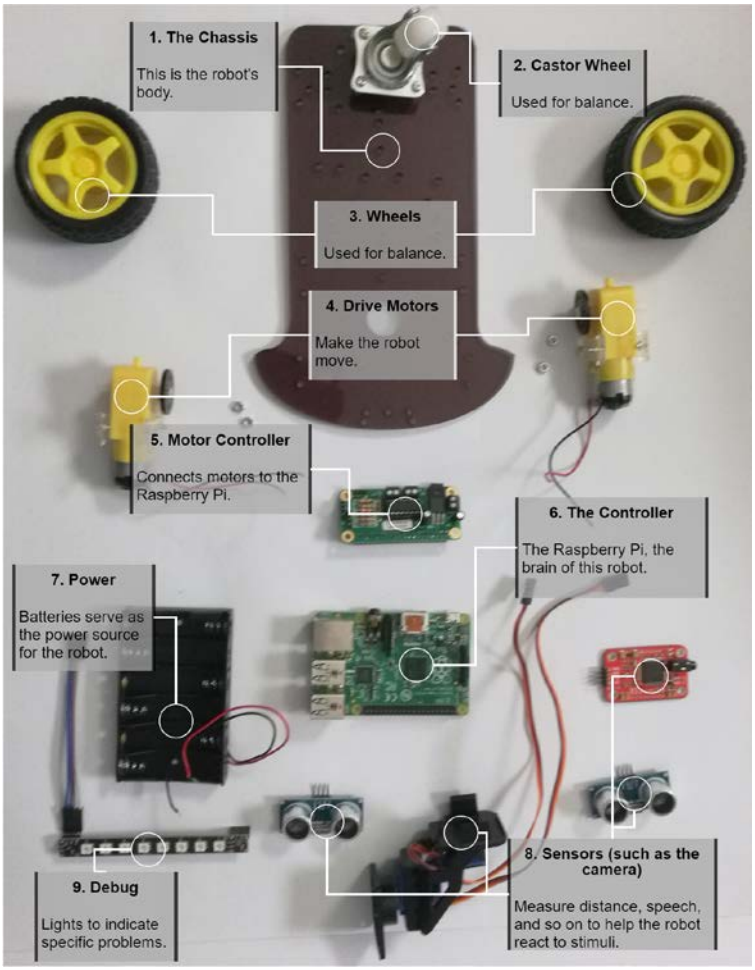


Figure 2.2 – A hobby robot's components laid out

The component groups in *Figure 2.2* include nine types of components:

1. The chassis or body forms the main structure of the robot; other parts are attached here.
2. A castor wheel balances this robot.
3. Two drive wheels. Other robots may use more wheels or legs here.
4. Motors are essential for the robot to move.
5. A motor controller bridges between a controller and connected motors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. A controller, here a Raspberry Pi, runs instructions, takes information from the sensors, and processes this information to drive outputs, such as motors, through the motor controller.
7. All robots must have power, usually one or more sets of batteries.
8. Sensors provide information about the robot's environment or the state of its physical systems.
9. Finally, debug devices are outputs that allow the robot to communicate with humans about what its code is doing, and are also useful for looking good.

We will examine these components in more detail later in this chapter.

We can visualize a robot as a block diagram (*Figure 2.3*) of connected parts. Block diagrams use simple shapes to show a rough idea of how things may be connected:

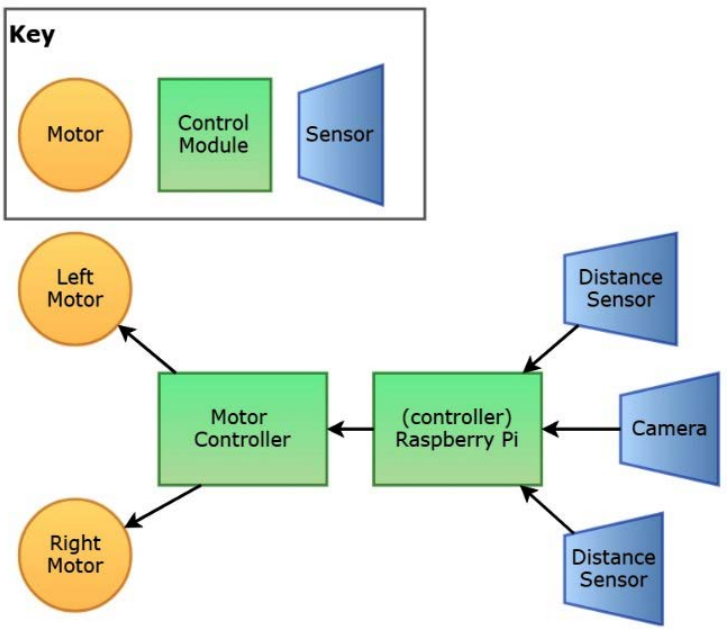


Figure 2.3 – A robot block diagram

The block diagram in *Figure 2.3* does not use a formal notation. The key I've created is off the top of my head, but it should identify sensors, outputs, and controllers. It could be as simple as a sketch on some scrap paper. The critical factor is that you can see blocks of functionality in the hardware, with the high-level flow of data between them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

It is from this diagram that you can develop more detailed plans, plans containing details in terms of electrical connections, power requirements, the hardware, and how much space is needed. Sketching a block diagram about a robot you'd like to create is the first step toward making it.

> **Important note**
>
> A block diagram is *not* a schematic, nor a scale drawing of a finished robot. It doesn't even try to show the actual electronic connections. The picture ignores small details, such as how to signal an ultrasonic distance sensor before it responds. The connection lines give a general idea of the data flow. A block diagram is the right place to show the type and number of motors and sensors, along with additional controllers they may need.

This was a very brief overview of robot components, seeing a robot similar to the one you will build, along with it disassembled into parts. We took a look at a simple robot block diagram and its intent. In the next section, we will take a closer look at each of the robot's components, starting with motors.

Exploring types of robot components

Before we look at the types of motors and sensors, let's get a brief understanding of what each of them is.

A *motor* is an output device that rotates when power is applied. Motors are a subset of a type of machinery called an *actuator*. It is an output device that creates motion from electrical power. This power can be modulated with signals to control movement. Examples of actuators are solenoids, valves, and pneumatic rams.

A *sensor* is a device that provides input to a robot, allowing it to sense its environment. There are more sensor types than a single book can list, so we'll keep to the commonly available and fun-to-use ones. Displays and indicators are debug output devices, for giving feedback on the robot's operation to a human user/programmer. A few of these will be covered in this section.

Now, let's look at them in detail.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Types of motors

There are a number of different kinds of motors that robots commonly use. Let's take a look at what each one does and how we might use them for different types of motion:

> **Important note**
>
> Torque is a rotating/twisting force, for example, the force a motor will need in order to turn a wheel. If the torque increases, a motor will require more power (as current), and will slow down while trying to cope. A motor has a limit, the **stall torque**, at which point it will stop moving.

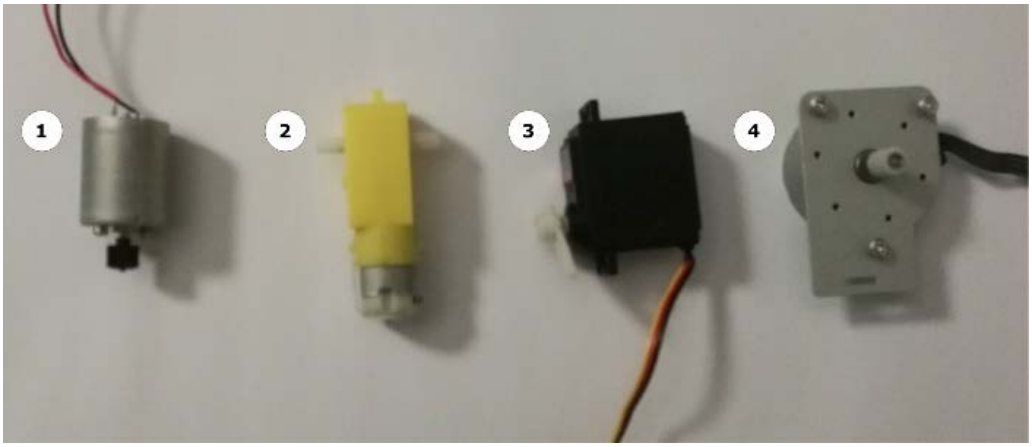


Figure 2.4 – Different motor types – a DC motor, DC gear motor, servo motor, and stepper motor

To identify what each of these motors do, let's look at them in detail:

1. **DC motor**: This is the most simple type of motor in robotics and forms the basis of gear motors. It uses **Direct Current** (**DC**) voltage, which means it can be driven simply by voltage running one way through it. The motor speed is in proportion to the voltage running through it versus the torque required to move. A bare DC motor like the one in *Figure 2.4* can spin too fast to be useful. It will not have much torque and stall easily.
2. **DC gear motor**: This is a DC motor fitted with a gearbox. This gearbox provides a reduction in speed and increases the torque it can handle. This mechanical advantage increases the motor's ability to move a load. Note that this gear motor is missing soldered leads! I recommend these motor types for robot wheels. We will use gear motors such as this on our robot in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*, and *Chapter 7*, *Drive and Turn – Moving Motors with Python*.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. **Servo motor** (or **servomechanism**): This type of motor combines a gear motor with a sensor and a built-in controller as shown in *Figure 2.5*. A signal to a controller states a motor position, and the controller uses feedback from the sensor to try to reach this position. Servo motors are used in pan and tilt mechanisms, along with robot arms and limbs. We will look more closely at, and program, servo motors in *Chapter 10*, *Using Python to Control Servo Motors*:

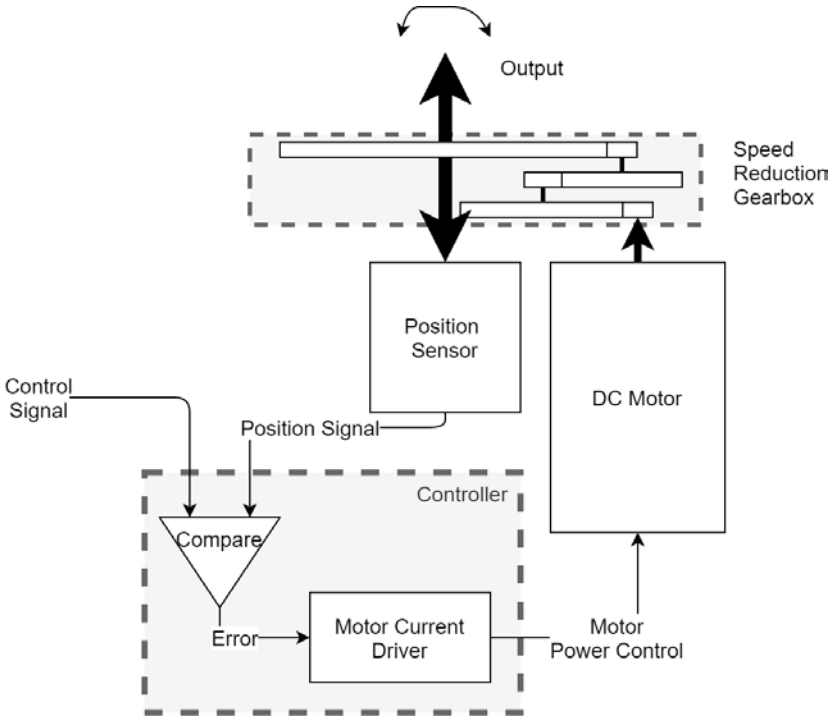


Figure 2.5 – Pictorial diagram of a servo motor mechanism

4. **Stepper motor**: These have coils powered in a sequence to let the motor step a certain number of degrees. Where exact motions are needed, engineers use steppers. Stepper motors tend to be slower and generate a lot of heat compared with DC motors or servo motors. You will find these in fine-control applications, such as 3D printers and high-end robot arms. They are heavier and more expensive than other motors.
5. **Brushless motor**: These are not shown in the diagram. They are driven with specialized controllers, and can be capable of high speed and torque. They run quieter and are popular in drones. There are no gear motor equivalents, so creation of a gearbox may be necessary.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Important note**
>
> All but servo motors require hardware for a controller such as the Raspberry Pi to drive them. This hardware allows the Pi to control power-hungry devices without destroying them. Never connect DC motors, stepper motors, or solenoids directly to a Raspberry Pi!

Let's look at some other types of actuators next.

Other types of actuators

Linear actuators, like those shown in *Figure 2.6*, are devices that convert electrical signals into motion along a single axis. These can be a stepper motor driving a screw in a fixed enclosure, or use arrays of coils and magnets:

Figure 2.6 – Linear actuators: By Rollon91, [Image credit: https://commons.wikimedia.org/wiki/File:Uniline.jpg?uselang=fr [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)]

A **solenoid** is a simple linear actuator using an electromagnetic coil with a metal core that is pulled or pushed away when powered. A common use of this type is in hydraulic or pneumatic valves. Hydraulic and pneumatic systems generate powerful motions like those seen in excavators.

Status indicators – displays, lights, and sounds

Another helpful output device is a display. A single LED (a small electronic light) can indicate the status of some part of the robot. An array of LEDs could show more information and add color. A graphical display can show some text or pictures, like those found on a mobile phone. We will be connecting a multicolor LED strip to the robot as a display in *Chapter 9*, *Programming RGB Strips in Python*.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Speakers and beepers can be used for a robot to communicate with humans by making sounds. The sound output from these can range from simple noises through to speech or playing music.

Many robots don't have any displays and rely on a connected phone or laptop to display their status for them. We will use a phone to control and see the status of our robot in *Chapter 17*, *Controlling the Robot with a Phone and Python*.

Types of sensors

Figure 2.7 shows a collection of sensor types used in robotics. They are similar to those that we will explore and use in this book. Let's examine some of them and their uses. Note that these may look different from the same sensor types seen previously – there is a wide variation in sensors that do the same job. When we add them to the robot, we will cover their variants in more detail:

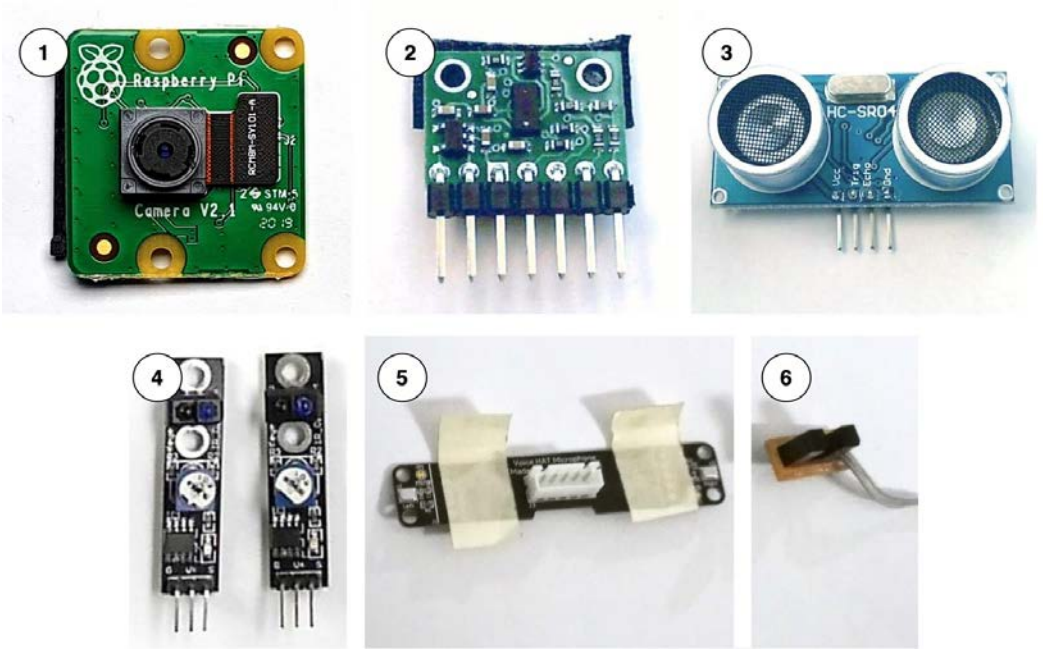


Figure 2.7 – A selection of sensors from my robots: 1 - A Raspberry Pi camera, 2 - an optical distance sensor, 3 - an ultrasonic distance sensor, 4 - line sensors, 5 - microphones, and 6 - an optical interrupt sensor

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's understand each sensor from *Figure 2.7* in detail:

1. **Raspberry Pi camera module**: This module connects to a Raspberry Pi to provide it with imaging capabilities. We'll use it for visual processing programming in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*. This module captures images or video sequences. It can generate a lot of data quickly, which is one of the problems associated with robot vision. It is sensitive to lighting conditions.
2. **Optical distance sensor**: The VL53L0X Time of Flight laser ranging sensor in *Figure 2.7* is a distance sensor. It uses an infrared laser to bounce off objects and detect how far away they are. It can be affected by lighting conditions.

   The VL53L0X sensors use I2C to send a detected range to the Raspberry Pi and can share their two communication pins with many other devices. I2C is useful when you have many sensors and outputs and are starting to run out of places to connect things. I2C sensors can be a more expensive option.
3. **Ultrasonic distance sensor**: The HC-SR04 is another distance/ranging sensor that bounces sound pulses off objects to detect distance. It is affected by the types of material an object is made from and will fail to detect certain surfaces, but is impervious to lighting conditions. Some surfaces, for example, fabrics, absorb the sound too much and never send it back, while other surfaces, such as grids or meshes, do not interact much with sound waves and will be transparent to the sensor.

   The HC-SR04 requires precise timing in the controller to time the echo, which we will have to manage in code. It has a longer range than the VL53L0X laser sensor, and is cheaper, but is also less sensitive at close distances. We will be programming sound-based range sensors in *Chapter 8*, *Programming Distance Sensors with Python*.
4. **Line sensors**: These are a set of three line-sensors that use light to detect transitions from light to dark. They can be adjusted to sense in different conditions. There are a few variations of these modules. These provide an on or off signal, depending on light or dark areas beneath it. They are the simplest of the sensors.
5. **Microphone**: The fifth sensor is a pair of microphones. These can connect directly to the PCM pins on a Pi. Some other microphones need to be connected to electronics to process their signal into something the Raspberry Pi uses. We will use microphones for voice processing in *Chapter 15*, *Voice Communication with a Robot Using Mycroft*.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. **Optical interrupt sensor**: This detects infrared light passing through a gap between two posts, sensing whether something between the posts is interrupting the beam. These are used with notched wheels to detect rotation and speed by counting notches. When used with wheels, they are also known as encoders. We use encoders in *Chapter 11*, *Programming Encoders with Python*.

There are many more sensors, including ones to detect positions of limbs, light, smoke, heat sources, and magnetic fields. These can be used to make more advanced robots and add more exciting behavior.

We have covered motors, displays, indicators, and sensors, together with examples and some details regarding their types. These are the parts that allow our robot to interact with the world. Now we will move on to the controllers, the parts of a robot that run code and connect sensors and motors together.

Exploring controllers and I/O

At the center of the robot block diagram, as in *Figure 2.3*, are controllers. Robots usually have a primary controller, a computer of some kind. They may also have some secondary controllers, and some more unusual robots have many controllers. This book keeps things simple, with your code running on a conventional central controller. The controller connects all the other parts together and forms the basis of their interactions.

Before we look at controllers, we need to get a better understanding of an important component that connects controllers to other components, I/O pins.

I/O pins

I/O pins are used for input and output from the controller. They give the controller its ability to connect to real-world sensors and motors.

The number of I/O pins on the controller is a limiting factor in what you can connect to a robot without using secondary controllers. You may also see the term **General Purpose Input Output** (**GPIO**). Controller I/O pins have different capabilities.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The simplest I/O pins are only able to output or read an on/off signal, as shown in *Figure 2.8*. These are known as digital I/O pins. They can be programmed to perform complicated tasks through signal timing. This is the principle used in the HC-SR04 distance sensor. In *Figure 2.8*, this graph represents a voltage level over time. So, as we move along the *x* axis, the voltage is on the *y* axis. The upper level represents a digital logic high (`1`, `True`, `On`). The lower level represents a digital logic low (`0`, `False`, `Off`). The controller will attempt to interpret any value as high or low:

Figure 2.8 – A digital signal

Analog input pins can read varying levels, like the signal in *Figure 2.9*, which is another voltage-over-time graph. If a sensor produces a changing resistance or continuous scale of values, then an analog pin is suitable. There is a resolution limit to this, for example, an 8 bit analog input will read 256 possible values:

Figure 2.9 – An analog signal

Pulse Width Modulation (**PWM**) pins output a cycling digital waveform shown in *Figure 2.10*. This diagram also shows voltage over time, although the timing of the pulses represents a continuous level, so the dashed line shows the continuous level produced by the timing. PWM outputs allow the code to select the frequency and how much time they are on for. The length of on-time versus off-time in a cycle changes to vary an output signal. This is often used to control the speed of motors:

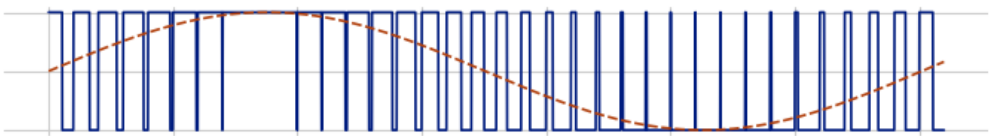


Figure 2.10 – A PWM signal in blue, with the dashed line showing its approximate value

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will spend more time on PWM pins in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*, and *Chapter 7*, *Drive and Turn – Moving Motors with Python*.

Some I/O pins can be used to form data transmission lines, such as serial, I2S, I2C, and SPI buses. They are known as data buses. Data buses are used to send data to or from other controllers and intelligent sensors. We'll use an SPI data bus for the RGB LEDs in *Chapter 9*, *Programming RGB Strips in Python*.

Microcontroller pins can be used for digital or analog input and output, or part of a data bus. Many controllers allow the usage mode of pins to be configured in the software you run on them, but some capabilities are restricted to specific pins.

Controllers

Although it's possible to use bare microcontroller chips with the right skills to create surrounding electronics and your own PCBs, we'll keep things simple in this book by using controller modules. These tend to come in packaged and easy-to-use systems:

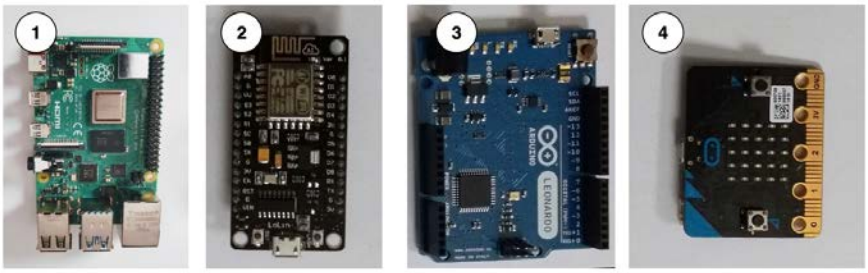


Figure 2.11 – A selection of controller modules: a Raspberry Pi, NodeMCU, Arduino, and micro:bit

Figure 2.11 shows some of my favorite controllers. They can all be powered via a USB connection. All but the Raspberry Pi can also be programmed over a USB. They all have connectors for easy access to their I/O pins. For each of the controllers, let's see what they are, along with their pros and cons:

1. **Raspberry Pi**: This is powerful enough for visual processing. It tends to consume a little more power and is more expensive, but is similar to a mobile phone in capability. It has the most flexible environment for programming. There are several models to consider. They have many I/O pins, but none are analog input pins.
2. **NodeMCU**: This is based on the ESP8266 controller. This controller has built-in Wi-Fi and can be programmed with Arduino C++, MicroPython, or Lua. It has plenty of I/O pins, but only one can read analog signals. It supports many data bus types. It is somewhat faster and can hold larger programs than the Arduino. It is the cheapest controller in this lineup.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. **Arduino Leonardo**: This is based on the Atmega 328 chip. Arduino controller modules formed the basis of most of my robots around 2010-2012. The Arduino was important for the ease with which it could be connected to a PC via a USB and programmed to immediately interact with devices attached to its I/O pins.

   The Arduino is mostly programmed in the C++ language. It has the most flexible built-in I/O pins – seven analog pins, many digital pins, PWM output pins, and can be set up to handle most data buses. The Arduino's processor is very simple; it is not capable of visual or speech processing tasks. The Arduino has the lowest power consumption of all the options shown here.

4. **micro:bit**: This was released in 2015 for use in education, and is ideal for children. Its use in robotics requires an additional adapter if you need more than the 3 I/O pins that it ships with, but it is still a pretty capable robot controller and comes with a handy built-in LED matrix. This can be programmed in MicroPython, C, JavaScript, and several other languages.

An honorable mention should go to the PIC microcontroller, not pictured here. These were used for hobby robotics long before any of the others, and have a thriving community.

Here is a comparison of controllers based on the pros and cons:

Controller Name	Pros	Cons
Arduino	Very low power consumption, very flexible I/O.	Quite large, least capable processor, not suitable for visual processing or speech. Limited programming environments.
Micro:bit	LED matrix, easy to program.	Needs external adapters for connections, not suitable for visual processing or speech. Limited I/O capabilities.
ESP8266/ NodeMCU	Many programming languages, onboard Wi-Fi, cheap, flexible I/O, can be very small.	Not suitable for visual processing or speech. Only a single analog input pin.
Raspberry Pi	Powerful. Easy access to I/O pins. Runs a full Linux system with a selection of many languages. Suitable for visual processing and speech recognition. Wi-Fi and Bluetooth.	External devices needed for analog input. It can be more expensive and requires more power.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Where the other controllers may run a simple interpreter or compiled code, the Raspberry Pi runs a complete operating system. Current models have Wi-Fi and Bluetooth capabilities, which we will use to make a robot headless and connect with game controllers.

Choosing a Raspberry Pi

Figure 2.12 shows a few current Raspberry Pi models. As new Raspberry Pis are released, a robot builder may have to adapt this to the latest version. All these models have Wi-Fi and Bluetooth capabilities. The Raspberry Pi I/O pins support many of the data bus types and digital I/O. External controllers are needed for analog reading and some other I/O functions:

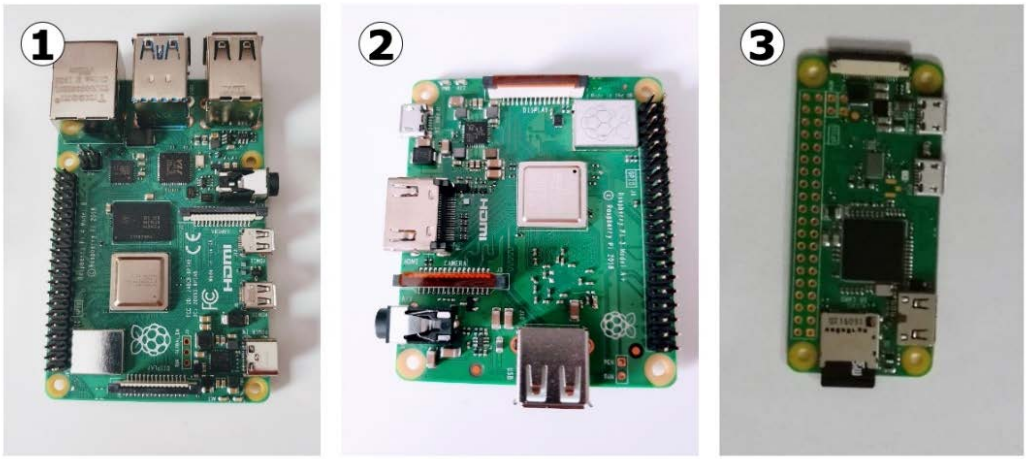


Figure 2.12 – Raspberry Pi models – 4B, 3A+, and Zero W

Let's look at each of these models in a little detail:

1. **Raspberry Pi 4B**: This is the latest in the Raspberry Pi line at the time of writing. As the latest model, it is the fastest and most potent in the lineup. It takes up more space, is the most expensive in this group, and uses the most power.
2. **Raspberry Pi 3A+**: This is the controller we will use for our robot. It provides an excellent compromise on size and power. It is fully capable of visual processing through a camera. It's not quite as fast as the 4B+, but definitely quick enough for our purposes.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. **Raspberry Pi Zero W**: This is an inexpensive, lighter alternative to the other Raspberry Pi models. Cameras and speakers are still supported. The Zero WH model includes headers for I/O too. It performs speech and visual recognition slower than on a Raspberry Pi 3 and 4. Their small size makes them an interesting option for a remote-control pad too.

Now that we know each of the models, let's compare their pros and cons:

Controller Name	Pros	Cons
Raspberry Pi 4B	Fastest Pi, largest memory. It will perform visual processing very quickly. Has four USB ports (including two high-speed USB 3 ports) and HDMI.	Can get very warm, drains batteries fast. The most expensive option. Largest board and heaviest.
Raspberry Pi 3A+	Significantly faster than the Zero W. Uses less battery power, and has a lower weight and smaller size than the 4B. Cheaper than the 4B. Has one USB connection and HDMI.	Larger, and heavier than the Zero W. Slower and less memory than the 4B.
Raspberry Pi Zero W	Small and very cheap. The lightest model. Lowest power consumption.	The slowest Pi here. Headers need to be soldered (unless opting for the WH model). Need adaptors to use USB OTG and HDMI ports. A smaller camera CSI port.

The Raspberry Pi 4B may be the most powerful, but the 3A+ is powerful enough to be responsive to all the activities here.

Planning components and code structure

You've now briefly seen some components you might use in a robot, and you've encountered a block diagram to put them together. This is where you may start taking the next step and thinking further about how to connect things, and how the code you write for them will be structured.

Code is easier to reason about when taken as logical blocks instead of one large lump. Arranging code in ways that are similar to a hardware functionality diagram will help navigate your way around as it becomes more complicated.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

So, let's return to the robot block diagram in *Figure 2.3* to think about what we'll need to handle in our code for it. That diagram has three sensors and two outputs. Each component (sensor, output, and controller board) may need some code to deal with it, and then you need some code for the behavior of combined modules.

Motor controllers come in many flavors. They have different ways to output to motors, and they may have monitoring for battery levels. Some smart motor controllers interface with wheel encoders directly to ensure the wheels have traveled a specified amount. When we write behavior for a robot, we may not want to rewrite it if we change the motor controller. Mixing the direct motor controller code with the behavior code also makes it harder to reason about. For this, I recommend creating an interface layer, an *abstraction* between the real motor controller code and a standard interface, which will make swapping components possible. We will see this in practice in *Chapter 7*, *Drive and Turn – Moving Motors with Python*.

This is similar for each sensor. They will have some code to manage how they get signals and turn them into usable data. All these devices may have setup and teardown code that needs to run when starting or stopping behavior that connects to them. The camera is a sophisticated example of this, requiring processing to get the data values we can use to perform a task:

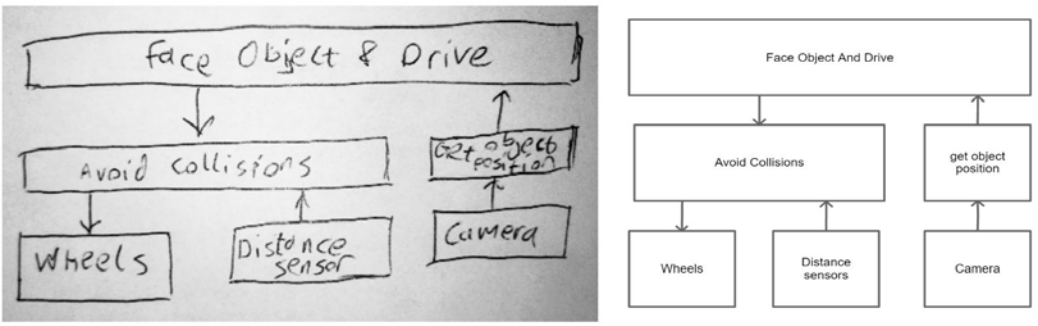


Figure 2.13 – A quick software block diagram, in pen on an envelope, and the same diagram using a computer

Just like the hardware, a simple diagram can represent the software. This can be made in a drawing program or sketched on any paper you have to hand. In *Figure 2.13*, I've deliberately chosen a hand-drawn one so you don't feel that you need a drawing tool to do this. This won't be tidy, but it's quickly redrawn, and can even be done on the back of receipt paper if an idea comes to you while out dining. What is relevant here is the fact that if you use a pencil, go back over it in a pen or fine liner so that it doesn't fade. To make it clearer to the reader, I have made a computer drawing too, but don't feel you need to do this.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Tip**
> Scan your hand-drawn documents. If you have a scanner, or just a phone, I recommend scanning or getting a photo of your sketches, for later reference. Putting them into software such as Evernote or OneNote as images/PDFs with useful tags lets you look them up quickly later.

After making a hand-drawn sketch, you can use a software tool. This will take longer than a hand-drawn version, and try not to be distracted by the quirks and styling of a tool.

In terms of the design itself, this is still a very simplistic view. The **Wheels** box will be a block of code, dealing with asking the wheel motor controller to do things. This may sit on top of code written by the motor controller company, or use I/O pins connected to the controller.

Distance sensors are blocks of code to read distances from the sensors, triggering them when necessary. We will look at two different kinds of sensors and compare them. By having a block of code like this, changing the sensors at this level means the other code won't have to change.

There is also a block of code for the **Camera**, doing fiddly stuff like setting it up, resolution, white balancing, and other parts that we will cover. On top of this is a layer that will use the camera images. That layer could get the position of a colored object, returning this position to the layer above.

Across the motors and distance sensors is a behavior layer that allows the robot to avoid collisions, perhaps when it is below a threshold on one side. This will override other behavior to turn away from that obstacle and drive off a bit.

The top layer is another behavior that takes positional data from the **Get Object Position** code. Use this position to choose a direction, and then instruct the motors to drive to the object. If this behavior goes through the **Avoid Collisions** behavior, there could be a complicated interaction that leads the robot to seek the correct object, while avoiding obstacles and going around things. It will also not come close enough to the detected object to collide with it.

Each module is relatively simple, perhaps with the lower layers that are closer to the hardware being more complex, especially in the case of the camera.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Breaking the code down into blocks like these means that you can approach a single block at a time, test, and tweak its behavior, and then focus on another one. When you have written blocks like this, you can reuse them. You will likely need the motor code multiple times and now will not need to write it multiple times.

Using blocks to describe our software lets us implement the blocks and their interactions in different ways. We can consider whether we will use functions, classes, or services for these blocks. I will spend more time on this as we start writing the code for this and show the different approaches.

Planning the physical robot

Let's now put all of this to use and plan the layout of the physical parts of robot that we are making in this book. While we go through chapters, we will be adding new components each time, and keeping an overall map in our minds as we go helps us to see where we are. It is quite exciting to start to picture all the things a robot will do. Let's start with a list of what our robot will do and be:

- It will have wheels and be able to drive around the floor.
- It will have a Raspberry Pi 3A+ controller.
- It will have a motor controller for the wheels.
- It will be able to indicate its status with a set of multicolored LEDs.
- The robot will use a pair of servo motors for a pan and tilt mechanism.
- It will be able to avoid walls and navigate around obstacles with either ultrasonic or laser distance sensors.
- It will have an encoder per wheel to know how far it has moved.
- The robot will use a camera to sense colored objects or faces.
- It will be able to follow lines with the camera.
- The robot will have a microphone and speaker to work with voice commands.
- It will have a gamepad as a remote control.
- It will need power for all of these things.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Phew! That is a lot of functionality. Now, we need to draw the hardware blocks. *Figure 2.14* shows our block diagram. While done with Draw.io, a simple back-of-an-envelope sketch of a block diagram is an excellent start to robot planning. Most of my robots start off that way:

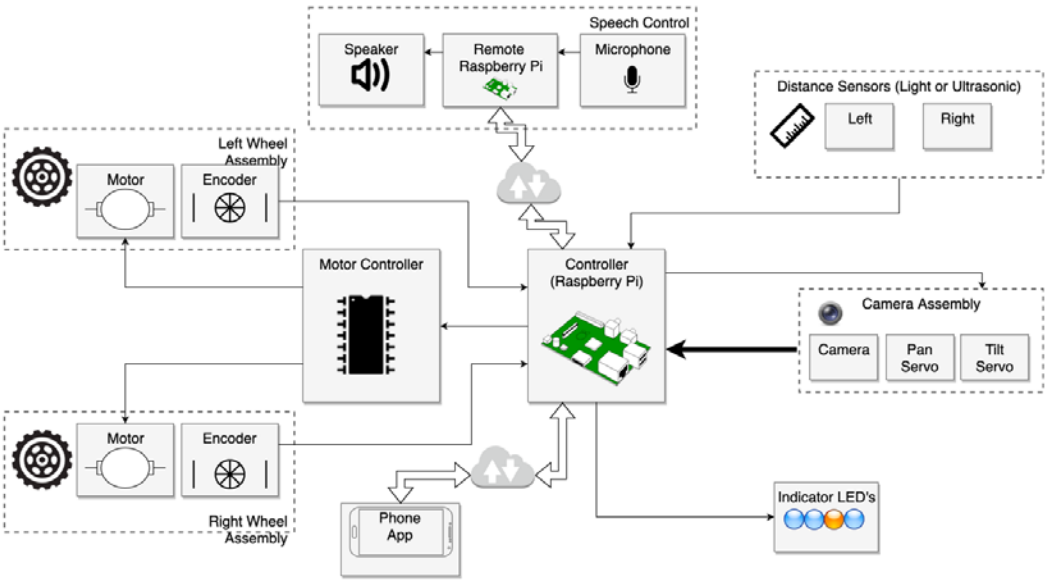


Figure 2.14 – Block diagram of the robot we will build, created using the draw.io web app

Although this looks like a daunting amount of robot, we will be focusing on an area of functionality in each chapter and building it before moving to other areas. The annotation here is not any formal notation, it is just a way of merely visualizing all the parts that will need to be connected. Along with this, I usually sketch roughly where I would physically place sensors and parts with one another:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

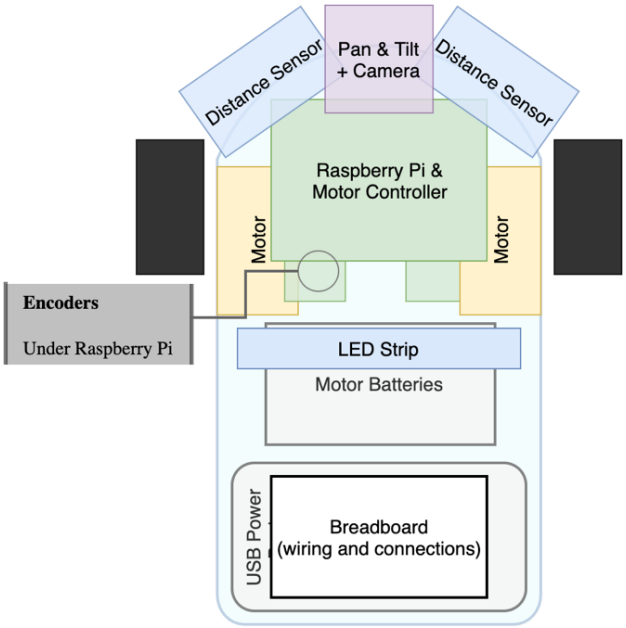


Figure 2.15 – An overview of how the robot could be physically laid out, created with Draw.io

The sketch in *Figure 2.15* is not exhaustive, accurate, or to scale, but just an idea of where I want the parts to end up. Note the following things in this diagram:

- Sensors have a clear field of view, and the distance sensors are pointing out to the sides. I'll go into more detail in the relevant sensor chapters on why this is important.
- Encoders are placed over the wheels where they will be used.
- Heavy items, specifically batteries, should be kept low (below the center of gravity) to avoid a robot tipping over.
- Batteries need to be changed, so think about access to them.
- Try to keep components that are directly connected quite close to one another.
- This is a rough plan. It need not be this detailed, and this is *not* the test fit. Real dimensions, design compromises, and hitches will mean that this will change. This is just a starting point.

As we work through the book, we will look at the details in these diagrams, and start to flesh out the real robot, making some of this less fuzzy. Any diagram like this, at the start of a project, should be taken as a bit rough. It is not to scale and should not be followed blindly. It is a guide, or a quick map from which to start working.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Summary

In this chapter, you've been able to see a number of the different component parts that go into a robot, and through a block diagram as a plan, start to visualize how you'd combine those blocks to make a whole robot. You've seen how you can quickly sketch your robot ideas on an envelope, and that drawing tools on a computer can be used for a neater version of the same diagram. You've had a quick tour of motors, sensors, and controllers, along with a few ways, including analog, digital, PWM, and data buses, for controllers to communicate with the other devices connected to them. Following on from this, you've seen a plan of the robot we will build in this book.

In the next chapter, we will look at Raspbian, the operating system used on the Raspberry Pi in our robot, and start configuring it.

Exercise

1. Try creating a block diagram for a different robot, thinking about inputs, outputs, and controllers.
2. Are the Raspberry Pi 4B and 3A+ still the most recent versions? Would you use another model, and what would be the trade-offs?
3. What are the drawbacks of the laser ranging sensor versus the ultrasonic distance sensor?
4. Try drawing an approximate physical layout diagram for a different type of robot with a different controller.

Further reading

- *Raspberry Pi Sensors*, *Rushi Gajjar*, *Packt Publishing*: Learn to integrate sensors into your Raspberry Pi projects and let your powerful microcomputer interact with the physical world.
- *Make Sensors: A Hands-On Primer for Monitoring the Real World with Arduino and Raspberry Pi*, *Tero Karvinen*, *Kimmo Karvinen*, *Ville Valtokari*, *Maker Media, Inc.*: Learn to use sensors to connect a Raspberry Pi or Arduino controller with the real world.
- *Make: Electronics: Learning Through Discovery*, *Charles Platt*, *Make Community, LLC*: This is a useful resource if you want to find out more about electronic components and dive deeper into the individual components.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3
Exploring the Raspberry Pi

In the previous chapter, we saw the Raspberry Pi in the deconstruction of a robot. It's no surprise, then, that we'll build a robot using the **Raspberry Pi**.

In this chapter, we will be using the **Raspberry Pi 3A+** as a **controller**. We'll look at various options when examining this choice, and look at features such as the connections on the Raspberry Pi and how we will use them to understand our decision. We'll move on to exploring **Raspberry Pi OS**, and will finish by preparing the OS for use on the Raspberry Pi.

The following topics will be covered in this chapter:

- Exploring the Raspberry Pi's capabilities
- Choosing the connections
- What is Raspberry Pi OS?
- Preparing an SD card with Raspberry Pi OS

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

For this chapter, you will need the following:

- A Micro SD card storing 16 GB or more
- A Raspberry Pi 3A+
- A Windows, Linux, or macOS computer or laptop connected to the internet and able to read/write to SD cards

Check out the following video to see the code in action: `https://bit.ly/3bBJQt9`.

Exploring the Raspberry Pi's capabilities

As we saw in *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*, the controllers used for a robot can be one of the most critical choices you make. This will determine what kinds of inputs and outputs you have, what the power requirements of your electronics will be, what types of sensors you will be able to use, and what code you will run. Changing a controller could mean rewriting the code, redesigning where the controller would fit, and changing the power requirements.

The Raspberry Pi is a range of small computers designed for use in education. Having I/O pins for connecting it to custom hardware, while being a complete computer, makes it a favorite of makers (a term for people who like to make things for a hobby, like robots and gadgets). This is helped by the relatively cheap cost and small size of a microcontroller compared to standard computing devices. All Raspberry Pi models have abilities including attaching a camera, display, and keyboard, as well as some kind of networking.

Speed and power

The Raspberry Pi is powerful enough to handle some visual processing tasks, such as facial recognition and tracking objects, with later models being able to perform this faster. The same can be said for voice recognition tasks too. It is for this reason that the faster 4B, 3B+, and 3A+ models are recommended. The Zero and Zero W models are much slower, and although the system will still work, the speed may be frustrating.

The Raspberry Pi is a **Single-Board Computer** (**SBC**) that is powerful enough to run a complete computer OS, specifically versions of **Linux**. We will explore this in the *What is Raspberry Pi OS?* section, but this allows us to use **Python** to perform visual processing and voice processing using libraries and tools that are well maintained by others. Microcontrollers, such as the **Arduino**, **Esp8266**, and **micro:bit**, simply do not have the capabilities to perform these tasks.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Some alternative SBCs that are usable as controllers run Linux, such as the **BeagleBone**, **CHIP**, **OnionIoT**, and **Gumstix Linux** computers, but these are either more costly than the Raspberry Pi or less capable. Only some come with camera integration. Although the BeagleBone has superior analog IO connectivity, the Raspberry Pi 3A+ is more of an all-rounder and has many options to extend it.

Connectivity and networking

The Raspberry Pi 3A+ comes with USB ports and HDMI ports too. We don't plan on using them in this book, although they are handy for debugging if things go wrong and you lose contact with a robot. With that in mind, having an additional screen and keyboard available is recommended.

The Raspberry Pi 4, 3, and Zero W series all have Wi-Fi and Bluetooth onboard. Throughout this book, we will be using Wi-Fi to connect to the robot, so we recommend a model that has this. Wi-Fi can be used to program the robot, drive it, and start code running on it.

The Raspberry Pi has I/O pins to allow you to connect it to the sensors. In the Raspberry Pi 3A+, the **General Purpose Input/Output** (**GPIO**) connections are ready to use, due to having the pins (known as headers) already soldered in place. The Raspberry Pi Zero and Zero W models come without the headers attached. The first Raspberry Pi boards also had different I/O connectors. These reasons make the 3 and 4 series Raspberry Pi the best choice.

Picking the Raspberry Pi 3A+

Putting all this together, the Raspberry Pi 3A+ is a complete computer. The following list of features meets all our needs:

- I/O
- A connector for a camera
- Capable of visual and speech processing
- Onboard Wi-Fi and Bluetooth
- Runs Python code
- Pre-soldered headers ready for connecting to robot devices
- Small and relatively cheap

In addition to this, the 3A+ has a quad-core ARM-based CPU running at 1.4 GHz, which will be enough for our use case. Later Raspberry Pi versions may supersede this model with faster processing and additional capabilities.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Choosing the connections

When building the robot, we will use a subset of the connections the Raspberry Pi has to offer. Let's take a look at what those connections are and how we will use them. As we connect sensors and parts to the Raspberry Pi, we will cover the connections in detail, so do not feel like you need to memorize these now. However, the following pin diagram can serve as a reference for these connections.

In *Figure 3.1*, the highlighted areas show the connections in use:

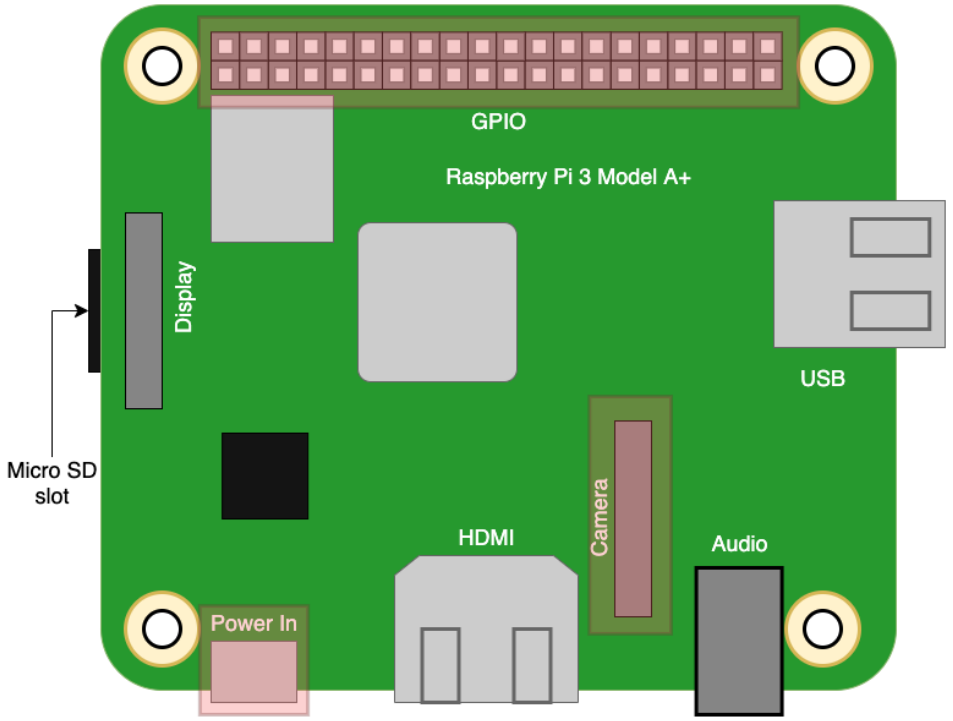


Figure 3.1 – Raspberry Pi connections

First, we will be using the power connector, labeled **Power In** and located at the bottom left of the diagram. This plugs in via a micro-USB connector similar to that on many phones. We will use this while learning to go headless, and this is one of the options for powering a robot. We can plug USB battery packs into this port if they can provide the correct amount of power. Raspberry Pi recommends 2.5 A power supplies, although anything over 2 A will usually suffice.

The lower-middle highlighted port is the camera (**Camera Serial Interface** (**CSI**)) port; this is for the **Pi camera**, which we will attach when preparing to do visual processing.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will be using the Micro SD card slot under the Raspberry Pi to run our code. We will not be using Ethernet or HDMI, as we will be talking to the Raspberry Pi via Wi-Fi. The large connector across the top of *Figure 3.1* is the GPIO port:

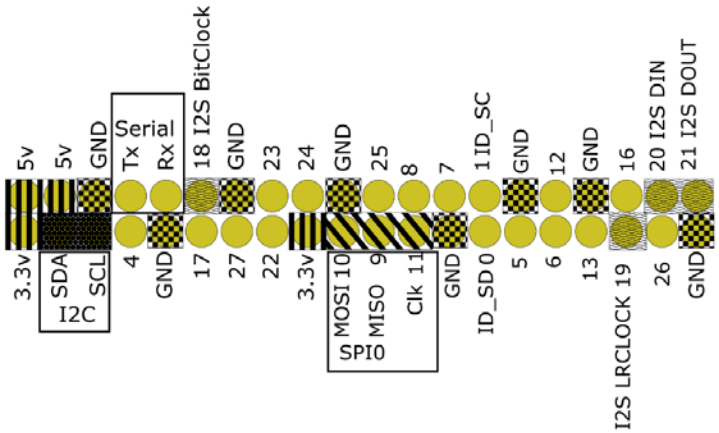


Figure 3.2 – The Raspberry Pi GPIO port (B+, 2, 3, 3B+, Zero, and Zero W)

Figure 3.2 shows a close-up of the GPIO port with the names and uses of some of the pins. This is where we will connect most of our sensors and motors. External devices can be attached to **SPI**, **I2C**, **Serial**, and **I2S** data buses, or to digital I/O pins.

Power pins

The 5 V and 3.3 V pins are used for power, along with the pins marked **GND**. GND is an abbreviation of **ground**, which is the equivalent of a minus terminal on a battery or power supply. The 5 V pin can be used to supply the Pi with power from batteries. 5 V and 3.3 V can be used to supply small electronics or sensors.

Data buses

SPI, I2C, and Serial are used to send control and sensor data between a controller and smart devices. I2S is used to carry encoded digital audio signals (**PCM**) to and from the Raspberry Pi. The ports for these data buses can be enabled through configuration, or the pins can be used as general digital pins when the data buses are turned off.

The pins marked **SDA** and **SCL** are an I2C data bus. We use this for sensors and motor control boards. Instructions are sent over this port.

Pins **9**, **10**, and **11** form the SPI port, which we use to drive RGB LEDs.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Although there is an audio port on the Raspberry Pi, this is not really suitable for driving a speaker, so we will be using the I2S pins on the GPIO port for this. The I2S pins are **18**, **19**, **20**, and **21**. As they also have pins for audio input, we use this for voice processing.

General IO

The other pins that are numbered, without a specific word or shading type, are general-purpose I/O pins. General I/O pins are used for digital inputs and outputs with servo motors, encoders, and ultrasonic sensors.

> **Important note**
>
> Why are the numbers mixed up? The numbers used in most Raspberry Pi documentation are BCM numbers, which correspond to pins on the main Broadcom chip. Use *Figure 3.2* for reference.

Raspberry Pi HATs

Raspberry Pi HATs (also named Bonnets) are circuit boards designed to plug into the GPIO header and conveniently connect the Raspberry Pi to devices such as motors or sensors.

Some boards carry through GPIO pins for further boards/connections to use them, and others will need extender boards to gain access to the pins.

HATs use GPIO pins for different purposes; for example, audio HATs will use the I2S pins for audio interfacing, but some motor controller HATs use the same pins to control motors instead. Using these HATs together can be problematic, so be aware of this when using multiple HATs or specific buses. We will explore this more in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*, when we choose a motor controller.

What is Raspberry Pi OS?

Raspberry Pi OS is the choice of software we use to drive the Raspberry Pi, an OS that our code will run in. It is the Raspberry Pi Foundation's official OS and comes with software prepared to make working with the Raspberry Pi easier. Raspberry Pi OS can support a full desktop or a minimal command line and network-only system.

Raspberry Pi OS is based on the **Debian** Linux distribution. Debian is a collection of software set up to run together, giving lots of functionality and many possibilities. Linux distributions like this are the basis of many internet servers, mobile phones, and other devices. The OS's software is optimized for the Raspberry Pi hardware, namely the kernel and drivers, which are made specifically for it. It also has some neat ways to configure the specialized features that Raspberry Pi users might need.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will use it in a more minimal way than a desktop, forgoing the keyboard, mouse, and monitor support. This minimal version is known as **Raspberry Pi OS Lite** because it is a much smaller download when desktop software is not required, and it uses less space on the micro SD card. Not running a window manager frees up memory and uses less of the processing power of the Raspberry Pi, keeping it free for activities such as visual processing. We will extend Raspberry Pi OS Lite with the software and tools we will use to program our robot.

As you work through the book, you will mostly interact with the robot through code and the command line. Linux and Raspberry Pi OS are written with command-line usage over a network in mind, which is a good fit for the headless nature of programming a robot.

We use Linux's strong support for the Python programming language and the network tools that Linux provides. Raspberry Pi OS is widely used in the Raspberry Pi community and is among the easiest to find answers for when help is needed. It is not the only OS for the Pi, but it is the most useful choice for someone starting on the Raspberry Pi.

Preparing an SD card with Raspberry Pi OS

To use Raspberry Pi OS on a Raspberry Pi, you need to put the software onto a micro SD card in a way that means the Raspberry Pi can load it.

Raspberry Pi has created the Raspberry Pi Imager to put software onto an SD card. Let's download it and get the right image on our card:

1. Visit the Raspberry Pi software downloads page at `raspberrypi.org/software`, and select the **Download for** button for your computer, as shown in the following screenshot:

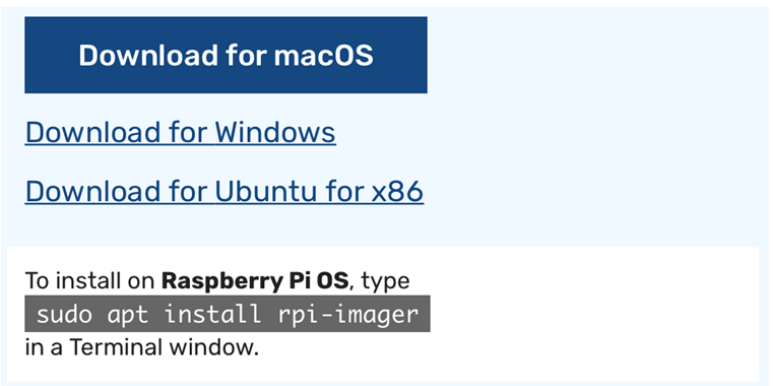


Figure 3.3 – Downloading the Raspberry Pi Imager

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 3.3 shows what this will look like; it should highlight the correct download button for your computer.

2. Install this using the instructions from Raspberry Pi.
3. Insert your micro SD card into the correct port on your laptop. You may need an adaptor.
4. Launch the Imager. We'll start here by choosing the OS. Select the **CHOOSE OS** button:

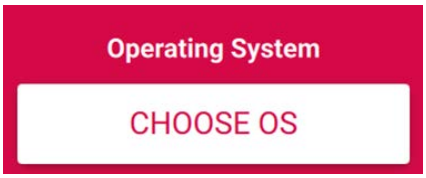


Figure 3.4 – The CHOOSE OS button

Figure 3.4 shows the **CHOOSE OS** button, found in the lower right of the Imager screen.

5. When you select this button, it will bring up a list of OSes to flash on the card:

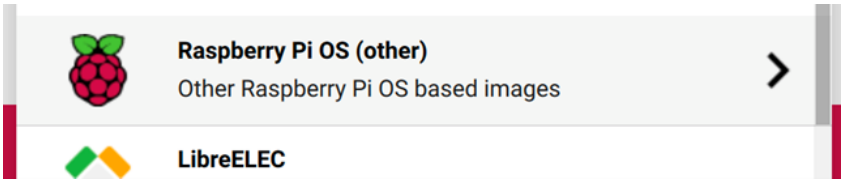


Figure 3.5 – The OS list

Figure 3.5 shows the list of OSes offered by the image. Choose **Raspberry Pi OS (other)**.

6. Under the other menu, there is a further selection of Raspberry Pi OS flavors:

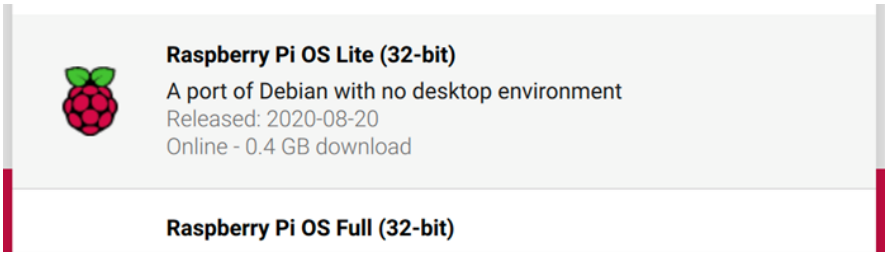


Figure 3.6 – The Raspberry Pi OS selection screen

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Because we are trying to keep things minimal, select **Raspberry Pi OS Lite (32-bit)** from this menu.

7. You should now click **CHOOSE SD CARD**:

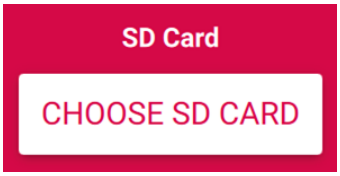


Figure 3.7 – Choosing an SD card

8. This will pop up a list of SD cards, which should show the card you are using:

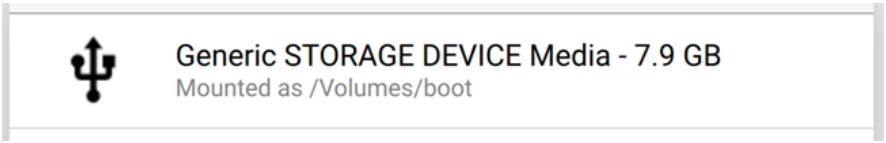


Figure 3.8 – The SD card selection

Select this to continue.

9. You are now ready to write this. Click the **WRITE** button:

Figure 3.9 – The WRITE button

10. It will ask you if you are sure here; click **YES** to continue. It will take some time to download and write the image.

You can load this onto a Raspberry Pi with a screen and keyboard, but before we can use this Raspberry Pi for a robot, we'll make changes to the SD card on your computer in the next chapter.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Summary

In this chapter, you've seen more of what the Raspberry Pi is, and which connections on the Raspberry Pi we will use.

We've learned about the Raspberry Pi OS, which is derived from Linux, how to download it, and how to put this software onto a micro SD card for use in the Raspberry Pi.

In the next chapter, we will make this card headless so that we do not need a screen, keyboard, or mouse to use this Raspberry Pi and contact it from our computer.

Assessment

1. I've recommended a Raspberry Pi 3A+. There are likely to be new models of the Raspberry Pi not considered. What would be their trade-offs? Think about cost, size, power consumption, and computing speed.
2. Try other Raspberry Pi OS or Raspberry Pi distributions; some will need a keyboard and mouse. Be sure to return to Raspberry Pi OS Lite before carrying on in the book.
3. I've mentioned the camera (CSI) connector, power, and GPIO ports. Take a look at the other ports on the Raspberry Pi, and perhaps see what they can be used for.

Further reading

Refer to the following links:

- The Raspberry Pi Foundation guide to installing Raspberry Pi operating systems: `https://www.raspberrypi.org/documentation/installation/installing-images/README.md`.
- *Raspberry Pi By Example*, *Ashwin Pajankar* and *Arush Kakkar*, *Packt Publishing*, which has a section on alternative OSes for a Raspberry Pi, along with many exciting Raspberry Pi projects.
- Raspberry Pi GPIO pinout (`https://pinout.xyz/`): This describes how different boards are connected to the Raspberry Pi in terms of the pins they actually use. It's useful to know that most boards only use a subset of these pins.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4

Preparing a Headless Raspberry Pi for a Robot

In this chapter, you will learn why the Raspberry Pi controller on a robot should be wireless and headless, what headless means, and why it's useful in robotics. You will see how to set up a Raspberry Pi directly as a headless device, and how to connect to this Raspberry Pi once on the network, and then send your first instructions to it. By the end of the chapter, you will have your own ready-to-use Raspberry Pi without needing to connect a screen, keyboard, or wired network to it, so it can be mobile in a robot.

We'll cover the following topics in this chapter:

- What is a headless system and why is it useful in a robot?
- Setting up Wi-Fi on the Raspberry Pi and enabling SSH
- Finding your Raspberry Pi on the network
- Using PuTTY or SSH to connect to your Raspberry Pi
- Configuring Raspberry Pi OS

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

To complete the exercises in this chapter, you will require the following:

- A Raspberry Pi, preferably a 3A+ (but a Pi 3 or 4 will do)
- A USB power supply capable of 2.1 amps with a Micro-USB cable
- The MicroSD card you prepared in the previous chapter
- A Windows, Linux, or macOS computer connected to the internet and able to read/write to SD cards
- A text editor on your computer – VS Code is a suitable multiplatform option
- PuTTY software on Windows (SSH software is already available on Mac and Linux desktops)

The GitHub link for the code is as follows:

`https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter4`

Check out the following video to see the Code in Action: `https://bit.ly/3bErI1I`

What is a headless system, and why is it useful in a robot?

A **headless system** is a computer designed to be operated from another computer via a network, at times or in places where keyboard, screen, and mouse access to a device is inconvenient. Headless access is used for server systems, for building robots and making gadgets:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

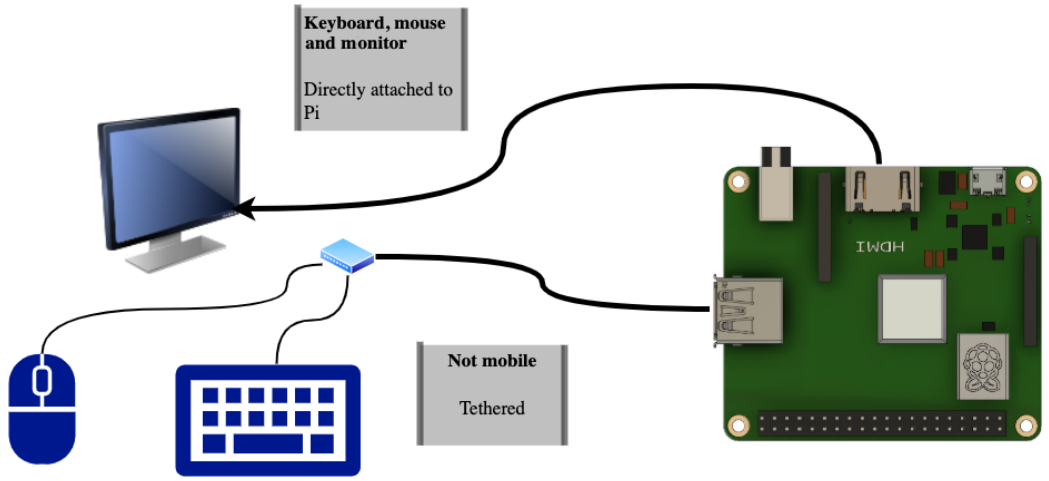


Figure 4.1 – A Raspberry Pi tethered to a screen, keyboard, and mouse

Figure 4.1 shows a system with a head where a user can sit in front of the device. You need to attach a screen, keyboard, and mouse to your robot, and hence it is not very mobile. You may be able to attach/detach them as required, but this is also inconvenient. There are portable systems designed to dock with Raspberry Pis like this, but when a robot moves, you'll need to disconnect it or move with the robot.

At some events, I have seen robots with tiny onboard screens, controlled by a wireless keyboard and mouse. However, in this book, we use a robot as a headless device:

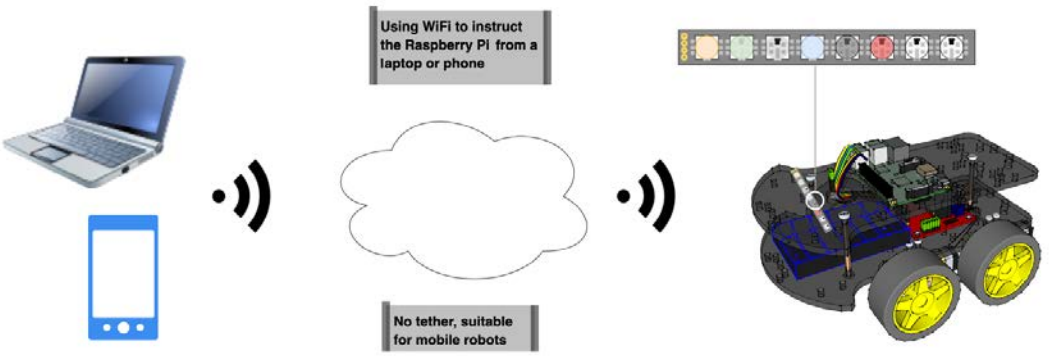


Figure 4.2 – A Raspberry Pi on a robot in a headless configuration

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Raspberry Pi in *Figure 4.2* is mounted on a robot as a headless device. This Raspberry Pi is not weighed down by a screen and keyboard; those are handled by another computer. Code, instructions, and information is sent to and from the Raspberry Pi via a wireless network from a laptop. Many code examples run autonomously, and the computer can start/stop these. We add indicator LEDs in *Chapter 9*, *Programming RGB Strips in Python*. We also show you how to use a mobile phone to drive the robot in *Chapter 17*, *Controlling the Robot with a Phone and Python*. The phone can start and stop autonomous behaviors, view the robot's status, or just drive it without needing to hook up the laptop at all. This Raspberry Pi is free from the screen and keyboard.

> **Tip**
> Although you won't usually need a screen and keyboard, it is worth having these around in case you lose contact with the Raspberry Pi, and it refuses to respond via the network. You can then use a screen and keyboard to connect to it and see what is going on.

For our headless access to the Raspberry Pi, we will be using a **Secure Shell** (**SSH**). SSH gives you a command line to send instructions to the Pi and a file transfer system to put files onto it.

Making a Pi headless makes it free to roam around. It keeps a robot light by not needing to carry or power a screen and keyboard. Being headless makes a robot smaller since a monitor and keyboard are bulky. It also encourages you, the maker, to think about autonomous behavior since you can't always type commands to the robot.

Setting up Wi-Fi on the Raspberry Pi and enabling SSH

Now you've seen what you get with a headless system, let's modify the SD card so the Raspberry Pi starts up ready to use as a headless device. We need to set up Wi-Fi first:

1. Remove and reinsert the MicroSD card we made earlier into your computer so that the computer can recognize the new state of the drive.
2. Now you will see the card shows up as two disk drives. One of the drives is called `boot`; Windows will ask whether you want to format the other drive. Click **Cancel** when Windows asks you. This part of the SD card holds a Linux-specific filesystem that Windows cannot read.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Now, in `boot`, create two files as follows. I suggest using an editor such as VSCode for plain text files, seeing file extensions, and making empty files:

- `ssh`: An empty file with no extension.
- `wpa_supplicant.conf`: This file contains your Wi-Fi network configuration as shown here:

```
country=GB
update_config=1
ctrl_interface=/var/run/wpa_supplicant

network={
  ssid="<your network ssid>"
  psk="<your network password>"
}
```

Let's go over this file line by line:

The first line must specify an ISO/IEC alpha2 country code. You can find the appropriate country code for your location at `https://datahub.io/core/country-list`. The Wi-Fi adapter will be disabled by Raspberry Pi OS if this is not present. In my case, I am in Great Britain, so my country code is GB.

The next two lines allow other tools to update the configuration.

The last 4 lines of the file define the Wi-Fi network your robot and Raspberry Pi will connect to. Use your own network details instead of the placeholders here. The **Pre-Shared Key** (**PSK**) is also known as the Wi-Fi password. These should be the same details you use to connect your laptop or phone to your Wi-Fi network. I recommend keeping a copy of the `wpa_supplicant.conf` file on your computer to use on other Raspberry Pi SD cards.

> **Important note**
>
> The `ssh` file must have no extension. It must not be `ssh.txt` or some other variation.

4. Eject the MicroSD card. Remember to use the menus to do so. This ensures that the files are entirely written before removing it.
5. Now, with these two files in place, you can use the MicroSD card to boot the Raspberry Pi. Plug the MicroSD card into the slot on the underside of the Raspberry Pi. It only fits into the slot in the correct orientation.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Finally, plug a Micro-USB cable into the side of the Raspberry Pi and connect it to a power supply. You should see lights blinking to show it is starting.

> **Important note**
> For a Raspberry Pi, you need a power supply providing at least 2.1 amps.

Your Raspberry Pi headless system has now been set up. With this, we have taken a major step toward being mobile. Now we'll need to go find it on our network so we can connect to it.

Finding your Pi on the network

Assuming your SSID and PSK are correct, your Raspberry Pi is now registered on your Wi-Fi network. However, now you need to find it. The Raspberry Pi uses dynamic addresses (DHCP). Every time you connect it to your network, it may get a different address. Visiting the admin page on your Wi-Fi router and writing down the IP address works in the short term. Doing that every time the address changes is frustrating, and may not be available in some situations.

Luckily, the Raspberry Pi uses a technology known as **mDNS** (**Multicast Domain Name System**), so nearby computers can find it. A client computer will broadcast a local message to ask for devices with the name `raspberrypi.local`, and the Raspberry Pi will respond with the address to find it. This is also known by the names Zeroconf and Bonjour. So, the first thing you'll need to do is ensure your computer can do this.

If you are using macOS, your computer will already be running the Bonjour software, which is already mDNS capable. Also, Ubuntu and Fedora desktop versions have had mDNS compatibility for a long time. On other Linux desktops, you will need to find their instructions for Zeroconf or Avahi. Many recent ones have this enabled by default.

But if you are using Windows, you will need the Bonjour software. So let's see how to set it up.

Setting up Bonjour for Microsoft Windows

If you have installed a recent version of Skype or iTunes, you will have this software. You can use this guide to check that it is already present and enable it: `https://smallbusiness.chron.com/enable-bonjour-65245.html`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can check whether it is already working with the following command in Command Prompt:

```
C:\Users\danny>ping raspberrypi.local
```

If you see this, you have Bonjour installed already:

```
PING raspberrypi.local (192.168.0.53) 56(84) bytes of data.
64 bytes from 192.168.0.53 (192.168.0.53): icmp_seq=1 ttl=64
time=0.113 ms
64 bytes from 192.168.0.53 (192.168.0.53): icmp_seq=2 ttl=64
time=0.079 ms
```

If you see this, you'll need to install it:

```
Ping request could not find host raspberrypi.local. Please
check the name and try again.
```

To do so, browse to the Apple Bonjour For Windows site at `https://support.apple.com/downloads/bonjour_for_windows` and download it, then install **Download Bonjour Print Services for Windows**. Once this has run, Windows will be able to ask for mDNS devices by name.

Testing the setup

The Raspberry Pi's green light should have stopped blinking, and only a red power light should be visible. This indicates that the Pi has finished booting and has connected to the network.

In Windows, summon a command line by pressing the Windows key and type `CMD` in the windows search bar. In Linux or macOS, open Terminal. From Terminal, we will try to `ping` the Raspberry Pi, that is, find the Pi on the network and send a short message to get a response:

```
ping raspberrypi.local
```

If everything has gone right, the computer will show that it has connected to the Pi:

```
$ ping raspberrypi.local
PING raspberrypi.local (192.168.0.53) 56(84) bytes of data.
64 bytes from 192.168.0.53 (192.168.0.53): icmp_seq=1 ttl=64
time=0.113 ms
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
64 bytes from 192.168.0.53 (192.168.0.53): icmp_seq=2 ttl=64
time=0.079 ms
64 bytes from 192.168.0.53 (192.168.0.53): icmp_seq=3 ttl=64
time=0.060 ms
64 bytes from 192.168.0.53 (192.168.0.53): icmp_seq=4 ttl=64
time=0.047 ms
```

What if you cannot reach the Raspberry Pi? In the next section, we'll try some troubleshooting steps.

Troubleshooting

If the Raspberry Pi does not appear to be responding to the ping operation, these are some steps you can take to try to diagnose and remedy the situation. Try the following:

1. Double-check your connections. You should have seen a few blinks of the green light and a persistent red light. If not, unplug the power, ensure that the SD card is seated firmly and that the power supply can give 2.1 amps, then try again.
2. Use your Wi-Fi access point settings with the Raspberry Pi booted and see if it has taken an IP address there.
3. If you find the Raspberry Pi on your Wi-Fi router, this may mean that mDNS is not running on your computer correctly. If you have not installed it, please go back and do so. On Windows, the different versions of Bonjour print services, Bonjour from Skype, and Bonjour from iTunes can conflict if installed together. Use the Windows add/remove functions to see whether there is more than one and remove all Bonjour instances, then install the official one again.
4. Next, turn the power off, take out the SD card, place this back into your computer, and double-check that the `wpa_supplicant.conf` file is present and has the correct Wi-Fi details and country code. The most common errors in this file are the following:

   a) Incorrect Wi-Fi details

   b) Missing quotes or missing/incorrect punctuation

   c) An incorrect or missing country code

   d) Keywords being in the wrong case (keywords should be lowercase, country code in uppercase)

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. The SSH file is removed when the Raspberry Pi starts. If you are sure it was there and has been removed, this means the Pi actually booted.
6. Finally, you may need to boot the Pi with a screen and keyboard connected, and attempt to diagnose the issue. The display will tell you whether there are other issues with `wpa_supplicant.conf` or other problems. Use the screen text and search the web for answers. I cannot reproduce all those here, as there are many kinds of problems that could occur here. I also recommend asking on Twitter using the tag `#raspberrypi`, on Stack Overflow, or in the Raspberry Pi Forums at `https://www.raspberrypi.org/forums/`.

We have now verified that our Pi is connected to the network, troubleshooting issues along the way. We've been able to find it with `ping`. Now we know it is there, let's connect to it.

Using PuTTY or SSH to connect to your Raspberry Pi

Earlier, we added a file to our Raspberry Pi boot named `ssh`. This activates the SSH service on the Pi. As mentioned before, SSH is an abbreviation for secure shell, intended for secure network access. In this case, we are not specifically targeting the secure encryption capabilities, but are using the remote networking capability to send instructions and files to and from the Raspberry Pi without having physical access to it.

> **Important note**
>
> If you already use an SSH client, note that not all of the Windows command-line SSH clients support mDNS.

PuTTY is a handy tool for accessing SSH and is available for Windows, Linux, and Mac. Its installation information for these operating systems can be found at `https://www.ssh.com/ssh/putty/`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Once you have PuTTY installed from the preceding link, let's get it connected to your Raspberry Pi. Follow along:

1. Start PuTTY. You will see a screen like in *Figure 4.3*. In the **Host Name (or IP address)** box, type `raspberrypi.local` and click **Open** to log in to your Pi:

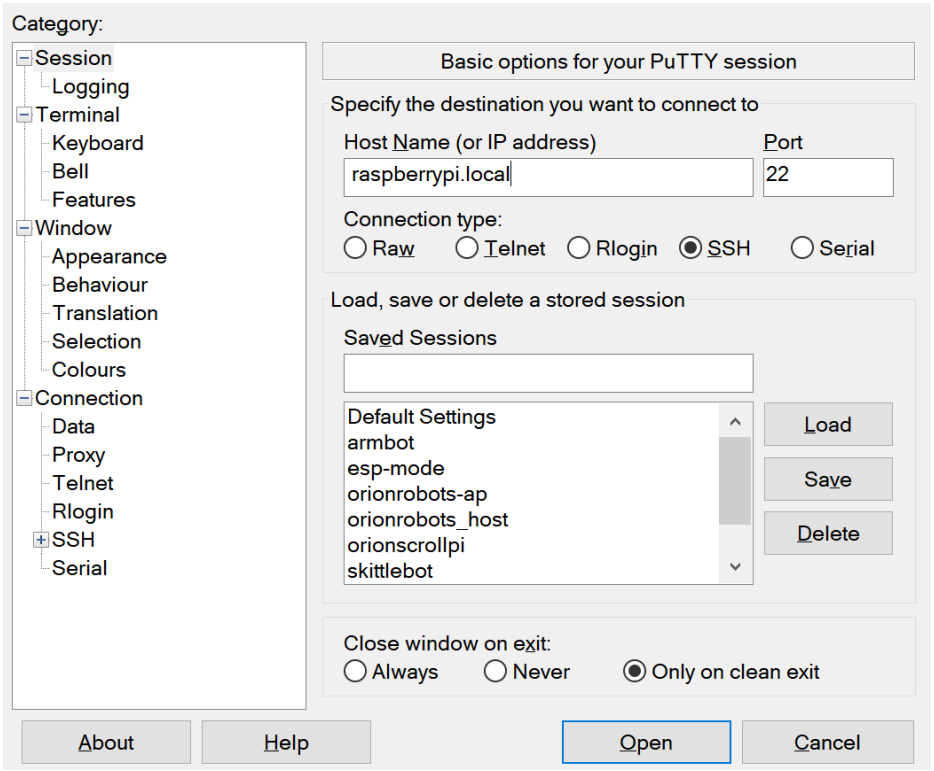


Figure 4.3 – Connecting to the Pi

2. The first time you do this, PuTTY displays a security warning asking you to add the Pi's key if you trust it. Click **Yes**; it only asks you this again if another device with the same hostname (for example, a fresh Raspberry Pi) shows up with a different key.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. When you see the **Login as** prompt, type `pi`, press *Enter*, and use the password `raspberry`. You'll now see something like *Figure 4.4*, showing that you have connected to the Pi:

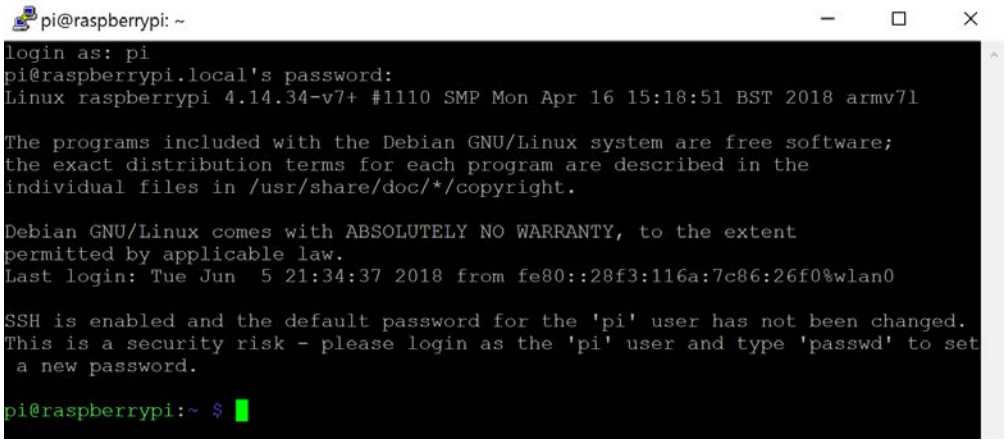


Figure 4.4 – Successfully connected

In this section, you've used PuTTY or your preferred SSH client to connect to the Raspberry Pi, setting you up to configure it, send commands to it, and interact with it. Next, we'll see how to configure it.

Configuring Raspberry Pi OS

Now we are connected, let's do a few things to prepare the Raspberry Pi for use, such as changing the user password and changing the hostname to make the Pi more secure.

We can perform many of these tasks with the `raspi-config` tool, a menu system to perform configuration tasks on Raspberry Pi OS. We start it with another tool, `sudo`, which runs `raspi-config` as root, a master user. Refer to the following command:

```
sudo raspi-config
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `raspi-config` interface will appear, as shown in *Figure 4.5*:

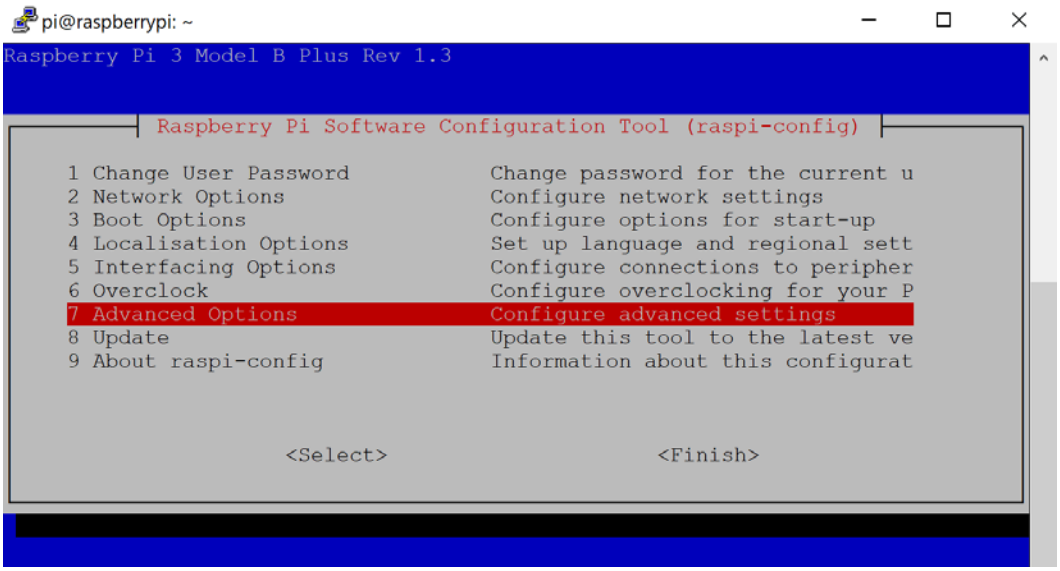


Figure 4.5 – The raspi-config tool

Now we've accessed `raspi-config`, we can use it to change some of the settings on the Raspberry Pi.

Renaming your Pi

Every fresh Raspberry Pi image is called **raspberrypi**. If there is more than one of those in a room, your computer will not be able to find yours. It's time to think of a name. For now, we'll use `myrobot`, but I am sure you can think of something better. You can change this later too. It can be letters, numbers, and dash characters only. Use the following steps:

1. In `raspi-config`, select **Network Options**, shown in *Figure 4.6*. Use the arrow keys on your keyboard to highlight and press *Enter* to select the entry:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

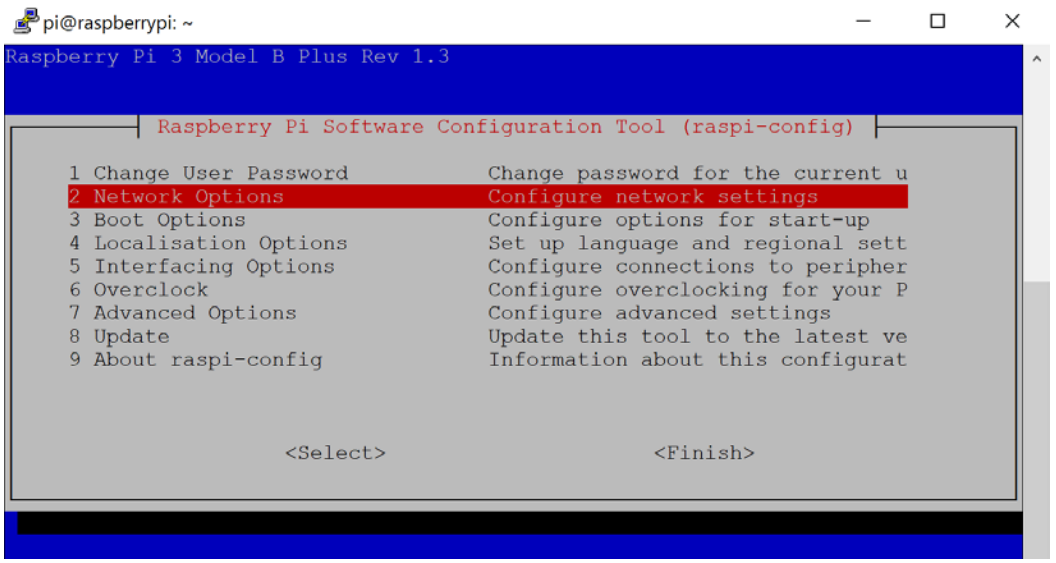


Figure 4.6 – Network options

2. Now select **Hostname** as shown in *Figure 4.7*:

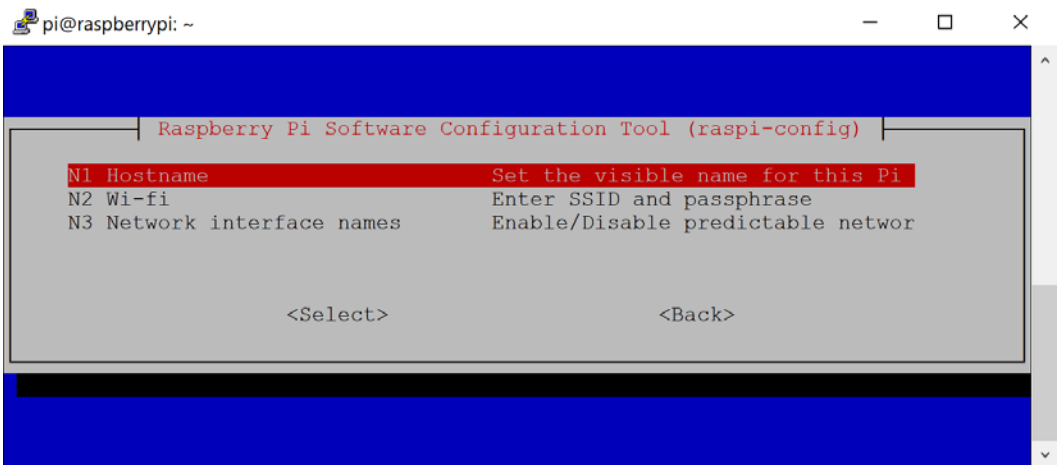


Figure 4.7 – Change the hostname

3. You should be on a screen waiting for hostname input. Type in a name for your robot (I called mine `myrobot`), then press *Enter* to set it. Please be more inventive than my name.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You have now named your robot, which will be more important if you have other Raspberry Pis but also gives the robot a little character. Let's also change the password to something a bit less generic.

Securing your Pi (a little bit)

Right now, your Raspberry Pi has the same password as every other Raspberry Pi fresh from an image. It's recommended you change it. Perform the following steps:

1. In the top menu of `raspi-config`, select **Change User Password** (*Figure 4.8*):

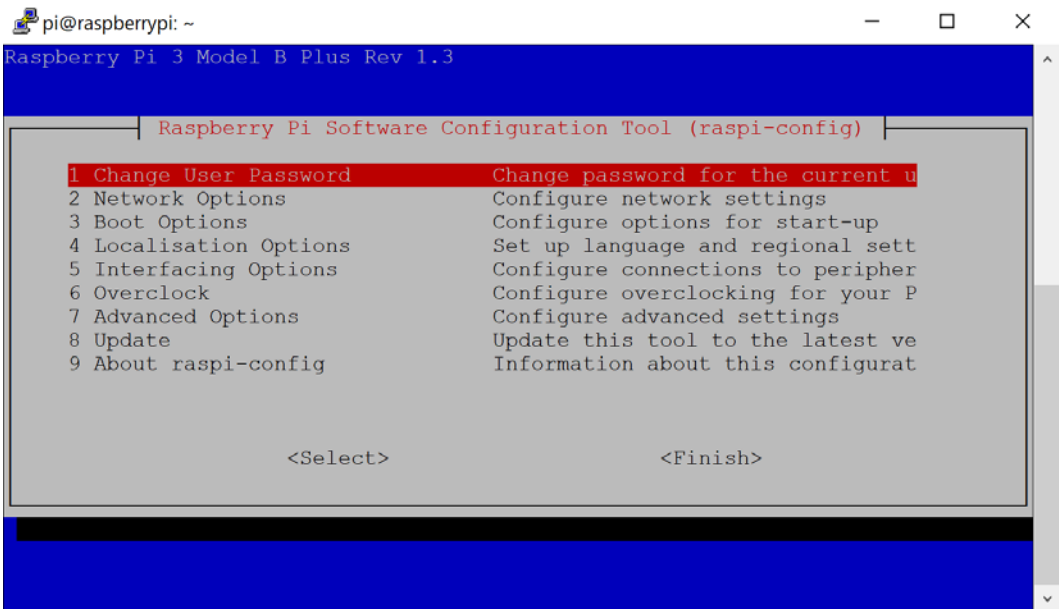


Figure 4.8 – Changing the password

2. Type a new password for your robot – something you will remember, more unique than `raspberry`. It should not be anything you've used for something sensitive such as email or banking.

Changing the password personalizes the Raspberry Pi on your robot, and also drastically reduces the likelihood of someone else connecting to your robot and ruining your hard work. Now we've made configuration changes, we need to restart the Raspberry Pi for them to take effect.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Rebooting and reconnecting

It's time to finish the configuration and restart the Pi:

1. Use the *Tab* button to get to the **Finish** item (*Figure 4.9*) and press *Enter*:

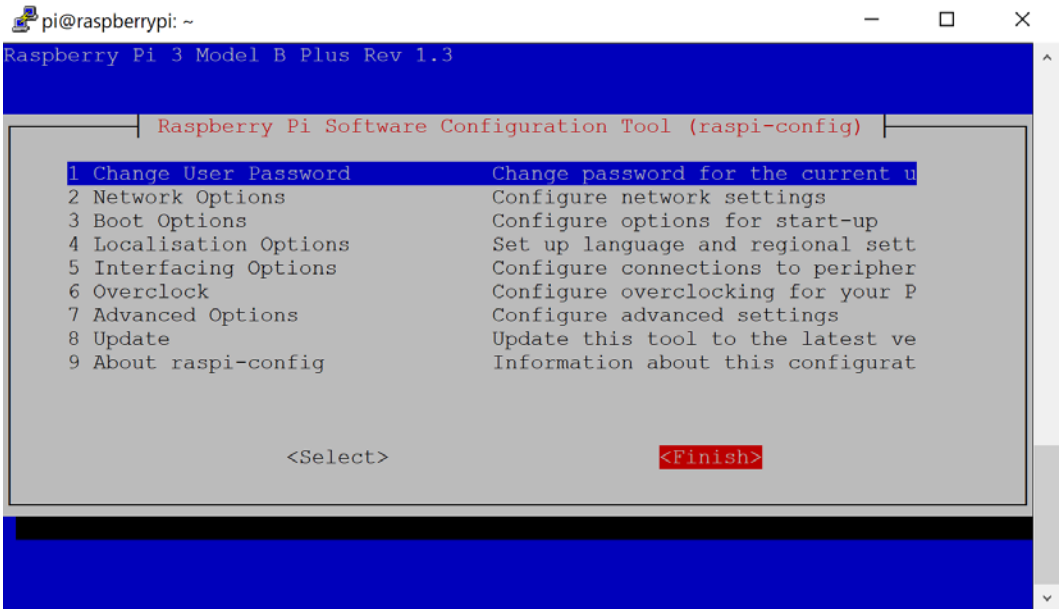


Figure 4.9 – Select Finish

2. The next screen asks whether you want to reboot the Pi. Select **Yes** and press *Enter* (*Figure 4.10*):

Figure 4.10 – Say Yes to rebooting

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Raspberry Pi will start to reset, and the PuTTY session will be disconnected as it does so (*Figure 4.11*). Wait for a few minutes; the green activity light on the Pi should blink a bit and then settle down. PuTTY will tell you it has lost connection to it. The Pi is now shut down. The red light will stay on until you remove the power:

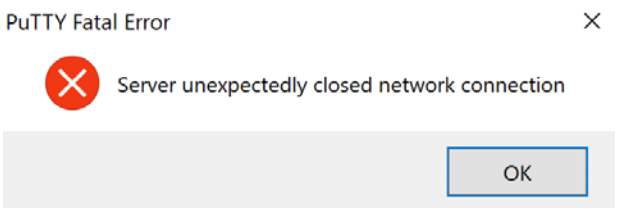


Figure 4.11 – PuTTY telling you the Pi connection has gone

PuTTY only sends commands to and from the robot; it does not understand that this command has shut down the Pi. It does not expect the connection to close. You and I know better, as we told the Pi to reboot. Click **OK** to dismiss the error.

3. Connect to it again with PuTTY using the new hostname you gave your robot (in my case, `myrobot`), with the `.local` ending and the fresh password, as shown in *Figure 4.12*:

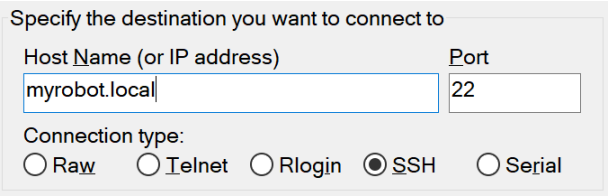


Figure 4.12 – Reconnect to the Raspberry Pi

4. Now, you should be able to log in and see your prompt as **pi@myrobot**, or whatever your robot's name is, as shown in *Figure 4.13*:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

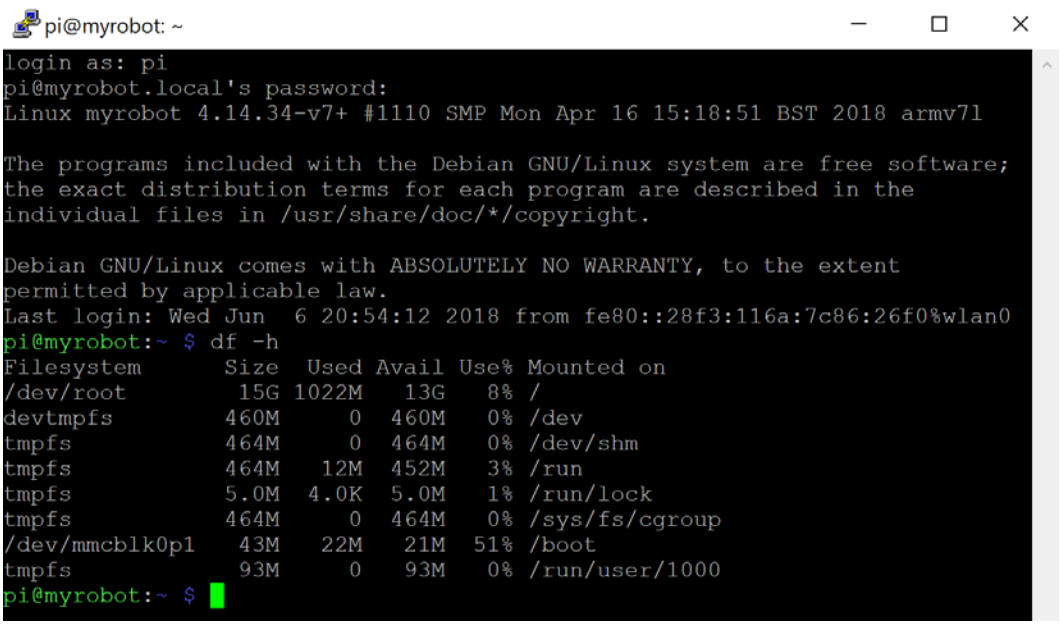


Figure 4.13 – Reconnected to the Raspberry Pi

We are now reconnected to the Raspberry Pi. We can try a simple Linux command to see how much of the SD card we are using. Linux commands are often abbreviations of things you want to ask the computer to do.

The `df` command in *Figure 4.13* shows the space used in the various storage locations connected to your Raspberry Pi. The additional `-h` makes `df` display this in human-readable numbers. It uses `G`, `M`, and `K` suffixes for gigabytes, megabytes, and kilobytes. Type the `df -h` command, as shown in the preceding screenshot, and it will show that `/dev/root` is close to the full size of the SD card, with some other devices taking up the rest of the space.

Updating the software on your Raspberry Pi

One last thing to do here is to ensure your Raspberry Pi has up-to-date software on it. This is the kind of process you start off and leave going while getting a meal, as it will take a while. Type the `sudo apt update -y && sudo apt upgrade -y` command, and you should see something similar to the following:

```
pi@myrobot:~ $ sudo apt update – y && sudo apt upgrade -y
Hit:1 http://raspbian.raspberrypi.org/raspbian buster InRelease
Hit:2 http://archive.raspberrypi.org/debian buster InRelease
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
.
.
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages will be upgraded:
  bluez-firmware curl libcurl3 libcurl3-gnutls libprocps6
pi-bluetooth procps
  raspi-config wget
9 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 1,944 kB of archives.
After this operation, 16.4 kB of additional disk space will be
used. Get:1 http://archive.raspberrypi.org/debian buster/main
armhf bluez-firmware all 1.2-3+rpt6 [126 kB]
.
.
.
```

Please let the Pi continue until it is complete here, and do not interrupt or turn the power off until the `pi@myrobot:~ $` prompt has reappeared.

> **Tip**
>
> You've probably seen a pattern with `sudo` in a few of the commands. This command tells the Raspberry Pi to run the following command (such as `raspbi-config` or `apt`) as a *root* user, the Linux administrator/superuser. You can read it as *superuser do*. This is needed for software that will make changes to the system or perform updates. It's usually not required for user programs, though.

It's worth doing this update/upgrade step monthly while actively working on a project. With the Raspberry Pi up to date, you will be able to install additional software for the robot code. When the software gets stale, `apt-get` installations may not work. An update will usually solve that.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Shutting down your Raspberry Pi

When you are done with the Pi for the session, shut it down as shown:

```
pi@myrobot:~ $ sudo poweroff
```

Wait for the green light activity to stop; PuTTY will detect that it has disconnected. You can now safely disconnect the power.

> **Important note**
>
> Pulling power from the Raspberry Pi when it is not expected can cause the loss of files and SD card corruption. You may lose your work and damage the SD card. Always use the correct shutdown procedure.

Summary

In this chapter, you've seen how to free a Raspberry Pi from a screen and keyboard by making it headless. You set up an SD card to connect to your Wi-Fi and to enable SSH so you could connect to it. You've used `raspi-config` to personalize your Pi and secure it with your own password. You then made the first small steps in looking around the Linux system it has running on it. You also ensured the Raspberry Pi is up to date and running the most current software. Finally, we saw how to safely put the Pi into shutdown mode, so that filesystem damage does not occur when you unplug it.

You have now learned how to make a Raspberry Pi headless. You have seen how to keep it upgraded and connected to your network and the Pi is ready to start building with. You can use this to build Raspberry Pi-powered gadgets, including robots.

In the next chapter, we look at ensuring you don't lose valuable code or configuration when things go wrong. We will learn about what can go wrong and how to use Git, SFTP, and SD card backups to protect our hard work.

Assessment

1. What other gadgets or projects could you build with a headless Raspberry Pi?
2. Try giving your Raspberry Pi a different hostname and connecting to this locally with PuTTY and mDNS.
3. Try other Linux commands on the Raspberry Pi, such as `cd`, `ls`, `cat`, and `man`.
4. Shut down the Raspberry Pi correctly after trying these.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

Please refer to the following to get more information:

- *Internet of Things with Raspberry Pi 3*, *Maneesh Rao*, *Packt Publishing*: This book uses a wired headless Raspberry Pi for the demonstrations and experiments in it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5

Backing Up the Code with Git and SD Card Copies

As you create and customize the code for your robot, you will invest many hours in getting it to do awesome things that, unless you take precautions, could all suddenly disappear. The programs are not the whole story, as you've already started configuring Raspberry Pi OS for use on the robot. You want to keep your code and config in case of disaster, and to be able to go back if you make changes you regret.

This chapter will help you understand how exactly code or configuration can break and the disasters you might face while customizing code for your robot. We'll then take a look at three strategies for preventing this.

In this chapter, you will learn about the following:

- Understanding how code can be broken or lost
- Strategy 1 – Keeping the code on a PC and uploading it
- Strategy 2 – Using Git to go back in time
- Strategy 3 – Making SD card backups

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

For this chapter, you will require the following:

- The Raspberry Pi and the SD card you prepared in the previous chapter
- The USB power supply and cable you used with the Pi
- A Windows, Linux, or macOS computer or laptop, connected to the internet and able to read/write to SD cards
- Software: FileZilla and Git
- On Windows: Win32DiskImager

Here is the GitHub link for the code files of this chapter:

`https://github.com/PacktPublishing/Learn-Robotics-Fundamentals-of-Robotics-Programming/tree/master/chapter5`

Check out the following video to see the Code in Action: `https://bit.ly/3bAm94l`

Understanding how code can be broken or lost

Code and its close cousin, configuration, take time and hard work. Code needs configuration to run, such as Raspberry Pi OS configuration, extra software, and necessary data files. Both need research and learning and to be designed, made, tested, and debugged.

Many bad situations can lead to the loss of code. These have happened to me a week before taking robots to a show after weeks of work, and I learned the hard way to take this quite seriously. So, what can happen to your code?

SD card data loss and corruption

SD card corruption is when the data on the SD card used to hold your code, Raspberry Pi OS, and anything you've prepared on it gets broken. Files become unreadable, or the card becomes unusable. The information on the SD card can be permanently lost.

If a Raspberry Pi unexpectedly loses power, the SD card can be damaged, causing data loss. A hot Pi can slowly bake an SD card, damaging it. Visual processing on a Pi is one way it can get hot. SD cards get damaged if something terrible happens electrically to the Pi via the GPIO pins or its power supply. MicroSD cards are also tiny and are quickly lost when not in the Pi.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Changes to the code or configuration

We all make mistakes. Coding means trying things out. While some work out, some do not and things break. At those times, you'll want to go back and see what you've changed. You might be able to use differences to find the bug, or if your experiment looks like a dead end, you may want to go back to a known working version.

You can also render your robot useless with the wrong configuration, such as the Pi not being on the network or booting anymore. An upgrade to system packages may go wrong and lead to code not working or needing extensive changes to the code for it to work again.

These problems can combine to cause real nightmares. I've seen changes in the code lead a robot to misbehave and damage itself in a way that made the SD card corrupted. I've been updating packages on the operating system when I knocked the power cable out, corrupting the SD card and breaking Raspberry Pi OS 2 weeks before a significant robot event, and it was painful rebuilding it. This was a lesson learned the hard way.

Back up the code and back up the SD card configuration. Over the rest of this chapter, we'll look at some solutions to keep your robot's software safe from many kinds of disasters.

Strategy 1 – Keeping the code on a PC and uploading it

Secure File Transfer Protocol (**SFTP**) lets you transfer files from a computer to a Pi. This strategy enables you to write code on your computer, then upload it to the Raspberry Pi. You can choose your editor and have the safety of more than one copy.

> **Important note**
>
> But wait – which editor? Editing code requires software designed for this purpose. Recommendations for Python are Mu, Microsoft VS Code, Notepad++, and PyCharm.

SFTP uses SSH to copy files to and from the Raspberry Pi over the network. So, let's see how to do it:

1. First, make yourself a folder on the PC to store your robot code in; for example, `my_robot_project`.
2. Inside that folder, make a test file, using your editor, that will just print a bit of text. Put this code into a file named `hello.py`:

```
print("Raspberry Pi is alive")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We will copy this to the robot and run it. You can make the copy using the SFTP tool FileZilla from `https://filezilla-project.org`. Download this and follow the installation instructions:

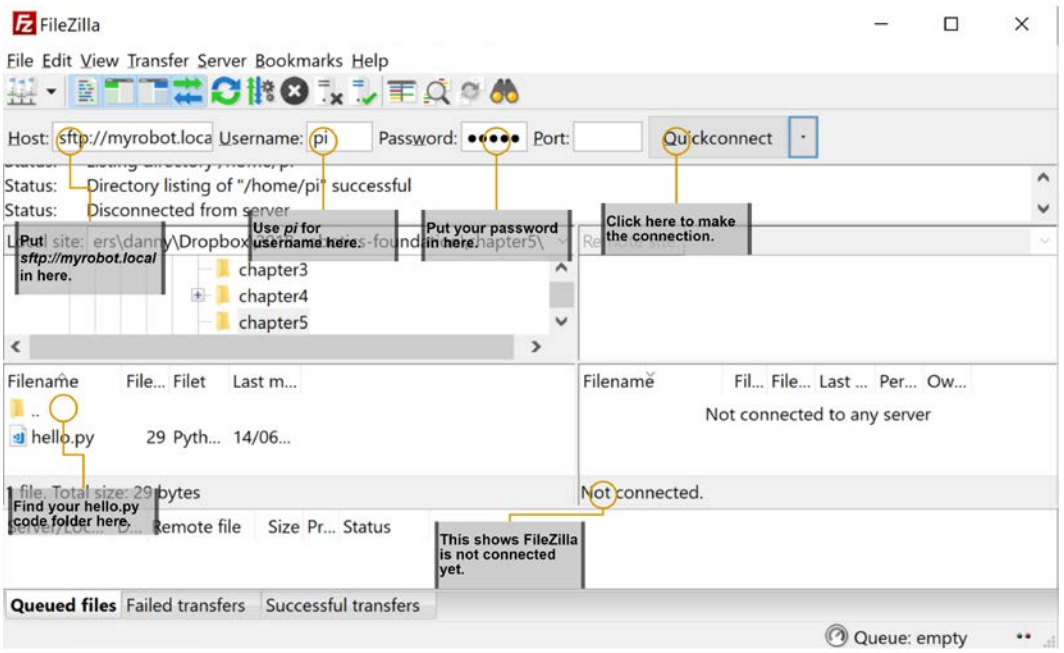


Figure 5.1 – FileZilla

4. Plug in and power up your Raspberry Pi. You will notice at the bottom of the right-hand panel (*Figure 5.1*), FileZilla says **Not connected**.
5. In the **Host** box, type the local hostname you gave your robot Pi in the headless setup, prefixed with `sftp://`; for example, `sftp://myrobot.local`.
6. In the **Username** box, type `pi`, and in the **Password** box, enter the password you set up before.
7. Click the **Quickconnect** button to connect to the Raspberry Pi.
8. When connected, you'll see files on the Raspberry Pi in the right-hand **Remote site** panel, shown in *Figure 5.2*:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

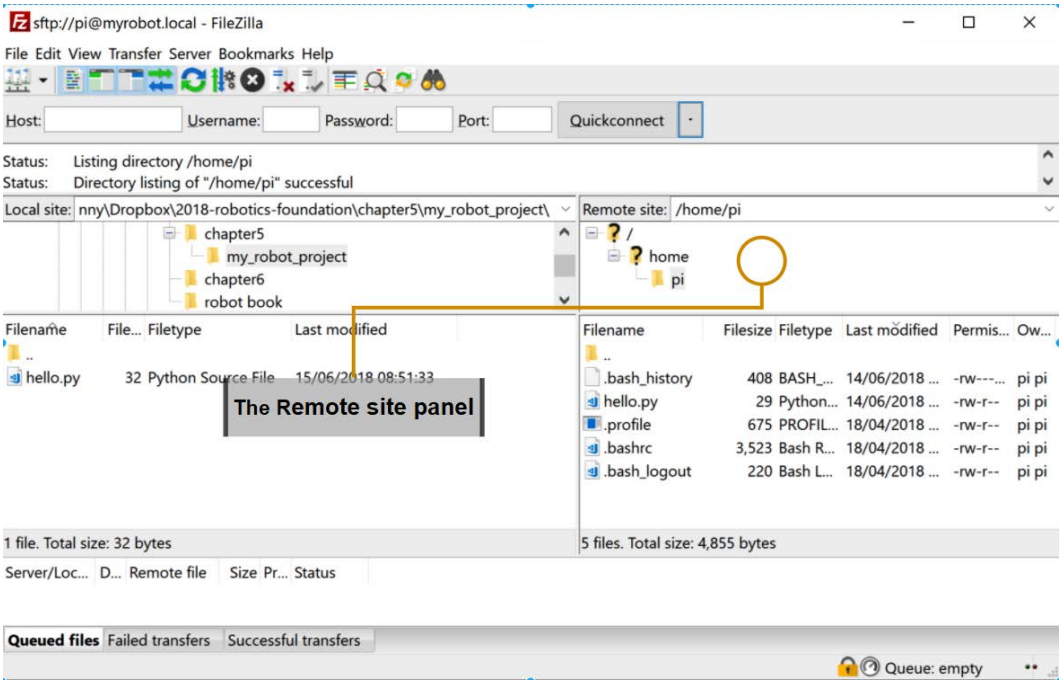


Figure 5.2 – The Raspberry Pi connected

9. Use the left-hand **Local site** panel to go to your code on your computer.
10. Now click `hello.py`, highlighted at the top left of *Figure 5.3*, and drag it to the lower right-hand panel to put it on the Raspberry Pi:

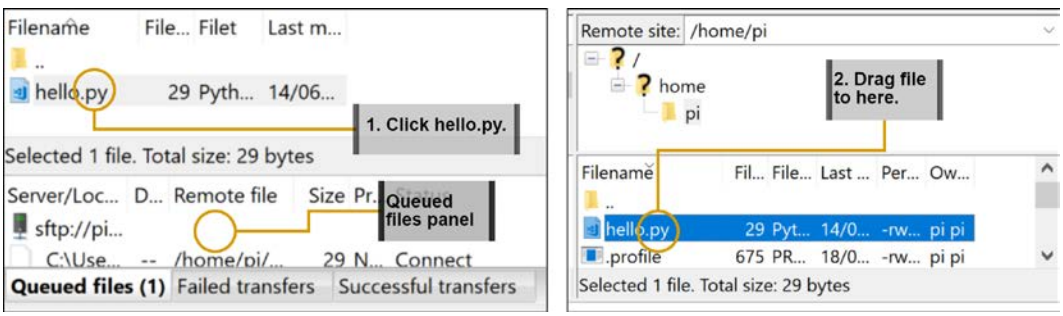


Figure 5.3 – Transferring a file

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. When you drag the file over, you should see it in the **Queued files** section, as shown in *Figure 5.3*. Since this file is small, it will only be in this queued state for an instant. You can also use the same system for whole folders. You'll soon see the file over in the remote site (the Raspberry Pi), shown on the right-hand panel in *Figure 5.3*.
12. To run this code, use PuTTY to log in to the Pi and try the following command:

```
pi@myrobot:~ $ python3 hello.py
Raspberry Pi is alive
```

This strategy is a great start to making code safer. By working on your laptop/PC and copying to the Pi, you've guaranteed there is always one copy other than the one on the robot. You've also got the ability to use any code editor you like on the PC and spot some errors before they even get to the Raspberry Pi. Now we have a copy, let's see how we can track changes to our code and see what we've changed.

Strategy 2 – Using Git to go back in time

Git is a popular form of source control, a way to keep a history of changes you've made to code. You can go back through changes, see what they were, restore older versions, and keep a commented log of why you made the changes. Git also lets you store code in more than one location in case your hard drive fails. Git stores code and its history in repositories, or repos. In Git, you can make branches, copies of the whole set of code, to try ideas in parallel with your code, and later merge those back to the main branch.

I will get you started, but this section can only scratch the surface of what you can do with Git. Let's begin:

1. Install Git, by following the instructions at `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git` for your computer.

> **Tip**
> If you are using Windows or macOS, I would suggest using the GitHub app for easier setup.

2. Git requires you to set your identity using a command line on your computer:

```
> git config --global user.name "<Your Name>"
> git config --global user.email <your email address>
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. To put this project under source control, we need to initialize it and commit our first bit of code. Make sure you are in the folder for your code (`my_robot_project`) in a command line on your computer and type the following:

```
> git init .
Initialized empty Git repository in C:/Users/danny/
workspace/my_robot_project/.git/
> git add hello.py
> git commit -m "Adding the starter code"
[master (root-commit) 11cc8dc] Adding the starter code
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

`git init .` tells Git to make the folder into a Git repository. `git add` tells Git you want to store the `hello.py` file in Git. `git commit` stores this change for later, with `-m <message>` putting a message in the journal. Git responds to show you it succeeded.

4. We can now see the journal with `git log`:

```
> git log
commit 11cc8dc0b880b1dd8302ddda8adf63591bf340fe (HEAD ->
master)
Author: Your Name <your@email.com>
Date: <todays date>

Adding the starter code
```

5. Now modify the code in `hello.py`, changing it to this:

```
import socket
print('%s is alive!' % socket.gethostname())
```

If you copy this to the Pi using SFTP, this will say `myrobot is alive!` or whatever you set the hostname of your robot to be. However, we are interested in Git behavior. Note – more advanced Git usage could let you use Git to transfer code to the Raspberry Pi, but that is beyond the scope of this chapter. Let's see how this code is different from before:

```
> git diff hello.py
diff --git a/hello.py b/hello.py
index 3eab0d8..fa3db7c 100644
--- a/hello.py
+++ b/hello.py
@@ -1 +1,2 @@
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
-print("Raspberry Pi is alive")
+import socket
+print('%s is alive!' % socket.gethostname())
```

The preceding is Git's way of showing the differences. Git interprets the changes as taking away a `print` line, and in its place adding an `import` and then a `print` line. We can add this into Git to make a new version, and then use `git log` again to see both versions:

```
> git add hello.py
> git commit -m "Show the robot hostname"
[master 912f4de] Show the robot hostname
 1 file changed, 2 insertions(+), 1 deletion(-)
> git log
commit 912f4de3fa866ecc9d2141e855333514d9468151 (HEAD ->
master)
Author: Your Name <your@email.com>
Date: <the time of the next commit>

Show the robot hostname

commit 11cc8dc0b880b1dd8302ddda8adf63591bf340fe (HEAD ->
master)
Author: Your Name <your@email.com>
Date: <todays date>

Adding the starter code
```

With this method, you can go back to previous versions, or just compare versions, and protect yourself against changes you might regret. However, we have only just scratched the surface of the power of Git. See the reference in the *Further reading* section on how to branch, use remote services, roll back to previous versions, and find tools to browse the code in the Git history.

Now we can go back and forward in time (at least for our code), we can be more confident in making changes. Just remember to make commits frequently – especially after making something work! Next, we will look at how to keep the configuration and installed packages.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Strategy 3 – Making SD card backups

Git and SFTP are great for keeping code safe, but they don't help you reinstall and reconfigure Raspberry Pi OS on a card. The procedures for Windows, Linux, and macOS are quite different for this. The basic idea is to insert the SD card and use a tool to clone the whole card to a file known as an image, which you can restore with balenaEtcher when you need recovery.

> **Important note**
>
> You should only restore images to cards of the same size or larger. Putting an image on a smaller device is likely to fail to write, creating a corrupt SD card.

Before we begin, properly shut down your Raspberry Pi, take out its SD card, and put that into your computer. These clean images are large, so do not put them in your Git repository. It's beyond the scope of this chapter, but I recommend finding a way to compress these files as they are mostly empty right now. In all cases, expect this operation to take 20-30 minutes due to the image sizes.

Windows

For Windows, we'll use Win32DiskImager. So, our first step will be to install and set this up. Follow along:

1. Get an installer for this at `https://sourceforge.net/projects/win32diskimager`.
2. Run this and follow the installation instructions.

