

Raspberry Pi Computer Vision Programming

Second Edition

Design and implement computer vision applications with Raspberry Pi, OpenCV, and Python 3



Ashwin Pajankar

Packt

www.packt.com

Raspberry Pi Computer Vision Programming

Second Edition

Design and implement computer vision applications
with Raspberry Pi, OpenCV, and Python 3

Ashwin Pajankar

Packt

BIRMINGHAM—MUMBAI

Raspberry Pi Computer Vision Programming

Second Edition

Copyright © 2020 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.

Commissioning Editor: Sunith Shetty

Acquisition Editor: Reshma Raman

Senior Editor: Ayaan Hoda

Content Development Editor: Nazia Shaikh

Technical Editor: Utkarsha S. Kadam

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Joshua Misquitta

First published: May 2015

Second edition: June 2020

Production reference: 1260620

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80020-721-9

www.packtpub.com



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](http://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](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://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.

Contributors

About the author

Ashwin Pajankar is a polymath. He is a science popularizer, a programmer, a maker, an author, and a YouTuber. He graduated from IIIT Hyderabad with an MTech in computer science and engineering. He has a keen interest in the promotion of **science, technology, engineering, and mathematics (STEM)** education.

About the reviewers

Lentin Joseph is an author, roboticist, and robotics entrepreneur from India. He runs a robotics software company called Qbotics Labs in Kochi/Kerala. He has 10 years of experience in the robotics domain, primarily working with the **Robot Operating System (ROS)**, OpenCV, and PCL. He has authored several books on ROS, namely *Learning Robotics Using Python First Edition* and *Second Edition*, *Mastering ROS for Robotics Programming First Edition* and *Second Edition*, *ROS Robotics Projects First Edition* and *Second Edition*, and *Robot Operating System for Absolute Beginners*. He has masters in Robotics and Automation from Amrita Vishwa Vidapeetham University in India and also worked at the Robotics Institute, CMU, USA. He is also a TEDx speaker.

Arush Kakkar is an author, entrepreneur, and a computer vision and deep learning researcher. He is currently the CEO of Agrex.ai, a company started by him. It's a video analytics company utilizing Artificial Intelligence to analyze human behavior. The applications include analyzing customer behavior in retail stores, security threats in sensitive installations, thermal vision processing, face recognition, and improving operational efficiency.

He has authored two books *Raspberry Pi By Example* and *Raspberry Pi: Amazing Products from Scratch* both published by Packt Publishing. Arush has extensive experience in using Raspberry Pi to build products that use computer vision to accomplish certain tasks.

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.

Table of Contents

Preface

1

Introduction to Computer Vision and the Raspberry Pi

Understanding computer vision	2	Setting up Raspbian on a Raspberry Pi	14
OpenCV	3	Downloading the necessary software	20
Single-board computers	4	Preparing the microSD card manually	23
The Beagleboard family	6	Booting up the Raspberry Pi for the first time	26
ASUS Tinkerboard	6	Connecting various RPi board models to the internet	34
NVIDIA Jetson	7		
Intel boards	7		
Raspberry Pi	8	Updating the RPi	37
Raspberry Pi models	8	Summary	38
OSes for Raspberry Pi	13		

2

Preparing the Raspberry Pi for Computer Vision

Remotely logging into the RPi with SSH	40	Installing OpenCV on an RPi board	46
Remote desktop access	44	Heatsinks and overclocking RPi 4B	47
		Summary	49

3

Introduction to Python Programming

Technical requirements	52	The basics of NumPy	62
Understanding Python 3	52	Matplotlib	65
Python on RPi and Raspberry Pi OS	53	RPi GPIO programming with Python 3	71
Python 3 IDEs on Raspberry Pi OS	54		
Working with Python 3 in interactive mode	59	LED programming with GPIO	72
The basics of Python 3 programming	59	Push-button programming with GPIO	80
The SciPy ecosystem	62	Summary	83

4

Getting Started with Computer Vision

Technical requirements	86	Webcam video recording	103
Exploring image datasets	86	Capturing images with the webcam using Python and OpenCV	103
Working with images using OpenCV	86	Live videos with the webcam using Python and OpenCV	104
Using matplotlib to visualize images	89	Webcam resolution	105
Drawing geometric shapes with OpenCV and NumPy	92	FPS of the webcam	106
Working with a GUI	95	Saving webcam videos	107
Event handling and a primitive paint application	96	Playing back the video with OpenCV	109
Working with a USB webcam	99	The Pi camera module	110
Capturing images with the webcam	100	Capturing images and videos with the raspistill and raspivid utilities	112
Timelapse photography	100	Using picamera with Python 3	113
		Using the RPi camera module and Python 3 to record videos	116
		Summary	116

5

Basics of Image Processing

Technical requirements	118	Blending and transitioning images	126
Retrieving image properties	118	Multiplying images by a constant and one another	130
Basic operations on images	120	Creating a negative of an image	131
Splitting the image into channels	121	Bitwise logical operations on images	132
Adding a border to an image	121	Summary	134
Arithmetic operations on images	123		

6

Colorspaces, Transformations, and Thresholding

Technical requirements	136	The translation, rotation, and affine transformation of images	145
Colorspaces and converting them	136	Perspective transformation of images	150
HSV colorspace	138	Thresholding images	152
Tracking in real time based on color	140	Otsu's binarization method	156
Performing transformation operations on images	143	Adaptive thresholding	156
Scaling	144	Summary	158