> **Tip**
>
> Since we will use it immediately, I suggest leaving the **Launch immediately** checkbox ticked.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Highlighted on the right of *Figure 5.4* is the **Device**; this should have automatically found the SD card device. Use the folder icon highlighted to choose where the image file will be stored:

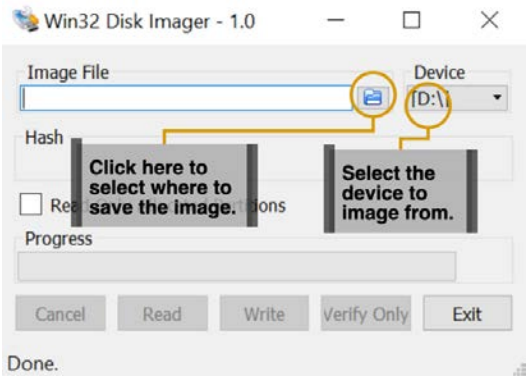


Figure 5.4 – Win32 Disk Imager

4. In *Figure 5.5*, I name my image `myrobot.img` in the **File name** box. You then click the **Open** button to confirm this:

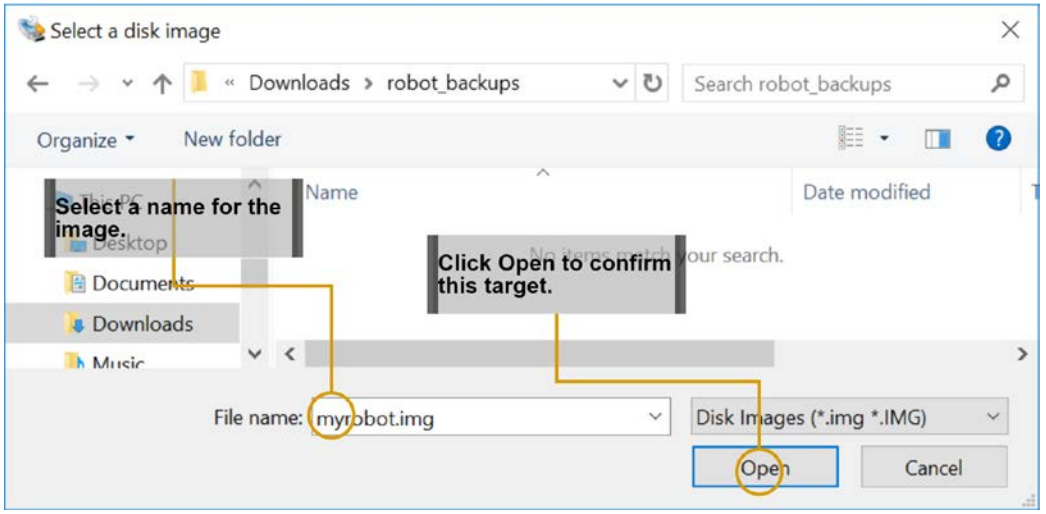


Figure 5.5 – Choose the location

5. After clicking **Open**, you'll see a screen like the left side of *Figure 5.6* with your selected location in the **Image File** box. Click on the **Read** button to start copying the image. As it reads the image, you'll see a progress bar and an estimation of the time remaining. When the image is done, Win32 Disk Imager will tell you that the read was successful, and you can then exit the software:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

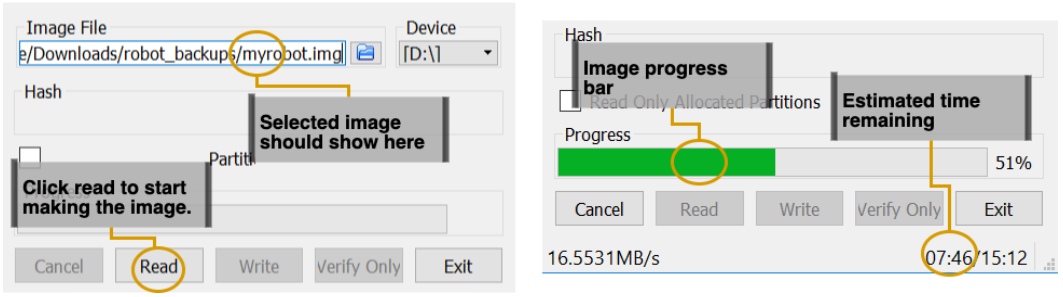


Figure 5.6 – Reading the image

You have now created a complete copy of the data on the SD card. If you have corruption or configuration issues, you can write this image back to an SD card to restore it to this point.

Mac

MacOS X has a built-in way to make SD card and disk images. This is by using the built-in Disk Utility tool. Let's see how this works:

1. Start the **Disk Utility** tool. When loaded, it should look like *Figure 5.7*:

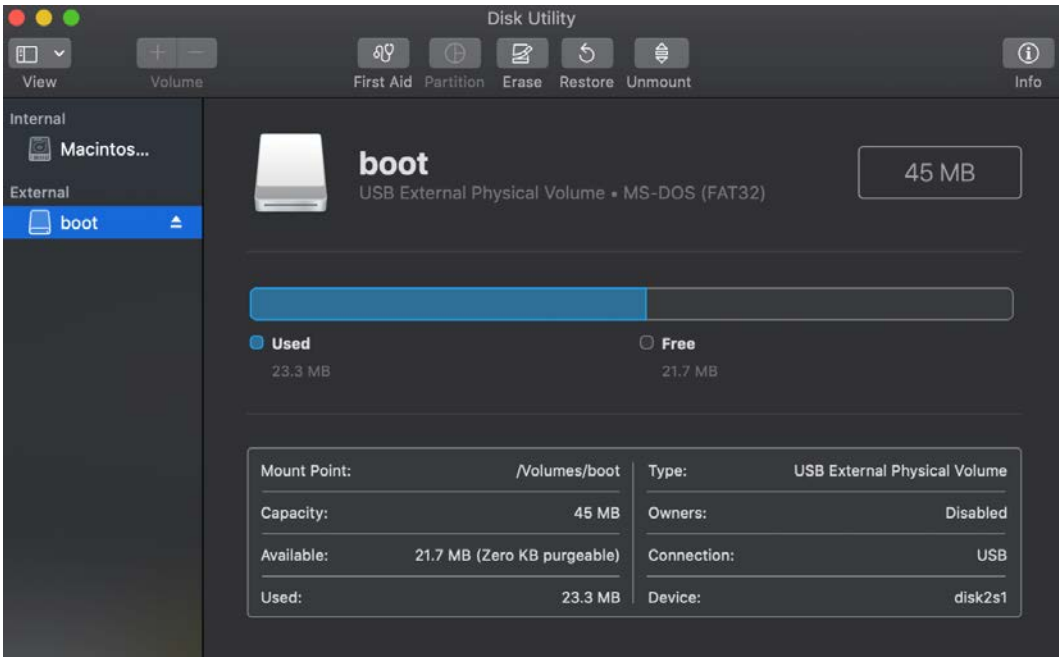


Figure 5.7 – The Disk Utility tool

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Click the **View** menu to show *Figure 5.8*:

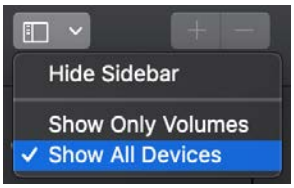


Figure 5.8 – The View menu

3. Now click on the **Show All Devices** option.
4. You should now see the screen shown in *Figure 5.9*. Select the device that contains a boot volume:

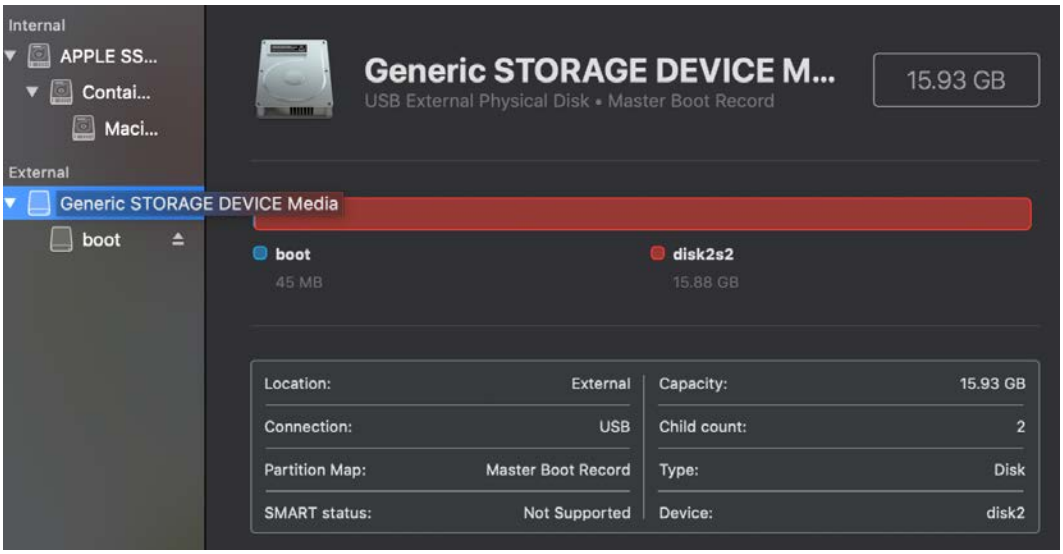


Figure 5.9 – Disk Utility with Show All Devices enabled

5. In the menu bar, select **File | New Image** (*Figure 5.10*):

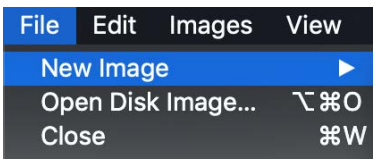


Figure 5.10 – New Image menu

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Under this, select **Image from <your storage device>** (*Figure 5.11*):

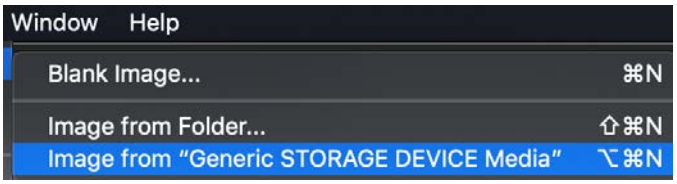


Figure 5.11 – Image from STORAGE DEVICE

7. Disk Utility will show a dialog (*Figure 5.12*). Set the file name and location, and **Format** to **DVD/CD master**:

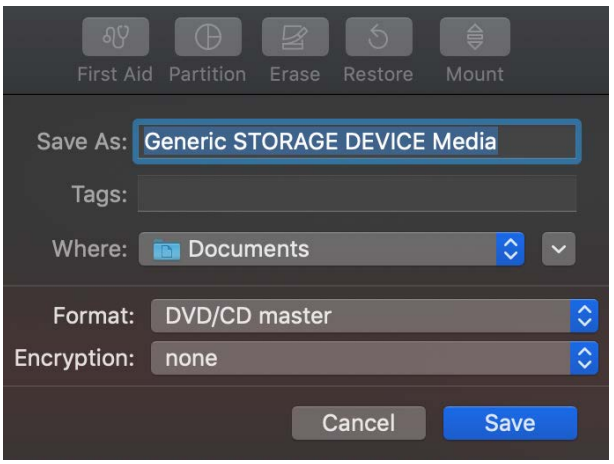


Figure 5.12 – Save dialog

8. Disk Utility gives these files a `.cdr` extension (*Figure 5.13*):

Figure 5.13 – File with .cdr extension

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Rename this to a `.iso`:

Figure 5.14 – Renamed to .iso

10. You will need to confirm you want this (*Figure 5.15*):

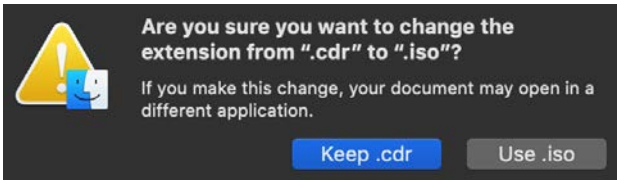


Figure 5.15 – Confirm extension change

You are now able to create SD images ready to use with balenaEtcher on macOS.

Linux

Backing up SD cards is done on the command line in Linux by using the `dd` command. Before we see how this works, we will first need to find the device's location. Let's begin:

1. Insert the card and type the following to find the device's location:

```
$ dmesg
```

2. This command will output a lot of stuff, but you are interested only in a line near the end that looks like the following:

```
sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

The card is in the square brackets, `[sdb]`, which may be different on your computer. The SD card location will be `/dev/<drive location>`, for example, `/dev/sdb`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Important note**
>
> Be careful to get the locations right, as you could destroy the contents of an SD card or your computer hard drive. If you are at all unsure, **do not** use this method.

3. Once you have the SD location (such as `/dev/sdb` or `/dev/disk1`), you can then start the clone with the `dd` command. This command dumps data to and from drives:

```
$ sudo dd if=/dev/sdb of=~/myrobot.img bs=32M
Password:
474+2 records in
474+2 records out
15931539456 bytes (16 GB, 15 GiB) copied, 4132.13 s, 3.9
MB/s
```

The `if` parameter is the **input file**, which in this case is your SD card. The `of` parameter is the **output file**, the `myrobot.img` file you are cloning your card into.

The `bs` parameter is the **block size**, so making this large, such as `32M`, will make the operation quicker.

You will need to type your user password for this to start. The `dd` command creates the `myrobot.img` file as a clone of the whole SD card in your home directory. `dd` will give no output until it is complete, and will then show you stats about the operation.

Summary

In this chapter, you have learned how to look after your code and configuration. You have seen how things can go wrong, and the strategies to protect your work from them. You have a starting point with Git, SFTP, and SD card backups that you can use together to be a bit more experimental and fearless about changes to your robot. You can use SFTP to edit on your computer, giving you at least one copy other than the code on your robot and letting you use powerful editors. You can use Git to go back in time, so you can wind back from mistakes and experiments, or just see the differences. You can use SD card backups to get a complete image of the storage your Raspberry Pi is using, and restore it if it goes wrong.

In the next chapter, we'll start to build a basic robot. We'll assemble the robot chassis with motors and wheels, determine what power systems to use, then test fit the overall shape of our robot. Bring a screwdriver!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Assessment

- Try creating a file on your computer – a simple image or text. Try using SFTP to send it to the Raspberry Pi, then, using PuTTY, see if you can list the file with the `ls` command. The file could be a simple Python script, which you could try running on the Raspberry Pi.
- Make a change that is incorrect to `hello.py`. Use `diff` to see the difference. Use Git resources (see the *Further reading* section) to find out how to return this to how it was before the change.
- Make a backup of your Raspberry Pi SD card using the preceding instructions, make some changes to the data in `/home/pi`, then restore the image using balenaEtcher. You could even restore your backup to another SD card, and plug it into the Raspberry Pi as if it was the original.
- I recommend finding out more about how Git can be used to look after your code, and even as a method of getting code onto the Raspberry Pi. Use the *Further reading* section to find out more about Git, and ways to work it into your coding workflow. Git can be complicated, but it is a tool worth learning.

Further reading

Please refer to the following for more information:

- The Git Handbook on GitHub: `https://guides.github.com/introduction/git-handbook/`. This document is a comprehensive look at what Git is, the problems it solves, and a starting point to using its functionality.
- Hands-On Version Control with Git: `https://www.packtpub.com/application-development/hands-version-control-git-video`. This is a video tutorial on using Git.
- The GitHub Guides: `https://guides.github.com/`. A series of guides on getting the best out of Git and GitHub.
- GitLab Basics: `https://docs.gitlab.com/ee/gitlab-basics/`. GitLab is an excellent alternative to GitHub, with a large community and some excellent guides on using Git.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Section 2: Building an Autonomous Robot – Connecting Sensors and Motors to a Raspberry Pi

In this section, we will build a robot and use code to get it to move. We will also connect sensors and motors to a controller and write code for basic autonomous behaviors.

This part of the book comprises the following chapters:

- *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*
- *Chapter 7*, *Drive and Turn – Moving Motors with Python*
- *Chapter 8*, *Programming Distance Sensors with Python*
- *Chapter 9, Programming RGB Strips in Python*
- *Chapter 10*, *Using Python to Control Servo Motors*
- *Chapter 11*, *Programming Encoders with Python*
- *Chapter 12*, *IMU Programming with Python*

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6
Building Robot Basics – Wheels, Power, and Wiring

In this chapter, we will start building the robot. We will choose a robot chassis kit with wheels and motors, a motor controller, and some power for the robot, talking through the trade-offs and things to avoid. We'll see how to ensure that everything fits and then build the robot. By the end of the chapter, you will have your basic robot structure ready.

Getting the trade-offs and plan right now gives you a robot you can build upon and experiment with, ensuring you know components are suitable before buying them.

In this chapter, you will learn about the following:

- Choosing a robot chassis kit
- Choosing a motor controller board
- Powering the robot
- Test fitting the robot
- Assembling the base
- Connecting the motors to the Raspberry Pi

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

For this chapter, you will require the following:

- A computer with access to the internet.
- The Raspberry Pi and an SD card.
- A set of screwdrivers: M2.5, M3 Phillips, and some jeweler's screwdrivers.
- A pair of long-nose pliers. Optionally, a set of miniature metric spanners.
- Some electrical tape.
- Hook and loop or Velcro tape.
- Drawing software such as `app.diagrams.net`, Inkscape, Visio, or similar software.
- Nylon standoff kits for M2.5 and M3 threads.
- Some insulation tape.
- Four AA batteries, charged.

> **Important note**
> You will be choosing and purchasing a chassis, motor controller, and battery compartment in this chapter, but do not buy them yet.

Check out the following video to see the Code in Action: `https://bit.ly/3oLofCg`

Choosing a robot chassis kit

The chassis, like the controller, is a fundamental decision when making a robot. Although these can be self-made using 3D printing or toy hacking, the simplest place to start is with a chassis kit. These kits contain sets of parts to start your robot build. A chassis can be changed, but it would mean rebuilding the robot.

The internet has plenty of chassis kits around – too many. So how do you choose one?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Size

Getting the size of a robot right matters too. Make too small a robot and you will need miniaturization skills for electronics; too large and you will need far more serious power handling. These are both things to avoid for a beginner:

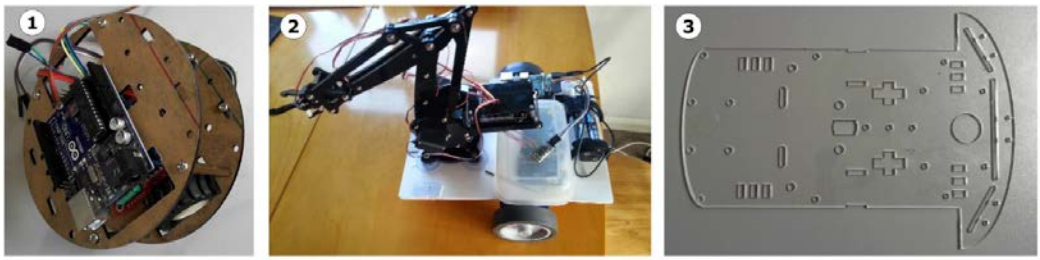


Figure 6.1 – Robot chassis sizes compared

We can compare different robot sizes from the chassis photos in *Figure 6.1*:

- **Chassis 1** has a diameter of 11 cm and just about fits a controller in but is too tiny. Being so small makes it hard to build your robot. Squeezing the controller, power, and all the sensors into this small space would need skill and experience beyond the scope of a first robot build.
- **Chassis 2** is Armbot. This bigger robot is 33 cm by 30 cm wide, giving it lots of space and an arm reach of another 300 mm. It needs eight AA batteries, big motors, and a powerful motor controller. These add to the expense and may cause issues around power handling, weight, and rigidity for a new builder. Armbot is one of my most expensive robots, excluding the cost of the arm!
- **Chassis 3** fits the Pi, batteries, and sensor, but without being large and bulky. It is around the right dimensions, being between 15-20 cm long and 10-15 cm wide. Those that have split levels might work for this, but no more than two levels, as three or four can make a robot top-heavy and cause it to topple. Chassis 3 has enough space and is relatively easy to build.

Let's look at the wheel count next.

Wheel count

Some chassis kits have elaborate movement methods – legs, tank tracks, Mecanum wheels, and tri-star wheels, to name a few. While these are fun and I encourage experimenting with them, this is not the place to start. I recommend thoroughly sensible and basic wheels on the your first robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

There are kits with four-wheel drive (shown in *Figure 6.2*) and six-wheel drive. These can be quite powerful and require larger motor controllers. They may also chew through batteries, and you are increasing the likelihood of overloading something. The additional motors can mean trickier wiring:

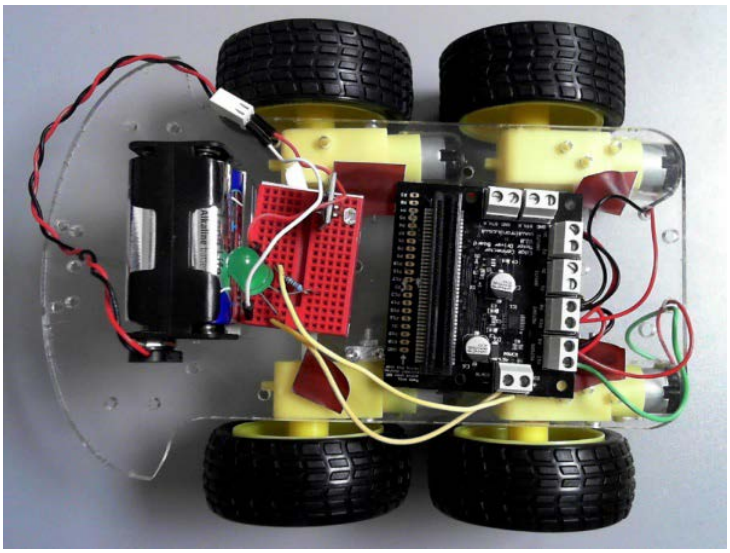


Figure 6.2 – Four-wheel drive robot

Two-wheel drive is the simplest to wire in. It usually requires a third wheel for balance. The third wheel can be a castor wheel (shown in *Figure 6.3*), a rollerball, or just a Teflon sled for tiny robots. Two wheels are also the easiest to steer, avoiding some friction issues seen with robots using four or more wheels:

Figure 6.3 – Two wheels with a castor

Two wheels won't have the pulling power of four- or six-wheel drive, but they are simple and work. They are also the least expensive.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Wheels and motors

A kit for a beginner should come with the wheels and the motors. The wheels should have simple rubber tires. *Figure 6.4* shows a common style for inexpensive robot wheels. There are many kits with these in them:

Figure 6.4 – Common inexpensive robot wheels

The kit should also have two motors, one for each wheel, and include the screws or parts to mount them onto the chassis. I recommend DC gear motors, as the gearing keeps the speed usable while increasing the mechanical pushing power the robot has.

Importantly, the motors should have the wires connected, like the first motor in *Figure 6.5*:

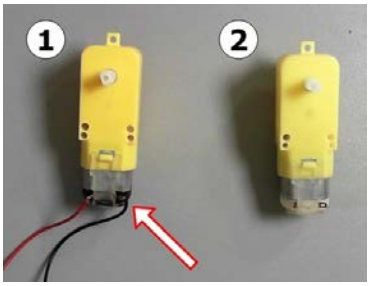


Figure 6.5 – Gear motors with and without wires

It is tricky to solder or attach these wires to the small tags on motors, and poorly attached ones have a frustrating habit of coming off. The kits you want to start with have these wires attached, as can be seen in *Figure 6.6*:

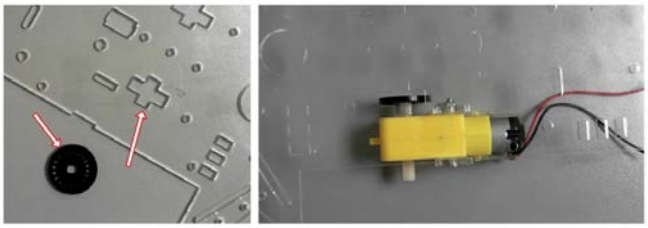


Figure 6.6 – Encoder wheel and slot close up

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Where the motors are mounted, the kits should have some encoder wheels, and a slot to read them through. The encoder wheels are also known as odometry, tachometer, or tacho wheels.

Simplicity

You don't want to use a complex or hard-to-assemble kit for your first robot build. I've repeated this throughout that with two-wheel drive, you want two motors with the wires soldered on. I steer clear of large robots, or unique and exciting locomotion systems, not because they are flawed, but because it's better to start simple. There is a limit to this, a robot kit that is a fully built and enclosed when bought leaves little room for learning or experimentation. An entirely premade robot may require toy hacking skills to customize.

Cost

Related to simplicity is cost. You can buy robot chassis kits from around $15, up to thousands of dollars. Larger and more complex robots tend to be far more costly. For this book, I am aiming to keep to the less costly options or at least show where they are possible.

Conclusion

So, now you can choose a chassis kit, with two wheels and a castor, two motors with wires soldered on them, slots, and encoder wheels. These laser-cut bases are not expensive, and are widely available on popular internet shopping sites as *Smart Car Chassis*, with terms like *2WD*:

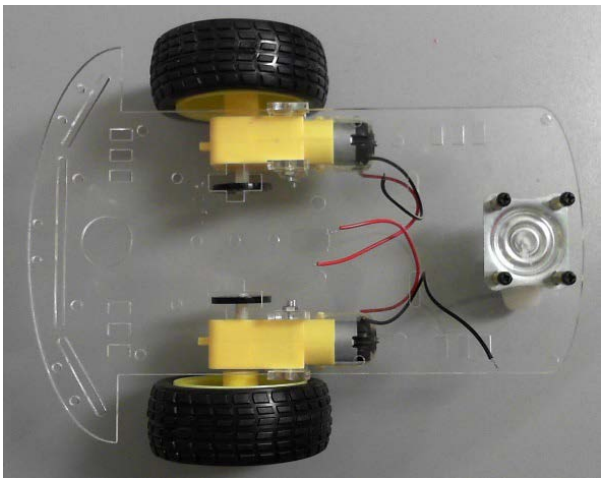


Figure 6.7 – The robot kit I'm using

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The kit I'm working with looks like *Figure 6.7* when assembled without the Raspberry Pi.

We've chosen a chassis kit. It is a medium-sized one that doesn't need large power handling. It has some space constraints but is not tiny. We can use these attributes to choose the motor controller.

Choosing a motor controller board

The next important part you'll need is a motor controller. You cannot connect a Raspberry Pi directly to DC motors, as they require different voltages and high currents that would destroy GPIO pins. Motor controller boards can also add interfaces to other devices like sensors and other motor types.

It is a vital robot component that will guide many later decisions. Much like the motors, there are some trade-offs and considerations before buying one:

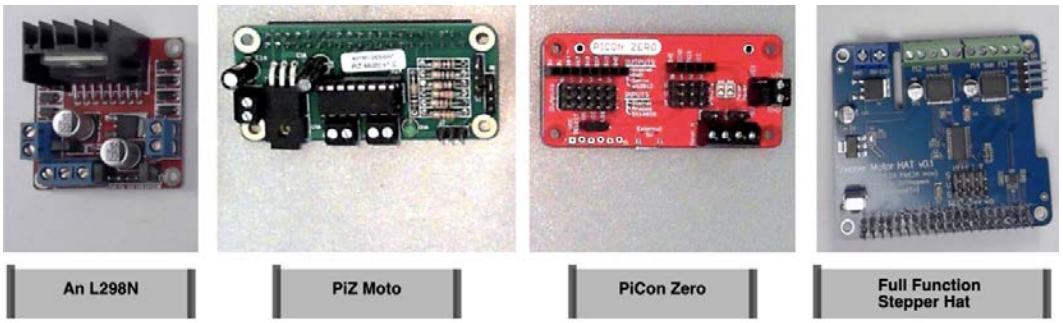


Figure 6.8 – A selection of motor control boards

Figure 6.8 shows a small sample group of motor controller boards. As we compare the requirements of our motor board, we refer to the boards pictured there as examples.

Integration level

Motor controllers may only control a motor (usually 2) like the L298N, containing the barest minimum to run this chip safely. They are not designed to sit on a Raspberry Pi and must be wired into the Pi's I/O output.

Controllers like the PiZMoto sit on top of the Raspberry Pi, reducing wiring. They still contain only a single circuit to control a motor. They also have pins to connect additional devices like a distance sensor and line sensor, which shift voltage levels, as we'll see later. This and the L298N require the Raspberry Pi to generate the PWM signal to move motors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

At a higher level of integration is the Full Function Stepper HAT. It has a chip that specializes in PWM control and is capable of driving multiple servo motors along with DC motors. This HAT frees up Raspberry Pi I/O pins by using I2C.

The PiConZero is the most integrated device here. It uses a microcontroller that is the equivalent of an Arduino to control DC motors and servo motors, to output to lights and take input from various sensors. It also uses I2C, reducing the number of pins needed.

Pin usage

When buying a motor controller in Raspberry Pi HAT form, it's essential to consider which I/O pins are in use. Having boards that make use of the same pins for incompatible functions won't work.

> **Important note**
>
> Although a HAT plugs into all pins, this doesn't mean they are all used. Only a subset of the pins is usually actually connected on a HAT.

To get an idea of how pins in different boards interact on the Raspberry Pi, take a look at `https://pinout.xyz`, which lets you select Raspberry Pi boards and see the pin configuration for them.

Using the L298N would require four I/O pins for the DC motors, and using sensors or servo motors with it requires further pins. The PIZMoto functions in a similar way to the L298N, requiring four I/O pins for DC motor control. The PiZMoto also assigns a pin for the line detector, two for distance sensing, two for LEDs, tying up a total of nine GPIO pins. The PiZMoto would also require additional support for servo motors.

The PiConZero and Full Function Stepper HAT both use the I2C bus. Using the I2C or serial bus makes efficient use of pins, as they use only two pins for I2C. Multiple devices can share the I2C bus, so even the use of these pins is shared. The PiConZero assigns some other pins to functions like the Ultrasonic device. The Full Function Stepper HAT leaves all other pins free to use while supporting the DC motors and 5 servo motors from a single I2C connection, making it one of the most flexible control boards in the selection.

Size

The choice of the motor controller size depends on the chassis and the size of the motors you have. In simple terms, the larger your chassis, the larger a controller you need. We specify the power handling capacity of a motor controller in amps. For a robot like the one shown in *Figure 6.7*, around 1 to 1.5 amps per channel is good.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The consequence of too low a rating can range from a robot that barely moves up to catching fire. Too large a controller has consequences for space, weight, and cost. A heatsink is one way to keep a controller cool while handling current but makes the controller larger, as shown in *Figure 6.9*:

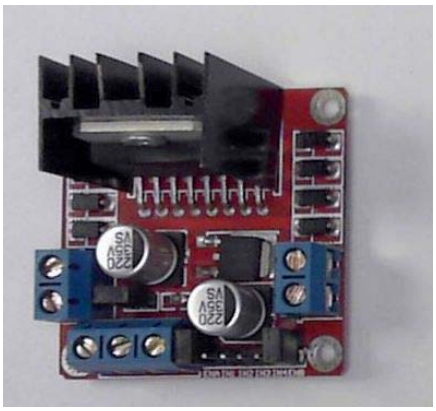


Figure 6.9 – An L298N with a heatsink

The level of integration can also contribute to size. A tiny board that stacks on a Pi takes up less space than separate boards.

Another size or shape consideration is whether the board restricts access to the Raspberry Pi camera port. Some boards, such as the Pimoroni Explorer HAT Pro, cover the camera slot entirely, making it tricky to use. Some boards have a slot for the camera port, like the Full Function Stepper HAT, and others are half-size hats (pHat) that don't cover the area, such as the PiConZero and PizMoto.

As we are using the camera in this book, it is a requirement that the camera port is accessible, either through a slot or by being a half-size (PHat) board.

Soldering

As you choose boards for a robot, note that some come as kits themselves, requiring you to solder parts on them. If you have experience with this, it may be an option, with a time cost. A small header is going to be a very quick and easy job. A board that comes as a bag of components with a bare board will take a chunk of an evening and could require debugging itself.

> **Tip**
>
> Soldering is an essential skill for robot building but is not needed for a first robot, so for this book, I mostly recommend pre-assembled modules.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Power input

Motors can require different voltages from the 5 V power supply the Raspberry Pi uses. They are also likely to consume a high current. We cover power supply choices later, but it is usually good to separate the power supply for the motor from the supply for the Raspberry Pi. All the boards in *Figure 6.8* have a separate motor power supply input.

The PiConZero and L298N let the user power the Raspberry Pi from the motor supply, but this can lead to reset and dropout conditions. Some motor interface boards, such as the Adafruit Crickit and the Pimoroni Explorer HAT Pro use the same 5 V supply for the motors and the Raspberry Pi, requiring a high current capable 5 V supply. My recommendation is to ensure the motor board has a separate power input for motors.

Connectors

Closely related to soldering and power input are the connectors for the motors and batteries. I tend to prefer the screw type connectors shown in *Figure 6.10*:

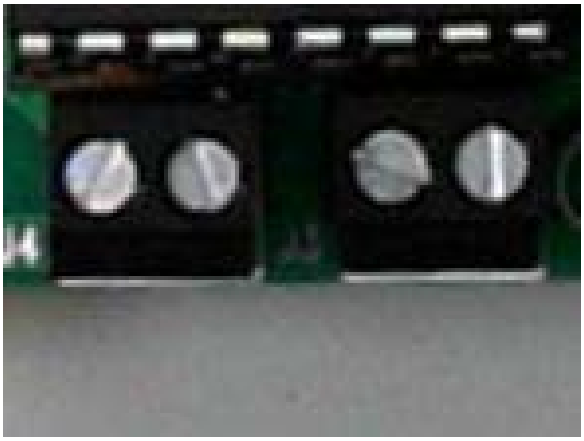


Figure 6.10 – Screw terminals for motor and battery connections

Other types may require motors with special connectors, or a robot builder to have crimping skills.

Conclusion

Our robot is space-constrained; for this reason, we use the Raspberry Pi HAT type form factor. We are also looking to keep the number of pins it uses low. An I2C-based HAT lets us do this. The **Full Function Stepper Motor HAT** is shown in *Figure 6.11*. It's also known as the Full Function Robot Expansion Board and gets us access to all the Pi pins while being a powerful motor controller:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

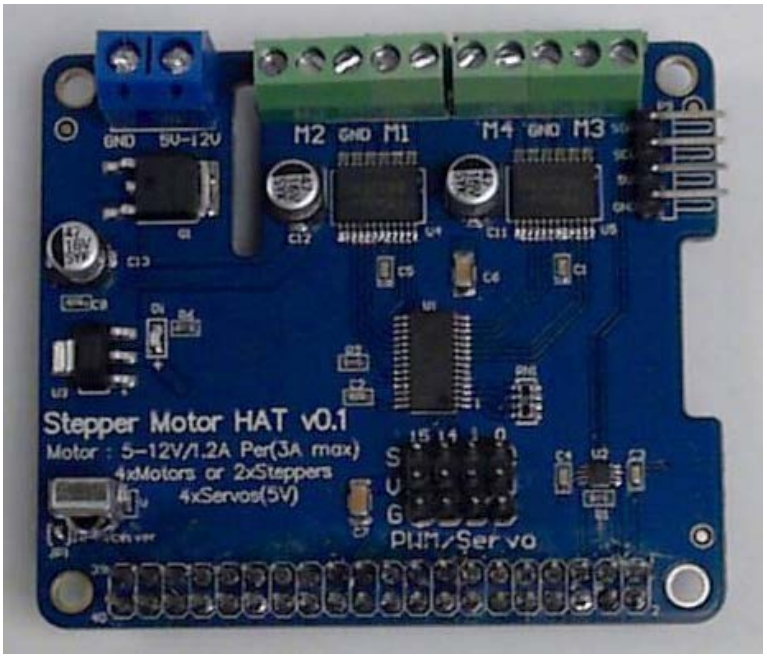


Figure 6.11 – The Full Function Stepper Motor HAT

It's available in most countries, has space for the ribbon for the camera, and controls servo motors. I recommend this HAT for the robot in this book. Our code is directly compatible with boards based on the PCA9685 chip used on the Full Function board. With some minor changes in code and wiring, the 4tronix PiConZero would also be a suitable choice.

Powering the robot

The robot needs power for all its parts. We must consider two major power systems: the power for all the digital parts, such as the Raspberry Pi and sensors, and then the power for the motors.

Motors need a separate power system for a few reasons. First, they consume far more electrical power than most other components on the robot. They may require different voltages; I've seen low-voltage, high-current motor supplies and high-voltage supplies too. The other reason that they need their own power system is that they can cause interference. They can pull enough power that other circuitry has brownouts. A brownout is when circuitry has a voltage drop that is low or long enough to get into an inconsistent or reset state. Resets can lead to SD card corruption on a Pi. Motors can also introduce electrical noise to a power line as they are used, which could cause digital parts to misbehave.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

There are two primary strategies for powering a robot with motors:

- **Dual batteries**: The motors and the rest of the robot have entirely separate sets of batteries, ensuring that their power is independent.
- **Battery eliminators**: Use a single battery or are set with a BEC/regulator. A **BEC** is a **battery eliminator circuit**, shown in *Figure 6.12*:

Figure 6.12 – Picture of a BEC

Dual batteries are the surest option to avoid any brownout, loss of power, or interference issues. This option takes more space than a BEC. However, the dual power option, with a USB power bank for the Raspberry Pi (shown in *Figure 6.13*) is a simple and effective way to avoid power issues. Choose one with small outercase dimensions, but a high power-rating, such as 10,000 mAh, and an output of at least 2.1 A:

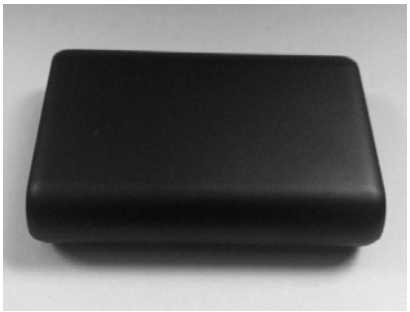


Figure 6.13 – The USB power bank from Armbot – slightly old and battered, but still effective

Check that the USB power bank comes with the cable to connect the Raspberry Pi – a Raspberry Pi 3 (A+ and B+) requires a USB micro connector, and the Raspberry Pi 4 requires a USB-C connector. Power bank outputs usually use a USB-A connector. If they are not included with the power bank, you will need to buy one of these cables too.

Motor controller boards sometimes have power supplies to regulate power for a Pi from the motor power. These often have too low an output rating. They can be very inefficient, wasting a lot of battery power. Unless using very high current handling batteries, they are very likely to lead to brownouts.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

A battery eliminator circuit is lighter and takes up less space. Types include a BEC, uBEC, switching supply, or a regulator. Battery eliminators usually need high-power batteries such as Li-Ion types. By sharing a supply with motors, batteries need enough current capacity not to be vulnerable to the voltage drops that cause controller resets and line noise for the controller. This requirement affects switching power supply Pi SHIMs like the Wide Input SHIM and the power supplies built into some motor controllers.

You need to ensure that the BEC output can handle at least 2.1 A, preferably more. It's common to see 3.4 A and 4.2 A power banks. UBECs with 5 A ratings are also reasonably common.

To keep things simple in this robot, and not have to deal with reset issues, we use the dual battery approach and accept the cost in bulk and weight:

Figure 6.14 – The 4 x AA battery box we use with the motors

For the motors, 4 x AA batteries work. I recommend using nickel metal hydride rechargeable batteries for them. That is not just because you can recharge them, but also because they can deliver more current if needed than alkaline batteries. To save space, we use the *two up/two down* or *back to back* configuration, like the battery box shown in *Figure 6.14*.

In conclusion, for our battery selection, we will use a 4 x AA metal hydride set for the motors and a USB power bank for the Raspberry Pi and logic.

We have now made part selections. We know which chassis, controller, and battery configurations we will use. We have seen the size of the motors and wheels, and have, in previous chapters, selected a Raspberry Pi model to use. Before we go ahead and buy all this, we should make a further check to see that it will all fit. In the next section, let's learn about test fitting robot parts.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Test fitting the robot

I recommend test fitting before actually ordering parts. The test fit helps a builder be more confident that components fit, and you'll know roughly where these parts go. This step saves you time and money later.

You can use paper and a pen for test fitting, or an app such as `diagrams.net`. First, I find the dimensions for all the parts. *Figure 6.15* has a screenshot from Amazon showing how to spot product dimensions:

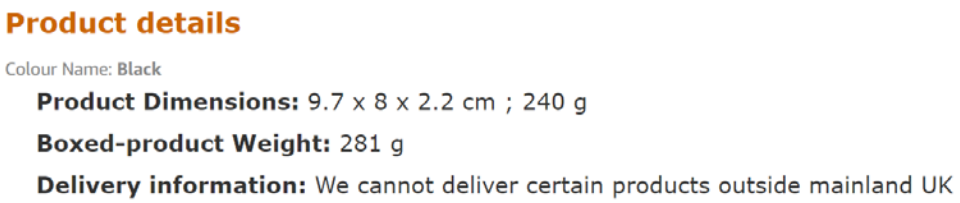


Figure 6.15 – Finding product specifications

Some information-digging is needed to find these for your parts. For each, first find a shop you can buy them at, such as Amazon, several online shops, or eBay. You can then search for or ask for information about the dimensions of each board or item. Make sure you are looking at the dimensions of the part and not its packaging:

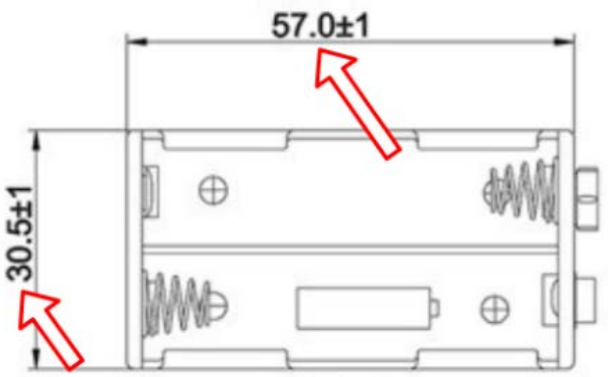


Figure 6.16 – A battery box product drawing with dimensions

You can find diagrams like the battery box in *Figure 6.16* by doing image searches on dimensions or part datasheets. In this case, the dimensions are in mm. The ± signs show the manufacturing variation of plus or minus the next number. When test fitting, err on the higher side, so take 57±1 as 58 mm for that dimension.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

So, the dimensions I have are as follows:

- The Raspberry Pi 3a+: 65 mm x 56 mm.
- The chassis: Mine suggests it is 100 mm x 200 mm. Be aware that the dimensions here are outer dimensions and include the wheels.
- The motor controller fits over the Pi, so it is counted here as the Pi. This controller makes things taller but is only really a concern for a multi-level robot chassis.
- The 4 x AA battery box: The type I suggested is 58 mm x 31.5 mm.
- The USB power bank: 60 mm x 90 mm.

For this, drawing rectangles to scale is enough detail. In `diagrams.net`, create a new blank diagram:

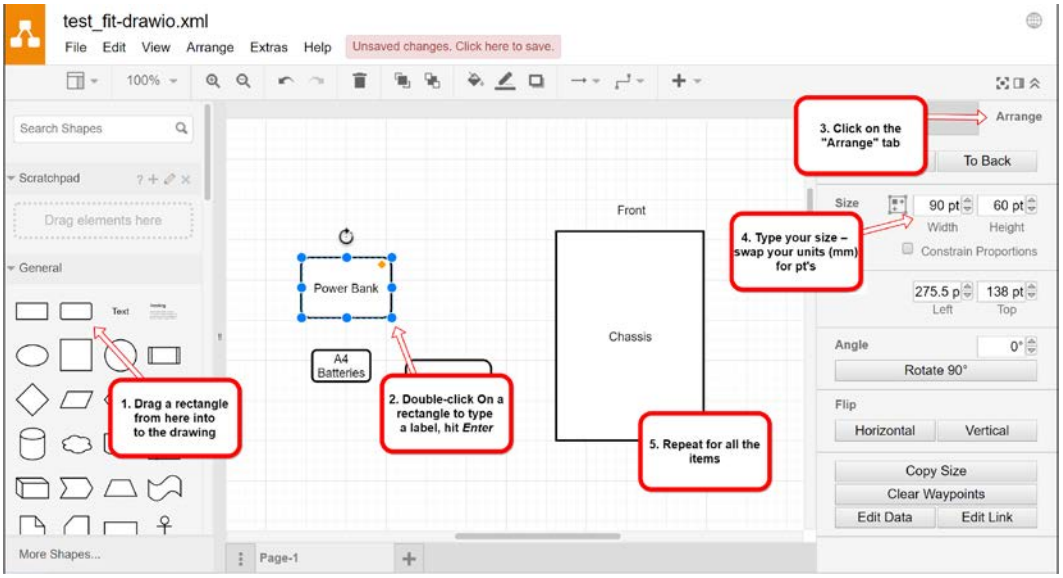


Figure 6.17 – Using diagrams.net to create test fit parts

You can follow the instructions to create test fit parts as shown in *Figure 6.17*:

1. Use the general palette on the left to drag out rectangles.
2. It helps to label each part clearly. Double-click a rectangle and type a label into it. Press *Enter* to accept the label. I've added a text label on the *front* of the chassis.
3. Select the item so it has a blue highlight. Click the tabs on the right to select the **Arrange** tab.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Here, type your dimensions (swap millimeters for points) into the **Width** and **Height** boxes.
5. You can repeat *steps 1* through *4* for all the items and then drag the parts together:

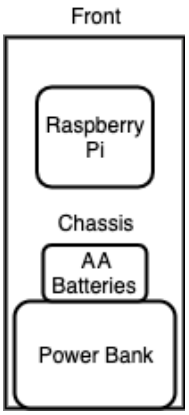


Figure 6.18 – The test fit

The Pi should be near the front of the robot, as we'll later have sensors here, and the motor wires can go forward into it. In *Figure 6.18*, I've dragged the rectangles into place. `Draw.io` helps you here by showing blue guidelines for centering and aligning objects. I put the power bank at the rear, with the AA batteries closer to the Pi so they can go into the motor controller easily.

The parts look like they fit. It's not 100% accurate, but good enough to say this probably works.

Now, it's time to buy parts. My shopping list looks like this:

- The chassis kit.
- The Full Function Stepper Motor HAT.
- 4 x AA battery box.
- 4 x metal hydride AA batteries. If you don't have one, you need a charger for these too.
- 1 x USB power bank able to deliver 3 amps or more.

Now you've selected your parts and made the trade-offs. You've followed that by test fitting them to see how they're laid out and check they all fit. It's time for you to go and buy your parts and chassis. Once done, come back here because then we can begin our build.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Assembling the base

Assuming you bought a chassis similar to mine, you can assemble it with these steps. For a completely different chassis, I strongly recommend consulting the documentation for assembly instructions. A chassis that is very different from the recommendations here may make it harder to follow the next few chapters.

Some parts may be covered in a layer of paper (shown in *Figure 6.19*). This layer prevents the plastic from getting scratches and can be safely removed. You can do this by getting a nail under it or using a craft knife. It's not essential to remove it, but a robot looks better without it:

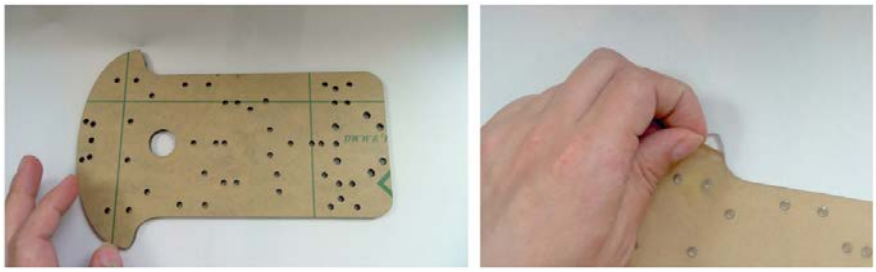


Figure 6.19 – Removing the protective backing from robot parts

With the laser-cut kits that use the yellow motors, there are two main ways the motors are attached. One type has plastic brackets, and the other has metal motor brackets. Given that you may buy kits of either style, let's look at how the kits could differ. The difference only matters in assembly steps, so buy what is available to you.

For a kit with a plastic motor bracket, you should have the parts shown in *Figure 6.20*:

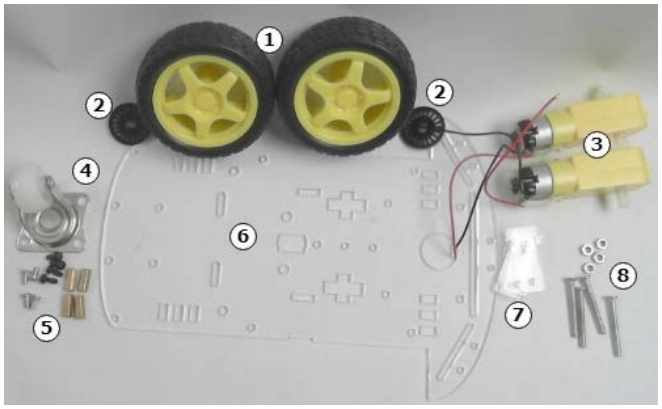


Figure 6.20 – Robot kit parts

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In the kit, you should have the following:

1. Two wheels.
2. Encoder wheels.
3. A pair of motors with wires.
4. A castor wheel.
5. Bolts and brass standoffs to mount the castor wheel. I've replaced one set of bolts with non-conductive nylon ones. You should be able to do the same from the nylon standoff kit.
6. The chassis plate.
7. Plastic brackets to mount the motors. Your kit may have metal types, which work slightly differently and come with four extra screws.
8. Four bolts and nuts to mount the motors.

Figure 6.21 shows how the metal motor bracket parts differ:

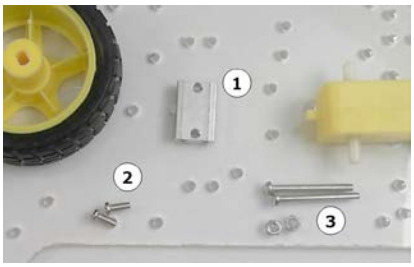


Figure. 6.21 – Metal type motor bracket

In the kit, you should have the following:

1. The metal bracket replaces the plastic brackets here.
2. Chassis to bracket bolts – instead of slotting through, these brackets need to be bolted to the chassis.
3. The long bolts for the bracket to the motor will still be the same.

> **Important note**
> The other components not shown are going to be very similar to this.

Now, let's see how to attach encoder wheels.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Attaching the encoder wheels

We'll start by attaching the encoder wheels to the motor. We need these for a later chapter on sensors:

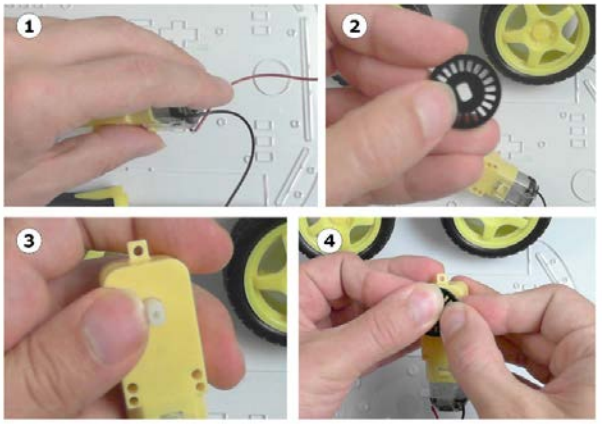


Figure 6.22 – The encoder wheel is attached to the motor

Follow the steps in *Figure 6.22*:

1. Observe on which side the wires are connected to it. The encoder wheel should attach on the same side as the wires.
2. Find the axle hole with flattened sides in the encoder wheel.
3. The axles on the motors are shaped to match this hole.
4. Line up the axle hole with the motor axle on the same side as the wires and gently push it on. It should have a little friction. Repeat this for the other motor.

You should now have two motors with encoder wheels on them, on the same side as their wires. Next, we fit the motor brackets to the robot.

Fitting the motor brackets

There are two kinds of motor brackets commonly used in the pictured laser-cut chassis. You should use the section that is most similar to the type you have.

Fitting plastic motor brackets

> **Important note**
> If you have the metal type, skip this section.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

To fit the plastic type of bracket, first look for the slots to attach it, as shown in *Figure 6.23*:

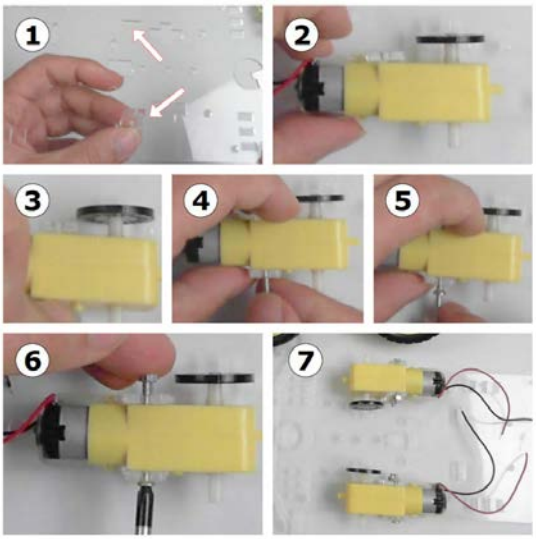


Figure 6.23 – Plastic motor mount

To fit these, follow these steps:

1. The arrows point at the slots. Push the plastic brackets through the slots.
2. Push the motor against the bracket. Note that the wires and the encoder wheel face the inside. The encoder should be under a cutout in the chassis body for it.
3. There is a slot in the chassis for an outer bracket, sandwiching the motor. Push another bracket into this slot.
4. Push the long screws through from the outside.
5. Then, push a nut onto the screws and use a screwdriver to screw them in.
6. For the nut closest to the chassis, one of its flattened edges should hold it in place as you tighten the screw. For the outer nut, use a spanner or pliers.
7. You need to repeat the same steps for the other side.

This section has covered attaching motors to a chassis with plastic brackets. If you have metal brackets, you should use the following section instead.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Fitting metal motor brackets

Please skip this section if you have assembled the plastic motor brackets. The metal type of bracket is slightly different; *Figure 6.24* shows its assembly:

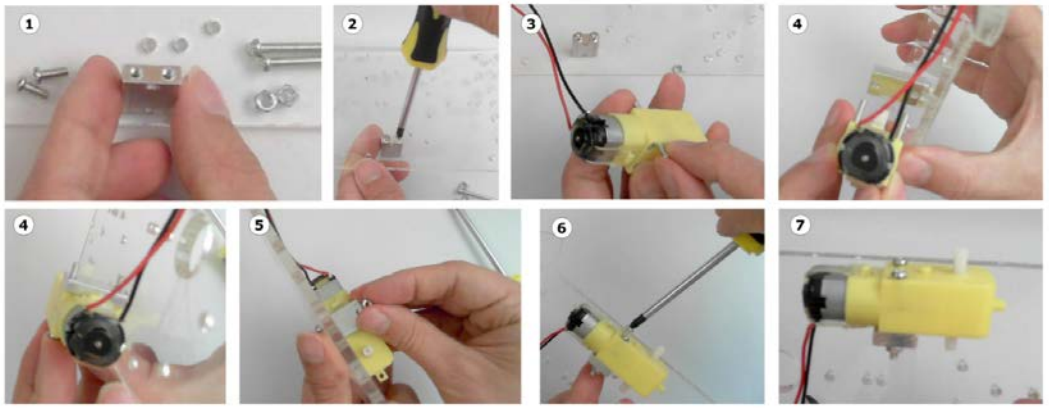


Figure 6.24 – Assembling metal motor brackets

To do this, perform the following steps:

1. You should be able to see two small screw holes in the top of the bracket; these are threaded. There are two short screws per bracket.
2. The chassis has holes in the wheel mount area that match these. Line these up, and then screw the short screws through the chassis holes into the bracket.
3. Take the motor and ensure the wires are facing away from you. Push the long screws through the two holes in the motor.
4. Then, take this motor assembly and push the long threads into the holes on the side of the bracket.
5. It should fit through like this.
6. Now, push nuts onto the threads that stick out the other end of the bracket.
7. You can tighten the nuts furthest from the chassis with pliers, or a spanner and a screwdriver. The closer nuts catch on one flat side, so you'll only need a screwdriver.
8. You now have the completed assembly and need to repeat these steps for the other side.

After completing either set of steps, you should now have a motor mounted on each side. We will use these for the drive wheels. But first, we need the castor wheel.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Adding the castor wheel

Next, it is time to fit the castor wheel. The castor wheel balances the robot. It is not driven, so will be dragged along by the robot. It's important that it has little friction. *Figure 6.25*, along with the steps that follow, will teach you exactly how to get this done:

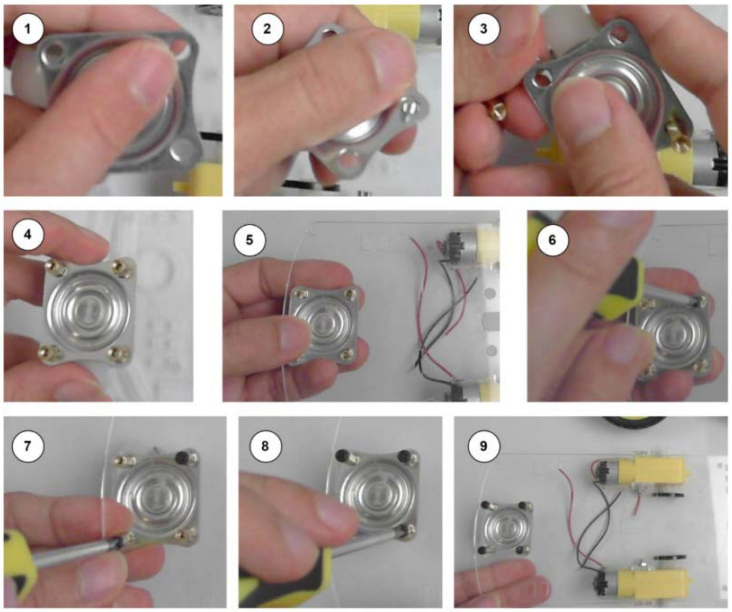


Figure 6.25 – Fitting the castor wheel

Use these steps with the *Figure 6.25*:

1. This is the castor. It has four screw holes.
2. You need to push a metal screw through the hole, so the thread is facing away from the wheel.
3. Now, screw one of the brass standoffs into this screw.
4. Repeat this for the four other sides.
5. Line the other side of the standoffs with the four holes on the chassis. Note that this castor wheel is a rectangle, not a square. Make sure the wheel is facing down.
6. Push one of the screws through and screw it down.
7. I suggest you screw the opposite corner.
8. This makes the remaining two screws easier to put in.
9. The castor should now be attached to the robot like this.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

With the castors attached, the robot will balance, but it needs wheels to move anywhere, so let's add wheels.

Putting the wheels on

The wheels now need to be pushed on, as shown in *Figure 6.26*:

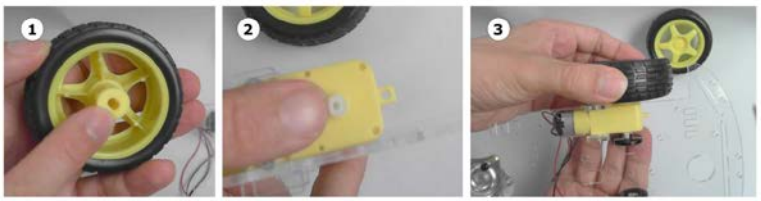


Figure 6.26 – Fitting the wheels

Follow these steps:

1. First, note that they have two flattened sides in their axle hole, like the encoder wheel.
2. Line the wheels up with the axles, taking into account the flat edges, and push them on. Do not push on the wires or the encoder disk as they may break.
3. Sometimes, rotating the wheels until they push in helps. You should be able to push the wheel on, being sure to support the motor from the other side. After doing this, you may want to realign the encoder wheels with their slots.

The wheels are on the motors and the robot is starting to take shape. The robot should now be able to sit on three wheels. You can roll it around manually, but it's not quite ready to drive itself yet.

Bringing the wires up

A last minor step in chassis assembly is to bring the wires up. The motor controller will be on the top of the robot, so the wires need to be above the chassis too:

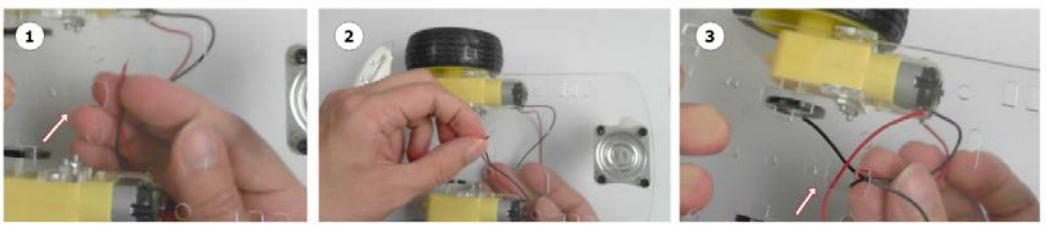


Figure 6.27 – Bringing the wires up

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Follow the steps in *Figure 6.27*:

1. First, gather the two wires from one motor. Locate the small slot in the middle of the chassis.
2. Push the wires through.
3. Gently pull them through to the top of the chassis, so they are poking out as shown. Repeat this for the other motor.

We should now have a robot that looks like *Figure 6.28* (motor brackets vary):

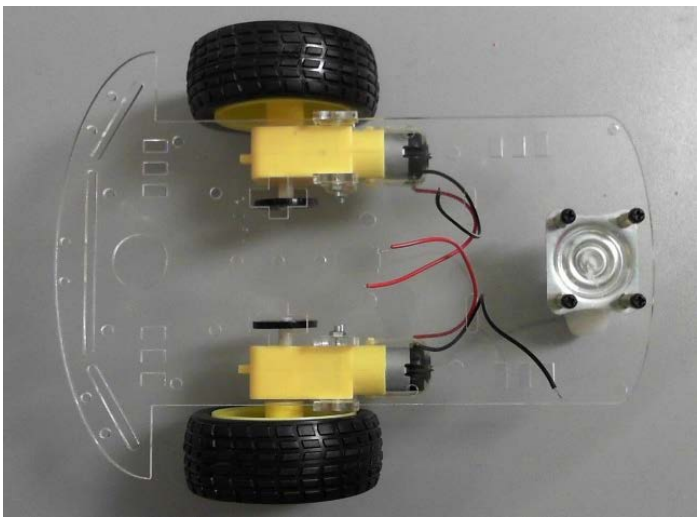


Figure 6.28 – The assembled chassis

With the motors in place and wheels ready, we can see how the robot will move. You've built the mechanical section of the robot. The wires are in position, and we're now ready to add the electronics. We'll start by adding the central controller, the Raspberry Pi.

Fitting the Raspberry Pi

We will not fit the motor controller yet – we'll address that in the next chapter, but we can mount the Raspberry Pi now and prepare it to have other boards connected to it. We need to put standoffs on the Pi to bolt it onto the chassis, but leave room for the motor bracket mounting, and later sensors that go under the Pi:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

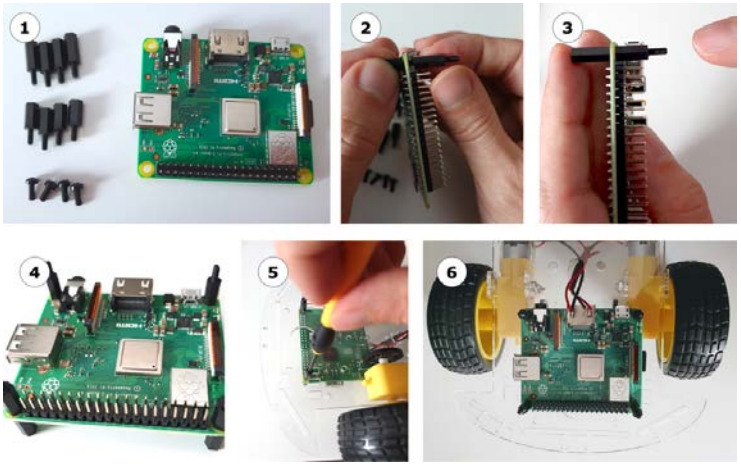


Figure 6.29 – Fitting the Raspberry Pi

Perform the steps shown in *Figure 6.29*:

1. You need a small posidrive screwdriver, a small spanner or pair of pliers, 4 x M2.5 5mm screws, 4 x M2.5 8 mm standoffs with threads, 4 x M2.5 12 mm standoffs, and the Raspberry Pi.
2. Push an 8 mm standoff thread up through the screw hole from the bottom of the Pi.
3. Then, screw a 10 mm standoff onto the top of these, with the thread facing upward, using the pliers/spanner to hold the standoff.
4. Repeat for all four corners.
5. Line two of these up with some slots or screw holes on the chassis and screw them in from underneath.
6. On the chassis I used, there were only two holes that line up, so I screwed those in and used the other standoffs to keep the Pi level.

This robot now has a main controller, which will be able to run code and command the robot. However, before it will do much, the controller and motors need power.

Adding the batteries

There are two sets of batteries that you have bought: the 4 x AA battery holder, as in *Figure 6.14* (with a set of rechargeable metal hydride batteries), and a USB power bank, as in *Figure 6.13*. The power bank contains a lithium-ion cell and a USB charging system.

We mount these on the back of the robot, where they'll counterbalance some of the sensors that we'll add later.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Setting up the USB power bank

Do not connect the power bank to the Raspberry Pi yet (or be sure to log in and shut it down properly before pulling the power cable out if you have done so):

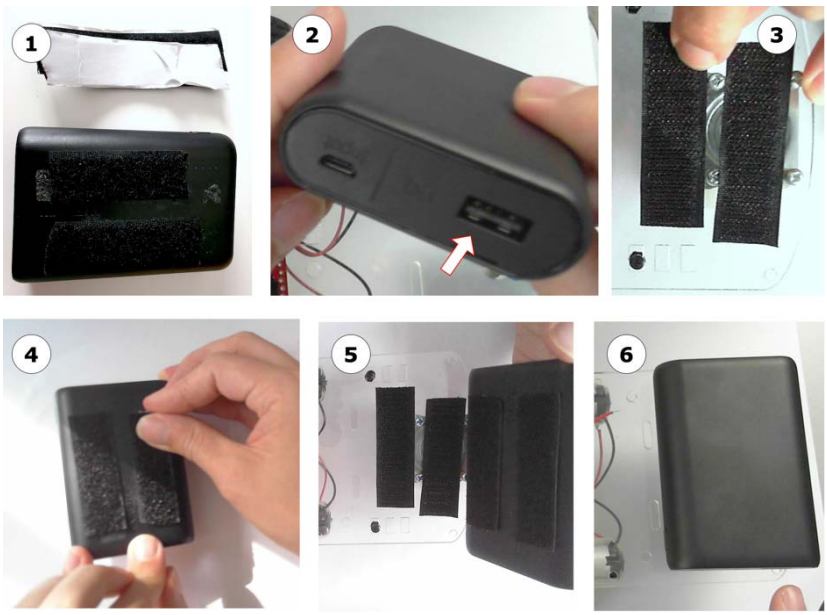


Figure 6.30 – Mounting the power bank

To attach a USB power bank to the chassis, use the following instructions and *Figure 6.30*:

1. For this power bank, we use some hook and loop tape.
2. Take a look at the power bank and note that one side has the USB connector on it. This connector should end up on the left of the robot. If it has an LED charge display, this should be on top.
3. Measure two lengths of hook and loop tape. Stick the rough sides in two strips on the robot.
4. Stick the soft sides to the power bank, and line these up with the robot.
5. Push the power supply down, so the hook and loop have stuck together. This connection holds well.
6. This is how the power bank should sit on the robot.

Alternatives are to use sticky tack (for a cheap but flimsy connection), cable ties, double-sided tape, or rubber bands to hold the battery in place. These are suitable for different sizes of batteries.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Mounting the AA battery holder

We will also use hook and loop tape to add an AA battery holder for motor power. We need the holder to be easy to remove so the batteries can be replaced. Hook and loop tape is a convenient choice for this. Adding the AA battery holder is shown in *Figure 6.31*:

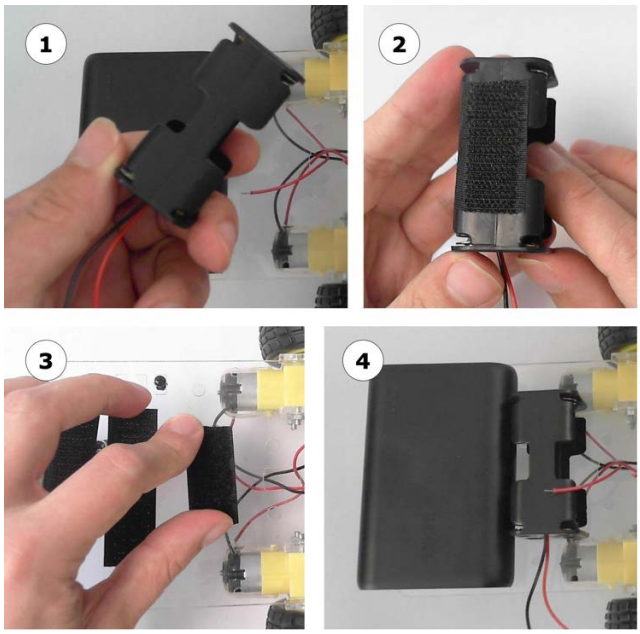


Figure 6.31 – Mounting the AA battery holder

To mount the AA battery holder, use *Figure 6.31* with the following steps:

1. Make sure that the AA battery holder does not have batteries in it yet.
2. Cut and stick a small strip of hook and loop tape to the bottom of the battery holder.
3. Stick the opposite hook and loop strip to the robot just in front of the power bank (removed for clarity – you do not need to remove the power bank).
4. Attach the battery holder here using the hook and loop strips.

At a pinch, sticky tack can be used for this, but remember that the AA battery box needs to be removable to replace the cells in it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The completed robot base

You have now completed the robot base, which should look something like *Figure 6.32*:

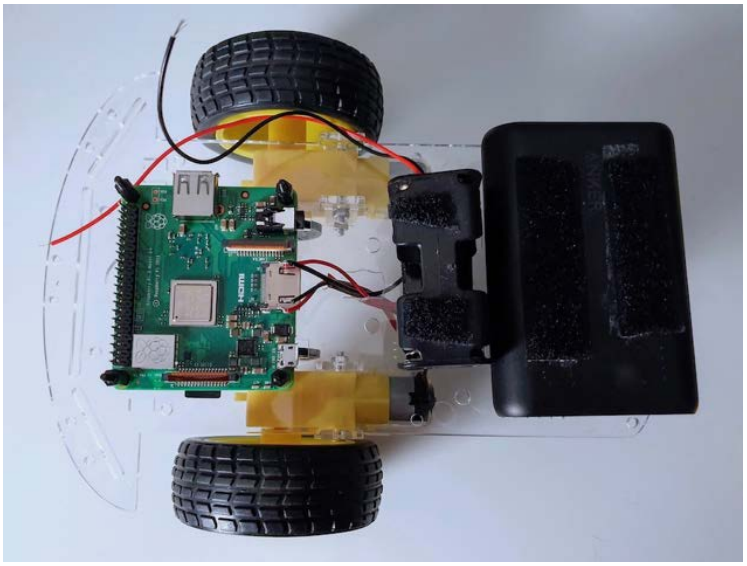


Figure 6.32 – The completed chassis

You've now built your first robot chassis! Hopefully, the first of many. With the robot chassis completed, sporting wheels, a Raspberry Pi, and battery compartment, it is nearly ready to roll. It needs some wiring to a motor controller and code to really come to life.

Connecting the motors to the Raspberry Pi

In this section, we will connect the motors to the Raspberry Pi. Once we have connected them, we can use code on the Raspberry Pi to control the motors and make the robot move. *Figure 6.33* is the block diagram for the robot that we are building in this chapter. We will be using the Full Function Stepper Motor HAT as the controller board, calling it the Motor HAT for short.

This block diagram is similar to the type shown in *Chapter 3*, *Exploring the Raspberry Pi*. First, it starts with the Raspberry Pi, here in gray, as we've chosen that as our controller. Connected to the Pi is the Motor HAT, with instructions flowing from the Raspberry Pi to this board. The Motor HAT and its connections are highlighted as we are adding these parts in this chapter. As we build on this block diagram in later chapters, existing components will be in the color gray. Added components will be highlighted with red to show what is new. Finally, the two motors are added to the left and right of the Motor HAT, with arrows going from the Motor HAT to show it is controlling the motors:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

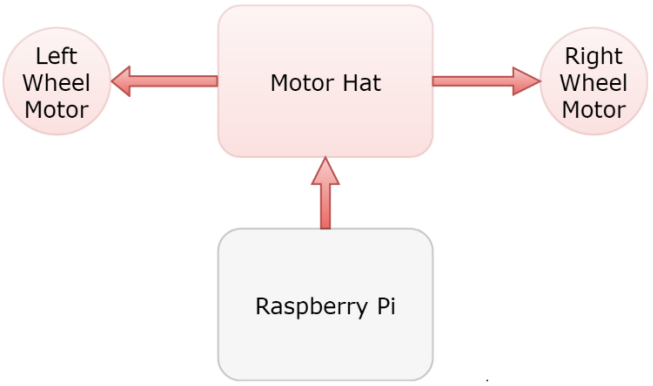


Figure 6.33 – Block diagram of the robot

The first step in connecting the motors is to fit the Motor HAT onto the Raspberry Pi. *Figure 6.34* shows the Motor HAT:

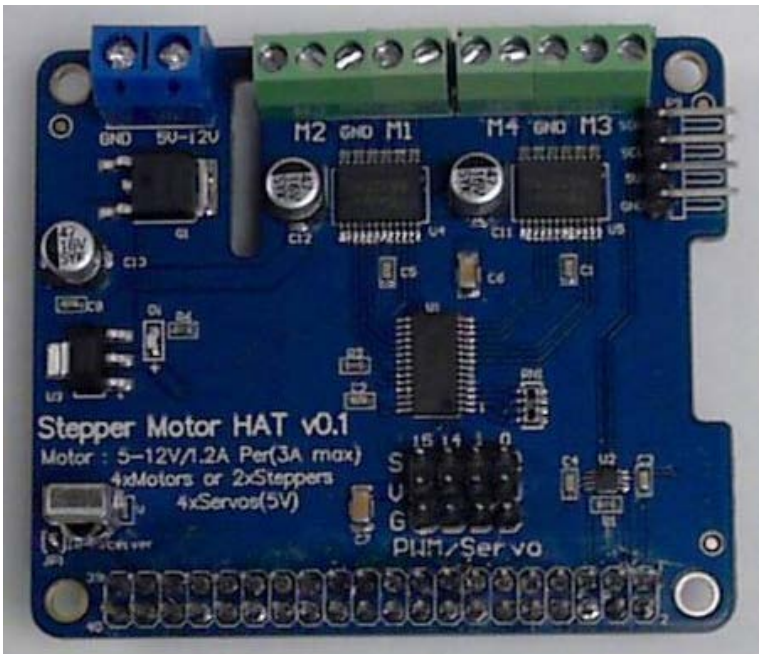


Figure 6.34 – The Full Function Stepper Motor HAT

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's attach this HAT to our robot and wire it in so we can start programming our robot:

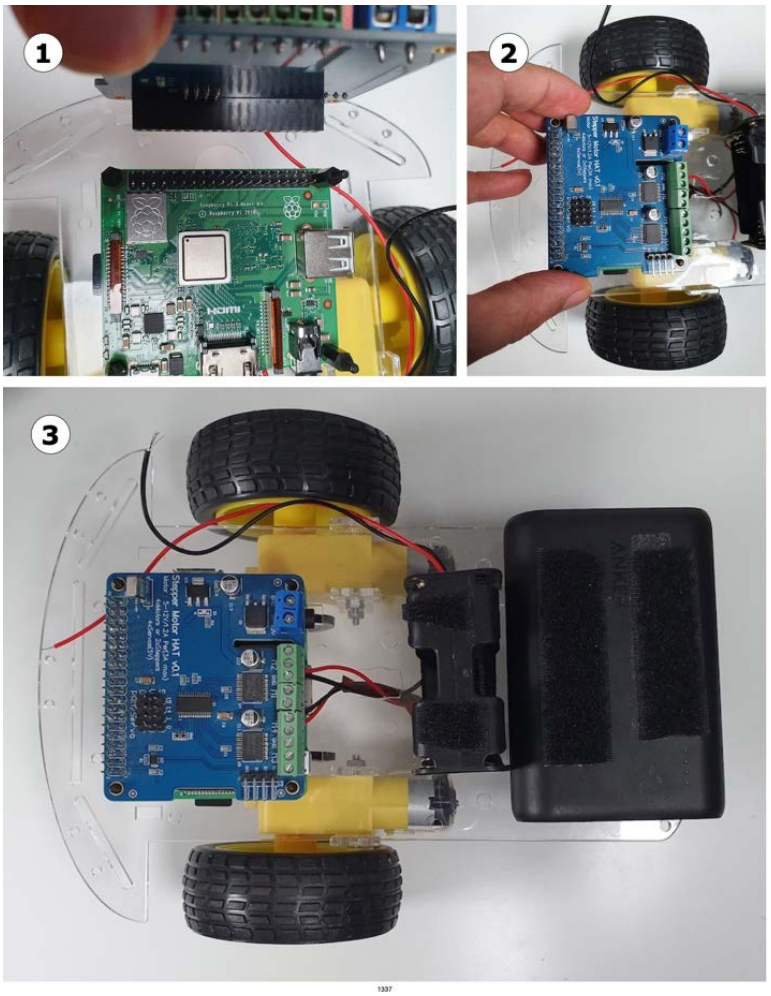


Figure 6.35 – Fitting the motor controller

Refer to *Figure 6.35* and follow these steps:

1. Line up the motor board socket with the Pi header. The four holes in the corners should also line up with the screw threads facing up.
2. Gently and evenly push the motor board onto the Raspberry Pi, guiding the screw threads through. Continue until the board is firmly seated on the GPIO header.
3. The robot should now look like this.

The board is now attached to the Raspberry Pi, and we can start wiring it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Wiring the Motor HAT in

We will now wire in the Motor HAT, first to the motors, and also partially to the motor batteries. We wire the motor batteries so the motors have their own source of power and do not cause low-power reset conditions on the Raspberry Pi. The motor controller needs to be wired to the motors to power and control them.

Figure 6.36 shows how we wire this up. Don't wire in the ground (black) wire on the batteries until we are ready to power it up. I suggest using a little insulation tape to tape the tip of it down to a plastic part of the chassis, so it does not catch on anything. Leaving ground unwired lets us use it as a kind of makeshift switch:

> **Important note**
>
> The black wire on a battery may be referred to as ground, GND, and negative. The red wire can be referred to as **positive** (**+ve**), **vIn** (**voltage In**), or by the voltage input rating – for example, 5 V – 12 V.

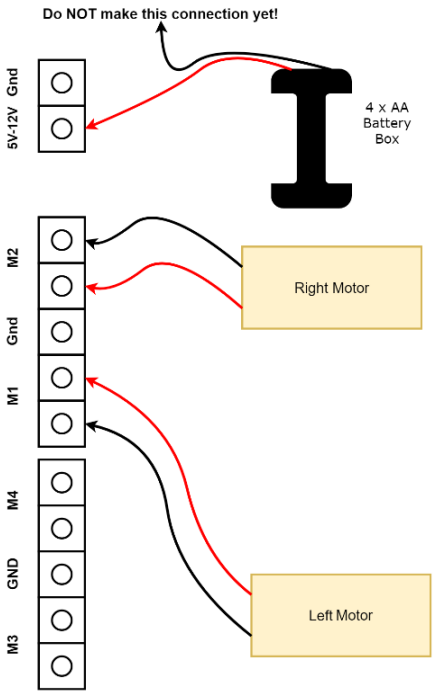


Figure 6.36 – How to wire up the motors and batteries

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 6.37 shows the steps for the connections:

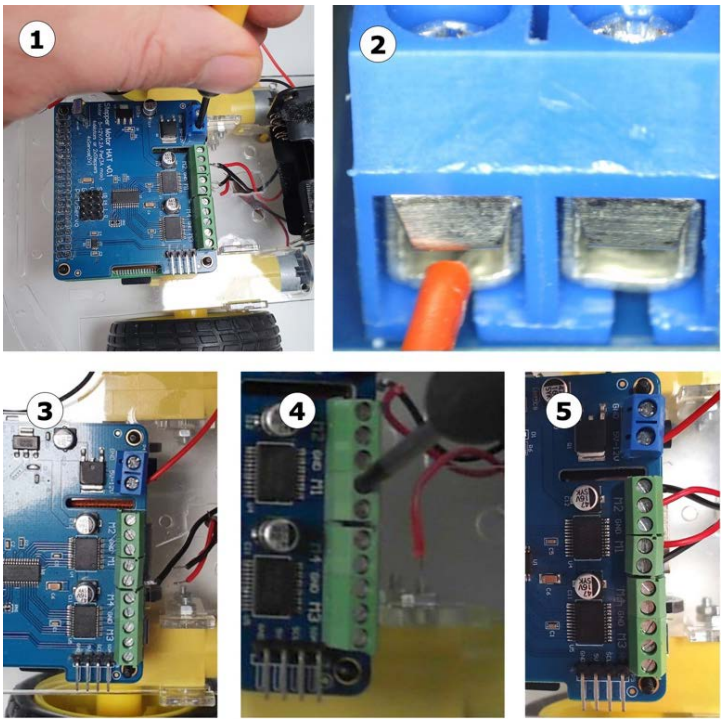


Figure 6.37 – Steps for connecting the wires

Perform the following steps with reference to *Figure 6.37* to connect the wires:

1. Loosen up the screw terminals for the 5 V – 12 V connection, GND, the two M2 connectors, and the M1 connectors.
2. Push the red wire from the AA battery box into the screw terminal marked 5 V – 12 V, so the metal part of the wire (the core) is in the slot formed by the metal cover.
3. Screw it down firmly, so the wire does not pull out easily. Ensure that the core is being gripped, and not the plastic outer layer (its insulation).
4. Repeat for the motor terminals, making the connections shown in image 4.
5. The result should look like image 5.

We have now connected the motors to the controller, so it can drive them. We have partially connected the battery power, but we have left one connection free, so we can use this as a power switch. Let's try powering up the robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Independent power

So far, although we have set up a headless Raspberry Pi, we have still been plugging it into the wall. Now it is time to try powering it independently. We will power up the motors from the AA batteries, and the Raspberry Pi from the USB power bank. We will see lights on the devices to tell us they are powered:

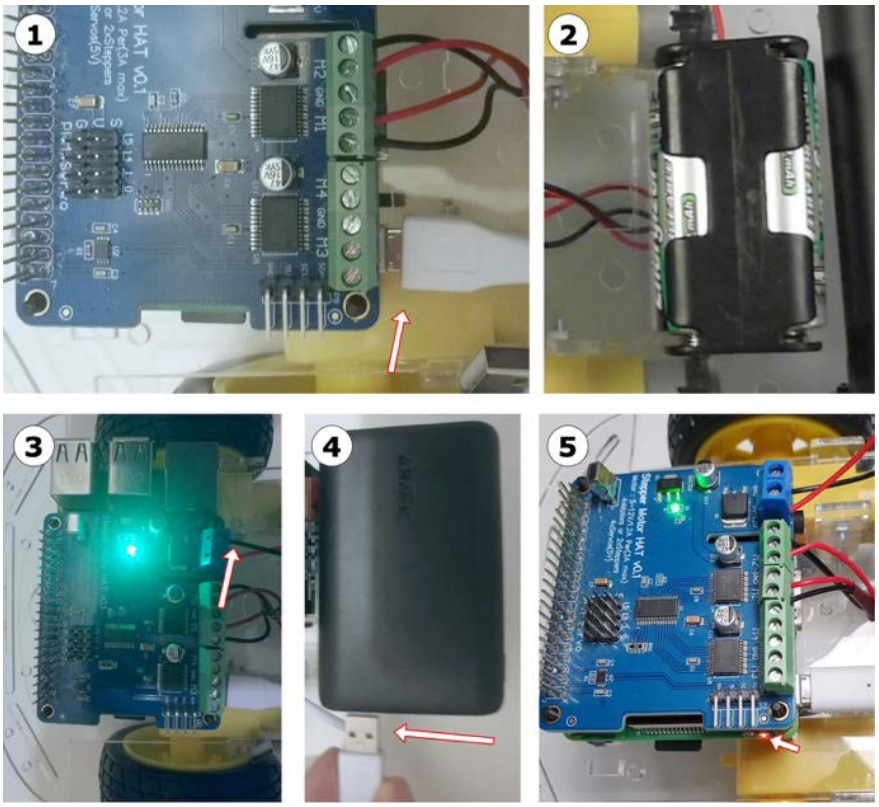


Figure 6.38 – Going on to independent power

Follow the steps shown in *Figure 6.38*:

1. Plug the Micro USB (tiny) end of the cable into the Pi in the USB micro-socket indicated by the arrow.
2. Fit the four AA batteries; you may need to pop the battery box up and push it back down again after this.
3. You can power up the motor board now. Connect the black wire from the battery box into the GND terminal indicated by the arrow, next to 5 V – 12 V. When you do so, a light appears on the motor board to show it is active.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Turn on the Pi by plugging the USB A (wide) end into the power bank. The intention from here is to keep the micro-USB tiny end in, and only connect/disconnect the USB A (wide) end when powering the Pi.
5. The Raspberry Pi and motor board are now powered, as shown in image 5.

Congratulations, your robot is now running on independent power, freeing the Raspberry Pi from the wall and giving the motors power too:

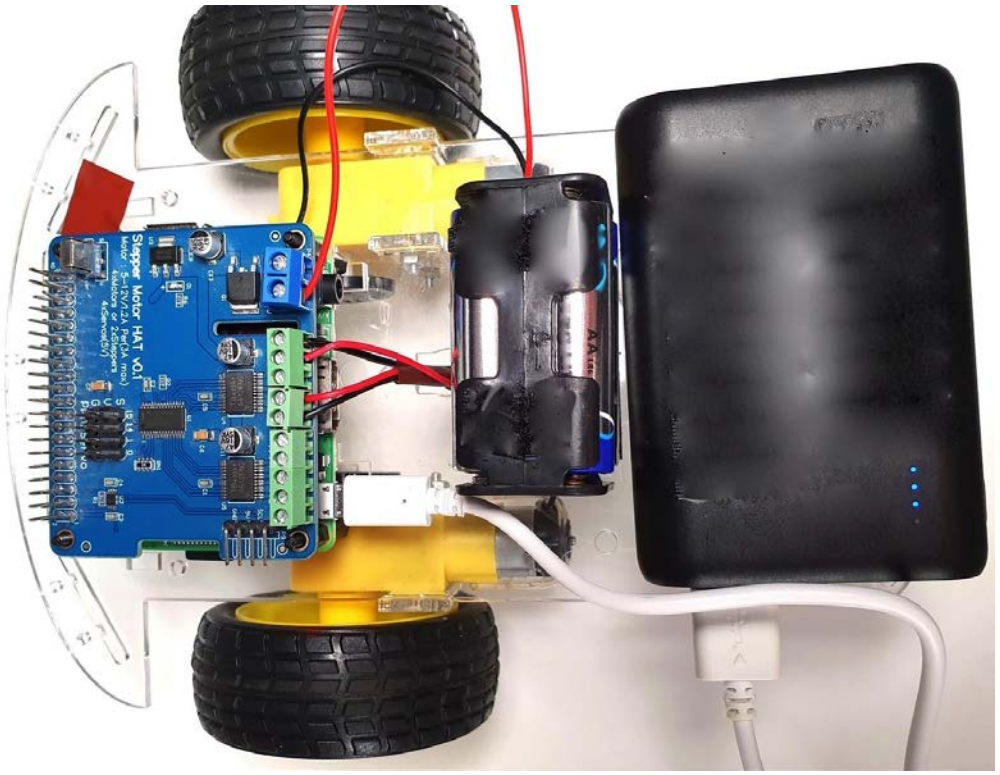


Figure 6.39 – The complete Chapter 6 robot

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The photograph in *Figure 6.39* shows our robot so far. It has motors on the chassis, with a Raspberry Pi and a motor control board fitted. The robot has a power supply for the Raspberry Pi – currently powered up. It has a power supply for the motors, currently disconnected with the black ground wire carefully taped out of the way of the rest of the robot. The motors are wired into the control board. This is enough robot to start making things move.

> **Important note**
>
> SD cards can be corrupted by removing power from the Pi without shutting it down. When turning it off, log in with PuTTY and use `sudo poweroff` before removing the power.

Your motors are ready to drive, and the Raspberry Pi is ready to run code without needing to be plugged into a wall. Combining independent power with headless Wi-Fi control means the robot can be driven around by instructions from your computer.

Summary

In this chapter, you've now learned how to choose the parts for a robot by reasoning and making some important design decisions. You used a simple tool to test fit these parts and see what works before buying anything. Finally, you bought the parts and built your starting robot platform.

By considering the trade-offs and test fitting again, you have gained skills for planning any hardware project, including finding dimensions on the datasheets/vendor websites, making a simple test-fit sketch, and considering how the parts will interact together. You've learned how the size of a robot affects motor and controller decisions. You've seen how to make parts easy to remove using hook and loop tape and considered other options for this.

By hooking up independent power, and connecting the motors, the robot has the hardware it will need to drive itself around without being tethered to a wall. What it doesn't yet have is any code to move with. In the next chapter, we will start writing the code to get this robot moving!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercises

- In *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*, you created block diagrams for a different robot. Consider what chassis, power, and parts it needs. Use online vendors to find suitable parts.
- Consider whether there is a suitable combination of Raspberry Pi hats or bonnets that fit your design. Use resources such as `https://pinout.xyz` to check their pin usage is compatible.
- What power systems might be suitable for the Raspberry Pi and the output devices in your new design?
- Are there components that may need to be removed easily? How could you approach that? Can you come up with alternatives to hook and loop tape? Keep it simple.
- Make a test-fitting sketch for your new robot parts, checking that they fit using part dimensions.

Further reading

Please refer to the following for more information:

- For further reading on chassis designs, consider *Raspberry Pi Robotic Blueprints*, *Dr. Richard Grimmett*, *Packt Publishing*. This includes modifying an RC car into a robot.
- For more robot chassis types, the community sharing website `https://www.instructables.com` has many buildable examples. Some of these are very interesting and more advanced than our robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7
Drive and Turn – Moving Motors with Python

In this chapter, we will take the robot we started building in the last chapter, connect the motors to the Raspberry Pi, and build the Python code to make them move. We will cover programming techniques to create a layer between the physical robot and its behavior code, to reduce the impact of hardware changes. Our code and build will get the robot moving! We finish by programming the robot to drive a small set path. The robot code layer will serve as a foundation for all our robot behaviors, and the set path will demonstrate how to use it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We cover the following topics in this chapter:

- Writing code to test your motors
- Steering a robot
- Making a `Robot` object—code for our experiments to talk to the robot
- Writing a script to follow a predetermined path

Technical requirements

To complete the experiments in this chapter, you will require the following:

- A computer with access to the internet
- The chassis built in the *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*
- The motor controller bought in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*
- A 2-meter by 2-meter flat space for the robot to drive on

> **Important note**
> Be prepared to stop your robot from driving over the edges if you use a table! It's best to use the floor.

Check out the following video to see the code in action: `https://bit.ly/39sHxWL`

Writing code to test your motors

Before we get stuck in and do fancy things with the motors, we need to get them set up and test them. This way, we can make sure they work and iron out any problems.

We need to download the library to work with the motor board we have chosen. Many robot parts, apart from the simplest ones, have an interface library to control the motors and other devices on the board. It's time to log in to your Pi using PuTTY again.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Preparing libraries

We download this code from a project on GitHub using Git on the Raspberry Pi. So, we need to install Git on the Pi; we also need I2C (`i2c-tools` and `python3-smbus`) and `pip` to install things into Python. Type the following command:

```
pi@myrobot:~ $ sudo apt-get install -y git python3-pip python3-
smbus i2c-tools
```

To get the library for the motor board, `Raspi_MotorHAT`, we use Git and download it from GitHub, installing it for use in any of your scripts with the following command:

```
pi@myrobot:~ $ pip3 install git+https://github.com/orionrobots/
Raspi_MotorHAT
Collecting git+https://github.com/orionrobots/Raspi_MotorHAT
  Cloning https://github.com/orionrobots/Raspi_MotorHAT to /
tmp/pip-c3sFoy-build
Installing collected packages: Raspi-MotorHAT
  Running setup.py install for Raspi-MotorHAT ... done
Successfully installed Raspi-MotorHAT-0.0.2
```

We now have the libraries prepared for starting the robot. Documentation for the `Raspi_MotorHAT` library is sparse but is at `https://github.com/orionrobots/Raspi_MotorHAT`, along with examples of using it.

Test – finding the Motor HAT

The Raspberry Pi uses I2C to connect to this Motor HAT. **I2C buses** let you send and receive data, and are flexible in that we can connect many devices to the same bus. To enable I2C, use `raspi-config` again. We also enable the **Serial Peripheral Interface** (**SPI**) while we are here. We may need this to connect other boards and sensors. Type the following command:

```
$ sudo raspi-config
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, we use interfacing settings on this. *Figure 7.1* shows how, as follows:

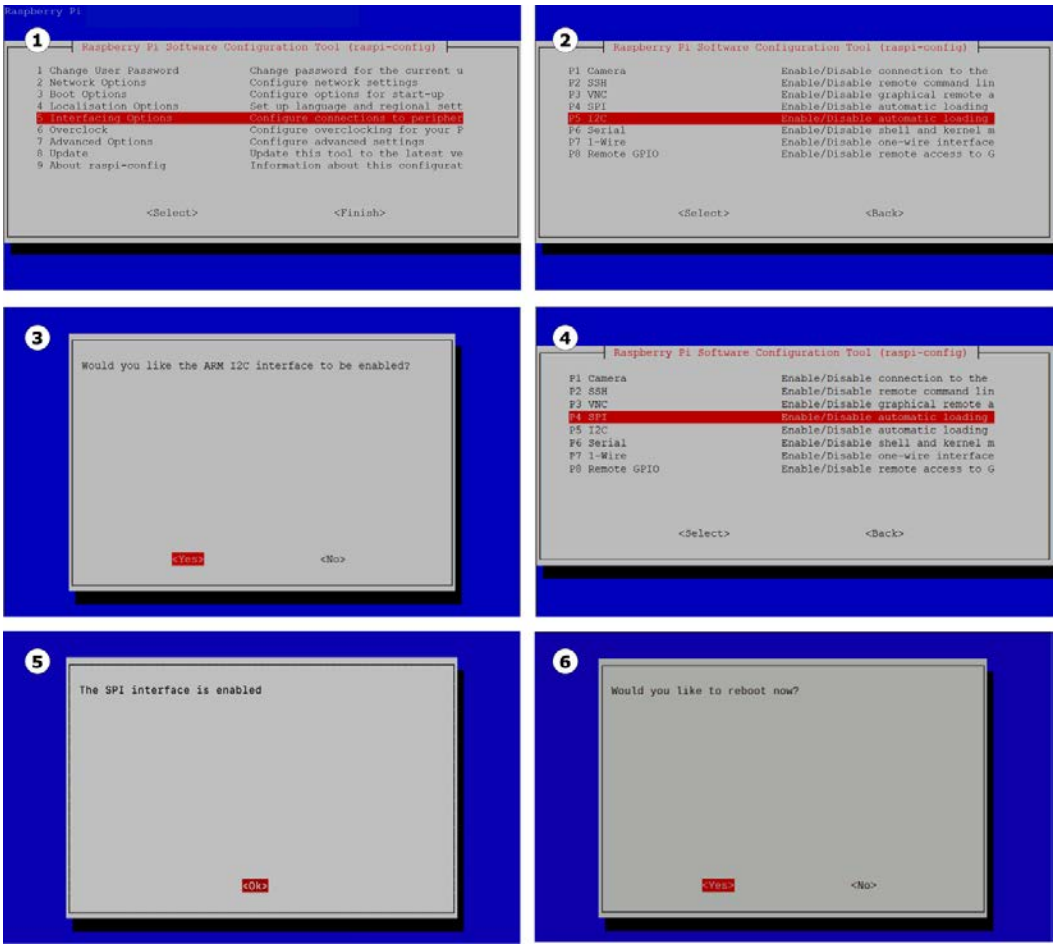


Figure 7.1 – Using raspi-config to enable SPI and I2C

Refer to the screenshots in *Figure 7.1* and perform the following steps:

1. First, select **Interfacing Options**.
2. Next, select **I2C**.
3. The Pi asks if you want this interface to be enabled. Select **<Yes>**.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. You are then taken back to the initial screen and need to navigate again to the **Interfacing Options** screen. From there, select **SPI** and **<Yes>** again.
5. A confirmation screen tells you now that SPI is enabled. Select **<Ok>**.
6. Finally, press *Esc* twice to finish `raspi-config`. It asks if you want to reboot. Select **<Yes>**, and then wait for the Pi to reboot and reconnect to the Raspberry Pi. If it doesn't ask, please use `sudo reboot` to reboot it.

With I2C, we need a way to choose which device we are talking with. Just as with houses along a road, an address allows us to say which one we specifically want.

We should check that the Raspberry Pi can see the Motor HAT with `sudo i2cdetect -y 1` by running the following code:

```
pi@myrobot:~ $ sudo i2cdetect -y 1
     0 1 2 3 4 5 6 7 8 9 a b c d e f
00: -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- 6f
70: 70 -- -- -- -- -- -- --
```

This scans the I2C bus `1` for devices attached to our Raspberry Pi. It shows numbers at the addresses if something is found. The device found at addresses `6f` and `70` is our motor controller. If you cannot see this, power down the Raspberry Pi and carefully check that the Motor HAT has been plugged in, then try again.

The addresses are hexadecimal, where each digit counts to 16, using the digits `0-9`, then letters `A-F` instead of counting only 10. When used in code, these get a `0x` prefix. This is a *zero* and then a lowercase x.

We have enabled the I2C (and SPI) bus, and we then used the `i2cdetect` tool to find our motor device. This confirms first that it is connected and responding, and secondly that we have the right address—`0x6f`—for it. We can now start to send commands to it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Test – demonstrating that the motors move

We need a test file to demonstrate that the motors work. Carry out the following steps:

1. Create the following file, called `test_motors.py`:

```
from Raspi_MotorHAT import Raspi_MotorHAT

import time
import atexit

mh = Raspi_MotorHAT(addr=0x6f)
lm = mh.getMotor(1)
rm = mh.getMotor(2)

def turn_off_motors():
  lm.run(Raspi_MotorHAT.RELEASE)
  rm.run(Raspi_MotorHAT.RELEASE)

atexit.register(turn_off_motors)

lm.setSpeed(150)
rm.setSpeed(150)

lm.run(Raspi_MotorHAT.FORWARD)
rm.run(Raspi_MotorHAT.FORWARD)
time.sleep(1)
```

2. Upload this file to your Raspberry Pi using the methods found in *Chapter 5*, *Backing Up the Code with Git and SD Card Copies*.

> **Important note**
>
> Move your robot from your desk and down to the floor for this next step, as when it moves, it might not go in the direction you expect!

3. To run this code, through PuTTY on the Pi, type the following:

```
pi@myrobot:~ $ python3 test_motors.py
```

Your robot should now drive roughly forward. It may move slightly to the side, but it should not be turning or going backward, and both motors should be moving.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

If you see any problems, try this troubleshooting chart and go back:

Symptom	Likely cause and solution
You see `ImportError: No module named smbus`.	You may not have installed the required packages. Ensure you have followed the preceding `apt-get` install steps and resolved any errors with them.
You see errors from Python code.	Please go back and carefully check that you have typed and uploaded the preceding code to the Raspberry Pi. Ensure you are running with `python3`. Ensure that the `pip install` steps worked without errors.
One or both sides are going backward.	The motor wires are the wrong way. On the motor terminals only, swap the black and red wire. Refer to *Figure 7.4* and *Figure 7.5* for help.
The light is on the controller, but one or both motors are not moving.	Please ensure that you've screwed both motors' wires firmly into the terminals. Please see the steps in the Wiring in section in *Chapter 6, Building Robot Basics – Wheels, Power, and Wiring*.
No light on the motor controller and no movement.	Please make sure you have attached both the battery wires into the correct terminals, as shown in the Independent power section in *Chapter 6, Building Robot Basics – Wheels, Power, and Wiring*. Please ensure that your batteries are well charged.
Turning or veering.	You can expect some veer. If it is severe, ensure that both motors' connections make firm contact and that neither the wheels nor the encoder disks are binding/caught on the chassis.

By this point, you should have a robot that will drive forward, have seen it move, and dealt with the preceding troubleshooting issues.

Understanding how the code works

Now, our motors are moving and the robot drives using the `test_motors.py` code. But how does our motor test code really work? In this section, let's take a closer look and understand this.

The first few lines of code here are imports:

```
from Raspi_MotorHAT import Raspi_MotorHAT

import time
import atexit
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Imports are how Python code *pulls in* other libraries of code to use them. The `Raspi_MotorHAT` library is the one we installed for interacting with our motors. The `time` library allows us to work with time; in this case, we use it for a delay between starting and stopping motors. The `atexit` library allows us to run code when this file exits.

In the following lines, we connect the library to the Motor HAT and the two motors we have connected:

```
mh = Raspi_MotorHAT(addr=0x6f)
lm = mh.getMotor(1)
rm = mh.getMotor(2)
```

The first line here makes a `Raspi_MotorHAT` object with the I2C address `0x6f` passed in as `addr`, which we saw in the scan. We call the returned object `mh` as an abbreviation for the connected `Raspi_MotorHAT`.

We then create shortcuts to access the motors: `lm` for the left motor and `rm` for the right motor. We get these motor controls from the `mh` object, using the motor number shown on the board. Motor 1 is left, and motor 2 is right.

We now define a function, `turn_off_motors`, which runs `Raspi_MotorHAT.RELEASE` on each motor on this board—an instruction to make the motors stop, as illustrated in the following code snippet:

```
def turn_off_motors():
  lm.run(Raspi_MotorHAT.RELEASE)
  rm.run(Raspi_MotorHAT.RELEASE)
atexit.register(turn_off_motors)
```

We pass that into `atexit.register(turn_off_motors)`, a command that runs when this file finishes—when Python exits. `atexit` runs even when there are errors. Without this, the code could break in some interesting way, and the robot keeps driving. Robots without this kind of safeguard have a habit of driving off tables and into walls. If they carry on trying to drive when their motors are stuck, it can damage the motors, motor controllers, and batteries, so it's better to stop.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The speed of the motors for this controller/library ranges from `0` to `255`. Our code sets the speed of each motor to just above half speed and then runs the `Raspi_MotorHAT.FORWARD` mode, which makes each motor drive forward, as illustrated in the following snippet:

```
lm.setSpeed(150)
rm.setSpeed(150)

lm.run(Raspi_MotorHAT.FORWARD)
rm.run(Raspi_MotorHAT.FORWARD)
```

Finally, we ask the code to wait for 1 second, as follows:

```
time.sleep(1)
```

The sleep allows the motors to run in their forward-drive mode for 1 second. The program then exits. Since we told it to stop motors when the code exits, the motors stop.

We've now written and understood the code to test the motors. You've also seen it running. This confirms that you have a viable robot, and you have also started using Python imports. You've learned the `atexit` trick to turn things off and about using a timer so that the robot has some time to run before exiting. Now, we look at how we can steer the robot.

Steering a robot

Now, we've made a robot drive forward. But how do we steer it? How does it turn left or right? In order to understand this, we need to first learn about a few significant forms of steering that exist. Let's take a look at some, settle on the one our robot uses, and write some test code to demonstrate it.

Types of steering

The most common techniques for steering a wheeled vehicle (including a robot) fall into two major categories—steerable wheels and fixed wheels, as discussed in the following subsections. Each of them comes with a couple of slightly unusual variants.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Steerable wheels

In movable wheel designs, one or more wheels in a robot face in a different direction from the others. When the robot drives, the differently positioned wheel makes the robot turn. There are two common styles of movable wheel steering on a robot, as shown here in *Figure 7.2*:

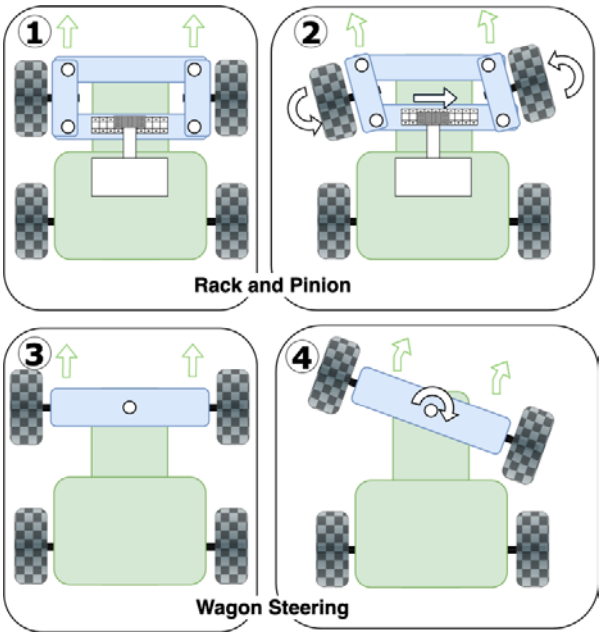


Figure 7.2 – Steerable wheel types

The green arrows show the direction of movement. The white arrows show changes to the shape of the robot and the angle of the wheels. Going through *Figure 7.2*, we can note the following:

1. Cars typically use **rack and pinion steering**. When straight, the car goes forward.
2. When the lower bar is moved, shown by the white arrows, the car turns.
3. The other common type is **wagon-style steering**, used in homemade racing karts. When straight, it goes forward.
4. By turning the front bar, you can steer the vehicle.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

There are also other variants besides the ones we discussed previously. They include the following:

- Robots with the ability to independently reorient each wheel and drive sideways
- Ackerman steering, where the amount each wheel rotates is different
- Rear steering, where a front set of wheels and a rear set of wheels steer—used in long vehicles

A good example of wagon-style steering is the Unotron robot, shown here in *Figure 7.3*. This was built by my son from the Unotron chassis by 4tronix, with an Arduino Nano controller:

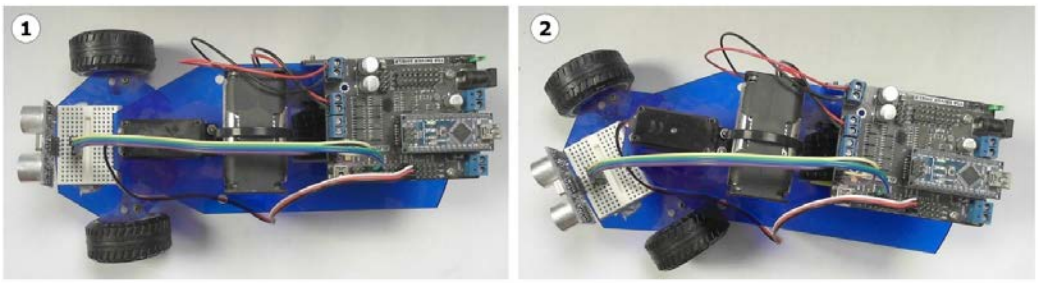


Figure 7.3 – Wagon-style steering Unotron robot

In the Unotron design, there is a single motor-driven wheel at the back (under the motor controller). A servo motor turns the whole front plate, steering the two front wheels.

The disadvantages of this type of steering are related to space, weight, and complexity. A chassis set up for movable wheel steering requires more moving parts and space to house them. Unotron is as simple as it gets. There is more complexity in other designs, which can lead to required maintenance.

The distance needed to make a turn (known as the turning circle) or for robots with steerable wheel systems is longer, as these must drive forward/backward to steer.

You require one large motor for the fixed axle, as you cannot distribute power across two motors, or you need complex mechanisms to balance the input. If the mechanism does not center after steering, then the robot veers.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Fixed wheels

Fixed-wheel steering is used frequently in robots, whereby the wheels' axes are fixed in relation to the chassis. The relative speed of each wheel or set of wheels sets the direction of the robot. That is, the wheels do not turn from side to side; however, by one side going faster than the other, the robot can make turns. A typical use of this is known as skid steering, which is illustrated in the following screenshot:

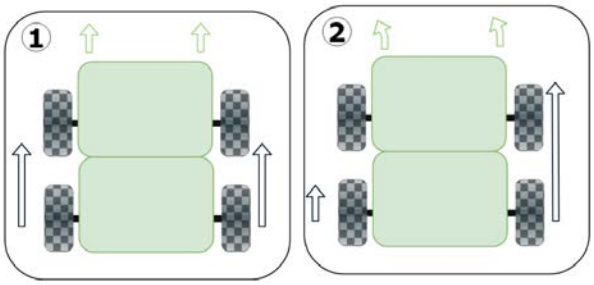


Figure 7.4 – Fixed-wheel steering or skid steering

Figure 7.4 shows this in action. The white arrows show the relative speed of the motors. The green arrows show the direction of the robot.

In the preceding figure, we can see the following:

1. The motors are going at the same speed, so the robot is driving straight forward.
2. The motors on the right are going fast; the motors on the left are going slow. This robot is driving forward and left.

This has several advantages. If you intend to use tank tracks, you need this type of drive system. It is mechanically simple in that a drive motor per wheel is all that is needed to make turns. Skid steering allows a robot to turn on the spot, doing a full 360 degrees in a turning circle the width of the widest/longest part of the robot.

There are some disadvantages to using this. When turning, a skid-steer system may drag wheels sideways, causing friction. Also, any minor differences in the motors, their gearing, or the controller output can result in a veer.

Other steering systems

The controller we are using on our robot allows us to control four motor channels. A builder can use four motors for special wheel types, known as Mecanum wheels. These wheels allow skid-steering style motions along with crabbing motions so that a robot can drive left or right without turning. Technically, this is still fixed-wheel steering. *Figure 7.5* here shows a base with Mecanum wheels:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 7.5 – Mecanum wheels on the Uranus Pod by Gwpcmu [CC BY 3.0 (https://creativecommons.org/licenses/by/3.0)]

These are amazingly flexible but mechanically complex, high maintenance, heavy, and a bit pricier than normal wheels. They are fun, however.

Steering the robot we are building

Based on the three-wheel chassis we have chosen, with one castor wheel and then a driven wheel on each side, independently controlled, we are using skid steering. By varying the speed and direction of these wheels, we steer our robot. We can also spin 360 degrees with it. The castor wheel negates the problem mentioned with the drag seen on four- and six-wheel skid-steer robots.

We can make the robot spin on the spot with one change to the previous code. Making one motor go back while the other goes forward spins the robot. Let's see how to do this, as follows:

1. Find the following lines in `test_motors.py`:

```
lm.run(Raspi_MotorHAT.FORWARD)
rm.run(Raspi_MotorHAT.FORWARD)
```

2. Modify this as follows so that one motor goes `BACKWARD`:

```
lm.run(Raspi_MotorHAT.FORWARD)
rm.run(Raspi_MotorHAT.BACKWARD)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Run this on the Pi with `python3 turn_motors.py`, and your robot now spins to the right. Swap them so left (`lm`) is `BACKWARD`, and right (`rm`) is `FORWARD`, and it spins the other way.
4. What about less aggressive turns? In the previous code, before the direction lines, we also set the speed of each motor, as follows:

```
lm.setSpeed(150)
rm.setSpeed(150)

lm.run(Raspi_MotorHAT.FORWARD)
rm.run(Raspi_MotorHAT.FORWARD)
```

   We can make a gentler turn by setting both `lm` and `rm` modes to `FORWARD`, and then making one of the speeds smaller than the other, like this:

```
lm.setSpeed(100)
rm.setSpeed(150)

lm.run(Raspi_MotorHAT.FORWARD)
rm.run(Raspi_MotorHAT.FORWARD)
```

   This code makes the robot drive forward and turn gently to the left.

You've now seen a few ways to steer robots. Based on the design our robot has, you've then put one of them into practice, making a robot spin on the spot, and also drive forward and turn too. In the next section, we'll turn this into a layer for different behaviors to use the robot.

Making a Robot object – code for our experiments to talk to the robot

Now we have seen how to move and turn our robot, we come on to a layer of software to group up some of the hardware functions and isolate them from **behaviors**. By behaviors, I mean code to make a robot behave a certain way, for example following a line or avoiding walls. Why would we want that isolation?

When we chose our motor controller, we made many trade-offs to find what works for our project. Motor controllers can change when the considerations change or when we simply want to build our next robot. Although controlling the speed and direction of two motors is the same kind of operation, each controller does it slightly differently. Creating a layer in front of a controller lets us use the same commands for it, even if it changes. This layer acts as a façade or interface to robot functionality.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Each controller has quirks. With this one, we set a run mode and speed. Many controllers use 0 to mean stop, but this one uses a `RELEASE` mode, which is slightly different from speed 0, which holds the motors. Controllers often use negative numbers to mean go backward; this one has a `BACKWARD` mode. The speed values on this controller go from 0-255. Some go from -128-128, or 0-10. What we can do is to create an object with an interface to hide quirks specific to this controller.

Why make this object?

You design an interface to give you a way to interact with some other code. It can simplify, or make more consistent, different underlying systems to make them behave the same way, as with all the types of motor controller mentioned. It also provides a way to cleanly separate parts of code into layers. Different layers mean that you can change one part of some code without it making considerable changes in another, as illustrated in the following figure:

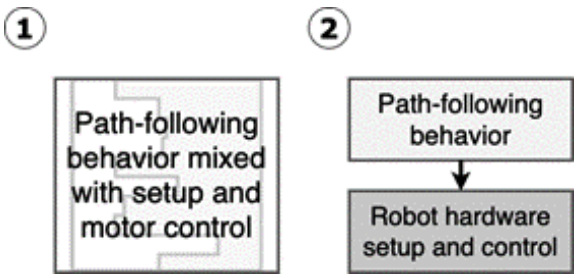


Figure 7.6 – Software layers

In *Figure 7.6*, panel 1 shows a block of code that has different systems mixed. It's hard to change; adding a new behavior or swapping the motor controller in this code would be quite tricky. It's good to avoid mixing responsibilities in this way.

The code represented by panel 2 shows two separate systems interacting. They have a relationship where the *path-following behavior* is in control of the *robot hardware setup and control* code.

Throughout the book we write many behaviors, and we can reuse the hardware control library, perhaps extending it occasionally. After all, who wants to keep writing the same code? When you extend and make new behaviors, you can use this layer again too.

The robot hardware setup/control block in the second panel of *Figure 7.6* is our `Robot` object. It is an interface to hide the quirks of the *Full Function Stepper HAT* board.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This standard interface means we could make an object that looks the same from the outside on other robots, and our behaviors still work. Some serious robot builders use interfaces to swap real controllers for simulations of robots, to test complex behaviors.

What do we put in the robot object?

An object is a building block to make interfaces in Python. Objects have methods—things we can call on it to perform tasks. Objects also have members, bits of data, or references to other objects.

The next section builds code in the `Robot` object to do the following:

- Set up the Motor HAT and store its motors as members: `left_motor` and `right_motor`.
- Deal with the `exit` state.
- Stop motors with a `stop_motors` method.
- Let us use percentages to mean speeds—values of 0 to 100. We map this to what the controller wants.
- The modes are particular to this controller. Our interface uses negative values to mean going backward.
- At a later stage, the `Robot` object can act as a gatekeeper to data buses that require code to hold exclusive locks on them and some of the hardware.
- Our interface (and therefore our object) does not contain behavior, other than the stopping-on-exit safeguard.

We put it in a file named `robot.py`, as follows:

```
from Raspi_MotorHAT import Raspi_MotorHAT

import atexit

class Robot:
    def __init__(self, motorhat_addr=0x6f):
        # Setup the motorhat with the passed in address
        self._mh = Raspi_MotorHAT(addr=motorhat_addr)

        # get local variable for each motor
        self.left_motor = self._mh.getMotor(1)
        self.right_motor  = self._mh.getMotor(2)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        # ensure the motors get stopped when the code exits
        atexit.register(self.stop_motors)

    def stop_motors(self):
        self.left_motor.run(Raspi_MotorHAT.RELEASE)
        self.right_motor.run(Raspi_MotorHAT.RELEASE)
```

This `class` has a `__init__` method, a special one that sets this layer up. The `__init__` method stores the output of the `getMotor` methods from the `Raspi_MotorHat` library in the `left_motor` and `right_motor` members. This method also registers a stop system. I have added some comments to state what the fragments of code do.

So far, our `Robot` object has set up our Motor HAT and has a way to stop the motors. The code is the same setup code we have seen before but is structured slightly differently.

We can test this in another file named `behavior_line.py`, as illustrated in the following code snippet:

```
import robot
from Raspi_MotorHAT import Raspi_MotorHAT
from time import sleep

r = robot.Robot()
r.left_motor.setSpeed(150)
r.right_motor.setSpeed(150)
r.left_motor.run(Raspi_MotorHAT.FORWARD)
r.right_motor.run(Raspi_MotorHAT.FORWARD)
sleep(1)
```

This starts by pulling in the `robot.py` file we just created with an import. It goes forward for 1 second and stops. Run with `python3 behavior_line.py`.

We still have to set speeds specific to this board (not out of 100). Let's fix that in `robot.py` (new code is in bold), as follows:

```
from Raspi_MotorHAT import Raspi_MotorHAT

import atexit

class Robot(object):
    def __init__(self, motorhat_addr=0x6f):
        self._mh = Raspi_MotorHAT(addr=motorhat_addr)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        self.left_motor = self._mh.getMotor(1)
        self.right_motor  = self._mh.getMotor(2)
        atexit.register(self.stop_motors)

    def convert_speed(self, speed):
        return (speed * 255) // 100

    def stop_motors(self):
        self.left_motor.run(Raspi_MotorHAT.RELEASE)
        self.right_motor.run(Raspi_MotorHAT.RELEASE)
```

We can now use `convert_speed`, to use speeds from 0 to 100. This returns speeds from 0 to 255 for this Motor HAT. For other motor boards, this returns something else.