7

Let's Make Some Noise

Technical requirements	160	Filtering and blurring with OpenCV	169
Noise	160	2D convolution filtering	169
Introducing noise to an image	160	Low-pass filtering	170
Working with kernels	166	Summary	172
2D convolution with the signal processing module in SciPy	167		

8**High-Pass Filters and Feature Detection**

Technical requirements	174	transforms	182
Exploring high-pass filters	174	Harris corner detection	185
Working with the Canny edge detector	179	Exercise	186
Finding circles and lines with Hough		Summary	187

9**Image Restoration, Segmentation, and Depth Maps**

Technical requirements	190	quantization	194
Restoring damaged images using inpainting	190	Comparison of k-means and the mean shift algorithm	200
Segmenting images	192	Disparity maps and depth estimation	201
Mean shift algorithm segmentation	192	Summary	202

10**Histograms, Contours, and Morphological Transformations**

Technical requirements	204	Visualizing image contours	212
Computing and visualizing histograms	204	Applying morphological transformations to images	214
Histogram equalization	210	Summary	219

11**Real-Life Applications of Computer Vision**

Technical requirements	222	Implementing background subtraction	224
Implementing the Max RGB filter	222	Computing the optical flow	226

Detecting and tracking motion	228	Implementing the chroma key effect	238
Detecting barcodes in images	232	Summary	244

12

Working with Mahotas and Jupyter

Technical requirements	246	Combining Mahotas and OpenCV	250
Processing images with Mahotas	246	Other popular image processing libraries	252
Reading images and built-in images	247	Exploring the Jupyter Notebook for Python 3 programming	252
Thresholding images	247	Summary	261
The distance transform	248		
Colorspace	249		

13

Appendix

Technical requirements	263	Windows	266
Performance measurement and the management of OpenCV	263	Tour of the raspi-config command-line utility	268
Reusing a Raspbian OS microSD card	264	Installation and the environment setup on Windows, Debian, and Ubuntu	271
Formatting the SD card using the SD card formatter	264	Python implementations and Python distributions	273
The Disk Management utility in			

Other Books You May Enjoy

Leave a review - let other readers know what you think 277

Index

Preface

Computer vision and image processing have extended from being a field of niche research to everyday usage. However, despite this revolution, one of the key challenges faced in computer vision development and application development is a lack of a well-designed guide that teaches you how it works step by step. This book addresses this key challenge.

We will start with the basics of Raspberry Pi and Python and explore Python 3 programming with various supporting libraries, such as NumPy, SciPy, and Matplotlib. Next, we will understand the basics of **General-Purpose Input Output (GPIO)** pins on Raspberry Pi and learn about its programming with Python 3. We will look at a lot of examples of Raspberry Pi and computer vision programming with Python and GPIO throughout the entirety of this book.

Then, we will move on to the installation of OpenCV on Raspberry Pi. We will look at the basics of OpenCV programming and explore the concepts of advanced image processing and computer vision. We will learn about and demonstrate concepts such as thresholding, segmentation, image quantization, image restoration, mathematical morphology, and contours. Then, we will implement a few real-life applications with OpenCV, Python, and GPIO.

We will also learn how to use another library—Mahotas—and the Jupyter Notebook. Additionally, we will learn how to install all the libraries that we will discuss on a Windows computer. Finally, the *Appendix* section has a range of useful topics relating to Raspberry Pi that are not included in other chapters.

Who this book is for

This book is for Python 3 developers, computer vision professionals, and Raspberry Pi enthusiasts who are looking to implement computer vision applications on a low-cost platform. Basic knowledge of programming, mathematics, and electronics will be beneficial. However, even beginners in this area will be comfortable with covering the contents of the book as they are presented in a step-by-step manner.

What this book covers

Chapter 1, Introduction to Computer Vision and Raspberry Pi, illustrates the concept of single-board computers, OpenCV, and Raspberry Pi. We will also learn how to set up Raspbian OS on Raspberry Pi.

Chapter 2, Preparing Raspberry Pi for Computer Vision, teaches us how to set up Raspberry Pi for demonstrations of computer vision.

Chapter 3, Introduction to Python Programming, introduces us to Python 3 programming. We will learn about libraries such as NumPy and Matplotlib. We will also demonstrate the use of a few programs with LEDs and push buttons in detail.

Chapter 4, Getting Started with Computer Vision, focuses on the basics of computer vision programming and interfacing various camera modules with Raspberry Pi. We will also learn how to work with images and the GUI in this chapter in detail.

Chapter 5, Basics of Image Processing, looks at basic operations on images, such as bitwise arithmetic and bitwise logical operations.

Chapter 6, Colorspaces, Transformations, and Thresholding, is where we will analyze the concept of basic segmentation and thresholding. We will learn about various geometric and perspective transformations. We will also learn about colorspaces and their application in detail.

Chapter 7, Let's Make Some Noise, explores the concept of filters and how to use low-pass filters to reduce noise in images. We will learn about concepts such as kernels and convolution in detail.

Chapter 8, High-Pass Filters and Feature Detection, goes into the aspects of detecting various features, such as lines, circles, edges, and corners, using high-pass filtering techniques.

Chapter 9, Image Restoration, Segmentation, and Depth Map, investigates restoring degraded and damaged images, segmenting with Python's implementation of the k-means and mean-shift algorithms, and estimating depth maps.

Chapter 10, Histograms, Contours, and Morphological Transformations, analyzes images with histograms, and we will learn how to enhance images by equalizing histograms. We will also dig deeper into contours and mathematical morphological operations.