We multiply the speed by 255 and divide that by 100. This formula is a way of turning a percentage into a fraction of 255. We multiply first because we are doing integer (whole number) math, and dividing 80/100 with whole numbers gives 0, but dividing (80*255) by 100 returns 204.

This code is still unwieldy, though—to use it, we need the following in `behavior_line.py`:

```
import robot
from Raspi_MotorHAT import Raspi_MotorHAT
from time import sleep

r = robot.Robot()
r.left_motor.setSpeed(r.convert_speed(80))
r.right_motor.setSpeed(r.convert_speed(80))
r.left_motor.run(Raspi_MotorHAT.FORWARD)
r.right_motor.run(Raspi_MotorHAT.FORWARD)
sleep(1)
```

This still uses the `run` and `setSpeed` methods of the `Raspi_MotorHAT` library, which are specific to this control board. Other boards don't work the same way. We can also collect up the cumbersome conversion a little.

We start by modifying the `convert_speed` method. It can be convenient for robots to use negative values to mean the motor goes backward. We still need to scale the speed, but we need to determine the run mode too.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We need to do the following two things:

- Determine if the speed is above, below, or equal to zero, and set the mode for the `run` function.
- Remove the sign from the speed for `setSpeed`, so it's always a positive value.

The default mode that we get at speed zero is `RELEASE` or stop. If the speed is above 0, we return the `FORWARD` mode, and if it's below 0, we return `BACKWARD`.

We can use a simple `if` statement to get the correct `mode`. Let's replace the `convert_speed` method in the class to return the mode and positive value. I've used comments to show the two sections to this function. Modify this in `robot.py`, as follows:

```
def convert_speed(self, speed):
        # Choose the running mode
        mode = Raspi_MotorHAT.RELEASE
        if speed > 0:
            mode = Raspi_MotorHAT.FORWARD
        elif speed < 0:
            mode = Raspi_MotorHAT.BACKWARD

        # Scale the speed
        output_speed = (abs(speed) * 255) // 100
        return mode, int(output_speed)
```

We've added one more operation to our speed calculation: `abs(speed)`. This operation returns the absolute value, which removes the sign from a number. For example, -80 and 80 both come out as 80, which means there is always a positive output from the method.

Next, we add some methods to directly set the speed and direction of the left and right motors in the robot. These call `convert_speed` and use the mode and output speed from it to make calls to the `Raspi_MotorHAT` functions.

We then need to change our motor movement methods to use this speed conversion, as follows:

```
    def set_left(self, speed):
        mode, output_speed = self.convert_speed(speed)
        self.left_motor.setSpeed(output_speed)
        self.left_motor.run(mode)

    def set_right(self, speed):
        mode, output_speed = self.convert_speed(speed)
        self.right_motor.setSpeed(output_speed)
        self.right_motor.run(mode)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

So, for each motor, we get the mode and output speed from the passed-in speed, then call `setSpeed` and `run`.

The whole of `robot.py` should now look like the following:

```
from Raspi_MotorHAT import Raspi_MotorHAT

import atexit

class Robot:
    def __init__(self, motorhat_addr=0x6f):
        # Setup the motorhat with the passed in address
        self._mh = Raspi_MotorHAT(addr=motorhat_addr)

        # get local variable for each motor
        self.left_motor = self._mh.getMotor(1)
        self.right_motor = self._mh.getMotor(2)

        # ensure the motors get stopped when the code exits
        atexit.register(self.stop_motors)

    def convert_speed(self, speed):
        # Choose the running mode
        mode = Raspi_MotorHAT.RELEASE
        if speed > 0:
            mode = Raspi_MotorHAT.FORWARD
        elif speed < 0:
            mode = Raspi_MotorHAT.BACKWARD

        # Scale the speed
        output_speed = (abs(speed) * 255) // 100
        return mode, int(output_speed)

    def set_left(self, speed):
        mode, output_speed = self.convert_speed(speed)
        self.left_motor.setSpeed(output_speed)
        self.left_motor.run(mode)

    def set_right(self, speed):
        mode, output_speed = self.convert_speed(speed)
        self.right_motor.setSpeed(output_speed)
        self.right_motor.run(mode)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    def stop_motors(self):
        self.left_motor.run(Raspi_MotorHAT.RELEASE)
        self.right_motor.run(Raspi_MotorHAT.RELEASE)
```

Our simple behavior in `behavior_line.py` is now only a few lines, as can be seen in the following code snippet:

```
import robot
from time import sleep

r = robot.Robot()
r.set_left(80)
r.set_right(80)
sleep(1)
```

This simplification means we can build on this code to create more behaviors. I have a common interface, and versions of the `Robot` object for my other robots. An exciting outcome is I can run this `behavior_lines.py` code on ArmBot (the robot seen at the end of *Chapter 1*, *Introduction to Robotics*) or my other Raspberry Pi robots. They all go forward for 1 second at 80% of their motor speed.

Writing a script to follow a predetermined path

So, we now get to the first behavior that feels like a robot. Let's make a quick sketch of a path for us to get our robot to follow. For an example, see *Figure 7.7* here:

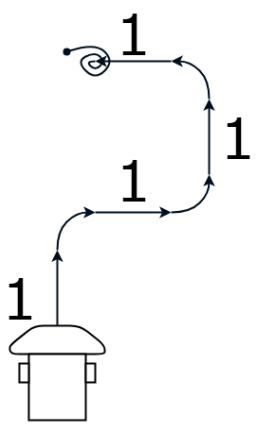


Figure 7.7 – Path for our robot

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In *Figure 7.7*, I've drawn a path. The straight lines are for driving forward; the **1**s mean 1 second. We don't yet have a way to consider distance traveled, only time. We may be able to guess at times relative to distances, but this isn't very precise or repeatable. The gentle curves are a turn where we slow one motor down more than the other.

The final spiral means a victory spin on the spot when the path is complete—we can do this by putting one motor in reverse while the other drives forward.

Let's write this code. First, we want the imports: `sleep` and `robot`. But before we do anything, let's make some helper functions for this behavior. I called my file `behavior_path.py`, and the code is shown in the following snippet:

```
import robot
from time import sleep

def straight(bot, seconds):
    bot.set_left(80)
    bot.set_right(80)
    sleep(seconds)

def turn_left(bot, seconds):
    bot.set_left(20)
    bot.set_right(80)
    sleep(seconds)

def turn_right(bot, seconds):
    bot.set_left(80)
    bot.set_right(20)
    sleep(seconds)

def spin_left(bot, seconds):
    bot.set_left(-80)
    bot.set_right(80)
    sleep(seconds)
```

The helpers use the same language we used to describe the behavior. We have `straight`, `turn_left`, `turn_right`, and `spin_left`. These are not in the `Robot` object because other behaviors may use more continuous behavior than this. I've called the `Robot` object `bot` now because one-letter variable names such as `r` become less easy to find, read, or reason about when there is more code.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

These helpers each set the motor speeds, and then sleep for a determined number of seconds. We can then create the `Robot` object and sequence them by adding the following code to `behavior_path.py`:

```
bot = robot.Robot()
straight(bot, 1)
turn_right(bot, 1)
straight(bot, 1)
turn_left(bot, 1)
straight(bot, 1)
turn_left(bot, 1)
straight(bot, 1)
spin_left(bot, 1)
```

Now, we can upload this to the Raspberry Pi, and run it via PuTTY with the following:

```
$ python3 behavior_path.py
```

Now, if your robot is anything like mine, you saw it drive and make turns, but the turns have overshot in some way, and the robot may be veering to one side. We can fix the overshoot here by reducing the amount of time in the turn steps, like this:

```
bot = robot.Robot()
straight(bot, 1)
turn_right(bot, 0.6)
straight(bot, 1)
turn_left(bot, 0.6)
straight(bot, 1)
turn_left(bot, 0.6)
straight(bot, 1)
spin_left(bot, 1)
```

You need to tweak these values to get close to 90-degree turns. This tweaking takes patience: change them and upload them. Tweaking values in code is a crude form of calibration to match the quirks of our robot. If you move between surfaces (for example, from a wooden floor to a carpet), then the timings will change.

You may be able to account for some of the veering by tuning one motor to be slower in the `straight` function (adjust for your own robot's veer), like this:

```
def straight(bot, seconds):
    bot.set_left(80)
    bot.set_right(70)
    sleep(seconds)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This code holds up for a while but may be hard to fine-tune. Why do we get this veer?

Motor speeds can vary, even those from the same manufacturer. Other causes of minor variations are wheel diameters, axle positioning, weight distribution, slippery or uneven surfaces, wiring resistance, and motor controller variations. This variation makes it unlikely that you'd get a perfectly straight line from a robot this way. Depending on which sensors we are using, this may or may not be a problem. To account for this problem, we introduce encoders/speed sensors in a later chapter and calibrate those sensors to get a more accurate version of a path behavior.

Without sensors, a robot is not able to determine where it is or if it has bumped into anything. If the robot ran into a wall, you'd probably have to go and move it to where it had room to move.

Summary

In this chapter, we've learned how to install the libraries for the motor board and demonstrate that our motors work. We then started building the first layer of code for our behaviors to use, while noting how we could make a layer like that for other robots. We saw our robot move in a path and tuned it, while finding out some of the shortcomings of using motors without any sensors.

You can now use this when starting any hardware project: get the motors/output devices tested first, then create a layer for a behavior to use them, such that if their hardware later changes, you only need to change the motor code.

In the following chapters, we start adding sensors and building behaviors using these sensors.

Exercises

Try these further ideas to enhance your learning from this chapter:

1. Sketch out another simple path and write code for the robot to follow it. For example, try to follow a figure-of-8 shape using your experience.
2. Which methods would you add to the `Robot` object if you had an additional output to control, perhaps a single **light-emitting diode** (**LED**)?
3. Consider how you would lay out a `Robot` object for a robot with kart-style steering. Which methods would it have? You don't need to write the code yet, but having an interface in mind is a good start. Hint—it probably has one motor speed for the drive and a motor position for the steering.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

Please refer to the following for more information:

- For more information on the style used for the `Robot` object, along with the use of similar interfaces and classes, I recommend *Learning Object-Oriented Programming*, *Gastón C. Hillar*, *Packt Publishing*. This book not only works through these concepts in Python but takes them more generally and shows how **object-oriented** (**OO**) concepts also apply to the C# and JavaScript languages.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8
Programming Distance Sensors with Python

In this chapter, we look at distance sensors and how to use them to avoid objects. Avoiding obstacles is a key feature in mobile robots, as bumping into stuff is generally not good. It is also a behavior that starts to make a robot appear smart, as if it is behaving intelligently.

In this chapter, we find out about the different types of sensors and choose a suitable type. We then build a layer in our robot object to access them and, in addition to this, we create a behavior to avoid walls and objects.

You will learn about the following topics in this chapter:

- Choosing between optical and ultrasonic sensors
- Attaching and reading an ultrasonic sensor
- Avoiding walls – writing a script to avoid obstacles

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

To complete the hands-on experiments in this chapter, you will require the following:

- The Raspberry Pi robot and the code from the previous chapters.
- Two HC-SR04P, RCWL-1601, or Adafruit 4007 ultrasonic sensors. They must have a 3.3 V output.
- A breadboard.
- 22 AWG single-core wire or a pre-cut breadboard jumper wire kit.
- A breadboard-friendly **single pole, double toggle (SPDT**) slide switch.
- Male-to-female jumpers, preferably of the joined-up jumper jerky type.
- Two brackets for the sensor.
- A crosshead screwdriver.
- Miniature spanners or small pliers.

The code for this chapter is available on GitHub at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter8`.

Check out the following video to see the Code in Action: `https://bit.ly/2KfCkZM`

Choosing between optical and ultrasonic sensors

Before we start to use distance sensors, let's find out what these sensors actually are, how they work, and some of the different types available.

The most common ways in which to sense distance are to use ultrasound or light. The principle of both of these mechanisms is to fire off a pulse and then sense its reflected return, using either its timing or angle to measure a distance, as can be seen in the following diagram:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

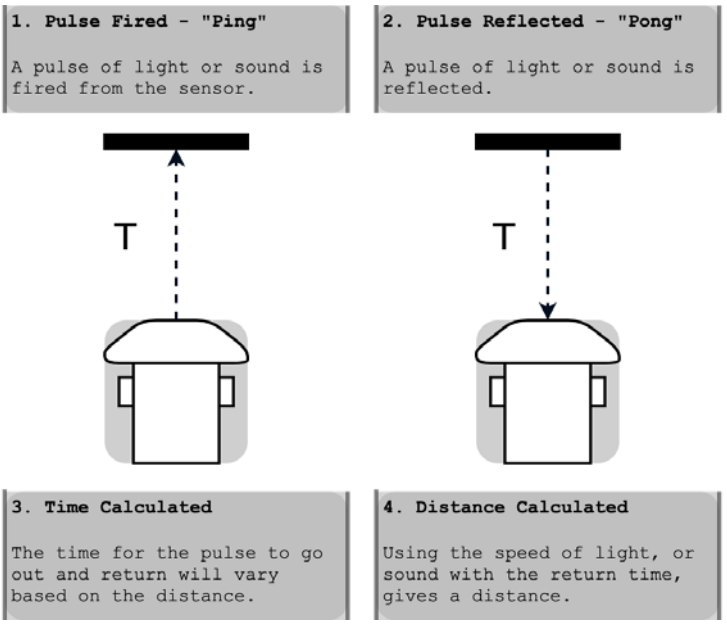


Figure 8.1 – Using pulse timing in a distance sensor

We focus on the sensors that measure the response time, otherwise known as the **time of flight**. *Figure 8.1* shows how these sensors use reflection time.

With this basic understanding of how sensors work, we'll now take a closer look at optical sensors and ultrasonic sensors.

Optical sensors

Light-based sensors, like the one in *Figure 8.2*, use infrared laser light that we cannot see. These devices can be tiny; however, they can suffer in strong sunlight and fluorescent light, making them misbehave. Some objects reflect light poorly or are transparent and are undetectable by these sensors:

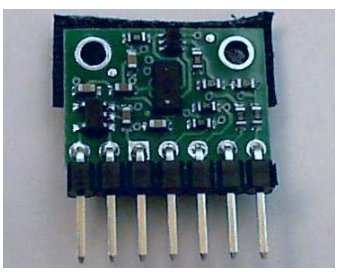


Figure 8.2 – A VL530LOx on a carrier board

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In competitions where infrared beams detect course times, the beams and these sensors can interfere with each other. However, unlike ultrasonic sensors, these are unlikely to cause false detections when placed on different sides of a robot. Optical distance sensors can have higher accuracy, but over a more limited range. They can be expensive, although there are cheaper fixed range types of light sensors out there.

Ultrasonic sensors

Many sound-based distance measuring devices use ultrasonic sound with frequencies beyond human hearing limits, although they can annoy some animals, including dogs. Mobile phone microphones and some cameras pick up their pulses as clicks. Ultrasonic devices tend to be larger than optical ones, but cheaper since sound travels slower than light and is easier to measure. Soft objects that do not reflect sound, such as fabrics, can be harder for these to detect.

Figure 8.3 shows the HC-SR04, a common and inexpensive sound-based distance sensor:

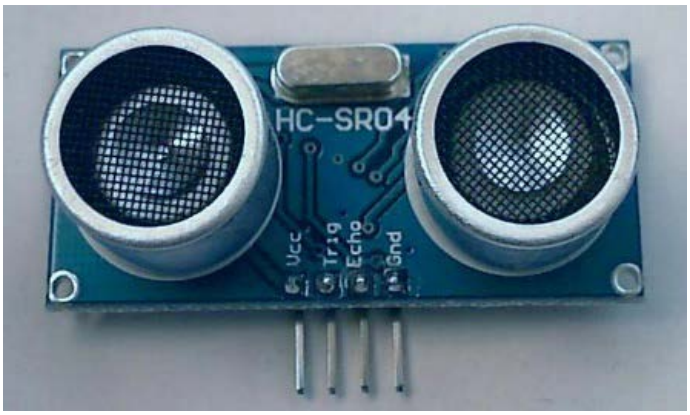


Figure 8.3 – The HC-SR04

They have a range of up to 4 meters from a minimum of about 2 cm.

There are a number of ultrasonic-based devices, including the common HC-SR04, but not all of them are suitable. We'll look at logic levels as this is an important factor in choosing which sensor to buy.

Logic levels and shifting

The I/O pins on the Raspberry Pi are only suitable for inputs of 3.3 V. Many devices in the market have a 5 V logic, either for their inputs when controlling them, or from their outputs. Let's dig into what I mean by logic levels, and why it is sensible to try and stick to the native voltage level when possible.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Voltage is a measure of how much pushing energy there is on an electrical flow. Different electronics are built to tolerate or to respond to different voltage levels. Putting too high a voltage through a device can damage it. On the other hand, putting too low a voltage can cause your sensors or outputs to simply not respond or behave strangely. We are dealing with logic devices that output a high or low voltage to represent a true/false value. These voltages must be above a threshold to be true, and below it to be false. We must be aware of these electrical properties, or we will destroy things and fail to get them to communicate.

The graph in *Figure 8.4* shows the effects that different levels have:

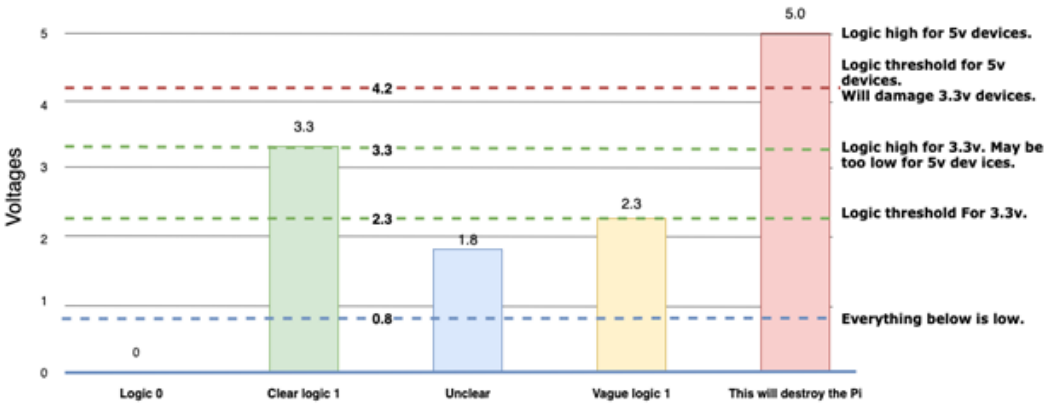


Figure 8.4 – Voltages and logic levels

In *Figure 8.4*, we show a graph. On the *y*-axis (left), it shows voltage labels from 0 to 5 V. The *y*-axis shows different operating conditions. There are 4 dashed lines running through the graph. The lowest dashed line is at 0.8 V; below this, an input will consider it as logic 0. The next line, at around 2.3 V, is where many 3.3 V devices consider things at logic 1. The line at 3.3 V shows the expected input and output level for logic 1 on a Raspberry Pi. Above this line, damage may occur to a Raspberry Pi. At around 4.2 V is what some 5 V devices expect for logic 1 (although some will allow as low as 2 V for this) – the Raspberry Pi needs help to talk to those.

Along the graph are 5 bars. The first labeled bar is at 0 – meaning a clear logic 0 to all devices. The next bar is a clear logic 1 for the Raspberry Pi at 3.3 V, but it is also below 4.2 V, so some 5 V devices won't recognize this. The bar labelled unclear is at 1.8 V – in this region, between the low and the high thresholds, the logic might not be clear, and this should be avoided. The bar labeled **Vague logic 1** is above the threshold, but only just, and could be misinterpreted or cause odd results on 3.3 V devices. The last bar is at 5 V, which 5 V devices output. This must not go to the Raspberry Pi without a level shifter or it will destroy that Raspberry Pi.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

There are bars in *Figure 8.4* at 1.7 V and 2.3 V. These voltages are very close to the logic threshold and can result in random data coming from the input. Avoid intermediate voltages between the required logic levels. 3 V is OK, but avoid 1.5 V as this is ambiguous.

> **Important note**
>
> Putting more than 3.3 V into a Raspberry Pi pin damages the Raspberry Pi. Do not use 5 V devices without logic level shifters.

If you use devices that are 5 V, you require extra electronics to interface them. The electronics come with further wiring and parts, thereby increasing the cost, complexity, or size of the robot's electronics:

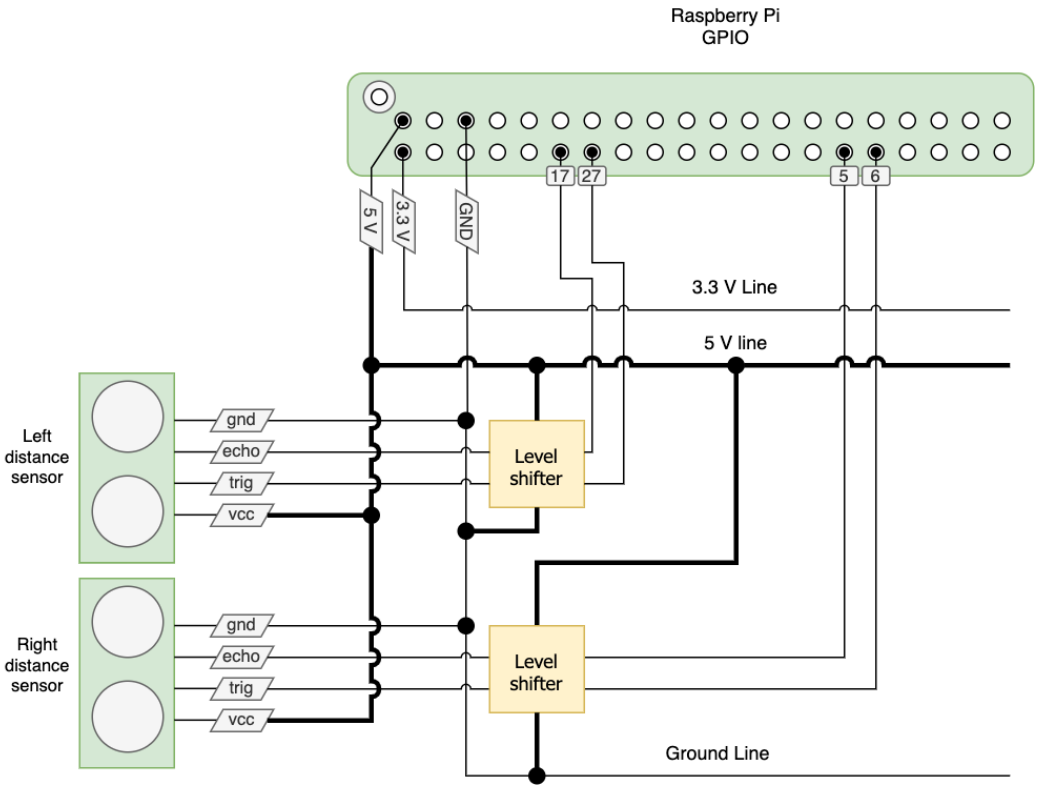


Figure 8.5 – Wiring the HC-SR04 sensors into the level shifters

Figure 8.5 shows a wiring diagram for a robot that uses HC-SR04 5v sensors that require logic level shifting. This circuit diagram shows the Raspberry Pi GPIO pins at the top. Coming from 3 pins to the left are the 5 V, 3.3 V (written as 3v3), and ground (GND) lines. Below the GPIO pins are the 3.3 V and 5 V lines.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Below the power lines (or rails) are two level shifters. Going into the right of the level shifters are connections from the Raspberry Pi GPIO pins 5, 6, 17, and 27. In this style of diagram, a black dot shows a connection, and lines that do not connect are shown with a bridge.

The bottom of the diagram has a ground line from the ground pin. This is shown as it's normal that additional electronics will require access to a ground line.

The left of the diagram has the two distance sensors, with connections to 5 V and GND. Each sensor has the **trig** and **echo** pins wired to the level shifters. It's not hard to see how adding more sensors that also require level shifters to this would further increase complexity.

Thankfully, other options are now available. Where it is possible to use a 3.3 V native device or a device that uses its supply voltage for logic high, it is worth choosing these devices. When buying electronics for a robot, consider carefully what voltage the robot's main controller uses (like the Raspberry Pi), and check that the electronics work with the controller's voltages.

The HC-SR04 has several replacement parts that have this ability. The HC-SR04P, the RCWL-1601, and Adafruit 4007 models output 3.3 V and can connect directly to the Raspberry Pi.

Why use two sensors?

Having two sensors allows a behavior to detect which side is closer. With this, the robot can detect where open spaces are and move toward them. *Figure 8.6* shows how this works:

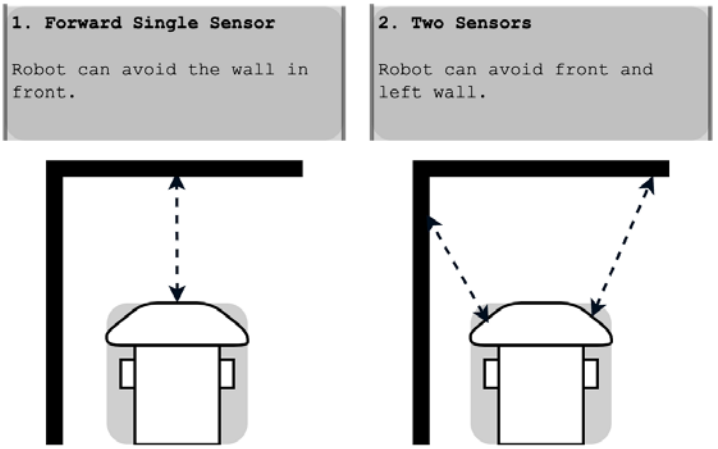


Figure 8.6 – Using two sensors

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In *Figure 8.6*, the second robot can make more interesting decisions because it has more data from the world with which to make those decisions.

Considering all of these options, I recommend you use a 3.3 V variant like the HC-SR04P/RCWL-1601 or Adafruit 4007 because they are cheap and because it is easy to add two or more of these sensors.

We've seen some distance sensor types and discussed the trade-offs and choices for this robot. You've learned about voltage levels, and why this is a crucial consideration for robot electronics. We've also looked at how many sensors we could use and where we could put them. Now let's look at how to add them.

Attaching and reading an ultrasonic sensor

First, we should wire in and secure these sensors to the robot. We then write some simple test code that we can use to base our behavior code on in the next section. After completing this section, the robot block diagram should look like *Figure 8.7*:

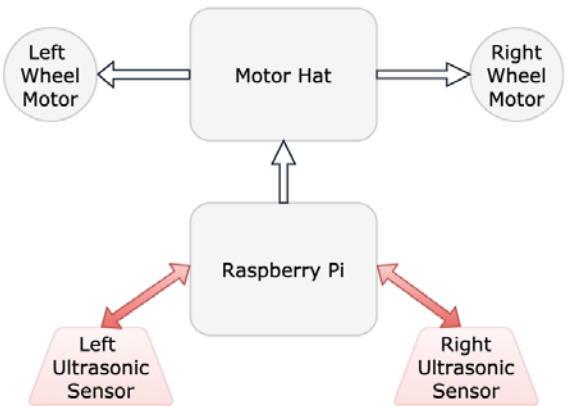


Figure 8.7 – Robot block diagram with ultrasonic sensors

This diagram builds on the block diagram in *Figure 6.33* from *Chapter 6, Building Robot Basics – Wheels, Power, and Wiring* by adding left and right ultrasonic sensors. Both have bi-directional arrows to the Raspberry Pi, since, being an active sensor, the Raspberry Pi triggers a sensor measurement and then reads back the result. Let's attach the sensors to the robot chassis.

Securing the sensors to the robot

In the *Technical requirements* section, I added an HC-SR04 bracket. Although it is possible to make a custom bracket with CAD and other part making skills, it is more sensible to use one of the stock designs. *Figure 8.8* shows the bracket I'm using:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 8.8 – Ultrasonic HC-SR04 sensor brackets with the screws and hardware

These are easy to attach to your robot, assuming that your chassis is similar enough to mine, in that it has mounting holes or a slot to attach this bracket:

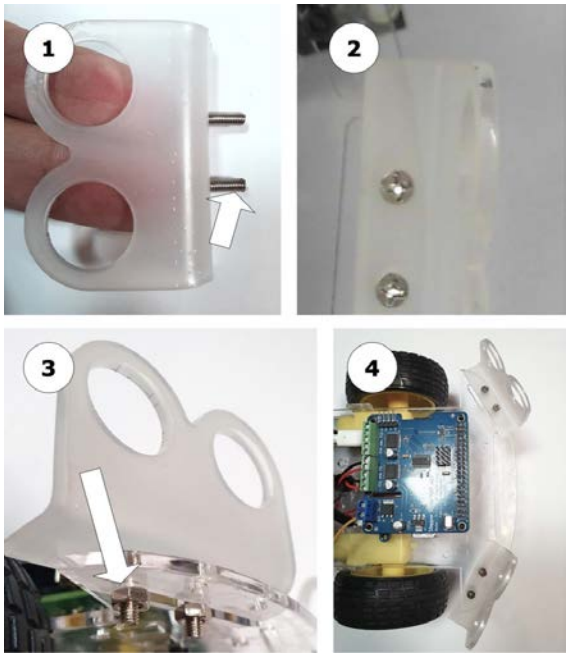


Figure 8.9 – Steps for mounting the sensor bracket

To mount the sensor bracket, use *Figure 8.9* as a guide for the following steps:

1. Push the two bolts into the holes on the bracket.
2. Push the bracket screws through the holes at the front of the robot.
3. Thread a nut from underneath the robot on each and tighten. Repeat this for the other side.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. The robot should look like this with the two brackets mounted.

   Figure 8.10 shows how to push the sensors into the brackets:

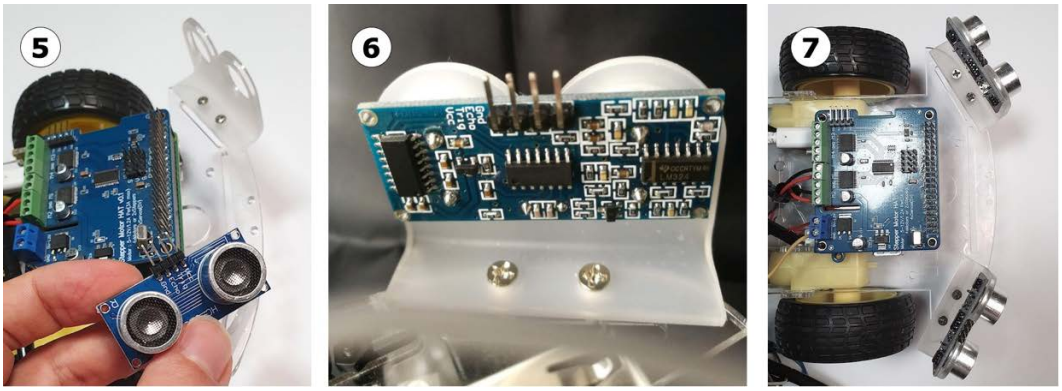


Figure 8.10 – Pushing the sensors into the brackets

5. Look at the sensor. The two transducer elements, the round cans with a gauze on top, will fit well in the holes in the brackets.
6. The distance sensors can simply be pushed into the brackets, since they have a friction fit. The electrical connector for the sensor should be facing upward.
7. After putting in both sensors, the robot should look like panel 7 of *Figure 8.10*.

You've now attached the sensors to the chassis. Before we wire them, we'll take a slight detour and add a helpful power switch.

Adding a power switch

Before we turn on the robot again, let's add a switch for the motor power. This switch is more convenient than screwing the ground wire from the battery into the terminal repeatedly. We'll see how to do this in three simple steps. Follow along:

1. Make sure you have the following equipment ready, as shown in *Figure 8.11*: a breadboard, some velcro, a mini breadboard-friendly SPDT switch, and one length of single-core 22 AWG wire:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

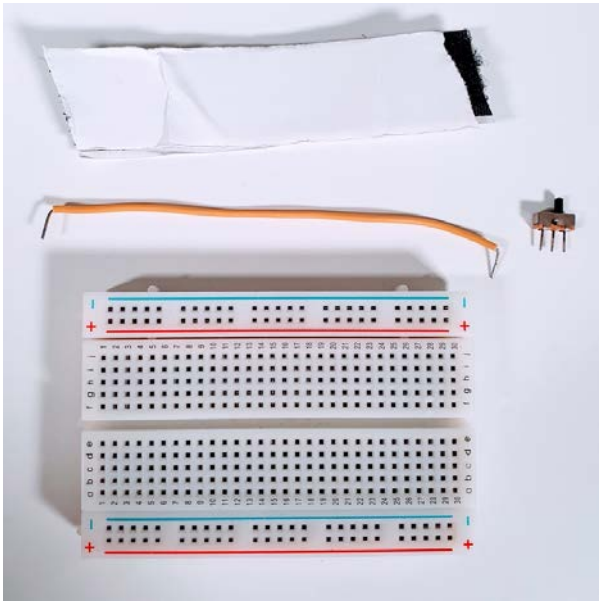


Figure 8.11 – Items needed to add a power switch

2. Now use two strips of Velcro to stick the breadboard on top of the robot's battery, as shown in *Figure 8.12*. The velcro holds firm but is easy to remove if you need to disassemble the robot:

Figure 8.12 – Adding velcro strips

With the breadboard in place, we can now add a switch.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Take a look at *Figure 8.13* for details on how the switch is connected:

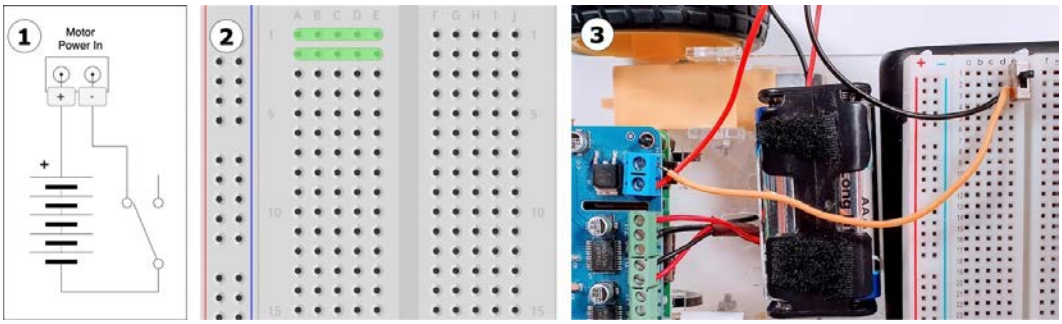


Figure 8.13 – Wiring the switch

Figure 8.13 shows a circuit diagram, a close-up of a breadboard, and a suggested way to wire the physical connections on the robot. Let's look at this in detail:

1. This is a circuit diagram showing the batteries, switch, and motor power input connectors. At the top is the *motor power in* terminal. From the positive (+) side of that terminal, a wire goes down the left to the batteries, shown as alternating thick and thin bars. From the batteries, the bottom terminal is their negative side. A wire goes from this around to the switch on the right of the diagram. The top of the switch is then connected via a wire to the negative (-) side of the *motor power in terminal*. This is the important diagram for making the connections.
2. Before we physically wire the switch, it's worth talking about the rows of the breadboard. This panel shows a close-up of a breadboard, with 2 of the rows highlighted in green lines. The green lines show that the rows are connected in groups of 5. The arrangement of a breadboard has two wired groups of 5 holes (tie-points) for each of the rows (numbered 1 to 30). It has a groove in the middle separating the groups.
3. The physical wiring uses the breadboard to make connections from wires to devices. It won't match the diagram precisely. The left shows the motor board, with a red wire from the batteries, their positive side, going into the positive (+ or VIN) terminal on the *motor power in terminal*. The batteries are in the middle. A black wire goes from the batteries into the breadboard in row 3, column *d*. In column *e*, a switch is plugged into the breadboard going across rows 1, 2, and 3. An orange precut 22 AWG wire goes from row 2 to the GND terminal, where it is screwed in. Sliding this switch turns on the power to the robot motors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've now given our robot a power switch for its motor batteries, so we can turn the motor power on without needing a screwdriver. Next, we will use the same breadboard to wire up the distance sensors.

Wiring the distance sensors

Each ultrasonic sensor has four connections:

- A trigger pin to ask for a reading
- An echo pin to sense the return
- A VCC/voltage pin that should be 3.3 V
- A GND or ground pin

Ensure that the whole robot is switched off before proceeding any further. The trigger and echo pins need to go to GPIO pins on the Raspberry Pi.

Figure 8.14 shows a close-up of the Raspberry Pi GPIO port to assist in making connections:

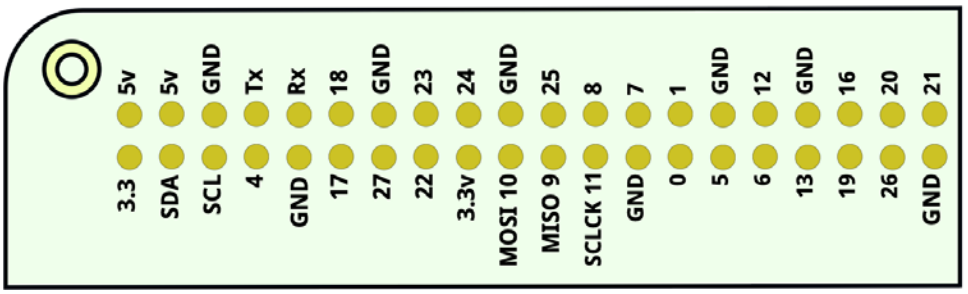


Figure 8.14 – Raspberry Pi connections

Figure 8.14 is a diagram view of the GPIO connector on the Raspberry Pi. This connector is the 40 pins set in two rows at the top of the Pi. Many robots and gadgets use them. The pin numbers/names are not printed on the Raspberry Pi, but this diagram should assist in finding them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We use a breadboard for this wiring. *Figure 8.15* shows the connections needed for these:

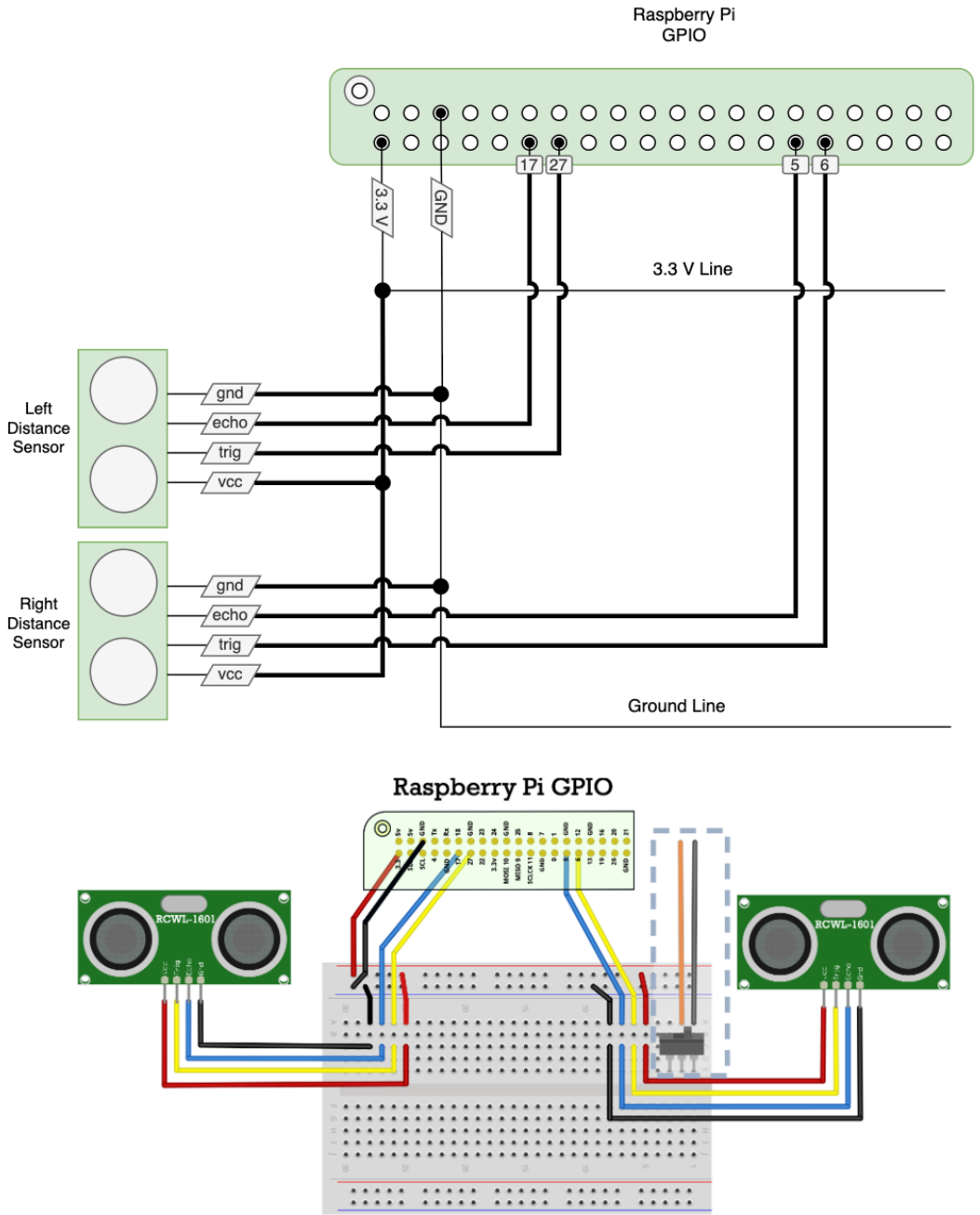


Figure 8.15 – Sensor wiring diagram

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Wires from the Raspberry Pi to the breadboard, and from the sensor to the breadboard, need male-to-female jumper wires. Wires on the breadboard (there are only 4 of these) use short pre-cut wires. *Figure 8.15* shows a circuit diagram above, and a breadboard wiring suggestion below.

To wire the sensors, use *Figure 8.15* as a guide, along with these steps:

1. Start with the power connections. A wire goes from the 3.3 V (often written as 3v3 on diagrams) pin on the Raspberry Pi to the top, red-marked rail on the breadboard. We can use this red *rail* for other connections needing 3.3 V.
2. A wire from one of the GND pins on the Pi goes to the black- or blue-marked rail on the breadboard. We can use this blue *rail* for connections requiring GND.
3. Pull off a strip of 4 from the male-to-female jumper wires for each side.
4. For the left-hand sensor, identify the four pins—VCC, trig, echo, and GND. For the connection from this to the breadboard, it's useful to keep the 4 wires together. Take 4 male-to-female connectors (in a joined strip if possible), from this sensor, and plug them into the board.
5. On the breadboard, use the precut wires to make a connection from ground to the blue rail, and from VCC to the red rail.
6. Now use some jumper wires to make the signal connections from the trig/echo pins to the Raspberry Pi GPIO pins.

> **Important note**
>
> Depending on where you've placed your breadboard, the distance sensor wires may not reach. If this is the case, join two male-to-female wires back to back, and use some electrical tape to bind them together.

For neatness, I like to wrap wires in spiral wrap; this is entirely optional but can reduce the clutter on the robot.

Please double-check your connections before you continue. You have now installed the distance sensors into your robot's hardware, but in order to test and use them, we need to prepare the software components.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Installing Python libraries to communicate with the sensor

To work with the GPIO sensor, and some other hardware, you need a Python library. Let's use the `GPIOZero` library, designed to help interface with hardware like this:

```
$ pip3 install RPi.GPIO gpiozero
```

With the library now installed, we can write our test code.

Reading an ultrasonic distance sensor

To write code for distance sensors, it helps to understand how they work. As suggested previously, this system works by bouncing sound pulses off of objects and measuring the pulse return times.

The code on the Raspberry Pi sends an electronic pulse to the **trigger** pin to ask for a reading. In response to this pulse, the device makes a sound pulse and times its return. The **echo** pin responds using a pulse too. The length of this pulse corresponds to the sound travel time.

The graph in *Figure 8.16* shows the timing of these:

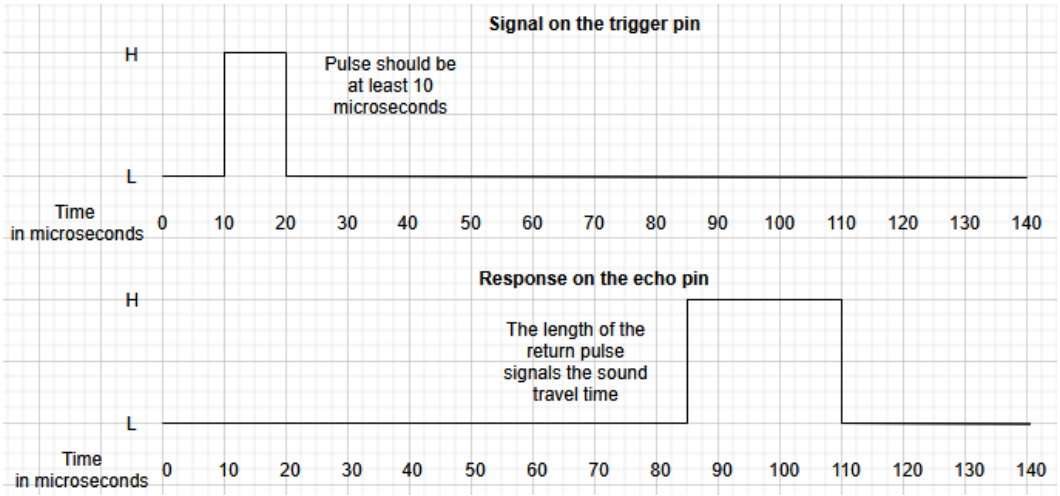


Figure 8.16 – Timing of a pulse and the response for an ultrasonic distance sensor

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `GPIOZero` library can time this pulse, and convert it into a distance, which we can use in our code.

The device might fail to get a return response in time if the sound didn't echo back soon enough. Perhaps the object was outside the sensor's range, or something dampened the sound.

As we did with our servo motor control class previously, we should use comments and descriptive names to help us explain this part of the code. I've called this file `test_distance_sensors.py`:

1. Begin by importing `time` and the `DistanceSensor` library:

```
import time
from gpiozero import DistanceSensor
```

2. Next, we set up the sensors. I've used `print` statements to show what is going on. In these lines, we create library objects for each distance sensor, registering the pins we have connected them on. Try to make sure these match your wiring:

```
print("Prepare GPIO Pins")
sensor_l = DistanceSensor(echo=17, trigger=27, queue_
len=2)
sensor_r = DistanceSensor(echo=5,  trigger=6,  queue_
len=2)
```

   You'll note the extra `queue_len` parameter. The `GPIOZero` library tries to collect 30 sensor readings before giving an output, which makes it smoother, but less responsive. And what we'll need for our robot is responsive, so we take it down to 2 readings. A tiny bit of smoothing, but totally responsive.

3. This test then runs in a loop until we cancel it:

```
while True:
```

4. We then print the distance from our sensors. `.distance` is a property, as we saw with the `.count` property on our LED system earlier in the book. The sensors are continuously updating it. We multiply it by 100 since `GPIOZero` distance is in terms of a meter:

```
    print("Left: {l}, Right: {r}".format(
        l=sensor_l.distance * 100,
        r=sensor_r.distance * 100))
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. A little sleep in the loop stops it flooding the output too much and prevents tight looping:

```
        time.sleep(0.1)
```

6. Now, you can turn on your Raspberry Pi and upload this code.
7. Put an object anywhere between 4 centimeters and 1 meter away from the sensor, as demonstrated in the following image:

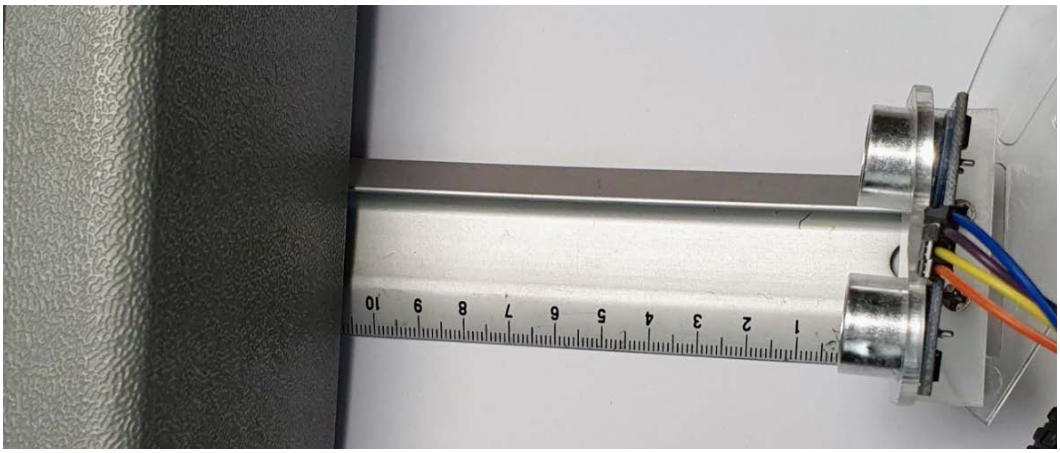


Figure 8.17 – Distance sensor with object

Figure 8.17 shows an item roughly 10.5 cm from a sensor. The object is a small toolbox. Importantly it is rigid and not fabric.

8. Start the code on the Pi with `python3 test_distance_sensors.py`. As you move around the object, your Pi should start outputting distances:

```
pi@myrobot:~ $ python3 test_distance_sensors.py
Prepare GPIO Pins
Left: 6.565688483970461, Right: 10.483658125707734
Left: 5.200715097982538, Right: 11.58136928065528
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Because it is in a loop, you need to press *Ctrl* + *C* to stop the program running.
10. You'll see here that there are many decimal places, which isn't too helpful here. First, the devices are unlikely to be that accurate, and second, our robot does not need sub-centimeter accuracy to make decisions. We can modify the print statement in the loop to be more helpful:

```
        print("Left: {l:.2f}, Right: {r:.2f}".format(
            l=sensor_l.distance * 100,
            r=sensor_r.distance * 100))
```

    `:.2f` changes the way text is output, to state that there are always two decimal places. Because debug output can be essential to see what is going on in the robot, knowing how to refine it is a valuable skill.

11. Running the code with this change gives the following output:

```
pi@myrobot:~ $ python3 test_distance_sensors.py
Prepare GPIO Pins
Left: 6.56, Right: 10.48
Left: 5.20, Right: 11.58
```

You've demonstrated that the distance sensor is working. Added to this is exploring how you can tune the output from a sensor for debugging, something you'll do a lot more when making robots. To make sure you're on track, let's troubleshoot anything that has gone wrong.

Troubleshooting

If this sensor isn't working as expected, try the following troubleshooting steps:

- Is anything hot in the wiring? Hold the wires to the sensor between the thumb and forefinger. *Nothing should be hot or even warming*! If so, remove the batteries, turn off the Raspberry Pi, and thoroughly check all wiring against *Figure 8.12*.
- If there are syntax errors, please check the code against the examples. You should have installed Python libraries with `pip3` and be running with `python3`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- If you are still getting errors, or invalid values, please check the code and indentation.
- If the values are always `0`, or the sensor isn't returning any values, then you may have swapped trigger and echo pins. Try swapping the trigger/echo pin numbers in the code and testing it again. *Don't* swap the cables on a live Pi! Do this one device at a time.
- If you are still getting no values, ensure you have purchased 3.3 V-compatible systems. The HC-SR04 model will not work with the bare Raspberry Pi.
- If values are way out or drifting, then ensure that the surface you are testing on is hard. Soft surfaces, such as clothes, curtains, or your hand, do not respond as well as glass, wood, metal, or plastic. A wall works well!
- Another reason for incorrect values is the surface may be too small. Make sure that your surface is quite wide. Anything smaller than about 5 cm square may be harder to measure.
- As a last resort, if one sensor seems fine, and the other wrong, it's possible that a device is faulty. Try swapping the sensors to check this. If the result is different, then a sensor may be wrong. If the result is the same, it is the wiring or code that is wrong.

You have now troubleshooted your distance sensor and made sure that it works. You have seen it output values to show that it is working and tested it with objects to see its response. Now, let's step up and write a script to avoid obstacles.

Avoiding walls – writing a script to avoid obstacles

Now that we have tested both sensors, we can integrate them with our robot class and make obstacle avoidance behavior for them. This behavior loop reads the sensors and then chooses behavior accordingly.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Adding the sensors to the robot class

So, before we can use the sensors in a behavior, we need to add them to the `Robot` class, assigning the correct pin numbers for each side. This way, if pin numbers change or even the interface to a sensor changes, behaviors will not need to change:

1. To use the `DistanceSensor` object, we need to import it from `gpiozero`; the new code is in bold:

```
from Raspi_MotorHAT import Raspi_MotorHAT
from gpiozero import DistanceSensor
```

2. We create an instance of one of these `DistanceSensor` objects for each side in the robot class. We need to set these up in the constructor for our robot. We use the same pin numbers and queue length as in our test:

```
class Robot:
    def __init__(self, motorhat_addr=0x6f):
        # Setup the motorhat with the passed in address
        self._mh = Raspi_MotorHAT(addr=motorhat_addr)

        # get local variable for each motor
        self.left_motor = self._mh.getMotor(1)
        self.right_motor = self._mh.getMotor(2)

        # Setup The Distance Sensors
        self.left_distance_sensor =
DistanceSensor(echo=17, trigger=27, queue_len=2)
        self.right_distance_sensor =
DistanceSensor(echo=5, trigger=6, queue_len=2)

        # ensure the motors get stopped when the code
exits
        atexit.register(self.stop_all)
```

Adding this to our robot layer makes it available to behaviors. When we create our robot, the sensors will be sampling distances. Let's make a behavior that uses them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Making the obstacle avoid behaviors

This chapter is all about getting a behavior; how can a robot drive and avoid (most) obstacles? The sensor's specifications limit it, with smaller objects or objects with a soft/fuzzy shell, such as upholstered items, not being detected. Let's start by drawing what we mean in *Figure 8.18*:

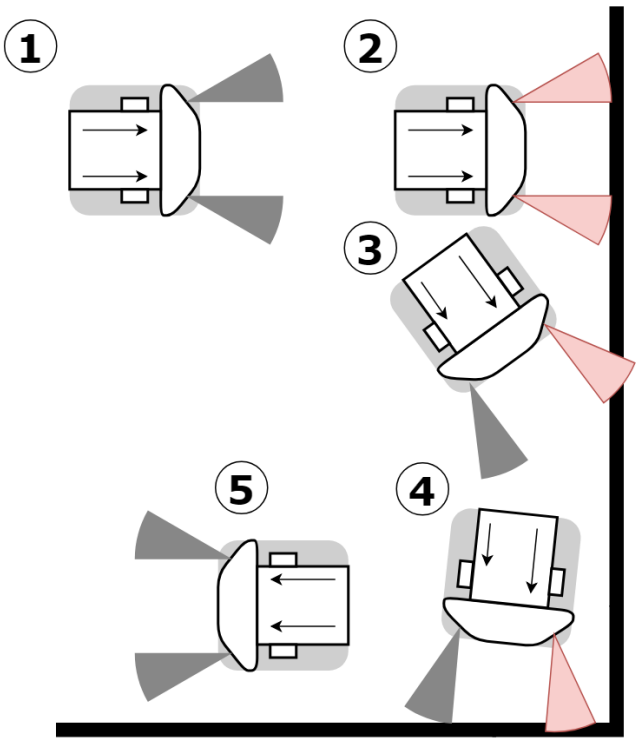


Figure 8.18 – Obstacle avoidance basics

In our example (*Figure 8.18*), a basic robot detects a wall, turns away, keeps driving until another wall is detected, and then turns away from that. We can use this to make our first attempt at wall-avoiding behavior.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

First attempt at obstacle avoidance

To help us understand this task, the following diagram shows a flow diagram for the behavior:

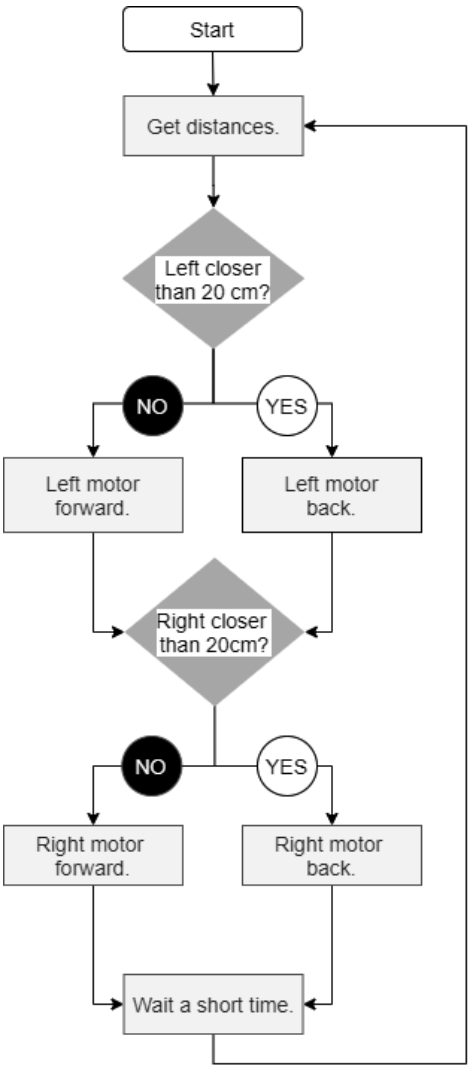


Figure 8.19 – Obstacle avoidance flowchart

The flow diagram in *Figure 8.19* starts at the top.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This diagram describes a loop that does the following:

1. The **Start** box goes into a **Get Distances** box, which gets the distances from each sensor.
2. We test whether the left sensor reads less than 20 cm (a reasonable threshold):

   a) If so, we set the left motor in reverse to turn the robot away from the obstacle.

   b) Otherwise, we drive the left motor forward.
3. We now check the right sensor, setting it backward if closer than 20 cm, or forward if not.
4. The program waits a short time and loops around again.

We put this loop in a `run` method. There›s a small bit of setup required in relation to this. We need to set the pan and tilt to `0` so that it won't obstruct the sensors. I've put this code in `simple_avoid_behavior.py`:

1. Start by importing the robot, and `sleep` for timing:

```
from robot import Robot
from time import sleep
...
```

2. The following class is the basis of our behavior. There is a robot object stored in the behavior. A speed is set, which can be adjusted to make the robot go faster or slower. Too fast, and it has less time to react:

```
...
class ObstacleAvoidingBehavior:
    """Simple obstacle avoiding"""
    def __init__(self, the_robot):
        self.robot = the_robot
        self.speed = 60
        ...
```

3. Now the following method chooses a speed for each motor, depending on the distance detected by the sensor. A nearer sensor distance turns away from the obstacle:

```
    ...
    def get_motor_speed(self, distance):
        """This method chooses a speed for a motor based
on the distance from a sensor"""
        if distance < 0.2:
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            return -self.speed
        else:
            return self.speed
    ...
```

4. The `run` method is the core, since it has the main loop. We put the pan and tilt mechanism in the middle so that it doesn't obstruct the sensors:

```
    ...
    def run(self):
        self.robot.set_pan(0)
        self.robot.set_tilt(0)
```

5. Now, we start the main loop:

```
        while True:
            # Get the sensor readings in meters
            left_distance = self.robot.left_distance_
sensor.distance
            right_distance = self.robot.right_distance_
sensor.distance
            ...
```

6. We then print out our readings on the console:

```
            ...
            print("Left: {l:.2f}, Right: {r:.2f}".
format(l=left_distance, r=right_distance))
            ...
```

7. Now, we use the distances with our `get_motor_speed` method and send this to each motor:

```
            ...
            # Get speeds for motors from distances
            left_speed = self.get_motor_speed(left_
distance)
            self.robot.set_left(left_speed)
            right_speed = self.get_motor_speed(right_
distance)
            self.robot.set_right(right_speed)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Since this is our main loop, we wait a short while before we loop again. Under this is the setup and starting behavior:

```
            ...
            # Wait a little
            sleep(0.05)

bot = Robot()
behavior = ObstacleAvoidingBehavior(bot)
behavior.run()
```

The code for this behavior is now completed and ready to run. It's time to try it out. To test this, set up a test space to be a few square meters wide. Avoid obstacles that the sensor misses, such as upholstered furniture or thin obstacles such as chair legs. I've used folders and plastic toy boxes to make courses for these.

Send the code to the robot and try it out. It drives until it encounters an obstacle, and then turns away. This kind of works; you can tweak the speeds and thresholds, but the behavior gets stuck in corners and gets confused.

Perhaps it's time to consider a better strategy.

More sophisticated object avoidance

The previous behavior can leave the robot stuck. It appears to be indecisive with some obstacles and occasionally ends up ramming others. It may not stop in time or turn into things. Let's make a better one that drives more smoothly.

So, what is our strategy? Well, let's think in terms of the sensor nearest to an obstacle, and the furthest. We can work out the speeds of the motor nearest to it, the motor further from it, and a time delay. Our code uses the time delay to be decisive about turning away from a wall, with the time factor controlling how far we turn. This reduces any jitter. Let's make some changes to the last behavior for this:

1. First, copy the `simple_avoid_behavior.py` file into a new file called `avoid_behavior.py`.
2. We won't be needing `get_motor_speed`, so remove that. We replace it with a function called `get_speeds`. This takes one parameter, `nearest_distance`, which should always be the distance sensor with the lower reading:

```
...
    def get_speeds(self, nearest_distance):
        if nearest_distance >= 1.0:
            nearest_speed = self.speed
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            furthest_speed = self.speed
            delay = 100
        elif nearest_distance > 0.5:
            nearest_speed = self.speed
            furthest_speed = self.speed * 0.8
            delay = 100
        elif nearest_distance > 0.2:
            nearest_speed = self.speed
            furthest_speed = self.speed * 0.6
            delay = 100
        elif nearest_distance > 0.1:
            nearest_speed = -self.speed * 0.4
            furthest_speed = -self.speed
            delay = 100
        else: # collison
            nearest_speed = -self.speed
            furthest_speed = -self.speed
            delay = 250
        return nearest_speed, furthest_speed, delay
...
```

These numbers are all for fine-tuning. The essential factor is that depending on the distance, we slow down the motor further from the obstacle, and if we get too close, it drives away. Based on the time delay, and knowing which motor is which, we can drive our robot.

3. Most of the remaining code stays the same. This is the `run` function you've already seen:

```
    ...
    def run(self):
        # Drive forward
        self.robot.set_pan(0)
        self.robot.set_tilt(0)
        while True:
            # Get the sensor readings in meters
            left_distance = self.robot.left_distance_
sensor.distance
            right_distance = self.robot.right_distance_
sensor.distance            # Display this
            self.display_state(left_distance, right_
distance)
            ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. It now uses the `get_speeds` method to determine a nearest and furthest distance. Notice that we take the `min`, or minimum, of the two distances. We get back the speeds for both motors and a delay, and then print out the variables so we can see what's going on:

```
            ...
            # Get speeds for motors from distances
            nearest_speed, furthest_speed, delay = self.
get_speeds(min(left_distance, right_distance))
            print(f"Distances: l {left_distance:.2f},
r {right_distance:.2f}. Speeds: n: {nearest_speed}, f:
{furthest_speed}. Delay: {delay}")
            ...
```

We've used an *f-string* here, a further shortcut from `.format` (which we used previously). Putting the letter prefix `f` in front of a string allows us to use local variables in curly brackets in the string. We are still able to use `.2f` to control the number of decimal places.

5. Now, we check which side is nearer, left or right, and set up the correct motors:

```
            ...
            # Send this to the motors
            if left_distance < right_distance:
                self.robot.set_left(nearest_speed)
                self.robot.set_right(furthest_speed)
            else:
                self.robot.set_right(nearest_speed)
                self.robot.set_left(furthest_speed)
            ...
```

6. Instead of sleeping a fixed amount of time, we sleep for the amount of time in the `delay` variable. The delay is in milliseconds, so we need to multiply it to get seconds:

```
            ...
            # Wait our delay time
            sleep(delay * 0.001)
...
```

7. The rest of the code remains the same. You can find the full code for this file at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter8`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

When you run this code, you should see smoother avoidance. You may need to tweak the timings and values. The bottom two conditions, reversing and reverse turning, might need to be tuned. Set the timings higher if the robot isn't quite pulling back enough, or lower if it turns away too far.

There are still flaws in this behavior, though. It does not construct a map at all and has no reverse sensors, so while avoiding objects in front, it can quite quickly reverse into objects behind it. Adding more sensors could resolve some of these problems. Still, we cannot construct a map just yet as our robot does not have the sensors to determine how far it has turned or traveled accurately.

Summary

In this chapter, we have added sensors to our robot. This is a major step as it makes the robot autonomous, behaving on its own and responding in some way to its environment. You've learned how to add distance sensing to our robots, along with the different kinds of sensors that are available. We've seen code to make it work and test these sensors. We then created behaviors to avoid walls and looked at how to make a simplified but flawed behavior, and how a more sophisticated and smoother behavior would make for a better system.

With this experience, you can consider how other sensors could be interfaced with your robot, and some simple code to interact with them. You can output data from sensors so you can debug their behavior and create a behavior to make a robot perform some simple navigation on its own.

In the next chapter, we look further into driving predetermined paths and straight lines using an encoder to make sure that the robot moves far more accurately. We use an encoder to compare our motor's output with our expected goals and get more accurate turns.

Exercises

1. Some robots get by with just a single sensor. Can you think of a way of avoiding obstacles reliably with a single sensor?
2. We have a pan/tilt mechanism, which we use later for a camera. Consider putting a sensor on this, and how to incorporate this into a behavior.
3. The robot behavior we created in this chapter can reverse into things. How could you remedy this? Perhaps make a plan and try to build it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

Please refer to the following links for more information:

- The RCWL-1601 is still quite similar to the HC-SR04. The HC-SR04 data sheet has useful information about its range. You can find the data sheet at `https://www.mouser.com/ds/2/813/HCSR04-1022824.pdf`.
- ModMyPi has a tutorial with an alternative way to wire the original HC-SR04 types, and level shift their IO: `https://www.modmypi.com/blog/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi`.
- Raspberry Pi Tutorials also has a breadboard layout and Python script, using `RPi.GPIO` instead of `gpiozero`, at `https://tutorials-raspberrypi.com/raspberry-pi-ultrasonic-sensor-hc-sr04/`.
- We've started to use many pins on the Raspberry Pi. When trying to ascertain which pins to use, I highly recommend visiting the Raspberry Pi GPIO at `https://pinout.xyz/`.
- We briefly mentioned debug output and refining it. W3schools has an interactive guide to Python format strings at `https://www.w3schools.com/python/ref_string_format.asp`.
- There are many scholarly articles available on more interesting or sophisticated object behavior. I recommend reading *Simple, Real-Time Obstacle Avoidance Algorithm* (`https://pdfs.semanticscholar.org/519e/790c8477cfb1d1a176e220f010d5ec5b1481.pdf`) for mobile robots for a more in-depth look at these behaviors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9
Programming RGB Strips in Python

LED lights can be used with a robot to debug and give it feedback so that the code running on the robot can show its state. Colored RGB LEDs let you mix the red, green, and blue components of light to make many colors, adding brightness and color to a robot. We have not paid much attention to making it look fun, so this time we will focus on that.

Mixing different sequences of LEDs can be used to convey information in real time. You can use which are on/off, their brightness, or their color to represent information. This feedback is easier to read than a stream of text, which will help as sensors are added to the robot. This also means the code on the robot can show state without relying on the SSH terminal to do so.

In this chapter, we will learn the following:

- What is an RGB strip?
- Comparing light strip technologies
- Attaching the light strip to the Raspberry Pi

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Making a robot display a code object
- Using the light strip for debugging the avoid behavior
- Making a rainbow display with LEDs

Technical requirements

To build this you will need the following:

- A computer with internet access and Wi-Fi
- The robot, a Raspberry Pi, and the code from the previous chapter
- The Pimoroni LED SHIM

The code for this chapter is on GitHub at `https://github.com/PacktPublishing/Learn-Robotics-Fundamentals-of-Robotics-Programming-Second-Edition/blob/master/chapter9`.

Check out the video at the following link to see the Code in Action: `https://bit.ly/39vglXm`.

What is an RGB strip?

Using lights to display data can be a simple yet flexible way to get data to the user without connecting a full display. For example, a single light could be turned on or off to indicate whether a robot is powered on or the state of a simple sensor. A multicolor light can change color to show more detail, to indicate a few different states that the robot is in. RGB in this chapter stands for Red-Green-Blue, so by controlling the intensity levels of these color channels in a light, multiple colors can be shown. We'll investigate how this happens later in the *RGB values* section.

Adding multiple lights lets you show more data. These can be in a strip (a line of lights), as well as panels/matrixes, rings, and other interesting shapes.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Comparing light strip technologies

There are many competing technologies for lights and light strips. For light types, incandescent lights, such as old light bulbs, tend to use a lot of power and take up too much space to be useful in robots. Fluorescent lights, such as kitchen strips or curly compact types, need complex power systems that also take up too much space. Electroluminescent wire, also known as EL wire, is often used to decorate objects by outlining them; it looks interesting but is tricky to control. **Light Emitting Diode** (**LED**) technology is low power and tends to be small and easy to control, which makes it best suited for robots such as ours. LEDs are also cheap.

The most useful kind, in our case, which we will use in this chapter, are addressable RGB LEDs. Addressable means that each individual LED in the strip can be set to different colors and brightness, allowing a sequence of colors along the strip. To keep it simple, we will use a type with a built-in controller.

Figure 9.1 shows some of the types of addressable RGB LED configurations I have experimented with:

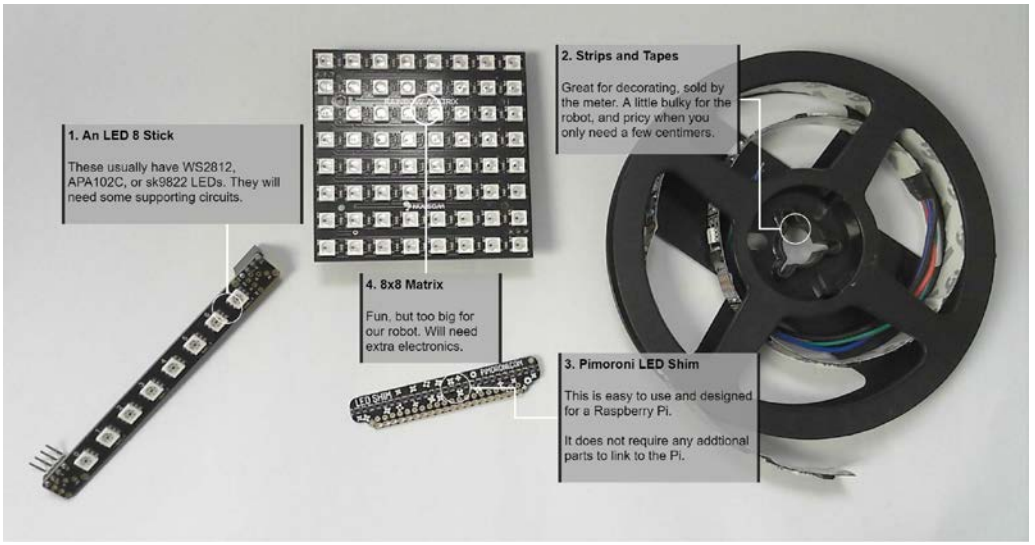


Figure 9.1 – Types of addressable RGB LEDs

All of these LED controllers take a stream of data. Some types, such as the Neopixel, WS2812, SK9822, APA102C, DotStar, and 2801 types, take the red, green, and blue components that they need, and then pass the remaining data to the next LED. Designers arrange these LEDs into strips, rings, or square matrixes, chaining them to take advantage of how they pass the data along. The LED strips can come as rigid sticks or as flexible strips on a reel. For our robot, eight or more LEDs make for a great display.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

There are also some completely different technologies, such as the LED SHIM from Pimoroni and LED matrices in color using shift registers. The Pimoroni LED SHIM is one of the easiest to use (with a Raspberry Pi) of the LED strips. It houses a controller (the IS31FL3731), which is controlled over the I2C data bus. The Pimoroni LED SHIM has 24 LEDs, which is more than enough to cater to our needs. It doesn't need any extra power handling and is also widely available.

Our robot uses the I2C data bus for the motor controller, which happily shares with other devices, such as the LED SHIM, by having a different address. I2C instructions are sent as an address for the device, followed by an I2C register to write and a value for it.

Because of its simplicity and compatibility with our robot, I will continue this chapter with the Pimoroni LED SHIM. This can be bought from Mouser Electronics in most countries, along with Pimoroni, Adafruit, and SparkFun.

RGB values

The colors red, green, and blue can mix to make almost any color combination. Systems express these as RGB values. RGB is the same principle used by most, if not all, color display screens you see. TVs, mobile phones, and computer screens use this. Multicolor LEDs use the same principle to produce many colors. Code usually specifies the amounts of each color to mix as three-number components, as shown in the following diagram:

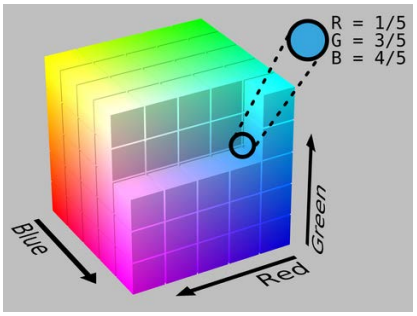


Figure 9.2 – The RGB color space

[SharkD / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0)]

The diagram in *Figure 9.2* shows an RGB color cube. It has arrows showing axes for increasing each of the red, green, and blue components. The exposed surfaces of the cube show different shades and intensities as the color combinations mix throughout the cube.

The corner to the bottom-front-right is blue, the top-front-right is turquoise (mixing blue and green), the bottom-front-left is purple (mixing red and blue), the bottom-far-left is red (with no green or blue), and the top-far-left is yellow (high red and green, no blue).

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

As each value is increased, we get different colors from them being mixed. The top-front-left corner would be the maximum of all three – white. The bottom-rear-right corner would be the minimum of all three – black. The cutout shows a color with intensities as fractions.

In our code, we will use numbers ranging from 0 for absolutely turned off to 255 for full intensity, with values in between for many levels of intensity. The colors are mixed by adding, so adding all of them at full brightness makes white.

Although this theoretically gives many colors, in practice, the differences between the intensity of 250 and 255 are not discernible on most RGB LEDs.

You have seen some of the LED technologies and a little information about how to mix colors for them. We have also made a decision about which technology to use, the Pimoroni LED SHIM. Since we'll be attaching this to our robot, please buy one and come back for the next section.

Attaching the light strip to the Raspberry Pi

Before we write code to display color sequences on the LED SHIM, we need to attach it to the Raspberry Pi on our robot. After we have finished this section, the robot block diagram will look as in *Figure 9.3*:

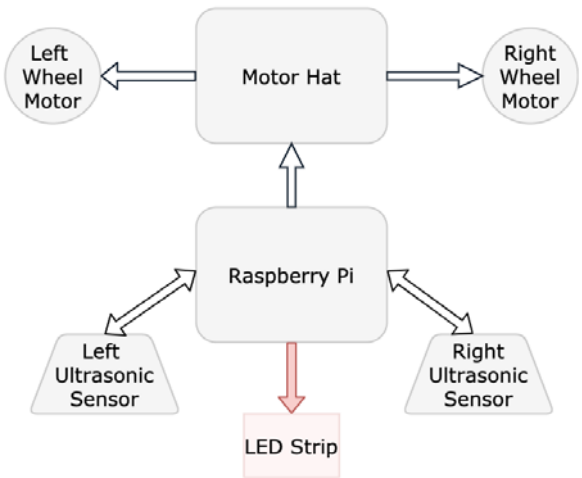


Figure 9.3 – The robot block diagram with the LED strip

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The block diagram now shows the LED strip connected to the Raspberry Pi, with an arrow indicating information flow from the Raspberry Pi to the strip. The strip is highlighted as a new addition to the system. Let's see how this works.

Attaching the LED strip to the robot

The Pimoroni LED SHIM attaches quite readily to the Raspberry Pi. We put it on top of the motor controller, with its pass-through header, so that we can see the lights on top. Take a look at *Figure 9.4* to see how:

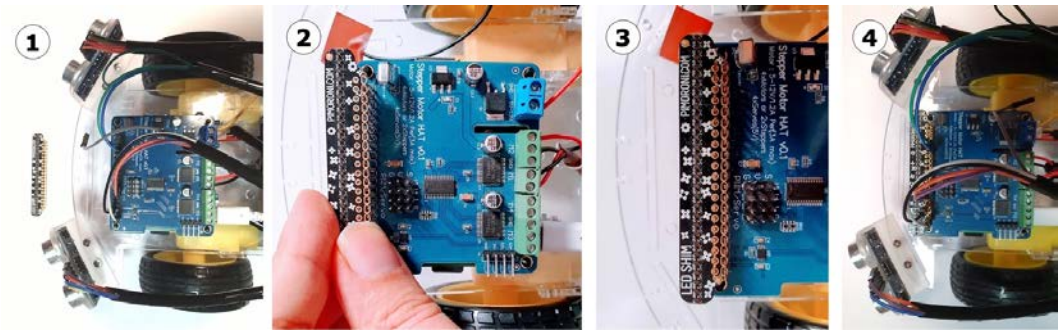


Figure 9.4 – Fitting the LEDs

Use *Figure 9.4* with the following steps to attach the strip:

1. The strip is small. Line the strip up with the header pins coming from the top of the motor HAT. You will need to unplug the wires already plugged into the Raspberry Pi to add the SHIM.
2. The wider part should be sticking out from the HAT. Gently push the SHIM onto the pins, only a little at first, working across the strip until all the pins are in the holes – it is slightly stiff but should grip on.
3. Once all the pins are in, evenly push the SHIM down so that the pins mostly stick out.
4. Now, you'll need to replace the wires. Refer to *Figure 8.15* in *Chapter 8*, *Programming Distance Sensors with Python*, for the distance sensor wiring information.

Now that you've attached the LED SHIM, this robot is ready to light up. Let's program it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Making a robot display the code object

Although we are building around the Pimoroni LED SHIM, we've already seen that there are other types of RGB LED systems. Since we might later swap the SHIM out for a different system, it would be a good idea to make an interface on top of the LEDs. Like the motor's interface, this decouples handling hardware and making behaviors.

Making an LED interface

So, what interface do we want for the LEDs? First, we want them to be available on the robot as `robot.leds`. We want to clear the LEDs (turn them all off), set each individual LED to a different color, and set a bunch/range of LEDs to a list of colors.

It's useful for the code to tell us how many LEDs we have, so if the number changes, the animations or displays still make some sense.

For the colors, we use three values – `r`, `g`, and `b` – to represent the red, green, and blue components. Python has a type called a **tuple**, perfect for making a group from a small number of items such as these colors. When we use `color` as a parameter, this is a tuple of `(r, g, b)`.

The LEDs in strips are addressable, so our code uses an LED number starting at `0`.

So, as a structure, our code starts with `robot.leds`. `leds` will be a member of the existing `robot` class. It is an object with these members:

- `set_one(led_number, color)`: This sets one LED at `led_number` to the specified color.
- `set_range(led_range, color)`: This sets all the LEDs defined by a Python iterable `led_range` to `color`. A Python iterable can be a list of LED numbers [`0`, `3`], or it can be a range made using the `range` function. For example, `range(2,8)` creates the list `[2, 3, 4, 5, 6, 7]`.
- `set_all(color)`: This sets all of the LEDs to the color.
- `clear()`: This clears all of the LEDs to black, turning them all off.
- `show()`: All of the other methods prepare a display, allowing you to set combinations of LEDs. Nothing is updated on the LED device until your code calls this. This method reflects how most LED strips expect to set all the LEDs from one stream of data.
- `count`: This holds the number of LEDs in the strip.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Keeping the preceding points in mind, let's write this code for the LED SHIM:

1. First, we need to install the LED SHIM library. So, on the Raspberry Pi, type the following:

```
pi@myrobot:~ $ pip3 install ledshim
```

2. Our code must start by importing this and setting up the device. Put the following code in `leds_led_shim.py` (named after the device type):

```
import ledshim

class Leds:
    @property
    def count(self):
        return ledshim.width
```

   Our code only needs to use `import ledshim` to set the device up.

   We have set up a property for the number of LEDs in our LED class, called `count`. This property can be read like a variable but is read-only, and our code can't accidentally overwrite it.

3. Now, we create the methods to interact with the strip. Setting a single LED is fairly straightforward:

```
    def set_one(self, led_number, color):
        ledshim.set_pixel(led_number, *color)
```

   While our interface uses a tuple of `(r, g, b)`, the LED SHIM library expects them to be separate parameters. Python has a trick for expanding a tuple into a set of parameters by using an asterisk with the variable name. This expansion is what `*color` means on the second line.

   The LED SHIM code raises `KeyError` if the user attempts to set an LED out of range.

4. Setting a bunch of LEDs is also a simple wrapper in our code:

```
    def set_range(self, led_range, color):
        ledshim.set_multiple_pixels(led_range, color)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. We also want a way to set all of the LEDs. This code is similar to setting a single LED:

```
    def set_all(self, color):
        ledshim.set_all(*color)
```

6. Let's add a method for clearing the LEDs:

```
    def clear(self):
        ledshim.clear()
```

7. Finally, we need the `show` code, to send the colors we've configured to the LEDs. The Pimoroni LED SHIM library has made this very simple:

```
    def show(self):
        ledshim.show()
```

We have installed the LED SHIM library and created an interface for ourselves. We can use this interface to communicate with the LEDs, and it is designed to be swapped out for compatible code for a different type of LED device. Now, we'll make this LED interface available in our `Robot` object.

Adding LEDs to the Robot object

Next, we update our `robot.py` file to deal with an LED system. For this, we do the following:

1. Start by adding the `leds_led_shim` file to the imports (the new code is in bold):

```
from Raspi_MotorHAT import Raspi_MotorHAT
from gpiozero import DistanceSensor
import atexit
import leds_led_shim
```

2. Next, we add an instance of the SHIM to the constructor (`init`) method for `Robot` (the new code is in bold):

```
class Robot:
    def __init__(self, motorhat_addr=0x6f):
        # Setup the motorhat with the passed in address
        self._mh = Raspi_MotorHAT(addr=motorhat_addr)

        # get local variable for each motor
        self.left_motor = self._mh.getMotor(1)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        self.right_motor = self._mh.getMotor(2)

        # Setup The Distance Sensors
        self.left_distance_sensor =
DistanceSensor(echo=17, trigger=27, queue_len=2)
         self.right_distance_sensor =
DistanceSensor(echo=5, trigger=6, queue_len=2)

         # Setup the Leds
         self.leds = leds_led_shim.Leds()
```

3. As we have to stop more than just the motors, we'll swap `stop_motors` for a new `stop_all` method in the `atexit` call to stop other devices (such as the LEDs) too:

```
         # ensure everything gets stopped when the code
exits
         atexit.register(self.stop_all)
```

4. Create the `stop_all` method, which stops the motors and clears the LEDs:

```
    def stop_all(self):
        self.stop_motors()

        # Clear the display
        self.leds.clear()
        self.leds.show()
```

> **Important note**
>
> The complete code can be found at `https://github.com/PacktPublishing/Learn-Robotics-Fundamentals-of-Robotics-Programming-Second-Edition/blob/master/chapter9`.

We have now added support for the LEDs to the `Robot` class, making the interface we designed earlier available, and ensuring that the LEDs are cleared when the robot code exits. Next, we will start testing and turning on LEDs.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Testing one LED

We have installed some hardware, along with a library for it, and then added code to make this available in our robot. However, before we go further, we should make sure that everything works with a test. This is a good place to find any problems and troubleshoot them.

Let's try testing a single LED. One aspect of our robot running Python that we've not explored is that it can run the Python **REPL** – **read, eval, print loop**. What that means is you can start Python and immediately type code to run there. We'll use this to test our LEDs a little:

1. Copy the `leds_led_shim.py` and `robot.py` code onto the Raspberry Pi.
2. SSH into the robot, and just type `python3`. The Raspberry Pi should respond like this:

```
pi@myrobot:~ $ python3
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

3. Let's get our `robot` library ready to use. Type the parts shown in bold:

```
>>> import robot
>>> r = robot.Robot()
```

4. Now, try turning on an LED, setting it to red:

```
>>> r.leds.set_one(0, (255, 0, 0))
```

5. Hmm – nothing happened. Remember, we need to call `leds.show` to display our setting on them:

```
>>> r.leds.show()
```

   You should now see a single red LED.

6. Let's try and set another to purple by mixing red and blue LEDs:

```
>>> r.leds.set_one(5, (255, 0, 255))
>>> r.leds.show()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This adds LED 5, so now two LEDs are lit.

> **Important note**
> Do not forget to use `leds.show` to send the colors to the LED device.

7. To stop this session, press *Ctrl +D* on an empty line. The `atexit` code automatically turns all the LEDs off.

You should now have seen an LED working and lighting up in multiple colors. This demonstrates that the code so far is good. If not, please refer to the following section. If this is all working, skip on ahead to the *Testing all LEDs* section.

Troubleshooting

If you encounter problems trying to light the LEDs, there are some troubleshooting steps you can take.

If running the code shows errors, do the following:

- Check that you have enabled I2C (as shown in *Chapter 7*, *Drive and Turn – Moving Motors with Python*).
- Use `sudo i2cdetect -y 1`, as seen in *Chapter 7*, *Drive and Turn – Moving Motors with Python*. You should see the LEDs at the address `74`.
- Check that you have installed the `ledshim` Python package with `pip3`.
- Carefully check the code for mistakes and errors. If it's the code from GitHub, create an issue!

If the LEDs do not light at all, do the following:

- Try running the example code that comes with the SHIM at `https://github.com/pimoroni/led-shim/tree/master/examples`.
- Ensure you have installed the LEDs the correct way around, as shown in *Figure 9.4*.
- Ensure you have evenly pushed the LED strip down onto the header pins.
- Did you remember to use `leds.show()`?

By following these troubleshooting tips, you will have eliminated the most common issues with this system. You should now have a working LED and be able to proceed to the next section.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Testing all LEDs

Now, we can try the `set_all` method. We'll make something that simply flashes a couple of different colors on the LEDs. Create a file called `leds_test.py`:

1. First, we need imports. We need to import our `Robot` library and `time` to animate this:

```
from robot import Robot
from time import sleep
```

2. Now, let's set up our bot, along with a couple of named colors:

```
bot = Robot()
red = (255, 0, 0)
blue = (0, 0, 255)
```

3. The next part is the main loop. It alternates between the two colors, with `sleep`:

```
while True:
    print("red")
    bot.leds.set_all(red)
    bot.leds.show()
    sleep(0.5)
    print("blue")
    bot.leds.set_all(blue)
    bot.leds.show()
    sleep(0.5)
```

The print methods are there to show when the system is sending the data to the LEDs. We use the `set_all` method to set all the LEDs to `red` and call the `show` method to send it to the device. The code uses `sleep` to wait for half a second, before switching to `blue`.

> **Important note**
>
> The complete code is at `https://github.com/PacktPublishing/Learn-Robotics-Fundamentals-of-Robotics-Programming-Second-Edition/blob/master/chapter8/leds_test.py`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. When you have uploaded these files to the Raspberry Pi, type the following to show the red/blue alternating LED display:

```
pi@myrobot:~ $ python3 leds_test.py
```

5. Press *Ctrl* + *C* on the terminal to stop this running.

We have now shown all the LEDs working. This has also shown them switching between different colors, using timing to produce a very simple animation. We can build on this, producing more interesting uses of color and animation, but first, we will divert to learn a bit more about mixing colors.

Making a rainbow display with the LEDs

Now we get to use these for some fun. We will extend our avoiding behavior from the previous chapter to show rainbow bar graphs on a side corresponding to the distances read. We could also use this for sensors. Before we can link the movement to the animation, how is a rainbow created?

Colour systems

RGB is how the hardware expects colors. However, RGB is less convenient for expressing intermediate colors or creating gradients between them. Colors that appear close to the eye can be a little far apart when in RGB. Because of this, there are other color systems.

The other color system we use is **Hue, Saturation, and Value** (**HSV**). We use HSV in this chapter to make rainbow-type displays and when doing computer vision in a later chapter to assist our code in detecting objects.

Hue

Imagine taking the colors of the spectrum and placing them on a circle, blending through red to orange, orange to yellow, yellow to green, green to blue, blue to purple, and back around to red. The **hue** expresses a point around this circle. It does not affect the brightness of the color or how vivid it is. *Figure 9.5* shows how we can represent these points on a color wheel:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

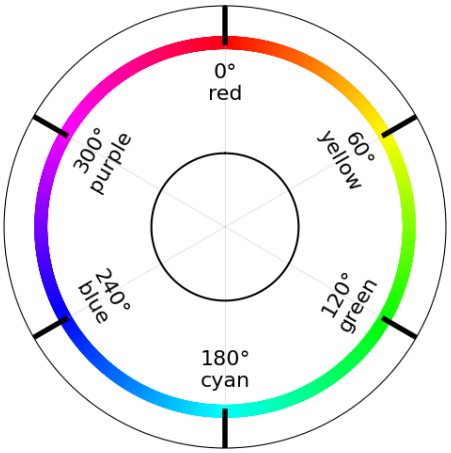


Figure 9.5 – A color wheel of hues

In *Figure 9.5*, the circle shows that around 0 degrees, a red hue is seen. The compass points around the circle correspond to different colors. Colors are blended as you move around from one hue to another. You may have seen something like this in a color wheel gadget on a painting or drawing computer program. A continuous wheel setup like this is what lets us make a rainbow.

Saturation

If you take a color such as red, it can be a grayish/dull red or a vivid, intense red. **Saturation** is an expression of the vividness of the color. As you go toward zero, it only makes shades of gray. As we increase the saturation, color begins to emerge – first in pastel tones, through to poster colors, and then to a striking hazard sign or pure colors at the high end of the saturation scale.

Value

The **value** of the color is its brightness. It ranges from black at 0, through to a very dark version of the color, to a very bright color. Note that this does not approach white (in other words, pink colors), but a very bright red. To make a white color, you need to reduce the saturation too. Other color systems (such as HSL) specify a *light* component that would make things white this way.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Converting HSV to RGB

There are complicated formulas to convert between one color system to another. However, Python can make this conversion itself.

We will use `colorsys.hsv_to_rgb` to make this conversion. It accepts the three HSV components as fractions between 0 and 1, inclusive.

In the case of the hue component, 0 is the start of the circle, 0.5 represents 180 degrees, the half circle, and 1 is all the way around to 360, the full circle.

The saturation component at 0 is gray, fully desaturated, and 1 is the most intense color.

The value component at 0 is black, fully dark, and at 1 is the brightest – a fully lit color.

To make a bright cyan color, we would go past the half circle to about 0.6 for hue, 1.0 for saturation, and 1.0 for value:

```
cyan = colorsys.hsv_to_rgb(0.6, 1.0, 1.0)
```

However, this is not enough. The output from the `colorsys` call is a tuple, a collection of three items for the R, G, and B components.

The output components are in terms of 0 to 1.0, too. Most RGB systems expect values between 0 and 255. To use them, we need to convert these values back up by multiplying them:

```
cyan_rgb = [int(c * 255) for c in cyan]
```

In the preceding line, we loop over each component, `c`, and multiply it by 255. By putting a `for` loop in square brackets like that in Python, we can loop over elements, and put the result back into a list.

Now that you know how to convert HSV values to RGB, let's use this information to make a rainbow.

Making a rainbow on the LEDs

We can use our color system understanding to make a rainbow on the LEDs:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

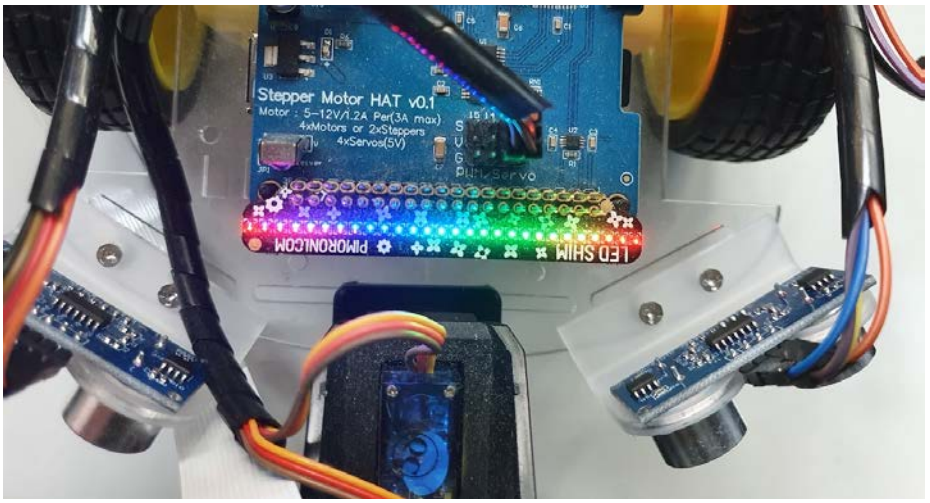


Figure 9.6 – LEDs showing a rainbow on the robot

Figure 9.6 shows a rainbow being displayed on the LEDs attached to the robot.

Let's make this! Make a new file called `led_rainbow.py`:

```
import colorsys

def show_rainbow(leds, led_range):
    led_range = list(led_range)
    hue_step = 1.0 / len(led_range)
    for index, led_address in enumerate(led_range):
        hue = hue_step * index
        rgb = colorsys.hsv_to_rgb(hue, 1.0, 0.6)
        rgb = [int(c*255) for c in rgb]
        leds.set_one(led_address, rgb)
```

Let's go over this file line by line:

- This code starts by importing `colorsys`.
- We define a function, `show_rainbow`, which takes two parameters, a link to our LEDs system (which would usually be given `robot.leds`) and an LED range to set.
- Because we want to know the length of our LED range, we need to make sure it is a list, so we cast this on the first line of our function.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- For a rainbow, the hue value should sweep a full circle. In Python, `colorsys` is the values from 0 to 1. We want to advance the hue a fraction of a step for each LED in the range. By dividing 1.0 by the number of LEDs in a range, we get this fraction.
- We then loop over the LEDs. `enumerate` gives us an index while `led_address` advances. This code makes no assumptions about the range so that it could use an arbitrary list of LEDs.
- We then multiply `hue_step` and `index` to give the `hue` value, the right fraction of 1.0 to use. The following line converts this into an RGB value with a fixed saturation and brightness value.
- Because `colorsys` outputs values between 0 and 1, the code needs to multiply this by 255 and make the resulting number into an integer: `rgb = [int(c*255) for c in rgb]`.
- The code uses the `leds.set_one` method with this RGB value and the LED address.

Let's test this with a file called `test_rainbow.py`:

```
from time import sleep
from robot import Robot
from led_rainbow import show_rainbow

bot = Robot()

while True:
    print("on")
    show_rainbow(bot.leds, range(bot.leds.count))
    bot.leds.show()
    sleep(0.5)
    print("off")
    bot.leds.clear()
    bot.leds.show()
    sleep(0.5)
```

This is quite similar to our previous red/blue test. However, in the first section, we use the `show_rainbow` function, which the code has imported from `led_rainbow`. It passed in the robot's LEDs and makes a range covering all of them.

The code waits for half a second and then clears the LEDs for half a second. These are in a loop to make an on/off rainbow effect. Start this with `python3 test_rainbow.py`, and use *Ctrl* + *C* to stop it after seeing it work.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now that you've seen some simple animation and multicolor LED usage, we can take this to the next level by making the LEDs respond to sensors.

Using the light strip for debugging the avoid behavior

LEDs in rainbows are fun, and switching colors looks nice. However, LEDs can be used for practical purposes too. In *Chapter 8*, *Programming Distance Sensors with Python*, we added sensors to our robot to avoid obstacles. You can follow along in a PuTTY window, and see what the sensors are detecting by reading the numbers. But we can do better; with the light strip, we can put information on the robot to tell us what it is detecting.

In this section, we will tie the LED output together to values from a behavior, first by basic lighting, and then by making some rainbow colors, too.

Adding basic LEDs to the avoid behavior

Before we get fancy and reintroduce the rainbow, let's start with the basic version. The intent here will be to make two *indicator* bars to the left and right side of the LED bar. For each bar, more LEDs will light when the corresponding distance sensor detects a closer obstacle. We'll make it so that the bars go into the middle, so when a single outer LED is lit, the obstacle is far away. When most or all of the LEDs on one side are lit, the obstacle is much closer.

We need to add a few parts to our avoid behavior:

- Some variables to set up the LED display, and how our distances map to it
- A way to convert a distance in to how many LEDs to show
- A method to display the state of our sensors on the LEDs using the preceding items
- To call the `display_state` method from the behavior's main loop

Let's see how to incorporate the preceding points. Open the `avoid_behavior.py` file that you made in *Chapter 8*, *Programming Distance Sensors with Python*, and follow along:

1. Before we can use the LEDs in this behavior, we need to separate them into the bars. In the `__init__` method of `ObstacleAvoidingBehavior`, add the following:

```
        # Calculations for the LEDs
        self.led_half = int(self.robot.leds.leds_count/2)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Next, we need a color for the LEDs when sensing. I chose red. I encourage you to try another:

```
        self.sense_colour = 255, 0, 0
```

3. With the variables' setup out of the way, let's add a method for converting the distance into LEDs. I added this after the `__init__` method:

```
    def distance_to_led_bar(self, distance):
```

4. The distances are in terms of meters, with 1.0 being 1 meter, so subtracting the distance from 1.0 inverts this. The `max` function will return the largest of the two values, here it is used to ensure we don't go below zero:

```
        # Invert so closer means more LED's.
        inverted = max(0, 1.0 - distance)
```

5. Now, we multiply this number, some fraction between 0 and 1, by the `self.led_half` value to get the number of LEDs to use. We round it up and turn this into an integer with `int(round())`, as we can only have a whole number of LEDs turned on. Rounding means that after our multiplication, if we end up with a value such as 3.8, we round it up to 4.0, then convert it into an integer to light four LEDs. We add 1 to this so that there's always at least one LED, and then return it:

```
        led_bar = int(round(inverted * self.led_half))
        return led_bar
```

6. The next method is a trickier one; it will create the two bars. Let's start by declaring the method and clearing the LEDs:

```
    def display_state(self, left_distance, right_
distance):
        # Clear first
        self.robot.leds.clear()
```

7. For the left bar, we convert the left sensor distance to the number of LEDs, then create a range covering 0 to this number. It uses the `set_range` method to set a bunch of LEDs to `sense_color`. Note that your LEDs might be the other way around, in which case swap `left_distance` and `right_distance` in this `display` method:

```
        # Left side
        led_bar = self.distance_to_led_bar(left_distance)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            self.robot.leds.set_range(range(led_bar), self.
sense_colour)
```

8. The right side is trickier; after converting to an LED count, we need to create a range for the LEDs. The variable `led_bar` holds the number of LEDs to light. To light the right of the bar, we need to subtract this from the count of the LEDs to find the first LED, and create a range starting there to the total length. We must subtract 1 from the length – otherwise it will count 1 LED too far:

```
            # Right side
            led_bar = self.distance_to_led_bar(right_
distance)
            # Bit trickier - must go from below the leds
count up to the leds count.
            start = (self.robot.leds.count - 1) - led_bar
            self.robot.leds.set_range(range(start, self.
robot.leds.count - 1), self.sense_colour)
```

9. Next, we want to show the display we've now made:

```
            # Now show this display
            self.robot.leds.show()
```

10. We then display our readings on the LEDs by calling `display_state` inside the behavior's `run` method. Here are a couple of lines for context, with the extra line highlighted:

```
                # Get the sensor readings in meters
                left_distance = self.robot.left_distance_
sensor.distance
                right_distance = self.robot.right_distance_
sensor.distance
                # Display this
                self.display_state(left_distance, right_
distance)
                # Get speeds for motors from distances
                nearest_speed, furthest_speed, delay = self.
get_speeds(min(left_distance, right_distance))
```

Save this, send it to the Raspberry Pi, and run it. When it's running, you should be able to see the LEDs light up in a bar based on the distance. This is both satisfying and gives a good feel for what the robot is detecting. Let's make this a little more interesting by adding rainbows.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Adding rainbows

We can use our LED rainbow to make our distance-sensing demo even more fun:

Figure 9.7 – Distance sensor rainbow bars

Figure 9.7 shows a photo of the rainbow bars for each distance sensor. This is a great visual demonstration of the LEDs animating.

Since we added a library for showing rainbows, we can reuse it here. Let's see how to do it:

1. Open up the `avoid_behaviour.py` code from the previous section.
2. At the top, import `led_rainbow` so that we can use it:

```
from robot import Robot
from time import sleep
from led_rainbow import show_rainbow
```

3. Our existing code displayed a bar for the left. Instead of a bar, display a rainbow here. We need to ensure we have at least one item:

```
        # Left side
        led_bar = self.distance_to_led_bar(left_distance)
        show_rainbow(self.robot.leds, range(led_bar))
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Once again, the right side will be a little bit trickier; as we want the rainbow to go the other way, we need to make the range count backward for the rainbow too. The Python `range` function, along with the `start` and `end` parameters, takes a step parameter. By making a step of -1, we can count down in the range:

```
        start = (self.robot.leds.count - 1) - led_bar
        right_range = range(self.robot.leds.count - 1,
start, -1)
        show_rainbow(self.robot.leds, right_range)
```

5. Upload this and run it, and the bar graph will be in rainbow colors instead of a solid color.

You have gone from a single LED to a number of LEDs. With some work on color systems, we were able to generate a rainbow and use it to show the status of a behavior.

Summary

In this chapter, you learned how to interact with and use RGB LEDs, as well as how to choose and buy RGB LED strips that work with the Raspberry Pi. You learned how to make code for the LEDs on the robot, using them with robot behaviors. You also saw how the HSV color system works, which can be used to generate rainbows.

You can take the techniques used here to add LED-based status displays to robots and write code to link them with behaviors.

In the next chapter, we will look at servo motors and build a pan and tilt mechanism for moving sensors.

Exercises

1. Try mixing a different RGB color, or looking one up, and using `set_one`, `set_all`, or `set_range` to light LEDs in that color.
2. Use the show left rainbow and show right rainbow functions to make the robot turn on rainbows corresponding to the side it's turning to in the `behaviour_path` code.
3. By making a timer loop and advancing an index or changing a range, it would be possible to animate the rainbows or make them *scan* across the LED bar. Try this out.
4. Could the other parts of the HSV color be used to make pulsing LED strips that change brightness?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

Please refer to the following for more information:

- *Make Electronics: Learning by Discovery*, *Charles Platt*, *Make Community, LLC*: I've only started to cover some basic electronics with the switch and breadboard. To get a real feel for electronics, *Make Electronics* is a superb introduction.
- For more advanced electronics, try *Practical Electronics for Inventors, Fourth Edition*, *Paul Scherz*, *Simon Monk*, *McGraw-Hill Education TAB*: This gives practical building blocks for electronics that can be used to interface a robot controller with almost anything or build new sensors.
- The `colorsys` library, like most Python core libraries, has a great reference: `https://docs.python.org/3/library/colorsys.html`.
- Pimoroni have some other demos with the LED SHIM at `https://github.com/pimoroni/led-shim/tree/master/examples`. These could be fun to adapt to our LED layer.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

10
Using Python to Control Servo Motors

Servo motors can make precise and repeatable motions. Motion such as this is vital for moving sensors, controlling robot legs or arms, and moving them to a known position. Its uses are so engineers can predict robot behavior, so a robot can repeat things in automation, or so code can move limbs to accurately respond to what their sensors are instructing them. Raspberry Pi or add-on boards can control them. In this chapter, we will use these motors to build a pan and tilt mechanism—a head to position a sensor.

In this chapter, we will cover the following topics:

- What are servo motors?
- Positioning a servo motor with the Raspberry Pi
- Adding a pan and tilt mechanism
- Creating pan and tilt code

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

For this chapter, you require the following:

- The robot with the Raspberry Pi built in the previous chapters
- Screwdrivers—small Phillips
- Small pliers or a set of miniature spanners
- Nylon bolts and standoffs kit—2.5 mm
- A two-axis mini pan-tilt micro servo kit
- Two micro SG90/9g or MG90s servo motors, with their hardware and servo horns. The pan-tilt kit may already include the following:
- Cutting pliers or side cutters
- Safety goggles

The code for this chapter is available at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter10/`.

Check out the following video to see the Code in Action: `https://bit.ly/2LKh92g`.

What are servo motors?

Servo motors, or **servomechanism motors**, are used to position robotic appendages such as arms, grippers, legs, and sensor mounts. They create other movements where the position is the main factor, unlike the wheel motors (DC motors), where speed is the controlling factor. Servo motors are used where (to some level of accuracy) turning to a specific place might be required. You can use code to control these precise positioning movements or a sequence of them:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

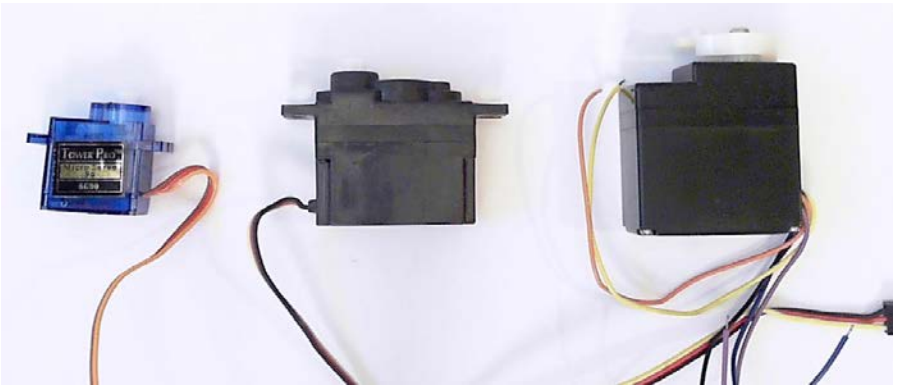


Figure 10.1 – A small selection of servo motors

Servos come in many sizes, from the very small at around 20-30 mm (shown in *Figure 10.1*) to those large enough to move heavy machinery. *Figure 10.2* shows some of the miniature hobby servos I use for my robots:

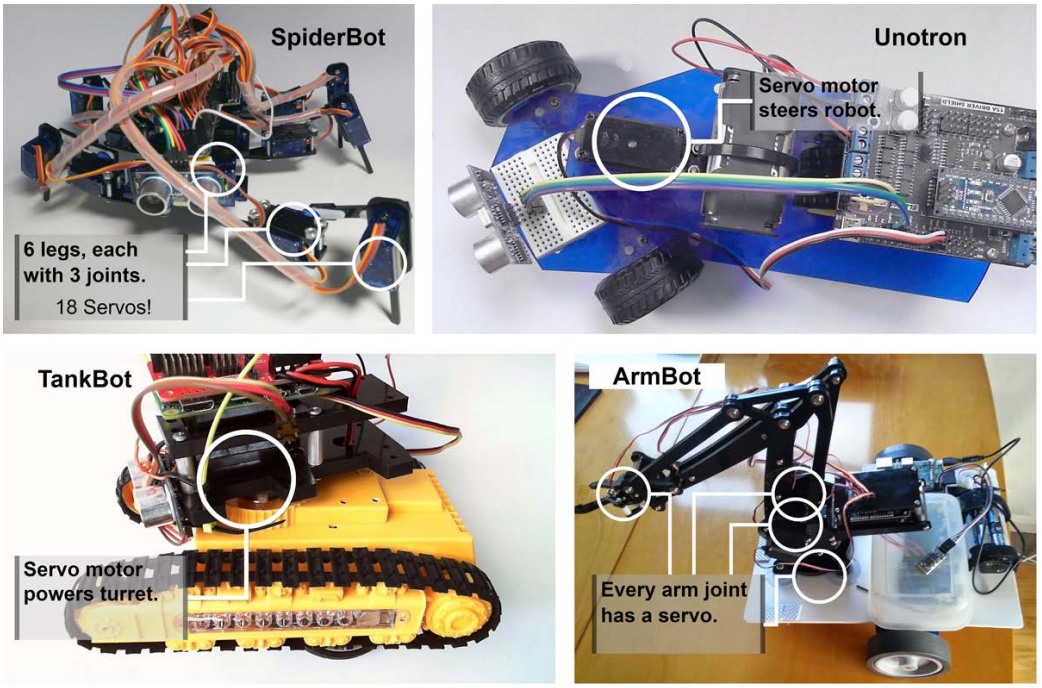


Figure 10.2 – A small selection of servo motors in robots

Now that you've seen where you might use servo motors, we can dive deeper and find out how a servo motor works.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Looking inside a servo

Inside the compact form of a servo motor hides a controller, a DC motor, a gearbox (usually with a stop), and a sensor. These motors have a built-in feedback system. A servo motor takes input from a controller, which specifies a position for the motor to go to. The servo's controller gets the motor's current direction from the internal sensor. The controller compares the current motor position with the position that has been requested and generates an error—a difference. Based on this difference, the servo drives its motor to try and reduce that error to zero. When the motor moves, the sensor changes, and that error value changes, generating feedback and making the control loop shown in *Figure 10.3*:

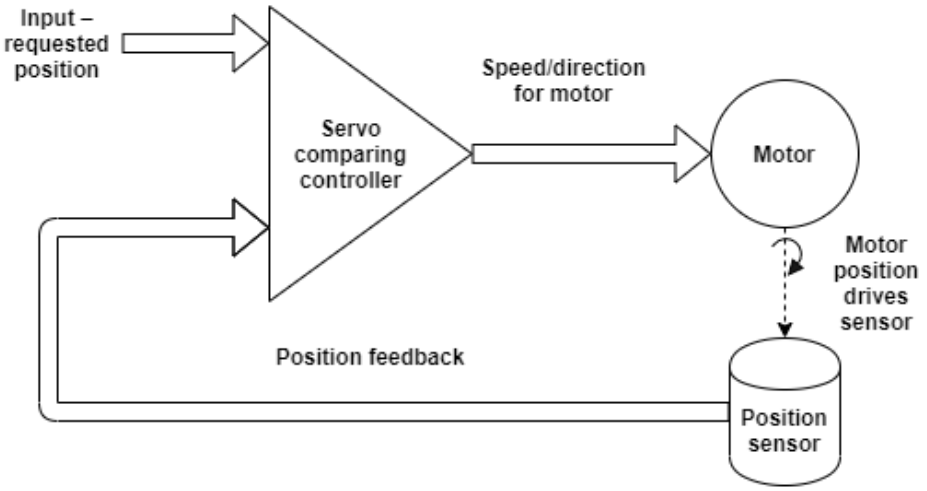


Figure 10.3 – The servo motor control loop

Some types of servo motors, such as those used in ArmBot (*Figure 10.2*), have an additional output allowing you to read the state of the position sensor in your code too.

Sending input positions to a servo motor

Signals are sent to servo motors using **Pulse Width Modulation** (**PWM**). PWM is the same system seen in *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*, and is used on our robot to drive the wheel motors. PWM is a square wave, which has two states, *on* or *off* only. It is the timing of the signal, shown in *Figure 10.4*, which is interesting. You can consider the wave as a stream of pulses, where the length in time of each pulse encodes the position information for the servo controller. These pulses repeatedly cycle, with a period or frequency. People usually express frequency as cycles-per-second or hertz. A shorter pulse is a lower value, and a longer pulse is a higher value. The controller keeps the period/frequency the same, and it is the duty cycle (on-time to off-time ratio) that changes. With servo motors, the pulse length is the information encoding feature:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

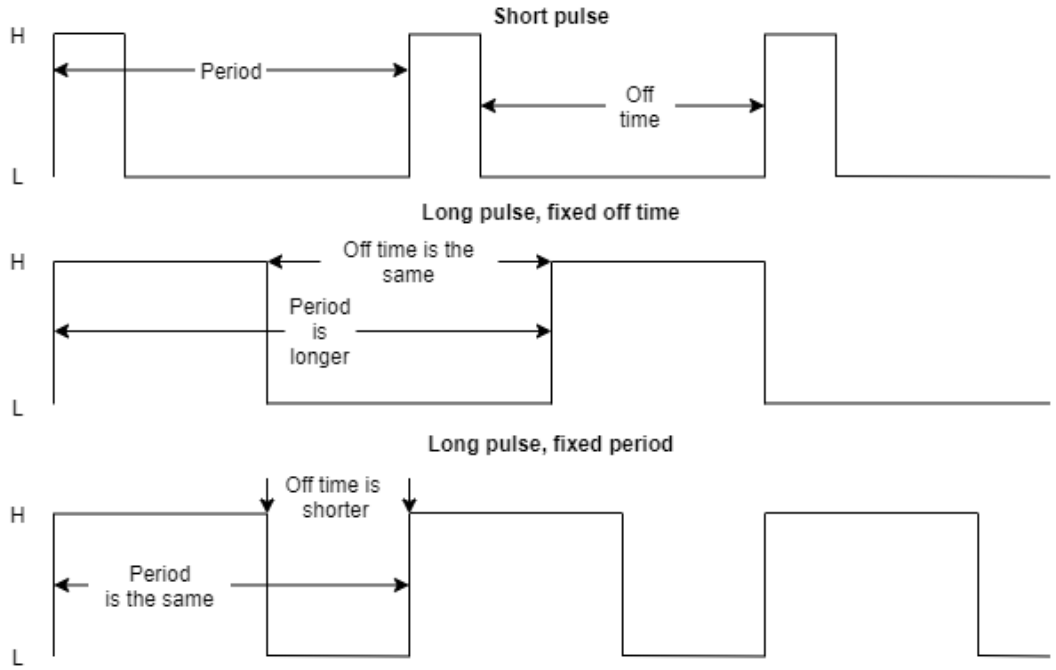


Figure 10.4 – PWM for servo motors

In *Figure 10.4*, each graph has time on the *x*-axis. The *y*-axis for each of the stacked graphs has **L** for a logic-low and **H** for a logic-high. The top graph shows short pulses. The charts at the bottom show increased pulse time; however, they vary in an important aspect. In the middle graph, the off-time has not changed, but the period has changed.

In the bottom graph of *Figure 10.4*, the period is the same as the first graph, but it has a longer pulse time, with shorter off-time. Servo motors measure pulse length and not frequency, so the third graph's variation is the correct type.

In our robot, we already have a chip in the motor controller that performs the fixed period PWM style. The chip is designed to control LEDs, but happily controls other PWM devices. We can control when the off-time and the on-times should start in a fixed period, which means it behaves like the bottom graph for longer pulse widths.

Pulse width control leads us nicely into the next section, where we make our robot generate PWM to position a servo. Let's prepare a servo motor, plug it in, and make it move.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Positioning a servo motor with the Raspberry Pi

To position a servo, we need to set up a **servo horn** to see it move, and then plug it into the motor controller board. A servo horn is a small collar with one or more arms, usually used to connect the servo spindle/axle to a mechanism they move. *Figure 10.5* shows how to attach a horn to a servo:

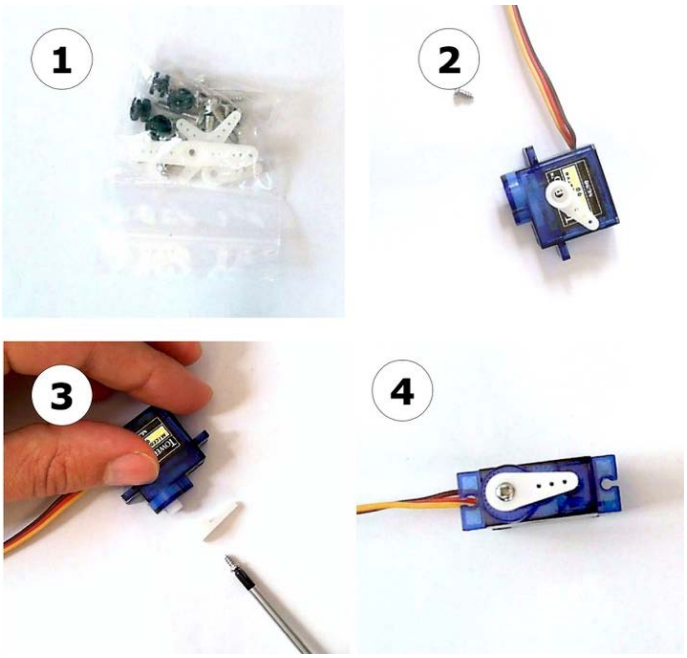


Figure 10.5 – Fitting a servo horn

The images in *Figure 10.5* show how to fit a servo horn. Perform the following steps:

1. Servo motors usually come with small bags of hardware, containing a few different horn types and screws to attach them to the servo and the parts you want them to move.
2. Use the very short small screws for this, as the longer screws can break the servo.
3. Screw a one-armed servo horn into the servo. The long collar of the horn fits over the servo's output spindle.
4. The servo should now look like this. Don't over-tighten the collar screw, as you may need to loosen it and set the middle again.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In the next image, we plug the servo into the control board to test it:

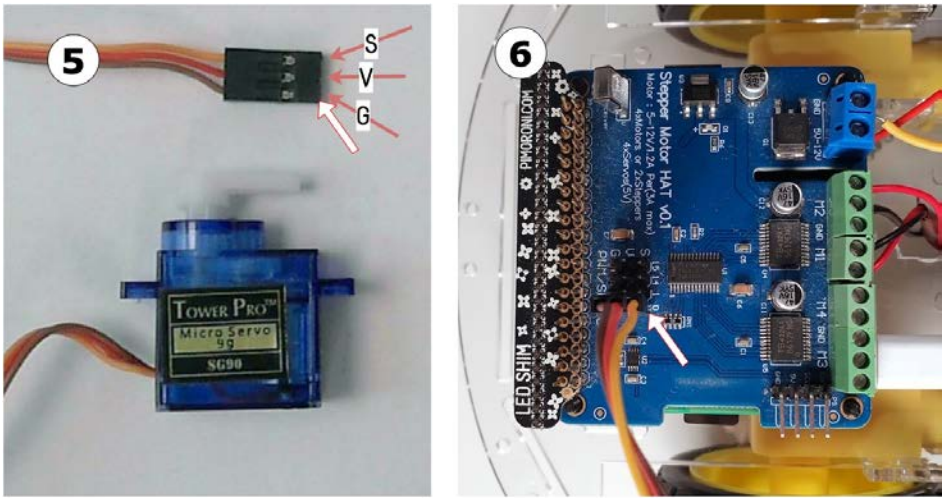


Figure 10.6 – Plugging a servo into the control board

Follow *Figure 10.6* to see how to connect the servo to the Full Function Motor HAT board on the robot. Make sure you power down the robot fully before connecting it:

1. The outline arrow here indicates the servo connector. Servo connectors have three pins shown with the solid arrows: brown for **ground** (**G**), red for **voltage** (**V**), and yellow/orange/white for **signal** (**S**).
2. This panel shows that the Motor HAT has a 4 x 3 block of connectors marked **PWM/Servo**, indicated by the arrows. The four servo channel columns are numbered (**0**, **1**, **14**, and **15**). Each channel has three pins, marked with a pin label (GVS). **GVS** refers to **ground**, **voltage**, and **signal**. Align the yellow wire from the servo connector with row S and the brown wire with row G, with the red wire in the middle. Plug this servo into channel 0.

The connection is similar to controllers such as the PiConZero, but may require some soldering work on other motor boards. Now that you have wired this motor in, we need to write some code to test it.

Writing code for turning a servo

Some board libraries can convert directly from degree to servo movement pulses. With this library, we need to write the code for that math. Some of the calculations result in constants that won't change once we know which servo controller and servo we are using. We can store the results as constants and reuse them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You could store the result of such calculations in a variable, but letting the computer do it has a few advantages:

- The computer is excellent and quick at calculating these constants. It may make rounding errors but is likely to make fewer errors than a human would copying from a calculator.
- It's clear what numbers came from where and how to calculate them. Putting in a *magic number* that came from a calculator makes it harder to see where it came from.
- If you change a tuning factor, the calculations stay fresh.
- If you find an error, it's easy to change it here.

> **Tip**
> When setting out calculations in code, use descriptive variable names and comments—being descriptive helps you understand your code and check whether the math makes sense. You read code many times after writing it, so this principle applies to any variable or function name.

Let's make some test code to move a servo to a position typed by the user in degrees. Put this code in `servo_type_position.py`:

1. The `Raspi_MotorHAT` library we are using for the robot has a `PWM` module, which we import and create an object to manage. We use `atexit` again to ensure that the controller stops signaling the motor:

```
from Raspi_MotorHAT.Raspi_PWM_Servo_Driver import PWM
import atexit
```

2. We must specify the address when setting up the PWM device — it's the same I2C device we are using for the motors and has the same address:

```
pwm = PWM(0x6f)
```

3. The servo works in cycles at 50 Hz; however, we can use 100 Hz so our motors can drive too. If the frequency is low, the DC motors will stall very easily. This frequency will be a time base for our PWM frequency, which we keep to also use in calculations:

```
# This sets the timebase for it all
pwm_frequency = 100
pwm.setPWMFreq(pwm_frequency)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Let's call the middle position 0 degrees. For most servos, turning to -90 degrees requires a pulse of 1 ms; going to the center requires 1.5 ms:

```
# Mid-point of the servo pulse length in milliseconds.
servo_mid_point_ms = 1.5
```

   This code is an example of using descriptive variable names. A variable named `m` or `p` means far less than `servo_mid_point_ms`.