Chapter 11, Real-Life Applications of Computer Vision, demonstrates applications in the real world with OpenCV, Python 3, and Raspberry Pi.

Chapter 12, Working with Mahotas and Jupyter, delves into the brief usage of another scientific image processing library known as Mahotas. We will also understand how to work with the Jupyter Notebook for Python 3 programming.

Chapter 13, Appendix, is a collection of assorted topics relating to Python, Raspberry Pi, and computer vision that did not fit in to earlier chapters.

To get the most out of this book

All of the programs included in all of the chapters in this book are executed on Raspberry Pi with Raspbian OS. You will need a Windows PC and an internet connection to set up Raspbian OS on the Raspberry Pi board. The instructions for setting up on Windows are included in the *Appendix* section. You are encouraged to use the latest revision of Raspberry Pi (at the time of writing, it is Raspberry Pi 4B); however, the programs will also run on any generation of the boards in the Raspberry Pi family. These programs are written for the Python 3 interpreter and they may not work with Python 2. Apart from the Raspberry Pi board and a Windows PC for the setup, you will also need a Raspberry Pi camera module and a USB webcam. Also, some electronic components, such as push buttons, LEDs, breadboards, and jumper cables, are needed to work with the Python 3 GPIO library.

If you are using the digital version of this book, we advise you to type the code in 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/pasting of code.

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming>. 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 (<https://bit.ly/3evn0ln>).

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: https://static.packt-cdn.com/downloads/9781800207219_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: "Just like lxterminal, we can run Linux commands from here too."

A block of code is set as follows:

```
p2 = Person()
p2.name = 'Jane'
p2.age = 20
print(p2.name)
print(p2.age)
```

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

```
sudo apt-get install xrdp -y
```

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: "Open the **Remote Desktop Connection** application on your Windows PC."

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.

1

Introduction to Computer Vision and the Raspberry Pi

OpenCV is a simple and powerful programming framework for computer vision. It is preferred by both novices and experts in the field of computer vision. We can easily learn computer vision by writing OpenCV programs using Python 3 as the programming language. The Raspberry Pi family of single-board computers uses Python as its preferred development language. Using a Raspberry Pi board and Python 3 for learning OpenCV programming is one of the best approaches that we can follow to commence our wonderful journey into the amazing field of computer vision programming. In this chapter, you will become familiar with all of the important concepts that you need in order to get started with the Raspberry Pi and computer vision. By the end of this chapter, you will be able to set up the Raspbian **Operating System (OS)** on various Raspberry Pi board models. You will also learn how to connect the boards to the internet.

In this chapter, we will cover the following topics:

- Understanding computer vision
- Single-board computers

- The Raspberry Pi family of single-board computers
- Setting up the Raspbian OS on a Raspberry Pi
- Connecting various Pi board models to the internet with LAN or Wi-Fi

By the end of this chapter, you will be able to set up your own Raspberry Pi board.

Understanding computer vision

The field of computer vision is a combination of different fields, including (but not limited to) computer science, mathematics, and electrical engineering. It includes ways to capture, process, and analyze images and videos from the real world in order to assist in decision making. Computer vision means mimicking biological (that is, human and non-human) vision. The end goal of most computer vision systems is to extract useful information from still images and videos (including prerecorded videos and live feeds) for the purpose of decision making. Biological vision systems work in a similar fashion. Additionally, unlike biological vision, computer vision can also acquire and work with images from the visual spectrum that are not visible to biological entities, for example, infrared and depth images.

Computer vision also relates to the area of extracting information from captured images and videos. A computer vision system may accept various types of data, such as images, videos, and live video streams, as inputs to further process, analyze, and extract meaningful information for the purpose of making important decisions.

The fields of artificial intelligence, machine vision, and computer vision overlap and share many topics, such as image processing, pattern recognition, and machine learning, as depicted in the following diagram:

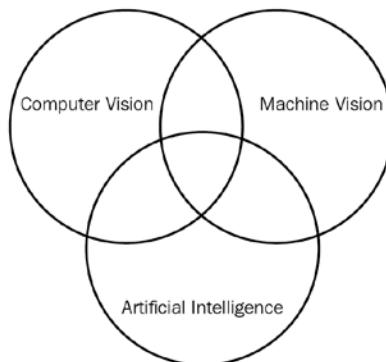


Figure 1.1 – The relationships between different scientific domains

In order to work as a researcher in the area of computer vision, you need to have a solid background and understanding of mathematics. However, to write programs for computer vision using OpenCV and Python 3, you don't need to know a lot of mathematics. Note that, in this book, you will be learning all of the mathematical and theoretical concepts required to get started with image processing and computer vision.

The typical objectives of a computer vision system could be one or more of the following:

- The recognition of objects, the classification of visual detection, and an analysis of motion
- The reconstruction of scenes using images
- Image denoising and restoration

Do not get stressed if you are unfamiliar with these key terms. We will explore and implement many of these concepts throughout our journey.

OpenCV

OpenCV (also known as **Open Source Computer Vision**) is an open source library for computer vision and machine learning. It has many functionalities for image processing and computer vision. It is a cross-platform library, and it works with many programming languages and OSes. It has a large collection of computer vision and machine learning-related functions. It also has several **Graphical User Interface (GUI)** and event handling features.