5. Turning it to 90 degrees requires a pulse of 2 ms; this is 0.5 ms from the mid-point, which gives us a primary calibration point for 90 degrees:

```
# What a deflection of 90 degrees is in pulse length in
milliseconds
deflect_90_in_ms = 0.5
```

6. The length of a pulse in our chip also depends on the frequency. This chip specifies a pulse's size as the number of steps per cycle, using 4,096 steps (12 bits) to represent this. A higher frequency would require more steps in the pulse to maintain the pulse length in milliseconds. We can make a ratio for this; steps per millisecond:

```
# Frequency is 1 divided by period, but working ms, we
can use 1000
period_in_ms = 1000 / pwm_frequency
# The chip has 4096 steps in each period.
pulse_steps = 4096
# Steps for every millisecond.
steps_per_ms = pulse_steps / period_in_ms
```

7. Now that we have steps per millisecond and know how many milliseconds a pulse should be for 90 degrees, we can get a ratio of steps per degree. We can also use this to redefine our middle point in steps, too:

```
# Steps for a degree.
steps_per_degree = (deflect_90_in_ms * steps_per_ms) / 90
# Mid-point of the servo in steps
servo_mid_point_steps = servo_mid_point_ms * steps_per_ms
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. We can use these constants in a `convert_degrees_to_pwm` function to get the steps needed for any angle in degrees:

```
def convert_degrees_to_steps(position):
    return int(servo_mid_point_steps + (position * steps_
per_degree))
```

9. Before we move anything, we should make sure the system stops. We stop by setting PWM to `4096`. This number may sound odd, but instead of giving a long pulse, it turns on an additional bit in the control board, which turns the servo pin entirely off. Turning off the pin releases/relaxes the servo motor. Otherwise, the motor would try to seek/hold the last position we gave it until it's powered down. We can use `atexit` to do this, just as we did with stopping motors:

```
atexit.register(pwm.setPWM, 0, 0, 4096)
```

10. We can now ask for user input in a loop. The `input` function in Python asks the user to type something and stores it in a variable. We convert it into an integer to use it:

```
while True:
    position = int(input("Type your position in degrees
(90 to -90, 0 is middle): "))
```

11. We can then convert the position to an end step using our preceding calculations:

```
    end_step = convert_degrees_to_steps(position)
```

12. We can then use `pwm.setPWM` to set our pulse in steps. It takes the servo channel number, a start step, which we'll hold at 0, and an end step, which we've calculated previously:

```
    pwm.setPWM(0, 0, end_step)
```

You can now turn the robot on and send this code to it. When you run this code, it will ask you to type a number. Start with number 0. When you press *Enter*, you will hear the servo move.

You can now try other values, but do not give it values outside of -90 to 90 degrees as you may damage the servo. We add code later to protect against this damage. If this system works, you should see the servo move between each different value.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

If you find problems getting this to run, try the following:

- Ensure that the servo motors are plugged into the correct ports and are the right way around. The **S** pin should go into a yellow cable on most servos.
- Lots of jittering or failing to get to the right position can mean you have less than fresh batteries—please ensure they are fresh.
- When running DC motor behaviors from other chapters, if the servo droops, this may also be down to lower battery power. Make sure you are using metal hydride rechargeables.

Before we move on, how does this Motor HAT control both servo motors and DC motors? Let's take a closer look at it.

Controlling DC motors and servo motors

The HAT that I've suggested for this book (and readers may choose others) is based on a PCA9685 chip, which is popular for making robots such as this. It is a multi-channel PWM controller. Take a look at the diagram for an overview of how it is connected in this robot:

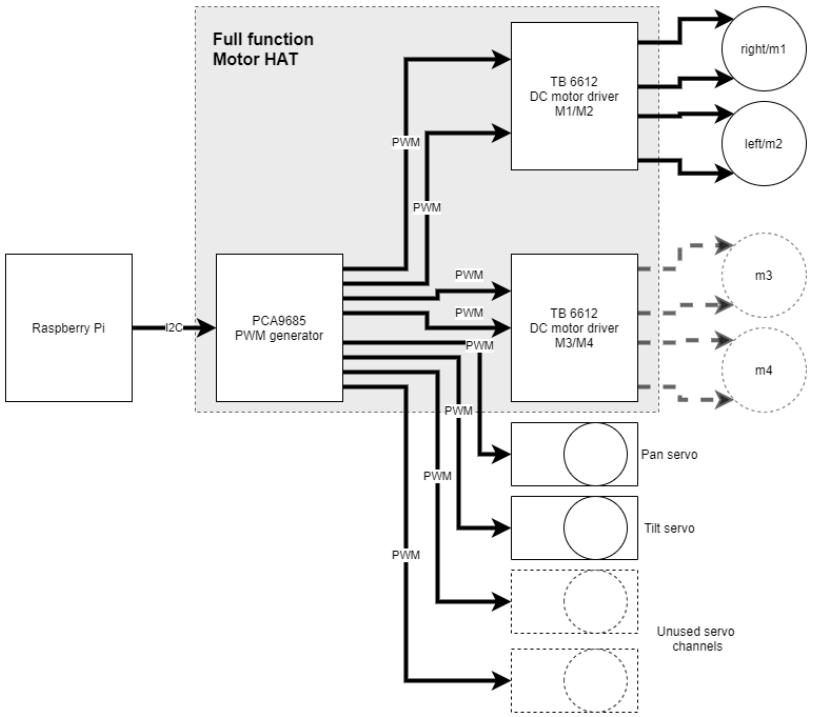


Figure 10.7 – Block diagram of the motor board

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 10.7 shows a block diagram of the Full Function Motor HAT. This is not a wiring diagram, it is still a functional block diagram, but it shows components and connections on the board. On the far left is the Raspberry Pi, with a line connecting the PCA9685 chip. This line is the I2C communication going into the hat, shown by the labeled gray box.

The PWM generator has many connections out. Eight of these outputs go to control TB6612 motor drivers. These have power outputs suitable for DC motors (or stepper motors). They are still in the gray box as they are part of the hat. We connected those power outputs to our right motor (m1), left motor (m2), and have a space m3/m4 connection for other motors.

The servo channels expose four of these PWM outputs directly. We'll connect the pan servo to one output and tilt to another.

In the preceding code, I mentioned driving the PWM chip at 100 Hz instead of 50. This is because if we combine servo motors and DC motors, the time base for the chip applies to all the PWM outputs, even if the duty cycle (on-time to off-time ratio) changes.

Now that you have tested the basics, we can calibrate the servo, finding where the 0 position is and making sure 90 degrees is moving by the right amount.

Calibrating your servos

The servo horn allows you to see the servo motor's movement. Zero should be close to the middle:

1. First, use a screwdriver with the horn to line zero up with the middle. Loosen it, lift it, move it around to the middle, and then push it down again. Do not tighten this much as we will be removing this again.

> **Important note**
> If the servo motor's motion is impeded, including an attempt to move it past its limits, it pulls higher currents to try and reach the position. Stalling a servo like that can cause a lot of heat and damage to the stalled motor.

2. Now try entering 90 and -90. You may find the two sides are not reaching 90 because servos can vary slightly. Increase the `deflect_90_in_ms` value to adjust the motor range. Do so in small 0.1 increments, as going too far here may lead to servo damage.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. When you've calibrated your servo, it is a good idea to ask each servo to go to position 0 before starting the next step. You can do this by plugging the second servo into channel 1 on the servo's connector, and then swapping the first parameter of every call to the `pwm.setPWM` method from 0 to 1.

You have now tested some basic servo code, and then tried both servos and two channels. We can now use the servo motors to build a pan and tilt mechanism to point sensors.

Adding a pan and tilt mechanism

We are now going to build and add a pan and tilt servo mechanism to our robot. This mechanism is like a head for our robot to mount sensors on it. A **pan and tilt mechanism**, shown in *Figure 10.8*, moves a sensor (or anything else) through two axes under servo motor control.

Pan means to turn left or right. *Tilt* means to tilt up or down:

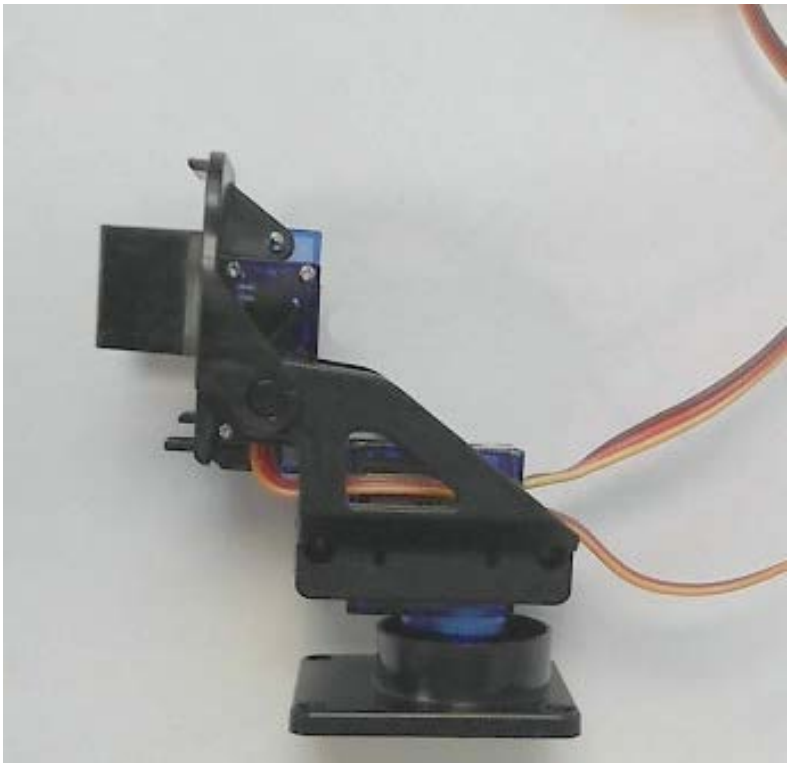


Figure 10.8 – A pan and tilt mechanism from a typical kit

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Kits like the one in *Figure 10.8* are available from Mouser, along with Adafruit outlets. You may need to purchase the servo motors separately. There are other types of pan-tilt. Ensure it is the type that uses two servos and refer to the manufacturer's documentation where it is different. We build the kit, mount it onto our robot, and plug it into the controller.

Our robot block diagram with the servos looks like *Figure 10.9*:

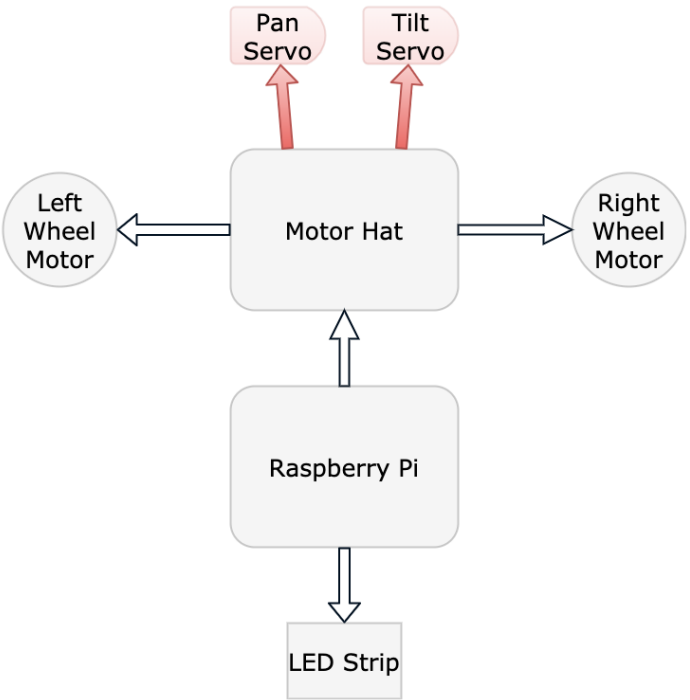


Figure 10.9 – Block diagram of the robot with servo motors added

The block diagram in *Figure 10.9* extends the block diagram from the previous chapter by adding the pan and tilt servos. These connect to the Motor HAT.

Now that you've seen how it fits in the robot block diagram, it's time to build it!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Building the kit

You need your pan and tilt kit, a screwdriver, and a cutter. *Figure 10.10* shows the parts of the mechanism laid out:

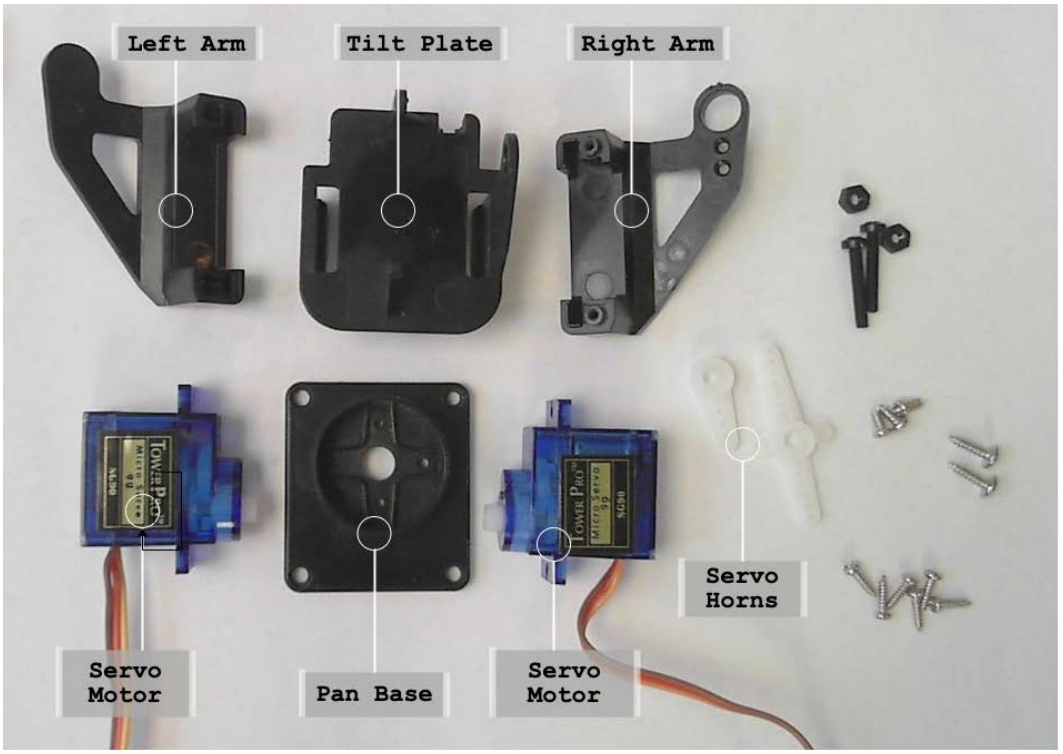


Figure 10.10 – The parts of the pan-tilt mechanism

Take note of the terms I use for the different plastic parts in *Figure 10.10*; I use those for the assembly. Next to these are the screws that would have come with the kit too. There are usually self-tapping M2 screws in a servo motor's hardware bag – please ensure you have them.

> **Important note**
>
> The plastic here may ping off, so don't do this without safety goggles. Be aware of other people in the room and tiny sharp plastic bits landing. Please wear safety goggles for this step!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Once you have your parts ready, we'll begin by assembling the base:

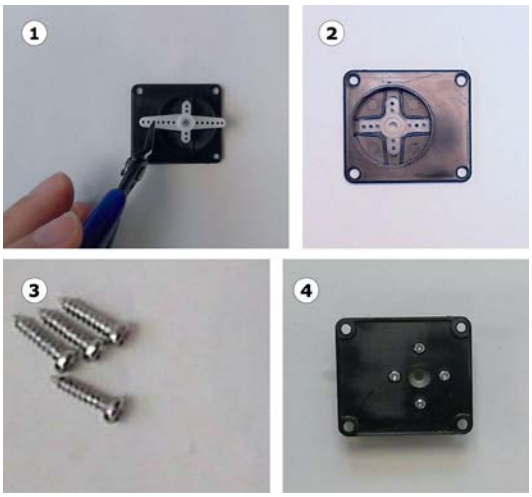


Figure 10.11 – Preparing the pan base

Let's assemble the base, as shown in *Figure 10.11*, with the help of the following steps:

1. Measure out and cut a cross-shaped servo horn to fit the base. There are ridges it must fit into in the base. Shorten the servo horn's long arms to just over three holes and make them slightly thinner with the cutters.
2. Line up the servo horn in the base, so the arms are in the recessed area, and the servo horn collar is facing away from the base.
3. Find four of the long M2 self-tapping screws.
4. Screw the servo horn into the base. Note that, with some servo horns, only the horizontal or the vertical screws may line up; two is sufficient, but four are more secure.

Our base is now ready. Next, we'll assemble the left arm:

Figure 10.12 – Assembling the left arm and tilt plate

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

To assemble the left arm, perform the following steps:

1. Line up the stud with the hole on the tilt plate, as shown in *Figure 10.12*.
2. Push the stud into this hole; you will need to hold this in place for the next step.
3. Take one of the servo motors and the two screws with collars. The servo rests on the two brackets on the tilt plate, and when screwed in, holds the left arm in place. Ensure that the servo's spindle aligns with the stud and hole before screwing it in.

Great! Now let's move on to the right arm:

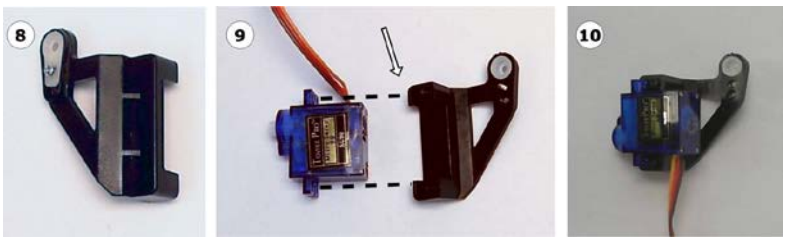


Figure 10.13 – Assembling the right arm

Follow these steps to assemble the arm:

1. To assemble the right arm, you need another servo horn—this time, the kind with just a collar and a single straight arm. As shown in *Figure 10.13*, the servo horn needs trimming to fit the intended recess on the right arm. Use one of the M2 self-tapping screws to bolt this onto the right arm of the mechanism. The servo horn you have attached is at the front of the mechanism.
2. Flip this assembly over and slot another servo (this is the pan servo) into the slots as indicated.
3. It should have the spindle facing the bottom of the photo, as shown in the third panel. This servo motor faces downward.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Our next step is to combine the left and right arm that we just created. Follow along:

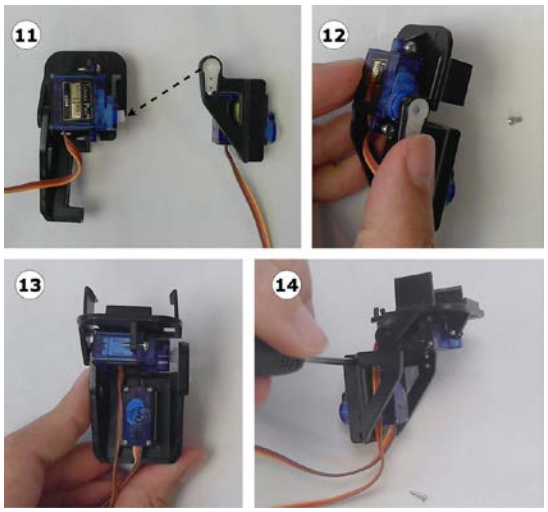


Figure 10.14 – Combining the left arm and right arm

You need to follow these steps to combine the arms:

1. *Figure 10.14* shows how to bring the left and right arm of the mechanism together. When combining the arms, the right arm servo horn's collar should clip around the tilt servo you screwed onto the tilt plate.
2. The pan servo, in the left-arm assembly, fits into a matching cut-out.
3. Use one of the short screws to attach the collar of the right arm to the tilt servo, keeping the tilt plate upright.
4. Use two of the small thin screws to screw the two arms together.

We're almost there. The last part is to combine the base that we initially created to the rest of the mechanism:

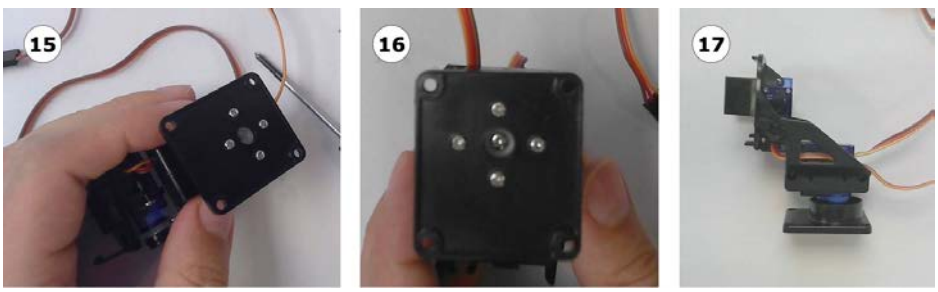


Figure 10.15 – Combining the base with the mechanism

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Continue with the procedure as follows:

1. *Figure 10.15* shows how to attach the mechanism to its base. Push the collar from the servo horn screwed into the base onto the pan servo spindle. Line it up so that the long axis of the base is in line with the bottom of the mechanism.
2. Use one of the very short screws to bolt the collar onto the servo.
3. The final panel shows the fully assembled pan and tilt mechanism.

You've seen how a pan and tilt mechanism goes together around the servos. Assembling constructions like this is valuable for seeing how these mechanisms work and getting a feel for what the servo motors will do when they move. Now that the pan and tilt mechanism has been assembled, we need to attach it to the robot before we can move the head around.

Attaching the pan and tilt mechanism to the robot

The mechanism needs to become part of the robot. We need both to attach it physically and wire it in place so that the motor controller can send signals to it.

Figure 10.16 shows how to attach the pan and tilt mechanism to the robot:

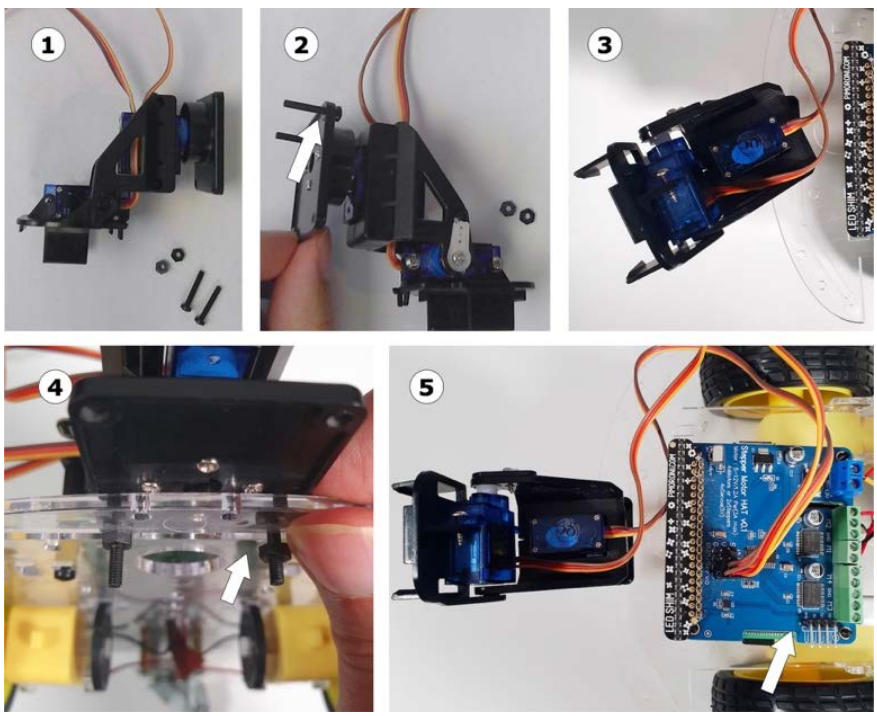


Figure 10.16 – Attaching the pan and tilt mechanism to the robot

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Follow these instructions, along with the steps shown in *Figure 10.16*:

1. For this, you need two long bolts and two nuts to attach the pan and tilt mechanism to the robot.
2. Drop the bolts into the short end of the pan and tilt base so that they are pointing down.
3. The chassis I recommended has a slot across the front, which came in handy for the line sensor. This slot is perfect for mounting this pan and tilt mechanism, with the screws through the slot. On another chassis, you may need to measure and drill holes for this.
4. Thread on the nuts and tighten from beneath the robot.
5. Wire in the servos. The tilt (up and down servo) should plug into servo channel 0, and the pan (left and right) should plug into servo channel 1.

Congratulations! Your robot is ready, and the hardware setup is complete. You are now ready to write code and try out the new head for your robot. So, let's dive straight in.

Creating pan and tilt code

We build our pan and tilt code in layers. We create a `Servos` class and put the previous calculations into it. We set up our robot class to have an instance of the `Servos` class, and ways to access the servo to pan and the servo to tilt.

Making the servo object

In this class, we encapsulate (internally manage the details of) converting an angle into a servo movement, and the quirks, such as channel numbers, of our servo board. We make a `Servos` class in a `servos.py` file for this:

1. The `servos.py` file starts with an import and then goes straight into the constructor (the `__init__` function):

```
from Raspi_MotorHAT.Raspi_PWM_Servo_Driver import PWM

class Servos:
  def __init__(self, addr=0x6f, deflect_90_in_ms=0.6):
```

   Here we have an address for the PWM device. There's a deflection/calibration parameter called `deflect_90_in_ms` so that it can be overridden with the value obtained while calibrating your servos.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Next, we will add a comment, so when we use the `Servos` class, we can see what we meant. The text in here will show up as help for our class in some code editors:

```
      """addr: The i2c address of the PWM chip.
      deflect_90_in_ms: set this to calibrate the servo
motors.
      it is what a deflection of 90 degrees is
      in terms of a pulse length in milliseconds."""
```

   The triple-quoted string at the top of the constructor is a convention known as a **docstring** in Python. Any string declared at the top of a function, method, class, or file becomes a special kind of comment, which many editors use to show you more help for the library. It's useful in any kind of library layer. The convention of using a docstring will complement all of the explanatory comments that we'll carry in from the test code.

3. The next section of the `__init__` method should look familiar. It sets up all the calculations created in *steps 3* to *7* of the *Writing code for turning a servo* section within the `servos` object. We are storing the PWM object in `self._pwm`. We only keep some of the variables for later by storing them in `self`, and the rest are intermediate calculations:

```
        self._pwm = PWM(addr)
        # This sets the timebase for it all
        pwm_frequency = 100
        self._pwm.setPWMFreq(pwm_frequency)
        # Mid-point of the servo pulse length in
milliseconds.
        servo_mid_point_ms = 1.5
        # Frequency is 1/period, but working ms, we can
use 1000
        period_in_ms = 1000 / pwm_frequency
        # The chip has 4096 steps in each period.
        pulse_steps = 4096
        # Steps for every millisecond.
        steps_per_ms = pulse_steps / period_in_ms
        # Steps for a degree
        self.steps_per_degree = (deflect_90_in_ms *
steps_per_ms) / 90
        # mid-point of the servo in steps
        self.servo_mid_point_steps = servo_mid_point_ms *
steps_per_ms
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. In the last part of the `__init__` method, we create `self._channels`; this variable lets us use channel numbers 0, 1, 2, and 3, and maps them to the quirky numbers on the board:

```
        # Map for channels
        self._channels = [0, 1, 14, 15]
```

5. Next, we want a safety function to turn all of the servo motors off. Sending no pulse at all does that and releases the servos, protecting power and saving the motors from damage. This function uses the trick seen in *step 9* of the *Writing code for turning a servo recipe*, setting a start time of `0` and `4096` for the off flag to generate no pulse:

```
    def stop_all(self):
        # 0 in start is nothing, 4096 sets the OFF bit.
        off_bit = 4096
        self._pwm.setPWM(self.channels[0], 0, off_bit)
        self._pwm.setPWM(self.channels[1], 0, off_bit)
        self._pwm.setPWM(self.channels[2], 0, off_bit)
        self._pwm.setPWM(self.channels[3], 0, off_bit)
```

6. Next is the conversion function, which we saw in *step 8* of the *Writing code for turning a servo section*, but localized to the class. We will only use this conversion internally. The Python convention for this is to prefix it with an underscore:

```
    def _convert_degrees_to_steps(self, position):
        return int(self.servo_mid_point_steps + (position
* self.steps_per_degree))
```

7. Also, we need a method to move the servo to an angle. It will take a channel number and an angle. I've used another docstring in this method to explain what it does and what the limits are:

```
    def set_servo_angle(self, channel, angle):
        """position: The position in degrees from the
center. -90 to 90"""
```

8. The next couple of lines validate the input. It limits the angle to protect the servo from an out-of-range value, raising a Python exception if it's outside. An exception pushes a problem like this up to calling systems until one of them handles it, killing the code if nobody does:

```
        # Validate
        if angle > 90 or angle < -90:
            raise ValueError("Angle outside of range")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. The last two lines of this method set the position:

```
        # Then set the position
        off_step = self._convert_degrees_to_steps(angle)
        self._pwm.setPWM(self.channels[channel], 0, off_
step)
```

You can find the full code at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter10/servos.py`.

This class is now ready to incorporate into our robot. Let's do this in the next section.

Adding the servo to the robot class

Before we start using the preceding `Servos` class in behaviors, we will incorporate it into our `robot.py` file and assign specific purposes to specific servo motors. This way, a behavior could use a different robot with differently configured pan and tilt mechanisms by swapping out the robot class:

1. Next, we need to patch this into the `Robot` class in `robot.py`. First, let's import it after the `leds` import:

```
import leds_led_shim
from servos import Servos
...
```

2. This `servos` object then needs to be set up in the constructor for `Robot`, passing along the address:

```
class Robot:
    def __init__(self, motorhat_addr=0x6f):
        # Setup the motorhat with the passed in address
        self._mh = Raspi_MotorHAT(addr=motorhat_addr)

        # get local variable for each motor
        self.left_motor = self._mh.getMotor(1)
        self.right_motor = self._mh.getMotor(2)

        # Setup the Leds
        self.leds = leds_led_shim.Leds()

        # Set up servo motors for pan and tilt.
        self.servos = Servos(addr=motorhat_addr)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        # ensure the motors get stopped when the code
exits
        atexit.register(self.stop_all)
...
```

3. Now, we should make sure it stops when the robot stops by adding it to the `stop_all` code:

```
    def stop_all(self):
        self.stop_motors()

        # Clear the display
        self.leds.clear()
        self.leds.show()

        # Reset the servos
        self.servos.stop_all()
...
```

4. The last thing to do in `robot` is to map setting pan and tilt values to the actual servo motors:

```
    def set_pan(self, angle):
        self.servos.set_servo_angle(1, angle)

    def set_tilt(self, angle):
        self.servos.set_servo_angle(0, angle)
```

Our `Robot` object now has methods to interact with the pan and tilt servos on the robot chassis. This gives us specific controls for servos on the robot and presents a layer to use in behaviors. In the next section, we will make a behavior that uses these to make circles.

Circling the pan and tilt head

In this section, we make the pan and tilt head move in small circles of around 30 degrees. This behavior demonstrates the mechanism and the parts of the code to talk to it. The code creates a repeating animated kind of behavior that uses a time base—a current time. We use the time base to draw the circle:

1. Create a new file; I suggest the name `circle_pan_tilt_behavior.py`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. We start with a number of imports; the `Robot` object, the `math` library, and some timing:

```
from time import sleep
import math

from robot import Robot
```

   We prepare the `math` library as we are going to use sine and cosine to calculate that circle.

3. As our behavior has local data, we will put it into a class of its own. The constructor (`the__init__` method) takes the `Robot` object:

```
class CirclePanTiltBehavior:
    def __init__(self, the_robot):
        self.robot = the_robot
```

4. This behavior is essentially an animation, so it has a time and count of *frames* or positions for each circle. We use a `frames_per_circle` variable to adjust how many steps it goes through:

```
        self.current_time = 0
        self.frames_per_circle = 50
```

5. The math functions work in radians. A full circle of radians is *2 times pi*. We divide that by `frames_per_circle` to make a multiplier we call `radians_per_frame`. We can multiply this back out with the current frame to give us a radian angle for the circle later:

```
        self.radians_per_frame = (2 * math.pi) / self.
frames_per_circle
```

   We work with radians and not degrees here because we'd end up with a constant multiplier, taking the degrees into radians and dividing by frames per circle, so we'd end up back with `radians_per_frame`.

6. Being a circle, it should also have a radius, representing how far our servos deflect from the middle:

```
        self.radius = 30
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. The next method in the behavior is `run`. This puts the behavior in a `while True` loop, so it runs until the user stops it:

```
    def run(self):
        while True:
```

8. When our behavior runs, we then take `current_time` and turn it into a frame number using the modulo (remainder) operation on `frames_per_circle`. The modulo constrains the number between zero and the number of frames:

```
            frame_number = self.current_time % self.
frames_per_circle
```

9. We then take this `frame_number` variable and turn it back into radians, a position around the circle, by multiplying it back with `radians_per_frame`. This multiplication gives us a value we call `frame_in_radians`:

```
            frame_in_radians = frame_number * self.
radians_per_frame
```

10. The formula for drawing a circle is to make one of the axes the cosine of the angle, times the radius, and the other the sine of the angle, times the radius. So, we calculate this and feed each axis to a servo motor:

```
            self.robot.set_pan(self.radius * math.
cos(frame_in_radians))
            self.robot.set_tilt(self.radius * math.
sin(frame_in_radians))
```

11. We perform a small `sleep()` to give the motors time to reach their position, and then add one to the current time:

```
            sleep(0.05)
            self.current_time += 1
```

12. That entire `run` method together (*steps 7-11*) is as follows:

```
    def run(self):
        while True:
            frame_number = self.current_time % self.
frames_per_circle
            frame_in_radians = frame_number * self.
radians_per_frame
            self.robot.set_pan(self.radius * math.
cos(frame_in_radians))
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            self.robot.set_tilt(self.radius * math.
sin(frame_in_radians))
            sleep(0.05)
            self.current_time += 1
```

13. Finally, we just want to start up and run our behavior:

```
bot = Robot()
behavior = CirclePanTiltBehavior(bot)
behavior.run()
```

So, we've built a `Servos` class, and incorporated it to control the pan and tilt mechanism in our `Robot` code. You've seen code to move servo motors in an animation-like way. We can combine this with the physical pan-tilt in the next section to see this run.

Running it

You need to send `servos.py`, `robot.py`, and `circle_pan_tilt_behavior.py` to the Raspberry Pi over SFTP. On the Raspberry Pi, type `python3 circle_pan_tilt_behaviour.py` to see it. The head should now be making circles.

This is demonstration code for the device, but will later be able to use the same device to track faces by mounting a camera on it. The use of frames here to create an animation is important for making smooth predetermined movements with a robot, controlling small movements over time.

Troubleshooting

If this does not run, please try the following:

- Ensure you were able to test the servos as shown in the *Writing code for turning a servo* section. There is no need to disassemble the pan and tilt mechanism for this, but please make sure you have made a servo move with the code there and followed that troubleshooting section.
- If you see errors while running this, please ensure you are running with `python3`. Please ensure that you have checked for typos in your code.
- If the code fails to import anything, ensure that you have copied over all the preceding files and that you have installed/set up the libraries in previous chapters.
- If the motors move to an extreme position, you may have missed the step to calibrate them. You will need to unbolt and pop each out of the servo horn, send it to position 0, using the test code from the *Writing code for turning a servo* section, and then push them back in at a neutral position, screwing them back in.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- If a servo refuses to move at all, check that it has been plugged in the right way, ensuring that G corresponds to the black servo wire, V corresponds to the red wire, and S to the yellow signal wire from the servo. The robot code has assumed that the servo motors are plugged into channels 0 and 1 of the motor control board.
- Ensure that there are no breaks in the wires or their insulation. I have seen a batch of servo motors of this type with wire problems and had to return them. You should not be able to see any bare patches of wire.
- Ensure that the connectors are pushed in firmly. If they are loose here, then a signal may not be getting to the servo motors.
- It's worth saying again: low batteries will make a servo jitter or fail to reach a set point.

You should now have troubleshot the common problems seen with running this circle servo motor behavior and see the head making small and slow circles. We will be able to use this system in a later chapter to look at faces.

Building a scanning sonar

Using the distance sensor we attached in *Chapter 8*, *Programming Distance Sensors with Python*, with the pan and tilt mechanism allows us to set up an interesting experiment. If we attach the distance sensor to the head, and then slowly sweep in a direction (for example, the pan direction), we can create a sensor sweep of an area. We can then use some Python code to plot this, making a small map of things in front of the robot.

A sensor similar to this combination is found in advanced robots (like those from Boston Dynamics) and autonomous cars. LIDAR and RADAR sensors use laser light or radio frequencies with a fast spinning drum to perform the same kind of sweeps far faster than our example. LIDAR sensors are starting to appear on the hobbyist market, but are still a little costly.

To visualize this, we are going to use a special kind of chart – a polar plot. This plots around a circle, with the *x*-axis being where we are around a circle (in radians – multiples of pi). The *y*-axis forms how far a plotted point is from the center of a circle – so a larger value will be further out. This lends itself very well to this example because we are sweeping the servo through angles and receiving distance data. We will have to account for the servo working in degrees and translate to radians for the graph output.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Attaching the sensor

In this step, we'll extend a sensor's wires and reposition the sensor onto the pan-and-tilt head. Start with the robot powered down:

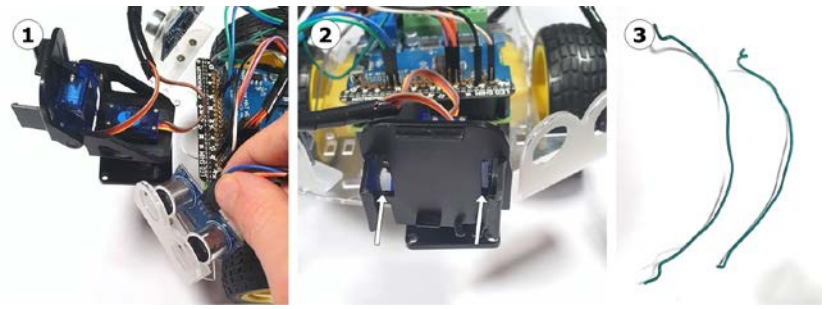


Figure 10.17 – Pop out the sensor and prepare two wires

Follow these steps to reposition the sensor together with *Figure 10.17*:

1. You will need to pop one of the distance sensors out of the mount on the robot's front. I used the left.
2. On the pan/tilt head there, identify the small slots.
3. Make two lengths of single-core wire or sandwich ties. A length of about 18 cm for each should suffice:

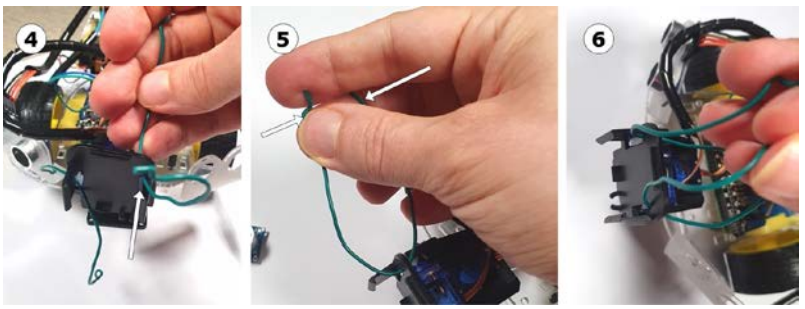


Figure 10.18 – Steps 4 to 6; putting wires in the slots

4. For each side, first push the wire through the indicated slot shown in *Figure 10.18*.
5. Bend it around a little, pushing the two ends nearly together so that it doesn't just drop out.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Prepare the other side the same way, so both sides are ready:

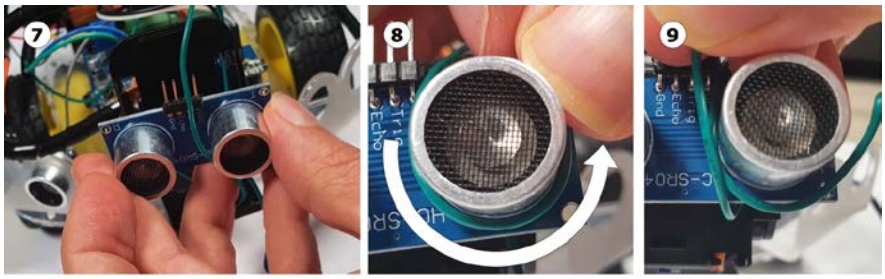


Figure 10.19 – Wrapping wire underneath the sensor

7. Put the sensor in place and bend the left wire coming from the top of the head around the sensor's front.
8. Now bend it under the big round element (the ultrasonic transducer) indicated by the white arrow in *Figure 10.19*.
9. Bend the wire sticking out underneath over the sensor:

Figure 10.20 – Twist the wire and repeat for the right

10. Wrap this wire around the top of the left transducer and bring the two wire ends together.
11. Twist the top and bottom ends together, as shown in *Figure 10.20*.
12. Repeat *steps 8* to *11* for the wires on the right:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 10.21 – The sensor temporarily secured to the pan-and-tilt head

13. The sensor should now be temporarily secured to the pan-and-tilt head, as shown in *Figure 10.21*. You are now ready to wire it back in:

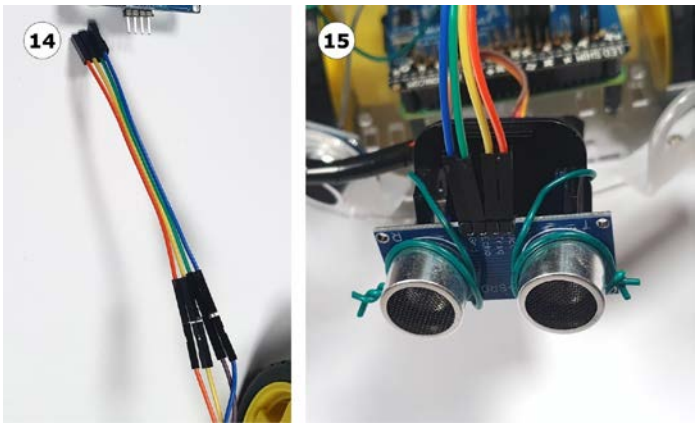


Figure 10.22 – Extend the wires and plug the wires in

You may need to extend the jumper wires, as shown in *Figure 10.22*. You are likely to have more male-female jumper cables left, so use four of those to extend the sensor. Be careful to ensure that you make the same connections through these to the sensor. It helps to use the same colors if you can.

14. Now make the wire connections to the sensor. Refer back to *Chapter 8*, *Programming Distance Sensors with Python*, for reference.

I suggest running the `test_distance_sensor.py` code from *Chapter 8*, *Programming Distance Sensors with Python*, and checking that the sensor is working before you continue.

Now that you have mounted the sensor on the head, it will move when we instruct the servo motors to move. Let's make sure that we have the right tools on the Raspberry Pi for the code first.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Installing the library

The code will use the Python tool `matplotlib` to output the data. It makes a **polar** plot, a graph originating radially from a point, which will look like a sonar scan you may have seen in movies. To do this, you will need to install Matplotlib (and its dependencies) onto your Raspberry Pi.

SSH (putty) into the Raspberry Pi and type the following to get the packages that Matplotlib requires:

```
$ sudo apt update
$ sudo apt install libatlas3-base libgfortran5
```

The next thing you need is Matplotlib itself:

```
$ pip3 install matplotlib
```

Matplotlib may take a short time to install and install many helper packages along the way.

With the library installed, you are ready to write the code.

Behavior code

To make our plot and get the data, the code will move the sensor to the full extent on one side, and then move back in steps (for example, 5 degrees), measuring at each. This example will show using a sensor and servo motor together and introduce other ways to view sensor data.

Create a file named `sonar_scan.py` and follow these steps for its content:

1. We'll start with some imports; the `time` import, with which we can give motor and sensors time to work, `math` to convert from degrees to radians, `matplotlib` to make the display, and `robot` to interface with the robot:

```
import time
import math
import matplotlib.pyplot as plt
from robot import Robot
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. We then have some setup parameters. We put these out here to encourage you to experiment with different turn speeds and extents:

```
start_scan =0
lower_bound = -90
upper_bound = 90
scan_step = 5
```

3. Next, let's initialize the `Robot` object and ensure the tilt is looking horizontally:

```
the_robot = Robot()
the_robot.set_tilt(0)
```

4. We need to prepare a place to store our scan data. We will use a dictionary mapping from a heading in degrees to the value sensed there:

```
scan_data = {}
```

5. The scan loop starts from the lower bound and increments by the scan step up to the upper bound; this gives us our range of facings:

```
for facing in range(lower_bound, upper_bound, scan_step):
```

6. For each facing in the loop, we need to point the sensor, and wait for the servo to move and for the sensor to get readings. We negate the facing here because the servo motor turns the opposite way to the polar plot:

```
    the_robot.set_pan(-facing)
    time.sleep(0.1)
```

7. We then store the sensor distance as centimeters in the scan data for each facing, using the facing as its key:

```
    scan_data[facing] = the_robot.left_distance_sensor.
distance * 100
```

8. The following loop converts the facings into radians. The servo works in degrees, but a quirk of Matplotlib is that the polar axis must be in radians:

```
axis = [math.radians(facing) for facing in scan_data.
keys()]
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Now we make our polar plot, telling Matplotlib the axis and the data, and that we want a green line with `g-`:

```
plt.polar(axis, list(scan_data.values()), 'g-')
```

10. The last line writes this plot out to a `png` image file, so we can use `scp` to download it from the Raspberry Pi and view the plot:

```
plt.savefig("scan.png")
```

Put this code on the Raspberry Pi, place the robot somewhere with a few obstacles less than a meter away, and run the code with `python3 sonar_scan.py`. You should see the servo motor make a sweep of the bounds.

When this runs, the `scan.png` output should look something like *Figure 10.23*:

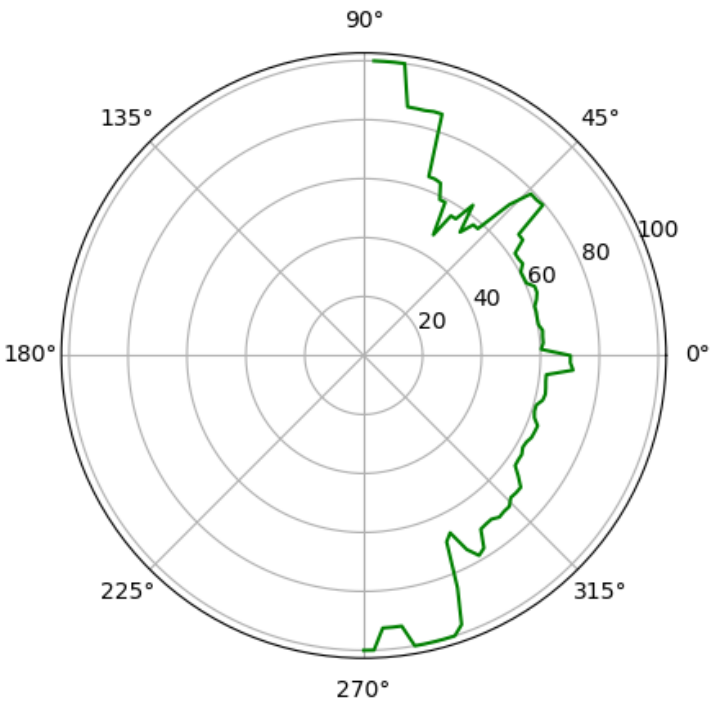


Figure 10.23 – A sonar scan plot of my lab

Figure 10.23 shows the sonar scan output on a polar plot. It shows the measurements in degrees, with a green line tracing the contours of items detected in front of the sensor. In this image, my lower bound was -90, my upper bound 90, and my step at 2 degrees for slightly finer resolution.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Choosing a finer resolution (less than 2 degrees) will make it slower. The sleep value could be tuned, but lower values risk the servo not settling or the sensor not producing further readings.

Troubleshooting

If you encounter problems while running this tool, please try the following:

- First, ensure the distance sensor works by following the troubleshooting and testing steps in *Chapter 8*, *Programming Distance Sensors with Python*. Verify the wiring and use the test code.
- Verify that the servo motors are working, as shown in the preceding *Creating the pan and tilt code* section. Follow the troubleshooting procedures there.
- If there are errors running the code, ensure that you have installed all the libraries needed.
- Check that there are no typos in the code you have entered.
- Remember, the file output will be on the Raspberry Pi, so you will have to copy it back to view it.
- It can be helpful to print values before they go into the `plt.polar` method. Add the following:

```
print(axis)
print(scan_data.values())
```

You should now have been able to make a sonar scan and get a plot like the one above. I suggest you experiment with the values to create different plot resolutions and put different object combinations in front of the sensor.

Let's summarize what we've seen in the chapter.

Summary

In this chapter, you have learned about servo motors, how to control them with your motor controller, and how they work. You've built a pan and tilt mechanism with them and added code to the `Robot` object to work with that mechanism. Finally, you've demonstrated all of the parts with the circling behavior.

You will be able to use the code you've seen to control other servo motor systems, such as robot arms. The animation style techniques can be useful for smooth movement and circular motions. I used a system a little like this when controlling the 18 motors in SpiderBot's legs.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You've seen how to use a servo with a sensor on a head to make some kind of map of the world and related it to the LIDAR systems used on bigger and more expensive robots.

In the next chapter, we will look at another way to map and observe the world with encoders. These sensors will detect wheels turning on our robot to determine how our robot is moving.

Exercises

1. Consider how you might build other servo-based extensions. A robot arm needs at least four servos, but a simple gripper/grabber can use the two additional channels our robot has left.
2. Look around at kits for a gripper, a design with a pincer, and perhaps an up/down control.
3. How would you write the code for this gripper? What would you add to the `Robot` object?
4. What demo behavior would you make for this gripper?
5. A gripper might move too violently when just given a different position to be in. How would you make a slower smooth movement?

Further reading

Please refer to the following links for more information:

- This servo motor control hat is based on the PCA9685 device. The PCA9685 product data sheet (`https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf`) contains full information about operating this chip. I highly recommend referencing this.
- I also recommend looking at the SG90 servo motor data sheet (`http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf`) for information about their operation.
- The AdaFruit guide to the pan and tilt mechanism (`https://learn.adafruit.com/mini-pan-tilt-kit-assembly`) has a set of assembly instructions. They are in a slightly different order from mine but may give a different perspective if this is proving to be tricky.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11
Programming Encoders with Python

It is useful in robotics to sense the movements of motor shafts and wheels. We drove a robot along a path back in *Chapter 7*, *Drive and Turn – Moving Motors with Python*, but it's unlikely that it has stayed on course. Detecting this and traveling a specific distance is useful in creating behaviors. This chapter investigates the choice made by the sensor, as well as how to program the robot to move in a straight line and for a particular distance. We then look at how to make a specific turn. Please note that this chapter does contain math. But don't worry, you'll follow along easily.

In this chapter, you will learn about the following topics:

- Measuring the distance traveled with encoders
- Attaching encoders to the robot
- Detecting the distance traveled in Python
- Driving in a straight line
- Driving a specific distance
- Making a specific turn

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

Before we get started, make sure you have the following parts:

- The Raspberry Pi robot and the code from the previous chapter: `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter10`.
- Two slotted speed sensor encoders. Search for slotted speed sensor, Arduino speed sensor, LM393 speed sensor, or the Photo Interrupter sensor module. Include the term *3.3 V* to ensure its compatible. See the *The encoders we are using* section for images of these.
- Long male-to-female jumper cables.
- A ruler to measure the wheels' size on your robot – or better yet, calipers, if you can use them.

The code for this chapter is available on GitHub: `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter11`.

Check out the following video to see the Code in Action: `https://bit.ly/2XDFae0`

Measuring the distance traveled with encoders

Encoders are sensors that change value based on the movement of a part. They detect where the shaft is or how many times an axle has turned. These can be rotating or sensing along a straight-line track.

Sensing how far something has traveled is also known as **odometry**, and the sensors can also be called **tachometers**, or **tachos** for short. The sensors suggested in the *Technical requirements* section may also show up as **Arduino tacho** in searches.

Where machines use encoders

Our robots use electronic sensors. Cars and large commercial vehicles use electronic or mechanical sensors for speedometers and tachos.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Printers and scanners combine encoders with DC motors as an alternative to stepper motors. Sensing how much of an arc the robot has turned through is an essential component of servomechanisms, which we saw in *Chapter 10*, *Using Python to Control Servo Motors*. High-end audio or electrical test/measurement systems use these in control dials. These are self-contained modules that look like volume knobs but users can turn them indefinitely.

With this basic understanding of what encoders are, let's now look at some of their types.

Types of encoders

Figure 11.1 shows four different encoder sensors, each of which uses different mechanisms to measure movement (1–3), along with an encoder wheel and strip in panel 4:

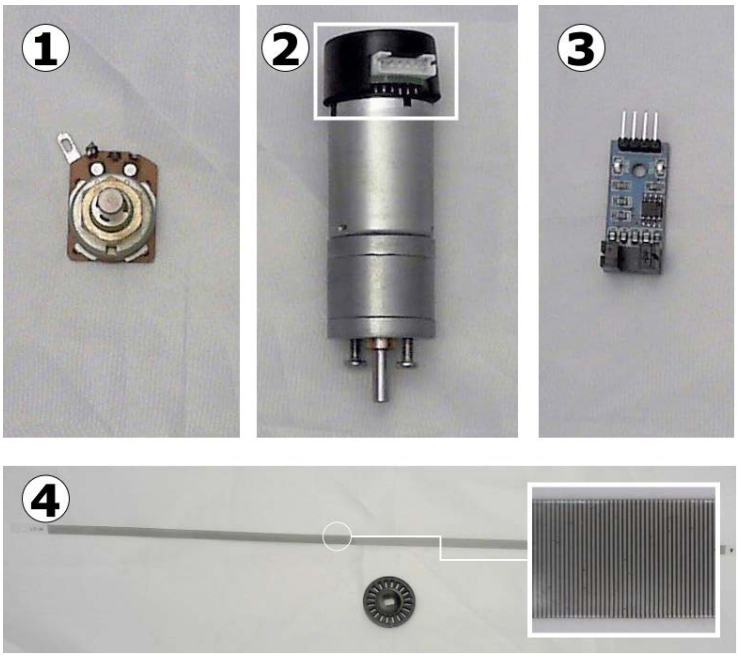


Figure 11.1 – Encoder sensors

These sensors fall into a few categories, as shown in *Figure 11.1*, and correspond to the points that follow:

1. This is a **variable resistor**. These analog devices can measure a turn but don't tend to allow continuous rotation. They have mechanical wipers on a metal or resistant track, which can wear down. This is not strictly an encoder but is handy. On the Raspberry Pi, they require analog-to-digital conversion, so they aren't suitable for this application. Servo motors also use these.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. This motor includes magnet-sensing encoders, highlighted by the white box, known as **hall-effect sensors**. Magnets on a wheel or strip pass next to the sensor, causing the sensor values to go high and low.
3. This is a standard optical sensor. Using a slot with an IR beam passing through, they sense when the beam is interrupted. Computer trackballs, printers, and robotics use these. These produce a chain of pulses. Due to the beam being interrupted, we call them **photo interrupters**, **optical encoders**, or **opto-interrupters**. We will be using this kind.
4. This shows a slotted wheel and a slotted strip for use with optical sensors. The ribbon is suitable for linear encoding and the wheel for encoding turns. They both have transparent and opaque sections. People make a variation using a light sensor and light/dark cells, but these are less common.

Having seen some of the types of encoders, let's take a closer look at how they represent speed and movement.

Encoding absolute or relative position

Relative encoders can encode a relative change in position – that we have taken a certain number of steps clockwise or anticlockwise, or forward/backward along an axis, for example. They can only measure a position relative to the last time we measured by counting the number of slots that have passed. These can also be called **incremental encoders**. They are inexpensive and straightforward in hardware. Relative encoders are limited in that they memorize the previous position to create the next and accumulate errors.

Another type is **absolute encoders**. These can encode the position along or around an axis to an exact position. Absolute encoders do not need information about the previous position but may need calibrating to determine how an encoding matches a real-world location.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The following figure shows the types in comparison:

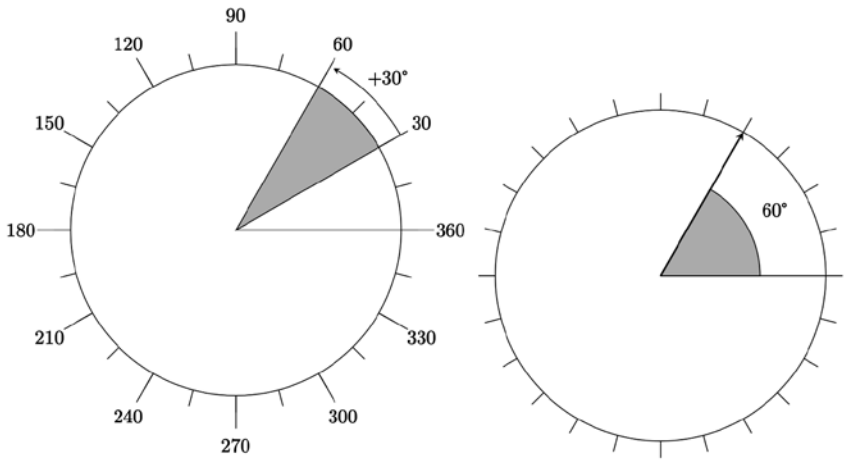


Figure 11.2 – Comparing absolute and relative sensing

The diagrams in *Figure 11.2* illustrate this difference. The circle on the left represents a movement *by* 30 degrees, from a memorized position of 30 degrees. This

works, assuming that the original memorized position is accurate. Every time it is measured, a movement is measured. The circle on the right shows a position *at* 60 degrees from a zero point. If a sensor can tell you where something is *at*, then it is absolute. If you can tell you how much it has moved *by*, it is relative.

In a crude form, a variable resistor can also be an absolute encoder, as used in servo motors to sense their position. Absolute position encoding can be done through optical or magnetic markers on a wheel or strip, allowing great precision in absolute positioning. These sensors can be bulky or expensive or require many inputs. An absolute encoder strip or wheel is also known as a scale.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Encoding direction and speed

Basic relative encoding measures how many wheel slots pass the sensor. This gives the speed and distance. By using two sensors slightly apart, you can also encode the direction. *Figure 11.3* shows how this works:

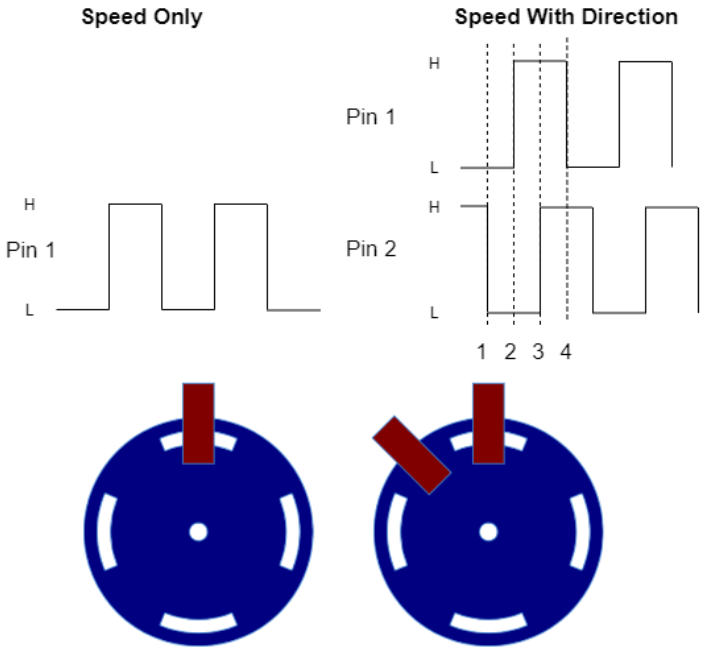


Figure 11.3 – Encoding speed and direction with multiple sensors

The system on the left encodes the speed. As the slots pass the sensors, they generate electronic pulses. Each pulse has a **rising** edge, where it goes up, and a **falling** edge, where it goes down. To count the number of pulses, we can count these edges. If you drive a motor with a direction, you can use a system with a single sensor like this, as we will do in our robot.

The wheel on the right encodes direction by adding a second sensor. The slot edges will make the sensor value change at different points in a cycle, with a sequence we've labeled **1**, **2**, **3**, **4**. The direction of the sequence indicates the direction of the wheel, along with its speed. As there are four phases, this is known as quadrature encoding.

Industrial robots use a **record-and-replay** interface. The user will hit a record button and push a robot, such as an arm, through a set of movements, then press a stop button. The user has recorded this set of movements, and they could ask the robot to replay them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

To build a robot with this record-and-replay system, or a mouse/trackball, the direction is essential information, so the additional complexity needed to encode the directions is required.

We will use the cheaper and simpler option in our robot, using a single sensor to measure relative speed only. In our code, we will assume that each wheel's speed sensor is going in the direction we drive.

The encoders we are using

We will use optical encoders in a slot shape that fits right above the encoder wheels we added in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*. These encoders have digital outputs, and we can count the pulses from them in Python to sense how far a wheel has turned. *Figure 11.4* shows two types of sensor that I recommend:

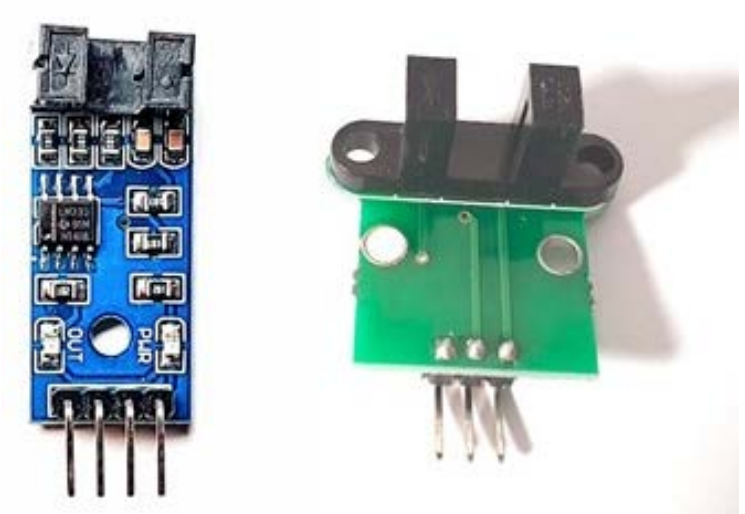


Figure 11.4 – The types of sensor we recommend

On the left is the FC-03 photo interrupter module, and on the right is the Waveshare photo interrupter module. Modules that have 3.3 V in their supply voltage specification are suitable. Using 5 V-only modules will require level shifters and additional wiring, as discussed in *Chapter 8*, *Programming Distance Sensors with Python*.

We are going to use encoder wheels that are attached to the motor shafts. These are in line with the wheels. If the encoder wheels are running at a different rate from the wheels, we need to account for this. There are conditions that they cannot account for, such as slipping, as well as wheel and tire sizes. Encoders attached to separate idler wheels give better data, but they are trickier to connect to a robot and keep in contact with the floor.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, you've seen what encoders can do, counting how much a wheel has turned to determine how far you've gone, and you've seen some of the types of encoders and which type to buy (3.3 V). You've also had a quick overview of how they work, by counting pulses. In the next section, we will build them into the robot.

Attaching encoders to the robot

Our robot is now getting quite busy, and our Raspberry Pi is above the encoder's slots. Due to the slots being under the Raspberry Pi, we should wire them in a little before returning the Raspberry Pi. After bolting in the Raspberry Pi, we wire the encoders to its GPIO, as well as the power and ground.

Figure 11.5 shows what the robot block diagram looks like after attaching the encoders:

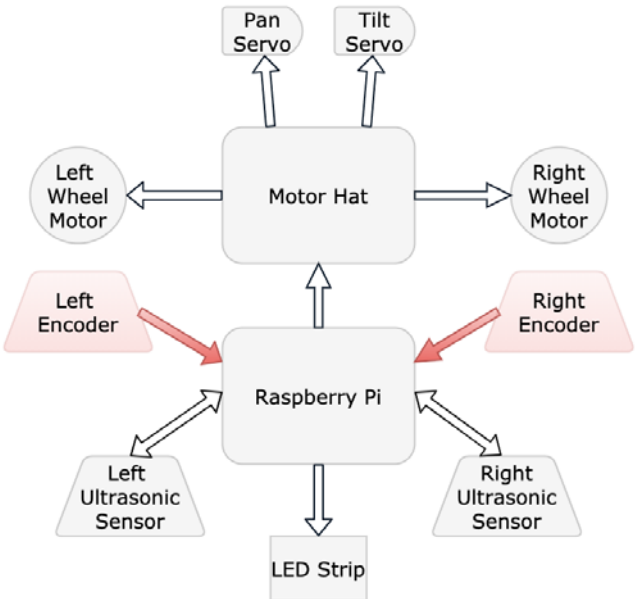


Figure 11.5 – Robot block diagram with encoders

This block diagram adds a left and right encoder, each with an arrow for the information flow connecting them to the Raspberry Pi. The highlighted elements are the new ones.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Before we start changing the robot and making it harder to see, we need to know the number of slots in the encoder wheel for later:

Figure 11.6 – Encoder wheel

My encoder wheels, shown in *Figure 11.6*, ended up having 20 slots. Ensure you use the number of spaces your encoder wheels have.

Preparing the encoders

Before we can use the encoder sensors, we need to prepare and fit them. As the encoders are going under the Raspberry Pi, we should attach the male-to-female jump wires to the sensors now. I suggest covering any electrical contacts that are sticking up under the Raspberry Pi with a little insulation tape:

Figure 11.7 – The sensors with cable connections

Notably, the cables should be plugged into the ground (**GND**), voltage (**3 V**, **Vin**, or **VCC**), and digital output (**D0**/**OUT**), as shown in *Figure 11.7*. If it is present, do not connect the analog output (**A0**) pin. If possible, the ground pin should have the darkest color, or the voltage should be the lightest color. To help keep this clear, I suggest wrapping a small strip of insulation tape around the signal line's end.

> **Important note**
>
> As these sensors' pin configurations can vary, get a reference photo of the sensor pin labels before putting it under the Raspberry Pi.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Lifting the Raspberry Pi

The encoder sensors need to go underneath the Raspberry Pi, so the Raspberry Pi needs to be gently lifted (without disrupting the wires) to accommodate them. The sequence of photos in *Figure 11.8* shows how to raise it:

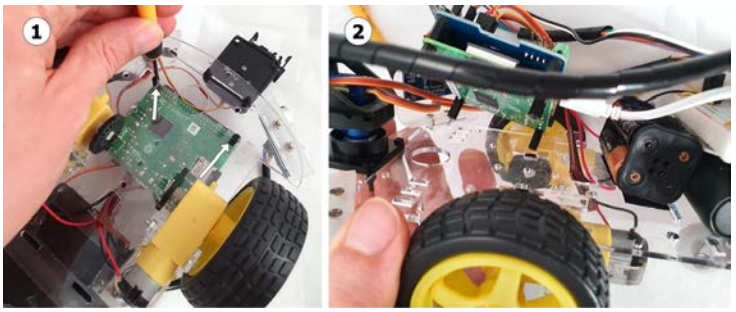


Figure 11.8 – Unscrewing and lifting off the Raspberry Pi

Refer to *Figure 11.8* and carry out the following steps:

1. You need to unscrew the bolts holding the Raspberry Pi to the chassis carefully. Put the screws aside for replacing the Raspberry Pi on the robot so that it can gently lift away.
2. Gently lift the Raspberry Pi away from the robot without disrupting the cables. The photo shows how your robot should look.

Great! Now that the Raspberry Pi is lifted, there is space for fitting the encoders under it.

Fitting the encoders onto the chassis

Now that you have wired the encoders, you can fit them to the robot chassis. As a guide, *Figure 11.9* shows a bare chassis with each of the sensor types fitted to show you where you push them in:

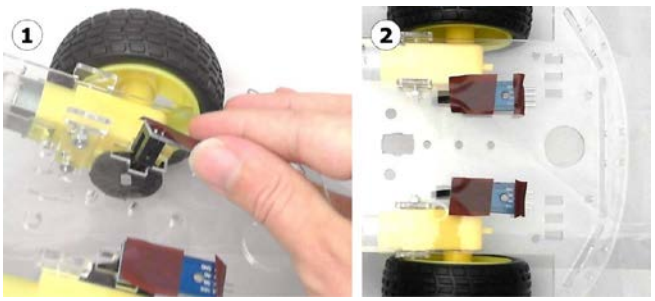


Figure 11.9 – Fitting the encoder sensors

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Refer to *Figure 11.9* and complete these steps:

1. Gently push the encoders into the slots around the encoder wheel.
2. The sensors should friction fit into the slots above the encoder wheels and stay in place.

Once these are in place, you can replace the screws to attach the Raspberry Pi to the chassis. You may need different-sized standoffs to accommodate it.

> **Important note**
> At this point, check that all your connections are back in place. The motor wires are likely to have come loose.

With the encoders attached and the Raspberry Pi back in place, you are ready to wire them in.

Wiring the encoders to the Raspberry Pi

We can now start making wire connections for the sensors using the breadboard. We'll use a circuit diagram to guide our connections.

Figure 11.10 shows the connections for these sensors:

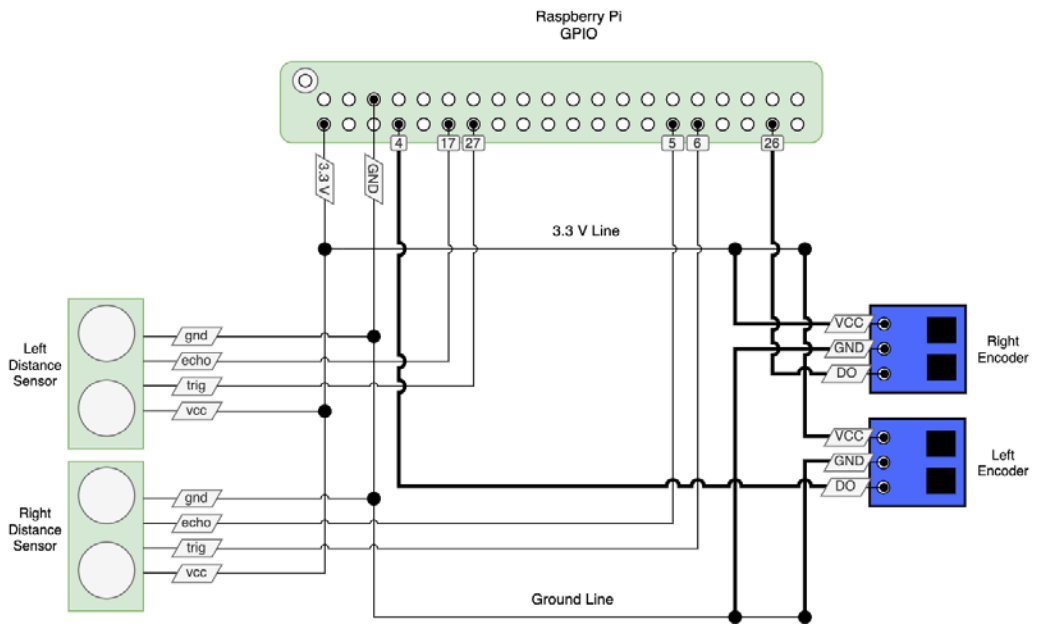


Figure 11.10 – Circuit for connecting the encoders to the Raspberry Pi

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 11.10 shows the circuit for connecting the encoders to the Raspberry Pi. This diagram continues from the circuit seen in *Chapter 8*, *Programming Distance Sensors with Python*. The new connections have thicker wire lines.

On the right of *Figure 11.10* are the right and left encoder modules. The sensor will have a **VCC**, **VIN**, **3 V**, or just **V** label. Connect this to the 3.3 V line. Find the GND pin labeled **GND**, **0V**, or just **G** and connect it to the black/blue band on the breadboard. The encoders also have a pin labeled **DO** (digital out), **SIG** (signal), or just **S**, connected to a GPIO pin for the digital signal. We connect the digital outputs from the sensors to GPIO pins 4 for the left encoder and 26 for the right encoder.

Some encoder sensors have additional connections, such as **Analog Out** (**AO**), which we will not use and will leave unconnected.

Let's perform the following steps:

1. Connect the sensor's pins to the breadboard. Pay attention to the labels on the sensor – some have a different pin ordering.
2. Connect the Raspberry Pi GPIO pins 4 and 26 to the breadboard.
3. Make the breadboard power connections using precut wires.

> **Important note**
>
> The number of wire-to-wire points on this robot makes it hard to add new connections or repair. Although beyond the scope of this book, making custom **Printed Circuit Boards** (**PCBs**) makes this thicket of cabling much neater. PCBs are also less fragile and takes up less space. Changing it does, however, come with a cost.

It is possible to shortcut the breadboard and wire the sensor into the Raspberry Pi; however, the breadboard helps distribute the power connections and groups the sensor connections.

We make this circuit in the context of a busy robot with other connections. If the cables from the sensors to the robot are not long enough, use a further set of male-to-female cables, using the same technique seen for the sonar scan in *Chapter 10*, *Using Python to Control Servo Motors*.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this section, you've learned how to connect the sensors and have made them ready to program. In the next section, we will write some code to test these sensors and then measure distances.

Detecting the distance traveled in Python

Using encoder sensor devices requires us to count pulses. In this section, we will create code to turn on the motors and count pulses for a while. The code validates that the sensors are connected correctly. We then take this code and make it part of the robot class as a behavior.

Introducing logging

So far in our Python code, we have been using `print` to output information to see what our robot is doing. This works, but prints can become overwhelming if we print everything we might want to inspect. Logging allows us to still display information, but we can control how much. By importing the `logging` module, we can take advantage of this.

First, there are logging levels, with `debug`, `info`, `warning`, and `error`. While fixing problems or initially developing, `debug` is useful – it can show everything – while `info` is used to show a little less than that. The `warning` and `error` levels are reserved only for those kinds of problems, so you can filter down to only this level when you are reasonably confident with some code. The `logging.basicConfig` function allows us to configure a logging level and other logging parameters; you need to configure logging to enable `info` and `debug` logging.

The other thing you can do with logging is to have differently named loggers using the `logging.getLogger` function. This lets you set different levels for different modules. Using named loggers helps to enable `debug` from a library module you are using while sticking to `info` on the main module.

In this chapter's code, we will start using logging and controlling logging levels to get more detail on what an example is doing, turning the parts on and off at will. When we introduce the PID controller later, this will become very useful indeed. We can use this logging in the next section to show a sensor pulse count.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Simple counting

This code counts the number of cycles up and down on each wheel's signal pin, printing the count as we test the sensors. Running this verifies that our sensor connections are working, letting us troubleshoot and move on. It also lays code foundations for tracking wheel movements. We run our motors for about 1 second. Note that this code assumes you are starting with the code from *Chapter 10*, *Using Python to Control Servo Motors*:

> **Important note**
>
> You can find the following code at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter11/test_encoders.py`.

1. Let's make a file called `test_encoders.py`, starting with the usual robot classes, `import` and `time`, and adding logging:

```
from robot import Robot
import time
import logging
...
```

2. Next, we add an import for a GPIO Zero input device. We can use the pin it sets up to count our pulses:

```
...
from gpiozero import DigitalInputDevice
logger = logging.getLogger("test_encoders")
...
```

   We've also set up a named logger for our system matching the filename.

3. The encoders generate pulses; we want to count them and track their state. We use more than one of them. Creating a class from the outset seems like the right strategy. From here, we can pass our I/O pin number to the constructor. The first thing it needs to do is to set up a pulse counter:

```
...
class EncoderCounter(object):
    def __init__(self, pin_number):
        self.pulse_count = 0
        ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Still in the constructor, we need to set up the device and how we count pulses with it. The device has a `.pin` object, which we set up using the pin number. `.pin` has a `when_changed` event, which we can drop our handler into to be called every time the pin changes. The pin changes from up to down (rising and falling) for every slot:

```
        ...
        self.device = DigitalInputDevice(pin=pin_number)
        self.device.pin.when_changed = self.when_changed
    ...
```

5. We need to define a `when_changed` method for our class to add one to `pulse_count`. This method must be as small/quick as possible, as GPIO Zero calls it in the background for every pulse change. It takes a `time_ticks` parameter and a `state` parameter. We will not use `time_ticks`, so mark this with an underscore:

```
    ...
    def when_changed(self, _, state):
        self.pulse_count += 1
...
```

6. We can set up our `robot` object and create an `EncoderCounter` for each side's sensor. We connected our devices to pins `4` and `26`:

```
...
bot = Robot()
left_encoder = EncoderCounter(4)
right_encoder = EncoderCounter(26)
...
```

7. To display values, instead of just using `sleep`, we loop, checking against an end time. Before we log anything, `logging.basicConfig` sets logging parameters. We start the motors and go into the main loop:

```
...
stop_at_time = time.time() + 1

logging.basicConfig(level=logging.INFO)
bot.set_left(90)
bot.set_right(90)

while time.time() < stop_at_time:
    ...
```

In this loop, we log the readings on both sensors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Since tight loops can cause things to break (such as GPIO Zero calling our code from a sensor thread), it should sleep a little too:

```
    ...
    logger.info(f"Left: {left_encoder.pulse_count} Right:
{right_encoder.pulse_count}")
    time.sleep(0.05)
```

   When the loop ends, the program is done, so our robot automatically stops. Notice we have used an f-string (format string), as we saw in *Chapter 8*, *Programming Distance Sensors with Python*. The `f` prefix lets us format variables into a string.

You can send this code to the robot and run it. You can now see the robot veering through the encoder's values. The output should look a little like this:

```
pi@myrobot:~ $ python3 test_encoders.py
INFO:test_encoders:Left: 0 Right: 0
INFO:test_encoders:Left: 0 Right: 1
INFO:test_encoders:Left: 2 Right: 2
INFO:test_encoders:Left: 3 Right: 4
INFO:test_encoders:Left: 5 Right: 7
INFO:test_encoders:Left: 8 Right: 10
INFO:test_encoders:Left: 10 Right: 14
...
INFO:test_encoders:Left: 56 Right: 74
```

The encoders are counting, and it shows that the robot moved less on the left wheel and more on the right wheel, and veered left. The `INFO:test_encoders:` part is introduced by the logging, showing the logging level and the logger name. The distances are in encoder *ticks*, a tick being each counted event.

You've now tried out this code, but refer to the troubleshooting section if you have any problems.

Troubleshooting

If you find problems when running this, try the following steps:

- Ensure you started from the *Chapter 10*, *Using Python to Control Servo Motors*, code. If you downloaded code from the current chapter, you will likely see GPIO conflicts with the code already set up for the encoders.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- If the encoder values stay at zero, turn off the Raspberry Pi, then go back and carefully check your wiring and pin number usage.
- Check your wiring – if anything is hot, immediately disconnect the power and verify your wiring.

You have tested the encoders on the robot, seeing feedback on your screen as they moved. This demonstrates that they are ready to be used in more interesting robot behaviors after adding them to the `Robot` object.

Adding encoders to the Robot object

To use this sensor in other code or behaviors, we should move it into the `Robot` object. We can then import our code into the `Robot` object and set up the two sides with the correct pins. You'll also need to add some cleanup code for the handlers.

Extracting the class

We've already made the `EncoderCounter` class, which you can copy from `test_encoders.py` to the `encoder_counter.py` (`https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter11/encoder_counter.py`) file. This code needs the import for `DigitalInputDevice`, the same constructor, and the `when_changed` handler:

1. Let's start by adding the imports and class declaration. The `EncoderCounter` class starts the same way as the last section:

```
from gpiozero import DigitalInputDevice

class EncoderCounter:
    def __init__(self, pin_number):
        self.pulse_count = 0
```

2. I'm adding a `direction` member to account for reversing:

```
        self.direction = 1
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The constructor (`__init__`) is finished by setting up the device and assigning a `when_changed` handler:

```
        self.device = DigitalInputDevice(pin=pin_number)
        self.device.pin.when_changed = self.when_changed
```

4. Our `when_changed` handler should add the direction instead of 1, so it can count up or down:

```
    def when_changed(self, time_ticks, state):
        self.pulse_count += self.direction
```

5. We should also have a method to set this direction, so we can assert to validate our setting, which throws an exception if it doesn't meet the condition with the given text – a cheap but brutal way of ensuring input values make sense:

```
    def set_direction(self, direction):
        """This should be -1 or 1."""
        assert abs(direction)==1, "Direction %s should be
1 or -1" % direction
        self.direction = direction
```

6. A reset method means we can handle restarting counters between movements:

```
    def reset(self):
        self.pulse_count = 0
```

7. For cleanup, we need a way to stop the counters so that they don't call the handler again:

```
    def stop(self):
        self.device.close()
```

With the encoder library ready, we can use this in our code. The library means we can reuse our encoder counter in different places, and also that we can substitute a different device with similar properties. To make it available to many behaviors, it will be handy to import it into the robot library.

Adding the device to the Robot object

We've used our `Robot` object as the main interface between code handling the hardware and the behavior code.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will modify the `robot.py` code from *Chapter 10*, *Using Python to Control Servo Motors* (`https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter10/robot.py`) to add the sensors:

1. Start by importing `EncoderCounter`:

```
...
import leds_led_shim
from servos import Servos
from encoder_counter import EncoderCounter
...
```

2. In the `__init__` constructor method, we need to set up left and right encoders. I did this just after the distance sensors:

```
        ...
        # Setup The Distance Sensors
self.left_distance_sensor = DistanceSensor(echo=17,
trigger=27, queue_len=2)
        self.right_distance_sensor =
DistanceSensor(echo=5, trigger=6, queue_len=2)

        # Setup the Encoders
        self.left_encoder = EncoderCounter(4)
        self.right_encoder = EncoderCounter(26)
        ...
```

3. To make sure that the code cleans up encoder handlers when our `Robot` object has stopped, we call the encoder's `stop` methods in the `stop_all` method:

```
        ...
        # Clear the display
        self.leds.clear()
        self.leds.show()

        # Clear any sensor handlers
        self.left_encoder.stop()
        self.right_encoder.stop()
        ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The finished code for `robot.py` with encoders is on GitHub (`https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter11/robot.py`). We can now use this to build a behavior to measure the distance in millimeters. To do so, we'll understand the relationship between encoder ticks and the distance moved in millimeters.

Turning ticks into millimeters

To calculate real distances, we need the sizes of the wheels. We cannot account for slipping, but we can find out how much a wheel has turned, which is the same as the encoders. Using the wheel's diameter, we can calculate how far it has turned. Using a ruler or caliper, measure the diameter across the wheel, as shown in *Figure 11.11*:

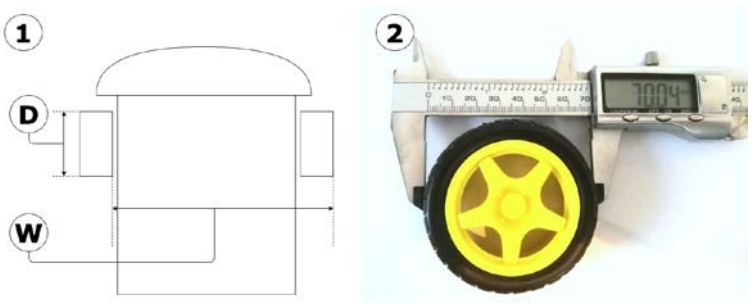


Figure 11.11 – Measuring the wheel

The measurements needed are shown in *Figure 11.11*:

1. You will need to measure the wheel's diameter, marked **D** in this figure, and the distance between the wheels, **W**. The distance **W** is equivalent to the width from midpoint to midpoint of the two motor-driven wheels on the robot. It is easier to measure, as shown here, the right side of one wheel, all the way across to the right side of the other wheel – which will be the same as midpoint to midpoint. Mine came to about 130 mm.
2. You can measure **D** with calipers, as shown here, by fitting them around the widest part of the wheel. My wheel came to 70 mm, to the nearest mm.

We know how many slots are on the encoders, and we expect two ticks (the rising and falling) per slot, so we can take the number of slots times 2, which is the number of ticks per whole-turn of the wheel – in my case, this is 40.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The number pi, or π, is the ratio of the diameter to the circumference of the wheel. To get the circumference, we multiply the diameter by pi, giving us πD, where D is the diameter. We can divide pi by the number of total ticks per revolution, and then when we multiply this by the number of ticks counted, T, and then the diameter, D, and we get a number for the distance, d, that the wheel has traveled:

$$d = \frac{\pi}{40} \times D \times T$$

So, how do we turn this into code? Refer to the following steps:

1. Make a new file called `test_distance_travelled.py`. At the top of the file, we need to import `math` for the calculations, the `Robot` object, and `time`:

```
from robot import Robot
import time
import math
import logging
logger = logging.getLogger("test_distance_travelled")
...
```

2. Next, we define our constants – the wheel's diameter and the number of ticks per revolution. Please use the values you obtained, not the ones that I have shown here:

```
...
wheel_diameter_mm = 70.0
ticks_per_revolution = 40.0
...
```

3. Create a function to convert the ticks counted into a distance. It's converted into integers since fractions of a millimeter are just not appropriate for this measurement. Since part of the conversion doesn't change, we make that a constant, too:

```
...
ticks_to_mm_const = (math.pi / ticks_per_revolution) *
wheel_diameter_mm

def ticks_to_mm(ticks):
    return int(ticks_to_mm_const * ticks)
...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Next, we define our robot, set up a stop time, and start the motors:

```
...
bot = Robot()
stop_at_time = time.time() + 1

logging.basicConfig(level=logging.INFO)
bot.set_left(90)
bot.set_right(90)
...
```

5. In the loop, we display the distance by calling `ticks_to_mm` on the pulse counts:

```
...
while time.time() < stop_at_time:
    logger.info("Left: {} Right: {}".format(
        ticks_to_mm(bot.left_encoder.pulse_count),
        ticks_to_mm(bot.right_encoder.pulse_count)))
time.sleep(0.05)
```

When uploaded to the robot and run, the output looks like this:

```
pi@myrobot:~ $ python3 test_distance_travelled.py
INFO:test_distance_travelled:Left: 0 Right: 0
INFO:test_distance_travelled:Left: 5 Right: 0
INFO:test_distance_travelled:Left: 16 Right: 10
INFO:test_distance_travelled:Left: 32 Right: 21
...
...
INFO:test_distance_travelled:Left: 368 Right: 384
INFO:test_distance_travelled:Left: 395 Right: 417
```

This output has shown a clear difference between the travel on the left and the right motors. The right motor is moving slightly quicker than the left. This difference accumulates, making the robot turn further. So, in the next section, let's use this information to straighten things up.

Driving in a straight line

By now, you have seen differences in the outputs – that is, a veer. In only 400 mm, my left side is around 20 mm behind the right, an error that is climbing. Depending on your motors, your robot may have some veer too. It is rare for a robot to have driven perfectly straight. We use the sensors to correct this.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Tip**
> This behavior works better on wooden flooring or MDF boards, and poorly on carpet.

This correction is still dead reckoning; slipping on surfaces or incorrect measurements can still set this off course. How can we use motors and encoders to correct our course and drive in a straight line?

Correcting veer with a PID

A behavior to self-correct steering and drive in a straight line needs to vary motor speeds until the wheels have turned the same amount. If the wheels turn the same amount soon enough, then they will account for major course deviations.

Our robot will use the encoder sensor to measure how much each wheel has turned. We can then consider the difference between these to adjust the motor control and try to keep the motors at the same speed.

A trick with this is working out how the difference in measurements relates to adjusting motor speeds. This leads us to look at a PID system designed to map errors into adjustment and output values.

Driving in a straight line needs a closed **feedback** loop. *Figure 11.12* shows how this loop works:

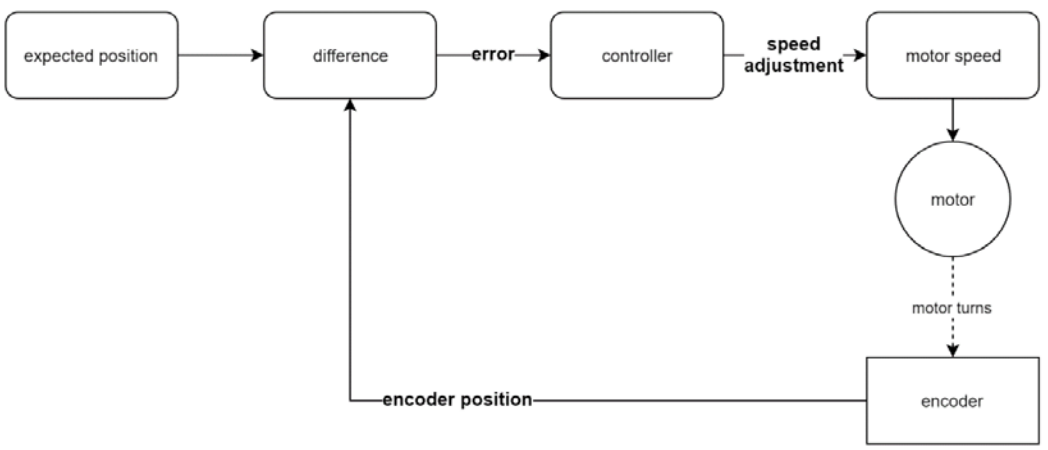


Figure 11.12 – Closed-loop control of a motor's speed

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We start with an *expected position* or **set point**. The *encoder position* gives feedback data from the real world. We get a difference between the setpoint and the encoder position, which we call the *error*. The code feeds this into a *controller*, which generates a *speed adjustment*. The system will then apply that adjustment to the *motor speed*, making the motor turn more or less, changing the encoder's feedback.

To go straight, we take the left motor's value and subtract the right motor to get an encoder difference. Our *expected position* is 0. Our *error* is then the difference between the encoders. We can then adjust the speeds using the controller.

We use a **PID** controller to adjust the speed of the motors; this has three components:

- **Proportional** (**P**): The error value multiplied by a constant. This corrects for immediate errors.
- **Integral** (**I**): The sum of the error values so far, multiplied by a constant. This corrects for continuing errors.
- **Derivative** (**D**): This takes the difference between the last error value and now and multiplies by a constant. This is to push back a little against sudden changes.

By manipulating the constants, we *tune* how much each factor influences the outcome of the controller. We won't be using the derivative component for our behaviors, which is equivalent to having its constant set to zero.

The integral can give the robot some self-tuning, but it needs to have a very small constant, as high values can make the robot start to wobble instead. We will add the adjustment onto one motor and subtract it from the other.

The right motor speed is as follows:

```
...
integral_sum = integral_sum  + error
right_motor_speed = speed + (error * proportional_constant) +
(integral_sum * integral_constant)
...
```

We need an unused motor speed capacity to be able to speed up a bit. If the speed is too close to 100%, we get clipping. An integral behavior with clipping can make the robot behave quite strangely, so watch out for clipping at 100%!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Tip**
> Leave headroom for PID adjustments – use no more than 80% of motor speed.

Now that we have some idea how a PID controller works, let's build one in code.

Creating a Python PID controller object

The PID controller code is a fundamental robotics building block for making straight lines and we'll use it again in later camera-driven chapters. You will use the basic concepts here in many robotic systems:

1. We use this in a few places, so let's make a simplified PID control object. I put this in a file named `pid_controller.py`. Note that this is only a **proportional-integral** (**PI**) controller; we can add a differential later if needed. Here is the class and its constructor:

```
import logging
logger = logging.getLogger("pid_controller")

class PIController:
    def __init__(self, proportional_constant=0, integral_
constant=0):
        self.proportional_constant = proportional_
constant
        self.integral_constant = integral_constant

        # Running sums
        self.integral_sum = 0
...
```

   The constructor takes the constants. I've preloaded these with zero, so you can isolate the components. The class stores these values. Then, we set up a variable to store the integral sum, which grows with time. It is not unusual to abbreviate `proportional_constant` as `pK` and `integral_constant` as `iK`. You can do so if you wish. I've used the longer names in the code examples to make it easier to read.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. The following code handles the values for the two components. Handling the integral has the effect of increasing the integral sum:

```
...
    def handle_proportional(self, error):
        return self.proportional_constant * error

    def handle_integral(self, error):
        self.integral_sum += error
        return self.integral_constant * self.integral_sum
...
```

3. The following bit of code handles the error to generate the adjustment:

```
...
    def get_value(self, error):
        p = self.handle_proportional(error)
        i = self.handle_integral(error)
        logger.debug(f"P: {p}, I: {i:.2f}")
        return p + i
```

   I've left the proportional and integral parts available here as `p` and `i`; since we log these values, we can configure logging to show them when debugging and tuning the controller.

With the PI code in place, we are ready to make a robot that can combine errors with previous values, scaling them to make them useful in the context of some movement. We will use this PID controller for our straight line adjusting in the next section.

Writing code to go in a straight line

I called this `straight_line_drive.py` (`https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/blob/master/chapter11/straight_line_drive.py`):

1. Let's import the `Robot` object, `time`, and our new PI controller. We'll set up logging to get debug from the PID controller. You can tune it back to `INFO` or take that line out if it's too much:

```
from robot import Robot
from pid_controller import PIController
import time
import logging

logger = logging.getLogger("straight_line ")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
logging.basicConfig(level=logging.INFO)
logging.getLogger("pid_controller").setLevel(logging.
DEBUG)
```

2. Set up the `Robot` object too, and set up a slightly longer `stop_at_time` value so that our robot drives a bit further:

```
bot = Robot()
stop_at_time = time.time() + 15
...
```

3. Start with a master speed value of `80`, and set both motors to this:

```
...
speed = 80
bot.set_left(speed)
bot.set_right(speed)
...
```

4. Before going into our main loop, set up the controller. You may need to tune these constants. Note how small the integral constant is:

```
...
pid = PIController(proportional_constant=5, integral_
constant=0.3)
...
```

5. In the loop, we `sleep` a little so that our encoders have something to measure. It's also usually a bad idea to have a "tight" loop that doesn't use `sleep` to give other things a chance to run. Get the encoder values and compute the error:

```
...
while time.time() < stop_at_time:
    time.sleep(0.01)
    # Calculate the error
    left = bot.left_encoder.pulse_count
    right = bot.right_encoder.pulse_count
    error = left - right
    ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. That error needs to be handled by the controller and used to make `right_speed`:

```
    ...
    # Get the speed
    adjustment = pid.get_value(error)
    right_speed = int(speed + adjustment)
    left_speed = int(speed - adjustment)
    ...
```

7. We can then log debug information here. Notice we have two levels: debug for the error and adjustment and info for the speeds. With the current config set to `INFO`, we won't see the debug without modifying it:

```
    ...
    logger.debug(f"error: {error} adjustment:
{adjustment:.2f}")
    logger.info(f"left: {left} right: {right}, left_
speed: {left_speed} right_speed: {right_speed}")
    ...
```

8. We then set the motor speeds to the adjusted values and finish the loop:

```
    ...
    bot.set_left(left_speed)
    bot.set_right(right_speed)
```

When we run this, the robot should be following a fairly straight course. It may start unstable, but should hone in on a constant adjustment:

```
pi@myrobot:~ $ python3 straight_line_drive.py
DEBUG:pid_controller:P: 0, I: 0.00
INFO:straight_line:left: 3 right: 3, left_speed: 80 right_
speed: 80
DEBUG:pid_controller:P: 0, I: 0.00
INFO:straight_line:left: 5 right: 5, left_speed: 80 right_
speed: 80
DEBUG:pid_controller:P: -4, I: -0.20
INFO:straight_line:left: 5 right: 6, left_speed: 84 right_
speed: 75
DEBUG:pid_controller:P: 0, I: -0.20
...
INFO:straight_line:left: 13 right: 15, left_speed: 89 right_
speed: 71
DEBUG:pid_controller:P: -8, I: -1.40
INFO:straight_line:left: 15 right: 17, left_speed: 89 right_
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
speed: 70
DEBUG:pid_controller:P: -8, I: -1.80
INFO:straight_line:left: 17 right: 19, left_speed: 89 right_
speed: 70
DEBUG:pid_controller:P: -8, I: -2.20
INFO:straight_line:left: 19 right: 21, left_speed: 90 right_
speed: 69
...
DEBUG:pid_controller:P: 0, I: 0.60
INFO:straight_line:left: 217 right: 217, left_speed: 79 right_
speed: 80
DEBUG:pid_controller:P: 0, I: 0.60
INFO:straight_line:left: 219 right: 219, left_speed: 79 right_
speed: 80
DEBUG:pid_controller:P: 0, I: 0.60
INFO:straight_line:left: 221 right: 221, left_speed: 79 right_
speed: 80
DEBUG:pid_controller:P: 0, I: 0.60
INFO:straight_line:left: 223 right: 223, left_speed: 79 right_
speed: 80
```

The robot starts with no error as the motors engage, but the right goes faster. At 13 ticks, the controller pulls the adjustment pretty high. Notice how `P` jumps, but `I` settles for a constant value after a while, which will keep the robot straight.

Tuning of the `P` and `I` constants and the loop timing may result in earlier corrections – the initial encoder values are too small to be useful.

Note that this may still end up off course; it accounts for reducing the veer but can adjust too late to stop a small *S* shape or other error. It is, however, much straighter than driving without. Adjusting the PID can help with this.

Troubleshooting this behavior

Here are a few steps to take if the robot is wobbling or doesn't manage to travel in a straight line:

- If the robot takes too long to compensate, increase the proportional component.
- If the robot overshoots massively (that is, it swerves one way, then the other), reduce the size of both the proportional and integral PID components.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- If the robot is making increasing wobbles, the integral is too high, and the right speed may be going above 100. Bring down the integral component, and perhaps the requested speed.
- You can set the `straight_line` logger or `basicConfig` to debug to see the error value too.

With the straight line working and driving, you have now corrected veer problems and differences between the sides. You can now build on this; let's take a known distance and drive to it, then stop.

Driving a specific distance

For driving a specific distance, we use the PI controller again and incorporate the distance measurements into our encoder object. We calculate how many ticks we want the left wheel to have turned for a given distance, and then use this instead of a timeout component.

Refactoring unit conversions into the EncoderCounter class

We want the conversions for our encoders in the `EncoderCounter` class to use them in these behaviors. Refactoring is the process of moving code or improving code while retaining its functionality. In this case, converting distances is one of the purposes of using encoders, so it makes sense to move this code in there:

1. Open up your `encoder_counter.py` class. First, we need the `math` import:

```
from gpiozero import DigitalInputDevice
import math
...
```

2. At the top of the class, add `ticks_to_mm_const` as a class variable (not an instance variable) to use it without any instances of the class. Set this to none initially so that we can calculate it:

```
...
class EncoderCounter:
    ticks_to_mm_const = None # you must set this up
before using distance methods
     ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. In our class, we want to retrieve the distance the wheel has traveled directly from the encoder, in mm. Add this to the end of the file:

```
    ...
    def distance_in_mm(self):
        return int(self.pulse_count * EncoderCounter.
ticks_to_mm_const)
    ...
```

   This code assumes that all encoders have the same diameter wheels and the same number of slots. That is why we pull `ticks_to_mm_const` from the class and not `self` (the instance).

4. We also want to calculate the opposite: the number of ticks from a distance in mm. To do that, divide the distance in mm by the same constant we multiplied by. This is set to `staticmethod` so that it does not require later code to use an instance:

```
    ...
    @staticmethod
    def mm_to_ticks(mm):
        return mm / EncoderCounter.ticks_to_mm_const
    ...
```

5. Add a way to set the constants in the file (for different robot configurations):

```
    ...
    @staticmethod
 def set_constants(wheel_diameter_mm, ticks_per_
revolution):
     EncoderCounter.ticks_to_mm_const = (math.pi / ticks_
per_revolution) * wheel_diameter_mm
    ...
```

When you have saved this, `EncoderCounter` can now convert between distance and encoder ticks. We now need to set up the wheel diameters for your particular robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Setting the constants

So far, we can use our robot metrics in our behaviors. Now, we want the `Robot` object to store our measurements and register them with the encoders. We can do this in two simple steps:

1. In `robot.py`, just before the constructor, specify some of these numbers:

```
...
class Robot:
    wheel_diameter_mm = 70.0
    ticks_per_revolution = 40.0
    wheel_distance_mm = 140.0
    def __init__(self, motorhat_addr=0x6f):
        ...
```

2. Register these with the encoders:

```
        ...
        # Setup the Encoders
        EncoderCounter.set_constants(self.wheel_diameter_
mm, self.ticks_per_revolution)
        self.left_encoder = EncoderCounter(4)
        self.right_encoder = EncoderCounter(26)
        ....
```

With the constants ready, we've primed our encoders to measure distance. We can use this to make a behavior to drive a distance.

Creating the drive distance behavior

I'll put this code into `drive_distance.py`:

1. Start by importing `EncoderCounter` to use its metrics, `PIController`, and the `Robot` object, and set up a logger:

```
from robot import Robot, EncoderCounter
from pid_controller import PIController
import time
import logging
logger = logging.getLogger("drive_distance")
...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Define the `drive_distance` function, which takes a robot instance, a distance in ticks, and an optional speed defaulting to 80. We start by making a primary and secondary motor and controller decision:

```
...
def drive_distance(bot, distance, speed=80):
    # Use left as "primary" motor; the right is keeping
up
    set_primary = bot.set_left
    primary_encoder = bot.left_encoder
    set_secondary = bot.set_right
    secondary_encoder = bot.right_encoder
    ...
```

   Note that we store the `set_left` and `set_right` functions in variables – we can just call the variables like functions.

3. We now have a well-defined primary and secondary motor. Set up a `PIController` and start the two motors:

```
    ...
    controller = PIController(proportional_constant=5,
integral_constant=0.3)

    # start the motors and start the loop
    set_primary(speed)
    set_secondary(speed)
    ...
```

4. Now, we are in the driving distance loop. We should continue the loop until both encoders reach the right distance. We need to sleep before the rest of the loop so that we have some data for our calculations:

```
    ...
    while primary_encoder.pulse_count < distance or
secondary_encoder.pulse_count < distance:
        time.sleep(0.01)
        ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Get the error and feed it into the controller:

```
        ...
        # How far off are we?
        error = primary_encoder.pulse_count - secondary_
encoder.pulse_count
        adjustment = controller.get_value(error)
        ...
```

6. We can send this to the motors and debug the data too. Because the adjustment is a non-integer, we allow two decimal places by using {:.2f}:

```
        ...
        # How fast should the motor move to get there?
        set_primary(int(speed - adjustment))
        set_secondary(int(speed + adjustment))
        # Some debug
        logger.debug(f"Encoders: primary: {primary_
encoder.pulse_count}, secondary: {secondary_encoder.
pulse_count},"
                     f"e:{error} adjustment:
{adjustment:.2f}")
        logger.info(f"Distances: primary: {primary_
encoder.distance_in_mm()} mm, secondary: {econdary_
encoder.distance_in_mm()} mm")
...
```

7. Set up the robot, let it calculate how far you want it to go, and get it moving:

```
...
logging.basicConfig(level=logging.INFO)
bot = Robot()
distance_to_drive = 1000 # in mm - this is a meter
distance_in_ticks = EncoderCounter.mm_to_ticks(distance_
to_drive)
drive_distance(bot, distance_in_ticks)
```

8. We let the robot cleanup (`atexit`) stop the motors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

When you run this, the robot drives a meter and stops. My robot, when stopping, looked like this:

```
INFO:drive_distance:Distances: primary: 997 mm, secondary: 991
mm
INFO:drive_distance:Distances: primary: 1002 mm, secondary:
1002 mm
```

There is a 2 mm overshoot, which it can lose in rounding values and detection time. We can't make partial ticks.

You have now seen how to make the robot drive a specific distance (or pretty close to it) while trying to stay in a straight line. You've combined the measuring and the PID adjustment tools that you've built throughout this chapter. But what if we want to make turns and measure those? We'll cover this in the next section.

Making a specific turn

The next task we can use our encoders for is to make a specific turn. When turning a robot, each wheel is going through an arc. *Figure 11.13* illustrates this:

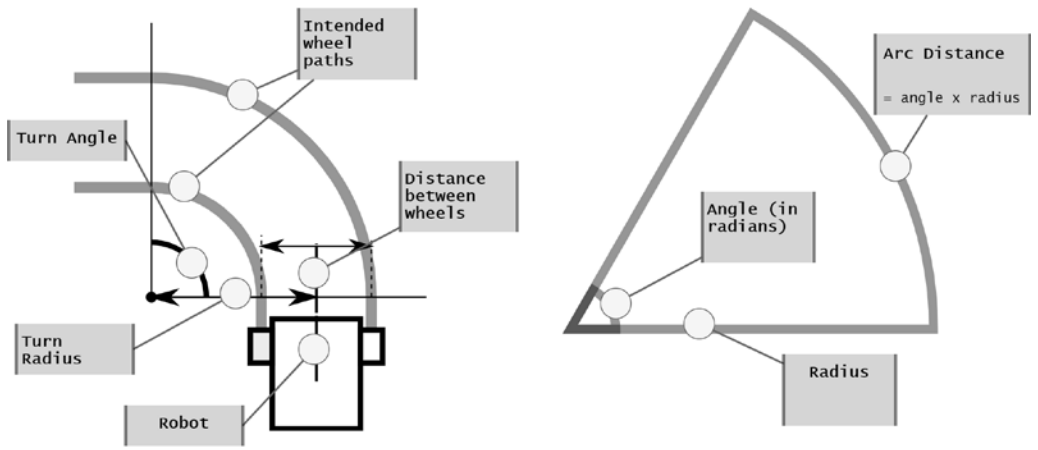


Figure 11.13 – Illustrating wheel movement when turning through an arc

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The inner wheel drives a shorter distance than the outer wheel, and from the basics of differential steering, this is how we make the turn. To make an exact turn, we need to calculate these two distances or the ratio between them. *Figure 11.14* shows how the wheels and the turn relate to each other:

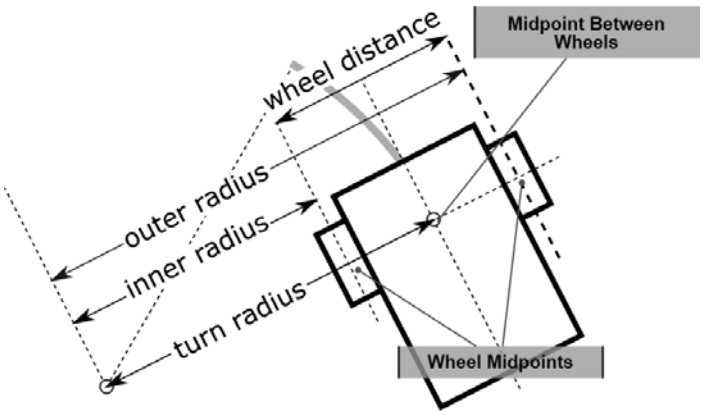


Figure 11.14 – Relating wheels to turn radiuses

If we consider the turn radius as setting where the middle of the robot is, an inner wheel's turn radius is the *difference* between the turn radius and half the distance between the wheels:

$$inner_wheel_turn_radius = turn_radius - \frac{wheel_distance}{2}$$

The outer wheel's turn radius is the turn radius *added* to half the distance:

$$outer_wheel_turn_radius = turn_radius + \frac{wheel_distance}{2}$$

We convert our angle to turn into *radians*, and we can then multiply this angle by each wheel radius to get the distances that each wheel needs to move through:

$$outer_wheel_turn_distance = angle_in_radians * outer_wheel_turn_radius$$

$$inner_wheel_turn_distance = angle_in_radians * inner_wheel_turn_radius$$

Python has math functions to convert degrees into radians.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's turn these functions into some code, demonstrating it by attempting to drive in a square, making measured 90-degree turns:

1. Start with a copy of `drive_distance.py` and call it `drive_square.py`. Add the `math` import, like so:

```
from robot import Robot, EncoderCounter
from pid_controller import PIController
import time
import math
import logging
logger = logging.getLogger("drive_square")
...
```

2. We can modify the end of this file to state what we want to do. It can help to name functions that you plan to have, and then implement them to fit. We make it a bit smaller than a meter, too. For a radius to test with, I've added 100 mm to the robot's wheel distance. Anything less than the wheel distance and the center of the turn is between the wheels instead of outside of them:

```
...
bot = Robot()

distance_to_drive = 300 # in mm
distance_in_ticks = EncoderCounter.mm_to_ticks(distance_
to_drive)
radius = bot.wheel_distance_mm + 100 # in mm
radius_in_ticks = EncoderCounter.mm_to_ticks(radius)
...
```

3. Since we are driving in a square, we want to drive four times. For straight lines, drive each wheel the same distance, then make 90-degree arcs of our radius. I've reduced the speed for the arc so that there is less of a slipping problem:

```
...
for n in range(4):
    drive_distances(bot, distance_in_ticks, distance_in_
ticks)
    drive_arc(bot, 90, radius_in_ticks, speed=50)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Let's go back up in the file to upgrade our method for driving a distance to one distance to driving two distances, one for each wheel. I've renamed the `drive_distance` function to `drive_distances`:

```
...
def drive_distances(bot, left_distance, right_distance,
speed=80):
    ...
```

5. Depending on the angle we want to turn, either motor could be the outer motor and driving a longer distance. Since there is an upper limit to speed, we choose our primary and secondary motors based on the longer distance. Swap the code that set up the primary/secondary with this:

```
    ...
    # We always want the "primary" to be the longest
distance, therefore the faster motor
    if abs(left_distance) >= abs(right_distance):
        logger.info("Left is primary")
        set_primary = bot.set_left
        primary_encoder = bot.left_encoder
        set_secondary = bot.set_right
        secondary_encoder = bot.right_encoder
        primary_distance = left_distance
        secondary_distance = right_distance
    else:
        logger.info("right is primary")
        set_primary = bot.set_right
        primary_encoder = bot.right_encoder
        set_secondary = bot.set_left
        secondary_encoder = bot.left_encoder
        primary_distance = right_distance
        secondary_distance = left_distance
    primary_to_secondary_ratio = secondary_distance /
primary_distance
    secondary_speed = speed * primary_to_secondary_ratio
    logger.debug("Targets - primary: %d, secondary: %d,
ratio: %.2f" % (primary_distance, secondary_distance,
primary_to_secondary_ratio))
    ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The encoders and motors are as they were in the preceding code. However, the code uses `abs`, the absolute value, to decide, because a longer distance in reverse should *still* be the primary motor. So, to determine how far the secondary wheel should go, we compute a ratio – to multiply with speed now, and later the primary encoder output.