OpenCV is free for academic and commercial usage as it is under the **Berkley Software Distribution (BSD)** license. It is written with the C++ programming language. It has interfaces for most of the popular programming languages, including (but not limited to) C/C++, Python, and Java. It runs on a variety of OSes, including Windows, Android, Linux, macOS, and other Unix-like OSes. In this book, we will write computer vision-related programs with OpenCV and Python 3.

The library has more than 2,500 optimized algorithms for machine learning and computer vision tasks. It has a community of more than 47,000 computer vision professionals, and it has been downloaded more than 18 million times. OpenCV is extensively used in academics for teaching, research organizations, government organizations, and various industry segments. Reputed able organizations such as Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, and Toyota all use OpenCV.

Let's take a look at the history of OpenCV. OpenCV was originally an in-house initiative of Intel Research and was used to develop a framework to work with images and videos. It was initially supported by *Willow Garage* and then *Itseez*.

Note

You can visit the website of Willow Garage at <http://www.willowgarage.com/>.

In August 2012, the responsibility for further development and support for OpenCV was assumed by an independent, not-for-profit, organization, OpenCV.org. It maintains the website for OpenCV. In May 2016, Intel acquired Itseez. The following URLs have the press announcement from Intel and OpenCV.org:

- <https://newsroom.intel.com/editorials/intel-acquires-computer-vision-for-iot-automotive/>
- <https://opencv.org/intel-acquires-itseez/>

Here's a brief timeline of the developments related to OpenCV:

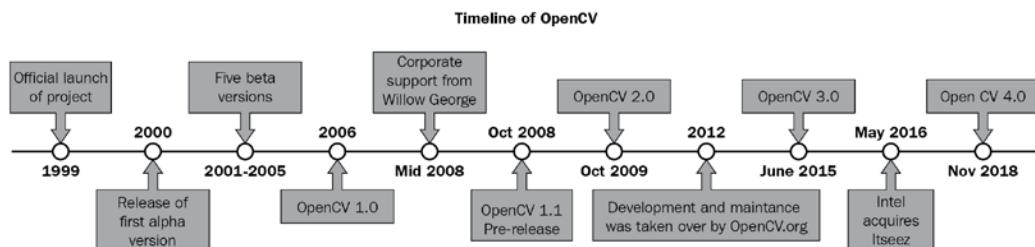


Figure 1.2 – Timeline of OpenCV

You can find all the details, including different versions and press releases of the OpenCV library, at <https://opencv.org/>.

As we will be writing computer vision programs with Raspberry Pi as the platform, we will study single-board computers and Raspberry Pi in detail. We will learn how to set up the Raspbian OS on various models of a Raspberry Pi single-board computer.

Single-board computers

A **single-board computer** (abbreviated to **SBC**) is a complete computer system on a single **printed circuit board** (abbreviated to **PCB**). The board usually has a processor(s), RAM, **input/output (I/O)**, an Ethernet port for networking, and USB ports for interfacing with USB devices. A few single-board computers have Wi-Fi and Bluetooth, too. SBCs run OS distributions such as Ubuntu, Windows, Debian, and more. These OS distributions have specially tailored versions for use with these SBCs.

Unlike traditional computers, an SBC is not modular and its hardware cannot be upgraded because all the components (such as the CPU, RAM, GPU, and interfacing ports) are integrated on a single PCB itself. SBCs are used as low-cost computers in academia, research, and various other industries. The use of SBCs in embedded systems is quite widespread, and many individuals, research organizations, and companies have developed and released fully functional and usable products based on SBCs. Many of these products are crowdfunded. The main advantage of SBCs is onboard **General-Purpose Input/Output (GPIO)** pins. These pins provide functionalities such as various buses (**Serial Peripheral Interface (SPI)**, I2C, and SMBus), digital I/O, analog input, and **Pulse Width Modulation (PWM)** output. Try not to get overwhelmed with all of this technical vocabulary. We will learn most of these concepts in more detail with the help of experiments. Almost all of the popular SBCs have GPIO in some form or other. Due to their small form factor and onboard GPIO, they are popular in schools, universities, training centers, boot camps, and maker spaces. They are frequently used in the areas of sensor networks and the **internet of things (IoT)**.

To summarize, the advantages of SBCs are as follows:

- Low cost
- Small size
- Low power consumption
- Provision for onboard networking and I/O

However, SBCs come with their own set of disadvantages. As all the components of an SBC are on the same PCB, it can be very difficult to repair if a component is damaged due to mechanical or electrical reasons. For the same reason, we cannot even upgrade anything on an SBC. These are the only major disadvantages of SBCs.

The Microcomputer Trainer MMD-1, designed by John Titus in 1976, is the first true single-board microcomputer that was based on the Intel microprocessor, C8080A. It was called **dyna-micro** during the prototyping phase, and the production units were called **MMD-1** (short for **Mini-Micro Designer 1**).

We are now going to take a look at the Raspberry Pi series in detail. However, before that, we will become acquainted with other popular SBC families.

The Beagleboard family

The BeagleBoard.org Foundation is an organization based in the USA. It is a non-profit entity, and its objective is to provide education and collaboration around the design, development, testing, and use of open source hardware and software in the area of embedded systems. They have developed various SBCs named after beagles (a popular breed of domestic canine species). You can find a list of the current SBCs that they developed, which are in production, at <http://beagleboard.org/boards>. You can also find related products and accessories for Beagle boards at the same URL.

Their latest product, at the time of writing, is PacketBeagle (<http://beagleboard.org/pocket>).