6. Since we are using this method more than once, reset the encoder counts. I put this in before setting up `PIController`:

```
    ...
    primary_encoder.reset()
    secondary_encoder.reset()

    controller = PIController(proportional_constant=5,
integral_constant=0.2)
    ...
```

7. Since we can be going in either direction, set the encoder direction. Python has a `copysign` method to determine the sign of a value. Then, start the motors:

```
...
    # Ensure that the encoder knows which way it is going
    primary_encoder.set_direction(math.copysign(1,
speed))
    secondary_encoder.set_direction(math.copysign(1,
secondary_speed))

    # start the motors, and start the loop
    set_primary(speed)
    set_secondary(int(secondary_speed))
...
```

8. When we start this loop, we again need to be aware that one or both motors could be going backward. We use `abs` again to take off the sign:

```
...
    while abs(primary_encoder.pulse_count) < abs(primary_
distance) or abs(secondary_encoder.pulse_count) <
abs(secondary_distance):
        time.sleep(0.01)
...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Calculating the error for the secondary depends on the ratio between the two distances:

```
...
        # How far off are we?
        secondary_target = primary_encoder.pulse_count *
primary_to_secondary_ratio
        error = secondary_target - secondary_encoder.
pulse_count
        adjustment = controller.get_value(error)
        ...
```

10. This still goes into the same adjustment calculation through `pid`; however, this adjustment may also cause a change in direction here. Now, we set the secondary motor speed:

```
      ...
      # How fast should the motors move to get there?
      set_secondary(int(secondary_speed + adjustment))
      secondary_encoder.set_direction(math.copysign(1,
secondary_speed+adjustment))

      # Some debug
      logger.debug(f"Encoders: primary: {primary_encoder.
pulse_count}, secondary: {secondary_encoder.pulse_count},
e:{error} adjustment: {adjustment:.2f}")
      logger.info(f"Distances: primary: {primary_encoder.
distance_in_mm()} mm, secondary: {secondary_encoder.
distance_in_mm()} mm")
      ...
```

11. You could expand the debug that we had to take into account for the secondary speed and targets. Now, because we are trying for precision, the primary motor may reach its goal before the secondary and isn't set up to reverse. So, stop this motor when it reaches its goal, and set the base speed of the secondary to zero, which means only adjustments apply, if any. Note that we still use the absolute values here:

```
        ...
        # Stop the primary if we need to
        if abs(primary_encoder.pulse_count) >=
abs(primary_distance):
            logger.info("primary stop")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            set_primary(0)
            secondary_speed = 0
        ...
```

And we are done with the driving distances function. We can use this to drive in a straight line or feed it a separate target distance for each wheel and use that to drive in an arc. We'll take advantage of that in the next section.

Writing the drive_arc function

Here is where we convert to radians, determine the inner radius, and set up the distances for each wheel to drive. Add this code in `drive_square_behaviour.py`, after the `drive_distances` function:

1. Start with a function definition and a helpful docstring:

```
...
def drive_arc(bot, turn_in_degrees, radius, speed=80):
    """ Turn is based on change in heading. """
    ...
```

2. We turn the robot's width into ticks, the internal measurement of distance, and use half of that to get the wheel radiuses. We also determine which is the inner wheel:

```
    ...
    # Get the bot width in ticks
    half_width_ticks = EncoderCounter.mm_to_ticks(bot.
wheel_distance_mm/2.0)
    if turn_in_degrees < 0:
        left_radius = radius - half_width_ticks
        right_radius = radius + half_width_ticks
    else:
        left_radius = radius + half_width_ticks
        right_radius = radius - half_width_ticks
    logger.info(f"Arc left radius {left_radius:.2f},
right_radius {right_radius:.2f}")
    ...
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We display the debug on what the radiuses are. Combine this with the turn in radians to get the distances. We convert the absolute value of the turn in degrees. We don't want to reverse into a turn, but to turn the other way:

```
    ...
    radians = math.radians(abs(turn_in_degrees))
    left_distance = int(left_radius * radians)
    right_distance = int(right_radius * radians)
    logger.info(f"Arc left distance {left_distance},
right_distance {right_distance}")
    ...
```

4. Finally, feed these distances into the `drive_distances` function:

```
    ...
    drive_distances(bot, left_distance, right_distance,
speed=speed)
...
```

The robot should be able to drive in a square shape. It can still miss due to slipping or inaccuracies in the measurements. Tuning of the proportional and integral control values is required.

Examining the full code for `drive_distances` and `drive_arc`, it may become apparent that there is some repetition in determining the inner/outer and the primary/secondary parts, which you could refactor if you choose.

This code may not behave correctly if reversing through a corner.

Summary

In this chapter, we saw how to incorporate wheel encoder sensors into our robot and used them to determine how far each wheel has turned. We saw how to use this to get the robot onto a straighter path using a reduced PID controller and then used this to drive a specific distance. We then took the calculations further to calculate turning a corner in terms of wheel movements and driving the robot in a square.

A PID controller can be used in many situations where you need to apply a difference between a measurement and expectation, and you have seen how to combine this with sensors. You could use the same system to control a heating element connected to a thermal sensor. You could also use encoders to move robots with some precision, where the restricted range of motion used in servo motors does not make sense.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In the next couple of chapters, we will explore giving our robot even more interactive and intelligent behaviors, with chapters on visual processing using a Raspberry Pi camera, speech processing with Mycroft, and using a smartphone to drive or select modes on the robot remotely.

Exercises

1. Try experimenting with turning on different logging levels and differently named loggers, tuning how much output a robot behavior creates.
2. For the PID behaviors, tune the PIDs, try high values for the proportional or the integral, and observe how this makes the robot behave. Could you combine this with graphing in `matplotlib` to observe the PID behavior?
3. There are a few ways that the drive distance code could be improved. Applying a **PID** controller to the distance moved by the primary could make it close in more precisely on the exact distance to travel. Detecting no movement in either encoder could be used to make the code stop after a timeout so that it doesn't drive off without stopping. Try this out.
4. You could now use this code to make further geometric shapes or to follow paths without a line. Try adding high-level left turn/right turn 90-degree functions as building blocks for right-angled path construction, then use this to make paths.
5. Consider combining the encoding sensors here with distance sensors; it may be possible to start memorizing distances between walls.

Further reading

Please refer to the following for more information:

- PID control is a deep subject. It is a key area in self-balancing robots, drones, and other autonomous control systems. Here is a great video series so that you can explore these further:

  YouTube: Brian Douglas – *PID Control – A brief introduction*: `https://www.youtube.com/watch?v=UR0hOmjaHp0`

- I've greatly simplified some of the corner-turning algorithms. A very in-depth article on how this was used for a competition-winning LEGO Mindstorms robot holds a more detailed method:

  GW Lucas – *Using a PID-based Technique For Competitive Odometry and Dead-Reckoning: http*`://www.seattlerobotics.org/encoder/200108/using_a_pid.html`

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

12
IMU Programming with Python

Modern robots need to know their position relative to the world. In *Chapter 11*, *Programming Encoders with Python*, we looked at how encoders can get an idea of how much the robot has moved or turned. However, this turning was relative to where the robot was and had no absolute reference. Wheel slipping could lead to false readings. In this chapter, you will see more ways the robot can read changes in its position and measure its movements.

In principle, an **inertial measurement unit** (**IMU**) can give absolute position measurements and not slip. In practice, they are complicated. This chapter is a small practical tour of adding an IMU to your robot. I will introduce the components of an IMU in this chapter. You will also learn how to solder in order to add headers to a breakout, a skill that opens up a world of additional robot parts.

We'll write some code to try the various functions and see the kind of output the sensors provides. We will then make animated visualizations of the sensor data. By the end of this chapter, you will be able to work with these advanced sensors, have some soldering experience, and put together dashboards for monitoring sensors. As you investigate more in robotics, you'll learn that animated dashboards will be vital if you want to see what your robot can see.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this chapter, we're going to cover the following main topics:

- Learning more about inertial measurement units
- Soldering – attaching headers to the IMU
- Attaching the IMU to the robot
- Reading the temperature
- Reading the gyroscope
- Reading the accelerometer
- Reading the magnetometer

Technical requirements

For this chapter, you will need the following items:

- The robot from *Chapter 7*, *Drive and Turn – Moving Motors with Python*
- The robot code from *Chapter 11*, *Programming Encoders with Python*, which can be found at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter11`
- An ICM20948 breakout board with headers, such as the Pimoroni PIM448 module
- A soldering iron and stand
- A soldering iron tip-cleaning wire
- Solder – should be flux-cored solder for electronics
- A solder sucker
- A well-lit bench for soldering
- A ventilated space or extractor
- Safety goggles
- A breadboard
- A 2.5 mm standoff kit
- Female-to-female Dupont jumper wires

For the complete code for this chapter, please go to `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter12`.

Check out the following video to see the Code in Action: `https://bit.ly/38FJgsr`

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Learning more about IMUs

An IMU is a combination of sensors designed to sense a robot's movement in a 3D space. These devices are found in drones, useful in floor-based robots, and critical for balancing robots. The IMU is not a single sensor, but a collection designed to be used together and have their readings combined.

These devices are tiny but have their roots in flight hardware with large spinning gyroscopes. IMUs use the **micro-electro-mechanical systems** (**MEMS**) technology to put mechanical devices on micro-scale chips. They do have tiny moving parts and use electronic sensors to measure their movements.

Since some measurements are analog (see *Chapter 2*, *Exploring Controllers and I/O*, IMU modules often include an **analog to digital converter** (**ADC**) and communicate over I2C.

There are different combinations of sensors on an IMU. These sensors are as follows:

- A temperature sensor, to account for temperature effects on other sensors
- A gyroscope, which measures rates of rotation
- An accelerometer, which measures accelerations or forces
- A magnetometer, which measures magnetic fields and can act as a compass

As we work with each of these sensor types, we will learn more about them and their quirks.

Now that we know a little about IMUs, let's learn how to choose one.

Suggested IMU models

IMUs can be constructed with a separate accelerometer, gyroscope, and magnetometer, along with devices to convert the output of the sensor. To reduce the wiring and space this needs, I suggest using a board with all the devices or a single chip solution. For the same reason, I recommend I2C or serial IMUs.

IMU systems use **degrees-of-freedom** (**DOF**) to denote how many sensor axes are present. A 9-DOF system has three axes (X, Y, and Z) for each sensor.

BNO sensors are easier to code for but are incompatible with the Raspberry Pi due to the way they use the I2C bus, and they may require an intermediate interface chip.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Another thing to consider is if there is documentation (readme files and manuals) and a supported library to control the device from Python. The following picture shows a suggested IMU breakout:

Figure 12.1 – Photo of the ICM20948

The preceding image is of the PIM448 breakout board for the ICM20948, a well-supported 9-DOF sensor for Python libraries. It also has a temperature sensor. It is also well distributed. Since IMUs are complex devices, I strongly suggest choosing the PIM448 for this chapter.

Now that we've explored what IMU devices are and how to choose one, it's time to prepare a PIM448 for our robot with a new skill: soldering.

Soldering – attaching headers to the IMU

Most IMU breakouts, including the suggested PM448, are likely to come with headers in a bag, which you will need to solder onto the board. You are going to need a small bit of tuition if you want to solder on these headers:

Figure 12.2 – Bare PIM448 with headers

The preceding image shows the PIM448 as it comes out of the bag. On the left is the ICM20948 board with only holes and no headers. In the middle are the male headers, while the female headers are on the right. We will use the male headers since these are easier to hold in place when soldering.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

As we mentioned in the *Technical requirements* section, you need a soldering iron and solder, a soldering iron stand, safety goggles, an extractor or well-ventilated space, an additional breadboard, and a well-lit workspace. Soldering creates fumes that you do not want to breathe in.

Wear your safety goggles at this point. Heat the soldering iron; depending on the iron, this may take a few minutes. Pull out a bit of solder ready to use too.

Making a solder joint

You are now ready to make a solder joint.

The following image shows three of the stages of soldering the module:

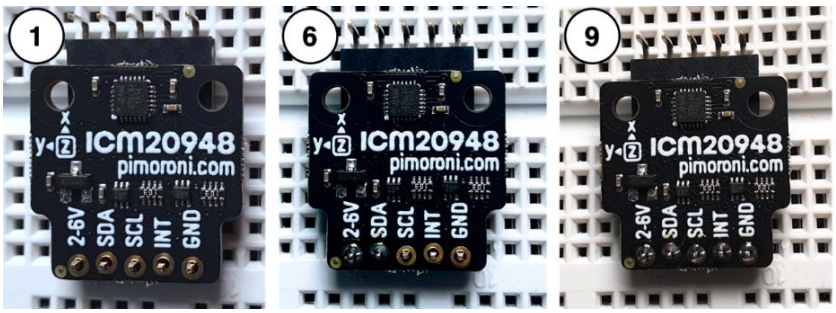


Figure 12.3 – Stages of soldering the PIM448

To make a solder joint, perform the follow steps while looking at the preceding image:

1. We need to ensure the part won't move while the solder dries. The preceding image shows the PIM448 lined up on the male headers, pushed into a breadboard on the bottom, with the female headers on the top. An excellent way to hold the part in place is to put the long side of the headers into the breadboard with our device on top. Since we are attaching male headers, we will use the female headers to prop up the other side.
2. The soldering iron tip should be hot at about 300°C and tinned. Melt a little solder on the tip to test that it's warm enough. Before you can solder, you need to tin the tip. Tinning is where you put a small layer of solder onto the iron to improve its heat conductivity and protect the tip from oxidizing (getting rusty when hot). To tin the tip, push a little solder into the iron's tip, and it will stick to it. The solder should melt freely.
3. Ensure the tip is clean – with the iron hot, push the tip of the iron into the brass cleaner, making a wiping motion with it in the wire.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Heat the pin from the header and the pad (the ring that the pin goes through). We'll start on the pin that reads **2-6V**. Heat both the pin and the pad to avoid a dry joint, where the solder will not flow properly over the pad. Dry joints are weak, both electrically and mechanically.
5. After a second or so, gently feed a little solder into the other side of the pin; when the pin is hot enough, the solder will melt and flow over the pad, making a rounded tent-like shape. This is just enough solder. You will see flux resin coming from the solder.
6. The preceding image shows the next step in the middle. Here, I've soldered two pins; things gets easier from here on out since the board can't move. Move on to the next pin and repeat – heat the pin and pad, then feed in the solder.
7. If you've added too much solder, use a solder sucker to remove the excess. Push down the plunger, bring the sucker up close to the joint, melt the solder, and press the release button of the plunger for it to suck any solder away. You can remake this joint with a bit less solder.
8. If you find you've connected two pins with a blob of solder (bridged them), you can draw the hot iron down between the pins to divide them again. You may also need to remove any excess solder, as mentioned in *Step 7*.
9. Repeat the preceding steps for the remaining pins. The right-hand side of the preceding image shows what your IMU should look like once all the pins have been soldered.

> **Important Note**
> For the sake of safety, ensure that you return the soldering iron to its stand and switch the iron off before you do anything else. A soldering iron can be a dangerous device, leading to burns or fires if left unattended.

10. Once the part is cool, unplug it from the breadboard. Optionally, you can use isopropyl alcohol and a cotton bud to clean away flux residue for a better look.

Before we wire this, make the following checks:

- You have soldered all five pins in place.
- Each soldered pin is like a silver "tent" shape. A bubble/round or flat shape is not right, and you will need to make that connection again.
- No two pins have solder *bridges* – blobs of solder connecting the pins.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Congratulations – you have soldered your first part! This is a skill you will need again as you build more robotic and electronic devices. Now that you have soldered the ICM20948 module, let's attach it to your robot.

Attaching the IMU to the robot

Before we can use the IMU and write code for it, we must securely mount it on the robot and wire it so that the Raspberry Pi can talk to it.

Physical placement

The IMU magnetometer is sensitive to magnetic fields and needs to be away from the motors. For this reason, it should be on a stalk above the robot.

The orientation of the IMU is essential for other experiments to make sense:

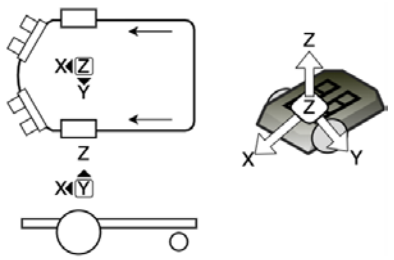


Figure 12.4 – Lining up the IMU with the robot

There is a diagram on top of the IMU. The preceding diagram shows how this diagram should line up with the robot. The X-axis should face forward, while the Z-axis should face up, with the little square on the IMU pointing upward. Finally, the Y-axis should point to the left.

The sensor uses I2C. I2C is sensitive to wire distances, so we should mount it above the Raspberry Pi and motor control board where the wire distances are low. The following image shows the parts you will need to do this:

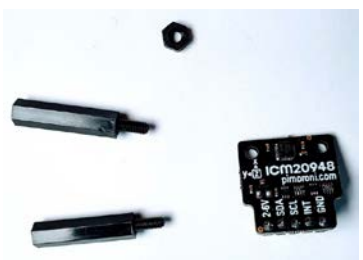


Figure 12.5 – Parts needed to attach the IMU

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

For this step, you will need the parts shown in the preceding image:

- The IMU, with headers mounted
- Many long standoffs, M2.5
- 1x M2.5 nut

We will assemble these parts using the standoffs to make a long post, as the following image shows:

Figure 12.6 – Joining the standoff posts

The following steps are intended to be used with the preceding image to help you mount the IMU:

1. As shown in the preceding image, you just need to screw the thread of one post into the socket of the other to end up with a long post. This should give the IMU a little distance so that it can stand above the robot. Aim to be just under the cable length:

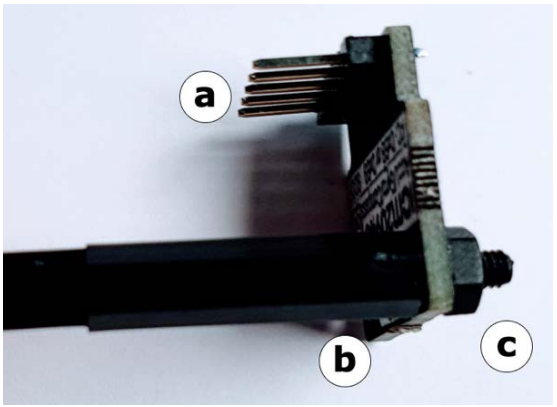


Figure 12.7 – Bolting the IMU to the post

2. As shown in the preceding image, push a post thread through the hole opposite the axis markers on the IMU *(a)*. The headers *(b)* should be facing down into the post. The thread is quite a snug fit but should fit through. Use the nut on top *(c)* to secure it in place:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

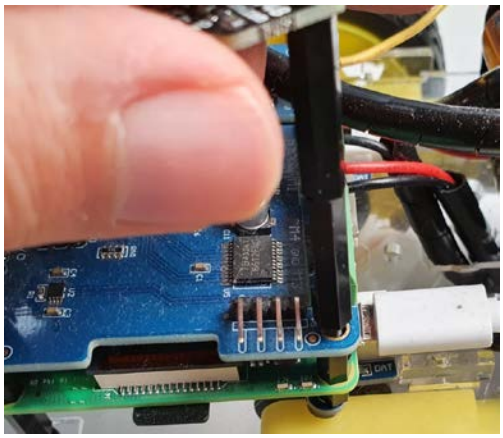


Figure 12.8 – Bolting the IMU post to the Raspberry Pi

3. The preceding image shows the IMU post screwed onto a thread sticking up from the motor board. The motor board we suggested in *Chapter 6*, *Robot Building Basics – Wheels, Power, and Wiring*, has an I2C connector to the rear left of this board. We can bolt the IMU post to a hole near that:

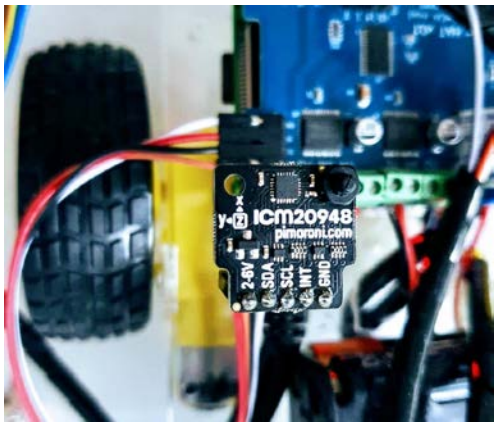


Figure 12.9 – The IMU ready for wiring

4. The preceding image shows the ICM20948 attached to the post, which you bolt into the top of the motor board, with its pins ready for wiring. Adjust it so that the X-axis points forward and the Y-axis points to the left while tightening the top nut. The closer this is to square with the robot, the better your results will be!

You have now mounted the IMU on the robot. You've lined up its axes, so we know what to expect from our sensors. Now that we have fitted this IMU module, it's firmly in place, but could be unbolted if we need to do that. The module is now ready for wiring.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Wiring the IMU to the Raspberry Pi

Next, you need to wire the IMU to the I2C pins on the Raspberry Pi. While this seems to be an easy job, you must watch out since some connections aren't straight through.

The motor board's handy I2C breakout should make this job a little easier:

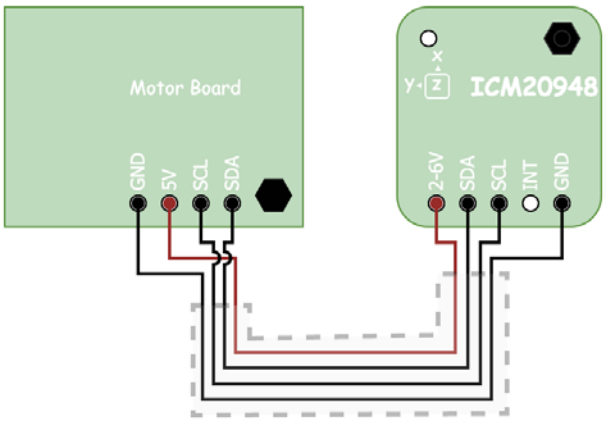


Figure 12.10 – The wiring of ICM20948

As shown in the preceding diagram, the wiring is pretty straightforward: the **GND** from the IMU goes to the **GND** on the motor board I2C breakout, **SDA** goes to **SDA**, **SCL** goes to **SCL**, and **2-6V** goes to **5V** (in the 2-6V range).

The **GND** goes from the left of the motor board to the right of the IMU. The four wires have a bend, with the **5V** line crossing it.

In practice, we would use a jumper cable strip of four wires, which is shown by the dashed lines in the preceding diagram. The end going to the IMU would go straight through. The end going to the motor board has the power cable crossing the other wires:

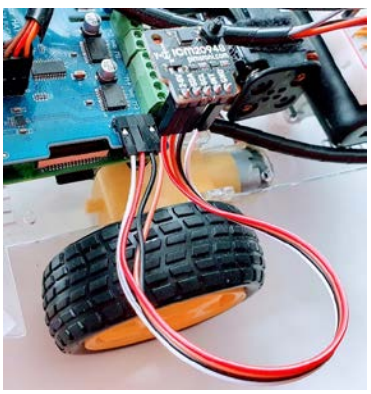


Figure 12.11 – ICM20948 IMU wired to the motor board

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In the preceding image, I've used a short female-to-female jumper wire to make the connections. The IMU board is at 90 degrees from its intended orientation to make the wiring more visible; it should have X facing forward. Notice that there is a twist in the wire, so the GND line (white here) ends up on the GND pin on the other side. Perform the following steps to make these connections:

1. Carefully pull off a strip of four wires. Aim to find a darker color for GND and a bright/vivid color for the 5V line.
2. Plug one side directly into the IMU, ensuring you skip the INT pin.
3. As you bring the wire to the motor board below, put a small turn in so that the cable faces the other way.
4. Plug the GND in first, to set the orientation.
5. Plug the 5V line in next, which will need to cross the other two wires.
6. The final two wires should now be in the right orientation for SDA and SCL; plug in both.
7. Use the wire colors to ensure you've made the right connections.

We do not intend to use the `INT` pin. This pin is designed to send an *interrupt* to the Pi, to notify us that there is a motion for wake-on-motion type behavior. However, use of this is beyond the scope of this book.

Now that we have wired this sensor in and attached it to our robot, you are ready to write some code. We'll start easy by reading the temperature.

Reading the temperature

With the device wired and attached, you'll want to try some code on it to confirm we can talk to this device and get data out of it. Let's get some tools installed and make it work.

Installing the software

Before we can start interacting with this device, as with most devices, we will install a helper library to communicate with it. Pimoroni, the suppliers of the ICM20948 module I've suggested, have made a handy library for Python to talk to it. I recommend taking their latest version from GitHub.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Perform the following steps to install it:

1. Boot up the Raspberry Pi on the robot. This Pi should have been used previously for the motor board and LED shim and have I2C enabled. If not, go back to *Chapter 7*, *Drive and Turn – Moving Motors with Python*, and follow the steps for preparing the I2C.
2. Type in `i2cdetect -y 1` to check that you've installed the device correctly. The output should look like this:

```
pi@myrobot:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- 6f
70: 70 -- -- -- -- -- -- —
```

3. The device at 0x68 is our new device. If you do not see this, please power down the Raspberry Pi and check your wiring. The other two devices (0x6f and 0x70) are the motor board and the LED shim.
4. Now, we can install the library:

```
pi@myrobot:~ $ git clone https://github.com/pimoroni/
icm20948-python
pi@myrobot:~ $ cd icm20948-python/
pi@myrobot:~ $ sudo ./install.sh
pi@myrobot:~ $ cd
```

5. You've now verified that the ICM20948 device is on the robot's I2C bus and installed the Pimoroni Python library so that it can talk with it. You are now ready to talk to it.

We also will add some new software to visualize our data in real time. There is a system called **Visual Python** (**VPython**) that's been designed to create graphs and 3D representations in real time:

```
pi@myrobot:~ $ pip3 install git+https://github.com/orionrobots/
vpython-jupyter.git
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, the device and library should be installed. If this didn't work for you, try looking at the *Troubleshooting* section, which is next.

Troubleshooting

Things can go wrong at this early stage. If you've not made things work so far, please try following these steps:

1. It is vital that the `i2cdetect` stage works here and shows the device at 0x68. If not, check your wiring. *Nothing* should be hot here.
2. Ensure you have followed all the soldering checks.
3. If the libraries fail to install, ensure you get connected to the internet. You may need to have the most recent version of Raspbian for them to work.

Now that you have installed the device and checked for common issues, we can try our first experiment with it and read the temperature sensor.

Reading the temperature register

In this section, we are going to set up an interface for the IMU, and then add a real-time graph for the temperature data from the Raspberry Pi.

Creating the interface

As with other sensors and outputs, we must create an interface because there are many IMU devices on the market. However, the same interface allows us to change them out without rewriting other behaviors using that interface:

1. Create a file named `robot_imu.py`
2. Start by importing the Pimoroni device library – this will be different if you use another IMU device:

```
from icm20948 import ICM20948
```

3. We'll make an IMU class to represent our device. This sets up a single IMU:

```
class RobotImu:
    def __init__(self):
        self._imu = ICM20948()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. For this exercise, we only need the temperature. Let's simply wrap that:

```
    def read_temperature(self):
        return self._imu.read_temperature()
```

With this, the interface is ready. Now, we can use it to read the device's temperature.

What is VPython?

VPython or Visual Python is a system designed to make visual – even 3D – displays in Python. It comes from a scientific community and will become very useful throughout this chapter. It serves output to a browser, and with the specific version installed here, it can be run on a Raspberry Pi while showing the output on a computer or smartphone.

It has a few quirks, with one of them being a slow startup time, but it is worth it for the results.

Graphing the temperature

A good way to observe temperature variations is by using a graph.

Let's use VPython and create a graph showing the temperature of our IMU module:

1. Create a file named `plot_temperature.py`.
2. Start by importing VPython and our robot IMU interface:

```
import vpython as vp
from robot_imu import RobotImu
```

   Notice how we've abbreviated `vpython` by importing it as `vp`.

3. We are going to plot temperature versus time on a graph, so we will need a time reference. Also, we will use logging to see what is going on:

```
import time
import logging
```

4. Let's configure logging so that we can see all the `INFO` level logs:

```
logging.basicConfig(level=logging.INFO)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Create our IMU instance:

```
imu = RobotImu()
```

6. We want a few things from the graph. Since the X-axis is time and is in seconds, setting the minimum to `0` and the maximum to `60` will show us a minute of data. We also want the graph to scroll so that it shows new data once we've recorded more than a minute:

```
vp.graph(xmin=0, xmax=60, scroll=True)
temp_graph = vp.gcurve()
```

7. Now that we have a time reference, let's record the start time before we get into the loop:

```
start = time.time()
```

8. The main loop is a `while true` type. However, we need to use `vp.rate` to let VPython know we are animating and set a frame/update rate for our system:

```
while True:
    vp.rate(100)
```

9. Now, we can capture our temperature, and while we are at it, we can log this:

```
    temperature = imu.read_temperature()
    logging.info("Temperature: {}".format(temperature))
```

10. To put this into the graph, we need to get the elapsed time for the X-axis. We can get this by subtracting the start time from the current time:

```
    elapsed = time.time() - start
```

11. Finally, we need to plot this in our temperature graph, with the elapsed time as `x` and the temperature as `y`:

```
    temp_graph.plot(elapsed, temperature)
```

The code for plotting the temperature is now live. Let's run this on the Raspberry Pi.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Running the temperature plotter

There are a few steps we need to follow to run this once we've copied the files to the Raspberry Pi. Our Raspberry Pi is headless, so we will need to view VPython remotely. To do so, we need to let VPython know we are doing this, and use a network port to make its view available. We can then use a browser to look at this. Let's see how:

1. In an SSH session to the Raspberry Pi, type the following:

```
$ VPYTHON_PORT=9020 VPYTHON_NOBROWSER=true python3 plot_
temperature.py
```

   We've chosen port `9020`, which is somewhat arbitrary but should be above `1000`. We will be using other web services later in this book on different ports, and this number is well clear of their ranges. When run, it should log a few messages to tell you it is ready:

```
INFO:vpython.no_notebook:Creating server
http://localhost:9020
INFO:vpython.no_notebook:Server created
INFO:vpython.no_notebook:Starting serve forever loop
INFO:vpython.no_notebook:Started
```

   Note that it shows a localhost address. We intend to use it remotely.

2. Next, point your browser (Chrome, Firefox, or Safari) from your desktop to the Raspberry Pi with the port number. In my case, based on my robot's hostname, this would be `http://myrobot.local:9020`.
3. Now, be prepared to wait a bit – it can take a bit of time for VPython to set up. After this, you will either see your graph or any errors/problems.

When it's running, you'll get a graph of the readings from the temperature sensor. You can experiment a little by carefully placing a finger on the sensor (the large black square on the PIM448) and watching the graph rise/fall in response to this. You could also find cold or hot objects, such as a hair dryer, to see how this manipulates it. However, be careful not to get the robot wet, and don't let metal touch the pins:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

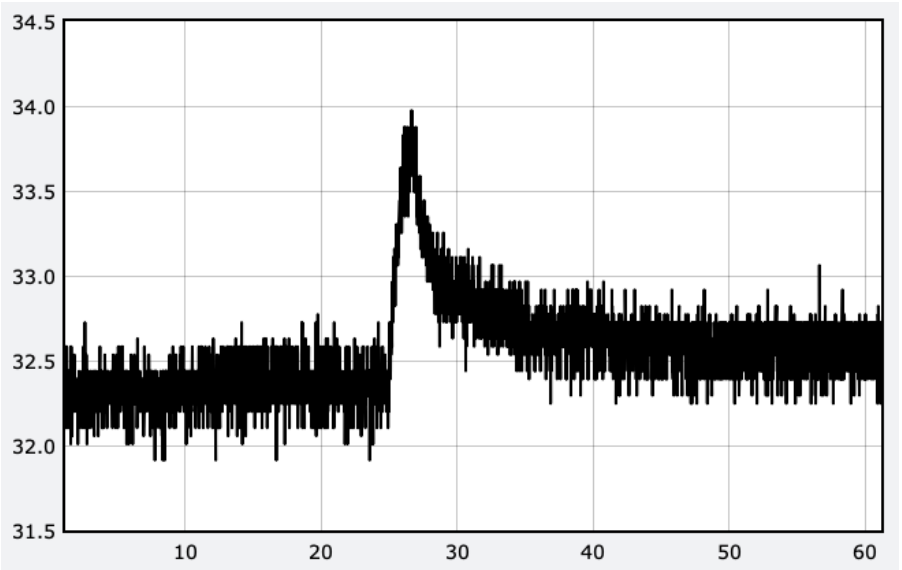


Figure 12.12 – A temperature graph

The preceding image is a graph that's showing temperature in degrees (Y-axis) versus time in seconds (X-axis). The thick black line indicates the current temperature reading. It wiggles a lot – this is a noisy system.

I placed my finger over the sensor at about 25 seconds. As shown in the preceding graph, the ambient temperature was 31 and raised to just under 34. It takes a few seconds to warm up. Keeping my finger there longer would have made it increase more. I had a fan present, so there was a sharp drop-off – there can be far slower drop-offs depending on your conditions. The code also logs the temperatures to the console:

```
INFO:root:Temperature 32.43858387995327
INFO:root:Temperature 32.726120945278105
INFO:root:Temperature 32.726120945278105
INFO:root:Temperature 32.39066103573247
INFO:root:Temperature 32.39066103573247
INFO:root:Temperature 32.63027525683649
```

There is a lot of noise in the decimal places that you can ignore here. When you close this browser tab, the code will stop graphing.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Important Note**
>
> A warning about testing temperatures: Do not put metal objects on the sensor – this may short out the pins and damage the robot. Also, do not put wet items on it. Very cold objects may have condensation on them. Water will even short pins and damage the sensor, and possibly the Raspberry Pi.

Troubleshooting

We are pulling two new components into our robot code here, so things may go wrong. Here are some things to check:

1. Be aware that VPython can be slow, so it may take a long time to start. Try refreshing the browser tab after 30 seconds.
2. With VPython, it may take a long time to show an error message. Patience is needed when trying new code here.
3. If you see I/O or communication errors, carefully check the wiring of the IMU. Please go back to the *Installing Software Troubleshooting* section for measures. I/O errors can also happen if you nudge a wire out while putting your finger on the sensor, or worse still if you try to cool it with a metal object and short the pins. DO NOT PUT A METAL OBJECT ON THE SENSOR!
4. Similarly, if you see import errors, check that you do not have typing errors in the imports and ensure you have checked the *Installing software troubleshooting* section.
5. If the temperature reading takes time to change, note that the IMU has some insulation/thermal resistance, so it takes a while to warm up (but it will) and cool down. The board also has a thermal mass, meaning it will all heat up or cool down, slowing the time it takes to reach the same temperature as the one you are measuring.
6. There can be a few reasons for the temperature reading not being accurate. For one, the IMU can produce some heat – we've already mentioned the thermal mass. It could have a calibration offset value applied to it to make it more accurate, but do not expect it to match a thermometer to a fraction of a degree perfectly. It should certainly be able to register a finger or palm as close to 37 degrees, but in practice, and with patience, I usually got to about 36 point something.

Our example is now working, but we could make it a bit easier to start our tests.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Simplifying the VPython command line

We will be using VPython a lot in this chapter, and we don't want to type in a mouthful of settings to run each Python file. Let's create an alias (a command-line shortcut) to save us from having to type that stuff in every time:

1. Let's set it up for the current session. The `alias` command makes an alias we can reuse later. Here, it's named `vpython`. It contains the settings and the `python3` command:

```
pi@myrobot:~ $ alias vpython="VPYTHON_PORT=9020 VPYTHON_
NOBROWSER=true python3"
```

2. So that we can use it again at some point, we will put it into the current user's `.bashrc` file – a file that Raspbian automatically runs when you `ssh` in:

```
pi@myrobot:~ $ echo 'alias vpython="VPYTHON_PORT=9020
VPYTHON_NOBROWSER=true python3"' >>~/.bashrc
```

   Wrapping something in `echo` will write text out instead of running a command. `>>` appends this to a file – in this case, `.bashrc`. The `~` mark picks the current user's home.

3. You can rerun the temperature demo with `vpython plot_temperature.py`.

In this section, you received data from the IMU device and saw how it responds to temperature. This confirms that the IMU is responding. You logged the data and graphed it and were introduced to the VPython system in the process, which can be used as a powerful graphics display system. We will use both the IMU and VPython to do far more throughout this chapter. Next, we will look at the gyroscope so that we can see how our robot is turning.

Reading the gyroscope in Python

In this section, we are going to use the gyroscope in the IMU. We will use it to approximate where the robot is facing in three dimensions.

But before we do that, let's understand it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Understanding the gyroscope

A gyroscope measures rotation as a rate of change in angle, perhaps in degrees per second. At each measurement, it can determine the speed of rotation around each axis:

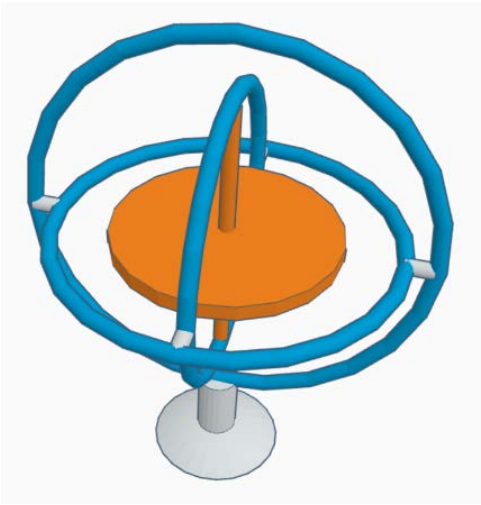


Figure 12.13 – Illustration of a gyroscope

A gyroscope is traditionally a mechanical system, as shown in the preceding image. It has a gimbal – a set of concentric rings – connected by pivots so that they can pivot around the X-axis, Y-axis, and Z-axis. The middle has a spinning mass, known as a rotor. When the rotor is spinning, moving the handle (shown as a stand at the bottom of the image) does not affect the spinning mass, which keeps its orientation, with the gimbals allowing it to turn freely.

In the case of a MEMS gyroscope, it moves a tiny mass back and forth (oscillates) very quickly. When the orientation of the gyroscope is changed, the mass will still be moving in another direction. This movement will change an electrical field that the sensor detects. In the original orientation, this movement appears to be a force, known as the Coriolis force.

Before we can write some code so that we can work with the gyroscope, we need to understand coordinate systems – on the robot and in VPython.

Representing coordinate and rotation systems

We are going to be using coordinate and rotation systems in this chapter. The following diagram should help you understand them:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

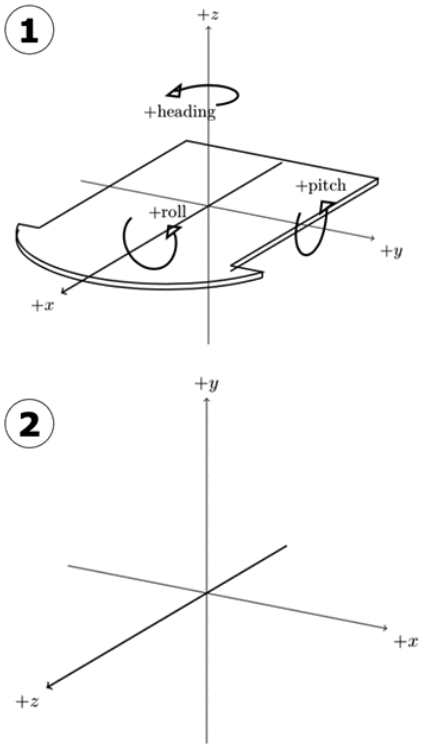


Figure 12.14 – The robot body coordinate system

The preceding diagram shows the different coordinate systems we will be using. Let's take a look at the different sections of it:

1. This is the robot's **Body Coordinate System** – a stylized 3D sketch of the robot with three axis arrows. First, there's the X-axis, which points toward the front of the robot. Rotating about this X-axis is known as **roll**. Then, there's the Y-axis, which indicates to the left of the robot (your right as you view the robot). Rotating about this axis is known as **pitch**. Finally, pointing up through the robot is the Z-axis. Rotating about this axis is known as **heading** or **yaw**.

   The direction of rotation is important. There is a rule of thumb for this: take your right hand and put your thumb up. If your thumb is pointing along the axis, then the fingers on your fist have wrapped the way the rotation will go.

2. This is the VPython **World Coordinate System**. We display 3D images in VPython here. VPython's coordinate system is a rotation of the robot body system.

   In the preceding diagram, the Y-axis is going up, the X-axis is going to the right, and the Z-axis is pointing forward.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will represent the coordinates in 3D as X, Y, and Z components – this is known as a **vector**.

When we apply our measurements to things in the VPython system, we will align our view with the robot coordinate system. When we're talking about a coordinate system relative to another, this is known as a **pose**. This is the robot's pose with respect to the VPython coordinate system.

Let's represent this with a bit of code to help us out:

1. Create a file named `robot_pose.py`.
2. We are manipulating the VPython view, so we need to import it, as follows:

```
import vpython as vp
```

3. We can then add our function to set the view up; I've called it `robot_view`:

```
def robot_view():
```

4. In this function, we need to set the two properties that VPython uses to control camera orientation:

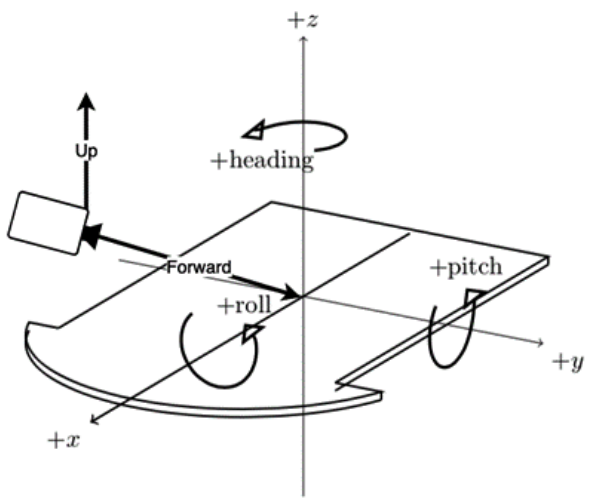


Figure 12.15 – Camera direction

The preceding diagram shows the camera looking at a robot in the coordinate space. The camera will look in the defined `forward` direction toward zero. It also needs an `up` direction to constrain the camera's roll.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We want it to look from the front of the robot, so `forward` should be pointed in a negative X direction. We also want the camera a little above and to the side:

```
vp.scene.forward = vp.vector(-3, -1, -1)
```

5. An axis tells us where to look along, but not which way *up* is. We need the camera to align its definition of up with the robot (which has Z pointing up); otherwise, the vectors could be upside down or to the side:

```
vp.scene.up = vp.vector(0, 0, 1)
```

We will use this pose more in later sections; however, for now, it's useful to see that the Z-axis is now up, as well as where we rotate around the different axes.

Now, let's set up the gyroscope for reading.

Adding the gyroscope to the interface

Before we can read the gyroscope, we'll need to add it to our `robot_imu.py` interface:

1. We are going to be dealing with a few x, y, and z groups from our IMU. We will import a vector type to store these. I've highlighted the new code here:

```
from icm20948 import ICM20948
from vpython import vector
```

2. A vector is a representation of three component coordinate systems. Now, we need to fetch the gyroscope data from the underlying IMU library and store it in a vector:

```
    def read_gyroscope(self):
        _, _, _, x, y, z = self._imu.read_accelerometer_
gyro_data()
```

The Pimoroni `ICM20948` library we are using does not have a call to return only gyroscope data, but it does have one that returns both accelerometer and gyroscope data.

This `ICM20948` library returns the data as a list of six items. In Python, when unpacking return values, the underscore character, _, can denote things to ignore.

3. We can now put the three gyroscope values into a body vector to return them:

```
        return vector(x, y, z)
```

The IMU library is now ready for us to read gyroscope data from it. Next, we are going to read it and plot the data on a graph.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Plotting the gyroscope

As we mentioned previously, the gyroscope measures the rate of rotation. It does so in degrees per second on each axis.

Let's graph the output of this device:

1. Create a file named `plot_gyroscope.py`.
2. We'll start with the imports, setting up logging, and the IMU, as we did previously:

```
import vpython as vp
import logging
import time
from robot_imu import RobotImu

logging.basicConfig(level=logging.INFO)
imu = RobotImu()
```

3. We set up three graphs for the three axes that the gyroscope outputs – X rotation, Y rotation, and Z rotation. Note that we give each graph a different color:

```
vp.graph(xmin=0, xmax=60, scroll=True)
graph_x = vp.gcurve(color=vp.color.red)
graph_y = vp.gcurve(color=vp.color.green)
graph_z = vp.gcurve(color=vp.color.blue)
```

   The three graphs will overlay on the same line.

4. Now, we need to set a start time, start a loop, and measure the elapsed time:

```
start = time.time()
while True:
    vp.rate(100)
    elapsed = time.time() - start
```

5. We can now read the IMU and put the three readings into the graphs:

```
    gyro = imu.read_gyroscope()
    graph_x.plot(elapsed, gyro.x)
    graph_y.plot(elapsed, gyro.y)
    graph_z.plot(elapsed, gyro.z)
```

6. Upload the files and run them with `vpython plot_gyroscope.py`.
7. Wait a minute or so and then point a browser at `myrobot.local:9020` – it can take up to 1 minute for this to appear.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Start to move the robot around – lift it and try tilting in each of the three axes. You should see something like the following graph:

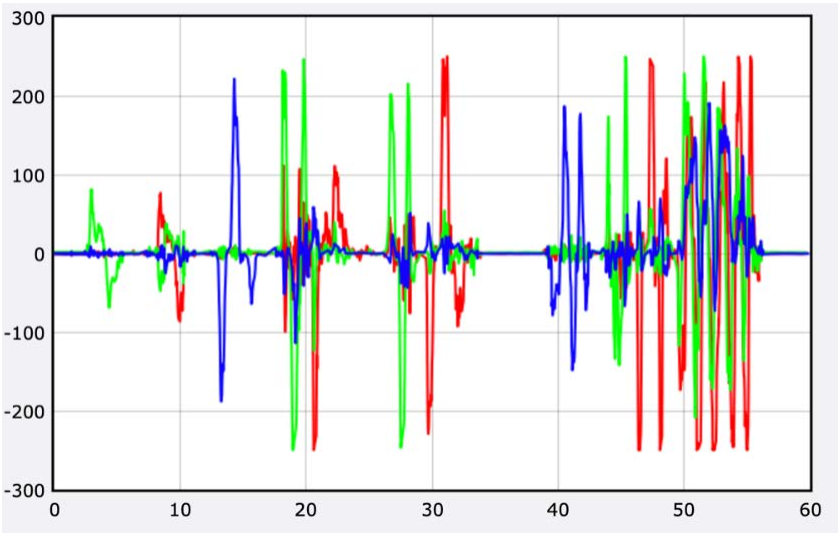


Figure 12.16 – VPython plot of gyroscope data

The preceding graph contains three lines. The Y-axis shows a movement rate in degrees per second, while the X-axis shows the time in seconds since the program started. On-screen, the graphs are in red, green, and blue.

The graph spikes when you make movements, and then returns to zero. Try pushing the front of the robot (the nose) down; this is positive around the Y-axis. The green line should move up (shown at about 3 seconds in the preceding graph). If you keep it there, the line will flatten. When you return the robot to flat, there will be a negative green spike on the line. Now, try lifting the left-hand side by turning it around the X-axis, creating a positive red spike on your graph. When you return it flat, you'll get a negative peak. Next, try turning the robot to the left; this will create a positive blue spike. Now, if you turn it to the right, a negative blue spike will be created. Move around the axes to get a feel for these measurements.

Unless you are spinning a robot constantly, you'll likely find that it's reasonably hard to keep up any turning force; this shows that the gyroscope data is a rate of turn, and not a measure of direction. What would be more useful is to approximate the heading of the robot. When we dive deeper, we'll learn how to use gyroscope data for this.

In this section, you've seen the gyroscope and how it measures rotation rates via a graph demonstrating this principle. Now, let's move on to the accelerometer so that we can see the forces that are acting on our robot!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Reading an accelerometer in Python

In this section, we will learn how to use an accelerometer to measure forces acting on the robot, and most often, which way is down. Let's find out more about it, then write some code to see how it works.

Understanding the accelerometer

An accelerometer measures acceleration or changes in speed, both in terms of size and direction. It does so by providing three values – one for each of the X, Y, and Z axes:

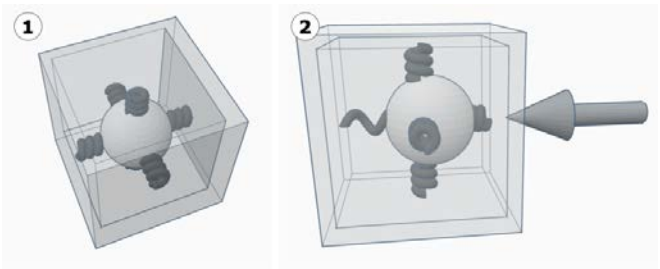


Figure 12.17 – Accelerometer concept – mass with springs

The preceding diagram shows a conceptual view of an accelerometer. Let's take a look at it in more detail:

1. This shows a ball (a mass) suspended by six springs in a box. When there are no forces on the box, the ball stays in the middle.
2. This shows how this system behaves when the large arrow pushes it. The mass retains inertia by moving to the right, compressing the right spring and extending the left spring.

Measuring the position of the mass shows the direction and size of an acceleration. A MEMS accelerometer is similar to this device and is constructed with a tiny silicon mass and springs. This measures an electric field that changes as the mass moves.

While on Earth, a mass is pulled downward by gravity. This system behaves like a force is holding the box up, so an accelerometer will usually register an upward force. We can use this measurement to determine which way down is and sense the tilt of a robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Adding the accelerometer to the interface

Let's start by adding the accelerometer measurement to our `RobotImu` library:

1. Open the `robot_imu.py` file.
2. Add the following code to do the reading:

```
    def read_accelerometer(self):
        accel_x, accel_y, accel_z, _, _, _ = self._imu.
read_accelerometer_gyro_data()
        return vector(accel_x, accel_y, accel_z)
```

   This uses the same library call as the gyroscope; however, it now discards the last three data items instead of the first three.

Now that the accelerometer is ready to read, we can render this to make the data visible.

Displaying the accelerometer as a vector

The acceleration is a vector; it points to a 3D space with a direction and size. A great way to show this is as an arrow in 3D. To clarify where this vector is, we can plot an indicator for each of the X, Y, and Z axes:

1. Create a file named `accelerometer_vector.py`. Start it with some simple imports, including the robot view, the logging setup, and initializing the IMU:

```
import vpython as vp
import logging
from robot_imu import RobotImu
from robot_pose import robot_view
logging.basicConfig(level=logging.INFO)
imu = RobotImu()
```

2. Let's look at this from the angle we tend to view the robot at:

```
robot_view()
```

3. Now, we want to define four arrows. VPython arrows point along an axis and can have their color and length set:

```
accel_arrow = vp.arrow(axis=vp.vector(0, 0, 0))
x_arrow = vp.arrow(axis=vp.vector(1, 0, 0),
                   color=vp.color.red)
y_arrow = vp.arrow(axis=vp.vector(0, 1, 0),
                   color=vp.color.green)
z_arrow = vp.arrow(axis=vp.vector(0, 0, 1),
                   color=vp.color.blue)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Now, we can start the main loop:

```
while True:
    vp.rate(100)
```

5. Read the accelerometer and log it:

```
    accel = imu.read_accelerometer()
    print(f"Accelerometer: {accel}")
```

6. Because bumps can knock our scale out, we will normalize the vector to so that its length is 1. We need to put this in the arrow axis:

```
    accel_arrow.axis = accel.norm()
```

7. Upload this to the Raspberry Pi and start it with `vpython accelerometer_vector.py`. Point your browser to it to see the following output:

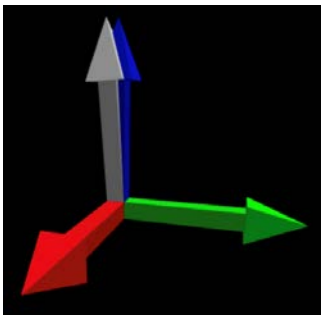


Figure 12.18 – The accelerometer vector

The preceding image shows the three colored arrows – red for the X-axis (pointing to the viewer), green for the Y-axis (pointing left), and blue for the Z-axis (pointing up). There is a gray arrow showing the accelerometer vector. The accelerometer points up, which shows what is holding it up against gravity.

8. Now, if you tilt the robot, the arrow will tilt to show you which way up is relative to the robot. You can tilt the robot a few ways to see how this feels.

This is exciting – you have now shown where up is, relative to your robot. To use this to rotate things, we need to turn this vector into pitch and roll angles, which we'll learn how to do when we dive deeper.

In this section, you have learned how to read data from the accelerometer component and how to display it as a vector. Now, we will move on to the next element of the IMU, known as the magnetometer, and read the magnetic fields that are acting on our system.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Working with the magnetometer

A magnetometer reads magnetic field strengths in 3D to produce a vector. Code you write can use this to find the magnetic north, in the same way as a compass. In this section, we'll look closer at the device, learn how to get a reading from it, and see what vectors it produces.

It may be useful to have a space with very few magnets present. Let's understand the magnetometer more.

Understanding the magnetometer

A compass measures a heading from the Earth's magnetic field by using a magnetized needle or disk. The following image is of a compass:

Figure 12.19 – A traditional compass

The compass shown in the preceding image has a rotating magnetized disk balanced on a center pin. This variety is a small *button compass*, which is about 25 mm in diameter.

Our chosen IMU contains a device known as a **magnetometer**. This electronically senses a magnetic field and can be used as a compass.

Most magnetometers pass electricity through a material that creates a current when it's exposed to a magnetic field, as shown in the following diagram:

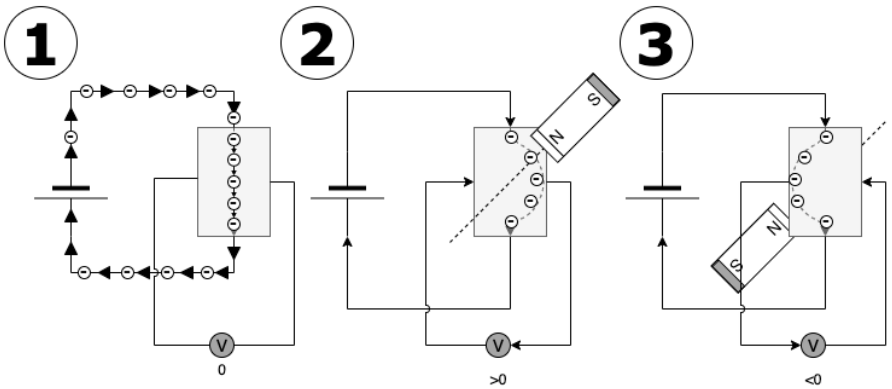


Figure 12.20 – A stylized picture of a Hall-effect sensor

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The preceding diagram shows an example of this in action:

1. This circuit passes an electric current from a battery (left) through a conducting plate (gray rectangle). The arrows demonstrate the electrons (negative charge carriers) that are moving around the circuit, from the top of the plate straight to the bottom. The small circle with a V inside it is a voltage (electric flow) sensor that's connected to the sides of the plate. The voltage sensor reads 0 since there's no flow to the sensor.
2. A magnet is above the plate, deflecting the electrons to one side. They give one side of the plate a negative charge, and the other side a positive charge. This difference in charge makes voltage flow through the sensor, as shown by the arrows. The reading below is now above zero.
3. Putting the magnet on the other side of the sensor changes the magnetic field; the electrons are deflected to the other side, causing reverse voltage to flow. The arrows going to the meter are going in the opposite direction, and the reading shows a voltage below zero.

This is known as the Hall effect. By measuring three plates, you can measure magnetic fields in three dimensions. Magnetometers are sensitive to magnetic fields and metal objects.

Another quirk is that on some IMUs, the magnetometer's axes are different from the other devices':

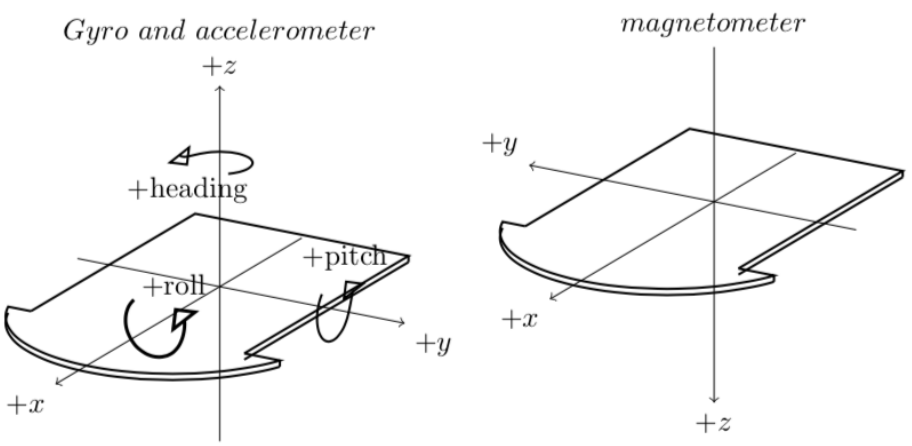


Figure 12.21 – The magnetometer's axes

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In the preceding diagram, the axes we looked at previously are shown on the left for the gyroscope and accelerometer. On the right, we can see the axes for the magnetometer. Here, we can see that the Z-axis points downward and that the Y-axis now points backward. It's like we've rotated 180 degrees around the X-axis.

Now, let's add some code so that we can read this information.

Adding the magnetometer interface

We'll wrap this the same way we wrapped the other readings; that is, by adding it to our interface library:

1. Open the `robot_imu.py` file.
2. In the `RobotIMU` class, after the `read_gyroscope` method, add the new read method:

```
    def read_magnetometer(self):
```

3. Unlike the accelerometer and gyroscope, this reads data from a separate call to the underlying device library. We wrap this up and return a vector. For a cheeky rotation by 180 degrees, we negate the Y and Z axes:

```
        mag_x, mag_y, mag_z = self._imu.read_
magnetometer_data()
        return vector(mag_x, -mag_y, -mag_z)
```

Now that this interface is ready to use, let's get some readings.

Displaying magnetometer readings

One way we can visualize this is to turn magnetometer output into a vector, like so:

1. Create a file named `magnetometer_vector.py`.
2. Add the familiar imports and setup:

```
import vpython as vp
import logging
from robot_imu import RobotImu
from robot_pose import robot_view
logging.basicConfig(level=logging.INFO)
imu = RobotImu()
robot_view()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Now, we will create an arrow for the magnetometer reading, along with the reference X, Y, and Z axes:

```
mag_arrow = vp.arrow(pos=vp.vector(0, 0, 0))
x_arrow = vp.arrow(axis=vp.vector(1, 0, 0), color=vp.
color.red)
y_arrow = vp.arrow(axis=vp.vector(0, 1, 0), color=vp.
color.green)
z_arrow = vp.arrow(axis=vp.vector(0, 0, 1), color=vp.
color.blue)
```

4. Next, we start the main loop:

```
while True:
    vp.rate(100)
```

5. Now, we can read the magnetometer:

```
    mag = imu.read_magnetometer()
```

6. Finally, let's set an arrow's axis that will match this vector. We can use the `.norm()` method to normalize this vector. We also need to print the data:

```
    mag_arrow.axis = mag.norm()
    print(f"Magnetometer: {mag}")
```

7. Send this to the robot and run it with the usual VPython settings. You should see something like the following:

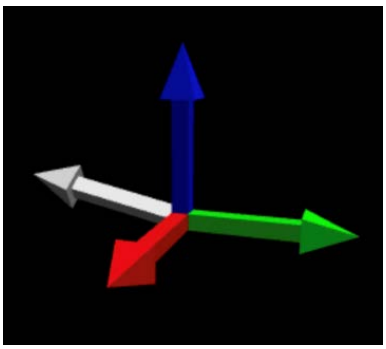


Figure 12.22 – The magnetometer's reading

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The preceding image shows a canvas with a red arrow for the X-axis pointing forward, a blue arrow for the Z-axis pointing up, and a green arrow for the Y-axis pointing right. There is a gray arrow showing the magnetometer vector (XZ only) pointing backward.

Yours may be pointing in a different direction compared to mine. This is because it is likely to be pointing to where the pin headers are on your IMU. Why is this?

Pin headers are usually made from magnetic metal. You can check this out for yourself by taking a magnet and seeing if it sticks to the headers (use some spares or do this when the power is off). You should also be able to observe what this does to the arrow. You could also take a bit of metal, such as a screwdriver, and wave it around the magnetometer. This should send the results all over the place.

Later, we will need to compensate for nearby metal as it may be creating a large offset, large enough to overwhelm Earth's relatively weak magnetic field completely.

Summary

In this chapter, you learned how to read four sensors on an inertial measurement unit, as well as how to display or graph data. You then had your first experience with soldering – a vital skill when it comes to making robots. You also learned about robot coordinate systems.

Later in this book, we will dive deeper into knitting the IMU sensors together to get an approximation of the robot's orientation.

In the next chapter, we will look at computer vision; that is, how to extract information from a camera and make the robot respond to what it can see.

Exercises

- In the temperature graph, you will notice a lot of noise in the graph and the output. The Python `round` function takes a number and the number of decimal places to keep, defaulting to 0. Use this to round off the temperature to a more reasonable value.
- Try putting the accelerometer values into an X, Y, and Z graph, as we did for the gyroscope. Observe the changes in the chart when you move the robot. Is it smooth, or is there noise?
- Could the gyroscope values be shown as a vector?
- Are there other sensors that can be soldered that you might find interesting for your robot to use?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- To learn more about VPython, take a look at the extensive help at `https://www.glowscript.org/docs/VPythonDocs/index.html`.
- Paul McWarter Arduino experiments with an IMU: `https://toptechboy.com/arduino-based-9-axis-inertial-measurement-unit-imu-based-on-bno055-sensor/`.
- Adafruit have guides on using IMUs with their libraries: `https://learn.adafruit.com/adafruit-sensorlab-magnetometer-calibration`.
- This YouTube talk by Google contains excellent information on the subject of *Sensor Fusion on Android Devices: A Revolution in Motion Processing*: `https://www.youtube.com/watch?v=C7JQ7Rpwn2k`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Section 3: Hearing and Seeing – Giving a Robot Intelligent Sensors

In this section, you will use OpenCV and Mycroft along with a smartphone to make a robot feel intelligent and interactive.

This part of the book comprises the following chapters:

- *Chapter 13, Robot Vision – Using a Pi Camera and OpenCV*
- *Chapter 14, Line-Following with a Camera in Python*
- *Chapter 15, Voice Communication with a Robot Using Mycroft*
- *Chapter 16, Diving Deeper with the IMU*
- *Chapter 17, Controlling the Robot with a Phone and Python*

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

13
Robot Vision – Using a Pi Camera and OpenCV

Giving a robot the ability to see things allows it to behave in ways to which humans relate well. Computer vision is still actively being researched, but some of the basics are already available for use in our code, with a Pi Camera and a little work.

In this chapter, we will use the robot and camera to drive to objects and follow faces with our pan-and-tilt mechanism. We'll be using the PID algorithm some more and streaming camera output to a web page, giving you a way to see what your robot is seeing.

The following topics will be covered in this chapter:

- Setting up the Raspberry Pi camera
- Setting up computer vision software
- Building a Raspberry Pi camera stream app
- Running background tasks when streaming
- Following colored objects with Python
- Tracking faces with Python

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Technical requirements

For this chapter, you will need the following:

- The robot with the pan-and-tilt mechanism from *Chapter 11*, *Programming Encoders with Python*.
- Code for the robot up to *Chapter 11*, *Programming Encoders with Python*, which you can download from GitHub at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter11`. We will be extending and modifying this for new functionality.
- A Raspberry Pi camera.
- A 300 mm-long Pi Camera cable, as the cable included with the camera is too short. Be sure that the cable is not for a Pi Zero (which has different connectors).
- Two M2 bolts and an M2 nut.
- A small square of thin cardboard—a cereal box will do.
- A small jeweler's screwdriver.
- A pencil.
- A kids' bowling set—the type with differently colored pins (plain, with no pictures).
- A well-lit space for the robot to drive in.
- Internet access.

The code for this chapter is on GitHub, available at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter13`.

Check out the following video to see the Code in Action: `https://bit.ly/39xfDJ9`

Setting up the Raspberry Pi camera

Before we can get into computer vision, we need to prepare the camera on your robot. There is hardware installation and software installation involved.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

When we have completed this installation, our robot block diagram will look like *Figure 13.1*:

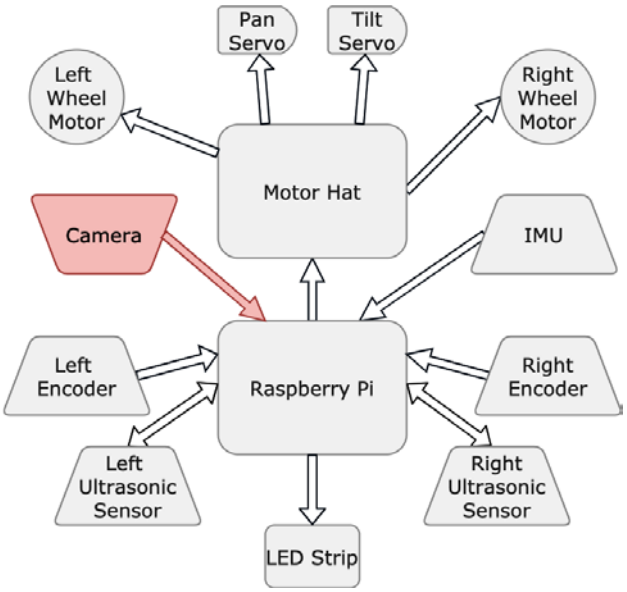


Figure 13.1 – Our robot block diagram with the camera added

Figure 13.1 continues the block diagrams we have shown throughout the book, with the camera's addition and its connection to the Raspberry Pi highlighted on the left.

We will first attach the camera to the pan-and-tilt assembly. We can then use a longer cable to wire the camera into the Pi. Let's start preparing the camera to be attached.

Attaching the camera to the pan-and-tilt mechanism

In *Chapter 10*, *Using Python to Control Servo Motors*, you added a pan-and-tilt mechanism to your robot. You will mount the camera onto the front plate of this mechanism. There are brackets and kits, but they are not universally available. Feel free to use one of these if you can adapt it to the pan-and-tilt mechanism; if not, I have a few plans.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Building a robot requires creative thinking and being adaptable, as well as the necessary technical skills. I frequently look through the materials I have for possible solutions before I go and buy something. Sometimes, the first thing you attempt will not work, and you'll need a plan B. My plan A was to use a hook-and-loop fastener (such as Velcro) stuck directly to the camera, but it does not adhere well to the back of the camera. So I had to move to plan B, that is, using a square of cardboard, making holes for 2 mm screws in it, bolting the camera to the cardboard, and then using the hook-and-loop fastener to attach the camera assembly to the Pi. Another possibility is to drill additional holes in the pan-and-tilt mechanism to line up with the camera screw holes.

> **Tip**
>
> Could I glue this? Yes, like most of our robot build, some glue—even crazy glue—could be used to adhere the camera to the pan and tilt. It would probably be an easier build. However, I can easily foresee that you would need to replace or remove the camera at some point. Reasons for that might be to reverse the camera cable or swap the camera out for another sensor, or even a newer camera with better features. It is for this reason that I generally avoid glue in my robot builds, looking for modular and replaceable solutions.

The parts needed are shown in *Figure 13.2*:

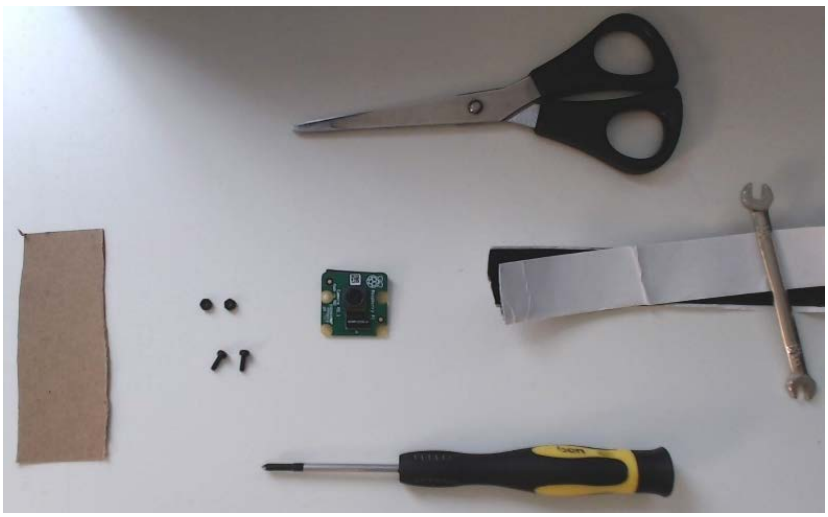


Figure 13.2 – The parts needed for our plan to fit the camera module

Figure 13.2 shows the tools and materials laid out: some thin card, 2 mm bolts and screws, the Pi Camera module, some scissors, a small spanner (or pliers), hook-and-loop tape, and a small screwdriver. You will also need a pencil.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

While making this, please try not to touch the camera's lens. So let's begin. The following figure shows you the steps to attach the camera:

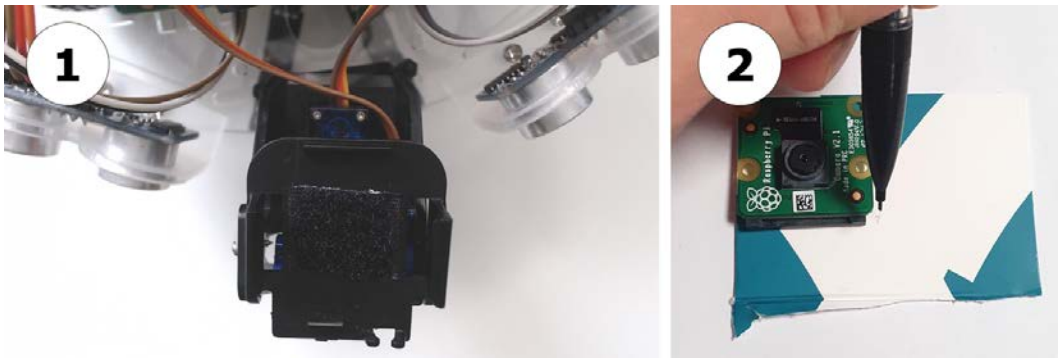


Figure 13.3 – Fitting the camera, steps 1–2

Here's how to use these parts to mount the camera:

1. First, cut a small amount for one side of the hook-and-loop fastener and adhere it to the pan-and-tilt mechanism, as shown in *Figure 13.3*.
2. Mark and cut out a small square of cardboard a little larger than the camera:

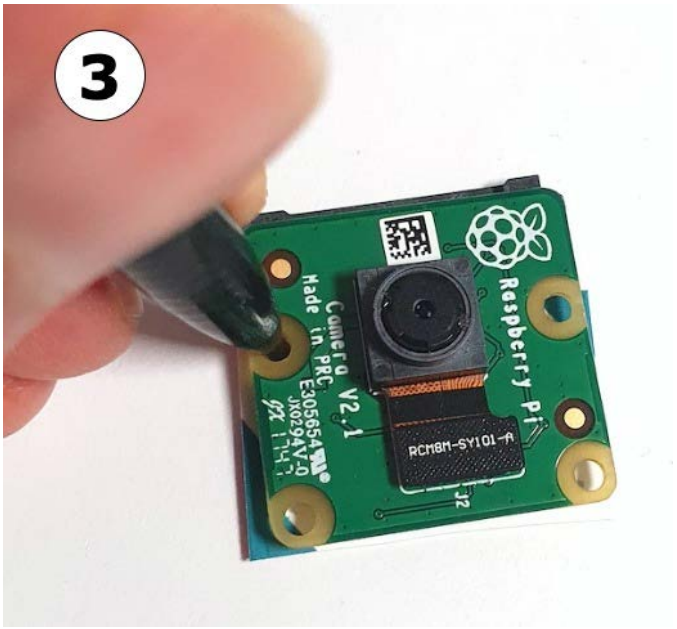


Figure 13.4 – Using a pen to mark the screw positions

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Then use a pen or pencil to poke through the camera screw holes to mark a dot, as shown in *Figure 13.4*. Then take a pointed tool (such as the point of a cross-headed jeweler's screwdriver or a math set compass), and on a firm surface, punch a hole where you made the mark:

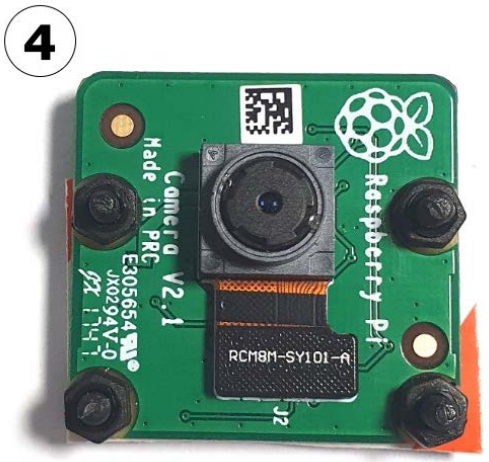


Figure 13.5 – Bolting the camera to the cardboard

4. Use a couple of M2 bolts and nuts to fasten the camera onto the cardboard carrier, as shown in *Figure 13.5*. Note that the bolt-facing side is at the back—this is so any protruding threads won't interfere with the hook and loop:

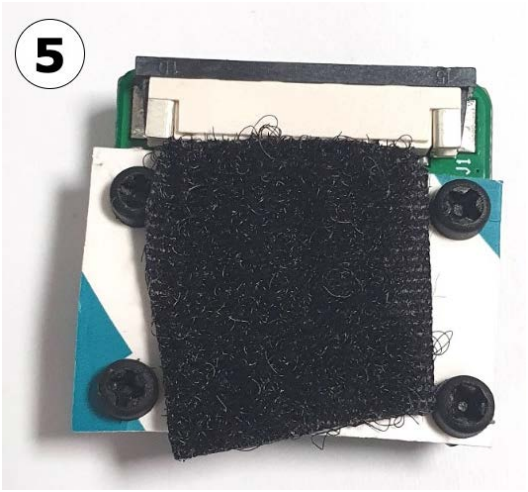


Figure 13.6 – The back of the cardboard/camera assembly with our hook-and-loop fastener

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Now cut a small amount of the hook-and-loop fabric, to which the material on the pan-and-tilt mechanism will fasten, and stick it to the back of the cardboard, as shown in *Figure 13.6*.

Note that the camera may have a film covering the lens—please remove this.

The camera is ready to be stuck to the robot. Don't attach the camera just yet, as we need to wire in the cable first. Let's see how in the next section.

Wiring in the camera

With the camera ready to attach, we'll need to use the Raspberry Pi camera cable to connect it to the Pi. We'll need to move some parts to get to the Raspberry Pi connector and thread the ribbon connector through.

The sequence of images in *Figure 13.7* shows how we will wire this:

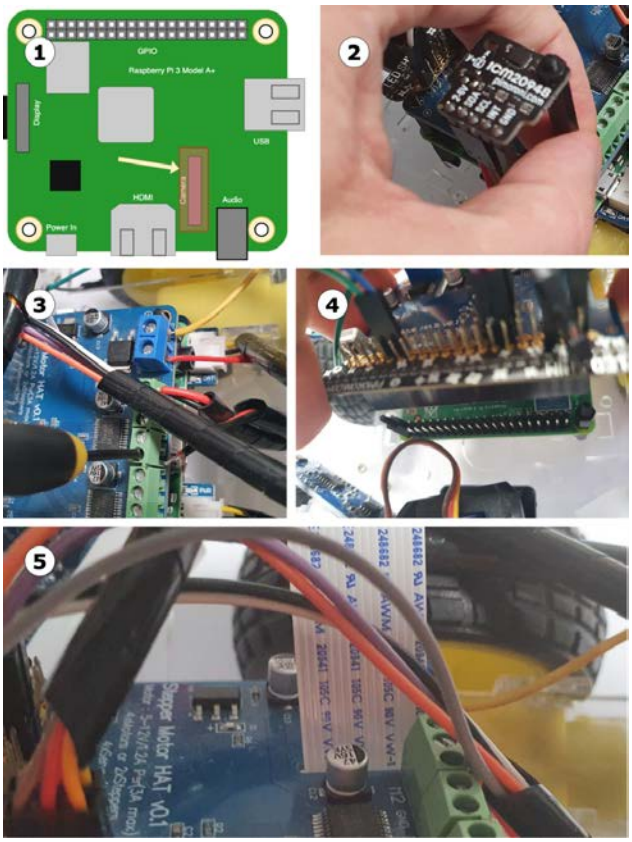


Figure 13.7 – The camera connector slot and the motor board

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The steps in *Figure 13.7* show how we'll prepare the cable connector:

1. The Raspberry Pi has a slot specifically for the camera—the camera cable fits into this. We will be wiring our camera into this slot, but the motor board covers the slot on our robot.
2. To get around the slot being covered, we will need to lift the other boards above the Pi. You'll temporarily need to unbolt the **Inertial Measurement Unit** (**IMU**), so the motor board isn't covered by it. Loosen the nut on top of the IMU; then you can turn the lower spacer post by hand to remove the IMU, leaving the IMU and standoff assembly complete.
3. Disconnect the motor wires (note how you'd previously connected them, or take a photo for later reference).
4. Now gently lift the motor board off the Raspberry Pi.
5. When you connect the camera to the Pi, the long cable will need to pass through the motor board. Keep this in mind as you perform the next step.

I recommend following *Connect ribbon cable to camera* in *The Official Raspberry Pi Camera Guide* (`https://magpi.raspberrypi.org/books/camera-guide`) for attaching the camera using the long 300 mm cable. After following the guide, you should have the ribbon installed the correct way around in the camera, then going through the slot in the motor board and into the port the right way around on the Raspberry Pi.

Double-checking that your connections are the right way around before replacing the motor board will save you a lot of time.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

To complete the reassembly, take a look at *Figure 13.8*:

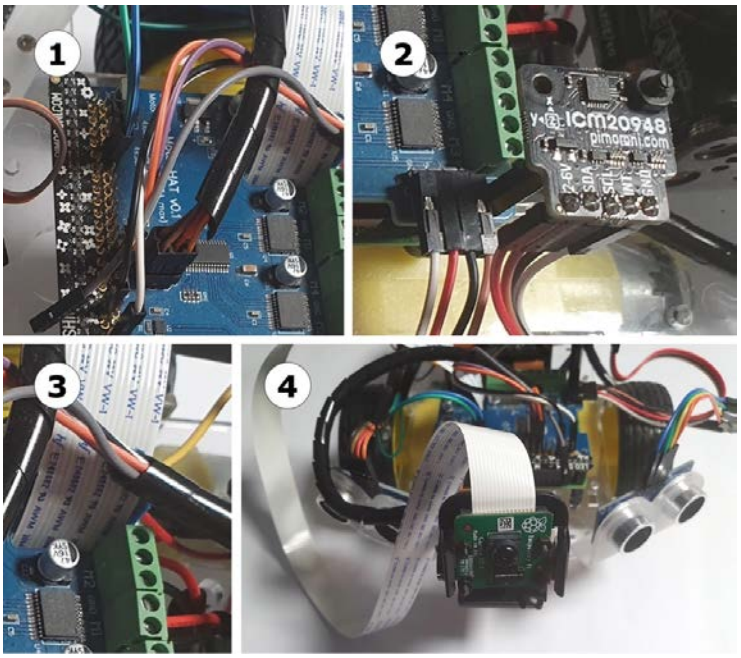


Figure 13.8 – Completing the camera interface

Follow these steps, using *Figure 13.8* as a reference:

1. Gently replace the motor board, pushing its header down onto the Raspberry Pi GPIO header and the holes onto spacers.
2. Bolt the IMU back in.
3. Reconnect the motor cables based on your reference.
4. Push the camera onto the hook-and-loop attachment on the pan-and-tilt head, with the cable facing upward.

You've seen how to wire in Raspberry Pi cameras. This camera is now wired and ready to use. Next, we will start preparing the software to get images from the camera.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Setting up computer vision software

Before we can start writing code, we'll need to set up drivers, tools, and libraries to interact with the camera and software to assist with computer vision.

In this section, we will activate the camera in Raspberry Pi OS Raspberry Pi OS and get a test picture. Then we will add the libraries to start interacting with the camera for visual processing.

We will then build our first app with the tool to demonstrate that the parts are in place and give us a starting point for the behaviors. Let's get into setting up the software.

Setting up the Pi Camera software

So that the camera is ready to use, we need to enable it:

1. Power up the Pi on external power (that is, plugged into a USB wall adapter) for this operation, leaving the motors powered down for now.
2. Log in via SSH. At the terminal, type the following:

```
pi@myrobot:~ $ sudo raspi-config
```

3. You should now see `raspi-config`. Select the **Interfacing Options** menu item by using the cursor keys and *Enter*.
4. Select **P1 camera**. `raspi-config` will then ask whether you would like the camera interface to be enabled. Select **Yes** and **Ok**, then **Finish**.
5. You will need to reboot for this to take effect:

```
pi@myrobot:~ $ sudo reboot
```

To verify that we can get pictures, we'll need the `picamera` package. At the time of writing, there is a copy of `picamera` already installed in Raspberry Pi OS.

Now that the camera is enabled, let's try getting our first picture.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting a picture from the Raspberry Pi

The first thing we need to do, to confirm that our setup was successful, is to ask the Pi Camera to take a picture. If the camera isn't detected, please go back and check that the cable connection is correct, that you have installed `picamera`, and that you have enabled the Raspberry Pi camera in `raspi-config`:

1. Reconnect to the Raspberry Pi and type the following to get a picture:

```
pi@myrobot:~ $ raspistill -o test.jpg
```

   The `raspistill` command takes a still image, and the `-o` parameter tells it to store that image in `test.jpg`. This command may take a while; taking a still can be slow if light conditions are poor.

2. You can then use your SFTP client (which we set up in *Chapter 4*, *Preparing a Headless Raspberry Pi for a Robot*) to download this image and verify it on your computer. You will notice that the picture is upside down, due to how the camera is mounted. Don't worry—we will correct this with our software.

With a picture taken, you know that the camera works. Now we can install the rest of the software needed to use the camera in visual processing applications.

Installing OpenCV and support libraries

We will need a few helper libraries to do the heavy lifting of visual processing and display the output in a useful way. **Open Computer Vision** (**OpenCV**) is a library with a collection of tools for manipulating pictures and extracting information. Code can use these OpenCV tools together to make useful behaviors and pipelines for processing images.

To run our code on the Raspberry Pi, we will need to install the Python OpenCV library there:

1. OpenCV has some dependencies that are needed first:

```
pi@myrobot:~ $ sudo apt install -y libavcodec58
libilmbase23 libgtk-3-0 libatk1.0-0 libpango-1.0-0
libavutil56 libavformat58 libjasper1 libopenexr23
libswscale5 libpangocairo-1.0-0 libtiff5 libcairo2
libwebp6 libgdk-pixbuf2.0-0 libcairo-gobject2 libhdf5-dev
pi@myrobot:~ $ sudo pip3 install "opencv_python_
headless<4.5" "opencv_contrib_python_headless<4.5"
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Raspberry Pi OS requires a library to be identified for OpenCV to work. This line identifies the library every time you log in to the Pi. We should also prepare it for this session:

```
pi@myrobot:~ $ echo export LD_PRELOAD=/usr/lib/arm-linux-
gnueabihf/libatomic.so.1 >>.bashrc
pi@myrobot:~ $ source .bashrc
```

3. **Flask** is a library for creating web servers that we'll use to stream the video data to a browser:

```
pi@myrobot:~ $ sudo pip3 install flask
```

4. **NumPy**, the numeric Python library, is excellent for the manipulation of large blocks of numbers. An image stored on a computer is essentially a large block of numbers, with each tiny dot having similar content to the three-color numbers we sent to the LEDs in *Chapter 9*, *Programming RGB LED Strips in Python*:

```
pi@myrobot:~ $ sudo apt install -y libgfortran5
libatlas3-base
pi@myrobot:~ $ sudo pip3 install numpy
```

5. We will need to install the large array extension for `picamera`. This will help us convert it's data for use in NumPy and OpenCV:

```
pi@myrobot:~ $ sudo pip3 install picamera[array]
```

We will continue testing on external power for the next few operations.

You've now prepared the software libraries and verified that the camera can take pictures. Next, we'll build an app to stream video from the camera to your browser.

Building a Raspberry Pi camera stream app

Downloading one picture at a time is fine, but we need to do things with those pictures on our robot. We also need a handy way to see what the robot is doing with the camera data. For that, we will learn how to use a Flask web server to serve up our pictures so we can view the output on a phone or laptop. We can use the core of this app to make a few different behaviors. We'll keep the base app around for them.

A video or video stream is a sequence of images, usually known as **frames**.

Let's design our streaming server.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Designing the OpenCV camera server

The diagram in *Figure 13.9* shows an image data pipeline, going from the camera, through the processing, and out to our web browser:

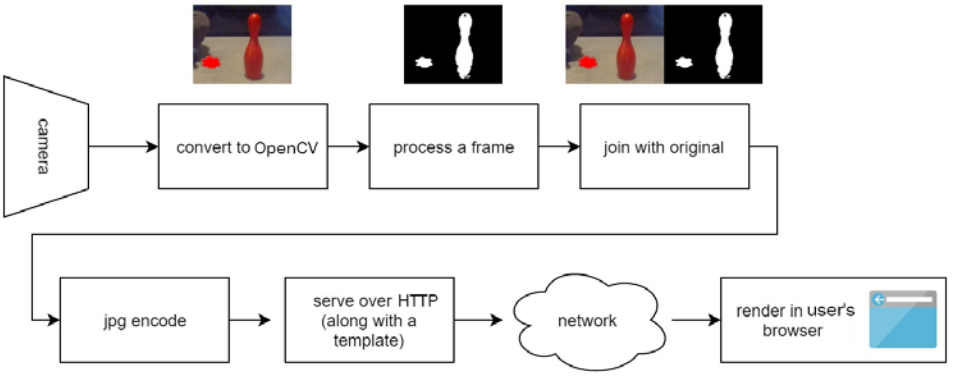


Figure 13.9 – The image server app

The image server app in *Figure 13.9* starts with the camera. The camera feeds image data to a **convert to OpenCV** step, with the raw photo given. Image data needs some processing for OpenCV to be able to manipulate it.

convert to OpenCV feeds data to **process a frame**, which can be anything we require; for this example, we'll apply a color mask, which we explore in more depth in the next section. Above the **process a frame** step is an example of an image after using a red color mask.

The raw frame and processed frame go into the next step, **join with original**, which creates a compound image with both images. Above the step are the two images joined into a single longer frame.

The joined images go into the **jpg encode** step. We need to encode with `jpeg`, an image encoding that a browser can show, and importantly, display as a sequence of frames, a streaming movie.

The encoded data goes to **serve over HTTP**, getting the data into a system you can view with a web browser. It uses a template (some layout and text for the browser) to serve this.

The image output then goes from **serve over HTTP**, via the network, to the users, browser. Finally, the browser shows the image to the user. The browser could be on a laptop or a phone.

It's time to start building the code. We'll break it down into two major parts: first, a `CameraStream` object, which will send our frames to the second part of our code project, an `image_server.py` script.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Writing the CameraStream object

As part of our system, we will create a helper library to set up the camera and get data streams from it:

1. Start the `camera_stream.py` file with the following imports:

```
from picamera.array import PiRGBArray
from picamera import PiCamera
import numpy as np
import cv2
```

   These imports give us the `PiCamera` code needed to access our camera. `cv2` is OpenCV, the computer vision library used to process the images. Here, NumPy is *aliased*, or nicknamed, `np`.

2. The next few lines set up parameters for the capture size and image quality:

```
size = (320, 240)
encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), 90]
```

   We will keep the images we capture at a small resolution of 320 by 240—this means we are sending less data, so we will process less too, which will keep the system reasonably quick. Higher resolutions may also lead to more noise and edge defects, which require cleaning up with further filters. Converting the images to send to the browser will use the `encode` parameter.

3. Add a function to set up the camera:

```
def setup_camera():
    camera = PiCamera()
    camera.resolution = size
    camera.rotation = 180
    return camera
```

   After initializing the camera, we set its resolution to the size. I mentioned that the camera is the wrong way up, so we set its rotation to 180 degrees to turn the pictures around.

4. We will need a function to start capturing a stream of images (a video, but a frame at a time):

```
def start_stream(camera):
    image_storage = PiRGBArray(camera, size=size)
    cam_stream = camera.capture_continuous(image_storage,
format="bgr", use_video_port=True)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

To store our image, we need to make a `PiRGBArray` instance, a type for storing RGB images. We then set up the stream of data with `capture_continuous`, a `picamera` method to take photos repeatedly. We pass it to the image store and tell it to format the output data as `bgr` (blue, green, red), which is how OpenCV stores color data. The last parameter to this is `use_video_port`, which, when set to `true`, results in a reduction in image quality in exchange for faster production of frames.

5. We can loop through `cam_stream` for frames until we choose to stop. Python has a concept of **iterators**—data structures for sequences such as lists and generators. Generators are sequences that produce the next bit of data just in time for when it's needed:

```
    for raw_frame in cam_stream:
        yield raw_frame.array
        image_storage.truncate(0)
```

   This `for` loop is a generator. Every cycle will yield the raw `.array` from the frame that the stream captured. What this means is that a loop can use the output of the `start_stream` function, so when looped over, the code in this `for` loop will run just enough to produce one raw frame, then the next, and so on. Python generators are a way to construct processing pipelines.

   The last line of the loop calls `truncate` to reset `image_storage` ready to hold another image. `PiRGBArray` can store many images in sequence, but we only want the latest one. More than one image may have arrived while we were processing a frame, so we must throw them away.

6. The final thing we add to the `camera_stream.py` script is a function to encode an image as `jpeg` and then into bytes for sending, as shown here:

```
def get_encoded_bytes_for_frame(frame):
    result, encoded_image = cv2.imencode('.jpg', frame,
encode_param)
    return encoded_image.tostring()
```

We will use the `camera_stream` library for a few of our behaviors, giving us the ability to fetch and encode camera frames, both ready for input and encoded for display. With that ready, let's use it in a test app to serve frames in a browser.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Writing the image server main app

This part of the app will set up Flask, start our camera stream, and link them together. We will put this in a new script named `image_server.py`:

1. We need to import all of these components and set up a Flask app:

```
from flask import Flask, render_template, Response
import camera_stream
import time

app = Flask(__name__)
```

   We import a few services from Flask: the `Flask` app object, which handles routing; a way to render templates into output; and a way to make our web app response. We import the `camera_stream` library we've just made, and we import `time` so we can limit the frame rate to something sensible. After the imports, we create a Flask `app` object for us to register everything with.

2. Flask works in routes, which are links between an address you hit a web server at and a registered handler function. A matching address asked for at our server app will run the corresponding function. Let's set up the most basic route:

```
@app.route('/')
def index():
    return render_template('image_server.html')
```

   The `'/'` route will be the `index` page, what you get by default if you just land on the robot's app server. Our function renders a template, which we'll write in the next section.

3. Now we get to the tricky bit, the video feed. Although `camera_stream` does some of the encoding, we need to turn the frames into an HTTP stream of data, that is, data that your browser expects to be continuous. I'll put this in a `frame_generator` function, which we'll need to break down a little. Let's start by setting up the camera stream:

```
def frame_generator():
    camera = camera_stream.setup_camera()
    time.sleep(0.1)
```

   `time.sleep` is here because we need to let the camera warm up after turning it on. Otherwise, we may not get usable frames from it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Next, we need to loop over the frames from `camera_stream`:

```
    for frame in camera_stream.start_stream(camera):
        encoded_bytes =camera_stream.get_encoded_bytes_
for_frame(frame)
```

   This function is another Python generator looping over every frame coming from `start_stream`, encoding each frame to JPG.

5. To send the encoded frame bytes back to the browser, we use another generator with `yield`, so Flask considers this a multipart stream—a response made of multiple chunks of data, with parts deferred for later—which many frames of the same video would be. Note that HTTP content declarations prefix the encoded bytes:

```
        yield (b'--frame\r\n'
               b'Content-Type: image/jpeg\r\n\r\n' +
encoded_bytes + b'\r\n')
```

   We place `b` in front of this string to tell Python to treat this as raw bytes and not perform further encoding on the information. The `\r` and `\n` items are raw line-ending characters. That completes the `frame_generator` function.

6. The next function, named `display`, routes from Flask to a loopable stream of HTTP frames from `frame_generator`:

```
@app.route('/display')
def display():
    return Response(frame_generator(),
        mimetype='multipart/x-mixed-replace;
boundary=frame')
```

   The Flask `display` route generates a response from `frame_generator`. As that is a generator, Flask will keep consuming items from that generator and sending those parts to the browser.

   The response also specifies a content type with a boundary between items. This boundary must be a string of characters. We have used `frame`. The boundary must match in `mimetype` and the boundary (`--frame`) in the content (*step 5*).

7. Now we can just add the code to start Flask. I've put this app on port `5001`:

```
app.run(host="0.0.0.0", debug=True, port=5001)
```

The app is nearly ready, but we mentioned a template—let's use this to describe what will go on the web page with the camera stream.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Building a template

Flask makes web pages using HTML templates, which route functions render into the output, replacing some elements at runtime if necessary. Create a `templates` folder, then make a file in that folder named `image_server.html`:

1. Our template starts with the HTML tag, with a title and a level 1 heading:

```
<html>
    <head>
        <title>Robot Image Server</title>
    </head>
    <body>
        <h1>Robot Image Server</h1>
```

2. Now, we add the image link that will display the output of our server:

```
        <img src="{{ url_for('display') }}">
```

   Note `url_for` here. Flask can use a template renderer, Jinja, to insert the URL from a route in Flask by its function name.

3. Finally, we just close the tags in the template:

```
    </body>
</html>
```

We can serve this template up in our main server app.

Now we can upload all three of these parts, ensuring that you upload the template into the `templates` directory on the Pi.

With the server code and templates ready, you should be able to run the image server.

Running the server

Start the app with `python3 image_server.py`.

Point your browser at the app by going to `http://myrobot.local:5001` (or your robot's address), and you should see a video served, as shown in *Figure 13.10*:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 13.10 – Screen capture of the robot image server

The screenshot in *Figure 13.10* shows our robot image server output in a browser. The top shows the browser search bar, with the `myrobot.local:5001` address in it. Below this is the **Robot Image Server** heading from the template. Below the heading is an image capture of a kids' red bowling pin taken from a robot camera—served up with the video stream code.

Troubleshooting

If you have problems running the server and seeing the picture, try the following steps:

- If you see errors while running the code, do the following:

  a) Ensure you can capture images with raspistill.

  b) Ensure you have installed all the required dependencies.

  c) If it's about `libatomic`, please ensure that you have performed the previous `LD_PRELOAD` exports.

  d) Check that the code is correct.
- If the image is black, check your lighting. The Raspberry Pi camera is susceptible to light conditions and needs a well-lit space to operate. Note that none of the following tracking will work if the camera is not getting enough light.
- Expect the rate to be slow—this is not a fast or high-quality capture.

Now you can stream images from a Raspberry Pi into a browser. Next, we will add a background worker task and control mechanism to the app, as this whole server depends on the slow browser request cycle.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Running background tasks when streaming

Our image service works but has a significant flaw. Currently it will wait between requests before taking each action, but what if we want our robot to be doing something? To do this, we need to be able to run a behavior in parallel with the server. That behavior and the server both need access to the image data.

We will approach this by making the Flask web app a secondary process, with the behavior as the primary process for the robot when it is running. Python has a handy tool for precisely this kind of structure, called multiprocessing. Find out more at `https://docs.python.org/3/library/multiprocessing.html`.

Communicating between multiple processes is tricky. If two processes try to access (read or write) the same data simultaneously, the results can be unpredictable and cause strange behavior. So, to save them trying to access data simultaneously, we will use the multiprocessing queue object. A queue allows one process to put data in at one end and another process to consume it safely at the other—it is a one-way flow of information. We will use one queue to send images to the server and another to get control data from user interactions in the browser.

The diagram in *Figure 13.11* shows the way data will flow through these behaviors:

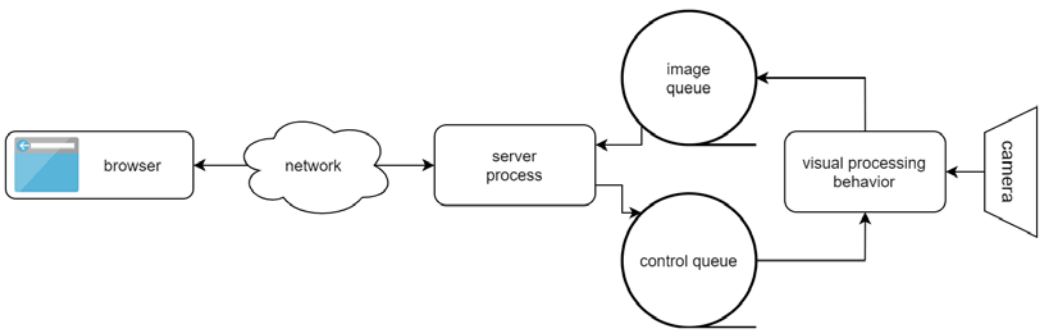


Figure 13.11 – Data flow between a browser, server process, and robot behavior

In *Figure 13.11*, we abridge some of the sections in *Figure 13.9*. First, there is data from the camera going into a visual processing behavior (for example, tracking an object). This behavior will output image frames to an image queue. The output will be the fully processed and joined image.

A server process, the web app, will take the images from the image queue to serve them to a browser via the network. However, the web app will also handle commands from user interaction in the browser. The app puts them in the control queue as messages. The visual processing behavior will read any messages from the control queue and act on them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

A few caveats: the visual processing behavior will only place images in the image queue when it's empty, so the queue will only ever contain one image. Allowing only one prevents the visual processing behavior from trying to overwrite an image in shared memory when a server tries to output it. The control queue has no such restriction; we'll just expect that user interactions will not produce control messages faster than the behavior loop can consume them.

We will separate the web app as a core and then write a behavior based on it. We can use the web app core multiple times. Let's write this code.

Writing a web app core

In this design, the web app core will handle setting up the queues, running the server process, and the Flask-based routing. We will write the library in Flask style, using plain Python functions in a module.

As an interface to the core, our other behaviors will be able to do the following:

- `start_server_process(template_name)` will start the web app server, using the named template.
- `put_output_image(encoded_bytes)` will put images into the display queue.
- `get_control_instruction()` is used to check and return instructions from the control queue. This function returns a dictionary of instruction data.

The Flask/web server part of the app is slightly independent of the behavior, allowing the user to *tune in* to see its display, but it should not stop the app running when a user is not present or a browser stalls:

1. Let's start with some imports. We'll put this code in `image_app_core.py`:

```
import time
from multiprocessing import Process, Queue

from flask import Flask, render_template, Response
```

   We import `Queue` and `Process` to create the process and communicate with it. We then use the same imports for Flask that we used previously. Note—we are *not* importing any of the camera parts in this module.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Next, we define our Flask app and the queues. We only really want one frame queued, but we put in one in case of hiccups while transmitting—although we can check whether a `Queue` instance is empty, this is not 100% reliable, and we don't want one part of the app waiting for the other:

```
app = Flask(__name__)
control_queue = Queue()
display_queue = Queue(maxsize=2)
```

3. We will also define a global `display_template` here, in which we'll store the main app template:

```
display_template = 'image_server.html'
```

4. Now we add routes for this Flask app. The index route is only different in that it uses `display_template`:

```
@app.route('/')
def index():
    return render_template(display_template)
```

5. Next, we will create the loop for getting frames: a modified version of `frame_generator`. This function is our main video feed. So that it doesn't *spin* (that is, run very quickly in a tight loop), we put in a sleep of 0.05 to limit the frame rate to 20 frames per second:

```
def frame_generator():
    while True:
        time.sleep(0.05)
```

6. After the sleep, we should try to get data from `display_queue` (we'll put frames into the queue later). Like we did in `image_server`, this loop also turns our data into multi-part data:

```
        encoded_bytes = display_queue.get()
        yield (b'--frame\r\n'
               b'Content-Type: image/jpeg\r\n\r\n' +
encoded_bytes + b'\r\n')
```

7. Now make that available through a display block:

```
@app.route('/display')
def display():
    return Response(frame_generator(),
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        mimetype='multipart/x-mixed-replace;
boundary=frame')
```

8. We need a way to post control messages to our app. The `control` route accepts these, takes their form data (a dictionary with instructions), and uses `control_queue.put` to pass that along to the robot behavior:

```
@app.route('/control', methods=['POST'])
def control():
    control_queue.put(request.form)
    return Response('queued')
```

9. That gives us all the core internals, but we also need to start the server process. The part of the app from earlier that started our server, we've now put into a function named `start_server_process`:

```
def start_server_process(template_name):
    global display_template
    display_template = template_name
    server = Process(target=app.run, kwargs={"host":
"0.0.0.0", "port": 5001})
    server.start()
    return server
```

   We intend for a behavior to start this function, passing in a custom template name. It stores `template_name` in the global `display_template`. The preceding `index` route uses the template. Instead of calling `app.run`, we create a `Process` object. The `Process` parameter `target` is a function to run (`app.run`), and some parameters need to be given to that function (the host and port settings). We then start the server process and return the process handle so our code can stop it later.

10. The next interface task is putting an image into the queue we created in *step 1*. To ensure that we don't run up a lot of memory, we only intend the queue to have a length of one. That means that the first frame will be stale, but the next frame will arrive soon enough for it not to affect the user:

```
def put_output_image(encoded_bytes):
    if display_queue.empty():
        display_queue.put(encoded_bytes)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. Finally, for this interface, we need a function to get the control messages out. This function will not wait and will return a message if there is one or `None` for *no message*:

```
def get_control_instruction():
    if control_queue.empty():
        return None
    else:
        return control_queue.get()
```

The `image_app_core.py` file establishes a controllable base for us to build visual processing robot behaviors with, or indeed any behavior with a web interface, control instructions, an output stream, and background process. Next, let's test this core with a simple behavior.

Making a behavior controllable

We can try out our core with a behavior that sends images to the web service and accepts a simple `exit` control message:

1. Let's make a new file called `control_image_behavior.py`, starting with imports for the `image_app_core` interface and `camera_stream`:

```
import time

from image_app_core import start_server_process, get_
control_instruction, put_output_image
import camera_stream
```

2. We then add a function that runs our simple behavior with the main loop. I've broken this function down as it's a little complicated. First, we'll set up the camera and use a sleep to give the camera warm-up time:

```
def controlled_image_server_behavior():
    camera = camera_stream.setup_camera()
    time.sleep(0.1)
```

3. Next, we get frames from a camera stream in a `for` loop and put those as encoded bytes on the output queue:

```
    for frame in camera_stream.start_stream(camera):
        encoded_bytes = camera_stream.get_encoded_bytes_
for_frame(frame)
        put_output_image(encoded_bytes)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. While still in the loop, we will try accepting a control instruction to exit. Normally the instruction will be `None`, signalling there are no control instructions waiting. But if we have a message, we should match the command in it to exit:

```
            instruction = get_control_instruction()
            if instruction and instruction['command'] ==
    "exit":
                print("Stopping")
                return
```

   This handler uses `return` to stop the behavior when it receives the `exit` instruction from the control queue.

5. We then need to start the server and start our behavior. We always want to stop the web server process. By surrounding the behavior with `try` and `finally`, it will *always* run anything in the `finally` part, in this case, making sure the process is terminated (stopped):

```
process = start_server_process('control_image_behavior.
html')
try:
    controlled_image_server_behavior()
finally:
    process.terminate()
```

We now have a simple controllable behavior; however, it mentions the `control_image_behavior.html` template. We need to provide that.

Making the control template

This template, in `templates/control_image_behavior.html`, is the same as the one before, but with two important differences, shown here in bold:

```
<html>
    <head>
        <script src="https://code.jquery.com/jquery-3.3.1.min.
js"></script>
        <title>Robot Image Server</title>
    </head>
    <body>
        <h1>Robot Image Server</h1>
        <img src="{{ url_for('display') }}"><br>
        <a href="#" onclick="$.post('/control', {'command':
'exit'}); ">Exit</a>
    </body>
</html>
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The differences are as follows:

- In this template, we load a library in our browser called `jquery`, which is handy for interactive web pages. There is great documentation for jQuery at `https://api.jquery.com/`.
- We have the image and header that we saw before, but new to this code is an `a` tag (for anchor), which when clicked will post the `exit` command to the `'/control'` route on our web app. `<br>` just creates a line break to show the exit link below the image.

If you wanted to run this where internet access is difficult, you would need the server to serve the `jquery` library. This template tells the browser to download `jquery` directly from the internet.

Now we have the components, we should try running our controllable behavior.

Running the controllable image server

Now we have the components, let's get this running and try out the commands:

1. To run the image server, you need to upload all three files:

   a) `image_app_core.py`

   b) `control_image_behavior.py`

   c) `templates/control_image_behavior.html`.
2. On your Pi, use `python3 control_image_behavior.py` to start the process.
3. Point your browser at `http://myrobot.local:5001` (or the address of your robot). You will see the pictures again.
4. If you click on the **Exit** link below the image, this will send a control instruction to your app, which should gracefully quit.

You've now seen how to get image data from a behavior while sending control data back to the behavior. With the control and streaming technique tested and ready, and a framework to use for it, we can build a more interesting behavior. In the next section, we'll make the robot follow an object with a specific color.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Following colored objects with Python

Now we have some basics ready; we can use this to build some more interesting behaviors.

We will create a behavior that will chase, but not get too close to, a colored object. This behavior will make the robot seem very intelligent. We will revisit color models, covered in *Chapter 9*, *Programming RGB Strips in Python*. We'll add color masking and filtering and use the OpenCV contours tools to detect the largest blob of color in an image and point the robot at it.

Building the color-chasing behavior requires a few steps. Let's start with a diagram showing an overview of this whole behavior in *Figure 13.12*:

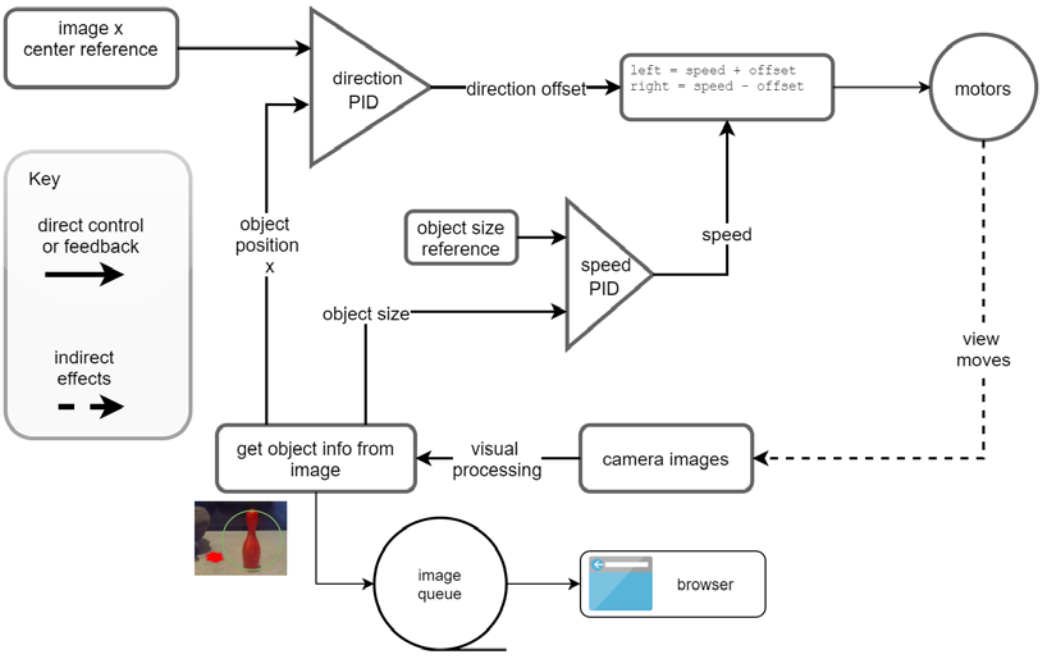


Figure 13.12 – The color-tracking behavior

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The flow of data in *Figure 13.12* starts from **camera images**. These go through **visual processing** to **get object info from image**. **get object info from image** outputs the object's size (based on the radius of a circle around it) and the object's position (the middle of the enclosing circle) and puts frames on the image queue for the web app/browser.

The object size goes into a speed **Proportional Integral Derivative** (**PID**) controller, which also has an object size reference as its set point. Depending on the difference between the expected size and actual size, this PID will output a speed for the motors, optimizing the radius to be the same as the reference size. That way, the robot will maintain a distance from an object of a known size. This is a base speed for both motors.

The object position has an `x` component and a `y` component. This behavior will turn to center the object, so we are interested in the `x` coordinate. The `x` coordinate goes into a PID for controlling the direction/heading. This PID takes a reference position—the center of the camera viewport. This direction PID will produce an output to try and get the difference between these coordinates to zero. By adding to one motor's speed and reducing the other's speed, the robot will turn to face the object (or, if you swap them for fun, it'll turn away!).

The images are sent, via an image queue using the app core, to the browser. A detail not shown in the diagram is the control queue with messages to start the motors, stop the motors, and exit the behavior.

The final part of this system, and probably the most interesting, is the color tracking. The box labeled **get object info from image** performs the tracking. Let's see how that works next.

Turning a picture into information

We are using colored pins from a kids' bowling set. They come in nice, bright, primary colors. I will use green as an example. We start with just a picture. However, a set of transformations to the data is needed to turn the picture into information the robot can use to make decisions.

A pipeline is a good way to design a set of transformations. Let's look at the color tracking as an image processing pipeline in *Figure 13.13*:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

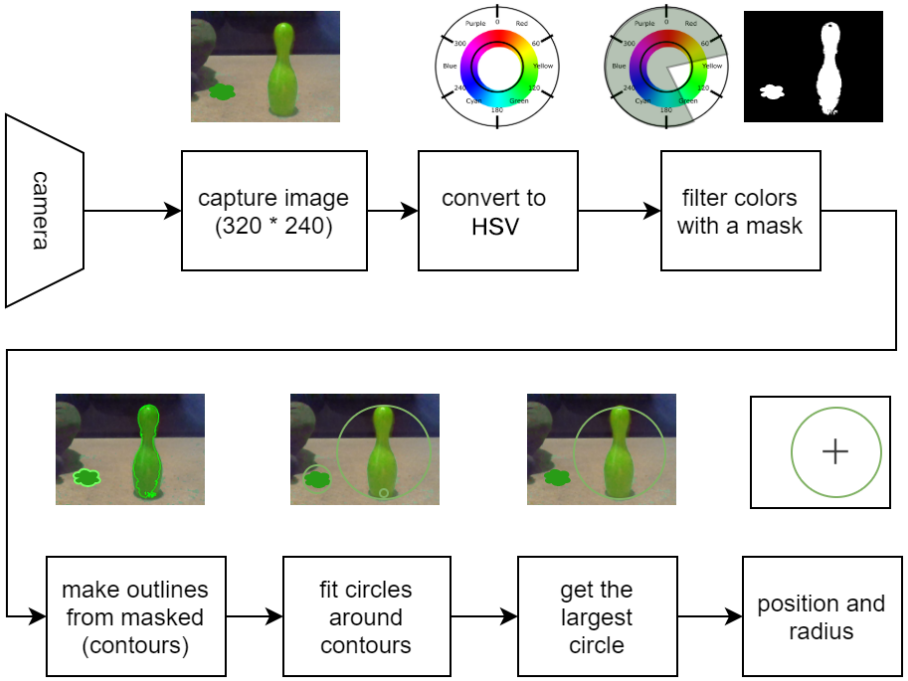


Figure 13.13 – Getting color object information from a camera

As with other pipelines, *Figure 13.13* starts from the camera. This is converted to a low resolution to keep things fast. The figure shows a camera image above the step.

The process converts the output from the image capture to HSV, the colorspace we mentioned in *Chapter 9*, *Programming RGB Strips in Python*. We use HSV because it means the process can filter colors in a specific range of hues, by their light (very dark objects may confuse us), and by saturation, so it won't include almost-gray objects. RGB (or BGR) images are tricky to filter, as getting the different light and saturation levels of a particular hue (say, the blues) is not viable. The figure shows the hue color wheel above this step.

OpenCV has a function, `cv2.cvtColor`, to convert whole images between colorspaces. Note that OpenCV uses 0–179 for the hue range, instead of 0–359. This is so it fits in a byte (0–255), but you can convert hue values by simply dividing by 2 if you know the value you want.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

After converting to HSV, we then filter the colors in the image with a mask, highlighting pixels in a particular range. It will output white if the object is in the range, or black if it's not. Above this step, the unshaded region on the hue color wheel shows the range, with the masked output next to it. There is a function in OpenCV to do this: `cv2.inRange`. This gives us a very easy binary output, a masked image, to draw around for our system.

Our pipeline then uses the contours system to draw around our masked image. The contour specifies only the boundary points of our object. OpenCV provides a `cv2.findContours` function to do exactly this, which returns a list of shapes, each defined by its outlines. The preceding figure shows the contours (taken from the mask) drawn onto the raw image. Note how light and shade have made the bottom of the bowling pin a bit rough as it doesn't quite fit the mask.

The processing pipeline then takes the contours (outlines) and uses `cv2.minEnclosingCircle` to draw circles around them. We will then have some circles, described by a center `x`, `y` coordinate, and radius. The preceding figure shows these circles projected on the raw image.

Our object may have highlights, producing more than one circle, and other objects may also produce smaller circles. We are only interested in one, the largest of these, so we can loop through the circles, and keep only the largest. Above the **get the largest circle** step is the raw image with only the largest circle drawn.

This largest circle's coordinates and radius give us enough information for our robot to start chasing an object. Above this last step is just the circle, with crosshairs showing its position.

> **Important note**
>
> A caveat about red objects: we will use green because red is slightly tricky, as it requires two masks. The hues for red cross a boundary between 179 (the upper limit of our hue range) and 0 (the lower limit), so we would have to mask the image twice and then combine these with an `or` operation. You could use the `cv2.bitwise_or` function to try masking red.

Now we have examined how the pipeline will work and its caveats. We've seen how this pipeline will fit with PID controllers to create an interesting behavior. Let's build this code.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Enhancing the PID controller

We are going to be using more PID controllers. We still don't require the differential component, but we will develop an issue with our integral component building up while the motors take time to move. The integral has a sum that starts to grow if there is a constant error. It is good to correct for that error but it can result in a large overshoot. This overshoot, due to the integral still growing after the robot has started to react, albeit slowly, is called **integral windup**.

We can prevent this sum from getting too large by introducing a windup limit to our PID:

1. Open up the `pid_controller.py` file and make the changes in bold in the following snippet. First, add the `windup_limit` parameter, which defaults to `None` if you don't set a limit:

```
class PIController(object):
    def __init__(self, proportional_constant=0, integral_
constant=0, windup_limit=None):
        self.proportional_constant = proportional_
constant
        self.integral_constant = integral_constant
        self.windup_limit = windup_limit
        self.integral_sum = 0
```

2. We want to prevent our integral growth if we have a limit and hit it. Our integral will change if any of the following occurs:

   a) There is no windup limit (you set it to `None`).

   b) The absolute value of the sum is below the windup limit.

   c) The sign of the error would reduce the sum (by being opposed to it).

   This prevents us from going above the limit if there is one.

   Let's see this in code—this code will replace the previous `handle_integral` method:

```
    def handle_integral(self, error):
        if self.windup_limit is None or \
                (abs(self.integral_sum) < self.windup_
limit) or \
                ((error > 0) != (self.integral_sum > 0)):
            self.integral_sum += error
        return self.integral_constant * self.integral_sum
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We can `start` and `stop` this behavior from the web page. If we start moving again, we won't want the PIDs to carry old values. Let's add a `reset` function to zero out the integral sum:

```
    def reset(self):
        self.integral_sum = 0
```

The PID controller is now able to reset and has a windup limit to stop big overshoots. Let's build the other behavior components that use it.

Writing the behavior components

This behavior has two files—a template to pass to our app core with the control buttons, and then the main behavior code. Let's start by writing the template.

Writing the control template

This template is for the stream app, with some different controls:

1. Copy the template from `templates/control_image_behavior.html` to `templates/color_track_behavior.html`.
2. We will add two further controls to this, `start` and `stop`, displayed here in bold:

```
        <img src="{{ url_for('display') }}"><br>
        <a href="#" onclick="$.post('/control',
{'command': 'start'});">Start</a>
        <a href="#" onclick="$.post('/control',
{'command': 'stop'})">Stop</a><br>
        <a href="#" onclick="$.post('/control',
{'command': 'exit'});">Exit</a>
```

   We intend to run the program with the robot stopped first, so we can tune in with our phone or browser, see what the robot is detecting, and click the **Start** button to get it moving.

With the template modified, we will need to write the behavior code next.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Writing the behavior code

We'll put this new behavior in a file called `color_track_behavior.py`:

1. It's no surprise that we start with the imports. Because we are bringing together many elements, there are quite a few, but we have seen them all before:

```
import time

from image_app_core import start_server_process, get_
control_instruction, put_output_image

import cv2
import numpy as np

import camera_stream
from pid_controller import PIController
from robot import Robot
```

2. Now, we add the `Behavior` class to find and get close to a colored object. We pass this the `robot` object:

```
class ColorTrackingBehavior:
    def __init__(self, robot):
        self.robot = robot
```

3. These values are intended to be tuned for the color mask and object size:

```
        self.low_range = (25, 70, 25)
        self.high_range = (80, 255, 255)
        self.correct_radius = 120
        self.center = 160
```

We use the `low_range` and `high_range` values for the color filter (as seen in *Figure 13.13*). Colors that lie between these HSV ranges would be white in the masked image. Our hue is 25 to 80, which correspond to 50 to 160 degrees on a hue wheel. Saturation is 70 to 255—any lower and we'd start to detect washed out or gray colors. Light is 25 (very dark) to 255 (fully lit).

The `correct_radius` value sets the size we intend to keep the object at and behaves as a distance setting. `center` should be half the horizontal resolution of the pictures we capture.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. The last member variable set here is `running`. This will be set to `True` when we want the robot to be moving. When set to `False`, the processing still occurs, but the motors and PIDs will stop:

```
        self.running = False
```

5. The next bit of code is to process any control instructions from the web app:

```
    def process_control(self):
        instruction = get_control_instruction()
        if instruction:
            command = instruction['command']
            if command == "start":
                self.running = True
            elif command == "stop":
                self.running = False
            if command == "exit":
                print("Stopping")
                exit()
```

This services the `start`, `stop`, and `exit` buttons. It uses the `running` variable to start or stop the robot moving.

6. Next, we have the code that will find an object from a frame. This implements the pipeline shown in *Figure 13.13*. We'll break this function down a bit, though:

```
    def find_object(self, original_frame):
        """Find the largest enclosing circle for all
contours in a masked image.
        Returns: the masked image, the object
coordinates, the object radius"""
```

Because this code is complex, we have a documentation string or **docstring** explaining what it does and what it returns.

7. Next, the method converts the frame to HSV, so it can be filtered using `inRange` to leave only the `masked` pixels from our frame:

```
        frame_hsv = cv2.cvtColor(original_frame, cv2.
COLOR_BGR2HSV)
        masked = cv2.inRange(frame_hsv, self.low_range,
self.high_range)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Now that we have the masked image, we can draw contours (outline points) around it:

```
        contours, _ = cv2.findContours(masked, cv2.RETR_
LIST, cv2.CHAIN_APPROX_SIMPLE)
```

   When you find contours, first you specify the image to find them in. You then state how the contours are retrieved; in our case, we've specified a simple list using `RETR_LIST`. OpenCV is capable of more detailed types, but they take more time to process.

   The last parameter is the method used to find the contours. We use the `CHAIN_APPROX_SIMPLE` method to simplify the outline to an approximate chain of points, such as four points for a rectangle. Note the _ in the return values; there is optionally a hierarchy returned here, but we neither want nor use it. The _ means ignore the hierarchy return value.

9. The next thing is to find all the enclosing circles for each contour. We use a tiny loop to do this. The `minEnclosingCircle` method gets the smallest circle that entirely encloses all points in a contour:

```
        circles = [cv2.minEnclosingCircle(cnt) for cnt in
contours]
```

   `cv2` returns each circle as a radius and coordinates—exactly what we want.

10. However, we only want the biggest one. Let's filter for it:

```
        largest = (0, 0), 0
        for (x, y), radius in circles:
            if radius > largest[1]:
                largest = (int(x), int(y)), int(radius)
```

    We store a `largest` value of `0`, and then we loop through the circles. If the circle has a radius larger than the circle we last stored, we replace the stored circle with the current circle. We also convert the values to `int` here, as `minEnclosingCircle` produces non-integer floating-point numbers.

11. We end this method by returning the masked image, the largest coordinates, and the largest radius:

```
        return masked, largest[0], largest[1]
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. Our next method will take an original frame and processed frame, then turn them into a dual-screen display (two images of the same scale joined together horizontally) on the output queue through to the web app:

```
    def make_display(self, frame, processed):
        display_frame = np.concatenate((frame,
processed), axis=1)
        encoded_bytes = camera_stream.get_encoded_bytes_
for_frame(display_frame)
        put_output_image(encoded_bytes)
```

The method uses the `np.concatenate` function to join the two images, which are equivalent to NumPy arrays. You could change the `axis` parameter to `0` if you wanted screens stacked vertically instead of horizontally.

13. The next method processes a frame of data through the preceding functions, finding the objects and setting the display. It then returns the object info as follows:

```
    def process_frame(self, frame):
        masked, coordinates, radius = self.find_
object(frame)
        processed = cv2.cvtColor(masked, cv2.COLOR_
GRAY2BGR)
        cv2.circle(frame, coordinates, radius, [255, 0,
0])
        self.make_display(frame, processed)
        return coordinates, radius
```

Note we use `cvtColor` to change the masked image to a three-channel image—the original frame and processed frame must use the same color system to join them into a display. We use `cv2.circle` to draw a circle around the tracked object on the original frame so we can see what our robot has tracked on the web app, too.

14. The next method is the actual behavior, turning the preceding coordinates and radius into robot movements. When we start our behavior, the pan-and-tilt mechanism may not be pointing straight forward. We should ensure that the mechanism is facing forward by setting both servos to `0`, then start the camera:

```
    def run(self):
        self.robot.set_pan(0)
        self.robot.set_tilt(0)
        camera = camera_stream.setup_camera()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

15. While the servos are moving and the camera is warming up, we can prepare the two PID controllers we need for speed (based on radius) and direction (based on distance from the horizontal middle):

```
        speed_pid = PIController(proportional_
constant=0.8,
            integral_constant=0.1, windup_limit=100)
        direction_pid = PIController(proportional_
constant=0.25,
            integral_constant=0.05, windup_limit=400)
```

    These values I arrived at through much tuning; you may find you need to tune these further. The *Tuning the PID controller settings* section will cover how to tune the PIDs.

16. Now we wait a little while for the camera and pan-and-tilt servos to settle, and then we turn off the servos in the center position:

```
        time.sleep(0.1)
        self.robot.servos.stop_all()
```

17. We let the user know, with a `print` statement, and output some debug headers:

```
        print("Setup Complete")
        print('Radius, Radius error, speed value,
direction error, direction value')
```

18. We can then enter the main loop. First, we get the processed data from the frame. Notice we use brackets to unpack `coordinates` into `x` and `y`:

```
        for frame in camera_stream.start_stream(camera):
            (x, y), radius = self.process_frame(frame)
```

19. We should check our control messages at this point. We then check whether we are allowed to move, or whether there is any object big enough to be worth looking for. If there is, we can start as follows:

```
            self.process_control()
            if self.running and radius > 20:
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

20. Now we know the robot should be moving, so let's calculate error values to feed the PID controllers. We get the size error and feed it into the speed PID to get speed values:

```
                radius_error = self.correct_radius -
radius
                 speed_value = speed_pid.get_value(radius_
error)
```

21. We use the center coordinate and current object, `x`, to calculate a direction error, feeding that into the direction PID:

```
                 direction_error = self.center - x
                 direction_value = direction_pid.get_
value(direction_error)
```

22. So we can debug this; we print a debug message here matching with the headers shown before:

```
                 print(f"{radius}, {radius_error}, {speed_
value:.2f}, {direction_error}, {direction_value:.2f}")
```

23. We can use the speed and direction values to produce left and right motor speeds:

```
                 self.robot.set_left(speed_value -
direction_value)
                 self.robot.set_right(speed_value +
direction_value)
```

24. We've handled what to do when the motors are running. If they are not, or there is no object worth examining, then we should stop the motors. If we have hit the **Stop** button, we should also reset the PIDs, so they do not accumulate odd values:

```
            else:
                self.robot.stop_motors()
                if not self.running:
                    speed_pid.reset()
                    direction_pid.reset()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

25. We have now finished that function and the `ColorTrackingBehavior` class. Now, all that is left is to set up our behavior and web app core, then start them:

```
print("Setting up")
behavior = ColorTrackingBehavior(Robot())
process = start_server_process('color_track_behavior.
html')
try:
    behavior.run()
finally:
    process.terminate()
```

This behavior code is built and ready to run. You've seen how to convert the image, then mask it for a particular color, and how to draw around the blobs in the mask, and then find the largest one. I've also shown you how to turn this visual processing into robot moving behavior by feeding this data through PIDs and using their output to control motor movements. Let's try it out!

Running the behavior

I'm sure you are keen to see this working and fix any problems that there are. Let's get into it:

1. To run this behavior, you will need to upload `color_track_behavior.py`, the modified `pid_controller.py` file, and the template at `templates/color_track_behavior.html`. I'll assume that you already have `robot.py` and the other supporting files uploaded.
2. Start the app with `python3 color_track_behavior.py`, which will start the web server and wait.
3. At this point, you should use your browser to connect to `http://myrobot.local:5001`, and you should be able to see your robot's image feed.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can see the object and its circle, along with links to control the robot, as shown in the screenshot in *Figure 13.14*:

Figure 13.14 – The color-tracking web app

Figure 13.14 shows a screenshot of our app server running the code to track a colored object. Under the address bar and heading is a dual-screen type output. The left has the direct feed from the camera, with a kids' green bowling pin close to the middle and a blue circle outlining the pin, generated by the behavior to show it's tracking the largest matching object. On the right is the mask's output, so we can see what aspects of the image match and tune the mask values if we need to. Under this are **Start**, **Stop**, and **Exit** links, to start the motors, stop the motors, and exit the program.

4. To make the robot start moving, press the **Start** button on the web page.

   When the robot starts moving, you will see the PID debug output in the console (PuTTY). This will only show when the robot is running.

5. You can press the **Stop** button on the web page to stop the robot moving or the **Exit** button to exit the behavior.

The robot won't be moving quite right; the movements may be understeering or overshooting. You'll need to tune the PID controllers to get this right, as shown in the next section.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Tuning the PID controller settings

I start with a proportional constant of 0.1, and raise it, using `nano` to make quick edits on the Pi, until the robot starts to overshoot—that is, it goes past its target, then returns far back—then I halve this proportional constant value.

It may then have a constant error, so I start raising the integral constant by about 0.01 to counter this error. Tuning PIDs is a slow process: start by getting the object close to dead center and tuning `direction_pid` until it's pretty good, then come back for `speed_pid`.

> **Important note**
> Do not try to tweak all the values at once—rather, change one thing and retry.

For a deeper look at this, see *Tuning a PID controller* in the *Further reading* section.

Troubleshooting

Color tracking is a tricky behavior, and there are some things that can go wrong:

- If the motors stop or slow down, the simplest fix is to use fresh batteries.
- If there are syntax errors, please check your code carefully.
- Ensure that the web app examples work with the camera and that you troubleshoot any problems there.
- You will need good lighting, as the mask may not pick up poorly lit objects.
- Beware of other objects in the view that may match; the mask may pick up things other than your intended items.
- Use the web app to check your object is in view and that the mask shows your object mostly in white. If not, then you may need to tune the upper and lower HSV ranges. The hue is the factor most likely to cause problems, as the saturation and value ranges are quite permissive.
- If the robot starts weaving from side to side, you may need to tune the direction PID. Reduce the proportional element somewhat.
- If the robot barely turns, you can increase the proportional element a little.
- If the robot is stopped but not facing the detected object, then increase the integral element for the direction PID by about 0.01. If you see the same problems moving back and forward, try applying the same tweaks.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You've seen how to track a brightly colored object with a camera, a technique you can use to spot objects in a room, or by industrial robots to detect ripe fruit. It is quite impressive to watch. However, some objects are more subtle than just a color, for example, a human face. In the next section, we look at how to use cascading feature matches to pick out objects.

Tracking faces with Python

Detecting faces (or other objects) by features is a smart behavior. Once our robot is detecting faces, it will point the pan-and-tilt mechanism at the nearest (well, largest) face.

Using **Haar cascades** is a common technique, well documented in a paper by Paul Viola and Michael Jones (known as *Viola Jones*). In essence, it means using a cascade of feature matches to search for a matching object. We will give an overview of this technique, then put it into use on our robot to create a fun behavior. Using different cascade model files, we could pick out faces or other objects.

Finding objects in an image

We will be using an algorithm implemented in OpenCV as a single and useful function, which makes it very easy to use. It provides a simple way to detect objects. More advanced and complex methods involve machine learning, but many systems use Haar cascades, including camera apps on phones. Our code will convert the images into grayscale (black through gray to white) for this detection method. Each pixel here holds a number for the intensity of light.

First, let's dig into a way of representing these images: integral images.

Converting to integral images

There are two stages applied in the function. The first is to produce an **integral** image, or **summed-area table**, as shown in *Figure 13.15*:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

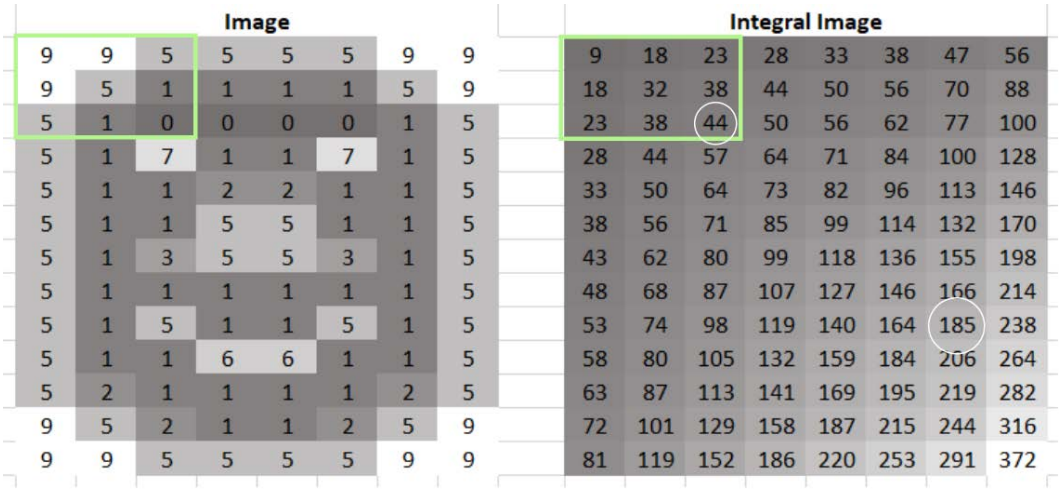


Image							
9	9	5	5	5	5	9	9
9	5	1	1	1	1	5	9
5	1	0	0	0	0	1	5
5	1	7	1	1	7	1	5
5	1	1	2	2	1	1	5
5	1	1	5	5	1	1	5
5	1	3	5	5	3	1	5
5	1	1	1	1	1	1	5
5	1	5	1	1	5	1	5
5	1	1	6	6	1	1	5
5	2	1	1	1	1	2	5
9	5	2	1	1	2	5	9
9	9	5	5	5	5	9	9

Integral Image							
9	18	23	28	33	38	47	56
18	32	38	44	50	56	70	88
23	38	44	50	56	62	77	100
28	44	57	64	71	84	100	128
33	50	64	73	82	96	113	146
38	56	71	85	99	114	132	170
43	62	80	99	118	136	155	198
48	68	87	107	127	146	166	214
53	74	98	119	140	164	185	238
58	80	105	132	159	184	206	264
63	87	113	141	169	195	219	282
72	101	129	158	187	215	244	316
81	119	152	186	220	253	291	372

Figure 13.15 – Integral images and summed-area tables

The left side of *Figure 13.15* shows a *smiling face* type image, with numeric pixels representing shades, with larger numbers making for a lighter color. Every shade has a number.

On the right of *Figure 13.15* is the integral image. Each pixel in the integral image is the sum or **integral** of the previous pixels. It adds itself to the original pixels above and left of it. The coordinate 2,2 is circled. It is the last in a 3x3 grid. The cell here has the value 44. 44 is the sum of the pixels in the highlighted box (9 + 9 + 5 + 9 + 5 + 1 + 5 + 1 + 0).

When the code sums the pixels, the integral process can use a shortcut and use the previous sums. The new sum is equal to the pixel to the left plus the pixel above. For example, for a pixel much further down (8,8), also circled in the image, we could add all the numbers, but it will be faster to reuse the results we already have. We can take the pixel value (1), add the sum above (166), and add the sum to the left (164). This sum will have included the middle pixels twice, so we need to subtract those, so take away the value up and to the left (146). The sum for this would be 1 + 164 + 166 – 146 = 185. The computer can do this pretty quickly.

This creates an array of numbers with the same dimensions as the image. Each coordinate is the sum of all the pixels' intensities between the current coordinate and 0,0.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Code can use the integral image to quickly find the intensity sum of any rectangle in it, of any size. You can start with the bottom-right pixel of the image, then subtract the top-right one, leaving the sum of pixels below the top-right pixel. We also then want to subtract the bottom-left pixel. This nearly constrains the sum to only the rectangle's pixels, but we would have taken away sections above the top-left pixel twice. To correct this, add back the value of the top-left pixel:

$$area_of_rectangle = bottom_right - top_right - bottom_left + top_left$$

The equation works for a small rectangle of 2x2 or a large 300x200 rectangle. See the Viola Jones paper in the *Further reading* section for more details. The good news is, you don't need to write this code as it's already part of the OpenCV classifier. The cascade stage can use this integral image to perform its next potent trick quickly.

Scanning for basic features

The next part of this puzzle is scanning the image for features. The features are extremely simple, involving looking for the difference between two rectangles, so they are quick to apply. *Figure 13.16* shows a selection of these basic features:

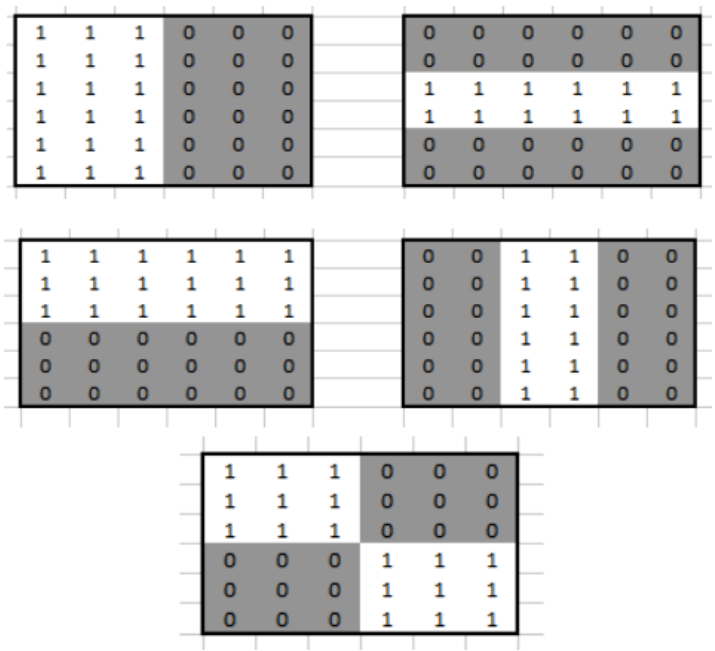


Figure 13.16 – Simple rectangular feature types

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The top left of *Figure 13.16* shows a left/right feature, where the left pixels are set to **1** and the right set to **0** (and shaded). This will match a vertical contrast feature. The figure's top right has two rows of **0**s (shaded), two rows of **1**s, and then two further rows of shaded **0**s; this will match a horizontal bar feature. The middle left has the top three rows set to **1**s and the lower three rows shaded and set to **0**s, matching a horizontal contrast feature. The figure's middle right has two columns of shaded **0**s, followed by two columns of **1**s, and then two further rows of shaded **0**s; this will match a vertical bar feature.

The bottom image shows a feature with the first few rows as three **1**s followed by three **0**s. It follows these rows with three rows of three **0**s andt three **1**s. This makes a small checkerboard pattern that will match a feature with diagonal contrast.

The algorithm will apply rectangles like those from *Figure 13.16* in a particular order and relative locations, then each match will *cascade* to a further attempt to match another feature. Files describe objects as a set of features. There are face cascades with 16,000 features to apply. Applying every single one to every part of an image would take a long time. So they are applied in groups, starting perhaps with just one. If a feature check fails, that part of the image is not subject to further feature tests. Instead, they cascade into later group tests. The groups include weighting and applying groups of these features at different angles.

If all the feature checks pass, then the checked region is taken as a match. For this to work, we need to find the feature cascade that will identify our object. Luckily, OpenCV has such a file designed for face recognition, and we have already installed it on our Raspberry Pi.

This whole operation of applying the summed area, then using the cascade file to look for potential matches, is all available through two OpenCV operations:

- `cv2.CascadeClassifier(cascade_filename)` will open the given cascade file, which describes the features to test. The file only needs to be loaded once and can be used on all the frames. This is a constructor and returns a `CascadeClassifier` object.
- `CascadeClassifier.detectMultiScale(image)` applies the classifier check to an image.

You now have a basic understanding of a common face (and object) recognition technique. Let's use cascade classifier visual processing with our existing behavior experience to plan the face-tracking behavior.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Planning our behavior

We can use code fairly similar to our color-tracking behavior to track faces. We'll set our robot up to use the pan-and-tilt mechanism to follow the largest face seen in the camera. The block diagram in *Figure 13.17* shows an overview of the face behavior:

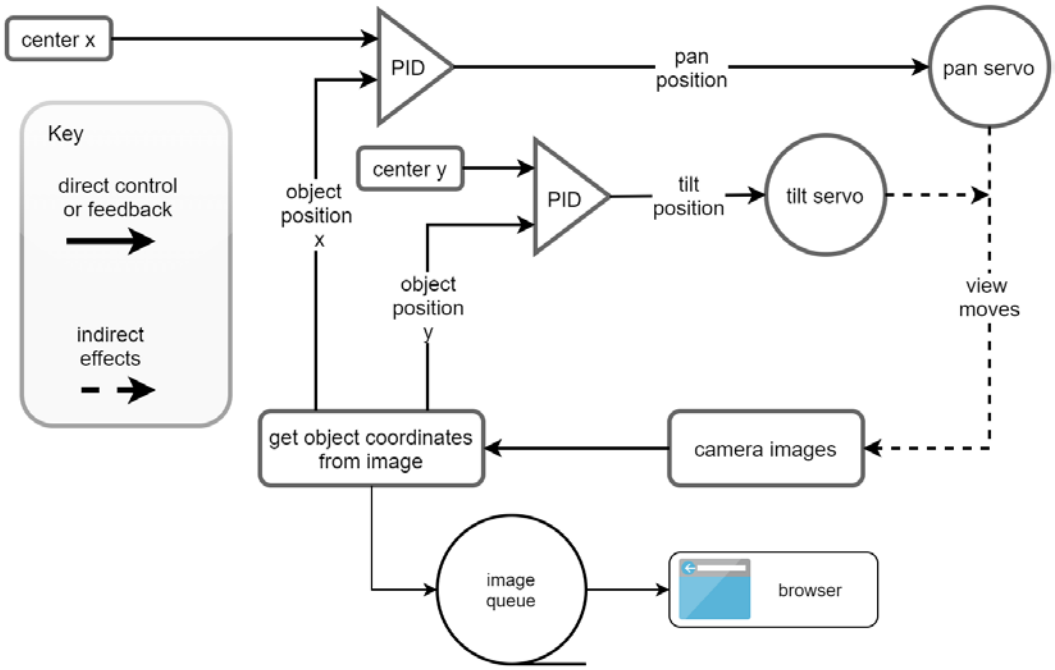


Figure 13.17 – The face-tracking behavior

The flow in *Figure 13.17* will look very familiar. We have the same camera to visual behavior to image queue we've seen before. This time, the visual processing is **get object coordinates from image**, which outputs an `x` and `y` coordinate for the item. We feed position `x` into a PID with center `x` to get a pan position, which is then used by the pan servo motor. Position `y` is fed to a PID with center `y` and outputs a tilt position to the tilt servos. The servos move the camera, creating a feedback loop where the view moves.

The differences are in the data we are sending to the PID controllers, and that each PID controls a different servo motor.

Now we have a plan; let's write the code.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Writing face-tracking code

The code for this behavior will seem very familiar—adapting the previous behavior code for this purpose. It's possible that refactoring could yield more common code, but it is currently simpler to work with a copy for now. This code will go into the `face_track_behavior.py` file. I've not even created a new template, as the color track template will work just fine for this:

1. The imports are nearly the same as our `color_track_behavior`:

```
import time

from image_app_core import start_server_process, get_
control_instruction, put_output_image

import cv2
import os

import camera_stream
from pid_controller import PIController
from robot import Robot
```

2. The `init` function for the behavior class is slightly different, starting with loading the Haar cascade. There are many other cascade files in the same directory, with which you could try to track things other than a face. This code uses `assert` to verify that the file exists at the location here because OpenCV will instead return cryptic errors in `detectMultiscale` if it cannot find it:

```
class FaceTrackBehavior:
    def __init__(self, robot):
        self.robot = robot
        cascade_path = "/usr/local/lib/python3.7/dist-
packages/cv2/data/haarcascade_frontalface_default.xml"
        assert os.path.exists(cascade_path), f"File
{cascade_path} not found"
        self.cascade = cv2.CascadeClassifier(cascade_
path)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The tuning parameters have center positions and a minimum face size. I've also brought the PID controllers out to the class, so they can be tuned here, and then reset in the control handler (you could add the reset to the previous behavior too):

```
        self.center_x = 160
        self.center_y = 120
        self.min_size = 20
        self.pan_pid = PIController(proportional_
constant=0.1, integral_constant=0.03)
        self.tilt_pid = PIController(proportional_
constant=-0.1, integral_constant=-0.03)
```

4. Our constructor still tracks whether the behavior is running motors or not:

```
        self.running = False
```

5. The process control here differs; when the `stop` instruction is received, it stops the motors and resets the PIDs:

```
    def process_control(self):
        instruction = get_control_instruction()
        if instruction:
            command = instruction['command']
            if command == "start":
                self.running = True
            elif command == "stop":
                self.running = False
                self.pan_pid.reset()
                self.tilt_pid.reset()
                self.robot.servos.stop_all()
            elif command == "exit":
                print("Stopping")
                exit()
```

6. This behavior still has a `find_object` method, taking the original frame. First, we convert the image to grayscale to reduce the amount of data to search:

```
    def find_object(self, original_frame):
        gray_img = cv2.cvtColor(original_frame, cv2.
COLOR_BGR2GRAY)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Next, we use the grayscale image with the cascade `detectMultiScale` method to get a list of matches:

```
        objects = self.cascade.detectMultiScale(gray_img)
```

   The `detectMultiScale` method creates the integral image and applies the Haar cascade algorithm. It will return several objects as rectangles, with `x`, `y`, width, and height values.

8. We can use a loop similar to the color-tracking behavior to find the largest rectangle by area. First, we need to set up a store for the current largest rectangle, in a data structure holding the area, then a sub-list containing the `x`, `y`, width, and height:

```
        largest = 0, (0, 0, 0, 0)
        for (x, y, w, h) in objects:
            item_area = w * h
            if item_area > largest[0]:
                largest = item_area, (x, y, w, h)
```

9. We return the position and dimensions of that largest rectangle:

```
        return largest[1]
```

10. The `make_display` method is simpler than the color-tracking behavior, as there is only one image. It must still encode the image, though:

```
    def make_display(self, display_frame):
        encoded_bytes = camera_stream.get_encoded_bytes_
for_frame(display_frame)
        put_output_image(encoded_bytes)
```

11. The `process_frame` method finds the object and then draws a rectangle on the frame for output. The `cv2.rectangle` function takes two coordinates: a starting `x`,`y` and an ending `x`,`y`, along with a color value. To get the ending coordinates, we need to add the width and height back in:

```
    def process_frame(self, frame):
        (x, y, w, h) = self.find_object(frame)
        cv2.rectangle(frame, (x, y), (x + w, y + w),
[255, 0, 0])
        self.make_display(frame)
        return x, y, w, h
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. Now comes the `run` function. We start with the camera setup and warm-up time:

```
    def run(self):
        camera = camera_stream.setup_camera()
        time.sleep(0.1)
        print("Setup Complete")
```

13. Like the color-tracking behavior, we start the main loop by processing the frame and checking for control instructions:

```
        for frame in camera_stream.start_stream(camera):
            (x, y, w, h) = self.process_frame(frame)
            self.process_control()
```

14. We only want to be moving if we've detected a large enough object (using height, as faces tend to be bigger in this dimension) and if the robot is running:

```
            if self.running and h > self.min_size:
```

15. When we know the robot is running, we feed the PIDs and send the output values straight to the servo motors for both pan and tilt. Note that to find the middle of the object, we take the coordinate and add half its width or height:

```
                pan_error = self.center_x - (x + (w / 2))
                pan_value = self.pan_pid.get_value(pan_
error)
                self.robot.set_pan(int(pan_value))
                tilt_error = self.center_y - (y + (h /2))
                tilt_value = self.tilt_pid.get_
value(tilt_error)
                self.robot.set_tilt(int(tilt_value))
```

16. So that we can track what is going on here, a debug `print` statement is recommended:

```
                print(f"x: {x}, y: {y}, pan_error:
{pan_error}, tilt_error: {tilt_error}, pan_value: {pan_
value:.2f}, tilt_value: {tilt_value:.2f}")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

17. Finally, we need to add the code to set up and run our behavior. Notice that we still use the color-tracking template:

```
print("Setting up")
behavior = FaceTrackBehavior(Robot())
process = start_server_process('color_track_behavior.
html')
try:
    behavior.run()
finally:
    process.terminate()
```

With the code ready, including the setup functions, we can try it out and see the behavior running.

Running the face-tracking behavior

To run this behavior, you will need to have uploaded the color-tracking behavior files already:

1. Upload the `face_track_behavior.py` file.
2. Start using `$ python3 face_track_behavior.py`.
3. Send your browser to `http://myrobot.local:5001`. You should see a single frame of the camera, with a rectangular outline around the largest face.
4. You must press the **Start** button for the robot to move.

The servo motors on the pan-and-tilt mechanism should move to try and put your face in the middle of the screen, which will mean the camera is pointed right at you. If you move your head around, the camera will (slowly) follow you. If you have someone stand behind you, the behavior won't pick them up, but if you cover half your face with your hand, it will stop recognizing you, and turn to their face instead.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

Start with the troubleshooting steps that we covered for the previous behavior—that should get you most of the way—then try these if you need to:

- If the app fails to find the Haar cascade file, check the location for the files there. These files have moved between OpenCV packaging versions and may do so again. Check that you haven't mistyped it. If not, then try the following command:

  ```
  $ find /usr/ -iname "haarcas*"
  ```

  This command should show the location of the files on the Raspberry Pi.
- If the camera fails to detect faces in the picture, try making sure the area is well lit.
- The detection algorithm is only for faces that face the camera head-on, and anything obscuring a part of the face will fool it. It is a little picky, so glasses and hats may confuse it.
- Faces only partially in the frame are also likely to be missed. Faces that are too far away or small are filtered. Reducing the minimum parameter will pick up more objects but generate false positives from tiny face-like objects.
- Please check the indentation matches, as this can change the meaning of where things happen in Python.

You have now made code that will detect and track faces in a camera view. Face-tracking behavior is sure to be impressive. Let's summarize what we've seen in this chapter.

Summary

In this chapter, you saw how to set up the Raspberry Pi Camera module. You then used it to see what your robot sees—the robot's view of the world.

You got the robot to display its camera as a web app on a phone or desktop, and then used the camera to drive smart color- and face-tracking behaviors. I've suggested ways the behaviors could be enhanced and hopefully given you a taste of what computer vision can do.

In the next chapter, we will extend our object-tracking visual processing to follow lines with the camera, seeing further ways to use the camera.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercises

This code is fun, but there are many ways you could improve the behaviors. Here are some suggested ways to extend this code and deepen your learning:

- Use the control pipeline to allow a user to tune the color filters, correct radius, and PID values from the web page. Perhaps the initial PID values should be close to the other tunable values?
- There is quite a lot of setup code. Could you put this into a function/method?
- Could the queues to the web page be used to send the debug data to the page, instead of printing them in the console? Could the data be plotted in a graph?
- The field of view for tracking with the Pi Camera is pretty narrow. A wide-angle lens would improve the field of view a lot, letting the robot see more.
- The camera doesn't perform too well when it's dark. The robot has an LED strip, but it's not illuminating much. Could you add a bright LED as a headlamp for the camera?
- You could track other objects by trying the other cascade files found in the `/usr/share/opencv/haarcascades` folder on the Raspberry Pi.
- Perhaps you could try swapping features of the two behaviors to use the servo motors to track the colored object, or chase the faces?
- Could you combine the pan-and-tilt mechanism with the main wheels to track an object, then engage the main wheels to chase the matching face and aim to center the pan while keeping the object in view? This may require some fancy PID controller thinking.

With these ideas, you should have plenty of ways to dig further into this type of visual processing.

Further reading

Visual processing is a deep topic, so this is only a small selection of places where you can read more about using a camera for visual processing:

- *The Official Raspberry Pi Camera Guide* at `https://magpi.raspberrypi.org/books/camera-guide` is an excellent resource for getting to know the camera, with many practical projects for it.
- To delve in far greater depth into using the Raspberry Pi Camera, I recommend the PiCamera documentation, available at `https://picamera.readthedocs.io/`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- To gain insight into further techniques, the PyImageSearch website, at `https://www.pyimagesearch.com`, has great resources.
- OpenCV and visual processing is a complex topic, only briefly covered here. I recommend *OpenCV 3 Computer Vision with Python Cookbook*, by *Alexey Spizhevoy* and *Aleksandr Rybnikov*, from *Packt Publishing*, available at `https://www.packtpub.com/application-development/opencv-3-computer-vision-python-cookbook`, for more information.
- Streaming video through Flask is a neat trick and is explored further in *Video Streaming with Flask*, at `https://blog.miguelgrinberg.com/post/video-streaming-with-flask`.
- I recommend `https://flaskbook.com/` for other neat ways to use Flask to manage your robot from your phone or laptop.
- Tuning a PID controller—we touched on this in *Chapter 11*, *Programming Encoders with Python*, and needed more in this chapter. *Robots For Roboticists | PID Control*, available at `http://robotsforroboticists.com/pid-control/`, is a little heavy on the math but has an excellent section on manually tuning a PID.
- *Rapid Object Detection Using a Boosted Cascade of Simple Features*, by Paul Viola and Michael Jones, available at `https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf`. This paper, from 2001, discusses in more detail the Haar cascade object-finding technique that we used.
- A good video introducing face tracking is *Detecting Faces (Viola Jones Algorithm) – Computerphile*, available at `https://www.youtube.com/watch?v=uEJ71VlUmMQ`, which dives into the combination of techniques used.
- The cascade classification OpenCV documentation, at `https://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html`, gives a reference for the library functions used in the face-tracking behavior.
- OpenCV also has a tutorial on face tracking (for version 3.0), called *OpenCV: Face Detection using Haar Cascades*, which is available at `https://docs.opencv.org/3.3.0/d7/d8b/tutorial_py_face_detection.html`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

14
Line-Following with a Camera in Python

In the last chapter, we saw how to use a camera to follow and track objects. In this chapter, we will be extending the camera code to create line-sensing behavior.

We will look at where robots use line following and how it is useful. We will also learn about some of the different approaches taken to following paths in different robots, along with their trade-offs. You will see how to build a simple line-following track.

We will learn about some different algorithms to use and then choose a simple one. We will make a data flow diagram to see how it works, collect sample images to test it with, and then tune its performance based on the sample images. Along the way, we'll see more ways to approach computer vision and extract useful data from it.

We will enhance our PID code, build our line detection algorithm into robot driving behavior, and see the robot running with this. The chapter closes with ideas on how you can take this further.

In this chapter, we're going to cover the following main topics:

- Introduction to line following
- Making a line-follower test track
- A line-following computer vision pipeline

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Trying computer vision with test images
- Line following with the PID algorithm
- Finding a line again

Technical requirements

For this chapter, you will need the following:

- The robot and code from *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*
- Some white or some black insulating tape
- Some A2 paper or boards – the opposite color to the insulating tape
- A pair of scissors
- Good lighting

The code for this section can be found at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter14`.

Check out the following video to see the Code in Action: `https://bit.ly/3slLzbQ`

Introduction to line following

Before we start building code, let's find out about line-following robot behaviors, where and how systems use them, and the different techniques for doing so.

What is line following?

Some robots are required to stay on specific paths within their tasks. It is simpler for a robot to navigate a line than to plan and map whole rooms or buildings.

In simple terms, line following is being able to follow a marked path autonomously. These can be visual markers, such as blue tape or a white line on a black road. As the robot drives along the line, it will continually be looking for where the line ahead is and correcting its course to follow that line.

In robot competitions, racing on lines is a common challenge, with speed being critical after accuracy.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Usage in industry

By far the most common usage of line-following behavior is in industry. Robots known as **automated guided vehicles** (**AGVs**) need to follow set paths for many reasons. These tasks can be warehouse robots staying on tracks between aisles of stacked products or factory robots staying on paths clear of other work areas. The line may mark a route between a storage shelf and a loading bay or a robot charging station and the robot's work area:

Figure 14.1 – IntellCart – a line-following industrial robot by Mukeshhrs [Public domain]

IntelliCart, shown in *Figure 14.1*, uses bright blue guide tape, although, in most industrial applications, robots use under-floor magnetic tracks.

The route may include choice points, with multiple lines coming from a particular location. Depending on their task, the robot may need extra clues to sense that it has reached these points. An engineer can set up a repeated path for a fully automated system.

Having these demarcated means that you can set safety boundaries and be clear on where humans and robots do or do not interact; this means that robots will rarely operate outside of well-understood areas.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Types of line following

There are a few major branches of line following and related systems.

Visual line following is by far the most commonly practiced and easy-to-set-up line following technique. It consists of a painted, drawn, or taped visual line that robots detect. Optical lines are simple, but surface dirt and light conditions can make this unreliable. How it is detected falls into a couple of major categories:

- **Detected with light sensors**: In this case, we'd attach small sensors to a robot's underside close to the line. They are tuned to output a binary on/off signal or analog signal. They usually have lights to shine off the surface. These are small and cheap but require extra I/O.
- **Detected with a camera**: This will save space if you already use a camera, along with I/O pins. It saves complexity in mounting them and wiring them. However, it comes at a trade-off cost of software complexity, as your robot needs computer vision algorithms to analyze this.

Magnetic line following is used when the line needs to be protected against the elements. Also, for some variations of this, you can guide a robot on multiple paths. There are the following variants:

- Running a magnetic strip along a floor allows Hall-effect sensors (such as the magnetometer in *Chapter 12*, *IMU Programming with Python*) to detect where the strip is. A series of these sensors can determine the direction of a line and follow it. This can be easier to alter than painting a line but can be a trip hazard.
- Running a wire with some current through it along or under a floor will achieve the same effect. With multiple wires and some different circuits for them, systems can steer a robot onto different paths.
- Concealing the line under a floor removes the trip hazard but means that you need to paint warnings for humans on paths that industrial robots follow.

Now, you have seen the two major types of line following; it's worth giving an honorable mention to some other ways to determine a robot's path in the real world:

- **Beacons**: Ultrasonic, light-emitting, or radio-emitting beacons can be placed around an environment to determine the path of a robot. These could just be reflectors for laser or other light.
- **Visual clues**: If you place QR codes or other visible markers on walls and posts, they can encode an exact position.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You've seen how a robot can perform line sensing with visible lines and hidden lines, such as wires under a floor and magnetic sensors. Because it is easier, we will use visible lines.

The simple optical sensors require additional wiring, but if we already have a camera capable of this, why not make use of it?

In this chapter, we will be focusing on using the camera we already have with a visual track and following the lines there. We will accept the code complexity while simplifying the hardware aspects of our robot.

Now that you have some idea of the different types of line following and where to use them, let's create a test track that our robot can follow.

Making a line-follower test track

Since you will be making your robot follow a line, we need to start with a section of line to follow. The track will be used at the beginning to test our line detection algorithm and can then be extended to more exciting tracks when we turn on the motors and start driving along the line. What I will show you in this section is easy to make and extendable. It allows you to experiment with different line shapes and curves and see how the robot responds.

You can even experiment with different color and contrast options.

Getting the test track materials in place

The following photo shows the main materials required:

Figure 14.2 – Materials for making a test track

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The photo in *Figure 14.2* shows a roll of black electrical tape on a large sheet of white paper. For this section, you'll need the following:

- Some A2 plain white paper or board.
- Some black electrical insulation tape or painter's tape. Make sure this tape is opaque.
- A pair of scissors.

You could replace the paper with boards if they are white-painted.

You can also swap things around by using dark or black paper and white tape. This tape must not be see-through so that it makes a good strong contrast against the background.

Making a line

Lay the sheet of paper flat. Then, make a line along the middle of the paper with the tape:

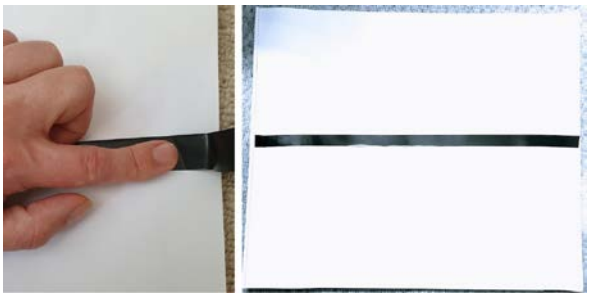


Figure 14.3 – Smoothing the tape on the paper

The photos in *Figure 14.3* show the paper with the tape line and me smoothing the tape with my finger. Be sure to smooth the tape down. You do not need to worry about making it perfectly straight, as the whole point of this system is to follow lines even when they curve.

Once you have a few lengths of this tape on sheet, why not make a few interesting pieces, such as the ones in the following figure:

Figure 14.4 – Some different shapes adjoining a straight line

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

As *Figure 14.4* shows, you can experiment with curves and intentionally not-quite-straight lines. You can join these together with the straight lines to make whole sections, like a robotic train set! These will be fun to test with later as you further tune and play with the line-following code.

Now that we have a test track ready, we can think about how we can visually process the line.

Line-following computer vision pipeline

As we did with the previous computer vision tasks, we will visualize this as a pipeline. Before we do, there are many methods for tracking a line with computer vision.

Camera line-tracking algorithms

It is in our interests to pick one of the simplest ones, but as always, there is a trade-off, in that others will cope with more tricky situations or anticipate curves better than ours.

Here is a small selection of methods we could use:

- **Using edge detection**: An edge detection algorithm, such as the Canny edge detector, can be run across the image, turning any transitions it finds into edges. OpenCV has a built-in edge detection system if we wanted to use this. The system can detect dark-to-light and light-to-dark edges. It is more tolerant of less sharp edges.
- **Finding differences along lines**: This is like cheeky edge detection, but only on a particular row. By finding the difference between each pixel along a row in the image, any edges will show significant differences. It's simpler and cheaper than the Canny algorithm; it can cope with edges going either way but requires sharp contrasts.
- **Finding brightness and using a region over an absolute brightness as the line**: This is very cheap but a little too simplistic to give good results. It's not tolerant to inversions but isn't tracking edges, so doesn't need sharp contrasts.

Using one of the preceding three methods, you can find the line in one picture area and simply aim at that. This means you won't be able to pre-empt course changes. It is the easiest way. The chosen area could be a single row near the bottom of the screen.

Alternatively, you can use the preceding methods to detect the line throughout the camera image and make a trajectory for it. This is more complex but better able to cope with steeper turns.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

It's worth noting that we could make a more efficient but more complicated algorithm using the raw YUV data from the Pi Camera. For simplicity, we will stick to the simple one. As you trade further up in complexity and understanding, you can find far faster and more accurate methods for this.

Another major limitation of our system is the view width of the camera. You could use lenses to let a camera take in a wide visual field so that the robot will not lose the line so often.

The method we will use is finding the differences along lines due to its simplicity and ability to cope with different line colors. We are also going to simply look along a single column, which results in more straightforward math.

The pipeline

We can visualize the way we process data as a pipeline. Before we can, let's quickly explain discrete differences. The brightness of each pixel is a number between 0 (black) and 255 (white). To get the difference, you subtract each pixel from the pixel to the right of it:

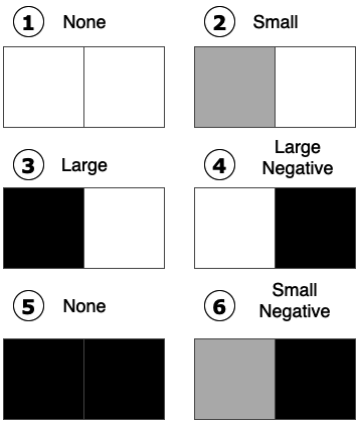


Figure 14.5 – Discrete differences between pixels

In *Figure 14.5*, there are six sets of pixels in varying shades:

1. The first shows two white pixels. There is no difference between them.
2. Where there is a gray pixel followed by a white one, it produces a small difference.
3. A black pixel followed by a white pixel produces a large difference.
4. A white pixel followed by a black pixel produces a large negative difference.
5. A black pixel followed by a black pixel will produce no difference.
6. A gray pixel followed by a black pixel will produce a small negative difference.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

It should be easy to see that a contrasting line edge will produce the largest differences, positive or negative. Our code will look for these.

The following diagram shows how we process camera data for this method:

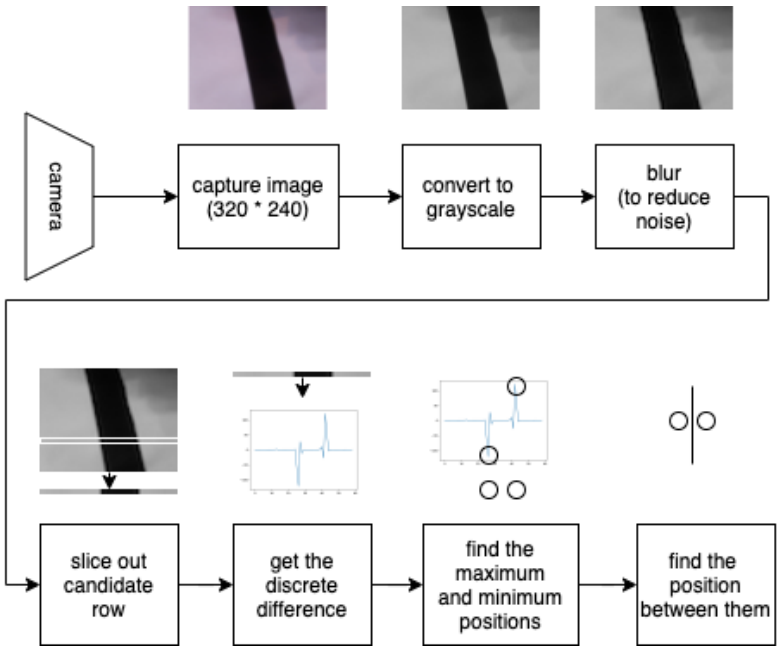


Figure 14.6 – Image processing pipeline for finding a line

In *Figure 14.6*, we show the process of finding a line to follow. It starts with the **camera**, from which we **capture images** at a 320-by-240-pixel resolution. The next step in the pipeline is to **convert to grayscale** – we are only interested in brightness right now.

Because images can have noise or grain, we **blur** it; this is not strictly necessary and depends on how clear the environment you are taking pictures from is.

We take that image and **slice out a candidate row**; this shouldn't be too high in the picture, as that line may be too far away or there may be random things above the horizon depending on the camera position. The row shouldn't be too low as it will then be too close to the robot for it to react in time. Above the **slice out candidate row** box is an example showing the sliced-out row and the image it came from.

We then treat this row as a set of numbers and **get the discrete difference across** them. The graph above the **discrete difference** box shows a large negative spike as the row goes from light gray to black, followed by a large positive spike as the row goes from black to light gray again. Notice that much of the graph shows a line along zero as patches of color have no difference.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The next step is to **find the maximum and minimum positions**, specifically where in the row they are. We want the position/index of the highest point above zero and the lowest point below zero. We now know where the boundaries of our line probably are.

We can **find the position between** these boundaries to find the center of the line, by adding them together and dividing by 2; this would be an *X* position of the line relative to the middle of the camera image.

Now, you've seen the pipeline with some test images. It's time to get some test images of your own and try this algorithm out with some code.

Trying computer vision with test images

In this section, we will look out how and why to use test images. We will write our first chunk of code for this behavior and try it on test images from our robot's camera. These tests will prepare us for using the code to drive the robot.

Why use test images?

So far, our computer vision work has been written directly with robot behaviors; this is the end goal of them, but sometimes, you want to try the visual processing code in isolation.

Perhaps you want to get it working or work out bugs in it, or you may want to see whether you can make the code faster and time it. To do this, it makes sense to run that particular code away from the robot control systems.

It also makes sense to use test images. So, instead of running the camera and needing light conditions, you can run with test images you've already captured and compare them against the result you expected from them.

For performance testing, trying the same image 100 times or the same set of images will give consistent results for performance measures to be meaningful. Avoid using new data every time, as these could result in unexpected or potentially noisy results. However, adding new test images to see what would happen is fascinating.

So now that we know why we use them, let's try capturing some test images.

Capturing test images

You may recall, from the previous chapter, using `raspistill` to capture an image. We are going to do the same here. First, we want to put our camera into a new position, facing down, so we are looking down onto the line.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This section requires the setup from the chapter *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV* and code from *Chapter 9*, *Programming RGB Strips in Python*.

Turn the motor power on to the Raspberry Pi, then with an `ssh` session into the Raspberry Pi on the robot, type the following:

1. We start Python by typing `python3`:

```
pi@myrobot:~ $ python3
Python 3.7.3 (default, Dec 20 2019, 18:57:59)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

2. Now, we need to import our robot object and create it so that we can interact with it:

```
>>> import robot
```

3. Let's create the robot object:

```
>>> r = robot.Robot()
```

4. Now, use this to set the pan servo to the middle:

```
>>> r.set_pan(0)
```

   The pan servo should center the camera.

5. Next, we set the tilt servo to face down to look at the line:

```
>>> r.set_tilt(90)
```

   The servo should look straight down here. It should not be straining or clicking.

6. Now, you can exit Python (and release the motors) by pressing *Ctrl* + *D*.

   The camera is facing downward. You can now turn off the motor switch and put this robot onto your test track. Try to position the robot so that the camera is right above the line.

7. In the `ssh` terminal, type the following to capture a test image:

```
$ raspistill -o line1.jpg
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can now download this image to your PC using FileZilla, as discussed in the book's earlier chapters. The next figure shows a test image, also used for the preceding examples:

Figure 14.7 – A test image of a line

Figure 14.7 shows one of my test images. Note that the line is roughly starting in the middle of the picture, but it isn't exact and doesn't need to be. Note also that the lighting is a bit rough and is creating shadows. These are worth watching out for as they could confuse the system.

Capture a few images of the line at different angles to the robot and slightly left or slightly right of the camera.

Now that we have test images, we can write code to test them with!

Writing Python to find the edges of the line

We are ready to start writing code, using our test images and the preceding pipeline diagram. We can make the results quite visual so that we can see what the algorithm is doing.

> **Tip**
>
> In computer vision, it's useful to use the lowest resolution you can to do the job. Every additional pixel adds more memory and processing to cope with. At 320*200, this is 76,800 pixels. The Raspberry Pi camera can record at 1920 x 1080 – 2,073,600 pixels – 27 times as much data! We need this to be quick, so we keep the resolution low.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The code in this section will run on the Raspberry Pi, but you can also run it on a PC with Python 3, NumPy, Matplotlib, and Python OpenCV installed:

1. Create a file called `test_line_find.py`.
2. We will need to import NumPy to process the image numerically, OpenCV to manipulate the image, and Matplotlib to graph the results:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

3. Now, we load the image. OpenCV can load `jpg` images, but if there is a problem in doing so, it produces an empty image. So, we need to check that it loaded something:

```
image = cv2.imread("line1.jpg")
assert image is not None, "Unable to read file"
```

   I am assuming the image is called `line1.jpg` and is in the same directory that we will run this file from.

4. The captured image will be at the large default resolution of the camera. To keep this fast, we resize it to a smaller image:

```
resized = cv2.resize(image, (320, 240))
```

5. We also only want grayscale; we aren't interested in the other colors for this exercise:

```
gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
```

6. Now, we'll pick out the row; for now, we'll use 180 as that is fairly low on an image with a height of 240. The images are stored such that row 0 is the top. Note that we are telling NumPy to convert this into an `int32` type:

```
row = gray[180].astype(np.int32)
```

   We convert to `int32` with a sign (plus or minus) so that our differences can be negative.

7. We can get a list of differences for every pixel of this row. NumPy makes this easy:

```
diff = np.diff(row)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. We are going to plot this `diff` list. We will need the *x*-axis to be the pixel number; let's create a NumPy range from 0 to that range:

```
x = np.arange(len(diff))
```

9. Let's plot the `diff` variable against the pixel index (`x`), and save the result:

```
plt.plot(x, diff)
plt.savefig("not_blurred.png")
```

   You'll note that I've called this file `not_blurred`. This is because we've not added the optional blurring step. With the graph, we'll be able to see the difference.

Pointing at my test picture, I get the following graph:

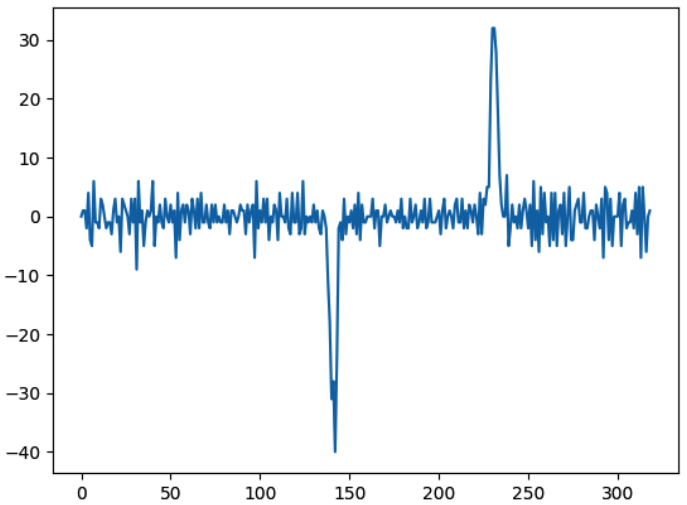


Figure 14.8 – Graph of differences without blurring

The graph in *Figure 14.8* has the column number as the *x* axis and the difference as the *y* axis. The line has a lot of noise in it. There are two distinct peaks – one below the zero line at around column 145 and one above the line at around 240. The noise here doesn't affect this too much as the peaks are very distinct:

1. Let's try adding the blurring to see how that changes things. Make the following change to the code. The bold areas show changed sections:

```
gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
blurred = cv2.blur(gray, (5, 5))
row = blurred[180].astype(np.int32)
diff = np.diff(row)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this code, we add the additional blurring step, blurring 5-by-5 chunks a little.

2. So that we can see different graphs, let's change the name of our output file:

```
plt.savefig("blurred.png")
```

Blurring a little should reduce noise without affecting our sharp peaks too much. Indeed, the following figure shows how effective this is:

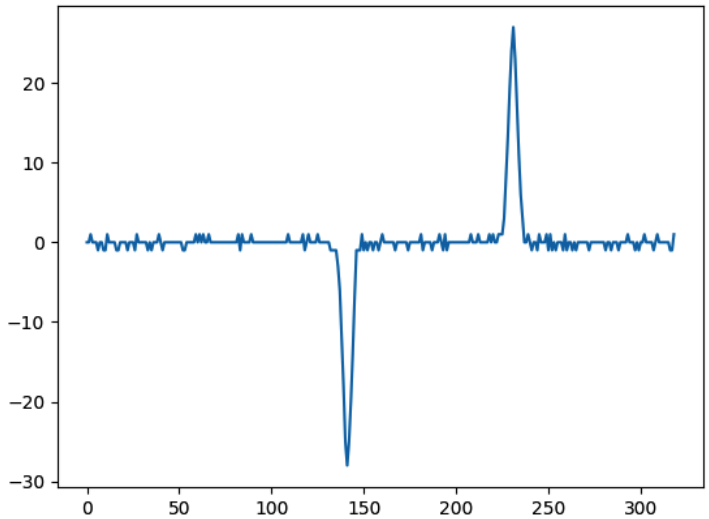


Figure 14.9 – The diff graph after blurring

The graph in *Figure 14.9* is similar to *Figure 14.8*. The axes are the same, and it has the same peaks. However, there is far less noise around the line at 0, showing that blurring makes the difference clearer. The question about this will be whether it changes the outcome. Looking at the position and size of the peaks, I would say not so much. So, we can leave it out of the final follow for a little extra speed. Every operation will cost a little time.

Now that we have the two peaks, let's use them to find the location of the line.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Locating the line from the edges

Those peaks are markers of our line edges. To find the middle of something, you add its left and right coordinates, then divide them by 2:

1. First, we must pick up the coordinates. Let's write code to ask for the maximum and minimum. We'll add this code between the analysis code and the chart output code:

```
diff = np.diff(row)
max_d = np.amax(diff, 0)
min_d = np.amin(diff, 0)
```

   This code finds the values of the array maximum and minimum. I've called them `min_d` and `max_d`, abbreviating the difference as `d`. Note that they cannot be called `min` and `max` as those names already belong to Python.

2. These are values, but not locations. We now need to find the index of the locations. NumPy has an `np.where` function to get indexes from arrays:

```
highest = np.where(diff == max_d)[0][0]
lowest = np.where(diff == min_d)[0][0]
```

   NumPy's `where` function returns an array of answers for each dimension – so, although `diff` is a one-dimensional array, we will still get a list of lists. The first `[0]` selects this first dimension's results list, and the second `[0]` selects the first item in the results. Multiple results mean it's found more than one peak, but we assume that there's only one for now.

3. To find the middle, we need to add these together and divide them by 2:

```
middle = (highest + lowest) // 2
```

4. Now that we have found it, we should display it in some way. We can plot this on our graph with three lines. Matplotlib can specify the color and style for a plot. Let's start with the middle line:

```
plt.plot([middle, middle], [max_d, min_d], "r-")
```

   The line is specified as a pair of *X* coordinates and a pair of *Y* coordinates, namely because Matplotlib expects data series. We use `max_d` and `min_d` for the Y coordinates, so the line draws from the highest peak to the lowest. The `r-` style specifier means to draw a solid red line.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. We can do the same for the `highest` and `lowest` locations, this time using `g--` for a green dashed line:

```
plt.plot([lowest, lowest], [max_d, min_d], "g--")
plt.plot([highest, highest], [max_d, min_d], "g--")
```

6. As we did for blurring, let's change the name of the output graph so that we can compare them:

```
plt.savefig("located_lines.png")
```

Running this should output the following figure:

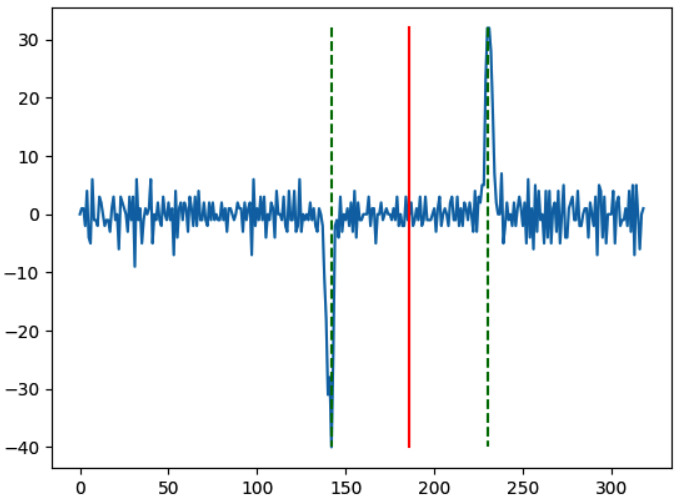


Figure 14.10 – Graph showing the highest, lowest, and middle line

The graph in *Figure 14.10* shows that we have found the middle line and the two nice clear peaks. This code looks usable for our robot.

However, what happens when things are not so clear?

Trying test pictures without a clear line

Let's see what our line-finding code does with a very different test picture. We will see what happens here, so we aren't so surprised by how the robot will behave and weed out some simple bugs.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

For example, what about putting our line on a very noisy surface, such as a carpet? Or how about the paper without a line, or the carpet without a line?

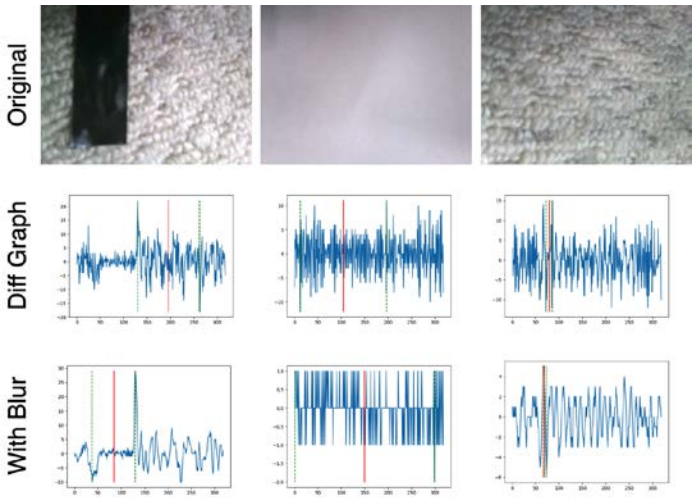


Figure 14.11 – diff graphs under noisier conditions

With a set of graphs such as those in *Figure 14.11*, we learn much about our system.

The top three images show the original photos. The next three graphs show what those images look like when finding the difference and middles without blurring. The bottom three graphs show what happens when enabling the blur.

First, when things get as noisy as the first image (and this is pushing it past what line following should cope with), the blur makes the difference between finding the line and a random artifact; although, in the second graph, a random artifact with a similar downward peak size was a close contender. In this case, making a larger *Y* blur might smooth out that artifact, leaving only the line.

Looking closely, the scale of those graphs is also not the same. The plain paper graph measures a difference with peaks of +10/-10 without blurring, and +1/-1 with blurring. So, when the differences are that low, should we even be looking for a peak? The story is similar in the carpet-only graphs.

We can make a few changes to our system to make it consider these as not-lines. The simplest is to add a condition that filters out a minimum above -5 and a maximum below 10. I say -5 since this would otherwise filter out the line in the first graph completely. However, a larger blur area might help with that.

Depending on the noisiness of the conditions, we will want to enable the blur. On a nicely lit track, the blur is probably not needed.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The next figure shows our line on the carpet, with a blur set to (5, 40), blurring further between rows and filtering out noise further:

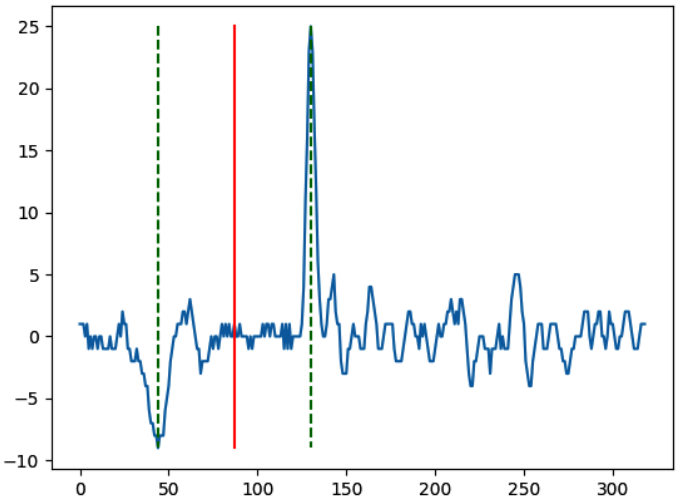


Figure 14.12 – The line on carpet with a larger blur

The graph in *Figure 14.12* has far less noise than before, with the blur smoothing out noise spikes a lot, while the actual line spikes remain. We would only want to do this in a noisy environment, as it risks being slower.

As you can see, testing the code on test images has allowed us to learn a lot about the system. By taking the same pictures and trying different parameters and pipeline changes, you can optimize this for different scenarios. As you experiment more with computer vision, make this a habit.

Now we have tried our visual processing code on test images, it's time to put it on a robot behavior!

Line following with the PID algorithm

In this section, we will combine the visual processing seen previously with the PID control loops and camera streaming seen in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*. Please start from the code in that chapter.

The files you will need are as follows:

- `pid_controller.py`
- `robot.py`
- `servos.py`

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- `camera_stream.py`
- `image_app_core.py`
- `leds_led_shim.py`
- `encoder_counter.py`
- The templates folder

We will use the same template for displaying this, but we are going to add a quick and cheeky way of rendering the `diff` graphs in OpenCV onto our output frame. Matplotlib would be too slow for this.

Creating the behavior flow diagram

Before we build a new behavior, creating a data flow diagram will help us get a picture of what happens to the data after we've processed it.

The system will look familiar, as it is very similar to those we made in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*. Take a look at the following figure:

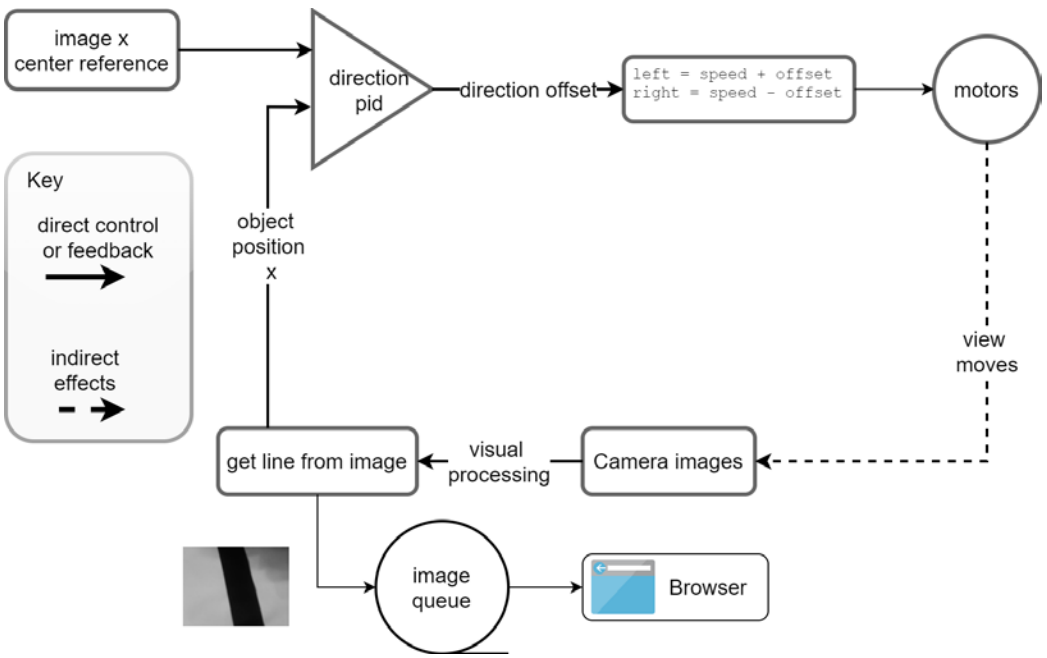


Figure 14.13 – The line-following behavior

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In *Figure 14.13*, we have camera images going through to the **get line from image** block. This block outputs an object's *X* position (the middle of the line), which goes to our PID. Note that image data also goes from the get line to the image queue so that you can see these in a browser.

The PID control also takes a reference middle point, where the middle of the camera should be. It uses the error between these to calculate the offset and uses that to drive the motors.

The figure shows the motors with a feedback line to the camera, as the indirect effects of moving those are that the view changes, so we will see a different line.

Before that, we are going to make our PID controller a little smarter again.

Adding time to our PID controller

Our robot behaviors have involved processing frames, then sending error data to the PID whenever the process has completed a cycle. There is much going on in a cycle, and that timing might vary. When we create the integral, we have been adding the data as if the time was constant. For a somewhat more accurate picture, we should be multiplying that by the time:

1. Open up the `pid_controller.py` file.
2. In the `handle_integral` method, change the parameters to take `delta_time`:

```
    def handle_integral(self, error, delta_time):
```

3. We will then use this when adding in the integral term:

```
            self.integral_sum += error * delta_time
```

4. We usually update the PID with the `get_value` method; however, since we already have code using this, we should make it behave as it did before for them. To do this, we will add a `delta_time` parameter but with a default value of `1`:

```
    def get_value(self, error, delta_time=1):
```

5. When this `get_value` method calls `handle_integral`, it should always pass the new `delta_time` parameter:

```
        p = self.handle_proportional(error)
        i = self.handle_integral(error, delta_time)
        logger.debug(f"P: {p}, I: {i:.2f}")
        return p + i
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

While this was not a big change, it will mean we can account for time variations between updates to the PID code.

We can now use this in our behavior.

Writing the initial behavior

We can take all the elements we have and combine them to create our line-following behavior:

1. Create a file named `line_follow_behavior.py`.
2. Start this with imports for `image_app_core`, NumPy, OpenCV, the camera stream, the PID controller, and the robot. We also have `time`, so we can later compute the delta time:

```
import time
from image_app_core import start_server_process, get_
control_instruction, put_output_image
import cv2
import numpy as np
import camera_stream
from pid_controller import PIController
from robot import Robot
```

3. Let's make the behavior class. The constructor, as before, takes the robot:

```
class LineFollowingBehavior:
    def __init__(self, robot):
        self.robot = robot
```

4. Now, we need variables in the constructor to track our behavior. First, we should set the row we will look for the differences in and a threshold (under which we will not consider it a line):

```
        self.check_row = 180
        self.diff_threshold = 10
```

5. As with our previous camera behaviors, we have a set point for the center, a variable to say whether the motors should be running, and a speed to go forward at:

```
        self.center = 160
        self.running = False
        self.speed = 60
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. We are going to make some interesting displays. We will store the colors we plan to use here too – a green crosshair, red for the middle line, and light blue for the graph. These are BGR as OpenCV expects that:

```
        self.crosshair_color = [0, 255, 0]
        self.line_middle_color = [128, 128, 255]
        self.graph_color = [255, 128, 128]
```

   That is the constructor complete for the behavior.

7. Now, we need the control to say whether the system is running or should exit. This code should be familiar as it is similar to the other camera control behaviors:

```
    def process_control(self):
        instruction = get_control_instruction()
        if instruction:
            command = instruction['command']
            if command == "start":
                self.running = True
            elif command == "stop":
                self.running = False
            if command == "exit":
                print("Stopping")
                exit()
```

8. Next, we'll make the `run` method, which will perform the main PID loop and drive the robot. We are setting the tilt servo to `90` and the pan servo to `0`, so it is looking straight down. We'll set up the camera too:

```
    def run(self):
        self.robot.set_pan(0)
        self.robot.set_tilt(90)
        camera = camera_stream.setup_camera()
```

9. Now, we set up the PID for the direction. These values aren't final and may need tuning. We have a low proportional value as the directional error can be quite large compared with the motor speeds:

```
        direction_pid = PIController( proportional_
 constant=0.4, integral_constant=0.01, windup_limit=400)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

10. We sleep for a second so that the camera can initialize and the servos reach their position:

```
            time.sleep(1)
            self.robot.servos.stop_all()
            print("Setup Complete")
```

    We stop the servos so that they won't be pulling further power once they've reached position.

11. Since we are going to be keeping track of time, we store the last time value here. The time is a floating-point number in seconds:

```
            last_time = time.time()
```

12. We start the camera loop and feed the frame to a `process_frame` method (which we'll write shortly). We can also process a control instruction:

```
            for frame in camera_stream.start_stream(camera):
                x, magnitude = self.process_frame(frame)
                self.process_control()
```

    From processing a frame, we expect to get an *X* value, and the magnitude is the difference between the highest and lowest value in the differences. The gap between the peaks helps detect whether it's actually a line and not just noise.

13. Now, for the movement, we need to check that the robot is running and that the magnitude we found was bigger than the threshold:

```
                if self.running and magnitude > self.diff_
threshold:
```

14. If so, we start the PID behavior:

```
                    direction_error = self.center - x
                    new_time = time.time()
                    dt = new_time - last_time
                    direction_value = direction_pid.get_
value(direction_error, delta_time=dt)
                    last_time = new_time
```

    We calculate a direction error by subtracting what we got from the middle of the camera. We then get a new time so that we can calculate the difference in time, `dt`. This error and time delta are fed to the PID, getting a new value. So, we are ready for the next calculation: `last_time` now gets the `new_time` value.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

15. We now log this and use the value to change the heading of the robot. We set the motor speeds to our base speed, and then add/subtract the motors' PID output:

```
                print(f"Error: {direction_error},
Value:{direction_value:2f}, t: {new_time}")
                self.robot.set_left(self.speed -
direction_value)
                self.robot.set_right(self.speed +
direction_value)
```

16. Now we've handled what happens when we have detected a line. What about when we don't? `else` stops the motors running and resets the PID, so it doesn't accumulate odd values:

```
            else:
                self.robot.stop_motors()
                if not self.running:
                    direction_pid.reset()
                last_time = time.time()
```

Notice how we are still keeping the last time up to date here. Otherwise, there would be a big gap between stops and starts, which would feed odd values into the PID.

17. Next, we need to fill in what happens when we process a frame. Let's add our `process_frame` method:

```
    def process_frame(self, frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        blur = cv2.blur(gray, (5, 5))
        row = blur[self.check_row].astype(np.int32)
        diff = np.diff(row)
        max_d = np.amax(diff, 0)
        min_d = np.amin(diff, 0)
```

This code should all look familiar; it is the code we made previously for our test code.

18. We should test to see that our readings have put us on either side of the zero line, and that we found two different locations. The maximum should not be below zero, and the minimum should not be above it. If they fail this, stop here – the main loop will consider this not a line:

```
        if max_d < 0 or min_d > 0:
            return 0, 0
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

19. We will find the locations on the row as we did before, along with their midpoint:

```
            highest = np.where(diff == max_d)[0][0]
            lowest = np.where(diff == min_d)[0][0]
            middle = (highest + lowest) // 2
```

20. So that we can use it to determine that we've got a positive match, we'll calculate the magnitude of the difference between the min and max, making sure we aren't picking up something faint:

```
            mag = max_d - min_d
```

21. We will want to display something useful to the user here. So, this method calls a `make_display` method, just like the other camera behaviors. We pass it some variables to plot onto that display:

```
            self.make_display(frame, middle, lowest, highest,
diff)
```

22. We then return the middle point and the magnitude:

```
            return middle, mag
```

23. This code will drive our robot, but we'll have a hard time tuning it if we can't see what is going on. So, let's create the `make_display` method to handle that:

```
    def make_display(self, frame, middle, lowest,
highest, diff):
```

    The parameters here are the original `frame`, the `middle` position for the line, the `lowest` difference position in the line, the `highest` difference position, and `diff` as the whole difference row.

24. The first thing we want in the display is the center reference. Let's make a crosshair about the center and the chosen row:

```
            cv2.line(frame, (self.center - 4, self.check_
row), (self.center + 4, self.check_row), self.crosshair_
color)
            cv2.line(frame, (self.center, self.check_row -
4), (self.center, self.check_row + 4), self.crosshair_
color)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

25. Next, we show where we found the middle in another color:

```
            cv2.line(frame, (middle, self.check_row - 8),
    (middle, self.check_row + 8), self.line_middle_color)
```

26. So that we can find it, we also plot the bars for the `lowest` and `highest` around it, in a different color again:

```
            cv2.line(frame, (lowest, self.check_row - 4),
    (lowest, self.check_row + 4), self.line_middle_color)
            cv2.line(frame, (highest, self.check_row - 4),
    (highest, self.check_row + 4), self.line_middle_color)
```

27. Now, we are going to graph `diff` across a new empty frame. Let's make an empty frame – this is just a NumPy array:

```
            graph_frame = np.zeros((camera_stream.size[1],
    camera_stream.size[0], 3), np.uint8)
```

    The array dimensions are rows then columns, so we swap the camera size *X* and *Y* values.

28. We will then use a method to make a simple graph. We'll implement this further down. Its parameters are the frame to draw the graph into and the *Y* values for the graph. The simple graph method implies the *X* values as column numbers:

```
            self.make_cv2_simple_graph(graph_frame, diff)
```

29. Now that we have the frame and the graph frame, we need to concatenate these, as we did for our frames in the color-detecting code:

```
            display_frame = np.concatenate((frame, graph_
    frame), axis=1)
```

30. We can now encode these bytes and put them on the output queue:

```
            encoded_bytes = camera_stream.get_encoded_bytes_
    for_frame(display_frame)
            put_output_image(encoded_bytes)
```

31. The next thing we will need to implement is this `make_cv2_simple_graph` method. It's a bit cheeky but draws lines between *Y* points along an *x* axis:

```
        def make_cv2_simple_graph(self, frame, data):
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

32. We need to store the last value we were at, so the code plots the next value relative to this – giving a line graph. We start with item 0. We also set a slightly arbitrary middle *Y* point for the graph. Remember that we know the `diff` values can be negative:

```
            last = data[0]
            graph_middle = 100
```

33. Next, we should enumerate the data to plot each item:

```
            for x, item in enumerate(data):
```

34. Now, we can plot a line from the last item *Y* position to the current position on the next *X* location. Notice how we offset each item by that graph middle:

```
                cv2.line(frame, (x, last + graph_middle), (x
+ 1, item + graph_middle), self.graph_color)
```

35. We then need to update the last item to this current one:

```
                last = item
```

    Okay – nearly there; that will plot the graph on our frame.

36. Our behavior is complete; we just need the outer code to run it! This code should also be similar to the previous camera examples:

```
print("Setting up")
behavior = LineFollowingBehavior(Robot())
process = start_server_process('color_track_behavior.
html')
try:
    behavior.run()
finally:
    process.terminate()
```

    Notice that we still use the `color_track_behavior.html` template here.

You can now upload this to your robot. Then, switch the motors on and run it. Because this is web-based, point a browser at `http://myrobot.local:5001`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You should see the following:

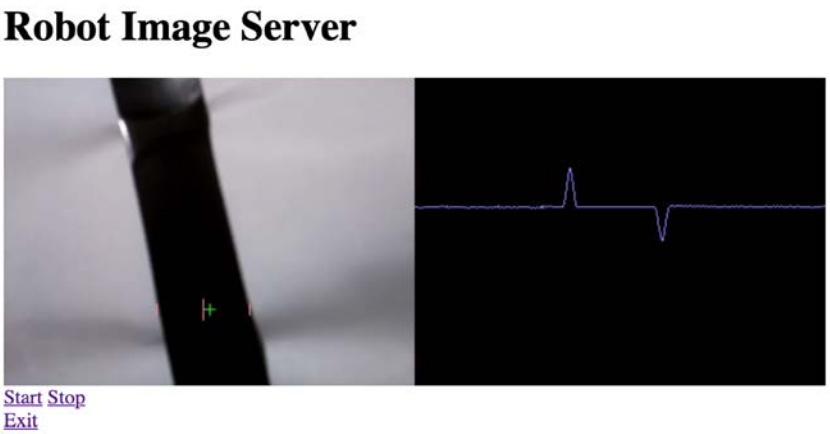


Figure 14.14 – Screenshot of the line-following behavior output

The screenshot in *Figure 14.14* shows the title above two pictures. On the left is the camera picture of the line. Drawn onto this frame is a green crosshair showing where the center point is. Also, there is a large red bar showing the middle of the line, and either side of this, two shorter red bars showing the sides of it. To the right is the graph plotting the intensity differences after blurring. The up peak and down peak are visible in the graph.

Below this are the **Start**, **Stop**, and **Exit** buttons.

Place the robot onto the line, with good lighting. If it looks like the preceding display, press the **Start** button to see it go. It should start shakily driving along the line.

Tuning the PID

You can get a little bolder with trying to track curved lines and find its limit. The robot will sometimes overshoot or understeer, which is where the PID tuning comes in:

- If it seems to be turning far too slowly, try increasing the proportional constant a little. Conversely, if it is oversteering, try lowering the proportional constant a fraction.
- If it had a slight continuous error, try increasing the integral constant.

PID tuning is a repeating process and requires a lot of patience and testing.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

If the behavior isn't quite working, please try the following steps:

- If the tilt servo doesn't look straight down when set to 90 degrees, it may not be calibrated correctly. Change the `deflect_90_in_ms` value parameter to the `Servos` object – increase in 0.1 increments to get this to 90 degrees.
- If it is having trouble getting a clear line, ensure that the lighting is adequate, that the surface it is on is plain, such as paper, and the line is well contrasting.
- If it is still struggling to find a line, increase the vertical blurring amount in steps of 5.
- If it's struggling to turn in time for the line, try reducing the speed in increments of 10.
- If you find the camera is wobbling horizontally, you can remove the `self.robot.servos.stop_all()` line from `line_follow_behavior`. Beware: this comes at the cost of motor battery life.
- If the robot is finding too much other random junk that isn't the line, try increasing the vertical blurring. Also, try increasing the threshold in steps of 1 or 2. The sharper the contrast in brightness, the less you should need to do this.
- Ensure that you double-check the code and that you have got the previous examples here and from *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*, to work.

Finding a line again

An important thing to consider is what the robot should do if it has lost the line. Coming back to our examples of an industrial setting, this could be a safety measure.

Our current robot stops. That requires you to put it back on the line. However, when you do so, the robot immediately starts moving again. This behavior is fine for our little robot, but it could be a dangerous hazard for a large robot.

Another behavior that you could consider is to spin until the robot finds the line again. Losing a line can be because the robot has under/oversteered off the line and couldn't find it again, or it could be because the robot has gone past the end of a line. This behavior is suitable perhaps for small robot competitions.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We need to consider things like this carefully and where you would use the robot. Note that for competition-type robots, or industrial robots, they will have either multiple sensors at different angles or a wider angled sensor – so, they are far less likely to lose the line like ours. Also, spinning for a larger robot, even slowly, could be very hazardous behavior. For this reason, let's implement a simple additional safety-type feature.

When it fails to find the line, it doesn't just stop the motors; it will set the running flag to false, so you need to start it manually again:

1. Open the `line_follow_behavior.py` file again.
2. Go to the `run` method and find the `else:` statement.
3. Now, we can modify the content of this statement:

```
                else:
                    self.robot.stop_motors()
                    self.running = False
                    direction_pid.reset()
                    last_time = time.time()
```

   We have made two small changes here. Instead of resetting the PID if `running` is false, we now set `running` to `False` every time. We also reset the PID every time.

Save the code to the robot, and run it until it loses the line. This could be by going off course or by reaching the end of the line. The robot should stop. It should wait for you to press the start button before trying to move again. Notice that you'll need to place it back on the line and press start for it to go again.

This robot now handles a lost line condition more predictably.

Summary

In this chapter, you saw how to use the camera to detect a line and how to plot data showing what it found. You then saw how to take this data and put it into driving behavior so that the robot follows the line. You added to your OpenCV knowledge, and I showed you a sneaky way to put graphs into frames rendered on the camera stream output. You saw how to tune the PID to make the line following more accurate and how to ensure the robot stops predictably when it has lost the line.

In the next chapter, we will see how to communicate with our robot via a voice agent, Mycroft. You will add a microphone and speakers to a Raspberry Pi, then add speech recognition software. This will let us speak commands to a Raspberry Pi to send to the robot, and Mycroft will respond to let us know what it has done.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercises

Now that we've got this to work, there are ways we could enhance the system and make it more interesting:

- Could you use `cv2.putText` to draw values such as the PID data onto the frames in the `make_display` method?
- Consider writing the PID and error data versus time to a file, then loading it into another Python file, using Matplotlib to show what happened. This change might make the under/oversteer clearer in retrospect.
- You could modify the motor handling code to go faster when the line is closer to the middle and slow down when it is further.
- A significant enhancement would be to check two rows and find the angle between them. You then know how far the line is from the middle, but you also know which way the line is headed and could use that to guide your steering further.

These exercises should give you some interesting ways to play and experiment with the things you've built and learned in this chapter.

Further reading

The following should help you look further into line following:

- Read about an alternative approach for line processing in the Go language on the Raspberry Pi from Pi Wars legend Brian Starkey at `https://blog.usedbytes.com/2019/02/autonomous-challenge-blast-off/`.
- Here is another line-following robot, using an approach like ours but more sophisticated: `https://www.raspberrypi.org/blog/an-image-processing-robot-for-robocup-junior/`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

15
Voice Communication with a Robot Using Mycroft

Using our voice to ask a robot to do something and receiving a voice response has been seen as a sign of intelligence for a long time. Devices around us, such as those using Alexa and Google Assistant, have these tools. Being able to program our system to integrate with these tools gives us access to a powerful voice assistant system. Mycroft is a Python-based open source voice system. We will get this running on the Raspberry Pi by connecting it to a speaker and microphone, and then we will run instructions on our robot based on the words we speak.

In this chapter, we will have an overview of Mycroft and then learn how to add a speaker/microphone board to a Raspberry Pi. We will then install and configure a Raspberry Pi to run Mycroft.

We'll also extend our use of Flask programming, building a Flask API with more control points.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Toward the end of the chapter, we will create our own skills code to connect a voice assistant to our robot. You will be able to take this knowledge and use it to create further voice agent skills.

The following topics are covered in this chapter:

- Introducing Mycroft – understanding voice agent terminology
- Limitations of listening for speech on a robot
- How to add a speaker/microphone board to a Raspberry Pi
- How to install and configure a Raspberry Pi to run Mycroft
- Programming a Flask control API
- How to create our own skills code to connect a voice assistant to our robot

Technical requirements

You will require the following hardware for this chapter:

- An additional Raspberry Pi 4 (model B).
- An SD card (at least 8 GB).
- A PC that can write the card (with the balenaEtcher software).
- The ReSpeaker 2-Mics Pi HAT.
- Mini Audio Magnet Raspberry Pi Speaker—a tiny speaker with a JST connector or a speaker with a 3.5 mm jack.
- It may be helpful to have a Micro-HDMI to HDMI cable for troubleshooting.
- Micro USB power supply.
- The robot from the previous chapters (after all, we intend to get this moving).

The code for this chapter is available on GitHub at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter15`.

Check out the following video to see the Code in Action: `https://bit.ly/2N5bXqr`

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introducing Mycroft – understanding voice agent terminology

Mycroft is a software suite known as a **voice assistant**. Mycroft listens for voice commands and takes actions based on those commands. Mycroft code is written in Python and is open source and free. It performs most of its voice processing in the cloud. After the commands are processed, Mycroft will use a voice to respond to the human.

Mycroft is documented online and has a community of users. There are alternatives that you could consider after you've experimented with Mycroft – for example, Jasper, Melissa-AI, and Google Assistant.

So, what are the concepts of a voice assistant? Let's look at them in the following subsections.

Speech to text

Speech to text (**STT**) describes systems that take audio containing human speech and turn it into a series of words that a computer can then process.

These can run locally, or they can run in the cloud on far more powerful machines.

Wake words

Voice assistants usually have a **wake word** – a phrase or word that is spoken before the rest of a command to get the attention of the voice assistant. Examples are the *Hey Siri*, *Hi Google*, and *Alexa* utterances. Mycroft uses the word *Mycroft* or the phrase *Hey Mycroft,* but that can be changed.

A voice assistant is usually only listening for wake words and ignores all other audio input until woken. The wake word is recognized locally on the device. The sounds it samples after the wake word are sent to a speech-to-text system for recognition.

Utterances

An **utterance** is a term for something a user says. Voice assistants use vocabulary you define to match an utterance to a skill. The specific vocabulary will cause Mycroft to invoke the intent handler.

The vocabulary in Mycroft comprises lists of interchangeable phrases in a file.

A good example of an utterance is asking Mycroft about the weather: *Hey Mycroft, what is the weather?*

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Intent

An **intent** is a task that the voice assistant can do, such as finding out what today's weather is. We will build intents to interact with our robot. An intent is part of a skill, defining the handler code for what it does and choosing a dialog to respond.

Using the weather skill as an example, the utterance *What is the weather?* triggers an intent to fetch the current weather for the configured location and then speak the details of this back to the user. An example for our robot is *ask the robot to test LEDs*, with an intent that starts the LED rainbow behavior (from *Chapter 9*, *Programming RGB Strips in Python*) on the robot.

Dialog

In Mycroft terminology, **dialogs** are phrases that Mycroft speaks to the user. An example would be *OK, the robot has been started*, or *Today, the weather is clear*.

A skill has a collection of dialogs. These have sets of synonymous words to say and can use different languages.

Vocabulary

Utterances you speak, once converted into text, are matched to **vocabulary**. Vocabulary files, like dialogs, are parts of an intent, matching utterances to action. The vocabulary files contain synonymous phrases and can be organized into language sets to make your skill multi-lingual.

This would make phrases like *what is the weather?*, *is it sunny?*, *do I need an umbrella?* or *will it rain?* synonymous. You may have things split – for example, *ask the robot to* as one vocabulary item and *drive forward* as another.

Skills

Skills are containers for a whole set of vocabulary for utterances, dialogs to speak, and *intents*. A skill for alarms might contain intents such as setting an alarm, listing the alarms, deleting an alarm, or changing an alarm. It would contain a dialog to say the alarm setting is complete or to confirm each alarm.

Later in this chapter, we will build a `MyRobot` skill with intents to make it move and stop.

Now you've learned a bit about the terminology and parts of a voice agent. We next need to consider what we will build. Where would we put a speaker and microphone?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Limitations of listening for speech on a robot

Before we start to build this, we should consider what we are going to make. Should the speaker and microphone be on the robot or somewhere else? Will the processing be local or in the cloud?

Here are some considerations to keep in mind:

- **Noise**: A robot with motors is a noisy environment. Having a microphone anywhere near the motors will make it close to useless.
- **Power**: The voice assistant is continuously listening. The robot has many demands for power already with the other sensors that are running on it. This power demand applies both in terms of battery power and the CPU power needed.
- **Size and physical location**: The speaker and voice HAT would add height and wiring complications to an already busy robot.

A microphone and speaker combination could be on a stalk for a large robot – a tall standoff with a second Raspberry Pi there. But this is unsuitable for this small and simple robot. We will create a separate voice assistant board that will communicate with our robot, but we won't be putting it directly on the robot. The voice assistant will be a second Raspberry Pi.

We will also be using a system that goes to the cloud to process the speech. While a fully local system would have better privacy and could respond quicker, at the time of writing, there is not a complete packaged voice assistant that works this way for a Raspberry Pi. The Mycroft software gives us flexibility in using our own skills and has a pluggable backend for voice processing, so that one day it may run locally.

Now we've chosen how we will make our voice agent with Mycroft and a second Raspberry Pi, it's time to start building it.

Adding sound input and output to the Raspberry Pi

Before we can use a voice processing/voice assistant, we need to give the Raspberry Pi some speakers and a microphone. A few Raspberry Pi add-ons provide this. My recommendation, with a microphone array (for better recognition) and a connection to speakers, is the ReSpeaker 2-Mics Pi HAT, which is widely available.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The next photograph shows the ReSpeaker 2-Mics Pi HAT:

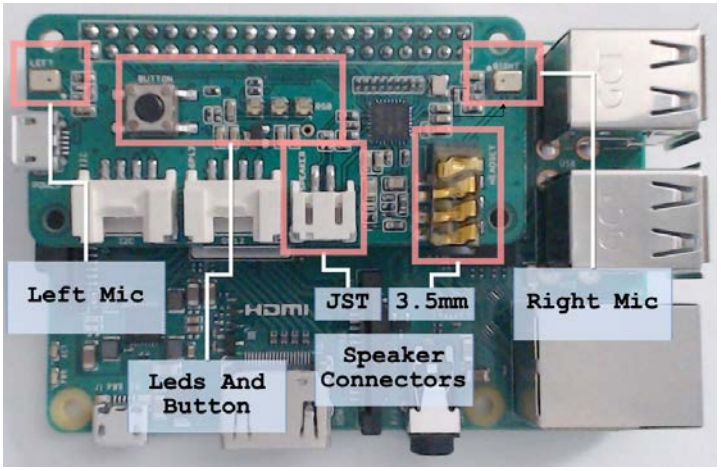


Figure 15.1 – The ReSpeaker 2-Mics Pi HAT

Figure 15.1 shows a photo of a ReSpeaker 2-Mics Pi HAT mounted on a Raspberry Pi. On the left, I've labeled the left microphone. The hat has two microphones, which are two tiny rectangular metal parts on each side. The next label is for 3 RGB LEDs and a button connected to a GPIO pin. After this are the two ways of connecting speakers – a 3.5mm jack or a JST connector. I recommend you connect a speaker to hear output from this HAT. Then, the last label highlights the right microphone.

I've chosen the ReSpeaker 2-Mic Pi HAT because it is an inexpensive device to get started on voice recognition. Very cheap USB microphones will not work well for this. There are expensive devices better supported in Mycroft, but they do not sit on the Pi as a hat. This ReSpeaker 2-Mics Pi HAT is a trade-off – great for hardware simplicity and cost but with some more software setup. Let's now look at how we physically install this HAT.

Physical installation

The ReSpeaker 2-Mics HAT will sit directly on the Raspberry Pi 4 headers with the board overhanging the Pi.

The speakers will have either a tiny two-pin connector (JST) type that fits the single two-pin socket on the board or a 3.5 mm jack. The next photograph shows the speaker plugged into it:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

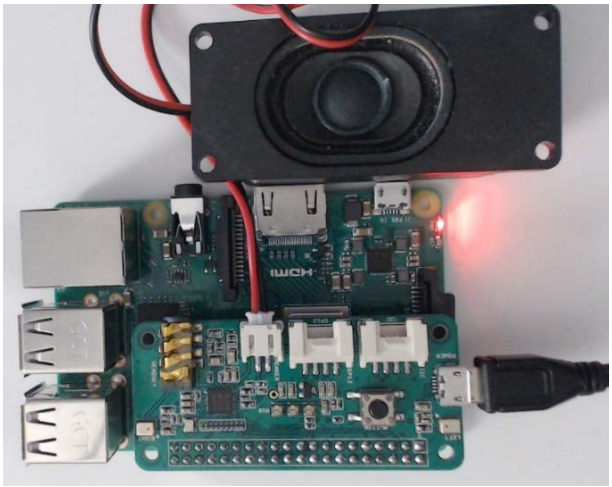


Figure 15.2 – The Mycroft Voice Assistant ReSpeaker setup

Figure 15.2 shows my Mycroft setup with the ReSpeaker 2-Mics Pi HAT set up on my desk. It is powered up, and the Raspberry Pi is lit. I've connected a speaker to it as well.

You could use a Raspberry Pi case or project box but ensure that the microphones are not covered up.

You also need an SD card and a power supply.

> **Important note**
>
> For the next few sections, I recommend using a mains power supply. Do not plug it in and power it up yet.

Now we have the hardware prepared, and it has speakers and a microphone. In the next section, we will set up Raspbian and the voice agent software.

Installing a voice agent on a Raspberry Pi

Mycroft has a Raspbian distribution prepared for this. Let's put that on an SD card:

1. Go to the Mycroft website to download the *Picroft* image: `https://mycroft-ai.gitbook.io/docs/using-mycroft-ai/get-mycroft/picroft` – this is based on Raspbian Buster. Select **stable disk image**.
2. Insert the SD card into your computer. Use the procedures from *Chapter 3*, *Exploring the Raspberry Pi*, in the *Flashing the card in balenaEtcher* section. Be sure to select the Picroft image instead of Raspbian.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Make sure this image works headlessly, enabling SSH and Wi-Fi as we did in *Chapter 4*, *Preparing a Headless Raspberry Pi for a Robot*, in the *Setting up wireless on the Raspberry Pi and enabling SSH* section.

With this SD card ready, it's time to try it out. Insert it into the voice assistant Raspberry Pi and power it up using the USB micro socket on the ReSpeaker 2-Mics Pi HAT (not the Pi).

> **Important note**
>
> Ensure you supply power via the ReSpeaker 2-Mics Pi HAT and not the Pi. This board requires power to drive its speaker. The documentation for the board suggests that if you power it through the Pi you don't get output from the speaker. See `https://wiki.seeedstudio.com/ReSpeaker_2_Mics_Pi_HAT/#hardware-overview` for details.

Its hostname starts as `picroft.local`. You use the username `pi` and password `mycroft`. Ensure it is connected to Wi-Fi, and you can reach it via SSH (PuTTY). With the Raspberry Pi started, you can start to set up Mycroft.

Installing the ReSpeaker software

When you log on, Mycroft will show you an installation guide. This will ask you some questions as listed:

1. When asked if you want a guided setup, press *Y* for yes. The Mycroft installation will download a load of updates. Leave it for 30 minutes to an hour to do so.
2. Mycroft will now ask for your audio output device:

```
HARDWARE SETUP
How do you want Mycroft to output audio:
  1) Speakers via 3.5mm output (aka 'audio jack' or 'headphone
jack')
  2) HDMI audio (e.g. a TV or monitor with built-in speakers)
  3) USB audio (e.g. a USB soundcard or USB mic/speaker combo)
  4) Google AIY Voice HAT and microphone board (Voice Kit v1)
  5) ReSpeaker Mic Array v2.0 (speaker plugged in to Mic board)
Choice [1-5]:
```

   This guided setup doesn't directly support the ReSpeaker 2-Mics Pi HAT we are using. Type `3`, to select USB speakers, which sets some basic defaults.

3. Press *Ctrl* + *C* to leave the guided setup and return to the `$` prompt.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. For the installation to work, we'll need the software on the SD card to be updated. At the prompt, type `sudo apt update -y && sudo apt upgrade -y`. The update will take some time.
5. Reboot the Pi (with `sudo reboot`) for the updates to take effect. After you reboot the Pi, `ssh` in. You will be at the guided setup again. Press *Ctrl* + *C* again.
6. Use the following commands to install the audio drivers for the ReSpeaker 2-Mics Pi HAT:

```
$ git clone https://github.com/waveshare/WM8960-Audio-HAT.git
$ cd WM8960-Audio-HAT
$ sudo ./install.sh
```

   The Git clone may take a minute or two. This board uses the WM8960 sound chip. The install script will take 20-30 minutes to finish.

7. Reboot again. Press *Ctrl* + *C* after to leave the guided mode.

   Before we move on, it's a good idea to test that we are getting audio here.

8. Type `aplay -l` to list playback devices. In the output, you should see the following:

```
card 1: wm8960soundcard [wm8960-soundcard], device 0: bcm2835-
i2s-wm8960-hifi wm8960-hifi-0 [bcm2835-i2s-wm8960-hifi wm8960-
hifi-0]
```

   This shows that it has found our card.

9. We can now test this card will play audio by getting it to play an audio file. Use this command: `aplay -Dplayback /usr/share/sounds/alsa/Front_Left.wav`.

   This command specifies the device named `playback` with the device `-D` flag, and then the file to play. The `playback` device is a default ALSA handler that ensures mixing is done and avoids issues with bitrate and channel number mismatches. There are other test audio files in `/usr/share/sounds/alsa`.

10. We can then check for recording devices with `arecord -l`. In the following output, we can see that `arecord` has found the card:

```
card 1: wm8960soundcard [wm8960-soundcard], device 0: bcm2835-
i2s-wm8960-hifi wm8960-hifi-0 [bcm2835-i2s-wm8960-hifi wm8960-
hifi-0]
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The card is now ready for use. Next, we need to show the Mycroft system how to choose this card for use.

Troubleshooting

If you haven't got audio output, there are some things you can check:

1. First, type `sudo poweroff` to turn off the Raspberry Pi. When it is off, check the connections. Ensure that the board is connected fully to the GPIO header on the Pi. Make sure you've connected the speaker to the correct port on the ReSpeaker 2-Mics Pi HAT.
2. When you power it again, ensure that you are using the power connector on the ReSpeaker 2-Mics Pi HAT, and not the Raspberry Pi.
3. If you are using the headphone slot instead of the speaker slot, you may need to increase the volume. Type `alsamixer`, select the WM8960 sound card, and turn the headphone volume up. Then try the playback tests again.
4. Make sure you have performed the `apt update` and the `apt upgrade` steps. The installation of the drivers will not work without it. You will need to reboot after this and then try reinstalling the driver.
5. When installing the driver, if the Git step fails, double-check the address you have fetched.
6. When attempting playback, the `-D` flag is case-sensitive. A lowercase `d` will not work here.

If these steps still do not help, please go to the `https://github.com/waveshare/WM8960-Audio-HAT` website, read their documentation, or raise an issue.

Now we've checked this, let's try to link the sound card with Mycroft.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting Mycroft to talk to the sound card

Now you need to connect Mycroft and the sound card. Do this by editing the Mycroft configuration file:

1. Open the Mycroft config file as root using `sudo nano /etc/mycroft/mycroft.conf`.
2. The file has lines describing various aspects of Mycroft. However, we are interested in two lines only:

```
    "play_wav_cmdline": "aplay -Dhw:0,0 %1",
    "play_mp3_cmdline": "mpg123 -a hw:0,0 %1",
```

   The first specifies that Mycroft will play wave audio files using the `aplay` command on device hardware `0,0` (the Pi headphone jack) – written as `hw:0,0`. This will be the wrong device. The second specifies it will play mp3 files using the `mpg123` command and on the same incorrect device. Using a direct hardware device may make assumptions about the format of the sound being played, so it needs to go through the mixer device. Let's fix these.
3. Edit both occurrences of `hw:0,0` to be the term `playback`. The two lines should look like this:

```
    "play_wav_cmdline": "aplay -Dplayback %1",
    "play_mp3_cmdline": "mpg123 -a playback %1",
```

4. Press *Ctrl* + *X* to write out and exit. Type *Y* for yes when asked to write out the file.
5. Reboot one more time; when you return, do not exit the guided mode.
6. Mycroft will ask to test the device. Press *T* to test the speaker. It may take a few seconds, but you will hear Mycroft speak to you. If it is a little quiet, try typing the number `9`, and test it again. An exciting moment! Press *D* to say you have done the test.
7. The guided installer will next ask about the microphone. Select **4** for **Other USB Microphone** and try the sound test. The installer will ask you to speak to the microphone, and it should play your voice back to you. Press *1* if this sounds good.
8. The guided installation will ask you about using the recommendations; select *1* to confirm you want that. There will be a series of questions about your password settings. I recommend not adding a sudo password but changing the default password for the Pi to something unique.
9. Mycroft will launch with a large section of purple installation text.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You have Mycroft configured and starting up. It can record your voice and play that back to you, and you have heard it speak a test word too. Now, it's time to start using Mycroft and see what it can do.

Starting to use Mycroft

Let's get to know Mycroft a little, and then try talking with it. We will start with the debug interface, the Mycroft client, which shows you what is going on with the system, and then we'll get into talking to it.

The Mycroft client

When you connect to Mycroft, you will see a display like the following figure:

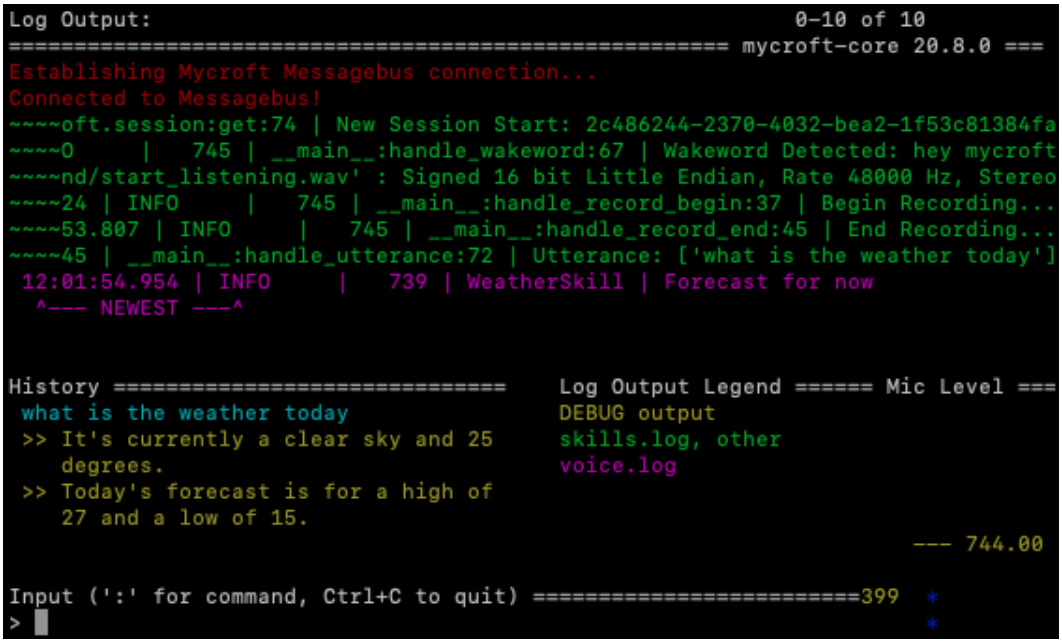


Figure 15.3 – The Mycroft client interface

The screenshot in *Figure 15.3* is the Mycroft client. It allows you to see what Mycroft is doing, but you don't need to connect to this for Mycroft to listen to you. The top right shows how many messages there are and how many you can see. In the screenshot, you can see messages **0**-**10**, out of a total of **10** messages.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The main middle section shows the messages. *Red* and *purple* messages come from the Mycroft system and plugins. If many *purple* messages are flashing by, Mycroft is installing plugins and updates, so you may need to leave it until it finishes. *Green* messages show Mycroft interacting with a user. It shows when it detects a wake word, when it starts to record, when it ends the recording, and the utterance it thinks you said. The messages are useful as if it isn't quite responding, you can check whether it's picking up the wake word and that the utterance matches what you are trying to say.

Below this, on the left, is the history. In the history, what Mycroft has processed from your utterance is in *blue*. The dialog Mycroft speaks is in *yellow*. You should hear *yellow* text repeated on the speaker; however, it can take a while if it is very busy. On the right, it shows a legend that matches colors to a log file. Further right is a microphone speaker level meter, and unless Mycroft is busy, or you are very quiet, you should see this moving up and down as it picks up noise in the room. Note – too much noise, and you may have trouble talking to it.

At the bottom of the screen is an input area, where you can type commands for Mycroft.

Give the system about 30-40 minutes to finish all the installations. If it is not responsive, it is not hung but is usually installing and compiling additional components.

Mycroft will then tell you it needs to be paired at `mycroft.ai`. You will need to register the device using the code it gives you; which you can do while Mycroft is installing. You will need to create an account there to do so (or log in if this is a second device/attempt). Please complete this before proceeding.

When you've paired Mycroft, and it finishes installing things, you can start to interact.

Talking to Mycroft

Now you should be able to speak to your voice assistant:

1. First, to get its attention, you must use the wake word *Hey Mycroft*. If it's ready (and not still busy), it will issue a speaker tone to show *Mycroft* is listening. You need to stand within about a meter of the microphones on the Raspberry Pi. It may respond with *Please wait a moment while I finish booting up*. Give it a minute and try again.
2. If you hear the tone, you can now ask it to do something. A good starting point is to tell it: *Say hello*. Mycroft should respond with *Hello* from the speaker after about 10 seconds. You will need to speak as clearly as possible. I've found that it needs you to pronounce each syllable; those *t* and *n* sounds are essential.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now that this works, you can have some fun with it! You can shorten *Hey Mycroft* to just *Mycroft*. Other things you can say include the following:

- *Hey Mycroft, what is the weather?*: This will use the weather skill and tell you the weather. It may be for the wrong location; use the `mycroft.ai` website to configure your device to your location.
- *Mycroft, what is 23 times 76*: This will use the Wolfram skill, which can handle mathematical questions.
- *Mycroft, wiki banana*: This will use a Wikipedia skill, and Mycroft will tell you what it has found out about the banana.

Try these out to get used to talking to Mycroft so it responds. It may say *I don't understand*, and the log will tell you what it heard, which can help you try to tune how you pronounce things for it.

We can now create a skill to connect Mycroft to our robot. But first, let's check for problems.

Troubleshooting

If you are not able to get Mycroft to speak or recognize talking, try the following:

- Make sure you are close enough to the microphone/loud enough. This can be checked by observing whether the mic (microphone) level goes above the dashed line in the Mycroft console.
- Ensure you have a good network connection from your Raspberry Pi. Mycroft is only going to work where you can reach the internet. See the Mycroft documentation for handling proxies. Mycroft can fail to boot correctly if the internet connection isn't great. Fixing the connection and rebooting it can help.
- Attaching a monitor while the Pi is booting may reveal error messages.
- Mycroft has a troubleshooting system starting with: *Troubleshooting and Known errors* (`https://mycroft.ai/documentation/troubleshooting/`).
- Mycroft is under active development. Taking the latest Picroft image and applying the ReSpeaker driver may help. In short, getting this installed and running is subject to change.

With Mycroft talking and responding, we need to prepare the robot for Mycroft to talk to it.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Programming a Flask API

This chapter aims to control our robot with Mycroft. To do so, we need to give our robot some way to receive commands from other systems. An **Application Programming Interface** (**API**) on a server lets us decouple systems like this to send commands across the network to another and receive a response. The Flask system is ideally suited to building this.

Web-based APIs have endpoints that other systems make their requests to and roughly map to functions or methods in a Python module. As you'll see, we map our API endpoints directly to functions in the Python `robot_modes` module.

Before we get into building much, let's look at the design of this thing – it will also reveal how Mycroft works.

Overview of Mycroft controlling the robot

The following diagram shows how a user controls a robot via Mycroft:

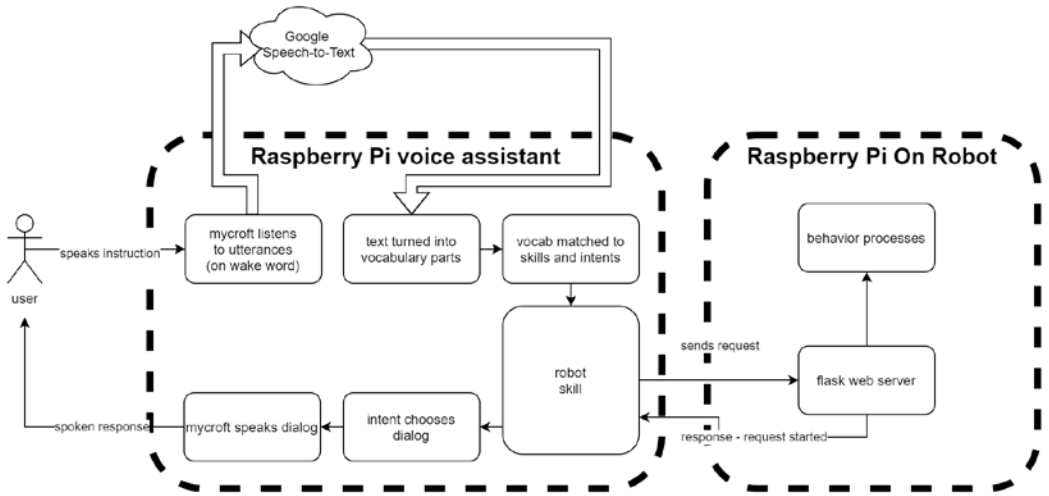


Figure 15.4 – Overview of the robot skill

The diagram in *Figure 15.4* shows how data flows in this system:

1. On the left, it starts with the user speaking an instruction to Mycroft.
2. On recognizing the wake word, Mycroft sends the sound to the Google STT engine.
3. Google STT returns text, an utterance, which Mycroft matches against vocabulary in skills/intents. We'll dig more into these later.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. This triggers intents in the robot skill, which we will build. The robot skill will send a request to the Raspberry Pi in the robot, on the right, as a request to a Flask control API (web) server.
5. That control API server will start the robot processes and respond to say it's done so.
6. The robot skill will choose dialog to say it has completed and sends this to Mycroft.
7. Mycroft will then speak this response to the user.

At this point, we are going to build the Flask server on the robot. You have seen Flask before in the visual processing chapters and have already installed this library.

Starting a behavior remotely

We will use HTTP and a web server for this, as it's simple to send requests to, so we can build other ways to control the robot remotely. HTTP sends requests in a URL—first, the `http://` protocol identifier; a server hostname, `myrobot.local`; a path, `/mode/foo`; and it may have additional parameters after that. We use the path of the URL to determine what our robot does.

As we have done with other systems, we create a few logical sections and blocks to handle different aspects of this:

- Code to manage the robot's modes and to start and stop known scripts. It can also give us a list of those known scripts.
- A web server to handle requests over the network.

We'll need to build the mode manager first.

Managing robot modes

We can manage modes by starting and stopping our behavior scripts as subprocesses. Let's make a configuration to tell the mode manager about the modes. This configuration maps a mode name to a file—a Python file. Note that we are specifying a list of files and not inferring it. Although we could take our mode/path section and add `.py` to get a file, this would be bad for two reasons:

- It would couple us directly to script names; it would be nice if we could change underlying scripts for the same mode name.
- Although the robot is not a secure environment, allowing arbitrary subprocesses to run is very bad; restricting it keeps the robot a little more secure.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's start building it:

1. Create a file called `robot_modes.py`. This file contains a class called `RobotModes` that handles robot processes.
2. The file starts with some imports and the top of the class definition:

```
import subprocess

class RobotModes(object):
```

3. Next, we create a few mode mappings, mapping a mode name to a filename:

```
    mode_config = {
        "avoid_behavior": "avoid_with_rainbows.py",
        "circle_head": "circle_pan_tilt_behavior.py",
        "test_rainbow": "test_rainbow.py"
    }
```

   The mode name is a short name, also known as a *slug*, a compromise between human-readable and machine-readable – they are usually restricted to lowercase and underscore characters and are shorter than a full English description. Our filenames are relatively close to slug names already.

4. With the fixed configuration aside, this class is also managing running behaviors as processes. It should only run one at a time. Therefore, we need a member variable to keep track of the current process and check whether it is running:

```
    def __init__(self):
        self.current_process = None
```

5. We should be able to check whether something is already running or it has completed:

```
    def is_running(self):
        return self.current_process and self.current_
process.returncode is None
```

   Python's `subprocess` is a way of running other processes and apps from within Python. We check whether we have a current process, and if so, whether it is still running. Processes have a return code, usually to say whether they completed or failed. However, if they are still running, it will be `None`. We can use this to determine that the robot is currently running a process.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. The next function is running a process. The function parameters include a mode name. The function checks whether a process is running, and if not, starts a process:

```
    def run(self, mode_name):
        if not self.is_running():
            script = self.mode_config[mode_name]
            self.current_process = subprocess.
Popen(["python3", script])
            return True
        return False
```

> **Important note**
>
> Before we run a new process, we need to check that the previous behavior has stopped. Running two modes simultaneously could have quite strange consequences, so we should be careful not to let that happen.

We use `self.mode_config` to map `mode_name` to a script name. We then use `subprocess` to start this script with Python. `Popen` creates a process, and the code stores a handle for it in `self.current_process`. This method returns `True` if we started it, and `False` if one was already running.

7. The class needs a way to ask it to stop a process. Note that this doesn't try to stop a process when it is not running. When we stop the scripts, we can use Unix signals, which let us ask them to stop in a way that allows their `atexit` code to run. It sends the `SIGINT` signal, which is the equivalent of the *Ctrl* + *C* keyboard combination:

```
    def stop(self):
        if self.is_running():
            self.current_process.send_signal( subprocess.
signal.SIGINT)
            self.current_process = None
```

After we have signaled the process, we set the current process to `None` – throwing away the handle.

We now have code to start and stop processes, which also maps names to scripts. We need to wrap it in a web service that the voice agent can use.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Programming the Flask control API server

We've used Flask previously to make the web server for our visual processing behaviors. We are going to use it for something a bit simpler this time, though.

As we saw with the start and stop buttons in the image servers, Flask lets us set up handlers for links to perform tasks. Let's make a script that acts as our control web service, which uses `Flask` and our `RobotModes` object.

Let's build this by following these steps:

1. Create a script called `control_server.py`. We can start by importing Flask and our robot modes:

```
from flask import Flask
from robot_modes import RobotModes
```

2. Now, we create a Flask app to contain the routes and an instance of our `RobotModes` class from before:

```
app = Flask(__name__)
mode_manager = RobotModes()
```

3. Next, we need a route, or API endpoint, to run the app. It takes the mode name as part of the route:

```
@app.route("/run/<mode_name>", methods=['POST'])
def run(mode_name):
    mode_manager.run(mode_name)
    return "%s running"
```

   We return a running confirmation.

4. We also need another API endpoint to stop the running process:

```
@app.route("/stop", methods=['POST'])
def stop():
    mode_manager.stop()
    return "stopped"
```

5. Finally, we need to start the server up:

```
app.run(host="0.0.0.0", debug=True)
```

This app is ready to start for speech control.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

> **Tip**
>
> What is a URL? You have already used these with other web services in the book. A URL, or uniform resource locator, defines how to reach some kind of resource; it starts with a protocol specification—in this case, `http` for a web (hypertext) service. This is followed by a colon (`:`) and then two slashes `//` with a hostname or host address—the network address of the Raspberry Pi the resource will be on. As a host can have many services running, we can then have a port number, with a colon as a separator—in our case, `:5000`. After this, you could add a slash `/` then select a specific resource in the service.

We can test this now:

6. Power up the robot and copy both the `control_server.py` and `robot_modes.py` files to it.
7. SSH into the robot and start the control server with `python3 control_server.py`. You should see the following:

```
$ python3 control_server.py
 * Serving Flask app "control_server" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production
environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

8. Now create another `ssh` window into the Mycroft Raspberry Pi – we can test that one talks to the other. Press *Ctrl* + *C* once into `pi@picroft.local` to get to the Linux command line (the `$` prompt).
9. The `curl` command is frequently used on Linux systems like the Raspberry Pi to test servers like this. It makes requests to web servers, sending/receiving data, and displaying the result. It's perfect for testing HTTP control APIs like this.

   We intend to make a `post` request. Type this command:

```
curl -X POST http://myrobot.local:5000/run/test_rainbow
```

   This should start the rainbows turning on and off, using the code from *Chapter 9*, *Programming RGB Strips in Python*. The `curl` command specifies that we are using the `POST` method to make a request, then a URL with the port, the robot hostname, then the instruction `run`, and then the mode name.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

10. You can stop the LEDs with `curl -X POST http://myrobot.local:5000/stop`. This URL has the instruction `stop`. The robot LED rainbow should stop.