ASUS Tinkerboard

The ASUS Tinkerboard is designed and manufactured by ASUS (a Taiwan-based multinational corporation). Its size, layout, and pins are compatible with second- and third-generation Raspberry Pi boards. You can find more details about all the editions of the ASUS Tinkerboard at <https://www.asus.com/us/Single-Board-Computer/>. The following photograph shows the top view of an ASUS Tinkerboard:

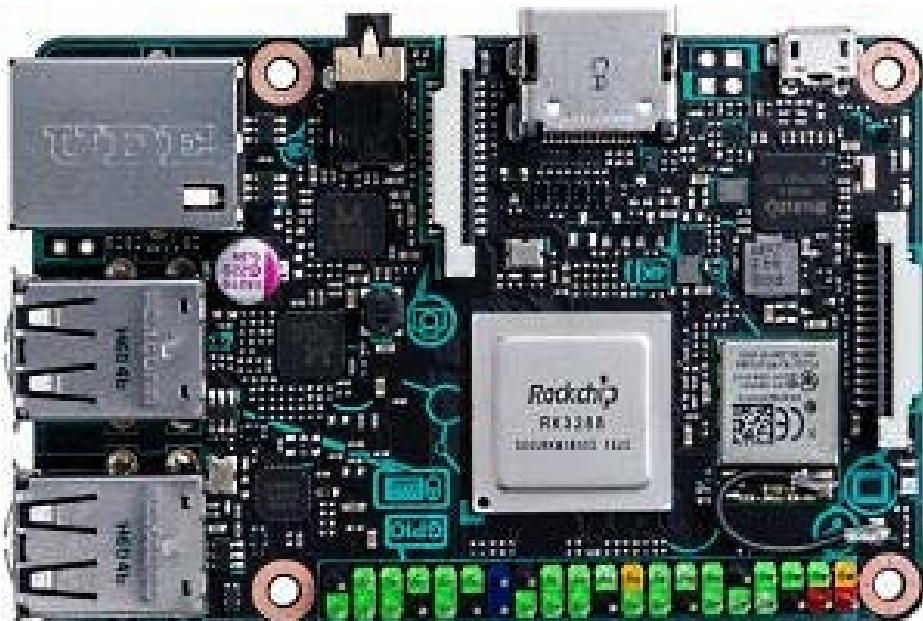


Figure 1.3 – ASUS Tinkerboard

NVIDIA Jetson

NVIDIA Jetson is a family of modules that is used for computer vision, AI, and speech processing tasks (<https://developer.nvidia.com/embedded/develop/hardware>). The best member for beginners to get started with is Jetson Nano. And the best place to begin is on the web page of **Jetson Nano Developer Kit** at <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>. Here's a side view of the developer kit:



Figure 1.4 – Nvidia Jetson Nano

Intel boards

Intel Corporation also produces many boards that can be called SBCs. You can find details on the current generation of modules that are in production at <https://software.intel.com/en-us/iot/hardware/all>. We have had the privilege of working with several of the excellent Intel SBCs and modules. Many of them are discontinued, and you can find the full list and support documentation for them at <https://software.intel.com/en-us/iot/hardware/discontinued>. Note that you might be able to get a good deal on used and discontinued boards from Intel. They are also great for learning. For beginners of computer vision, I like to recommend **Intel Up Squared Kit**. You can find out more at <https://software.intel.com/en-us/iot/hardware/up-squared-ai-vision-dev-kit>.

Raspberry Pi

Raspberry Pi is a series of low-cost and credit card-sized SBCs developed by the Raspberry Pi Foundation in the United Kingdom. The purpose of developing Raspberry Pi was to promote the teaching of basic computer skills and programming in schools, in which it has served very well. Raspberry Pi has expanded its footprint well beyond its intended purpose by gaining prominence in the embedded systems market and computer science research in academia and industrial applications.

The Raspberry Pi Foundation offers downloads for many popular OS distributions. We can use a variety of programming languages such as Python, C, C++, and Java with Raspberry Pi. You can find more information on the Raspberry Pi Foundation website (<https://www.raspberrypi.org/>).

Raspberry Pi models

The Raspberry Pi board comes in many models. Additionally, there are a lot of associated accessories with these models. You can find the current list of models under production on the products page of the Raspberry Pi Foundation (<https://www.raspberrypi.org/products/>). Unfortunately, the page does not have any information on the discontinued product boards of the Raspberry Pi family.

Additionally, Raspberry Pi is also available in a more flexible form that is intended for industrial and embedded applications. This is known as a **compute module**. The compute module also has many iterations. A compute module prototyping kit is also made available by the foundation. You can find out more about compute modules and the prototyping kit on the same Raspberry Pi products page we discussed earlier.

As we have discussed, there are many models of Raspberry Pi boards available. And while it is tempting to discuss the technical specifications in detail for all of those boards, it is difficult to achieve that in brief. In the first edition of the book, I discussed the specifications of all the available Raspberry Pi board models in detail, since the number of models was far lower and we could count them on our fingers. Since writing the second edition of this book, there are over a dozen Raspberry Pi models. Therefore, we will discuss the technical specifications of only a couple of board models of Raspberry Pi.

We will use the Raspberry Pi 4B 4 GB and Raspberry Pi Zero W with header models for our computer vision examples. However, these examples can also be run on the other board models of Raspberry Pi. This is because all the software that we use (the OS, the programming language, and the OpenCV library) is fully backward compatible.

Raspberry Pi model 4B

You can find the product specifications of the Raspberry Pi 4B at <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>.

The following table explains the product specifications in detail:

Component	Specification
Processor	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
RAM	Depending on the model (1GB/2GB/4GB)
Network Connectivity	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE Gigabit Ethernet
USB	2 USB 3.0 ports; 2 USB 2.0 ports
GPIO Pins	Raspberry Pi standard 40 pin GPIO header, which is backward compatible
Video Output	2 × micro-HDMI ports that support up to 4k and 60 FPS
Data Storage	Micro-SD card slot for loading operating system and data storage
Power supply	3A 5V DC power through USB-C connector or GPIO header
DSI Display	2-lane MIPI DSI display port
Camera	2-lane MIPI CSI camera port
Audio	4-pole stereo audio
Graphics	Broadcom VideoCore VI @ 500 MHz

Figure 1.5 – Product specification list of the Raspberry Pi model 4B

The following diagram shows all of the important connectors and components on a Raspberry Pi board:

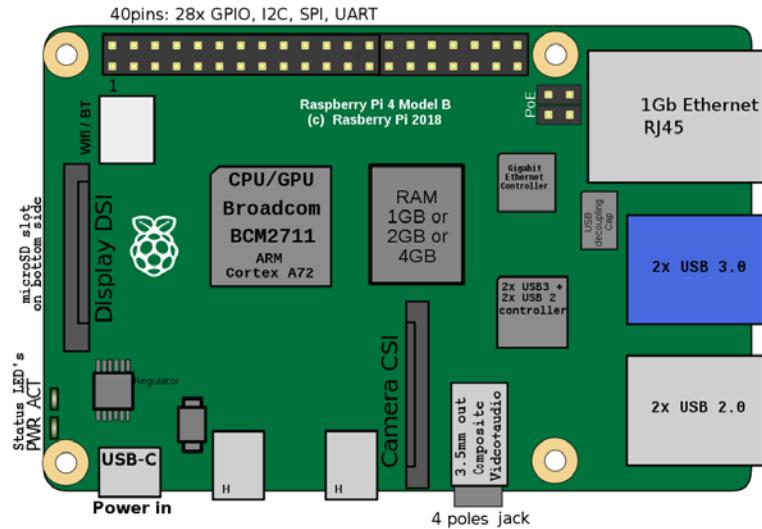


Figure 1.6 – Raspberry Pi 4B top view

The following photograph shows the top view of the Raspberry Pi model 4B:

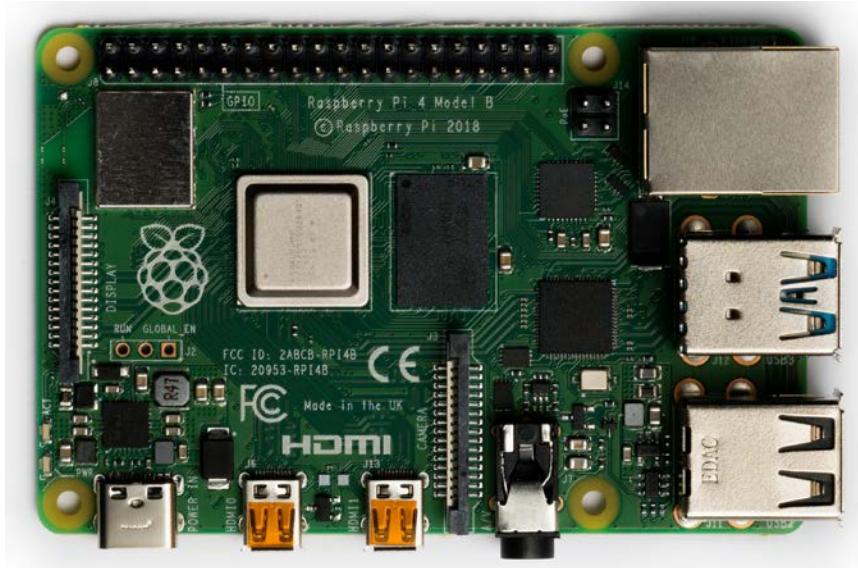


Figure 1.7 – The top view of the Raspberry Pi 4B

Here is a photograph of the model at an angle:



Figure 1.8 – Raspberry Pi 4B at an angle

We are going to use the 4GB variant of this model.

Raspberry Pi Zero W

You can find the specifications of the Raspberry Pi Zero W at <https://www.raspberrypi.org/products/raspberry-pi-zero-w/>.

The following table explains the specifications of this model in more detail:

Component	Specification
Processor	Broadcom BCM2835 (32-bit) ARMv6Z1 GHz Single core ARM1176JZF-S CPU
RAM	512 MB
Network Connectivity	802.11 b/g/n wireless LAN Bluetooth 4.1 Bluetooth Low Energy (BLE)
USB	One Mini USB port
GPIO Pins	Raspberry Pi standard 40 pin GPIO header, which is backward compatible
Video Output	Mini HDMI
Data Storage	Micro-SD card slot for loading operating system and data storage
Power supply	2A 5V DC power through micro-USB connector or GPIO header
Camera	CSI camera connector
Audio	4-pole stereo audio
Graphics	Broadcom VideoCore IV @ 250 MHz

Figure 1.9 – Product specification list of the Raspberry Pi Zero W

Where can you buy these models?

You can discover where to buy Raspberry Pi boards and their accessories on the products page of the RPi website. Here's a screenshot:



Figure 1.10 – Buying a Raspberry Pi

You can also find Raspberry Pi boards and their accessories on Amazon. If you live in a big city, then you can find a lot of hobby electronics stores that sell Raspberry Pi boards and related items.