    Notice how both these URLs have `http://myrobot.local:5000/` at their start. The address may be different for your robot, depending on your hostname. This is a base URL for this control server.

11. You can press *Ctrl* + *C* to stop this.

We can use this to build our Mycroft behaviors, but let's check for any problems before carrying on.

Troubleshooting

If this isn't working for you, we can check a few things to see what happened:

- If you receive any syntax errors, check your code and try again.
- Please verify that your robot and the device you are testing from have internet availability.
- Note that when we are starting the subprocess, we are starting Python 3. Without the `3`, other unexpected things will happen.
- First, remember the control server is running on the Raspberry Pi 3A+ on the robot. You will need to substitute it for your robot's address in the `curl` commands.
- Ensure you have installed Flask, as shown in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*.
- Make sure you have copied both the control server and the robot mode scripts to the robot. You will also need the code from *Chapter 9*, *Programming RGB Strips in Python* installed on the robot to run this test.

Now we've tested the control server, you can power down the Pi. There's some more code to write! Let's tie this into Mycroft.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Programming a voice agent with Mycroft on the Raspberry Pi

The robot backend provided by the Flask control system is good enough to create our Mycroft skill with.

In *Figure 15.4*, you saw that after you say something with the wake word, upon waking, Mycroft will transmit the sound you made to the Google STT system. Google STT will then return the text.

Mycroft will then match this against vocabulary files for the region you are in and match that with intents set up in the skills. Once matched, Mycroft will invoke an intent in a skill. Our robot skill has intents that will make network (HTTP) requests to the Flask control server we created for our robot. When the Flask server responds to say that it has processed the request (perhaps the behavior is started), the robot skill will choose a dialog to speak back to the user to confirm that it has successfully carried out the request or found a problem.

We'll start with a simple skill, with a basic intent, and then you can expand this to perform more. I've picked the rainbow LEDs test (`test_leds` from *Chapter 9*, *Programming RGB Strips in Python*) because it is simple.

It's worth noting that the time taken to get the speech processed by Google means that this is not suitable for stopping a robot in a hurry; the voice recognition can take some time. You could consider using `GPIOZero` in the intent and a `when_pressed` handler to trigger the control server's stop handler.

Building the intent

We can start with the intent, then look at some vocabulary. To build it, we will use a library built into Mycroft named `adapt`:

1. Create a folder called `my-robot-skill`, which we will work in to build the Mycroft skill.
2. The main intent file will be an `__init__.py` file in this folder. This filename means that Python will treat the whole folder like a Python library, called a **package**. Let's start putting some imports in `my-robot-skill/__init__.py`:

```
from adapt.intent import IntentBuilder
from mycroft import MycroftSkill, intent_handler
from mycroft.util.log import LOG

import requests
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The imports include `IntentBuilder` to build and define intents around vocabulary. `MycroftSkill` is a base class to plug our code into Mycroft. `intent_handler` marks which parts of our code are intents, associating the code with `IntentBuilder`. We import `LOG` to write information out to the Mycroft console and see problems there.

The last import, `requests`, is a tool to talk to our control server in Python remotely.

3. Next, we will define our skill from the `MycroftSkill` base. It needs to set up its parent and prepare settings:

```
class MyRobot(MycroftSkill):
    def __init__(self):
        super().__init__()
        self.base_url = self.settings.get("base_url")
```

The Python keyword `super` calls a method from a class we've made our base; in this case, `__init__` so we can let it set things up.

The only setting we have is a `base_url` member for our control server on the robot. It is consulting a settings file, which we'll see later. It's usually a good idea to separate the configuration from the code.

4. The next thing we need is to define an intent. We do so with a `handle_test_rainbow` method – but you need to decorate it using `@intent_handler`. In Python, decorating wraps a method in further handling – in this case, making it suitable for Mycroft:

```
    @intent_handler(IntentBuilder("")
                    .require("Robot")
                    .require("TestRainbow"))
    def handle_test_rainbow(self, message):
```

The `intent_handler` decorator takes some parameters to configure the vocabulary we will use. We will define vocabulary in files later. We require a vocabulary matching *robot* first, then another part matching *TestRainbow* – which could match a few phrases.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Next, this skill should make the request to the robot – using `requests.post`:

```
        try:
            requests.post(self.base_url + "/run/test_
rainbow")
```

   This segment posts to the URL in the `base_url` variable, plus the `run` instruction and the `test_rainbow` mode.

6. We need Mycroft to say something, to say that it has told the robot to do something here:

```
            self.speak_dialog('Robot')
            self.speak_dialog('TestingRainbow')
```

   The `speak_dialog` method tells Mycroft to pick something to say from dialog files, which allows it to have variations on things to say.

7. This request could fail for a few reasons, hence the `try` in the code snippet before last. We need an `except` to handle this and speak a dialog for the user. We also `LOG` an exception to the Mycroft console:

```
        except:
            self.speak_dialog("UnableToReach")
            LOG.exception("Unable to reach the robot")
```

   We are treating many error types as `Unable to reach the robot`, while not inspecting the result code from the server other than if the voice skill contacted the robot.

8. This file then needs to provide a `create_skill` function outside of the class, which Mycroft expects to find in skill files:

```
def create_skill():
    return MyRobot()
```

The code is one part of this system, but we need to configure this before using it.

The settings file

Our intent started by loading a setting. We will put this in `my-robot-skill/settingsmeta.json`, and it defines the base URL for our control server.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Please use the hostname/address of your robot Raspberry Pi if it is different. This file is a little long for this one setting, but will mean that you can configure the URL later if need be:

```
{
    "skillMetadata": {
        "sections": [
            {
                "name": "Robot",
                "fields": [
                    {
                        "name": "base_url",
                        "type": "text",
                        "label": "Base URL for the robot
control server",
                        "value": "http://myrobot.local:5000"
                    }
                ]
            }
        ]
    }
}
```

We have now set which base URL to use, but we need to configure Mycroft to load our skill.

The requirements file

Our skill uses the `requests` library. When Mycroft encounters our skill, we should tell it to expect this. In Python, requirements files are the standard way to do this. Put the following in `my-robot-skill/requirements.txt`:

```
requests
```

This file is not unique to Mycroft and is used with many Python systems to install libraries needed by an application.

Now we need to tell Mycroft what to listen for, with vocabulary.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Creating the vocabulary files

To define vocabularies, we need to define vocabulary files. You need to put them in a folder following the format `my-robot-skill/vocab/<IETF language and locale>`. A language/locale means we should be able to define a vocabulary for variants such as `en-us` for American English and `zn-cn` for simplified Chinese; however, at the time of writing, `en-us` is the most supported Mycroft language. Parts of the community are working on support for other languages.

You define each intent with one or more vocabulary parts matching vocabulary files. Vocabulary files have lines representing ways to phrase the intended utterance. These allow a human to naturally vary the way they say things, something people notice when a machine fails to respond to a slightly different way of asking for something. There is a bit of a trick in thinking up similar phrases for the vocabulary files.

We need two vocabulary files for our intent—one for `robot` synonyms and one for `TestRainbow` synonyms:

1. Create the folder `vocab` under `my-robot-skill`, and then the `en-us` folder under that.
2. Make a file there with the path and name `my-robot-skill/vocab/en-us/robot.voc`.
3. Add some phrases for *asking the robot to do something*:

```
robot
my robot
ask robot to
tell the robot to
```

   Mycroft will match these phrases where we have said `robot` in the intent handler.

4. Let's create the vocabulary for testing the rainbow. Put it into `my-robot-skill/vocab/en-us/TestRainbow.voc`:

```
test rainbow
test the leds
deploy rainbows
turn on the lights
```

> **Important note**
>
> Note that the vocabulary filename's capitalization must match the intent builder; I've then used the convention of capitalizing the non-shared vocab parts.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Inevitably, when you test this, you will eventually try to say a sensible sounding phrase that isn't there. Mycroft will tell you *Sorry, I don't understand*, and you will add another expression to the vocabularies above.

Dialog files

We also want to define the phrases Mycroft will say back to you. We have three phrases that our intent requires so far. These go into the `my-robot-skill/dialog/en-us` folder with a similar structure to vocabulary files. Let's build them:

1. Under `my-robot-skill`, create the folder `dialog`, and then under this, the folder `en-us`.
2. In the folder, create the file with the path `my-robot-skill/dialog/en-us/Robot.dialog`. We can add some phrases for that here:

```
The Robot
Robot
```

3. The next dialog we need is `TestRainbow.dialog` in the same folder:

```
is testing rainbows.
is deploying rainbows.
is starting rainbows.
is lighting up.
```

4. Since we have an error handler, we should also create `UnableToReach.dialog`:

```
Sorry I cannot reach the robot.
The robot is unreachable.
Have you turned the robot on?
Is the control server running on the robot?
```

By defining multiple possible dialogs, Mycroft will randomly pick one to make itself less repetitive. We've now seen how to make vocabulary phrases and dialog phrases. Let's just recap what we should have.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Current skill folder

Our skill folder should look like the following screenshot:

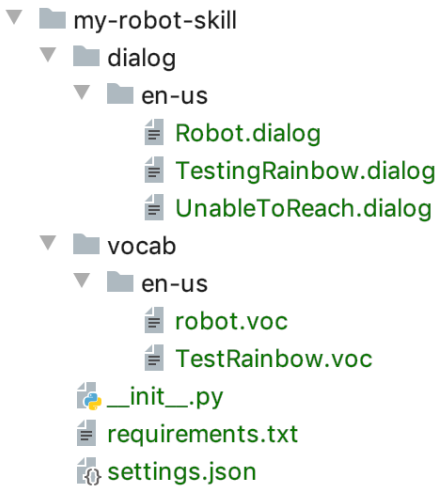


Figure 15.5 – Screenshot of the robot skill folder

In *Figure 15.5,* we see a screenshot showing the skill in a folder called `my-robot-skill`. This skill folder has the `dialog` folder, with the `en-us` subfolder and the three dialog files here. Below that is the `vocab` folder, with the `en-us` folder and two vocab files. Below the `vocab` folder, we have `__init__.py` defining the intents, requirements for Mycroft to install it, and a settings file. Whew – we've created a lot here, but it will be worth it!

We are going to now need to upload this whole folder structure to our robot:

1. Using SFTP (FileZilla), upload this folder to your Mycroft Pi, in the `/opt/mycroft/skills` folder.
2. Mycroft will automatically load this skill; you will see purple text for this flash past as it does the install.
3. If you need to update the code, uploading the files to this location again will cause Mycroft to reload it.

   Any problems loading or using the skill will be shown on the Mycroft output. You can also find the result in `/var/log/mycroft/skills.log`—the `less` Linux tool is useful for looking at log output like this, using *Shift* + *G* to jump to the end of the file or typing `/myrobot` to jump to its output.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can also use `tail -f /var/log/mycroft/skills.log` to see problems as they happen. Use *Ctrl* + *C* to stop.

4. Now, power up the robot, `ssh` in, and start the control server with `python3 control_server.py`.
5. You can then try out your skill with Mycroft: *Tell the robot to turn on the lights*.
6. Mycroft should beep to show the user it's awake and, once it has got the words from speech to text, it will send `/run/test_rainbow` to the control server on the robot. You should hear Mycroft say one of the dialog phrases, such as *The robot is testing rainbows* and see the LEDs light up.

Troubleshooting

If you encounter problems making the intent respond, please try the following:

- First, check the syntax and indenting of the previous Python code.
- Ensure that your robot and the voice assistant Raspberry Pi are on the same network; I've found this problematic with some Wi-Fi extenders, and IP addresses are needed instead of `myrobot.local`. Use the `settingsmeta.json` file to configure this.
- Ensure you have copied over the whole structure – with the `vocab`, `dialog`, `settingsmeta.json`, and `__init__.py` – to the `/opt/mycroft/skills` folder on the voice assistant Raspberry Pi.
- If your settings were incorrect, you will need to change them on the `https://account.mycroft.ai/skills` page. Look for the `My Robot` skill and change it here. You will need to save the change and may need to restart Mycroft or wait a few minutes for this to take effect.
- Ensure the way you have spoken to Mycroft matches your vocabulary files – it will not recognize your words otherwise.
- You can also type phrases into the Mycroft console if you are having trouble with it recognizing your voice.

We've got our first intent to work! You've been able to speak to a voice assistant, and it has instructed the robot what to do. However, we've now started the LEDs flashing, and the only way to stop them is with that inconvenient `curl` command. We should probably fix that by adding another intent.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Adding another intent

Now we have our skill, adding a second intent for it to stop becomes relatively easy, using another of the endpoints in our robot's control server.

Vocabulary and dialog

We need to add the vocabulary and dialog so our new intent can understand what we are saying and has a few things to say back:

1. We will need to create the `stop` vocabulary; we can put this in `my-robot-skill/vocab/en-us/stop.voc`:

```
stop
cease
turn off
stand down
```

2. We need a dialog file for Mycroft to tell us the robot is stopping in `my-robot-skill/dialog/en-us/stopping.dialog`:

```
is stopping.
will stop.
```

These will do, but you can add more synonyms if you think of them.

Adding the code

Now we need to add the intent code to our skill:

1. We will put this into the `MyRobot` class in `my-robot-skill/__init__.py`:

```
    @intent_handler(IntentBuilder("")
                    .require("Robot")
                    .require("stop"))
    def handle_stop(self, message):
        try:
            requests.post(self.base_url + "/stop")
            self.speak_dialog('Robot')
            self.speak_dialog('stopping')
        except:
            self.speak_dialog("UnableToReach")
            LOG.exception("Unable to reach the robot")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This code is almost identical to the test rainbows intent, with the `stop` vocabulary, the handler name (which could be anything – but must not be the same as another handler), and the URL endpoint.

Identical code like that is ripe for refactoring. Refactoring is changing the appearance of code without affecting what it does. This is useful for dealing with common/repeating code sections or improving how readable code is. Both the intents have the same try/catch and similar dialog with some small differences.

2. In the same file, add the following:

```
    def handle_control(self, end_point, dialog_verb):
        try:
            requests.post(self.base_url + end_point)
            self.speak_dialog('Robot')
            self.speak_dialog(dialog_verb)
        except:
            self.speak_dialog("UnableToReach")
            LOG.exception("Unable to reach the robot")
```

This will be a common handler. It takes `end_point` as a parameter and uses that in its request. It takes a `dialog_verb` parameter to say after the `Robot` bit. All of the other dialog and error handling we saw before is here.

3. The two intents now become far simpler. Change them to the following:

```
    @intent_handler(IntentBuilder("")
                    .require("Robot")
                    .require("TestRainbow"))
    def handle_test_rainbow(self, message):
        self.handle_control('/run/test_rainbow',
'TestingRainbow')

    @intent_handler(IntentBuilder("")
                    .require("Robot")
                    .require("stop"))
    def handle_stop(self, message):
        self.handle_control('/stop', 'stopping')
```

Adding new intents is now easier as we can reuse `handle_control`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Running with the new intent

You can now upload the folder structure again—since the `vocab`, `dialog`, and `__init__` files have changed. When you do so, note that Mycroft will automatically reload the changed skill (or show any problems trying to do so), so it is immediately ready to use.

Try this out by saying *Mycroft, tell the robot to stop*.

You've now added a second intent to the system, defining further vocabulary and dialogs. You've also refactored this code, having seen some repetition. You've now got the beginnings of voice control for your robot.

Summary

In this chapter, you learned about voice assistant terminology, speech to text, wake words, intents, skills, utterances, vocabulary, and dialog. You considered where you would install microphones and speakers and whether they should be on board a robot.

You then saw how to physically install a speaker/microphone combination onto a Raspberry Pi, then prepare software to get the Pi to use it. You installed Picroft – a Mycroft Raspbian environment, getting the voice agent software.

You were then able to play with Mycroft and get it to respond to different voice commands and register it with its base.

You then saw how to make a robot ready for an external agent, such as a voice agent to control it with a Flask API. You were able to create multiple skills that communicate with a robot, with a good starting point for creating more.

In the next chapter, we will bring back out the IMU we introduced in *Chapter 12*, *IMU Programming with Python*, and get it to do more interesting things – we will smooth and calibrate the sensors and then combine them to get a heading for the robot, programming the robot to always turn north.

Exercises

Try these exercises to get more out of this chapter and expand your experience:

- Try installing some other Mycroft skills from the Mycroft site and playing with them. Hint: say *Hey Mycroft, install pokemon*.
- The robot mode system has a flaw; it assumes that a process you've asked to stop does stop. Should it wait and check the return code to see if it has stopped?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- An alternative way to implement the robot modes might be to update all the behaviors to exit cleanly so you could import them instead of running in subprocesses. How tricky would this be?
- While testing the interactions, did you find the vocabulary wanting? Perhaps extend it with phrases you might find more natural to start the different behaviors. Similarly, you could make dialogs more interesting too.
- Add more intents to the skill, for example, wall avoiding. You could add a stop intent, although the response time may make this less than ideal.
- Could the RGB LEDs on the ReSpeaker 2-Mics Pi HAT be used? The project `https://github.com/respeaker/mic_hat` has an LED demonstration.

With these ideas, there is plenty of room to explore this concept more. Further reading will help too.

Further reading

Please refer to the following for more information:

- *Raspberry Pi Robotic Projects*, *Dr. Richard Grimmett*, *Packt Publishing*, has a chapter on providing speech input and output.
- *Voice User Interface Projects*, *Henry Lee*, *Packt Publishing*, focuses entirely on voice interfaces to systems. It shows you how to build chatbots and applications with the Alexa and Google Home voice agents.
- *Mycroft AI – Introduction Voice Stack* – a whitepaper from Mycroft AI gives more detail on how the Mycroft stack works and its components.
- Mycroft has a large community that supports and discusses the technology at `https://community.mycroft.ai/`. I recommend consulting the troubleshooting information of this community. Mycroft is under active development and has both many quirks and many new features. It's also an excellent place to share skills you build for it.
- Seeed Studio, the ReSpeaker 2-Mics Pi HAT creators, host documentation and code for this device, along with bigger four and six-microphone versions at `https://github.com/respeaker/seeed-voicecard`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

16
Diving Deeper with the IMU

In *Chapter 12*, *IMU Programming with Python*, we read data from an **inertial measurement unit** (**IMU**). We've now learned a bit more about processing sensor data, using math and pipelines to make decisions.

In this chapter, we will learn how to get calibrated data from the IMU, combine data from the sensors, and use this to make a robot have absolute orientation-based behavior. On the way, we'll see algorithms for better precision/speed or accuracy.

By the end of the chapter, you will be able to detect a robot's absolute orientation, display it on a screen, and incorporate this with the **Proportional-Integral-Derivative** (**PID**) behaviors.

In this chapter, we're going to cover the following main topics:

- Programming a virtual robot
- Detecting rotation with the gyroscope
- Detecting pitch and roll with the accelerometer
- Detecting a heading with the magnetometer

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Getting a rough heading from the magnetometer
- Combining sensors for orientation
- Driving a robot from IMU data

Technical requirements

For this chapter, you will need the following items:

- The robot from at least *Chapter 14*, *Line-Following with a Camera in Python*
- The robot code from *Chapter 14*, *Line-Following with a Camera in Python*, at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter14`
- The IMU code from *Chapter 12*, *IMU Programming with Python*, at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter12`
- A wide driving space without many magnets
- A magnetic compass

For the complete code for this chapter, go to `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter16`.

Check out the following video to see the Code in Action: `https://bit.ly/2LztwOO`

Programming a virtual robot

We will first detect our robot's orientation; it would be useful to show this as a 3D robot model. This part builds upon the *Representing coordinate and rotation systems* section in *Chapter 12*, *IMU Programming with Python*. In this section, we will construct a simple model of our robot in VPython.

Modeling the robot in VPython

We'll use shapes, known as **primitives**, to model the robot. They have a position, rotation, size, and color. The height-and-width parameters match the VPython-world coordinate system (see *Figure 12.14 – The robot body coordinate system* in *Chapter 12*, *IMU Programming with Python*), so we must rotate things to match the robot body coordinate system.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

First, we need to collect some robot measurements. The following diagram shows where they are. Once the major measurements are made, estimates can be used for smaller measurements:

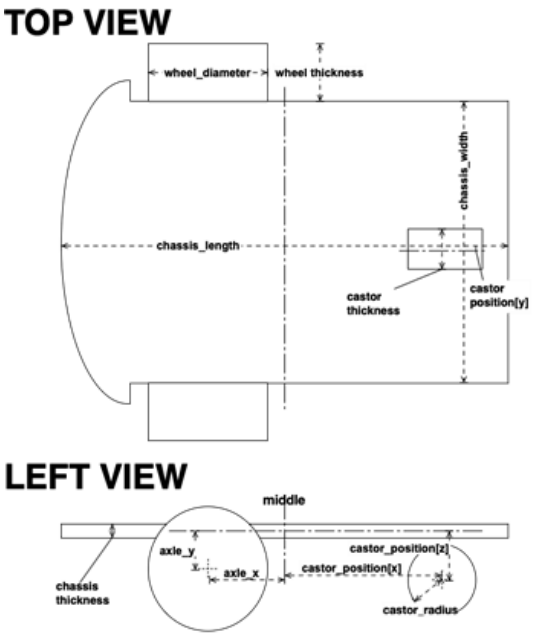


Figure 16.1 – Measurements for the virtual robot

Figure 16.1 shows the measurements across the robot. A top view and a left view show to cover the different aspects. This includes the width and height of the base—note that we are treating it as a rectangle for this purpose. The wheels' size and position, along with the castor-wheel size and position, are needed. Measure or guess these for your robot. For our purposes, guesses are good enough. Positions come from the middle of the chassis.

Let's write the code to make the basic shape, as follows:

1. Create a file called `virtual_robot.py` and start it by adding in the `vpython` import and our robot view, as follows:

```
import vpython as vp
from robot_pose import robot_view
```

2. We'll put the virtual bot in a function ready to use in a few different behaviors, like this:

```
def make_robot():
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We put the robot's measurements from *Figure 16.1* in variables. I've used **millimeters** (**mm**) for all of them. The code is shown in the following snippet:

```
chassis_width = 155
chassis_thickness = 3
chassis_length = 200
wheel_thickness = 26
wheel_diameter = 70
axle_x = 30
axle_z = -20
castor_position = vp.vector(-80, -6, -30)
castor_radius = 14
castor_thickness = 12
```

4. The base is a box with the position defaulting to `(0, 0, 0)`. The code is shown in the following snippet:

```
base = vp.box(length=chassis_length,
              height=chassis_thickness,
              width=chassis_width)
```

5. Rotate this box to match the body coordinate system by 90 degrees around the *x* axis, putting the *z* axis up, as follows:

```
base.rotate(angle=vp.radians(90),
            axis=vp.vector(1, 0, 0))
```

6. We'll use two cylinders for the wheels. The distance from each wheel to the middle is roughly half the chassis width. Let's use it to create the wheels' *y* positions, as follows:

```
wheel_dist = chassis_width/2
```

7. We set wheel positions to line up with the ends of the motor axles. The left wheel has a *y* coordinate; `-wheel_dist` moves it left of the platform, as illustrated in the following code snippet:

```
wheel_l = vp.cylinder(radius=wheel_diameter/2,
      length=wheel_thickness,
      pos=vp.vector(axle_x, -wheel_dist, axle_z),
      axis=vp.vector(0, -1, 0))
```

The VPython `cylinder` axis says which way it is pointing. We set *y* to *-1* to point it left.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Now, we set the right wheel, with a positive `wheel_dist` and *y* as *1* for the axis so that it points to the right, as illustrated in the following code snippet:

```
    wheel_r = vp.cylinder(radius=wheel_diameter/2,
          length=wheel_thickness,
          pos=vp.vector(axle_x, wheel_dist, axle_z),
          axis=vp.vector(0, 1, 0))
```

9. I've used a cylinder for the rear castor wheel, as illustrated in the following code snippet:

```
    castor = vp.cylinder(radius=castor_radius,
          length=castor_thickness,
          pos=castor_position,
          axis=vp.vector(0, 1, 0))
```

10. Now, we join all of these parts into a compound object—a single 3D object, like this:

```
    return vp.compound([base, wheel_l, wheel_r, castor])
```

11. For testing it, let's make a tiny `main` section. This code checks if you've launched it directly, so the following code won't run when we import the virtual robot as a library:

```
if __name__ == "__main__":
```

12. Set the view, putting the camera just in front of the robot, as follows:

```
    robot_view()
```

13. We'll add axes to show where things are, like this:

```
    x_arrow = vp.arrow(axis=vp.vector(200, 0, 0),
color=vp.color.red)
    y_arrow = vp.arrow(axis=vp.vector(0, 200, 0),
color=vp.color.green)
    z_arrow = vp.arrow(axis=vp.vector(0, 0, 200),
color=vp.color.blue)
```

14. And then, we'll draw the robot, as follows:

```
    make_robot()
```

15. Upload and test this code with `vpython virtual_robot.py`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

16. Open up a browser to port `9020` on your robot to see your virtual robot. You should see a figure like the following:

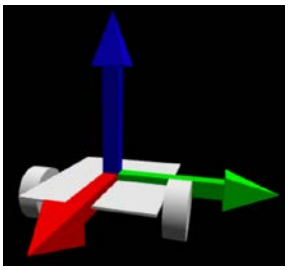


Figure 16.2 – Screenshot of the 3D virtual robot from VPython

In *Figure 16.2*, we can see the *x* axis facing forward in red, the *y* axis going right in green, and the *z* axis going up in blue. This follows a right-hand-rule coordinate system. It shows the virtual robot viewed from the front, with a wheel on either side. It's gray, boxy, and basic, but it will do for our remaining experiments.

17. You can right-click and drag this around to get another view. The mouse wheel will also zoom in or out. The following screenshot shows the rear castor:

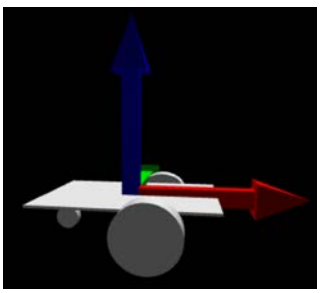


Figure 16.3 – A different view of the virtual robot

Figure 16.3 shows a left-hand view of this virtual robot.

Close the browser tab, then press *Ctrl* + *C* to finish this program when done. Let's just check you've been able to follow along.

Troubleshooting

If you haven't got this to work, let's check a few things, as follows:

1. If you receive errors saying **no such module vpython**, ensure that VPython is installed. Follow the steps in *Chapter 12*, *IMU Programming with Python*, in the *Reading the temperature* section. You need the code from the whole of *Chapter 12*, *IMU Programming with Python*, for this chapter to work.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. If you receive errors saying **no such command vpython**, ensure you have followed the *Simplifying the VPython command line* section from *Chapter 12*, *IMU Programming with Python*. The alias for VPython is necessary to be able to see a display.
3. If you see syntax errors, please check your code for typos.
4. If you cannot reach the display (and have checked *Step 1*), ensure you use port `9020` on your robot (mine is `http://myrobot.local:9020`).
5. Be patient—VPython can take a minute or two to start up.

Now that we have a visual robot to play with, we can revisit the gyroscope and try to make the onscreen robot move like our real robot.

Detecting rotation with the gyroscope

We've had some raw data from the gyroscope, but to use it more effectively, we'll need to perform two operations, calibrating the gyroscope, and then integrating it, as shown in the following diagram:

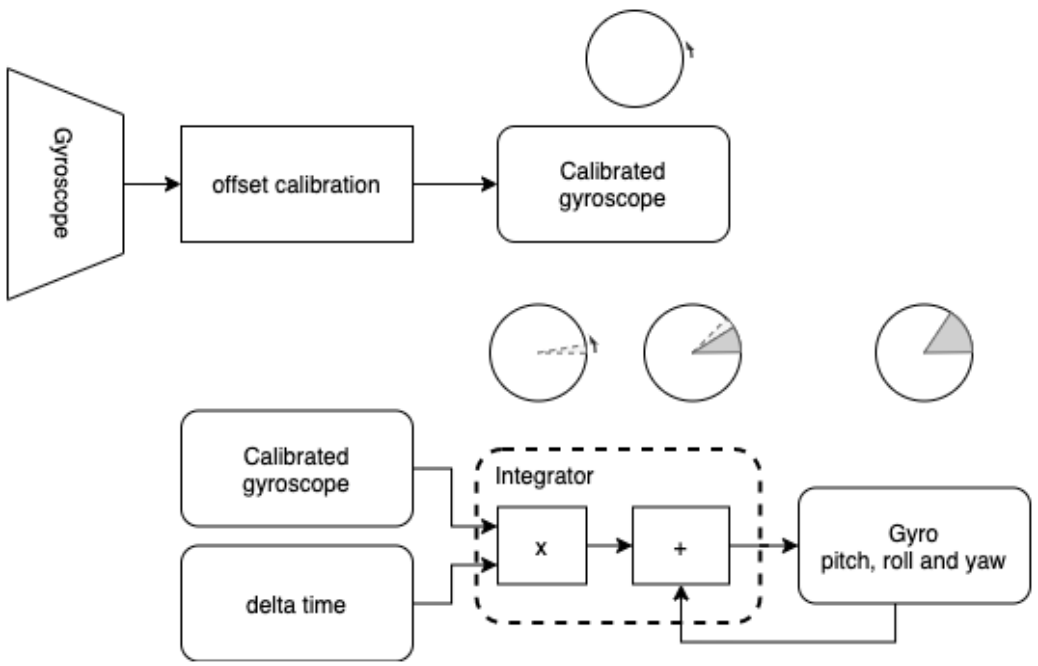


Figure 16.4 – The gyroscope data flow

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 16.4 shows the data flow, and we will look closer at the concepts later in this section. The first operation is shown at the top, which shows the gyroscope data going through an offset calibration to take out errors. This gives us a calibrated gyroscope, with a rate of change in degrees per second (per axis)—shown by the arrow around the circle. The gyroscope makes a relative measurement.

The lower part of the diagram is the second operation, combining delta time with the calibrated gyroscope (gyro). We need to **integrate** that to find an absolute measurement. An integrator multiplies an input value by delta time and adds this to a previous result. In this case, we multiply the gyroscope rate by delta time to produce a movement for that period (shown by the multiplication symbol in a box). The circle above it has a small slither of pie with dashed lines, denoting the amount moved.

We add the movement to the last value for that axis, shown by the plus symbol box. The circle above it shows a solid gray pie segment for the existing position and a new segment with dashed lines. When added, they make the total value for that axis—shown by the circle with a large, solid gray pie segment representing the addition's result. The system feeds the pitch, roll, or yaw result back into the next cycle.

Before we do this, we need to correct the errors in the gyroscope.

Calibrating the gyroscope

As they come from the factory, **microelectromechanical systems** (**MEMS**) gyroscopes usually have minor flaws that cause them to give slightly off readings. These flaws will cause drift in our integration.

We can make code to detect these and compensate; we call this **calibration**. Proceed as follows:

1. Create a file named `calibrate_gyro.py`.
2. We need VPython for vectors, time for a little sleep, and to set up the IMU, as illustrated in the following code snippet:

```
from robot_imu import RobotImu
import time
import vpython as vp
imu = RobotImu()
```

3. We need vectors to hold the minimum and maximum values of the gyroscope, as illustrated in the following code snippet:

```
gyro_min = vp.vector(0, 0, 0)
gyro_max = vp.vector(0, 0, 0)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Now, for the loop, we'll do a bunch of readings over time, as follows:

```
for n in range(500):
    gyro = imu.read_gyroscope()
```

5. To calibrate, we measure for a while to get the minimum and maximum values for each axis. The Python `min` function returns the lower of the two values given to it, as follows:

```
    gyro_min.x = min(gyro_min.x, gyro.x)
    gyro_min.y = min(gyro_min.y, gyro.y)
    gyro_min.z = min(gyro_min.z, gyro.z)
```

6. We do the same for the maximum values, using the Python `max` function, as follows:

```
    gyro_max.x = max(gyro_max.x, gyro.x)
    gyro_max.y = max(gyro_max.y, gyro.y)
    gyro_max.z = max(gyro_max.z, gyro.z)
```

7. The middle of these is an estimate of how far we are from zero. We can calculate this by adding the vectors and dividing by 2, as follows:

```
    offset = (gyro_min + gyro_max) / 2
```

8. Sleep a little before the next loop, as follows:

```
    time.sleep(.01)
```

9. We print the values so we can use them, as follows:

```
print(f"Zero offset: {offset}.")
```

10. This code is ready to run. Upload and run this with Python 3, leaving the robot still on a flat, stable surface until the program exits.
11. You should see console output ending with something like this:

```
pi@myrobot:~ $ python3 calibrate_gyro.py
Zero offset: <-0.583969, 0.675573, -0.530534>.
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

What we've measured here is how much the gyroscope changes, on average, when stationary. This is calculated as an offset for each axis. By subtracting this from the measurements, we will mostly offset any continuous errors from the gyroscope. Let's put this somewhere we can use it, as follows:

1. Create a file called `imu_settings.py`.
2. We'll import the `vector` type, and then set our calibration readings. You probably only need to run this once, or if you change IMU device. Please use the readings you got from your robot. Run the following code:

```
from vpython import vector
gyro_offsets = vector(-0.583969, 0.675573, -0.530534)
```

3. Next, we upgrade our `RobotImu` class to handle these offsets—open `robot_imu.py`.
4. We will make our class accept offsets if we pass them, or use zero if we leave them. Make the highlighted changes to the `__init__` method of `RobotImu`, as follows:

```
    def __init__(self, gyro_offsets=None):
        self._imu = ICM20948()
        self.gyro_offsets = gyro_offsets or vector(0, 0,
0)
```

5. We need to modify the `read_gyroscope` method to account for these too, as follows:

```
    def read_gyroscope(self):
        _, _, _, gyro_x, gyro_y, gyro_z = self._imu.read_
accelerometer_gyro_data()
        return vector(x, y, z) - self.gyro_offsets
```

Now, to see if this works, let's use it to move a virtual robot.

Rotating the virtual robot with the gyroscope

We've mentioned how we will integrate the gyroscope measurements. Take a look at the following diagram to see how this will work for a single axis:

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

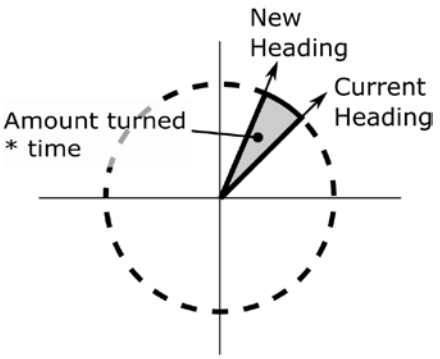


Figure 16.5 – Integrating a gyroscope axis

Figure 16.5 shows a dashed circle, indicating a turning circle of an axis. The crosshair through the circle shows its center. A thick arrow above and to the left of the circle indicates the current heading. A shaded area shows the change in rotation in degrees over some time, which we add to the current heading to get to the new heading estimate—another thick arrow.

We multiply the turning rate by time to get a movement; it is an estimate since we don't have intermediate values.

The concept of time-since-last-measurement is an important one, seen in the PID in *Chapter 14*, *Line-Following with a Camera in Python*. It's more commonly known as the delta time.

We can combine what we know about the gyroscope with the virtual robot and make it rotate on the screen. Let's use this to rotate our virtual robot, as follows:

1. Create a new file named `visual_gyroscope.py`. We have many imports here to bring the components together, as can be seen in the following code snippet:

```
import vpython as vp
from robot_imu import RobotImu
import time
import imu_settings
import virtual_robot
```

2. This time, when we set up the `RobotImu`, we will do so with the settings we made, as follows:

```
imu = RobotImu(gyro_offsets=imu_settings.gyro_offsets)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We are going to integrate three axes: `pitch`, `roll`, and `yaw`. Let's start them at zero, like this:

```
pitch = 0
roll = 0
yaw = 0
```

4. We should now set up the virtual robot and the view for it, as follows:

```
model = virtual_robot.make_robot()
virtual_robot.robot_view()
```

5. We are going to be tracking delta time, so we start by taking the latest time, like this:

```
latest = time.time()
```

6. We then start the main loop for this behavior. Since this is animating in VPython, we need to set the loop rate and tell it to update, as follows:

```
while True:
    vp.rate(1000)
```

7. We now calculate the delta time (`dt`), storing a new latest time, as follows:

```
    current = time.time()
    dt = current - latest
    latest = current
```

8. The code reads the gyroscope in the `gyro` vector, as follows:

```
    gyro = imu.read_gyroscope()
```

9. We integrate the current rate (in degrees per second) multiplied by `dt` (in seconds), as illustrated in the following code snippet:

```
    roll += gyro.x * dt
    pitch += gyro.y * dt
    yaw += gyro.z * dt
```

10. We reset the model's orientation to prepare it for rotation, like this:

```
    model.up = vp.vector(0, 1, 0)
    model.axis = vp.vector(1, 0, 0)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. We perform the rotations by each axis. We must convert these into radians, as follows:

```
    model.rotate(angle=vp.radians(roll), axis=vp.
vector(1, 0, 0))
    model.rotate(angle=vp.radians(pitch), axis=vp.
vector(0, 1, 0))
    model.rotate(angle=vp.radians(yaw), axis=vp.vector(0,
0, 1))
```

    The rotations each need an axis to rotate around. We specify rotating around the *x* axis with the vector (`1`, `0`, `0`).

12. This code is now ready to run; this will make the virtual robot change position when we rotate the robot in the real world! Upload the files and run with `vpython visual_gyroscope.py`.

13. As before, wait a minute or so, and point your browser to `myrobot.local:9020`. You should see the following display:

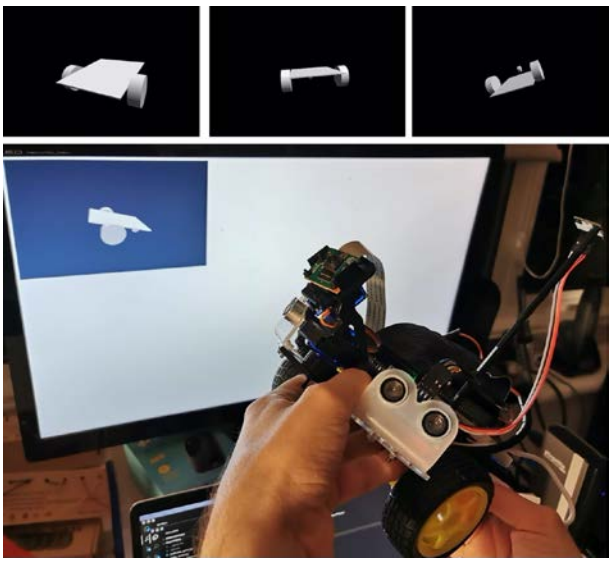


Figure 16.6 – The robot rotated

    Figure 16.6 shows the virtual robot rotated to an angle by having moved the real robot. Move your robot around a bit—try to approximate what you see here.

14. You'll notice that as you move the robot and return it, it won't line up correctly anymore—this is the accumulating errors or drift that gyroscope integration causes.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

From this experiment, while seeing some great movement you've also noticed that a gyroscope alone can't accurately track rotations. We are going to need to leverage the other sensors in the IMU device to improve this.

Let's check it is working before proceeding.

Troubleshooting

If it is not quite working, try some of these steps:

1. This code requires the use of the `vpython` command and a browser to see the results.
2. If the robot is still moving when held still, retry the calibration and offsets. The gyroscope's nature is that this won't be perfect—we'll fix that further on.
3. If the robot appears to spin uncontrollably or jump around, ensure you've remembered to convert to radians.
4. If the robot is rotating the wrong way (left/right instead of up/down), check the rotations' axis parameters.

Now that you have this working, let's move on to the accelerometer so that we can see forces acting on our robot!

Detecting pitch and roll with the accelerometer

In *Chapter 12*, *IMU Programming with Python*, we were getting a vector from the accelerometer, but we need to calculate angles to consider using it alongside the gyroscope and magnetometer. To use this to rotate things, we need to turn this vector into pitch-and-roll angles.

Getting pitch and roll from the accelerometer vector

The accelerometer describes what is going on in **Cartesian coordinates**. We need to convert these into a pair of pitch-and-roll angles perpendicular to each other. In *Chapter 12*, *IMU Programming with Python*, the *Coordinate and rotation systems* section shows roll as taking place around the *x* axis, and pitch as taking place around the *y* axis.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

A crude but effective way to consider this is as two planes. When rotating around the *x* axis, you can take a vector in the *yz* plane and find its angle. When turning around the *y* axis, then you consider the *xz* plane instead. Take a look at the next diagram:

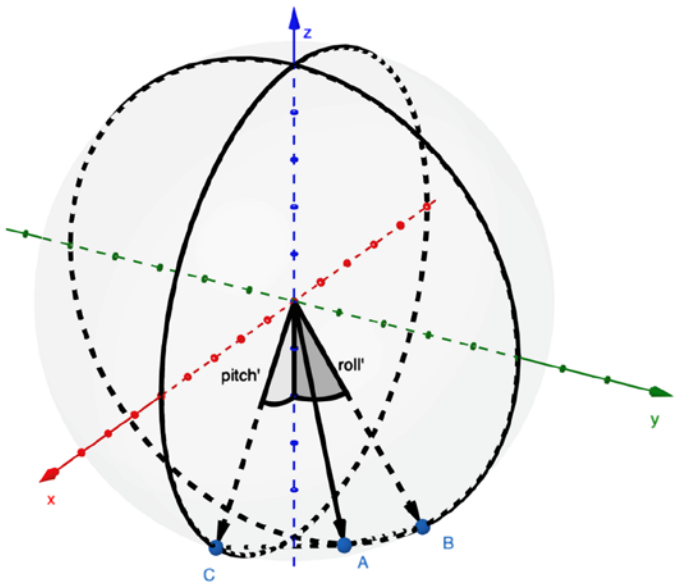


Figure 16.7 – The accelerometer vector and angles

In *Figure 16.7*, the background has *x*, *y*, and *z* axes and a sphere, with circles around the *xz* and *yz* planes.

The accelerometer vector is shown as vector **A**. By using only the *xz* components, we project this vector onto an *xz* circle at point **C**; so, the angle from the *z* axis to **C** is the pitch. We project **A** again onto a *yz* circle at point **B**; this angle from the *z* axis to **B** is the roll.

When we have two components (*x* and *z*, for example) on a plane, they can be used in the `atan2` function (present in most programming languages) to get an angle from them. A slight quirk here is that the orientation of the different sensor components means we must negate the pitch. The following diagram shows the process:

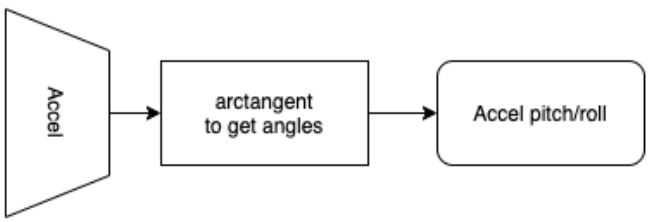


Figure 16.8 – Accelerometer data flow

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 16.8 shows the raw accelerometer data going into the arctangent to get the angles and outputting the accelerometer pitch/roll values.

Let's turn the accelerometer readings into pitch and roll, and then put them into a graph, as follows:

1. First, open up `robot_imu.py`.
2. Extend the imports to include the trigonometric functions, as follows:

```
from vpython import vector, degrees, atan2
```

3. After the `read_accelerometer` method, add the following code to perform the required math:

```
    def read_accelerometer_pitch_and_roll(self):
        accel = self.read_accelerometer()
        pitch = degrees(-atan2(accel.x, accel.z))
        roll = degrees(atan2(accel.y, accel.z))
        return pitch, roll
```

4. Let's show these angles on a graph, which will also reveal a major flaw with using the accelerometer on its own. Create a `plot_pitch_and_roll.py` file.
5. Start with imports, as follows:

```
import vpython as vp
import time
from robot_imu import RobotImu
imu = RobotImu()
```

6. We create the graphs, like this:

```
vp.graph(xmin=0, xmax=60, scroll=True)
graph_pitch = vp.gcurve(color=vp.color.red)
graph_roll = vp.gcurve(color=vp.color.green)
```

7. Now, we set up a start time so that we can make a time-based graph, as follows:

```
start = time.time()
while True:
    vp.rate(100)
    elapsed = time.time() - start
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. We can now get our new pitch-and-roll values, as follows:

```
pitch, roll = imu.read_accelerometer_pitch_and_roll()
```

9. And then, we can put both sets into graphs, like this:

```
graph_pitch.plot(elapsed, pitch)
graph_roll.plot(elapsed, roll)
```

10. Upload both the `robot_imu.py` and `plot_pitch_and_roll.py` files. Run this with `vpython plot_accel_pitch_and_roll.py`, and point your browser at port `9020` on the robot. This should result in the following:

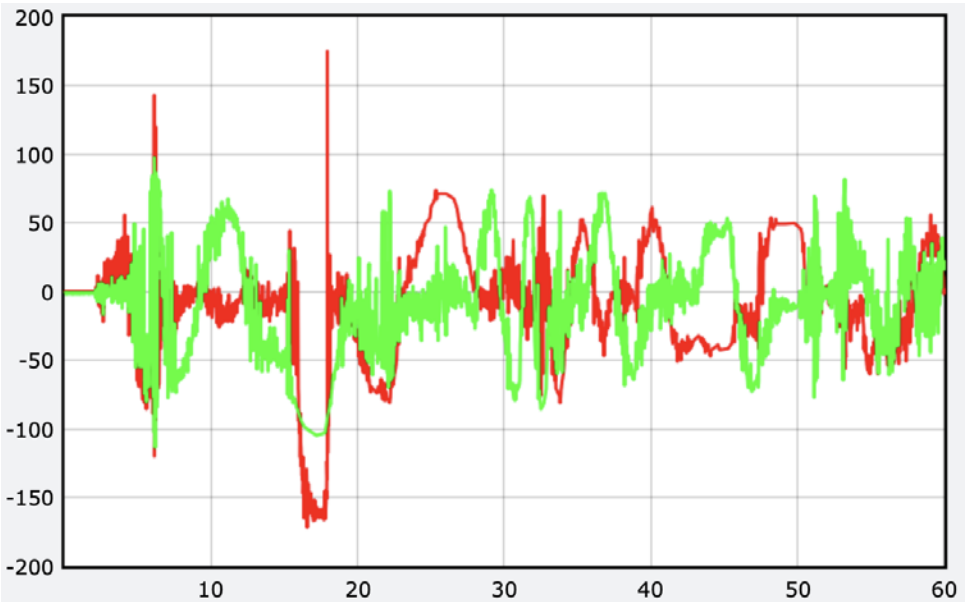


Figure 16.9 – Accelerometer pitch-and-roll graph

Figure 16.9 shows a screenshot of the graph. The red curve in the graph represents pitch, around the *y* axis, while the green curve represents roll, around the *x* axis. Although it shows swings between +90 and -90 degrees, what is also clear is that the graph has a lot of noise, so much so that movements of less than a second are blotted out by it.

We need to clean this up. A common way to do so is through a complementary filter, combining a new value with a previous value to filter out fast vibration noise. We will create such a filter, but it makes sampling slower.

Let's check that this is working.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

If it's not quite working, let's try a few fixes, as follows:

1. If it's very noisy, try a more severe turn, and try keeping your hands steady. This graph will be noisy due to the nature of the accelerometer alone.
2. If you see the graph break up or misbehave outside of the 0-90-degree range, ensure you are using the `atan2` function—this mathematically performs the trigonometric CAST rule.
3. Notice that the `read_accelerometer_pitch_and_roll` method has a negative sign in front of the `atan2` function.
4. If things misbehave at 180 degrees—this is a known and expected problem with this system—try to avoid hitting this yet.

Now, we have the pitch and roll, but it's quite rough—a suitable way to fix this is to combine sensors through a filter. We have another sensor that is giving us an integrated pitch-and-roll value: the gyroscope.

Smoothing the accelerometer

We can combine what we know about integrating the gyroscope with the accelerometer to make a smooth combination.

Since we will use the delta-time concept more, a class to help will save us some work later.

Delta time

We saw before how we tracked the elapsed time for graphing and the delta time between updates for integrating. Let's create the code to help, as follows:

1. Make a `delta_timer.py` file and start by importing `time`, as follows:

```
import time
```

2. We'll make a `DeltaTimer` class that will keep track of things, as follows:

```
class DeltaTimer:
    def __init__(self):
        self.last = self.start = time.time()
```

   The code initializes `last` and `start` variables with the current time.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The central part of this is an `update` method. Every loop calls this; let's start by getting the current time, as follows:

```
    def update(self):
        current = time.time()
```

4. The delta time will be the difference between the current time and last time, as illustrated in the following code snippet:

```
        dt = current - self.last
```

5. The elapsed time is the difference between the current time and the start time, as illustrated in the following code snippet:

```
        elapsed = current - self.start
```

6. We now need to update the last time for the delta time and return the parts, as follows:

```
        self.last = current
        return dt, elapsed
```

We can use this class whenever we need a delta time and an elapsed time for graphing. Let's start by using it to combine the accelerometer and gyroscope.

Fusing accelerometer and gyroscope data

By combining the sensors, we can let each of them complement the other's weaknesses. The accelerometer acts as an absolute measurement for pitch and roll to counteract the drift seen by the gyroscope. The gyroscope does not experience the same noise as the accelerometer but can make fast measurements. Let's see how to combine them in the following diagram:

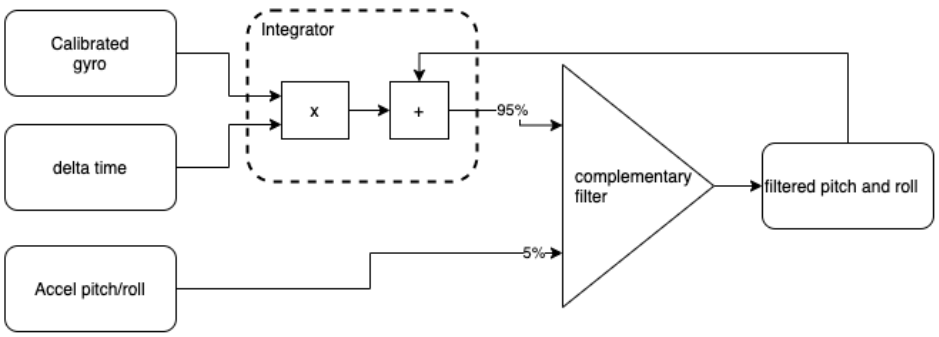


Figure 16.10 – Gyroscope and accelerometer fusion data flow

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 16.10 shows the data flow diagram, using a complementary filter to fuse gyroscope and accelerometer data. We'll take pitch as an example. First, the system feeds gyroscope data and delta time into an integrator. The integrator adds this to a previous position. We can then use 95% of this term to account for larger movement changes. The filter's other 5% is the accelerometer measurement. This 5% will drag the measurement to the average accelerometer reading while filtering out the chaotic noise element. The output is a filtered pitch or roll, fed back into the integrator for the next cycle.

Let's put this into code, starting with the filter, as follows:

1. Open up `robot_imu.py`.
2. Add the `ComplementaryFilter` class, as follows:

```
class ComplementaryFilter:
```

3. We can construct this filter with the left side's value, storing this and calculating the complement (one minus the left side) to make the right side, as follows:

```
    def __init__(self, filter_left=0.9):
        self.filter_left = filter_left
        self.filter_right = 1.0 - filter_left
```

4. This class has a `filter` method that takes the two sides and combines them using the filter values, as follows:

```
    def filter(self, left, right):
        return self.filter_left * left + \
               self.filter_right * right
```

That finishes the filter.

5. The next thing we'll want is code to combine the IMU sensors via filters – to fuse them. We'll add a class for this to `robot_imu.py`, as follows:

```
class ImuFusion:
```

6. In the constructor, we will store the `RobotImu` instance, create a filter, and seed the pitch-and-roll values, as follows:

```
    def __init__(self, imu, filter_value=0.95):
        self.imu = imu
        self.filter = ComplementaryFilter(filter_value).
filter
        self.pitch = 0
        self.roll = 0
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. The core part of this code is an `update` function. The function takes a `dt` (delta time) parameter. It will not return anything and just updates the pitch/roll members, as follows:

```
    def update(self, dt):
```

8. We start by taking the pitch-and-roll values from the accelerometer, as follows:

```
        accel_pitch, accel_roll = self.imu.read_
accelerometer_pitch_and_roll()
```

9. We also want the gyroscope values, so we run the following command:

```
        gyro = self.imu.read_gyroscope()
```

10. Now, we combine the gyroscope *y* reading and accelerometer pitch to get the pitch value, as follows:

```
        self.pitch = self.filter(self.pitch + gyro.y *
dt, accel_pitch)
```

    Notice here the multiply and addition operations from the preceding data flow.

11. We do the same for the roll, as follows:

```
        self.roll = self.filter(self.roll + gyro.x * dt,
accel_roll)
```

Now, we have prepared the `RobotImu` class with filters and fused the sensors. Let's give this code a test drive with a graph, as follows:

1. In the `plot_pitch_and_roll.py` file, we'll add the `DeltaTimer`, `ImuFusion`, and gyroscope calibration imports. Note in the following code snippet that `import time` has been removed:

```
import vpython as vp
from robot_imu import RobotImu, ImuFusion
from delta_timer import DeltaTimer
import imu_settings
```

2. Next, we set up the `RobotImu` with the gyroscope settings, and then create the `fusion` instance, as illustrated in the following code snippet:

```
imu = RobotImu(gyro_offsets=imu_settings.gyro_offsets)
fusion = ImuFusion(imu)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We need a `dt` (delta time) for the fusion calculations and an elapsed time for the graph. The `DeltaTimer` class provides these. We put this close before the loop starts, replacing the assignment of `start`, as follows:

```
timer = DeltaTimer()
```

4. Now, in the loop where we calculate `elapsed`, we use the delta timer, as follows:

```
while True:
    vp.rate(100)
    dt, elapsed = timer.update()
```

5. Now, replace the reading of the accelerometer with code to update the fusion with the delta time so that it makes its calculations, as follows:

```
fusion.update(dt)
```

6. We can now fetch pitch-and-roll values from the `fusion` object, as follows:

```
    graph_pitch.plot(elapsed, fusion.pitch)
    graph_roll.plot(elapsed, fusion.roll)
```

7. Upload `robot_imu.py`, `delta_timer.py` and `plot_pitch_and_roll.py` to the robot.
8. Run `vpython plot_pitch_and_roll.py`, again and point your browser at port `9020` on the robot.

Superficially, it should look similar to the accelerometer pitch-and-roll graph in *Figure 16.9*. However, as you move the robot around, you should notice that there is far less noise—the graph is smoother—and that when you place the robot down or hold it still, it levels off. It should quickly account for rapid turns. The system is smooth and accurate!

Troubleshooting

If you have issues, try these troubleshooting checks:

1. As always, if you see syntax errors or strange behavior, check the code carefully.
2. If things move strangely, ensure you are using `0.95` (and not `95`) for the filter value.
3. Ensure you've uploaded all the files.
4. This system will need a second or two after the graph starts to settle.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You've now seen how to get an accurate and smooth pitch and roll from these sensors. A robot on wheels may not encounter many reasons to use pitch and roll, but one of them will be to make a compass work. Let's dig further into the magnetometer.

Detecting a heading with the magnetometer

We saw in *Chapter 12*, *IMU Programming with Python*, how to plot a vector from the magnetometer, and how magnetic metal (such as bits of steel and iron) will interfere with it. Even the pin headers on the IMU board interfere. We can calibrate to compensate for this.

Getting *X*, *Y*, and *Z* components aren't that useful; we want a heading relative to a magnetic North. We can see how to use this for precise turns.

This section needs a space, with very few magnets present. Laptops, phones, speakers, and disk drives interfere with this sensor. Use a map compass to reveal magnetic fields in your space. I recommend making the standoff *stalk* on the robot as long as the cable allows, putting more standoffs in; the robot's motors have a strong magnetic field of their own.

Please avoid starting with the robot facing South—this will cause some odd results, which we will investigate and fix later. Starting with the robot roughly North is a good idea.

Calibrating the magnetometer

We are going to perform a calibration known as the **hard iron offset calculation**. Hard iron refers to any magnetic things near the magnetometer that move with it. We move the robot around to sample the field strength in each axis. We will use the middle of all readings for an axis to compensate, and add this to the IMU settings; this will seem similar to the gyroscope calibration but requires you to move the robot around.

Let's write the code, as follows:

1. Create a file named `magnetometer_calibration.py`, starting with imports and the `RobotImu` setup, as follows:

```
import vpython as vp
from robot_imu import RobotImu
imu = RobotImu()
```

2. We will find minimum and maximum vectors, as we did for the gyroscope, as illustrated in the following code snippet:

```
mag_min = vp.vector(0, 0, 0)
mag_max = vp.vector(0, 0, 0)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We are going to show the system as a set of three scatter charts with colored-dot clusters. Each of the three clusters is a plot combining two axes: *xy*, *yz*, and *xz*. Our goal is to make the sets line up by calibrating the device, as follows:

```
scatter_xy = vp.gdots(color=vp.color.red)
scatter_yz = vp.gdots(color=vp.color.green)
scatter_zx = vp.gdots(color=vp.color.blue)
```

4. We start the main loop and read the magnetometer, as follows:

```
while True:
    vp.rate(100)
    mag = imu.read_magnetometer()
```

5. Now, we update the minimums, in the same way we did for the gyroscope, as follows:

```
    mag_min.x = min(mag_min.x, mag.x)
    mag_min.y = min(mag_min.y, mag.y)
    mag_min.z = min(mag_min.z, mag.z)
```

6. And then, we update the maximums, as follows:

```
    mag_max.x = max(mag_max.x, mag.x)
    mag_max.y = max(mag_max.y, mag.y)
    mag_max.z = max(mag_max.z, mag.z)
```

7. We then calculate the offset in the same way as we did for the gyroscope, as follows:

```
    offset = (mag_max + mag_min) / 2
```

8. This `print` statement shows the current values and offsets:

```
    print(f"Magnetometer: {mag}. Offsets: {offset}.")
```

9. Now, we create the plots. They will guide you in getting enough calibration data and show where the axes do not line up. The code is shown in the following snippet:

```
    scatter_xy.plot(mag.x, mag.y)
    scatter_yz.plot(mag.y, mag.z)
    scatter_zx.plot(mag.z, mag.x)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

10. Upload this and run it with VPython. You should see the following screen:

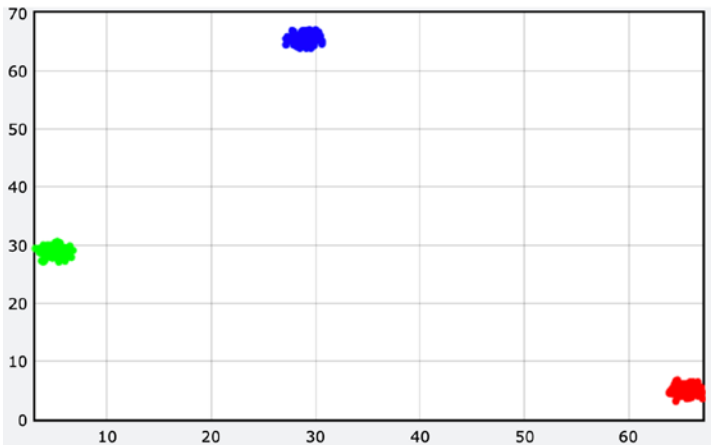


Figure 16.11 – Initial magnetometer calibration screen

Figure 16.11 shows the clusters as three colored blobs—the bottom right is red, (for *xy*), the top is blue (for *yz*), and on the right is green (for *zx*). These clusters will start in a different position for you, depending on the orientation of your robot.

11. You need to move the robot around, rotating it slowly around the whole *y* axis (around the wheels). The green graph should be more like an ellipse, as illustrated in the following screenshot:

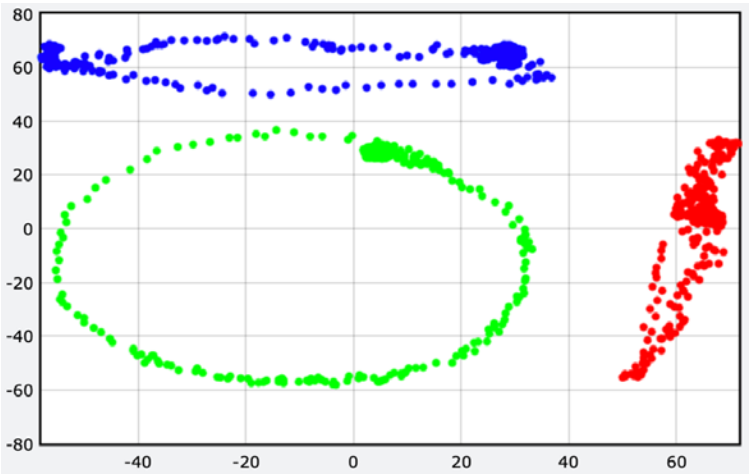


Figure 16.12 – The magnetometer partially calibrated

Figure 16.12 shows the ellipse for the green values, and more data for the red and blue scatter plots. The slower you are, the better the data is.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. Rotate the robot around the *x* axis (the length of the robot), and then around the *z* axis (around its height). The more angles you move it through, the better. Fill in the gaps by making 3D figures of 8. Eventually, it should look like the graph in the following screenshot:

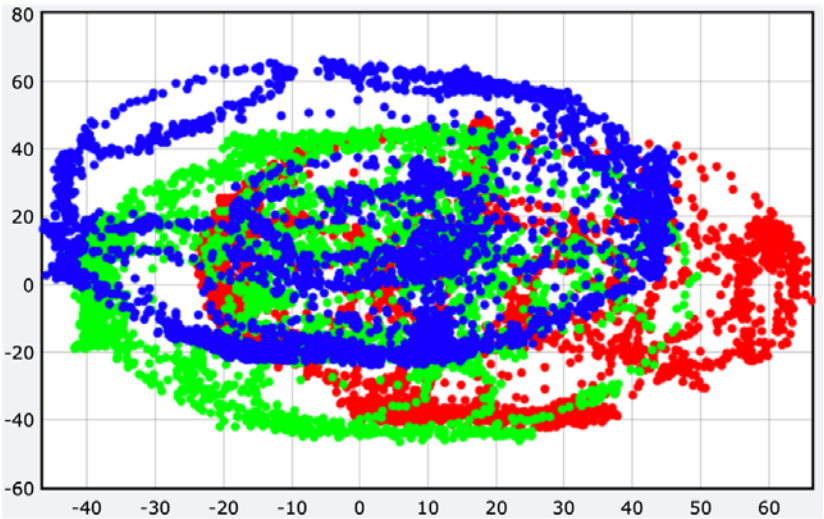


Figure 16.13 – Magnetometer calibration: a good combination

Figure 16.13 shows how a good collection of data should look, with circles of red, green, and blue. Note that there are outliers due to waving the robot too close to other magnets—beware of these!

13. You can close the browser now, having collected a load of calibration data.
14. The console will show the calibration offsets, as follows:

```
Magnetometer: <30.6, -36.9, 10.35>. Offsets: <21.15,
3.15, 0.225>.
Magnetometer: <31.5, -36.75, 11.25>. Offsets: <21.15,
3.15, 0.225>.
```

At the start, those offsets change a lot; however, as you collect more data, they will settle and stabilize, even when the magnetometer readings are changing.

We now have some calibration numbers; my example gave `21.15, 3.15, 0.225`. Let's make sure that everyone has some values.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

This calibration may not have worked—let's see why, as follows:

1. If the numbers don't appear to be settling, continue moving the robot. You must try to do full 360-degree movements with it to get a full range.
2. If strange dots appear outside of the circle, move somewhere else and restart the calibration—this is likely to be a magnetic field where you are testing, and will throw your results off.
3. There is a possibility your browser will get slow or run out of memory trying to do this—while I say move slowly, you cannot put this aside while running as it will continue adding dots.
4. If you don't get circles at all—lines or small patches—double-check that you have the right plot combinations of *xy*, *yz*, and *zx*.

You should now be getting calibration offsets. Let's use these values to line up the scatter plots.

Testing the calibration values

To see if these are effective we'll put them back into the code, and the same operation should show the dot clusters lining up. It starts by allowing us to set offsets in the `RobotImu` interface. Proceed as follows:

1. Open up the `robot_imu.py` file.
2. In the `__init__` method, we need to store the offsets. I've highlighted the new code, as follows:

```
    def __init__(self, gyro_offsets=None,
                 mag_offsets=None):
        self._imu = ICM20948()
        self.gyro_offsets = gyro_offsets or vector(0, 0, 0)
        self. mag_offsets = mag_offsets or vector(0, 0, 0)
```

3. The `read_magnetometer` method needs to subtract the magnetometer offsets, like this:

```
    def read_magnetometer(self):
        mag_x, mag_y, mag_z = self._imu.read_
magnetometer_data()
        return vector(mag_x, -mag_z, -mag_y) - self. mag_
offsets
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Our scripts can now include an offset for the magnetometer. We'll put these in the same settings file we used for the gyroscope calibrations. Proceed as follows:

1. Open `imu_settings.py`.
2. Add the magnetometer calibration readings from your robot, as follows:

```
from vpython import vector
gyro_offsets = vector(0.557252, -0.354962, -0.522901)
mag_offsets = vector(21.15, 3.15, 0.225)
```

3. Now, we can use them in our scatter plot. Open up `magnetometer_calibration.py` and add our settings to the imports, as follows:

```
import vpython as vp
from robot_imu import RobotImu
from imu_settings import mag_offsets
```

4. When we have created our `RobotImu` we can apply the offset, like this:

```
imu = RobotImu(mag_offsets=mag_offsets)
```

5. Send the files to the robot, and rerun `magnetometer_calibration.py`. You'll need to make rotations and figures of 8 again to get many sample points at different orientations. When you have collected the data you should have overlapping circles, as depicted in the following screenshot:

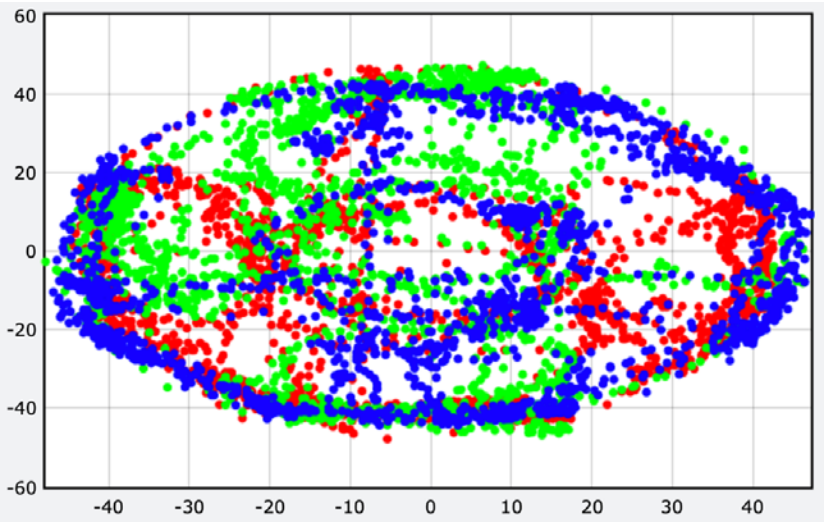


Figure 16.14 – Calibrated magnetometer

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 16.14 shows the red, blue, and green circles superimposed. Congratulations—you have calibrated your magnetometer!

With your calibrated magnetometer, we can try further experiments with more useful values. But first, let's troubleshoot any problems.

What to do if the circles aren't together

You may have reached this point, and the circles are not converging. Try these troubleshooting steps if this is the case:

1. You will need to rerun the calibration code. Before you do so, comment out the line that applies (sets) the offsets on the `RobotImu` class. Running the calibration code when you have offsets active will cause it to offset incorrectly.
2. Check your calibration and IMU code carefully for errors.
3. Ensure there are no strong magnets or big chunks of metal near the robot—speakers or hard disks, for example. Try to do this about a meter away from such things. Even your laptop or mobile phone can interfere.
4. Ensure you go through each axis slowly and try the figure of 8. Keep going until you can see three ellipses.
5. You can use the console output, rotating the robot and moving around in every orientation, and then seeing if the offset values output here settle.
6. When the outputs settle, try applying the offset again, and run the calibration to see if the circles converge.

After going through these, you should have the calibration values you need to continue. Let's put this back into the vector output we had and determine a heading.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting a rough heading from the magnetometer

Now that we've got calibration settings, we can start using magnetometer readings to estimate where North is, like a compass. The words *heading* and *yaw* mean the same thing —which way we face relative to a reference point—in this case, magnetic North. Let's see how we can do this. Have a look at the following screenshot:

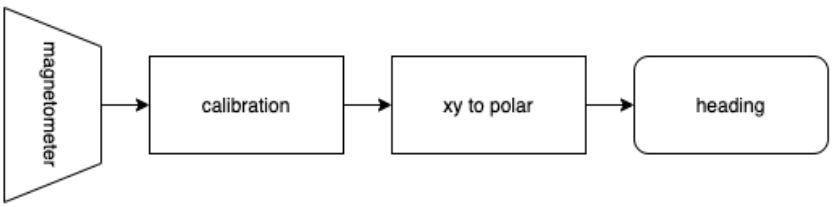


Figure 16.15 – Getting an approximate heading from the magnetometer

Figure 16.15 shows a method we will build. It takes the magnetometer with calibration data applied and uses `atan2`, as we did with the gyroscope to approximate the heading. We can also add a rough compass with it too.

Let's make this, as follows:

1. Create a `plot_mag_heading.py` file. Start with the imports, as follows:

```
import vpython as vp
from robot_imu import RobotImu
from delta_timer import DeltaTimer
import imu_settings
```

2. We can initialize the `RobotImu` with the settings, like this:

```
imu = RobotImu(mag_offsets=imu_settings.mag_offsets)
```

3. To make a compass display, we need a dial (cylinder) and needle (arrow) in red, as follows:

```
vp.cylinder(radius=1, axis=vp.vector(0, 0, 1),
            pos=vp.vector(0, 0, -1))
needle = vp.arrow(axis=vp.vector(1, 0, 0),
                  color=vp.color.red)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Next, let's make a graph to show the heading, and a delta timer for elapsed time, as follows:

```
vp.graph(xmin=0, xmax=60, scroll=True)
graph_yaw = vp.gcurve(color=vp.color.blue)
timer = DeltaTimer()
```

5. We start the main loop with a rate and fetch the elapsed time, as follows:

```
while True:
    vp.rate(100)
    dt, elapsed = timer.update()
```

6. Now, we read the magnetometer by running the following command:

```
    mag = imu.read_magnetometer()
```

7. We can take the *xy* plane and find the `atan2` function of these values of this to get a heading, as follows:

```
    yaw = -vp.atan2(mag.y, mag.x)
```

8. Then, we plot this on the graph in degrees, like this:

```
    graph_yaw.plot(elapsed, vp.degrees(yaw))
```

9. We can also set the needle axis to our direction, using `sin`/`cos` to convert it back into a unit direction, as follows:

```
    needle.axis = vp.vector(vp.sin(yaw), vp.cos(yaw), 0)
```

10. Save, upload, and run this in VPython. Send your browser to port `9020` on the robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. If you rotate the robot around, you will see a display like this:

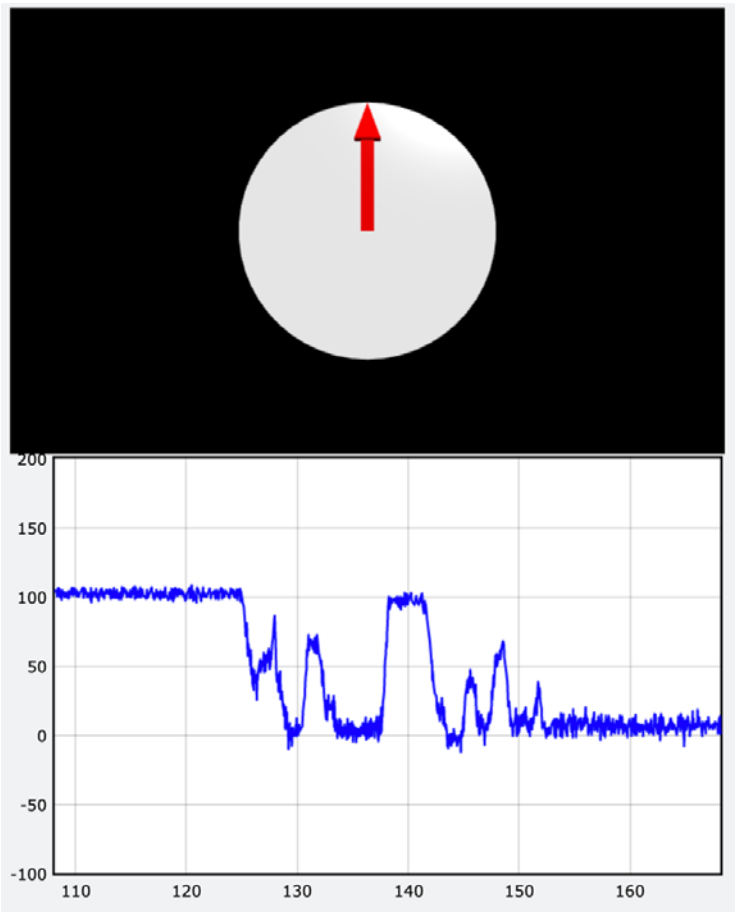


Figure 16.16 – Magnetometer heading estimate

Figure 16.16 shows a compass, with the top being what the robot perceives as North, and a red arrow. Below this is a blue graph, ranging between + and –180 degrees. As you move the robot, you will see this move, with 0 degrees being North. You will need the robot to be on a flat surface, though.

Note that the compass is reading where North is relative to the robot—not where the robot is relative to North!

This output is starting to appear reasonable. It can point North and make some compass measurements, and we have a heading.

It is a little chaotic, and you can make it incorrect by any pitch or roll. Again, by fusing this data with data from the other sensors, we can improve this.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Combining sensors for orientation

We've seen how we combined the accelerometer and gyroscope to get smooth readings for pitch and roll. We can combine the sensors again to correctly orient and smooth the magnetometer readings too. This system allows us to approximate the absolute orientation of the robot.

Take a look at the following data flow to see what we are doing—it builds on the previous stages:

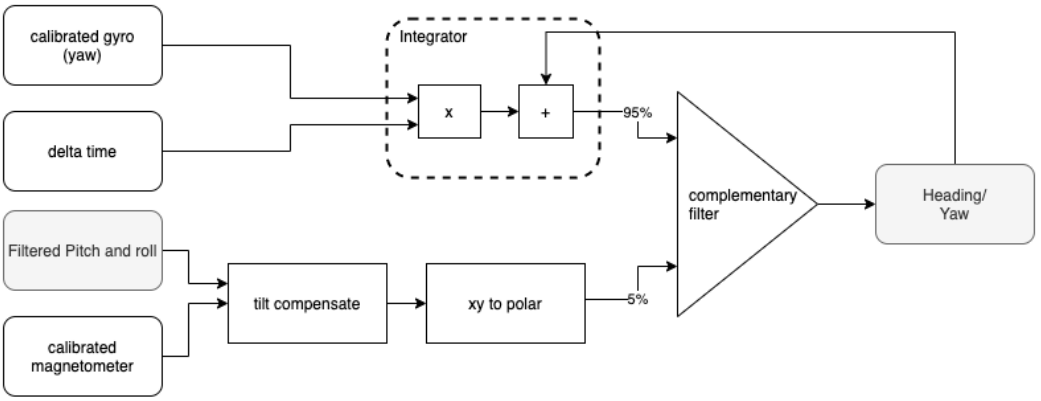


Figure 16.17 – Fusing all three sensors

Figure 16.17 starts on the left with data from our previous stages. We have the filtered pitch and roll in gray because it's also an output. There's the calibrated gyroscope yaw, delta time, and also the calibrated magnetometer as inputs. The filtered pitch and roll go through the tilt-compensate box, where we rotate the magnetometer vector. The magnetometer data then goes through an *xy*-to-polar box, using the `atan2` function to get a heading.

Above this, the calibrated gyroscope yaw and delta time go into an integrator, which adds to a previous yaw reading. The integrator and magnetometer heading output go into a complementary filter, with the integrator output being dominant. This filter output is then a heading/yaw output, which will be stable and quick to respond, and will return to the absolute heading. We now have three angles—pitch, roll, and yaw!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's modify the code to do this, as follows;

1. Open up `robot_imu.py` and head to the `ImuFusion` class.
2. We will need to convert back to radians, so we need to add this to the imports from VPython, as follows:

```
from icm20948 import ICM20948
from vpython import vector, degrees, atan2, radians
```

3. In the `__init__` method, we should add a variable to store the yaw in, as follows:

```
    def __init__(self, imu, filter_value=0.95):
        self.imu = imu
        self.filter = ComplementaryFilter(filter_value).
filter
        self.pitch = 0
        self.roll = 0
        self.yaw = 0
```

We are going to use the same filter for now.

4. In the `update` method, after calculating the pitch and roll, add the following line to get the magnetometer reading:

```
        mag = self.imu.read_magnetometer()
```

5. The `mag` variable is a vector. We rotate this using pitch and tilt to level the *xy* components, as follows:

```
        mag = mag.rotate(radians(self.pitch), vector(0,
1, 0))
        mag = mag.rotate(radians(self.roll), vector(1, 0,
0))
```

6. We can now calculate the magnetometer yaw from this, as follows:

```
        mag_yaw = -degrees(atan2(mag.y, mag.x))
```

7. To stabilize this, we can now use the complementary filter with the gyroscope, as follows:

```
        self.yaw = self.filter(self.yaw + gyro.z * dt,
mag_yaw)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `self.yaw` value will now have the compensated yaw (or heading) value, allowing this IMU to act as a compass. To make use of it, let's visualize it in three ways—as a graph, a compass, and the movement of the robot. Proceed as follows:

1. Put this in a new file called `visual_fusion.py`. The code will be very familiar. Only the magnetometer offsets and yaw values are new. The imports are shown in the following code snippet:

```
import vpython as vp
from robot_imu import RobotImu, ImuFusion
from delta_timer import DeltaTimer
import imu_settings
import virtual_robot
```

2. Prepare the `RobotImu` with magnetometer offsets, and initialize `fusion`, as follows:

```
imu = RobotImu(gyro_offsets=imu_settings.gyro_offsets,
               mag_offsets=imu_settings.mag_offsets)
fusion = ImuFusion(imu)
```

3. We are going to use a VPython canvas for the virtual robot, and a separate one for the compass. Each canvas lets us contain a 3D scene. Let's make the current canvas a robot view and put it on the left. The robot model will be associated with this. The code is shown in the following snippet:

```
robot_view = vp.canvas(align="left")
model = virtual_robot.make_robot()
virtual_robot.robot_view()
```

4. To accompany the robot view, we'll create a `compass` canvas, using the same cylinder and arrow as previously. Note that the most recent canvas is associated with the shapes created after it. The code is shown in the following snippet:

```
compass = vp.canvas(width=400, height=400)
vp.cylinder(radius=1, axis=vp.vector(0, 0, 1),
            pos=vp.vector(0, 0, -1))
needle = vp.arrow(axis=vp.vector(1, 0, 0),
                  color=vp.color.red)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Set up graphs for pitch, roll, and yaw, as follows:

```
vp.graph(xmin=0, xmax=60, scroll=True)

graph_roll = vp.gcurve(color=vp.color.red)
graph_pitch = vp.gcurve(color=vp.color.green)
graph_yaw = vp.gcurve(color=vp.color.blue)
```

6. Create a delta timer, start the loop, and fetch the time update, as follows:

```
timer = DeltaTimer()
while True:
    vp.rate(100)
    dt, elapsed = timer.update()
```

7. We now update `fusion` with the time (it will read the IMU and perform calculations), as follows:

```
    fusion.update(dt)
```

8. Now, we need to reset the virtual robot model before we rotate it, as follows:

```
    model.up = vp.vector(0, 1, 0)
    model.axis = vp.vector(1, 0, 0)
```

9. And then, we need to perform three rotations—roll, pitch, and yaw, as follows:

```
    model.rotate(angle=vp.radians(fusion.roll), axis=vp.
vector(1, 0, 0))
    model.rotate(angle=vp.radians(fusion.pitch), axis=vp.
vector(0, 1, 0))
    model.rotate(angle=vp.radians(fusion.yaw), axis=vp.
vector(0, 0, 1))
```

10. We position the compass needle—note that our yaw is in degrees, so we convert it, as follows:

```
    needle.axis = vp.vector(
            vp.sin(vp.radians(fusion.yaw)),
            vp.cos(vp.radians(fusion.yaw)),
            0)
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. Then, we plot the three-graph axes, as follows:

```
graph_roll.plot(elapsed, fusion.roll)
graph_pitch.plot(elapsed, fusion.pitch)
graph_yaw.plot(elapsed, fusion.yaw)
```

12. Upload `robot_imu.py` and `visual_fusion.py` to the robot. Start with `vpython visual_fusion.py` and point your browser at port `9020` on the robot.

You should see the visual robot, compass, and a graph for all three axes displayed, and each should be both relatively stable and responsive, as depicted in the following screenshot:

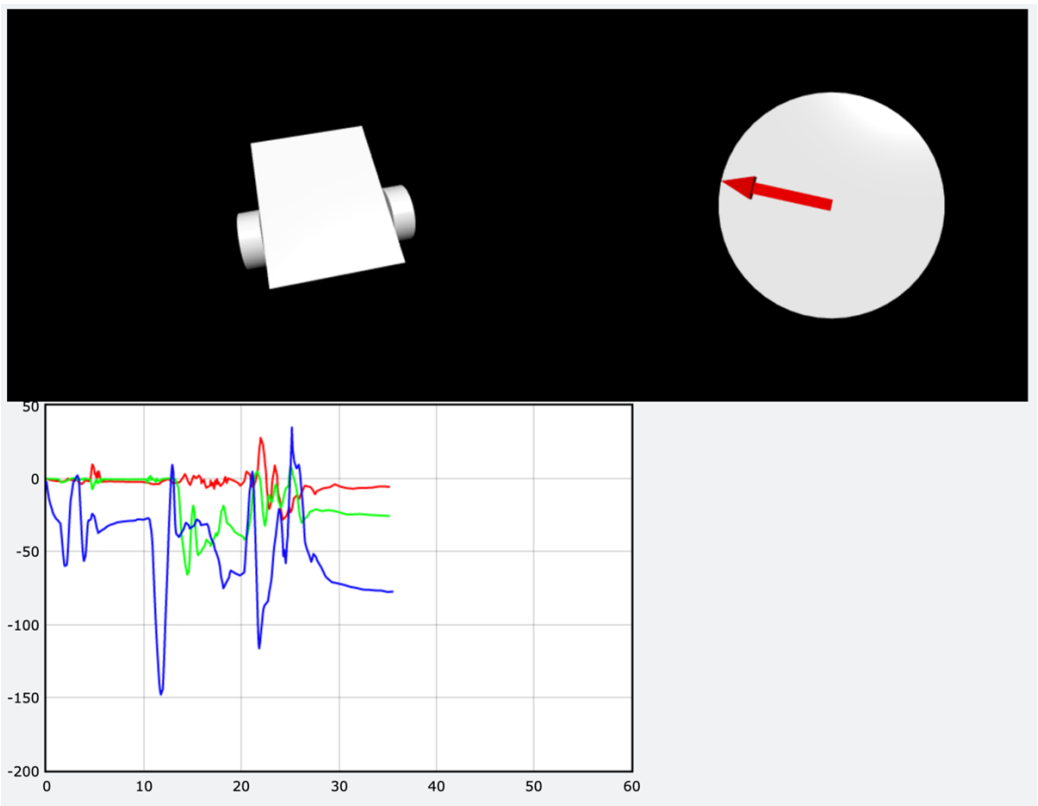


Figure 16.18 – Pitch, roll, and yaw graph

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The graph in *Figure 16.18* is a screenshot of the display. In the top left is the virtual robot—you can change its view by right-clicking. The top left shows the compass. Below that is a scrolling pitch, yaw, and roll graph. The roll is in red, pitch is in green, and yaw is in blue. The graphs will initially settle and then match your robot movements. When moving in one axis there is a small effect on the others, but they can move independently.

At +/-180 degrees, the graph will misbehave though. Let's see how to fix that.

Fixing the 180-degree problem

The fundamental thing to realize is that angles on a circle are cyclical; 200 degrees and -160 degrees are equivalent, and -180 degrees and 180 degrees are also equal. We've not made the filter or code aware of this, so when we reach the 180-degree point and the `atan2` function is flipping between -179.988 and 179.991 (or some similar very close mark), the graph becomes chaotic, treating the difference of less than 1 degree as 360 degrees, and then trying to filter between them.

This problem needs some changes to fix it. First, we can state that we intend angles to be numerically below 180 and above -180 and constrain them this way. Since we intend to use the complementary filter with angles, we can specialize it, as follows:

1. At the top of `robot_imu.py`, inside the `ComplementaryFilter` class, let's add a method to format the angle, like this:

```
    @staticmethod
    def format_angle(angle):
```

2. If the angle is below -180, we want to wrap it around by adding `360`, as follows:

```
        if angle < -180:
            angle += 360
```

3. If the angle is above 180, we wrap it around by subtracting `360`, as follows:

```
        if angle > 180:
            angle -= 360
        return angle
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. We will replace the inside of the `filter` function with something to constrain these angles more intelligently. When we filter, we start by formatting the incoming angles, as follows:

```
    def filter(self, left, right):
        left = self.format_angle(left)
        right = self.format_angle(right)
```

5. We also want to put the filtered angles in the same range. If there is a difference of more than 350 degrees, we can assume that something has wrapped around; so, we add 360 to the lowest one to filter them together, as follows:

```
        if left - right > 350:
            right += 360
        elif right - left > 350:
            left += 360
        filtered = self.filter_left * left + \
               self.filter_right * right
```

6. This operation could leave an answer outside of the range. So, we format it back, like this:

```
        return format_angle(filtered)
```

7. This filter is in use already, so we can rerun `visual_fusion.py` and try turning back through 180 degrees again. When you point your browser at the port, after settling, the robot there should be rotating with yours—and settling, not drifting!

Note that this system still doesn't deal well with facing South when it starts. We've solved at least one problem with the system and smoothed out its flaws.

This behavior is exciting: you can now get a robot on screen to mirror how you rotate it. However, while moving on the screen is fun, we'd like to see this used in the real world. Let's engage some motors!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Driving a robot from IMU data

In previous chapters, we saw how to use the PID algorithm, and in this chapter, how to detect a pitch, roll, and yaw from a magnetometer. Our robot can't move its pitch or roll, but it can change its heading.

In this demonstration, we'll get the robot to stay on course—to try to track North regardless of where we turn it. Let's see how. Have a look at the following diagram:

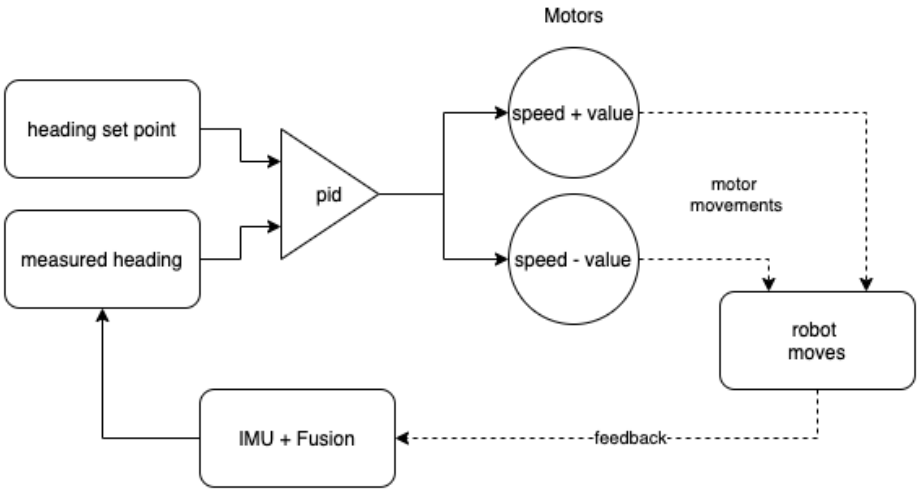


Figure 16.19 – Drive to heading behavior

Figure 16.19 shows the flow of data. The left of the diagram starts with a measured heading, and a heading setpoint going into a PID—the error value will be the difference between the two. The measured heading has come from the **IMU + Fusion** algorithm. We use the PID output to drive the motors so that they move at a fixed speed plus or minus the value, so the robot will turn to reduce the error. The robot moving will feed back into the **IMU + Fusion** algorithm, looping through the PID.

Let's take the preceding flow and use it to build the code, as follows:

1. Start a `drive_north_behavior.py` file with the following imports:

```
from robot_imu import RobotImu, ImuFusion
from delta_timer import DeltaTimer
from pid_controller import PIController
from robot import Robot
import imu_settings
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. We now initialize the `RobotImu`, `fusion`, and the `DeltaTimer`, as follows:

```
imu = RobotImu(mag_offsets=imu_settings.mag_offsets,
               gyro_offsets=imu_settings.gyro_offsets)
fusion = ImuFusion(imu)
timer = DeltaTimer()
```

3. We can then set up a PID (or PI) controller and the robot, as follows:

```
pid = PIController(0.7, 0.01)
robot = Robot()
```

4. And then, a couple of constants—the robot's base speed, and the heading setpoint in degrees from North, as illustrated in the following code snippet:

```
base_speed = 70
heading_set_point = 0
```

5. The main loop here updates the timer and IMU fusion. Note in the following code snippet that there's not a visual rate here:

```
while True:
    dt, elapsed = timer.update()
    fusion.update(dt)
```

6. We now calculate the error, and feed the PID with that and the delta time, as follows:

```
    heading_error = fusion.yaw - heading_set_point
    steer_value = pid.get_value(heading_error, delta_
time=dt)
```

7. We print the values to debug, and set our motor speeds, as follows:

```
    print(f"Error: {heading_error}, Value:{steer_
value:2f}, t: {elapsed}")
    robot.set_left(base_speed + steer_value)
    robot.set_right(base_speed - steer_value)
```

Upload this to the robot, turn on the motors, and run with regular Python 3. The robot will try to drive North. If you turn it off course it will correct back to North, and the more you turn it, the faster the motors will try to turn back. Playing with this behavior is quite fun!

Press *Ctrl* + *C* to stop this when you are done, and play with different heading set points.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this section, you've reinforced building data flow diagrams and writing code from them. You've further demonstrated that by converting sensor data to a number like this, you can build a PID-based behavior with it. You've then taken the heading that we've calculated and used it with the PID to create compass-based movement from your robot.

Summary

In this chapter, you've seen how to combine the IMU sensors to approximate an absolute orientation in space. You've seen how to render these in graphs and how to display them onscreen with a virtual robot. You've then seen how to hook this sensor system up to a PID controller and motor to get the robot to drive.

You've learned a little of the math needed to convert between vector components and angles, in 3D, along with how to use complementary filters to compensate for noise in one system and drift in another. You've started to see multiple sensors fused together to make inferences about the world. Your block diagram and data flow skills have been exercised, and you have had more practice with the PID algorithm.

In the next chapter, we will look at how you can control your robot and choose behaviors from a menu with a smartphone.

Exercises

Here are some ideas to further your understanding, and give you some ideas for more interesting things to do with the concepts from this chapter:

- A reader can use more colors and complicated shapes to make a better robot model. It's not the purpose of this chapter, but it is a fun and rewarding way to get more familiar with VPython.
- Our magnetometer settings were hardcoded, going into a Python file. It is good practice to load settings from a data file. A good starting point can be found at `http://zetcode.com/python/yaml/`.
- Could the visual robot be used to display or debug the other sensors and integrations?
- Could you combine the absolute positioning here with the encoders to make a square with very accurate turns?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

For more information on the topics covered in this chapter, refer to the following:

- The **World Wide Web Consortium** (**W3C**) has a guide on magnetometer devices in browsers, which makes for interesting reading on techniques, but also on how code on a smartphone might be able to perform these same algorithms to get the phone orientation: `https://www.w3.org/TR/magnetometer`.
- I've mentioned the `atan2` function a lot; this page has further information on it: `https://en.wikipedia.org/wiki/Atan2`.
- I recommend Paul McWhorter's Arduino experiments with an IMU, and his introduction to VPython—his guide was an instrumental part in the research for this book: `https://toptechboy.com/arduino-based-9-axis-inertial-measurement-unit-imu-based-on-bno055-sensor/`.
- This paper takes things a bit further and introduces a **Global Positioning System** (**GPS**) for further sensor fusion: `https://www.researchgate.net/publication/51873462_Data_Fusion_Algorithms_for_Multiple_Inertial_Measurement_Units`.
- If you wish to dig deeper into sensor fusion, and algorithms to combine them while filtering errors, Kalman filters are the way to go. This article is a starting point: `https://towardsdatascience.com/sensor-fusion-part-1-kalman-filter-basics-4692a653a74c`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

17
Controlling the Robot with a Phone and Python

The robot we've been programming has many behaviors, but when you run some of them, they result in the robot stopping on the other side of the room. You could try to write code to return it back to you, but this may be complicated. We've also got a neat camera with some visual feedback available on what the robot is doing. Wouldn't it be neat to take control and drive the robot sometimes?

We've been launching commands to drive our robot from a **Secure Shell** (**SSH**) terminal, but the robot will be more exciting and more comfortable to demonstrate if you could start the commands via a menu. We can build upon the web **application programming interface** (**API**) code you made in *Chapter 15*, *Voice Communication with a Robot Using Mycroft*.

In this chapter, we will see how to create a menu system to choose behaviors designed for a phone. We will then use the touch surface to build a control system, with the camera in view. You will learn about phone-ready web apps and get control of the robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will cover the following topics in this chapter:

- When speech control won't work—why we need to drive
- Menu modes—choosing your robot's behavior
- Choosing a controller—how are going to drive the robot, and why
- Preparing the Raspberry Pi for remote driving—get the basic driving system going
- Making the robot fully phone-operable
- Making the menu start when the Pi starts

Technical requirements

For this chapter, you will need the following items:

- Your Raspberry Pi robot with the camera set up and the code from previous chapters
- A touchscreen device such as a phone with Wi-Fi
- A wireless network

The GitHub code for this chapter is at `https://github.com/PacktPublishing/Learn-Robotics-Programming-Second-Edition/tree/master/chapter17`.

Use the `0_starting_point` folder to find the complete code from the previous chapters and the `full_system` folder on GitHub for this chapter's full code.

Check out the following video to see the code in action: `https://bit.ly/2Kb7rp8`

When speech control won't work – why we need to drive

In *Chapter 15*, *Voice Communication with a Robot Using Mycroft*, we built a Mycroft system to launch behaviors. If you have tried to build intents to make the robot stop in time or drive left or right, you will have probably noticed that it takes some time to respond even with the clearest speaking.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Speech control also only really works in a quiet room. If your robot is outside (you would like to drive it somewhere), this is not useful.

Mycroft is also utterly dependent on having access to the internet. It is one thing to have a small shared network for a robot and a controller; it's another to always require internet access, which can become tricky when not at your home, school, or lab.

Using an SSH session to log in to a robot, then typing commands to start and stop behaviors works well during testing stages, but it can be slow and cumbersome. In demonstration conditions, mistyping a command or just restarting the SSH session is time-consuming.

A phone-targeted browser app can be responsive, giving you tight control over the robot's movements. With a local network, it won't need external internet access. You can use this to drive a robot back to you after a behavior has run and you've stopped it, and it can be used to halt errant behavior. And with a bit of thought, it can be used to deliver useful—or plain interesting—feedback on what your robot is doing.

Menu modes – choosing your robot's behavior

Our book has introduced quite a collection of robot behaviors and invited you to create more. We've talked about how SSH can be cumbersome to start robot programs—even just remembering the options you have or pressing the *Ctrl + C* combination to stop can be frustrating.

In this section, we are going to create a menu system to select them. A convenient and phone-friendly way to do this is to serve it to the phone's browser, so we take that approach with our robot. We will also use a desktop browser to test this code.

We can extend the system we built in the *Starting a behavior remotely* section of *Chapter 15*, *Voice Communication with a Robot Using Mycroft*, adding a **user interface** (**UI**). We'll make this UI as templates, with some placeholders replaced by code.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's take a look in the following diagram at how this system will work:

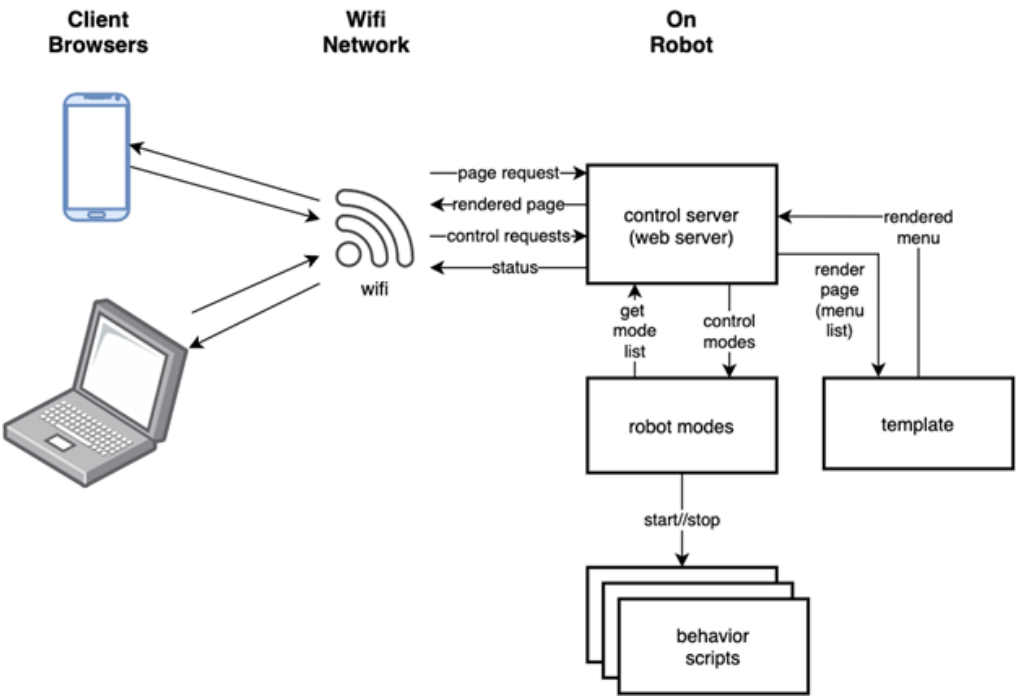


Figure 17.1 – How the control server and menu system will work

Figure 17.1 shows an overview of the system. Here's how it works:

1. The **client browser** (a phone or computer) makes a page request via **Wi-Fi** to the **web server** on the robot for a page to display.
2. The **web server** uses **robot modes** to get the mode list: a list of scripts it can start.
3. The **web server** sends this mode list to a **template** to render it into the menu page and send that rendered menu page to the user.
4. In the browser, when you touch or click the menu item links in the page, they make control requests to the **web server**.
5. The **web server** acts on control requests by making control mode calls such as `run` and `stop` to the **robot modes** system.
6. The **robot modes** system starts/stops the behavior scripts.
7. The **control server** sends a status back to the **client browser** to say it's been handled.

Let's start by extending the list of scripts (modes) and how the system handles them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Managing robot modes

We will revisit the code made in *Chapter 15*, *Voice Communication with a Robot Using Mycroft*, extend the list of modes to run, and add a menu configuration.

Let's expand the number of items our mode system knows about, as follows:

1. Open the file called `robot_modes.py`.
2. Find the `mode_config` variable in this file. We can extend it with a few more behaviors, as illustrated in the following code snippet:

```
    mode_config = {
        "avoid_behavior": "avoid_behavior.py",
        "circle_head": "circle_pan_tilt_behavior.py",
        "test_rainbow": "test_rainbow.py",
        "test_leds": "leds_test.py",
        "line_following": "line_follow_behavior.py",
        "behavior_line": "straight_line_drive.py",
        "drive_north": "drive_north.py"
    }
```

3. After the `mode_config` variable, we add a list configuring the menu. The order will match menu items on a screen. Each item has a `mode_name` setting–matching the short slug in the `mode_config` variable, and `text`—the human-readable label for the menu option, as illustrated in the following code snippet:

```
    menu_config = [
        {"mode_name": "avoid_behavior", "text": "Avoid
Behavior"},
        {"mode_name": "circle_head", "text": "Circle
Head"},
        {"mode_name": "test_leds", "text": "Test LEDs"},
        {"mode_name": "test_rainbow", "text": "LED
Rainbow"},
        {"mode_name": "line_following", "text": "Line
Following"},
        {"mode_name": "behavior_line", "text": "Drive In
A Line"},
        {"mode_name": "drive_north", "text": "Drive
North"}
    ]
```

If we want to add a behavior to the menu, we must add it to both the `menu_config` and the `mode_config` variables.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. To allow a menu user to choose a new mode without pressing a **Stop** button, we want to make the `run` method cope with this by stopping any existing process, as follows:

```
    def run(self, mode_name):
        while self.is_running():
            self.stop()
        script = self.mode_config[mode_name]
        self.current_process = subprocess.
Popen(["python3", script])
```

   This file will act as a configuration, and you can expand it to run other code. We can test this now.

5. Upload `robot_modes.py` to the robot. You should already have the *Chapter 15*, *Voice Communication with a Robot Using Mycroft*, `control_server.py` file uploaded.

6. Run this on the Pi with `python3 control_server.py`.

7. As we saw in *Chapter 15*, *Voice Communication with a Robot Using Mycroft*, we will use the `curl` command to make a request, like this:

```
curl -X POST http//myrobot.local:5000/run/test_leds
```

   This should start the **light-emitting diodes** (**LEDs**) flashing on the robot.

8. Let's change behavior—this should stop the current behavior and start a new one. Run the following code:

```
curl -X POST http//myrobot.local:5000/run/circle_head
```

   The LEDs should stop, and assuming the motors are turned on, the head should start moving.

9. Let's stop the robot by running the following code:

```
curl -X POST http//myrobot.local:5000/stop
```

We have added some further modes and configuration to `robot_modes.py` to describe those modes, and tested them. Let's check for any problems.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Troubleshooting

When requests to the menu sever fail, it can output error codes in the response. There are only three error codes we make use of in our system, as follows:

- `200`—This means that the server thinks everything is OK. There may still be a logic problem, but it's not caused a failure.
- `404`—This is shown when the server couldn't find a route. This means you may have a typo either in the request you made or in the routers on the server code. Check that they match and try again.
- `500`—This means that the server failed in some way. It is usually accompanied by a traceback/exception on the server. This can then be treated as a normal Python error.

Now that we have the mode configuration lists ready, we need the web service to display it.

The web service

In *Chapter 15*, *Voice Communication with a Robot Using Mycroft*, we'd already wired `robot_modes.py` into the `control_server.py` Flask web server. We have used Flask in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*, to render templates with a video box. In this section, we will make a menu template to show the user their options.

Let's make the necessary changes to render the template first, as follows:

1. Open `control_server.py`.
2. Extend the imports of Flask to include `render_template`, as follows:

```
from flask import Flask, render_template
from robot_modes import RobotModes
```

3. As we will have a style sheet we are changing, we need to stop devices holding a stale, cached copy of the sheet. We can do this by adding a header to all responses, like this:

```
@app.after_request
def add_header(response):
    response.headers['Cache-Control'] = "no-cache,
no-store, must-revalidate"
    return response
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. We now need to add the route that shows our menu. We will make a template called `menu.html`, which uses the `menu_config` variable to display. Most of our modes need this. Let's add the code to render the template, as follows:

```
@app.route("/")
def index():
    return render_template('menu.html', menu=mode_
manager.menu_config)
```

We now have code to render the template building on code we already had to handle `run` and `stop` requests. However, before we can run this service, we need to provide the template, `menu.html`.

The template

Our HTML template defines our display and lets us separate how the robot menu looks from how to handle the control system. This template combines HTML (seen in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*, and *Chapter 14*, *Line-Following with a Camera in Python*) and the **Jinja2** template system—a way of substituting data. See the *Further reading* section to find out more about these systems. We have a `templates` folder from *Chapter 15*, *Voice Communication with a Robot Using Mycroft*—we will add our menu template here. We could add further styling to this template; for now, we'll keep it simple.

Make a file called `templates/menu.html` and then proceed as follows:

1. Our template starts with a header that sets the page title and uses the same jQuery tool we saw before, as illustrated in the following code snippet:

```
<html>
<head>
    <script src="https://code.jquery.com/jquery-
3.5.1.min.js"></script>
    <title>My Robot Menu</title>
</head>
```

2. The body of our template has the `My Robot Menu` heading. Feel free to change this to your robot's name. The code is shown in the following snippet:

```
<body>
  <h1>My Robot Menu</h1>
```

3. Next, we have a space for a message; it's empty now though, as you can see here:

```
    <p id="message"></p>
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. The next section is a list—that is the menu itself. We use the `<ul>` tag and then a `for` loop, which creates a list item with a link for each menu item. The double brackets `{{ }}` are used to surround a placeholder, that will be replaced when run. It uses the `mode_name` setting and `text` to make that link, combining `/run` with the mode name, as illustrated in the following code snippet:

```
  <ul>
      {% for item in menu %}
        <li>
            <a href="#" onclick="run('/run/{{ item.mode_
name }}')">
                {{ item.text }}
            </a>
        </li>
      {% endfor %}
```

   We are using an onclick handler, so we can handle the `run` action in some code.

5. Before closing our list, we need to add one more menu item—the **Stop** button, as follows:

```
    <li><a href="#" onclick="run('/stop')">Stop</a></li>
  </ul>
```

6. We talked about handling the `run` action in some JavaScript code. The following code makes POST requests sending data to the web server, and then updates the message from the response. We need to put it in `<script>` tags, as follows:

```
  <script>
    function run(url) {
      $.post(url, '', response => $('#message').
html(response))
    }
  </script>
```

   The `run` function calls the `.post` method with the **Uniform Resource Locator** (**URL**) and empty data. When it receives a response, it will set the content of the message element to it. The `=>` operator is JavaScript shorthand for defining a small function—in this case, one that has the `response` parameter. An important idea in JavaScript is that a function can be a bit of data. In JavaScript, passing a function in as a parameter to another function is a common way to do things. Because we often use this, functions used that way are not even given names; they are anonymous functions or lambdas.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Now, we can close our HTML document, like this:

```
</body>
</html>
```

The nice thing with a template such as this is that you can preview this code in a browser without the server and make sense of how it should look. The following screenshot shows it in preview mode:

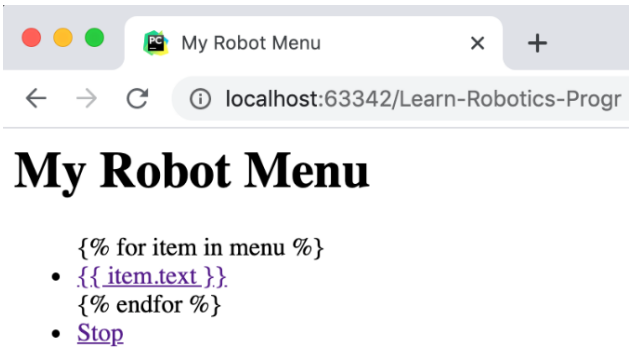


Figure 17.2 – Previewing the template

When you view the preview, as shown in *Figure 17.2*, the template placeholders are showing as the browser doesn't know how to render them.

You need to run the app to see it properly rendered.

Running it

Upload the `robot_modes.py` and `control_server.py` files to the robot, and then the `templates` folder. On the Raspberry Pi, via SSH, you can start it with the following command:

```
pi@myrobot:~ $ python3 control_server.py
 * Serving Flask app "control_server" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a
production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 270-549-285
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can now point your browser at your robot (`http://myrobot.local:5000/`) to see the menu. The following screenshot shows how it should look:

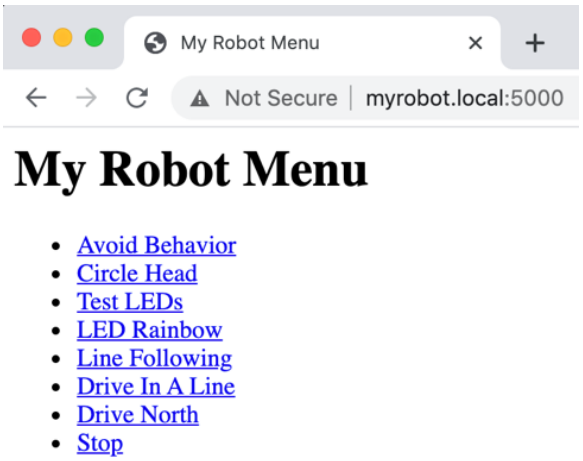


Figure 17.3 – My Robot Menu in a browser

Figure 17.3 now shows the list rendered. We now see all the menu items instead of the template placeholders. You should be able to click a mode and see the robot start that behavior. Clicking **Stop** causes the `robot_modes.py` code to send the equivalent of a *Ctrl* + *C* action to the running behavior script, making it stop.

When you click a behavior or stop, it shows the output in the message area, as shown in the following screenshot:

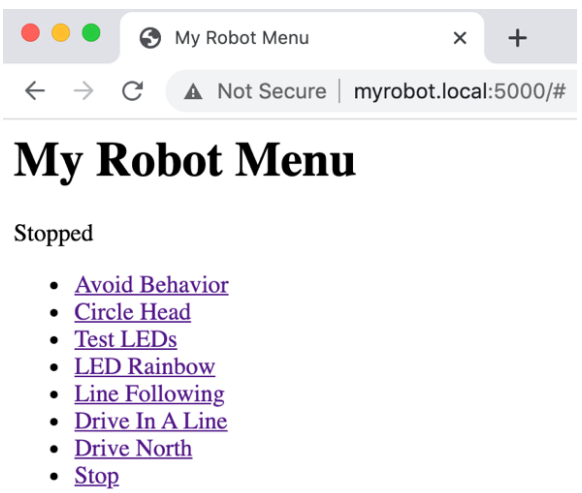


Figure 17.4 – The stopped message

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 17.4 shows the menu again. I've clicked the **Stop** button, so the menu shows the **Stopped** response message.

Notice in the following code snippet that the behavior's outputs—its `print` statements—are coming out in the web server console:

```
192.168.1.149 - - [17/Oct/2020 22:42:57] "POST /run/test_leds
HTTP/1.1" 200 -
red
blue
red
blue
red
Traceback (most recent call last):
  File "leds_test.py", line 16, in <module>
    sleep(0.5)
KeyboardInterrupt
192.168.1.149 - - [17/Oct/2020 22:43:41] "POST /stop HTTP/1.1"
200 -
```

You need to press *Ctrl* + *C* on the Pi to exit this menu server app.

> **Important note**
>
> This tiny robot web app has no security mechanism, authentication, or passwords. It is beyond this book's scope but is a serious consideration worth further research if you plan to use this on shared Wi-Fi.

There are ways to get the console output from a script onto the page. I recommend looking at the additional reading recommendations in the *Further reading* section for Flask.

Troubleshooting

Hopefully, this all works, but if you have any problems try the following steps:

- The output log shows the return codes from the web system. You can use these status codes—as you've seen before—to troubleshoot.
- `200`—The system thinks everything is OK. If it failed to run something, check the `run` function.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- `404`—Not found. Have you matched the routes?
- `500`—You should see a Python error with this too.
- If the render shows the display `{ item.text }`, this needs double curly brackets for the template system to work.
- If you see an error such as `jinja2.exceptions.TemplateSyntaxError: unexpected '<'`, then you'll need to verify you have typed out the preceding template—you are likely to have missed a closing curly bracket (}).

You now have a menu system to start different robot behaviors and stop them. You can point your phone at it—although it's not particularly phone-friendly yet. We have only scratched the surface of this, and this system is quite rudimentary.

We'll start looking at a more interesting phone interface for driving the robot, but we can first look at options other than smartphones.

Choosing a controller — how we are going to drive the robot, and why

We want to be able to control our robot with something that is handheld and wireless. Trailing a wire to our robot would make little sense. Having seen how our robot drives in *Chapter 7*, *Drive and Turn – Moving Motors with Python*, we will want a control system that directly affects the wheels.

One way to do this would be to use a Bluetooth joypad. There are a large number of these on the market, which may require specialist drivers to read. Bluetooth has a habit of dropping pairings at inopportune times.

Some joypads use a custom wireless dongle; these are far more reliable than Bluetooth but have a dongle that doesn't fit very nicely on the robot.

However, you already have a handheld device in your pocket: your phone. It has a touchscreen, capable of reading finger movements. With a bit of the right code, you can display the video between controller bars, creating a kind of robotic *periscope* you can drive around and see (it's quite tricky to drive on camera—harder than overhead). We've already been building web applications for our robot to access via Wi-Fi, and most phones can connect to that. So, instead of going out and buying a new joypad, we will make a web app that your phone can access to drive the robot and see a robot's-eye view of the world.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Design and overview

To make a phone web app, a little bit of design on how we would expect this to work is needed. This design could be as simple as a pen drawing on a scrap of paper or using a drawing tool to get professional-looking results. The next screenshot shows a screen mockup of this:

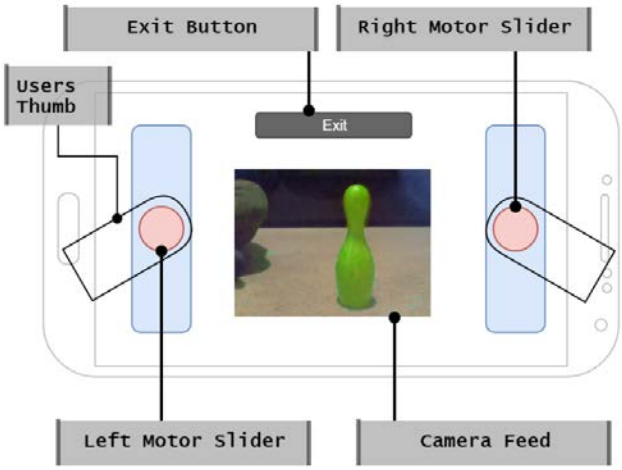


Figure 17.5 – Screen mockup of driving web app

The mockup in *Figure 17.5* shows a mobile phone screen in landscape mode. The top of the screen has an **Exit** button, and we can set this up to go to our menu after instructing the app to exit.

The middle of the screen has a video feed from the robot, using the mechanism from the *Building a Raspberry Pi camera stream app* in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*. The left and right have sliders. The next screenshot shows how this works:

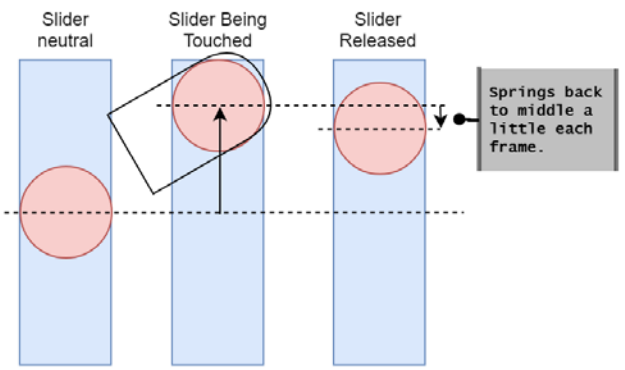


Figure 17.6 – Slider return to middle behavior

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Figure 17.6 shows the slider mechanism. As with an analog joystick, you can drag the sliders to any position on their track with touches, and when let go, they will spring back to the middle.

Note that they don't immediately drop to the middle when let go but animate back to this over a few frames. We'll need a little math to make that happen in our code.

These sliders let you drive the robot tank-style (with a joypad, you could use two analog sticks for this). Each slider controls the speed of a motor. While this sounds tricky (not like driving a car), it is an elegant way to drive a two-wheeled robot with a little practice. The further away from the middle you slide a slider, the faster the associated motor will go. We will also ensure that the robot motors will stop after a second if communication is lost.