OSes for Raspberry Pi

Many OSes have tailored distributions of OSes for Raspberry Pi boards. However, the early board models do not support all OSes. The latest model board, Raspberry Pi 4B, supports all the OSes mentioned at <https://www.raspberrypi.org/downloads/>.

The Raspbian OS supports all the models of the Raspberry Pi board, and it is the most recommended OS for beginners. We are going to demonstrate how to install this in the next section.

Raspbian is a free OS based on Debian, which is a popular distribution of Linux. Raspbian is optimized to Raspberry Pi hardware. You can find more information about the Raspbian project on its home page (<http://raspbian.org/>).

Note

Raspbian's home page mentions that it is not affiliated with the Raspberry Pi Foundation and is managed by fans of the Raspberry Pi and Debian projects.

The Raspbian web page provides a list of recommended Raspbian images at <http://raspbian.org/RaspbianImages>. An OS image is a file that can be written onto an SD card, and this SD card can then be used to boot the Raspberry Pi board. This is the easiest way of getting started with RPi; we will try to use it from now on. The image provided on the RPi Foundation's download page is the one most recommended by Raspbian. We will learn how to use this image to get started with RPi in the next section.

Setting up Raspbian on a Raspberry Pi

The setup is the one thing that usually deters many novice enthusiasts from getting started with SBCs. Many times, the instructions are very generic and do not cover all the cases for various types of hardware components. That is why I have dedicated an entire section to the setup of Raspbian on RPi. In this section, we will demonstrate the setup in detail with all the board models ever produced, with the exception of the compute modules.

We need the following components for the setup:

- A Raspberry Pi board of any model.
- If you have a Raspberry Pi 4B board, then you will need a power supply of 5V 3A with a USB Type-C pin. Here is a photograph of a USB Type-C pin:



Figure 1.11 – USB Type-C pin

- To be on the safe side, you might want to purchase the official **Raspberry Pi 15.3W USB-C power supply** by the Raspberry Pi Foundation. The URL for this product is <https://www.raspberrypi.org/products/type-c-power-supply/>.

- For all the other models of Raspberry Pi, a 5V 2.5A power supply with a Micro-USB type pin should be compatible. Here's a photograph of a Micro-USB pin:



Figure 1.12 – A Micro-USB pin

- You might want to purchase a **Raspberry Pi Universal Power Supply** (<https://www.raspberrypi.org/products/raspberry-pi-universal-power-supply/>) for this purpose.
- A USB keyboard and mouse: It is a good idea to purchase a USB keyboard with an integrated mousepad, as follows:



Figure 1.13 – A keyboard with an integrated mousepad

- For RPi Zero and RPi Zero W, the keyboard with a mousepad is mandatory because these board models have only one Micro-USB type of connector to the peripherals interface. Additionally, for RPi Zero and RPi Zero W, we need a USB to Micro-USB OTG converter, as follows:



Figure 1.14 – USB OTG cable

- Raspberry Pi boards of any model work with any microSD card. The guidelines say that we should use a class 10 microSD card with a minimum of 16 GB. You might want to visit <https://www.raspberrypi.org/documentation/installation/sd-cards.md> for guidelines, and https://elinux.org/RPi_SD_cards for a compatibility list. RPi 1 Model A and RPi 1 Model B use SD cards. Therefore, it is better to have a microSD to SD card adapter, as follows:



Figure 1.15 – MicroSD to SD card adapter/converter

- An HDMI monitor or a VGA monitor for visual display.
- All RPi board models, except RPi 4B, RPi Zero, and RPi Zero W, have an HDMI output and can be directly connected to the HDMI monitor with an HDMI male-to-male cable:



Figure 1.16 – HDMI cable

RPi 4B has a micro-HDMI output. Therefore, we need a micro-HDMI to HDMI converter. RPi Zero and RPi Zero W both have mini-HDMI outputs. So, for them, we need a mini-HDMI to HDMI converter. The following photograph shows the HDMI, mini-HDMI, and micro-HDMI ports, respectively:



Figure 1.17 – HDMI, mini-HDMI, and micro-HDMI ports

We also need to plug the mini- and micro-HDMI ends to the RPi boards and the HDMI to the monitor. If you are planning to use a VGA monitor, then we will need HDMI/mini-HDMI/micro-HDMI to VGA converters depending on the board models.

Here is a photograph of an HDMI to VGA converter:



Figure 1.18 – HDMI to VGA converter

The following is a photograph of a mini-HDMI to VGA converter:



Figure 1.19 – Mini-HDMI to VGA converter

The following is a photograph of a micro-HDMI to VGA converter:



Figure 1.20 – Micro-HDMI to VGA converter

We need a Windows computer and a wired or wireless internet connection.

Finally, we require an SD card reader, as follows:



Figure 1.21 – SD card reader

Many laptops have this (SD card reader) feature built-in. So, in that case, a separate reader is not required as we can use the built-in reader.

We will need a few more hardware components by the end of the chapter. We will discuss them when the need arises. For now, we are okay to proceed further.