This control of left speed and right speed is the same control system your behaviors have been using throughout the book, but has been made interactive. The next diagram shows some of the motions needed for typical moves:

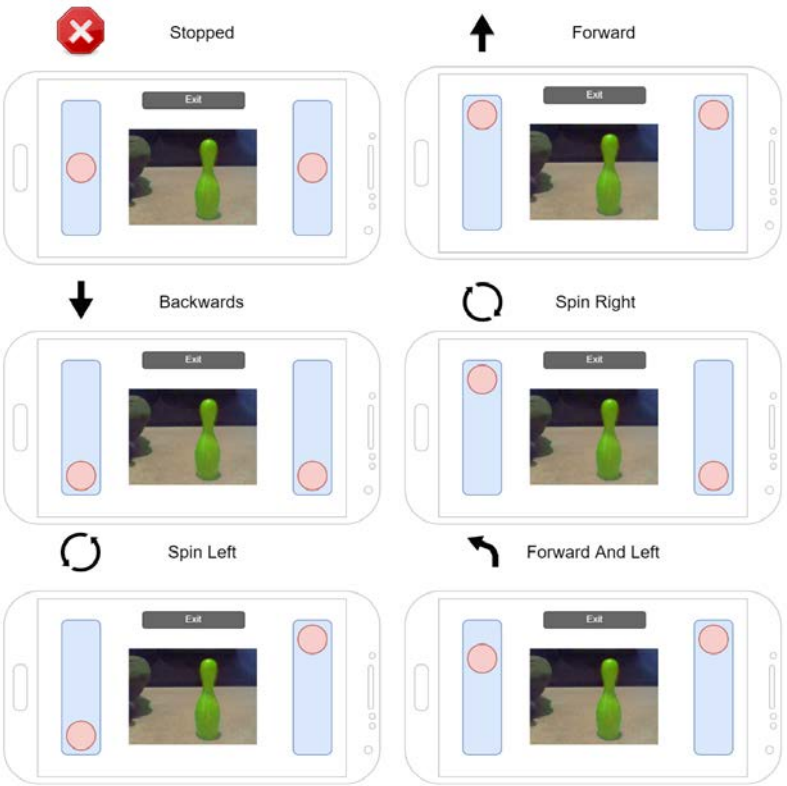


Figure 17.7 – Common moves on two sliders

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The red dots in *Figure 17.7* represent where your thumb is touching the screen. By sliding both forward the robot will drive forward, and the further you slide them, the faster it will go. A backward action slides them both back. To spin the robot, slide them in opposite directions. To drive forward and a little left or right, you slide both forward but bring the right slider a little higher than the left. You are also able to compensate for veer this way.

We have a nice user interface design. To start building this, we will plan the code blocks we will need, and write the code to make them work in the real world.

Preparing the Raspberry Pi for remote driving—get the basic driving system going

Our Raspberry Pi has already been able to run web services, using Flask to create a menu server and video servers. We can use image and control queues to make a behavior interact with a web server. We are going to reuse these capabilities. In the phone app, the slider controls will need to be smart. The next diagram shows the parts of our manual drive system:

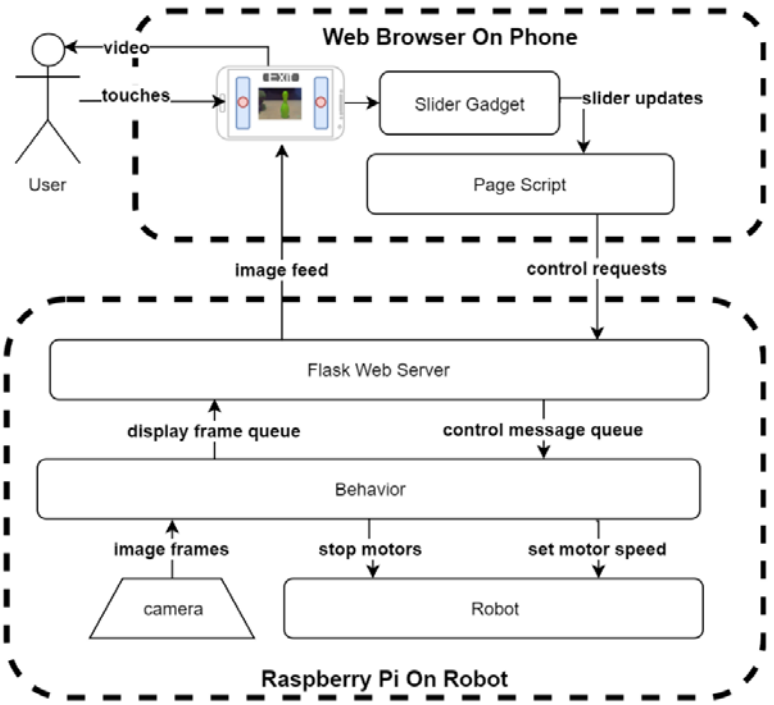


Figure 17.8 – The system overview of a manual drive app

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The dashed boxes in *Figure 17.8* show where the code is running, with the top dashed box being code running on the phone, and the lower box being code running on the Raspberry Pi in the robot. Inside the dashed boxes, the boxes with solid outlines are blocks of code or systems our code will need. At the bottom layer of *Figure 17.8*, the **Robot** box accepts the **stop motors** and **set motor speed** calls. These are from the **Behavior** box based on timeouts or the **control message queue** from the **Flask Web Server**. Meanwhile, the **Behavior** loop will also be taking **image frames** from the **camera**, encoding them and pushing them onto the **display frame queue**.

The next layer up is the **Flask Web Server**. This server consumes the **display frame queue** supplying frames to the multi-part **image feed**. The Flask server will handle **control requests** and push them onto the **control message queue**.

A **Page Script** handles **slider updates** and turns them into **control requests** using the jQuery library. The **Slider Gadget** turns **touches** into **slider updates** (it will be doing animation and converting for this).

The page itself uses an `img` tag to display the **video** feed, as before, and places the slider widgets. The **Exit** button makes a control request.

The **Page Script** and **Slider Gadget** will require JavaScript and **Cascading Style Sheets** (**CSS**) programming. Before we start that, we need to take the image core from *Chapter 14*, *Line-Following with a Camera in Python*, and build more features to deliver the code to the browser.

Enhancing the image app core

To build this, we will start by adding some static file links and reusing the image app core we last used in *Chapter 14*, *Line-Following with a Camera in Python*.

Static files do not cause the robot to do something; the system passively serves them. We will be serving JavaScript and CSS, along with a local copy of the jQuery library. Flask does this automatically.

Let's set up the static files folder, as follows:

1. Create a `static` folder. We will put JavaScript and CSS code in this `static` folder.
2. We will make a local copy of the jQuery library. Make a `lib` directory under `static`.
3. Download jQuery from `https://code.jquery.com/jquery-3.5.1.min.js`, press the browser **Save** button, and store it in the `lib` folder. You should have a `static/lib/jquery-3.5.1.min.js` file.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. In the `image_app_core.py` file, we also need to stop it using cached files so that it reloads our CSS and JavaScript files, as follows:

```
@app.after_request
def add_header(response):
    response.headers['Cache-Control'] = "no-cache,
no-store, must-revalidate"
    return response
```

The app core now has a static copy of jQuery we can use offline, so our phone doesn't have to rely on a good signal to talk to the robot.

Writing the manual drive behavior

The next part we will need is the behavior. It builds on concepts from the code seen before in *Chapter 14*, *Line-Following with a Camera in Python*, with control messages changing motor speeds and a plain video output.

This system will have a timeout—if no control messages arrive for 1 second, it will stop driving. It can be quite frustrating to watch a robot drive off into the distance or off a desk, so it will revert to stopping if nothing is making sense.

Let's build it, as follows:

1. Start a file called `manual_drive.py` with imports for the camera and control, like this:

```
import time
from robot import Robot
from image_app_core import start_server_process, get_
control_instruction, put_output_image
import camera_stream
```

2. We can declare what we want the timeout threshold to be in seconds, as illustrated in the following code snippet:

```
TIMEOUT_IN_SECONDS = 1
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We'll make a `ManualDriveBehavior` class. In this, we'll store a `robot` object and track time, as illustrated in the following code snippet:

```
class ManualDriveBehavior(object):
    def __init__(self, robot):
        self.robot = robot
        self.last_time = time.time()
```

4. Next, build the control section of this behavior. It resets the last time for every instruction. The code can be seen in the following snippet:

```
    def process_control(self):
        instruction = get_control_instruction()
        while instruction:
            self.last_time = time.time()
            self.handle_instruction(instruction)
            instruction = get_control_instruction()
```

   Notice that this will loop until there are no further instructions, since you could queue more than one while waiting for a camera frame. We prime this with the next instruction. It delegates handling to `self.handle_instruction`.

5. Our code processes the instruction in `handle_instruction`. This instruction is a dictionary, with an instruction name and parameters as its members. We can check if this command is `set_left` or `set_right`, as illustrated in the following code snippet:

```
    def handle_instruction(self, instruction):
        command = instruction['command']
        if command == "set_left":
            self.robot.set_left(int(instruction['speed']))
        elif command == "set_right":
            self.robot.set_
right(int(instruction['speed']))
```

   If it matches, we set the robot motor speed. The speed will currently be a string, so we use `int` to convert it into an integer number for our motors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. We also need to handle the `exit` command, as follows:

```
            elif command == "exit":
                print("stopping")
                exit()
```

7. It would be useful, at least when testing, to know whether we have an unknown instruction. Let's handle that case by raising an exception, as follows:

```
            else:
                raise ValueError(f"Unknown instruction:
{instruction}")
```

8. Our app also needs to make a display, putting the frame on the server image queue, as follows:

```
    def make_display(self, frame):
        encoded_bytes = camera_stream.get_encoded_bytes_
for_frame(frame)
        put_output_image(encoded_bytes)
```

9. The behavior then has a `run` method to perform the setup and the main loop. We start by setting the pan and tilt to look straight ahead, warm up the camera, and stop the servos, as follows:

```
    def run(self):
        self.robot.set_pan(0)
        self.robot.set_tilt(0)
        camera = camera_stream.setup_camera()
        time.sleep(0.1)
        self.robot.servos.stop_all()
        print("Setup Complete")
```

10. We then loop over frames from the camera and process control instructions, as follows:

```
            for frame in camera_stream.start_stream(camera):
                self.make_display(frame)
                self.process_control()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. We finally make it auto stop based on the timeout, as follows:

```
            if time.time() > self.last_time + TIMEOUT_IN_
SECONDS:
                self.robot.stop_motors()
```

12. We add the top-level code to create and start the components, as follows:

```
print("Setting up")
behavior = ManualDriveBehavior(Robot())
process = start_server_process('manual_drive.html')
```

13. We still want to ensure we stop the server when we exit or hit an error, so we run the following code:

```
try:
    behavior.run()
except:
    process.terminate()
```

The behavior backend is complete but it needs a template to see it, along with style and code to run on the phone.

The template (web page)

The template is where we will place our sliders and some of the code to handle them.

Let's get into it, as follows:

1. Create a `templates/manual_drive.html` file. Start with the HTML preamble, as follows:

```
<html>
    <head>
```

2. We want the display to fit on a phone, adapting to the display size. We also don't want the user's touch interactions to accidentally scale the display. This line of code tells the browser that this is our intention:

```
        <meta name="viewport" content="width=device-
width, initial-scale=1.0, user-scalable=no">
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. We want to style this (and potentially our other interfaces). For this, we use a `display.css` style sheet, as illustrated in the following code snippet:

```
        <link rel="stylesheet" type="text/css" href="/
static/display.css?">
```

4. We are going to use the jQuery library to make things interactive, and we'll build a touch-slider system. These are the HTML equivalent of imports:

```
        <script src="/static/lib/jquery-3.5.1.min.js"></
script>
        <script src="/static/touch-slider.js?"></script>
```

We are going to place a very specific bit of style here in this file. The rest comes from the style sheet. We want this behavior's view to take up the whole screen and not scroll. The code can be seen here:

```
        <style>html, body {margin: 0; height: 100%;
overflow: hidden}</style>
```

5. The head ends with a title to go on the top of the tab, as follows:

```
        <title>Manually Drive The Robot</title>
    </head>
```

6. We now start the body with the first slider; we define this with **Scalable Vector Graphics** (**SVG**). SVG let us draw things inside a browser. HTML requires us to contain SVG parts in a `svg` tag, which we'll use to make the slider track, as illustrated in the following code snippet:

```
    <body>
        <svg id="left_slider" class="slider_track"
viewBox="-10 -100 20 200">
```

The `slider_track` class lets us style both tracks—we use HTML classes to identify multiple objects. The `left_slider` ID will help us position and add touch events to it. IDs in HTML are usually used to reference one object uniquely.

The `viewBox` attribute defines the dimensions of the drawing internal to `svg` as lower *x*, lower *y*, width, and height. View box coordinates make sense even if we scale the `svg` element for a different device. The height range is -100/100, equivalent to the motor speeds, and the width range is -10 to +10.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Inside the container, we will draw a circle. The circle needs a radius `r`, which we can give in view box units. The center will be 0 in both directions. The code for this is shown here:

```
            <circle r="18" class="slider_tick"/>
```

   The circle's color will come from the style sheet.

8. Next, we need an `Exit` link to finish the behavior. It has a class and ID to style it, as illustrated in the following code snippet:

```
        <a class="button" id="exitbutton" href="/
exit">Exit</a>
```

   We will associate the `button` class with fonts and colors, and we can use the style for other buttons (for example, to enhance the menu app). We'll use the `exitbutton` ID to position this in the place we designed before.

9. Next, we have our video block. The `img` tag for the video is contained inside a `div` tag to preserve our video ratio on any size screen while letting it resize to fit the space, as illustrated in the following code snippet:

```
        <div id="video"><img src="{{ url_for('display')
}}" /></div>
```

10. The right slider is a repeat of the left, with only the ID being different. You could copy and paste the left code, changing the ID. The code can be seen here:

```
        <svg id="right_slider" class="slider_track"
viewBox="-10 -100 20 200">
            <circle r="18" class="slider_tick"/>
        </svg>
```

11. We will need some JavaScript code in our HTML for the sliders. The code on the page will link slider code to the graphics we have and to the motors. First, we declare the JavaScript block, as follows:

```
        <script type="text/javascript">
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. Add a function to send motor controls to the robot. It takes a name (left or right) and a speed, as illustrated in the following code snippet:

```
            function set_motor(name, speed) {
                $.post('/control', {'command': 'set_' +
name, 'speed': speed});
            }
```

We POST this control instruction to the server.

13. The next bit of code must only run after the page has completed loading; we want to ensure the preceding JavaScript libraries are fully loaded. jQuery has a special function, `$()`, which will run any function passed to it when the page has completed loading, as illustrated in the following code snippet:

```
            $(() => {
```

14. We need to link the exit button to a POST request, which forwards to the menu when done, as illustrated in the following code snippet:

```
                $('#exitbutton').click(function() {
                    $.post('/control', {'command':
'exit'});
                    window.location.replace('//' +
window.location.hostname + ":5000");
                });
```

15. We set up the sliders and link them with their `svg` element IDs and `set_motor` so that they will update this every time they change, as illustrated in the following code snippet:

```
                makeSlider('left_slider', speed => set_
motor('left', speed));
                makeSlider('right_slider', speed => set_
motor('right', speed));
            });
```

For each side, we use `makeSlider`, which uses `id` for the ID of the object we are turning into a slider (a `svg` track), and a function to call when the slider has changed.

16. We now end our page by closing all the tags, as follows:

```
        </script>
    </body>
</html>
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

This page has no style; by default, the video and sliders have no shape, size, or color—so, if you try to load this, it will show a blank page. We've yet to tell the browser where we want things on the page or which colors to make them. We also don't have the slider code yet.

We've made the code to send the exit button and link sliders with tags. In the next section, we'll add a style sheet to make things visible.

The style sheet

We can now give our app some style. Style sheets take time to tune and get just right, so this is just a taste of what it can do. If you think my color choices are terrible, please feel free to substitute your own; I suggest using w3c colors at `https://www.w3schools.com/colors/default.asp`. You can use named or hex (`#1ab3c5`) colors.

The essence of CSS is to select elements on the page and associate style attributes with them. CSS style sections start with a **selector** to match HTML page objects. Some examples are tag names, class names prefixed with a full stop, or IDs prefixed with a # mark. For a comprehensive look at CSS selectors, see the *Further reading* section. Each section uses braces { } to delimit a section of style. A section's styles consist of a property name, a colon `:`, and a setting. A semicolon `;` follows these to end each setting.

Let's make the style, as follows:

1. Create a `static/display.css` file to hold this style information.

   We can set our slider track to 10% of the viewport width—that is, 10% as big as the screen. CSS has a special unit, `vw`, for this, along with `vh` for percentage of the viewport height. See the *Further reading* section for notes on CSS units. This code uses the `.slider_track` CSS selector, which applies to all objects with that class. Both sliders have this class, so changes here affect both of them. The code can be seen here:

   ```
   .slider_track {
       width: 10vw;
       height: 90vh;
   ```

2. We'll give the slider track a solid blue border and a light blue background to match our mockups, as follows:

   ```
       border: 1px solid blue;
       background-color: lightblue;
   }
   ```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. To style the tick, the circle we see on the sliders, we can add a light pinkish fill color, like our mockups, as follows:

```
.slider_tick {
    fill: mistyrose;
}
```

4. Next, we want to position the sliders (by their IDs) to the left and right. When making a display match closely to the screen mockup, we can use **absolute positioning** with viewport percentages to say exactly where things should be, as follows:

```
#left_slider {
    position: absolute;
    left: 5vw;
    top: 5vh;
}
#right_slider {
    position: absolute;
    right: 5vw;
    top: 5vh;
}
```

5. You can try this now by uploading it, stopping the running behavior, starting it again, and then reloading. The sliders look better, but the exit button and video are in the wrong place.
6. Let's make the exit button more like a button. Styles under `.button` will apply to all buttons with the same class. We will make it a block—an element that uses width and height properties. This block is 10% of the viewport height. The code can be seen here:

```
.button {
    display: block;
    height: 10vh;
```

7. Then, we align the text in the middle with `line-height` and `text-align`, then use `2em` to mean twice normal text size, as follows:

```
    text-align: center;
    font-size: 2em;
    line-height: 10vh;
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. We want to take the underline off the button text, which you normally get with a link. We'll also give it some color, a blue background with white text, as follows:

```
    text-decoration: none;
    background-color: blue;
    color: white;
}
```

9. We specify more about the exit button using its ID. We will set its width and the top but use `auto` margins to center it, as follows:

```
#exitbutton {
    width: 40vh;
    margin-top: 5vh;
    margin-left: auto;
    margin-right: auto;
}
```

   Trying this out, you should now see the exit button in the right place.

10. Next, we style the video. We want to center the video on the screen. The outer video element can do that for us, like this:

```
.video {
  text-align: center;
}
```

11. We can then specify the position and size of the inner image block. We want it to be 20% from the top of the screen using a `vh` measurement. The `vmin` unit is a percentage of the screen's minimum dimension; it ensures that this block is never so large that it would obscure the two slider bars. We make the height automatically scale. We select `#video img` to apply this style to the `img` object contained in the `video` object, as illustrated in the following code snippet:

```
#video img {
    margin-top: 20vh;
    width: 80vmin;
    height: auto;
}
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Our page is fully styled. You can try this now to see how it looks. Upload the whole folder (including templates) to the robot, and then run `python3 manual_drive.py`. Point a desktop browser at `http://myrobot.local:5001/`, substituting your robot's hostname or address to see it. A desktop browser is good to discover errors in the HTML or JavaScript code. At the time of writing, Firefox and Chrome support emulating mobile devices in the browser and touch events. It should look like the mockup with real video, as illustrated in the following screenshot:

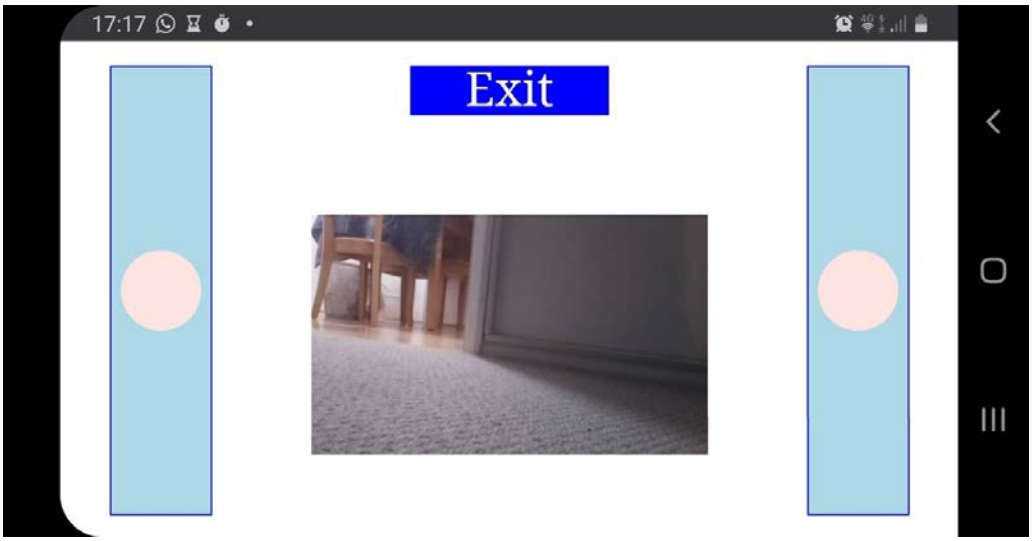


Figure 17.9 – Screenshot of app running on the phone

Figure 17.9 shows the app running on a real phone. The slider bars still don't do anything yet. Note that you may need to force your browser to reload the style sheet.

We now need to add the slider code.

Creating the code for the sliders

The sliders need to respond to touch events, moving the circle to match the touch location and sending an update message to show how far this movement is from the middle. The sliders will automatically return to the center when the touch events stop. JavaScript lets us run code in the browser, so we'll create a JavaScript `makeSlider` function.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

First, we want to see how touches translate to slider positions and motor speeds. This is illustrated in the following diagram:

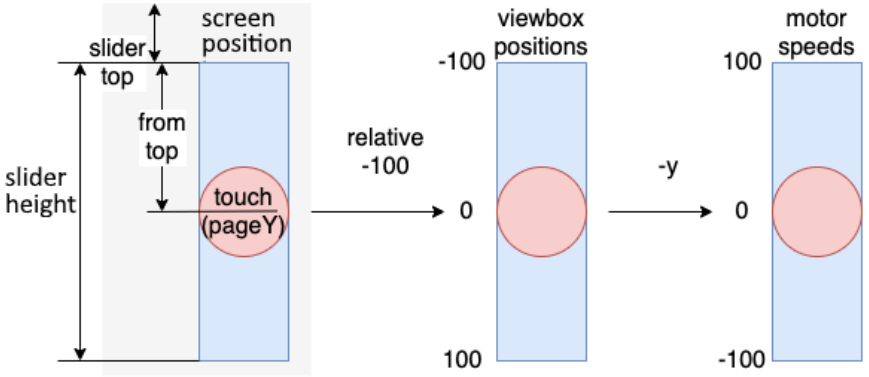


Figure 17.10 – Going from touch events to motor positions

Our sliders have some complexity in their positions, shown in *Figure 17.10*. When a user touches a screen, the position arrives in terms of screen coordinates. We first need to find where it is in the slider by taking away the slider's top coordinates. We will need to divide that result by the slider height, multiply by 200, and then subtract 100 to give us the viewbox position (the same system used to draw the SVG). In viewbox coordinates, the top is -100, but for our motors to go forward we need +100, so we must negate the viewbox position to get the motor speed.

Our script will set up the data needed to move the slider and internal functions to map to the slider events, manage the slider's movement, and call back the `manual_drive.html` code (or any other code) when we move the sliders. Let's make the slider code, as follows:

1. We will put this in `static/touch-slider.js`. As we are in a `.js` file, the `<script>` tags are not needed.
2. We create the `makeSlider`, function, a factory function to make everything the sliders need, as follows:

```
function makeSlider(id, when_changed) {
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The first thing we need is some internal data. The code needs to know if we are touching the slider so that it won't try to move back while we're still touching it. We need to know if the touch position has changed and keep track of its position. Finally, we'll find our slider by its ID and keep the found object to use, as follows:

```
    let touched = false;
    let changed = false;
    let position = 0;
    const slider = $('#' + id);
```

4. We then need some functions to deal with the slider. We'll start with a function to update the position, ensuring the tick is updated, that we only use whole numbers (because the browser won't accept decimal points here), and that we update the `changed` flag, as illustrated in the following code snippet:

```
    const set_position = function(new_position) {
        position = Math.round(new_position);
        slider.find('.slider_tick')[0].setAttribute('cy',
position);
        changed = true;
    };
```

5. The next thing is handling touch events. Event handlers are functions that get called when something happens (such as the exit button handler). Touch events have three events: `touchstart`—when someone starts touching a screen, `touchmove`—when a touch moves to another area, and `touchend`—when the touch stops. We won't use `touchstart`, so we'll start with making an anonymous `touchmove` function, as follows:

```
    slider.on('touchmove', event => {
        let touch = event.targetTouches[0];
```

Notice we immediately get a `touch` variable from the event data. We get a list of touches, but we are only using the first one.

6. We then get the relative position of this touch from the top of the slider, as follows:

```
        let from_top = touch.pageY - slider.offset().top;
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. We can use this with the height to convert the touch position into a number from -100 to +100, matching the SVG viewbox coordinates, as follows:

```
        let relative_touch = (from_top / slider.height())
* 200;
        set_position(relative_touch - 100);
```

8. Since the code has received a touch event, we should set the `touched` flag to `true`. We must also prevent the touch event from having any other effects, as illustrated in the following code snippet:

```
        touched = true;
        event.preventDefault();
    });
```

9. Since we've set a flag to say that the touch event is occurring, we should also clear it (set it to `false`) when the touch event ends, as follows:

```
    slider.on('touchend', event => touched = false);
```

10. Our system is animated, so it needs to have an update cycle to return to the middle. The update should only move the tick if we are not touching the slider, so it stays where you keep your thumb. When the touch has stopped and it's still not at the zero position, we should update the position, as follows:

```
    const update = function() {
        if(!touched && Math.abs(position) > 0) {
```

11. This next part looks a bit like the **Proportional-Integral-Derivative** (**PID**) controller code in that there's an error multiplied by a proportional component. We scale the error by a factor of 0.3 and add/subtract an extra 0.5 to get it closer to a 1% minimum. Every time this is updated, it moves the slider closer to the middle. The code can be seen here:

```
            let error = 0 - position;
            let change = (0.3 * error) + (Math.
sign(error) * 0.5);
            set_position(position + change);
            // console.log(id + ": " + position);
        }
    };
```

This code is also a great place to log the position—something we can use when it goes wrong.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. To run this `update` function frequently, we can use the `setInterval` built-in function, which runs a function repeatedly on every interval. This display update should be short to keep it responsive. The timings are in milliseconds. The code can be seen here:

```
        setInterval(update, 50);
```

13. Besides updating the image, we also need to call the `when_changed` function. We only want to do so when something has changed and then reset the `changed` flag, so we don't call it when idle. We'll call this `update_when_changed`. This checks for changes and runs less frequently than the display update, so it doesn't flood the `when_changed` handler and the queue on the robot. The code can be seen here:

```
        const update_if_changed = function() {
            if(changed) {
                changed = false;
                when_changed(-position);
            }
        };
        setInterval(update_if_changed, 200);
    }
```

    Note that we negate the position; this is so the top of the screen (-100) will become motors going full forward (+100). Don't forget the closing bracket for the `makeSlider` function.

You should now be ready to run the whole system.

Running this

You can now upload the entire set of files to a folder on your robot. As before, you can use `python3 manual_drive.py` to run this.

You can use developer mode on a browser to view the web page before trying it on a phone, as follows:

1. Point your browser (Chrome or Firefox) at `http://myrobot.local:5001` (using your robot's hostname or address).
2. Right-click on your page and click the menu item labeled **Inspect or Inspect Element**.
3. In the developer tools, there will be buttons for emulating phone devices and touch events. Enable the phone emulation.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Try to shake out any problems in a desktop browser first. Check that dragging the sliders has the desired results and click on the **Console** button to see if there are errors from the JavaScript.

   Common problems in JavaScript and CSS are missing punctuation such as semicolons, commas, or brackets. Having class or ID selectors that do not match (or are missing the required dot/hash mark syntax) will make styles fail to apply or element lookups in JavaScript produce no results.

5. To use on the phone, you will need to use your robot's IP address, as major smartphone brands do not support `.local` addresses. You can find this from your desktop with `ping myrobot.local`, as illustrated in the following code snippet:

   ```
   $ ping myrobot.local
   PING myrobot.local (192.168.1.107): 56 data bytes
   64 bytes from 192.168.1.107: icmp_seq=0 ttl=64 time=5.156
   ms
   ```

   This example shows the robot's IP address to be `192.168.1.107`. Your address will be different; note that down, and put that in the phone browser with the port. An example for my robot is `http://192.168.1.107:5000`.

6. With the phone, you should be able to use your thumbs to drive the robot.

It will take some practice to drive the robot manually. I suggest practicing overhead driving first, and when you have got the hang of that, try navigating through the camera. The camera frame rate is not very high, and this frame rate currently constrains the driving loop.

Troubleshooting

This is a fairly complex combination of Python, HTML, JavaScript, and CSS. Try these if you've run into trouble:

- If you see errors from Python, verify the line of code against the preceding code.
- If things are not working on the web page, try out the phone emulation in browser mode, as suggested previously, then select the inspector **Console** tab and try the operation again. This will show JavaScript errors.
- If the display appears wrong, with parts out of place or in the wrong color, verify that the CSS/style sheet sections and the HTML are correct.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- If you receive `404` errors, ensure that the URLs in the HTML match the routes in the Flask/Python code.
- If your robot seems to be pausing and then spending a while catching up with your events, you could adjust the `update_if_changed` interval time to something longer.

You now have a robot you can drive remotely with the phone while seeing through its camera. You've seen how to handle touch events and use style sheets with SVG to make custom widgets. You've used JavaScript to bring the widget to life with animation and send control messages back to the robot.

In our next section, we'll make the menu more touch-friendly so that we can control the robot mostly from the phone.

Making the robot fully phone-operable

The goal here is to make it so that we can drive the robot completely from the phone. We need to ensure that the robot is ready to run when we turn it on, and make sure that the menu is usable from a phone. The menu we made earlier doesn't seem very touch-friendly. It also will not successfully run any of the behaviors with displays using Flask. We will make the menu buttons bigger and more touch-friendly, using styles similar to our manual drive behavior. The menu will also load our server page after clicking a behavior with a server such as this one or the last chapter's visual tracking behaviors.

Let's fix the Flask behaviors first.

Making menu modes compatible with Flask behaviors

If you've already tried running Flask-based behaviors (such as those with a camera) in the control server, you will have noticed some very odd behavior. Your behavior will appear to do the right thing with sensors on the robot, but the web service fails to do anything useful on port `5001`.

Flask uses subprocesses to manage its debug mode, which interferes with our use of them. We don't need debug mode, so the fix is to remove debug mode by doing the following:

1. Open `control_server.py` and jump to the last few lines.
2. Remove `debug=True` from the `app.run` line by running the following code:

```
app.run(host="0.0.0.0")
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can now add the manual drive and color-tracking and face-tracking behaviors to the control server, and they will start properly.

Loading video services

When we click on a menu option for a video server-based behavior, after it starts we need to send our browser to port `5001` on our robot to see its output.

Our `menu.html` file currently takes the response from the `control_server` process and puts this into the message box. We can upgrade this to instruct the code to do something else. We can start by configuring the items that need to show a server page in `mode_config` variable.

Each item in the `mode_config` variable holds only the mode script; we can update this to have both a script and whether it needs to show a server, as follows:

1. Open `robot_modes.py`.
2. In `mode_config`, we will take the simple text naming the script (such as `"avoid_behavior.py"`) and replace it with a dictionary, allowing a simple case (`{"script": "avoid_behavior.py}`) or a more complex case (`{"script": "manual_drive.py", "server": True}`). You'll need to change that on all items throughout the `mode_config`. The code is shown in the following snippet:

```
    mode_config = {
        "avoid_behavior": {"script": "avoid_behavior.
py"},
        "circle_head": {"script": "circle_pan_tilt_
behavior.py"},
        ...
```

3. We then need to update the server-type scripts in `mode_config` variable using the more complex case, as follows:

```
        "color_track": {"script": "color_track_behavior.
py", "server": True},
        "face_track": {"script": "face_track_behavior.
py", "server": True},
        "manual_drive": {"script": "manual_drive.py",
"server": True}
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've added the new manual drive behavior too. When you added the `manual_drive` configuration here, you need to add this to `menu_config` variable too so that it shows up on the menu.

4. We need to modify the `run` method to pick the script from this changed structure, as follows:

```
    def run(self, mode_name):
        while self.is_running():
            self.stop()
        script = self.mode_config[mode_name]['script']
        self.current_process = subprocess.
Popen(["python", script])
```

5. Next, we need to check if we should redirect if the mode is a server and the current process is alive. I've added the explicit `is True`, to make it clearer that the value is a `True`/`False` flag, as illustrated in the following code snippet:

```
    def should_redirect(self, mode_name):
        return self.mode_config[mode_name].get('server')
is True and self.is_running()
```

We've prepared `robot_modes.py`. The `control_server.py` file sends responses to the web page. Let's use the same trick we did with the `mode_config` and return a dictionary with data instead of just a string, as follows:

1. We will use **JavaScript Object Notation** (**JSON**) to format the response. Open `control_server.py` and add `jsonify` to the Flask imports, as illustrated in the following code snippet:

```
from flask import Flask, render_template, jsonify
```

2. Next, we replace the `run` method so that it creates the `response` dictionary, as follows:

```
@app.route("/run/<mode_name>", methods=['POST'])
def run(mode_name):
    mode_manager.run(mode_name)
    response = {'message': f'{mode_name} running'}
```

This response is the basic message.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. If we intend to redirect, we should send the `redirect` setting with our response, as follows:

```
    if mode_manager.should_redirect(mode_name):
        response['redirect'] = True
```

4. We need to send the response, encoded as JSON. JSON is an easy way to get data to JavaScript from Python—it is especially good with dictionary data. Run the following code:

```
    return jsonify(response)
```

5. Since we also sent a message back in the `stop` command, we should wrap it in the same way, like this:

```
@app.route("/stop", methods=['POST'])
def stop():
    mode_manager.stop()
    return jsonify({'message': "Stopped"})
```

The control server is able to send the `response` dictionary, and redirect if needed. The other side of this, now receiving the JSON object, requires changes in the page scripts to handle the new response. Proceed as follows:

1. Open `templates/menu.html` and find the `run` function, as illustrated in the following code snippet:

```
    function run(url) {
```

2. The message handling here needs to change. We need to set the message element HTML using the message element from our response, as follows:

```
      $.post(url, '', response => {
          $('#message').html(response.message);
```

3. However, we can also check if we need to redirect. If so, we use the same trick we did in the previous *The template* section for the exit button in the manual drive behavior, but in a timeout, as follows:

```
          if(response.redirect) {
             setTimeout(() => window.location.
replace('//' + window.location.hostname + ":5001"),
3000);
          }
      })
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `setTimeout` function calls a function after a specified time. We give it 3,000 milliseconds (3 seconds), which gives a video behavior time to warm up first.

If you upload this and run `python3 control_server.py`, you'll see the menu is now more functional but looks quite plain. You should be able to click on the tracking or driving behaviors and, after 3 seconds, be redirected to their page. Clicking the exit buttons should take you back to the menu.

Time to give it some style.

Styling the menu

We've already used a style sheet in the manual drive demo to make the exit button look better. This menu is a set of buttons too. We can build on that style and make the menu more phone-friendly.

Making the menu template into buttons

We have an existing style sheet in `static/display.css`. We can make further use of this in the menu, perhaps with a few tweaks. Our menu template can be optimized to make the most of that style sheet too. Proceed as follows:

1. Open `templates/menu.html`. We will add a link to the style sheet. We can add a `charset` definition too, as follows:

```
<head>
    <script src="https://code.jquery.com/jquery-
3.5.1.min.js"></script>
    <title>My Robot Menu</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css" href="/static/
display.css">
</head>
```

2. The menu template uses a list of items for the menu. Adding a `menu` class to that list and a `button` class to the links lets us use the existing style for them, as illustrated in the following code snippet:

```
    <ul class="menu">
      {% for item in menu %}
        <li>
            <a class="button" href="#" onclick="run('/
run/{{ item.mode_name }}')">
                {{ item.text }}
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            </a>
          </li>
        {% endfor %}
      <li><a class="button" href="#" onclick="run('/
stop')">Stop</a></li>
```

3. Now, open up `static/display.css`, where we will define the style for the `menu` class, as follows:

```
.menu {
    width: 100%;
    margin-top: 0;
    margin-bottom: 0;
    padding: 0;
}
```

We make the list container fill the screen width without any extra margins (space around the outside of the item) or padding (space between the inside of the item and its child list items).

4. The menu consists of list items. By default, these get a dot: a bullet point. We want to set this to `none` (no shape) to remove the bullet point. We can use CSS `list-style` properties to change that. The selector here applies to list items (`li`) that are children of a `.menu` class object. The code can be seen in the following snippet:

```
.menu li {
    list-style-type: none;
    list-style-position: initial;
}
```

5. To make this touch-friendly, we make the buttons the same width. `60vw` (60% of the viewport width) should be wide enough. We use the margin `auto` trick to center this. We can also add a 1-pixel light blue border to them, as illustrated in the following code snippet:

```
.menu .button {
    margin-left: auto;
    margin-right: auto;
    width: 60vw;
    border: 1px solid lightblue;
}
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Upload the whole directory and start the menu server with `python3 control_server.py`. This menu should now look more phone-friendly.

You've now seen how to make our control server work nicely on a smartphone, and you should be getting a little more comfortable with the interactions of JavaScript, HTML, and CSS with Python. However, this system has a flaw—we are still starting it from an SSH terminal. Let's see how to fix this.

Making the menu start when the Pi starts

You now have a menu system launching robot behaviors. Using SSH to log in is great to debug, see problems, and fix them. However, when you want to demonstrate your robot, a SSH session will become inconvenient.

The ideal is to turn on the robot, wait for a light to come on, then point your phone browser at it to control it.

We are going to do two things to make this useful, as follows:

- Use an LED to indicate that it's ready (in menu mode) to allow the robot to tell us before our phone has linked to the page
- Use `systemd` to automatically start the menu Flask server when we turn on the robot

Let's get stuck in with the lights.

Adding lights to the menu server

We won't want the whole robot class loaded in our menu, but it can use the lights to indicate our robot is now ready. We will import the LED system, turn it on as the server starts, and then turn it off/release it when the first `/run` request arrives. Proceed as follows:

1. Open the `control_server.py` file and import the LEDs, like this:

```
from robot_modes import RobotModes
from leds_led_shim import Leds
```

2. We need to set up our LEDs and turn one LED green by running the following code:

```
mode_manager = RobotModes()

leds = Leds()
leds.set_one(1, [0, 255, 0])
leds.show()
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. When we run something, we know that someone's used the menu. In our `run` method, we can clear the LED. Since we only want to do it once, we can set the global LEDs to `None` and then check this next time. Note in the following code snippet that we are inserting the highlighted code into the existing `run` function:

```
def run(mode_name):
    global leds
    if leds:
        leds.clear()
        leds.show()
        leds = None
...
```

You can test this by uploading the menu server code and rerunning it. The LED should light when it starts, and then when you select another behavior, it will go out. It should work correctly to move from the menu to the LED test behavior.

Using systemd to automatically start the robot

The **systemd** tool is designed for automatically starting programs on Linux. This tool is perfect for starting the menu/control server so that the robot is ready to drive. See the *Further reading* section for more information about `systemd` in Raspberry Pi.

Registering a service is done by creating a unit file and copying it into the right folder on the Pi. Proceed as follows:

1. Make a `menu_server.service` file. Start this with a description, and tell `systemd` to start our service after we have networking on our Raspberry Pi, as illustrated in the following code snippet:

```
[Unit]
Description=Robot Menu Web Service
After=network.target
```

2. Now, we tell `systemd` we want this to start as the Pi is ready for users to log in, as illustrated in the following code snippet:

```
[Install]
WantedBy=multi-user.target
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The `Service` section shown in the following snippet configures how to run our code:

```
[Service]
```

4. The working directory is where you have copied your robot files to—for example, `/home/pi`. We can also set the `pi` user we've been using the whole time. The working directory is how your code finds its other components. Have a look at the following code snippet:

```
WorkingDirectory=/home/pi
User=pi
```

5. The `ExecStart` statement tells `systemd` the command to run the service. However, it does not assume a path the way a shell would, so prefix the `python3` command with `/usr/bin/env`, as follows:

```
ExecStart=/usr/bin/env python3 control_server.py
```

6. You now need to set this up on the Raspberry Pi. Upload this file to your Raspberry Pi home directory.
7. You'll need `sudo` to copy it into the system configuration. Type this via SSH on the Pi. Note you will see permission errors if you miss the `sudo` command. The code can be seen here:

```
$ sudo cp menu_server.service /etc/systemd/system/
```

8. We should now ask `systemd` to load our configuration and then enable our service, as follows:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable menu_server
```

9. The system will confirm you've enabled it with this message:

```
Created symlink /etc/systemd/system/multi-user.target.
wants/menu_server.service → /etc/systemd/system/menu_
server.service.
```

10. You can then try starting your service with this command:

```
$ sudo systemctl start menu_server
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

If starting this server is successful, you will see a green light go on, showing it is ready. You will then be able to point your browser at the robot and control it.

Let's just check that this has worked.

Troubleshooting

Things can go wrong here—if so, try these steps to fix it or find out more:

1. Starting/enabling the menu server with `systemd` may fail, and you will see `Unit menu_server.service is not loaded properly: Invalid argument` if there are problems with the `menu_server.service` file. Please verify its content, copy it back over, and rerun the `sudo` commands to install the new file.
2. If you want to see more of what the server is doing, you can use this command:

```
$ systemctl status menu_server
```

The Pi will then respond with something like this:

```
● menu_server.service - Robot Menu Web Service
  Loaded: loaded (/etc/systemd/system/menu_server.
service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2020-10-21 23:41:55
BST; 2s ago
 Main PID: 1187 (python3)
    Tasks: 1 (limit: 860)
   Memory: 10.0M
   CGroup: /system.slice/menu_server.service
           └─1187 python3 control_server.py

Oct 21 23:41:55 myrobot systemd[1]: Started Robot Menu
Web Service.
Oct 21 23:41:56 myrobot env[1187]:  * Serving Flask app
"control_server" (lazy loading)
Oct 21 23:41:56 myrobot env[1187]:  * Environment:
production
Oct 21 23:41:56 myrobot env[1187]:    WARNING: This is
a development server. Do not use it in a production
deployment.
```

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. `systemctl` can show some recent activity, but you may want to follow the output of behaviors as they run. To do this, you will need to use the `journalctl` command. Use `-u` to specify the service we created, and then `-f` to follow the log, as illustrated in the following code snippet:

```
$ journalctl -u menu_server -f
```

   We will then be able to see servers as they run—perhaps not as convenient for debugging, but handy for launching services. Use *Ctrl* + *C* to stop seeing the log.

You can now reboot the robot, wait for the green light, and start driving it. The green light will also mean that your Mycroft voice assistant can send requests to the robot too.

If you upload new code, you will need to restart the service. You can use the following command to do so:

```
$ sudo systemctl restart menu_server
```

Congratulations—your robot is now truly headless! It doesn't even need a PC or laptop to start doing things.

Summary

This chapter added a small menu system to our robot to start different modes from a connected web browser.

You've seen how to drive a robot from a mobile phone and how to create interesting-looking animated widgets with SVG and JavaScript.

Your robot has now gained the ability to be driven manually. It may take you a while to get used to handling it, and manually correcting for veer (motors behaving slightly differently) is more challenging than when the PID systems correct themselves. Still, you will gain skills in driving it with your phone. You can use the camera on the front of the robot to get a robot's-eye view of the world.

You've turned the control server into a menu server and then made that start automatically when you turn on the robot. You've also seen how to connect your menu server to the video-server apps such as manual driving, color-tracking, or face-tracking apps. By making the buttons more touch-friendly on the menu server, you can use a phone to launch most behaviors.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, we gave the menu server a way to indicate being ready on the robot with a LED and then set it up to automatically start when you turn on the robot. If your robot and phone can connect to the same network (perhaps you can set up your phone hotspot in a `wpa_supplicant.conf` file), you will be able to launch the behaviors from places outside your lab and demonstrate them to people. You've made the robot fully controllable with your phone!

In the next chapter, we will look at meeting the robot-making community and finding further robot building and programming skills to continue building.

Exercises

You could enhance the system in many ways. Here are some suggestions for building further:

1. In the `manual_drive.py` file, the `handle_instruction` function uses a bunch of `if` statements to handle the instruction. If this list of command handlers exceeds five, you could improve it by using a dictionary (such as `menu_modes`) and then calling different handler methods.
2. Could you change the touch interface into two circular pads—perhaps so the left controls motor movement and the right changes the camera position?
3. What about creating phone-friendly interfaces for other behaviors to control their parameters?
4. You could embellish the CSS by adding round buttons or putting spacing between the buttons.
5. The menu still uses text buttons. Could you find a way to associate an image with each behavior and make a button grid?
6. Adding a **Shutdown** menu button will mean you could more gracefully shut down the Pi, where it would start the `sudo poweroff` command.
7. For desktop compatibility, the manual driving system could be enhanced with keyboard interactions to drive the robot, which is not quite as fun as the phone but is a handy fallback.
8. A seriously advanced improvement to the driving system would be to control motors in terms of counts per second, with a PID per wheel, matching the number of pulse counts we get with those we expect from the encoders. This improvement would make the robot drive straighter and be therefore easier to drive.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Further reading

To find out more about the topics covered in this chapter, here are some suggestions:

- I highly recommend the Flask API documentation (`http://flask.pocoo.org/docs/1.0/api/`), both to help understand the Flask functions we've used and to learn other ways to use this flexible web server library.
- For a more guided look at the Flask web server, I suggest reading *Flask By Example*, *Gareth Dwyer*, *Packt Publishing* (`https://www.packtpub.com/product/flask-by-example/9781785286933`), showing you how to build more involved web applications using Flask.
- I also recommend the book *Mastering Flask*, *Jack Stouffer*, *Packt Publishing* (`https://www.packtpub.com/web-development/mastering-flask`).
- The HTML used in this chapter is elementary. To get a more in-depth look into how you could enhance the simple menu system, I recommend the e-learning video guide *Beginning Responsive Web Development with HTML and CSS [eLearning]*, *Ben Frain*, *Cord Slatton-Valle*, *Joshua Miller*, *Packt Publishing* (`https://www.packtpub.com/web-development/beginning-responsive-web-development-html-and-css-elearning-video`).
- We use CSS selectors throughout HTML, CSS, and JavaScript applications. You can find a good combination of reference and tutorials at the *W3C Schools CSS Selectors* website (`https://www.w3schools.com/cssref/css_selectors.asp`). I would recommend exploring the site for its information on most web application technologies. For CSS units, see *W3C Schools CSS Units* (`https://www.w3schools.com/cssref/css_units.asp`) to practice and find more types of units to use. *W3C Schools* provides in general great reference and learning material for these web technologies.
- For getting more familiar with the JavaScript, CSS, and HTML technologies used here, *freeCodeCamp* (`https://www.freecodecamp.org/`) is a valuable resource with self-learning modules.
- Raspberry Pi has handy documentation on user `systemd` files at `https://www.raspberrypi.org/documentation/linux/usage/systemd.md`.
- There is a chapter on understanding `systemd` in *Mastering Linux Network Administration*, *Jay LaCroix*, *Packt Publishing* (`https://www.packtpub.com/product/mastering-linux-network-administration/9781784399597`), published in 2015.
- A full reference for `systemd` services can be found on the *freedesktop* manuals at `https://www.freedesktop.org/software/systemd/man/systemd.service.html`.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Section 4: Taking Robotics Further

In this section, we will learn how to find more interesting robot projects, continue growing the skills started in this book, and discover what communities there are. We will also summarize the skills we have learned.

This part of the book comprises the following chapters:

- *Chapter 18, Taking Your Robot Programming Skills Further*
- *Chapter 19, Planning Your Next Robot Project – Putting It All Together*

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

18
Taking Your Robot Programming Skills Further

You've now learned some beginner building skills and some of the more exciting programming tricks we can use with robotics. However, this robot is only really suitable for a lab; it's not ready for competitions or touring, and this is only the start of your robotics journey. There is also a large community of robot builders and makers that come from many angles.

In this chapter, you will learn how to continue your journey, how to find communities, how to look for new challenges, and where to learn more robotics skills. You will learn what skill areas there are beyond this book, and why they will help you make more robots.

How can you be part of this? Let's find out!

In this chapter, we will cover the following topics:

- Online robot building communities – forums and social media
- Meeting robot builders – competitions, makerspaces, and meetups

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Suggestions for further skills – 3D printing, soldering, PCB, and CNC
- Finding more information on computer vision
- Extending to machine learning

Online robot building communities – forums and social media

Robot building is a topic that shares a space with the general community of makers. Makers are everywhere. There are ham radio and electronics enthusiasts who are more connected to the electronics side of robot building, and there are artists who are using devices such as the Arduino and Raspberry Pi to bring their creations to life. Teachers are using these devices to show children the world of technology or teach other subjects to them. There are also people with problems to solve or brilliant and sometimes crazy ideas to try out.

Robotics is part of the maker community, which has a strong presence on Twitter, Instagram, and YouTube. Search for tags such as `#raspberrypi` (`https://twitter.com/hashtag/RaspberryPi`), `#arduino` (`https://twitter.com/hashtag/Arduino`), and `#makersgonnamake` (`https://twitter.com/hashtag/makersgonnamake`) to find these communities. A rallying point is the `@GuildOfMakers` (`https://twitter.com/guildofmakers`) account on Twitter. I talk about making robots on my account, `@Orionrobots` (`https://twitter.com/orionrobots`), from which I follow many robot communities and share what I have been making.

Another part of the robotics community is far more focused on the AI side of robotics, with specialist groups in visual processing, speech recognition and its various implementations, and more advanced topics such as neural networks, deep learning, and genetic algorithms. These communities may be close to universities and company research bodies. For speech processing, you can use the `#mycroft` (`https://twitter.com/hashtag/mycroft`) and `#voiceassistant` (`https://twitter.com/hashtag/voiceassistant`) Twitter tags. For visual processing, you can use the `#computervision` (`https://twitter.com/hashtag/computervision`) and `#opencv` (`https://twitter.com/hashtag/opencv`) tags to find relevant conversations and blogs. Searching for TensorFlow and machine learning will help.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finding Twitter feeds from universities involved, such as MIT Robotics (`https://twitter.com/MITRobotics`), CMU Robotics Institute (`https://twitter.com/cmu_robotics`), and *The Standford Vision and Learning Lab* at `http://svl.stanford.edu/`, will reveal some fantastic projects. Industrial robotics companies tend to be less helpful to makers but can be a source of inspiration.

Robot parts vendors online often have great projects, along with community influence. They also deliver internationally. In the UK, we have Pimoroni (`https://blog.pimoroni.com/`), 4Tronix (`http://4tronix.co.uk/blog/`), and Cool Components (`https://coolcomponents.co.uk/blogs/news`), to name a few. In the US, there is Adafruit (`https://blog.adafruit.com/`) and Sparkfun (`https://www.sparkfun.com/news`). Finding these vendors on social media will often reveal robotics and maker discussions with sources for parts and projects.

The online Instructables (`https://www.instructables.com/`) community shares many projects, including robotics builds and other things that will help a robot maker, either with experience or tooling. The Hackaday (`https://hackaday.com/`) website also has many great stories and tutorials.

Along with online websites, there are communities of robot builders on YouTube.

YouTube channels to get to know

First, there's my own: Orionrobots (`https://www.youtube.com/orionrobots`). I share many of my robot builds, experiments with sensors, and code on my channel. I put the code on GitHub with the intent that people can learn from and build on my ideas.

James Bruton (`https://www.youtube.com/user/jamesbruton`), also known as XRobots, makes very complicated and large 3D printed robotic builds and uses them to make creations that rival the great university robots, robotic costumes with real functionality, and self-balancing walkers.

The Ben Heck show (`https://www.youtube.com/playlist?list=PLwO8CTSLTkijtGC2zFzQVbFnbmLY3AkIa`) is less about robotics and more general making, including robotics. The show is far more focused on the maker side than the coding side but is an incredibly inspiring resource.

Computerphile (`https://www.youtube.com/user/Computerphile`) is a YouTube channel with great videos on programming, including aspects of robotics, visual processing, and artificial intelligence. It includes interviews with some of the significant figures still around in computing.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Tested channel (`https://www.youtube.com/user/testedcom`) features Adam Savage from the the Mythbusters team, with very skilled makers doing in-depth builds and sharing their work and techniques.

The vendors Makezine (`https://www.youtube.com/user/makemagazine`), Adafruit (`https://www.youtube.com/user/adafruit`), Sparkfun (`https://www.youtube.com/user/sparkfun`), and Pimoroni (`https://www.youtube.com/channel/UCuiDNTaTdPTGZZzHm0iriGQ`) have YouTube channels (and websites) that are very tutorial-based and can help you get to know what is available.

These YouTube communities are good if you want to see what people are working on and see robot builders at work. There are also specific places on the internet to ask for help.

Technical questions – where to get help

For technical questions, Stack Exchange can help, with specialist areas for Raspberry Pi (`https://raspberrypi.stackexchange.com/`), Electronics (`https://electronics.stackexchange.com/`), Robotics (`https://robotics.stackexchange.com/`), and Stack Overflow (`https://stackoverflow.com`) for general programming help. Quora (`https://hi.quora.com/`) offers another question-and-answer community for technical questions. Raspberry Pi has a forum at `https://www.raspberrypi.org/forums/`, while Mycroft has a community forum at `https://community.mycroft.ai/`.

OpenCV has a forum for technical questions following the Stack Overflow style at `http://answers.opencv.org/questions/`.

Twitter is a more open format where you can ask technical questions. To do so, make sure you use hashtags for the subject matter and perhaps tag some influential Twitter robotics people to help you.

Video channels on the subject are good places to ask questions; of course, watch the video to see if the answer is there first.

A trick to finding alternative tech and solutions on search engines is to type the first technology you think of, then vs. (as in versus), and see what completions they suggest. The suggestions will give you new options and ways to solve problems.

While talking to people on the internet can help with many problems, nothing beats meeting real robot builders and talking things over with them. Where are they and how can you find them?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Meeting robot builders – competitions, makerspaces, and meetups

As you start to build more, meeting up with other makers is a must. First, you will gain from the experience and knowledge in the community, but there is also a great social aspect to this. Some events are free, but the larger ones will have fees associated with them.

Makerspaces

These spaces are for any kind of maker, be it robotics, crafting, arts, or radio specialists. They serve as tool collectives with a collection of tools any maker may need, along with space to use them.

You can expect to find a collection of **Computer Numerical Control** (**CNC**) machines such as 3D printers, laser cutters, lathes, and mills for cutting materials and drills. They also usually have a full electronics bench and many kinds of hand tools in these spaces.

Some have the materials for making **Printed Circuit Boards** (**PCBs**). Makerspaces also have a community of people using these tools for their projects. People are there for the community and are happy to share their experiences and knowledge with anyone.

Makerspaces are a great place to learn about making and practice skills. Some, such as the Cambridge Makerspace (`https://twitter.com/cammakespace`), have robot clubs.

There are Makerspaces in many cities and towns around the world. They are also known as maker collectives, Hackerspaces, and fab labs. For example, for South West London, there is the London Hackspace (`https://london.hackspace.org.uk/`), Richmond Makerlabs (`https://richmondmakerlabs.uk/`), and South London Makerspace (`https://southlondonmakerspace.org/`). Another example is Mumbai with Makers Asylum (`https://www.makersasylum.com/`). There is a Directory of Makerspaces at `https://makerspaces.make.co`, although searching Google Maps for `makerspace` and `hackspace` near you will probably yield results.

These spaces make themselves easy to find on search engines and social media. If there are none in your area, reaching out via social media to other makers may help you find like-minded individuals who can help you organize spaces like this. When you find a venue, be clear on what the venue allows, as, for instance, soldering can be a problem until you find a dedicated space with a large enough collective.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Maker Faires, Raspberry Jams, and Dojos

In terms of Maker Faires (`https://makerfaire.com/`), many countries host festivals based on making. This is where people gather to show and build things together, with robotics often being a part of such festivals. These can be 1-day events or camping festivals such as EmfCamp (`https://www.emfcamp.org/`) in the UK. These are places where you can get started learning new skills, show and tell things you've made, and see what others have been making.

Raspberry Jams (`https://www.raspberrypi.org/jam/`) and Coder Dojos (`https://coderdojo.com/`) are groups that get together to regularly exercise their programming and, sometimes, maker skills. A Coder Dojo is a community programming workshop. A Raspberry Jam is a similar event, closely related to Raspberry Pi. Some Raspberry Jams are aimed at adults, while others are aimed at kids, so find out what groups there are locally, if any, and what they are aiming at. Becoming a mentor for kids at a Dojo or Jam is a great way to get to know other interested makers and programmers.

The annual Raspberry Pi parties are a fun get-together, but the focus is much more on meeting and less on building together.

All these groups tend to have quite inspiring Twitter feeds.

Competitions

Robotics competitions are still relatively rare outside of academia. The FIRST (`https://www.firstinspires.org/robotics/frc`) engineering initiative in the US is about getting schools and colleges to build robots and compete, with a few sporadic FIRST teams outside the US. FIRST challenges can be autonomous and manually driven. Most countries do have some kind of **Science Technology Engineering and Mathematics** (**STEM**) organization, such as `https://www.stem.org.uk/` in the UK. They sometimes host robotics competitions, which you will be able to find out about on their websites and newsletters; be sure to see if they are open to the general public or just schools.

In the UK, the PiWars (`https://piwars.org/`) competition is run annually and involves many autonomous and manual challenges set around the Cambridge University School of computing. It has a strong community element and is a great place to meet robot builders as a competitor or spectator. The `#piwars` (`https://twitter.com/hashtag/PiWars`) Twitter tag has quite an active community discussing this, particularly when robot makers gather to build and test robots before the event.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Another competition in the UK is Micromouse, `http://www.micromouseonline.com/`, which is about maze-solving robots, though other kinds of robots are exhibited by makers. Both competitions also have small robot markets.

The Robotex International (`http://robotex.international`) robotics exhibition is held in Estonia and combines lots of show and tell with days of competitions and serious prizes. They welcome robot builders working with electronics and Raspberry Pi, alongside Lego and other materials.

As these require travel, you should probably consider a large enough box, with bubble wrap or packing foam, to safely transport your robot(s) to and from such events.

I advise that you remove the batteries to reduce the possibility of a stray wire causing a short and pack them into a plastic bag to insulate them from any metal. Start considering cable routing in robot design; although this is outside the scope of this book, it makes robots far more robust.

I also recommend having a field repair kit at hand with a breadboard, wires, spare batteries, a charger, all the screwdriver types, replacement components for logic-level shifters, hook and loop tape, a standoff kit, and possibly a multimeter. Robots often need a little tuning and repair when arriving at an event.

In this section, you've learned about some of the places where you can meet robot makers, use tools, and find some competition. Next, let's look at more skills you can utilize to build your robot.

Suggestions for further skills – 3D printing, soldering, PCB, and CNC

As you build more robots, you will want to create more elaborate or customized systems.

To build a competition-grade robot, you will need more hardware building skills.

Design skills

We've used block diagrams and simple drawings throughout this book. However, to become more serious about robot building, you'll want to design parts or check that bought parts will integrate with your robot. You will want to create cases, chassis, sensor mounts, brackets, wheel types, and any number of parts, for which **Computer-Aided Design** (**CAD**) is key.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

2D design for illustration and diagrams

For 2D design and illustration, I recommend Inkscape (`https://inkscape.org/`). Inkscape is more artistic than CAD-oriented, but it is handy if you wish to make logos and other designs. It is quite complicated, so I recommend a book such as *Inkscape Beginner's Guide*, *Bethany Hiitola*, *Packt Publishing*, to get started learning about it.

Draw.io (`https://app.diagrams.net`) is useful for creating diagrams like the ones in this book. You can combine these two systems using Inkscape to make new shapes that you can use in Draw.io. Inkscape allows more freedom in terms of shape manipulation, but Draw.io is better for placing shapes and connecting things.

3D CAD

It is thoroughly worth getting to know 3D CAD systems such as FreeCAD (`https://www.freecadweb.org`) and Fusion 360 (`https://www.autodesk.com/campaigns/fusion-360-for-hobbyists`). FreeCAD is free and open source; Fusion 360 has a free entry-level CAD system for makers.

3D CAD systems let you design parts and then create further designs so that you can test how to assemble them. You can also make drawings from these for hand tool usage or export them for 3D printing.

All of them will take some investment in time, so I recommend using tutorials and YouTube videos to get to grips with them. The Maker's Muse channel (`https://www.youtube.com/channel/UCxQbYGpbdrh-b2ND-AfIybg`) is a good place to get started with this.

The Thingiverse (`https://www.thingiverse.com/`) community share 3D designs for printing and making. One very effective technique can be to either draw inspiration from, reuse, or repurpose creations seen there. If you can, import a bracket into FreeCAD and add the particular holes/base or connectors you need; it could save you hours of work trying to draw the mount for a sensor from scratch. The community will also have tips on printing these. As you may not always find what you are looking for in Thingiverse, consider alternatives such as Pinshape (`https://pinshape.com/`) and GrabCad (`https://grabcad.com/`).

Once you have CAD drawings of parts, you can send them off to have them made or learn about techniques you can use to manufacture them yourself.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Skills for shaping and building

As a general recommendation, the MIT *How To Make Almost Anything* (`http://fab.cba.mit.edu/classes/863.14/`) course materials (which are updated annually) are a fantastic resource for finding ways to put things together – although they look plain, the links there are very useful. As we mentioned in *Online robot building communities – forums and social media*, YouTube and other channels are rich with practical examples and hands-on tutorials when it comes to making things.

Machine skills and tools

CNC milling, laser cutting, and 3D printing allow you to create solid parts and can give great results; however, each is a field of its own with many skills you must learn on the way. Laser cutting allows you to make flat parts, but with some ingenuity, flat parts can be assembled (like so many types of furniture) into sophisticated, solid 3D objects.

The YouTube channel NYC CNC (`https://www.youtube.com/user/saunixcomp`) covers a lot of CNC tips and usage; however, the online book *Guerrilla guide to CNC machining, mold making, and resin casting*, by *Michal Zalewski* is also a brilliant resource.

For these machining techniques, I would not suggest going out and buying your own, but to find out more about the local community Makerspaces we mentioned previously and use the facilities they have there. Some libraries are also getting into this and providing 3D printers and simple maker materials. Using these will be cheaper than buying your own; you will be among a community of others with experience, and it will be far easier than trying to do it alone.

If you just want the 3D printed or laser cut parts, there are places online that will make things for you. Ponoko (`https://www.ponoko.com/`), RazorLAB (`https://razorlab.online`), 3DIng (`https://www.3ding.in/`), Protolabs (`https://www.protolabs.co.uk/`), Shapeways (`https://www.shapeways.com/`), and 3D Hubs (`https://www.3dhubs.com/`) are some of the companies that offer such services. Looking for 3D printing and laser cutting services in your region via a search engine isn't difficult, but it will still help to gain some experience through a Makerspace to understand what is and isn't possible with these machines. Using the wrong machine for a job, or making the wrong design decisions, could lead to huge costs.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

3D printers, laser cutters, and CNC machines require routine maintenance and upkeep tasks, such as leveling a 3D print bed or tramming the CNC chuck. They also require consumables such as stock (plastic filament, wood to mill or laser cut), replacement components, and bed adhesive materials. Unless you are printing a lot, it is rarely an economy to own your own when you have access to another via a Makerspace or an online market.

While machine skills will create very precise parts, hand skills are needed either to finish or modify these parts. Some parts will always be more suitable if they're made by hand for robot one-offs.

Hand skills and tools

Having some basic woodworking and crafting skills always comes in handy. Practicing these at a Makerspace will help you see how things can go together. With this comes knowing how to choose suitable wood as an unsuitable wood might be too soft, too heavy, or too irregular. Wood can be carved by hand or used in a CNC machine, as mentioned previously.

Learning modeling skills, such as using plasticard (styrene sheets), creating molds, and casting, are other ways to make 3D parts. Plasticard is an inexpensive, flexible material of varying thickness that can be easily cut by hand, perhaps using a printed template, and then assembled.

You can use woodworking to create molds and makeshift robot chassis. Molds allow you to make multiple copies or use materials in high-quality parts. Casting can be tricky, especially if you're dealing with bubbles, but there are good books on this subject. For this, I recommend the articles `https://medium.com/jaycon-systems/the-complete-guide-to-diy-molding-resin-casting-4921301873ad`, `https://youtu.be/BwLGK-uqQ90`, and the *Guerilla Guide To CNC*, which was mentioned in *Machine skills and tools*.

Further interesting material skills, such as working with metal, allow for even bigger robots. This means learning how to cut, shape, and weld metal parts.

Carbon fiber or Kevlar materials are useful in larger robots, fighting robots, or those needing to handle heavier materials.

The Instructables (`https://www.instructables.com/`) and Hackaday (`https://hackaday.com`) communities will help you learn skills like those mentioned in this section. They have practical instructions and tutorials on building things. You can either follow along with complete projects or just skim read for techniques to borrow from them. As well as looking for robots, look at modeling techniques (often similar), plasticard builds, woodwork, or metalwork tutorials. Makerspaces run lessons on these skills too.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

With a pointer to where you can learn how to make the structural and mechanical parts, what about electrical parts?

Electronics skills

The next thing you must do is extend your electronic skills. We have been using Raspberry Pi hats and modules to build our robots. This is fine when we're starting out, but this starts to feel clumsy when there are many parts, with demands on space or fragile wiring making it far from ideal. You'll note that our wiring on the robot is very crowded.

Electronics principles

Learning more about the electronic components and common circuits' functions will help you understand your robot further, expand it, find ways to reduce its size, or eliminate problems on the robot.

Power electronics will give you a better understanding of your robot's motor controller and battery regulation circuits. Digital electronics will let you connect other logic devices, use new sensors, or aggregate them in useful ways. Analog electronics will also open up new types of sensors and actuators and give you tools to diagnose many electrical problems that can crop up.

For this, you should learn how to draw and read schematic circuits for the common parts. Online courses and YouTube channels teach electronics step by step, with books such as *Make: Electronics*, by *Charles Platt* giving a very hands-on learning path.

The *EEVBlog* (`https://www.eevblog.com/episodes/`) channel is less step by step but offers more general immersion in terms of electronic engineering concerns.

Taking soldering further

Although we've done a little soldering, it's just the bare minimum. There is far more stuff to learn about on the subject. Soldering is a skill that many makers use daily.

Some good places to start are the Raspberry Pi guide to soldering (`https://www.raspberrypi.org/blog/getting-started-soldering/`), *The Adafruit Guide To Excellent Soldering* (`https://learn.adafruit.com/adafruit-guide-excellent-soldering`), and the *EEVBlog Soldering* tutorial (`https://www.youtube.com/watch?v=J5Sb21qbpEQ`).

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

I recommend starting in a local Makerspace, where you will benefit from others and complete simple soldering projects. Soldering headers onto a module is a pretty basic way to start, along with using kits such as those made by Boltportclub (`https://boldport.com`) to stretch those skills a bit further. Soldering allows you to start thinking about creating boards or Raspberry Pi hats.

You will start off by soldering simple headers and what are known as **through-hole** components since they go through a hole in the board. This is the right type of construction you should be implementing to gain confidence with the technique.

As you become more confident, you will find kits that use surface mount soldering. Surface mount components do not have legs that go through holes but simple metal pads that are soldered directly onto the copper pads on the board. They take up far less space, allowing for smaller constructions, but they are also quite fiddly and eventually require fairly professional tools to be used. Some simple surface mount components, such as LEDs, resistors, and capacitors, can be soldered by hand. See the *EEVBlog Surface Mount* tutorial (`https://www.youtube.com/watch?v=b9FC9fAlfQE`) for a starting point.

Devices with tens of pins may not work and require solder ovens and solder paste. At that point, you may be making custom circuits, and a **Printed Circuit Board and Assembly** (**PCBA**) service might be the correct path to take.

Custom circuits

As you gain confidence with electronics and soldering, you will want to create more circuits and transfer them onto more professional-looking PCBs to save space and perhaps make them easier to wire. Breadboards are good for learning and experimenting, but they are not ideal for competing and quickly become bulky and untidy, while point-to-point wiring is fragile and prone to mistakes.

The first stage of custom, more permanent circuits is using stripboard or perfboard and soldering components onto them. This is a good further step from breadboards and will save space. They can still be a little bulky and messy, though. You may also want to use parts that are surface mounted or have irregularly laid out legs of different sizes that don't fit conveniently on perfboard or stripboard.

To take your circuits to the next level, you can learn to design PCBs. You will be able to save yet more space, have more robust circuits, and be able to use tiny surface mount parts. You could even design PCBs that are for light structural placement too.

For breadboards, you can use Fritzing (`http://fritzing.org/home/`), but I don't recommend it for schematic or PCB work. To design these, software such as KiCad (`https://kicad.org`) is a good hobbyist tool. I recommend the video course *KiCad Like a Pro*, *Peter Dalmaris*, *Packt Publishing*.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can use facilities at local Makerspaces to make PCBs or send them to board houses to have them beautifully made, with fine tracks, lettering, and fancy colored solder masks (you'll see more such terminology in the field). Custom PCBs allow you to tune the layout to avoid any point-to-point wiring, work with tiny surface mount parts, add helpful text right on the board for some wiring, and get a professional look. Some even use this to make other parts for the robot, including structural parts and front panels, in PCB.

Finding more information on computer vision

We started looking at computer vision in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*. We used OpenCV to track colored objects and faces but barely scratched the surface of computer vision.

Books

I recommend the book *OpenCV with Python By Example*, *Prateek Joshi*, *Packt Publishing*, if you wish to continue learning about OpenCV. This book uses computer vision to build augmented reality tools and to identify and track objects and takes you through different image transformations and checks, showing screenshots for each of them. It is also quite fun as it contains lots of hands-on code.

You can even extend computer vision further to 3D computer vision with the Xbox 360 Kinect sensor bar. Although they are no longer produced by Microsoft, they are extremely common and fairly cheap on eBay. Note that there is a modern Azure Connect device you can use for this, but at the time of writing, this is 20 times the price! The XBox 360 Kinect sensor bar has also been interfaced with the Raspberry Pi. The Kinect has a 3D sensing system that makes them valuable for use in robots. There is a section on connecting this to the Raspberry Pi in the book *Raspberry Pi Robotic Projects*, *Dr. Richard Grimmett*, *Packt Publishing*.

Online courses

PyImageSearch (`https://www.pyimagesearch.com/`) contains some of the best resources for learning OpenCV and experimenting with machine vision.

Learn Computer Vision with Python and OpenCV, *Kathiravan Natarajan*, *Packt Publishing* (`https://www.packtpub.com/in/application-development/learn-computer-vision-python-and-opencv-video`) dives in some depth into color tracking, feature detection and video analysis while using the excellent Jupyter tool to experiment with image transformations.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The TensorFlow Tutorials (`https://www.tensorflow.org/tutorials/`) website (a machine learning framework) contains tutorials specifically aimed at using TensorFlow in computer vision. Training machine learning systems to perform visual recognition can take a lot of time and sample data.

The video course *Advanced Computer Vision Projects*, *Matthew Rever*, *Packt Publishing* (`https://www.packtpub.com/big-data-and-business-intelligence/advanced-computer-vision-projects-video`) provides further computer vision projects, culminating in using the TensorFlow machine learning system to analyze human poses from camera input.

Social media

I mentioned the Twitter tags `#computervision` and `#opencv` in the *Online robot building communities – forums and social media* section, and they are a good place to ask questions or share your work about the subject.

Computerphile has a small computer vision playlist (`https://www.youtube.com/watch?v=C_zFhWdM4ic&list=PLzH6n4zXuckoRdljSlM2k35BufTYXNNeF`) that explains the concepts and theory of some visual processing algorithms, but does not tend to dive into any hands-on implementation.

With that, you've learned where you can find out more about computer vision. However, this leads to a more advanced robot subject: machine learning.

Extending to machine learning

Some of the smartest sounding types of robotics are those involved in machine learning. The code used throughout this book has not used machine learning and is instead used well-known algorithms. The **Proportional Integral Derivative** (**PID**) controller you used in this book is a system that makes adjustments to read a value, but it is not machine learning. However, optimizing PID values might come from a machine learning algorithm. We used Haar Cascade models to detect faces; this was also not machine learning, though OpenCV contributors probably used a machine learning system to generate these cascades.

Machine learning tends to be great at optimizing tasks and discovering and matching patterns, but poor at making fully formed intelligent-seeming behavior.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The basic overall idea of many machine learning systems involves having a set of starting examples, with some information on which are matches and which are not. The machine is expected to determine or learn rules based on what is or is not a match. These rules may be fitness scores based on learning rules to maximize such a score. This aspect is known as training the system.

For the PID control system, you could base fitness on settling to the set point in the fewest steps with little or no overshoot based on training values from data, such as machine variations, response times, and speed.

Once again, I recommend the Computerphile AI Video playlist (`https://www.youtube.com/watch?v=tlS5Y2vm02c&list=PLzH6n4zXuckquVnQ0KlMDxyT5YE-sA8Ps`) video series for getting to know the concepts around machine learning; it's not hands-on but is more focused on the ideas.

Machine learning can be quite focused on data and statistics, but the techniques you've learned throughout this book can be applied to sensor data to make this more relevant to robotics. There are many examples of the TensorFlow system being used to build object recognition systems. Genetic algorithm-evolving solutions have been used to great effect for robot gaits in multi-legged systems or finding fast ways to navigate a space.

Robot Operating System

Some of the robotics community make use of the **Robot Operating System** (**ROS**); see `http://www.ros.org/`. Robot builders use this to build common, cross-programming language abstractions between robot hardware and behaviors. It's intended to encourage common reusable code layers for robot builders. AI systems built on top of this can be mixed and matched with lower-level systems. The behaviors/robot layers we have built allow some reuse but are very simplified compared to ROS.

The book *ROS Programming: Building Powerful Robots*, *Anil Mahtani*, *Packt Publishing* covers linking the TensorFlow AI system to ROS-based robotics.

For a simpler introduction, *Learning Robotics using Python*, *Lentin Joseph*, *Packt Publishing* uses ROS with Python to build a smart AI robot using LIDAR.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Summary

In this chapter, you learned how to find out who else and where else robots like the ones we covered in this chapter are being made, as well as how to be part of those communities. Sharing knowledge with other robot builders will accelerate your journey.

You've also learned where to compete with a robot, where to get more advice, and how to find information to progress the different skills you've started building much further. This inspiration and direction should make it easy for you to keep growing your robot skills.

In the next chapter, we will summarize everything that we have learned throughout this book, with a view toward building your next robot.

Further reading

The following are further practical robotics books available that I enjoy:

- *Python Robotics Projects*, *Prof. Diwakar Vaish*, *Packt Publishing*: This book offers more Raspberry Pi and Python robotics projects for you to practice with.
- *Robot Building for Beginners*, *David Cook*, *Apress*: This book leads you through building *sandwich*, a scratch-built robot based on a lunchbox. It is a little more maker- and electronics-based, but it is quite a fun project to follow.
- *Learning Raspberry Pi*, *Samarth Shah*, *Packt Publishing*: You can dig further into what can be done with a Raspberry Pi here and find inspiration for enhancing your robots within the sections of this book.
- *Robot Builder's Bonanza (5th Edition)*, *Gordon McComb*, *McGraw-Hill Education TAB*: This was an influential book and is quite extensive in terms of how to make a robot. This is the best book for going beyond buying kits and constructing bigger and more mechanically complicated robots.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

19

Planning Your Next Robot Project – Putting It All Together

Throughout this book, you've now seen how to plan, design, build, and program a robot. We've covered many of the starting topics with some hands-on experience of them, an example demonstrating the basics, and some ideas of how you could improve them. In this chapter, we will think about your next robot. We'll answer questions such as the following: How would you plan and design it? Which skills might you need to research and experiment with? What would you build?

We will cover the following topics in this chapter:

- Visualizing your next robot—how will it look?
- Making a block diagram—identify the inputs/outputs and parts it would need
- Choosing the parts—which tradeoffs will you think about when selecting components for the robot?

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Planning the code for the robot—which software layers and components might this robot need, and which behaviors would be fun?
- Letting the world know—how would you share your plans, or your robot, with interested people?

Technical requirements

For this chapter, I recommend having some diagramming tools such as the following:

- Pen/pencils
- Paper—a sketchbook (or perhaps graph paper) is great, but the back of an envelope will do
- A computer with internet and access to `https://app.diagrams.net/`

Visualizing your next robot

When we started this book, in *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*, we first learned how to look at robots as a sketch. I suggested that you make quick drawings, and not worry if they are rough and sketchy—this is perfect at an early planning stage. Use a pencil or pen, and then move on to the more formal block and layout diagrams later.

Every robot starts with a bit of inspiration. Perhaps there is a competition you want to try; maybe you've seen something such as another robot or an animal you want to mimic (crabs are fascinating!). Other inspirations may come from seeing a unique new part or wanting to learn/play with a new skill. You may even have made a list of amazing robots you want to try to build.

Before building a robot, make a short bullet-point list of what it will do, which sensors/outputs it will have, and what it might have to deal with. This list lets you focus your efforts. Here is an example, which I made for my SpiderBot project seen in *Chapter 10*, *Using Python to Control Servo Motors*. This is what I planned for it to do:

- It will have six legs (yes—an insect, not a spider).
- I will use it to experiment with legs and gaits.
- It will be able to avoid walls.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Your quick sketches could first be a basic six-legged stick drawing, with some squares at one end to represent the ultrasonic sensor, and perhaps a few arrows with notes to depict what they mean. You've seen this technique in detail in *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*. The following photo shows a simple design:

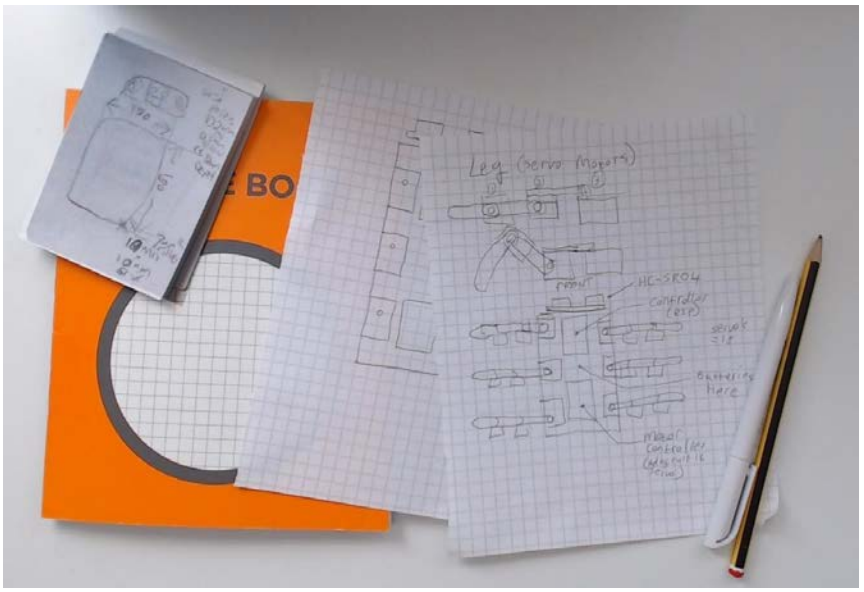


Figure 19.1 – Sketching your ideas on paper

As *Figure 19.1* shows, my preferred first sketches are with a biro on graph paper, but I'll use any paper I have.

Visualizing the robot can be made with 2D, 3D, or profile sketches. Here are a few tips:

- Draw lightly, then follow through with firmer strokes when you are more confident with the design.
- Annotate it a lot with anything that comes to mind.
- Don't worry about scale, dimensioning, or perfect drawing; this is simply to capture ideas. You'll flesh them out and make firmer decisions later.
- It can be a good idea to date the picture and put a working name on it, even if you have a better name later.
- Feel free to combine block-style representations with sketchy visual versions.
- Keep a biro/pencil and notepad/scrap paper with you somewhere so that you can quickly jot down ideas. A whiteboard is excellent if you are near one. A pencil can let you erase and rewrite, and a ballpoint pen is easy to keep in a bag or pocket.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Get the big ideas down first; come back for detail. It's easy to get bogged down in detail on one aspect, forget the other parts, and run out of time. You can always make a tiny note to remind yourself.

You can revisit this process at any time during the robot build, perhaps when you have further ideas, when you have a problem you are solving, or when you want to refine it. Most ideas start with some bullet points and a scribbled sketch; waiting for access to a computer or trying to draw it perfectly will detract from the next fantastic idea you already have in your mind—get it down first.

Now that you have a sketch of roughly what it will look like and have written a basic plan on paper, we can start to formalize it with block diagrams.

Making a block diagram

Recall how in *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*, and throughout the book, we've created block diagrams showing the robot we built there. You can represent any robot in this way. This diagram is where you would have a block for each input and output and then create controller and interface blocks to connect with them. Don't worry about the diagram being perfect; the main point is that the picture conveys which parts you'll connect to others. It's also quite likely that the initial diagram will need some change as you build a robot and come across constraints you were not aware of.

Here are two stages of a block diagram for SpiderBot:

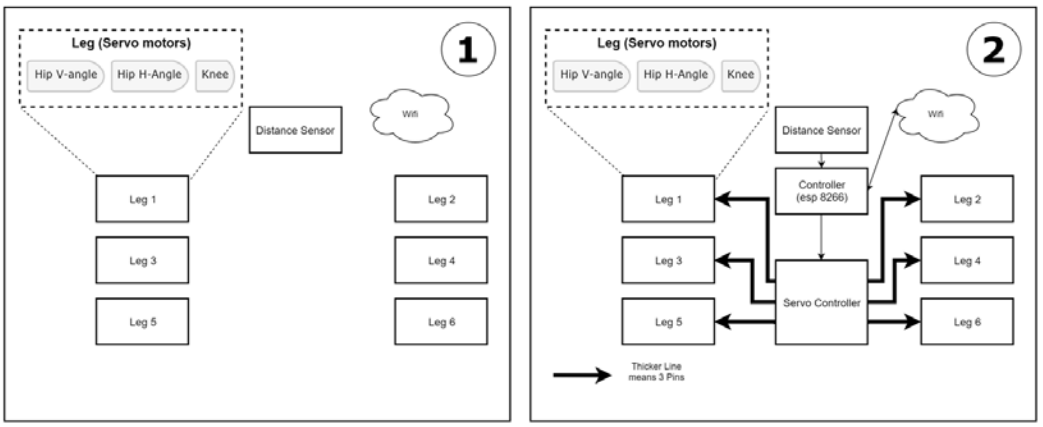


Figure 19.2 – SpiderBot block diagram stages

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

In *Figure 19.2*, I initially knew going in that each leg had three motors, but not a lot else. So, I drew those blocks in, along with the distance sensor I want it to have and a Wi-Fi connection.

In the next stage of the diagram, I added the controllers I'll use for it and then made the rough connections on the diagram. These are just blocks, and this is not a circuit diagram. These have been thrown together with `app.diagrams.net`. Note that things could still change as you learn more about your robot and its controllers.

You may also consider making diagrams for add-ons and subcomponents. Any part that seems complicated may need a further diagram to explore it. The important thing is to get ideas such as this out of your head and onto paper so that they are clearer, so that you don't forget them, and so that you may be able to spot flaws.

The other block diagram to consider is a software diagram, which we will visit in the *Planning the code for the robot* section.

Now, you have a rough sketch of the robot and the block diagram, and you are ready to start choosing the parts you would use to build a robot.

Choosing the parts

Throughout this book, we have looked at the tradeoffs between different kinds of sensors, different chassis kits, controllers, and so on. These are tradeoffs on weight, complexity, availability (you don't want an irreplaceable part), and cost, covered in detail in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*.

If a particular kit inspired the robot—for example, SpiderBot was inspired by a hexapod kit; yours could be a robot arm or caterpillar track kit—this will likely constrain the other part choices you need to make. I'd need to support 18 servo motors; however, I had a 16-motor controller available, so I elected to use two **input/output** (**I/O**) pins of the central controller for the remaining two servos. This added software complexity, though.

Another tradeoff was the main controller. I knew that I'd want SpiderBot to be Wi-Fi-enabled, but it wasn't going to be doing visual processing, so a small, cheap, and low-power controller such as the ESP8266 was an excellent choice for it.

For power, I knew that it would require a lot of current for all those servos but it wouldn't be able to carry a great deal of weight, so a more specialist **lithium polymer** (**LiPo**) battery would be needed, along with a charger/protection circuit.

A key stage of choosing the parts is to test fit them.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

The test-fit diagram

When choosing the parts, consider how they will fit together: is there a clear path to interfacing the choice of motor controller with your choice of main controller? Have these two components been used together, or are you prepared for the complexity of making a new interface? Based on the parts you think you will buy, collect their dimensions and try making a test-fit diagram, as we did in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*. Try to make a test fit before buying new parts.

Buying parts

It was then a matter of finding stockists to buy it. I do have some local favorites (such as `coolcomponents.co.uk`, `shop.pimoroni.com`, and `thepihut.com`), and you will find those in your region as you build more. Looking for local Pimoroni, SparkFun, Raspberry Pi, and Adafruit stockists will help you find the right kind of store.

You can find modules on your regional Amazon, Alibaba, or eBay but be very clear about what you are buying and how much support you will get. You'll find individual parts at large stockists such as Element14, Mouser, RS, and Digi-Key; although they tend not to have many prebuilt modules, they are reliable and have large catalogs.

Parts are mostly sold online. There may be high-street sellers of electronics and mechanical parts, but this is becoming rarer.

You may also use parts from an existing stock, which you will build up as you build robots. You can convert toys into robot chassis, a practice known as toy hacking. Robot builders can salvage motors and sensors from old printers and electromechanical systems (with care). In this case, the test-fit diagram will help you see what you may need to change to make things work with the salvaged parts.

Assembling your robot

Now, you are ready to assemble your new robot. The building guides in *Chapter 6*, *Building Robot Basics – Wheels, Power, and Wiring*, and *Chapter 7*, *Drive and Turn – Moving Motors with Python*, along with the basic soldering guide in *Chapter 12*, *IMU Programming with Python*, will get you started. However, the additional reading and skills suggested in *Chapter 18*, *Taking Your Robot Programming Skills Further*, will give you many more options for assembling the robot.

Now that you have the parts and you've started building the robot, the next thing to consider is the robot's code.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Planning the code for the robot

We started planning code in layers in *Chapter 2*, *Exploring Robot Building Blocks – Code and Electronics*, and then explored this further in *Chapter 7*, *Drive and Turn – Moving Motors with Python* under the *Robot Object* section.

Let's recall how we planned our code structure with layers.

System layers

The general idea is to create layers of code in the system, as shown in the following diagram:

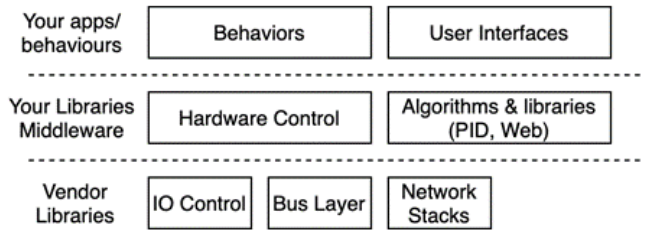


Figure 19.3 – Robot software layers

As *Figure 19.3* shows, there are some suggested layers, as follows:

- At the bottom of the stack, **Vendor Libraries**. These, as the name implies, usually come from the hardware supplier, third parties, or a community. These are things such as the `gpiozero` library we have been using or the various Arduino libraries for that ecosystem. This layer may include I/O control, bus layers, and network stacks.
- The next layer is **Libraries and Middleware**. This software may come from a third party—for example, higher-level libraries to interface with specific hardware. The middleware layer also includes abstractions you've written, such as making two-wheeled robots behave the same, even if the third-party libraries differ. On the same layer are algorithms and libraries. OpenCV exists at this layer, from a community. Your **Proportional-Integral-Derivative** (**PID**) algorithm or object recognition pipelines may belong in this layer. This layer has components to build apps and behaviors.
- The top layer is gluing everything together. Take hardware sensor inputs, through algorithms, to create hardware motor outputs and pleasing control interfaces or dashboards. The top layer is where you make behaviors and apps for the robot.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

For a basic robot, components in these layers could just be functions or classes. For a more complicated one, these may be different software components talking on a shared software bus (such as a message queue or as connected services). The library we have already built will work for many small-wheeled robots (at the middle layer). This library will need refining as you gain experience with it. You can adapt behaviors for new sensors and outputs if you have kept the behavior separate from the hardware concerns.

Use diagrams to draw the blocks and layers to express where those boundaries lie. Expect to write code in modules and blocks that you link together to reason about each part. It should not be necessary to get lost in the details of a **Serial Peripheral Interface** (**SPI**) data bus transaction (at the vendor hardware layer) when thinking about making pleasing LED patterns (at the behavior layer).

Once you've considered layers and some rough components, you will need to think about how information moves between them with a data-flow diagram.

Data-flow diagrams

You can also use diagrams to explore the behavior from a data-flow perspective, such as the PID and feedback diagrams used to express the behaviors in *Chapter 13*, *Robot Vision – Using a Pi Camera and OpenCV*. We also use this diagram style for the color object and face-tracking behaviors as data pipelines showing the image transformations. Don't expect to capture the whole story in one diagram; sometimes, a few are needed to approach the behavior's different aspects.

Spend time considering the tricky areas here, such as additional math that might be needed if the sensor/movement relationship is complicated. You might not get it right the first time, so building it and reasoning about why it behaved differently from your expectations will be necessary. At this stage of the design, going and finding similar works on the internet or reading one of the many recommended books will yield a deeper understanding of what you are attempting. In most cases, persistence will pay off.

Formal diagrams

There are formal representations for diagrams such as flowcharts or the **Unified Modeling Language** (**UML**) types. These are worth finding out about and learning as a resource to draw upon for drawing. The `app.diagrams.net` software has a nice library of diagram elements. The most important aspect of a diagram is to convey the information—you should try to express what is in your head in a way that makes sense to you 6 months later, or to your team (if you have one).

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Sometimes, building simple behaviors gives you a library to use for more complicated and interesting ones. Our straight-line drive behavior was a building block to start on driving in a square behavior.

Programming the robot

You can now program the robot, but be prepared to go around a few planning, implementing, testing, and learning loops. Do not be disheartened by testing failures, as these are the best opportunities to learn. The most learning is done in planning and determining test failures. If it all works the first time, it's far less likely to stick with you. Each behavior in this book took me multiple attempts to get right; tuning them is a trial-and-error process.

You've planned your code, and you're now building the robot. How do you let the world know you are making something and find help?

Letting the world know

You are bound to have questions about how to proceed and problems to be solved—perhaps you've already encountered them before building. When you have questions or have made some minor progress, it is the right time to get online and link with the robotics communities, as shown in the *Online robot building communities* section of *Chapter 18*, *Taking Your Robot Programming Skills Further*.

Use Twitter and Stack Overflow to ask questions or even answer questions from other robot builders. Use YouTube to share your creation or the story of your build and see other people's builds. You do not need to wait until you have a perfectly polished product. Share the steps you've taken, the frustrations you have encountered, and even the failures you've learned from. Failure situations make for some of the best stories. These stories can be just the right motivation for someone else to keep on persisting with complicated builds.

Use a combination of YouTube, Instructables, and online blogs to practice new skills, or—better yet—get to a nearby Makerspace, Coder Dojo, or Raspberry Jam to practice new skills with others who are also making and learning.

Being a robot builder will make you an eternal student; there is always more to learn on the subject, not least because it is still an area of much research. People are pushing the boundaries of human knowledge in robotics. You can push your skill and knowledge boundaries while becoming a mentor and helper to extend the boundaries of what others can do. Perhaps you will come up with kits, modules, and code to lower barriers to entry, and you may also find novel ways to use robotics or build a sensor that pushes the boundaries of human knowledge. Whichever way it is, engaging with the robot community is exciting, refreshing, and keeps you looking out for new stuff to try.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Testing your robot in lab conditions is OK, but the most rigorous testing happens outside, at competitions and demonstrations. You will shake out new bugs, find new problems to solve in these cases, and create a network of robot-building friends and peers. Robotics has a stereotype of being a very solitary hobby or profession, but this need not be the case as there are plenty of people making something, so make with them.

Building with a team can be very rewarding and challenging. It will allow you to create more ambitious builds than going it alone. Getting involved in any of the communities, especially local ones, will probably represent your best chance of finding team members.

You've now reviewed how to reach out to the wider world of robot builders and find help, inspiration, and competition. You've seen how robotics can be a collaborative and social hobby. Perhaps you've also been inspired to start your own robotics blog or YouTube channel. I look forward to seeing you in our community!

Summary

You've now seen throughout this book how to build and program your first robot. You've seen where to find out more and how to extend your knowledge. In this final chapter, we've summarized what you've learned and suggested how to use this to plan, build, and program your next robot, as well as taking it on tour and being a member of the robotics community.

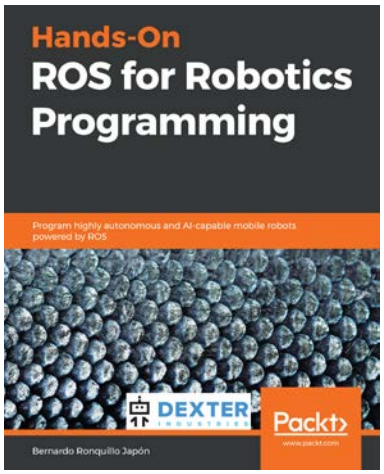
You've seen how to design and plan a robot and how to build, program, and test a robot. You've learned hardware skills such as soldering, simple software such as moving a robot, and have touched lightly on complex areas such as computer vision and inertial measurements. You've shaken out bugs, made tradeoffs, finely tuned systems, and learned how to keep backups. You've made user interfaces, smart behaviors, and taken control of a robot with a smartphone.

You have reached the end of this book, but I hope this is just the start of your robotics journey.

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



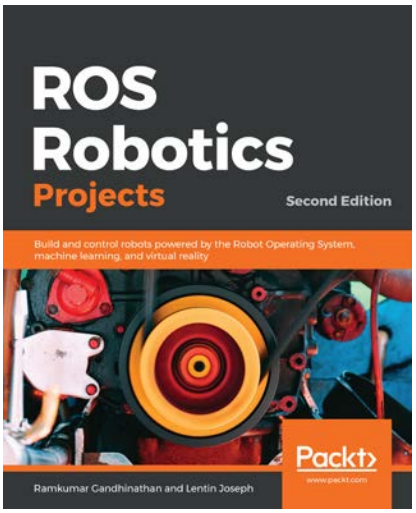
Hands-On ROS for Robotics Programming

Bernardo Ronquillo Japón

ISBN: 978-1-83855-130-8

- Get to grips with developing environment-aware robots
- Gain insights into how your robots will react in physical environments
- Break down a desired behavior into a chain of robot actions
- Relate data from sensors with context to produce adaptive responses
- Apply reinforcement learning to allow your robot to learn by trial and error
- Implement deep learning to enable your robot to recognize its surroundings

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use



ROS Robotics Projects - Second Edition

Ramkumar Gandhinathan, Lentin Joseph

ISBN: 978-1-83864-932-6

- Grasp the basics of ROS and understand ROS applications
- Uncover how ROS-2 is different from ROS-1
- Handle complex robot tasks using state machines
- Communicate with multiple robots and collaborate to build apps with them
- Explore ROS capabilities with the latest embedded boards such as Tinker Board S and Jetson Nano
- Discover how machine learning and deep learning techniques are used with ROS
- Build a self-driving car powered by ROS
- Teleoperate your robot using Leap Motion and a VR headset

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Index

Symbols

4Tronix
  URL 535

A

absolute encoders 244, 245
accelerometer
  about 310
  adding, to interface 311
  displaying, as vector 311, 312
  pitch-and-roll angles, detecting with 454
  reading, in Python 310
  smoothing 458
accelerometer and gyroscope data, fusing
  about 459-462
  troubleshooting 462
accelerometer vector
  pitch-and-roll angles, obtaining
    from 454-457
Adafruit
  URL 535, 536
addressable RGB LEDs 183
advanced and impressive robots
  exploring 6, 7
  Mars rover robots 8
Analog Out (AO) 252
analog to digital converter (ADC) 287
Application Programming
    Interface (API) 421, 485
Arduino 42
Arduino Leonardo 32
Arduino tacho 242
Armbot 16
automated guided vehicles (AGVs) 377
avoid behavior
  basic LEDs, adding to 199-201
  debugging, with light strip 199

B

background tasks, running
    while streaming
  about 340, 341
  behavior controllable, making 344, 345
  controllable image server, running 346
  control template, making 345, 346
  web app core, writing 341-344
basic LEDs
  adding, to avoid behavior 199-201
battery eliminator circuit (BEC) 100
Baxter 12
beacons 378

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

BeagleBone 43
behavior components
  behavior code, writing 353-359
  control template, writing 352
  writing 352
Body Coordinate System 305
Bonjour
  download link 57
  setting up, for Microsoft Windows 56
brushless motor 25

C

calibration 448
Camera Serial Interface (CSI) 44
Cascading Style Sheets (CSS) 501
CHIP 43
CMU Robotics Institute
  URL 535
code
  loss of code 72
code, for testing motors
  libraries, preparing 127
  working 131, 132
  writing 126
code, planning for robot
  about 555
  data-flow diagrams 556
  formal diagrams 556
  system layers 555
Coder Dojos
  URL 538
colored objects, following with Python
  about 347, 348
  behavior components, writing 352
  behavior, running 359, 360
  picture, turning into
    information 348-350
  PID controller, enhancing 351, 352
  PID controller settings, tuning 361
  troubleshooting 361
color systems 194
competitive robot 16
Computer-Aided Design (CAD) 539
Computer Numerical Control (CNC) 537
Computerphile
  URL 535, 546
computer vision pipeline
  about 381
  testing, with test images 384
computer vision pipeline, methods
  camera line-tracking
    algorithms 381, 382
  pixels, visualizing 382-384
computer vision, resources
  about 545
  books 545
  online courses 545, 546
  social media 546
computer vision software, setup
  about 330
  Open Computer Vision (OpenCV)
    library, installing 331
  Pi Camera software, setting up 330
  pictures, obtaining from
    Raspberry Pi camera 331
  Python OpenCV libraries, installing 331
connections, robot
  data buses 45
  general IO 46
  power pins 45
  selecting 44, 45
controller
  design 498-500
  overview 498-500
  selecting 497

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

controller, robot
  about 31
  Arduino Leonardo 32
  micro:bit 32
  NodeMCU 31
  Raspberry Pi 31
  types 31
Cool Components
  URL 535
Coriolis force 304

D

data buses 45
DC gear motor 24
DC motor 24
DC motors
  controlling 215, 216
dd command
  using 84
Debian 46
degrees-of-freedom (DOF) 287
delta time 451-459
design skills
  about 539
  2D design, for illustration
    and diagrams 540
  3D CAD 540
dialogs 410
Direct Current (DC) 24
Disk Utility tool
  using 81-84
distance sensors
  about 36
  pulse timing, using 153
  selecting 152
  wiring 163-165
distance traveled
  detecting, in Python 253
  measuring, with encoders 242
distance traveled, detecting with Python
  code counts 254-256
  logging 253
  troubleshooting 256
docstring 225
Draw.io
  URL 540
drive
  need for, when speech control
    stop working 486, 487
drive_arc function
  writing 281, 282
drive distance behavior
  creating 272-274

E

echo pin 157, 166
EeeBot 16
electronic skills
  about 543
  custom circuits, creating 544, 545
  electronic principles 543
  soldering 543, 544
EncoderCounter class
  unit conversions, refactoring
    into 270, 271
encoders
  adding, to Robot object 257
  attaching, to robot 248
  lifting, onto robot chassis 250, 251
  preparing 249
  types 243, 244
  used, for making specific turn 275-281

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

used, for measuring distance traveled 242
using 247, 248
using, in machines 242
wiring, to Raspberry Pi 251, 252
encoders, adding to Robot object class, extracting 257, 258
encoders, used for making specific turn drive_arc function, writing 281, 282
encoder ticks
turning, into millimeters 260-262
Esp8266 42

F

faces, tracing with Python
about 362
basic features, scanning 364, 365
behavior, planning 366
face-tracking behavior, running 371
face-tracking code, writing 367-371
integral images, converting 362-364
objects, finding in image 362
troubleshooting 372
fixed-wheel steering 136
Flask
installing 332
Flask API
programming 421
Flask behaviors
menu modes, making compatible with 518, 519
Flask control API server
programming 425-427
troubleshooting 427
frames 332
FreeCAD
URL 540
Fritzing
URL 544
Full Function Stepper Motor Hat 98
Fusion
about 360
URL 540

G

general IO 46
General Purpose Input Output (GPIO) 29, 43
Git
about 76
using 76-78
GPIOZero library 167
GrabCad
URL 540
ground, voltage, and signal (GVS) 211
Gumstix Linux 43
gyroscope
about 304
adding, to interface 307
calibrating 448-450
coordinate, representing 304-307
illustration 304
plotting 308, 309
reading, in Python 303
rotation, detecting with 447, 448
rotation systems, representing 304-307
virtual robot, rotating with 450-454

H

Haar cascades
using 362
Hackaday
URL 535, 542

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

hall-effect sensors 244, 314
hard iron offset calculation 463
HC-SR04 154
HC-SR04 sensors
  wiring, into level shifters 156, 157
headers
  attaching, to IMU 288, 289
heading
  about 305
  detecting, with magnetometer 463
headless system
  about 52-54
  uses, in robot 52-54
hobbyist robot 14, 16
hue 194, 195
Hue, Saturation, and Value (HSV)
  about 194
  converting, to RGB 196

I

ICM20948 288
IMU data
  robot, driving from 480, 481
IMU models 287, 288
Inertial Measurement Unit (IMU)
  about 287, 328
  headers, attaching to 288, 289
  wiring, to Raspberry Pi 294, 295
inertial measurement units (IMU),
      attaching to robot
  about 291
  physical placement 291-293
input/output (I/O) pins 29-31, 553
Instructables
  URL 535, 542
integral windup 351
intent
  about 410
  adding 436
  building 428-430
  code, adding 436, 437
  current skill folder 434, 435
  dialog files 433
  requirements file 431
  running 438
  settings file 430, 431
  troubleshooting 435
  vocabulary and dialog 436
  vocabulary files, creating 432, 433
iterators 335

J

JavaScript Object Notation (JSON) 520
jQuery documentation
  URL 346

K

Kismet 14

L

large array extension
  installing 332
LED interface
  making 187-189
LED rainbow
  using 202, 203
LED strip
  attaching, to robot 186
Light Emitting Diode (LED)
  about 183, 490
  adding, to Robot object 189, 190

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

rainbow display, making 194
rainbow, making on 196-198
testing 193
troubleshooting steps 192
light strip
attaching, to Raspberry Pi 185, 186
used, for debugging avoid behavior 199
light strip technologies
comparing 183, 184
line following
about 376
behavior, finding 404, 405
behavior flow diagram, creating 394
in industry, usage 377
line following, types
about 378
magnetic line following 378
visual line following 378
line following, with PID algorithm
about 393
initial behavior, writing 396-403
PID tuning 403
time, adding to PID controller 395
troubleshooting 404
lithium polymer (LiPo) battery 553
logic levels 155

M

machine learning
extending to 546, 547
magnetic line following
about 378
variants 378
magnetometer
about 313, 314
heading, detecting with 463
rough heading, obtaining from 470-472
working with 313
magnetometer calibration
about 463-466
troubleshooting 467
troubleshooting, for circle coverage 469
magnetometer calibration values
testing 467-469
magnetometer interface
adding 315
magnetometer readings
displaying 315-317
magnetometer's axes 315
Maker Faires
URL 538
makerspaces 537
Makezine
URL 536
Mars rover robots
about 8
headless by design 8
menu server
lights, adding 524, 525
micro:bit 32, 42
micro-electro-mechanical systems (MEMS) 287, 448
Microsoft Windows
Bonjour, setting up 56
MIT Robotics
URL 535
motor controller board
selecting 95
motor controller board selection
conclusion 98, 99
Motor HAT
finding 127-129
motors
troubleshooting chart 131

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

motors, connecting to Raspberry Pi
about 116-118
motor hat, wiring in 119, 120
Raspberry Pi, powering independently 121-123
motors move
demonstrating 130
Multicast Domain Name System (mDNS) 56
multiprocessing
reference link 340
Mycroft
about 409
client 418, 419
sound card, selecting 417
speaking to 419, 420
troubleshooting 420
using 418
Mycroft, voice assistant
concepts 409
dialogs 410
intent 410
skills 410
speech to text (STT) 409
utterance 409
vocabulary 410
wake word 409

N

NodeMCU 31
NumPy
installing 332

O

obstacle avoidance
basics 172-176
flowchart 173, 174
sophisticated object avoidance 176-178
odometry 94, 242
OnionIoT 43
online robot building communities
about 534
forums and social media 534, 535
technical questions 536
Open Computer Vision (OpenCV) library
about 331
installing 331
optical encoders 244
optical sensors 153
opto-interrupters 244
orientation
sensors, combining for 473-478
Orionrobots
URL 535

P

pan and tilt code
creating 224
running 231
servo, adding to robot class 227, 228
servo object, making 224-226
troubleshooting 231, 232
pan and tilt head
making, to move in circles 228-231
pan and tilt mechanism
about 8
adding 217, 218
attaching, to robot 223, 224
kit, building 219-223
pan-and-tilt mechanism
Raspberry Pi camera, attaching to 323-327

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

parts
  buying 554
  selecting 553
  test-fit diagram 554
photo interrupters 244
physical robot
  planning 37-39
Pi camera
  about 44
  software, setting up 330
PID controller
  components 264
  derivative (D) 264
  integral (I) 264
  proportional (P) 264
Pimoroni
  URL 535, 536
Pimoroni LED SHIM 184
Pimoroni Python library
  installing 295, 296
  troubleshooting 297
Pinshape
  URL 540
pitch 305
pitch-and-roll angles
  detecting, with accelerometer 454
pitch-and-roll angles, obtaining
    from accelerometer vector
  about 454-457
  troubleshooting 458
Pi Wars 15
plasticard 542
polar plot 236
pose 306
power pins 45
power switch
  adding, to motor power 160-162
prerequisites, for building
    competition-grade robot
  design skills 539
  electronic skills 543
  skills, for shaping and building 541
Pre-Shared Key (PSK) 55
primitives 442
Printed Circuit Board and
    Assembly (PCBA) 544
Printed Circuit Boards (PCBs) 252, 537
Proportional Integral Derivative (PID)
  about 348, 515, 546, 555
  veer, correcting with 263, 264
proportional-integral (PI) 265
Pulse Width Modulation (PWM) 30, 208
PuTTY
  used, for connecting to
    Raspberry Pi 59-61
PyImageSearch
  URL 545
Python
  accelerometer, reading 310
  colored objects, following with 347, 348
  distance traveled, detecting in 253
  faces, tracing with 362
  gyroscope, reading 303
Python libraries
  installing, to communicate
    with sensor 166
Python OpenCV libraries
  Flask 332
  installing 331
  large array extension 332
  NumPy 332
Python PID control object
  creating 265, 266

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

R

rack and pinion steering 134
rainbow
  making, on Light Emitting Diode (LED) 196-198
rainbow display
  making, with Light Emitting Diode (LED) 194
Raspberry Jams
  URL 538
Raspberry Pi
  about 31, 62
  encoders, wiring to 251, 252
  finding, on network 56
  IMU, wiring to 294, 295
  lifting 250
  light strip, attaching to 185, 186
  motors, connecting to 116-118
  physical installation 412, 413
  PuTTY or SSH, used for connecting to 59-61
  ReSpeaker software, installing 414
  selecting 33
  setup, testing 57, 58
  sound input, adding 411, 412
  sound output, adding 411, 412
  SSH, enabling 54-56
  troubleshooting 58
  voice agent, installing on 413, 414
  wireless, setting up 54-56
Raspberry Pi 3A+ 33, 43
Raspberry Pi 4B 33
Raspberry Pi camera
  attaching, to pan-and-tilt mechanism 323-327
  pictures, obtaining from 331
  setting up 322, 323
  wiring in 327-329
Raspberry Pi Camera Guide
  URL 328
Raspberry Pi camera stream app
  building 332
  CameraStream object, writing 334, 335
  image server main app, writing 336, 337
  OpenCV camera server, designing 333
  server, running 338, 339
  template, building 338
  troubleshooting 339
Raspberry Pi Forums
  URL 59
Raspberry Pi HATs 46
Raspberry Pi OS
  about 46, 47
  configuring 61, 62
  rebooting 65-67
  reconnecting 65-67
  renaming 62-64
  SD card, preparing with 47-49
  securing 64
  shutting down 69
  software, updating 67, 68
Raspberry Pi OS Lite 47
Raspberry Pi, pinouts
  reference link 96
Raspberry Pi, preparing for remote driving
  about 500, 501
  code, creating for sliders 512-516
  image app core, enhancing 501, 502
  manual drive behavior, writing 502-505
  running 516, 517
  style sheet 509-512
  template (web page) 505-508
  troubleshooting 517, 518

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Raspberry Pi's, capabilities
  connectivity 43
  exploring 42
  networking 43
  power 42
  speed 42
Raspberry Pi Zero W 34
record-and-replay interface
  using 246
relative encoders 244, 245
requirements, for selecting
    motor controller board
  connectors 98
  integration level 95
  pin usage 96
  power input 98
  size 96, 97
  soldering 97
requirements, for selecting
    robot chassis kit
  cost 94
  motors 93, 94
  simplicity 94
  size 91
  wheel count 91, 92
  wheels 93, 94
ReSpeaker software
  installing 414, 415
  troubleshooting 416
RGB
  HSV, converting to 196
RGB strip 182
RGB values 184, 185
robot
  about 4, 5
  code object, displaying to 187
  code structure, planning 34-36
  components, planning 34-36
  controller 31, 33
  controllers and I/O, exploring 29
  driving, from IMU data 480, 481
  encoders, attaching to 248
  IMU, attaching 291
  I/O pins 29-31
  LED strip, attaching to 186
  pan and tilt mechanism,
      attaching to 223, 224
  physical system 20-23
  powering 99-101
  programming 557
  Raspberry Pi, selecting 33, 34
  script, writing to follow
      predetermined path 145-147
  sensors, securing to 158, 159
  starting, automatically with
      systemd tool 525, 526
  steering 133, 137, 138
  test fitting 102-104
  visualizing 550-552
robot arms 12, 13
robot base, assembling
  about 105, 106
  AA battery holder, mounting 115
  batteries, adding 113
  castor wheel, adding 110
  completing 116
  encoder wheels, attaching 107
  metal motor brackets, fitting 109
  motor brackets, fitting 107
  plastic motor brackets, fitting 108
  Raspberry Pi, fitting 112, 113
  USB power bank, setting up 114
  wheels, pushing on 111
  wires up, bringing 111, 112
robot behaviors, menu modes
  HTML template 492-494

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

robot modes, managing 489, 490
running 494-496
selecting 487, 488
troubleshooting 491-497
web service 491, 492
robot builders
meeting 537
robot chassis sizes
comparing 91
robot chassis
encoders, lifting onto 250, 251
robot chassis kit
selecting 90
robot chassis kit selection
conclusion 94
Robot class
sensors, adding to 171
robot components
actuators, types 26
motors, types 24
sensors, types 27-29
status indicators 26, 27
types, exploring 23
robot components, motors
brushless motor 25
DC gear motor 24
DC motor 24
servomechanism 25
stepper motor 25
robot components, sensors
line sensor 28
microphones 28
optical distance sensor 28
optical interrupt sensor 29
Raspberry Pi camera module 28
ultrasonic distance sensor 28
robot, direction and speed
encoding 246, 247
robot fully phone-operable
making 518
menu modes, making compatible with Flask behaviors 518, 519
menu, styling 522
menu template, making into buttons 522-524
video services, loading 519-521
robotics
in education 14
robotics competitions
about 538, 539
Micromouse, URL 539
PiWars 538
Robotex International, URL 539
robot, modeling in VPython
about 442-446
troubleshooting 446, 447
Robot object
code, building 140-145
constants, setting 272
device, adding to 258-260
encoders, adding to 257
LEDs, adding to 189, 190
making 138
need for 139
Robot Operating System (ROS)
about 547
URL 547
robots
example, washing machine 9
exploring, in industry 11
household robots 10, 11
using, in homes 9
robots, inputs
controller 5
sensor 5

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

robot software layers
  apps/behaviors 555
  Libraries and Middleware 555
  Vendor Libraries 555
robots, outputs
  motor 5
robot, starting automatically
    with systemd tool
  troubleshooting 527, 528
robot, user controls via Mycroft
  behavior remotely 422
  overview 421, 422
  robot modes, managing 422-424
roll 305
rotation
  detecting, with gyroscope 447, 448
rotor 304
rough heading
  obtaining, from magnetometer 470-472

S

saturation 195
Scalable Vector Graphics (SVG) 506
scanning sonar
  building 232
  code, writing 236-238
  issues, troubleshooting 239
  library, attaching 236
  sensor, attaching 233-235
Science Technology Engineering and
    Mathematics (STEM) 538
script
  writing, to avoid obstacles 170
SD card
  preparing, with Raspberry Pi OS 47-49
SD card backups
  making 79
SD card corruption 72
Secure File Transfer Protocol (SFTP)
  about 73
  files, copying from Raspberry
    Pi over network 73-76
Secure Shell (SSH)
  about 54, 485
  used, for connecting to
    Raspberry Pi 59-61
sensors
  adding, to Robot class 171
  securing, to robot 158, 159
  troubleshooting steps 169, 170
  two sensors, using 157, 158
sensors, combining for orientation
  180-degree problem, fixing 478, 479
  about 473-478
Serial Peripheral Interface (SPI) 127, 556
servo horn
  about 210
  fitting 210, 211
servomechanism motors. *See* servo motors
servo motor 25
servo motors
  about 206, 207
  calibrating 216, 217
  controlling 215, 216
  exploring 208
  input positions, sending to 208, 209
  issues, troubleshooting 215
  positioning, with Raspberry Pi 210
  turning, by writing code 211-214
  working 208
shaping and building skills
  hand skills and tools 542
  machine skills and tools 541

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

Single-Board Computer (SBC)
about 42
BeagleBone 43
CHIP 43
Gumstix Linux 43
OnionIoT 43
single LED
testing 191
skills 410
Skittlebot 15
soldering 97, 288, 289
solder joint
creating 289, 290
sophisticated object avoidance 176-178
Sparkfun
URL 535, 536
speaker and microphone, on robot
considerations 411
specific distance
driving 270
speech to text (STT) 409
SpiderBot
block diagram 552, 553
steerable wheels 134, 135
steering
types 133
steering systems 136, 137
stepper motor 25
straight line
behavior, troubleshooting 269
code, writing to go in 266-269
driving in 262
Python PID control object,
creating 265, 266
veer, correcting with PID 263, 264
strategies, for powering robot
battery eliminators 100
dual batteries 100
streaming
background tasks, running 340, 341
systemd tool
using, to automatically start
robot 525, 526

T

tachometer 94
tachometers 242
tachos 242
tacho wheels 94
temperature
graphing 298, 299
reading, with Pimoroni Python
library 295, 296
temperature plotter
running 300, 301
temperature
reading 295
temperature register
reading 297
reading, by creating interface 297
reading, with VPython 298
TensorFlow Tutorials
URL 546
Tested channel
URL 536
test images
capturing 384-386
need for 384
Python code, writing for
edges of line 386-389
testing, without clear line 391-393
used, for locating line edges 390, 391
used, for testing computer
vision pipeline 384

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

test track
  creating 380, 381
  managing 379
  materials, obtaining 379
Thingiverse community
  URL 540
through-hole components 544
time of flight 153
trigger pin 157, 166
tuple 187

U

ultrasonic distance sensor
  reading 166-169
ultrasonic sensor
  attaching 158
  connections 163
  reading 158
ultrasonic sensors 154
Unified Modeling Language (UML) 556
Uniform Resource Locator (URL) 493
Unotron robot 135
utterance 409

V

value 195
variable resistor 243
vector 306
  accelerometer, displaying as 311, 312
virtual robot
  programming 442
virtual robot, rotating with gyroscope
  about 450-454
  troubleshooting 454
visual clues 378
visual line following
  about 378
  detecting, with camera 378
  detecting, with light sensors 378
Visual Python (VPython)
  about 296
  for reading temperature register 298
  robot, modeling 442-446
  troubleshooting 302
vocabulary 410
voice agent
  installing, on Raspberry Pi 413, 414
voice agent, programming with
    Mycroft on Raspberry Pi
  about 428
  intent, adding 436
  intent, building 428-430
voice agent terminology 409
voice assistant 409
voltage 155
VPython command line
  simplifying 303

W

wagon-style steering 134
wake word 409
warehouse robots 13
Wheels box 36
Win32DiskImager
  using 79-81
World Coordinate System 305

X

XRobots 535

Y

yaw 305

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 11:36 AM via . All use subject to https://www.ebsco.com/terms-of-use