Downloading the necessary software

To get started, we need to download all of the free software. Follow these instructions to download all the necessary software:

1. We need the latest image file of the Raspbian OS. This can be downloaded from the download page of the Raspberry Pi Foundation website at <https://www.raspberrypi.org/downloads/raspbian/>. The following screenshot shows the various options that are available for download:

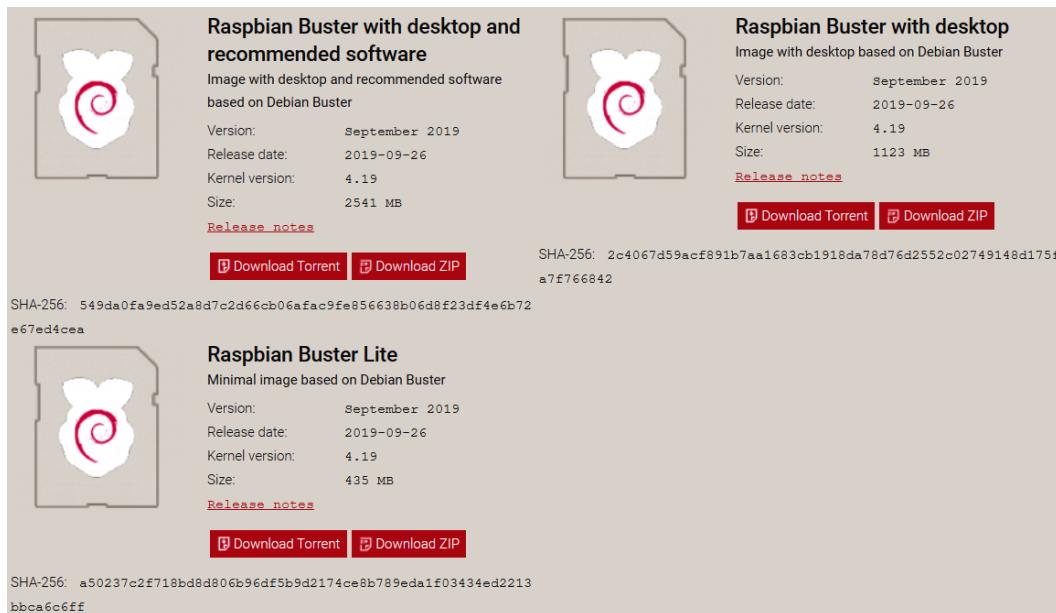


Figure 1.22 – Raspbian image download page

2. By the time you visit the URL, the page might have been updated, but the download options will usually remain the same. The first option is **Raspbian Buster with desktop and recommended software** and is the most recommended for beginners. The second option is **Raspbian Buster with desktop**. The third option is **Raspbian Buster Lite** and comes with the bare minimum software; it has the smallest size among all the download options.

3. We can either download the ZIP file directly, or we can download the torrent file of the image. I recommend downloading the torrent file. Once the torrent file for **Raspbian Buster with desktop and recommended software** is downloaded, we can download the torrent software from <https://www.bittorrent.com/>. Download the free classic version and install it on your PC. Then, open the torrent file with BitTorrent and the download will begin. The following is a screenshot of a finished download:

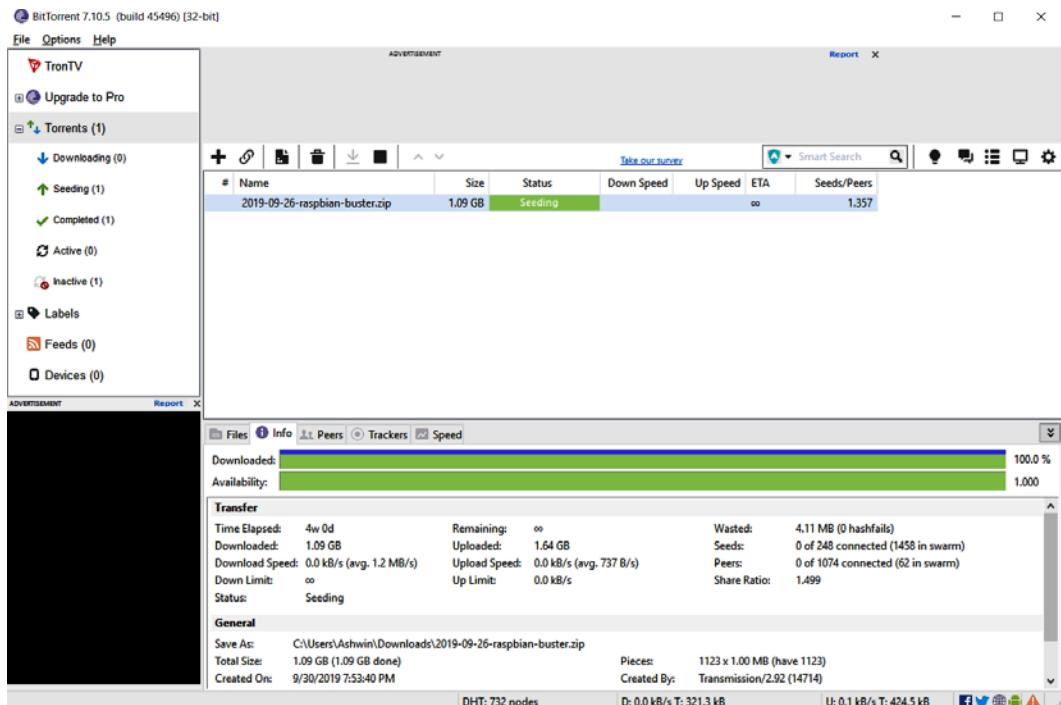


Figure 1.23 – BitTorrent application window

4. At the bottom of the screen, we can see the location of the download. Additionally, we can right-click on the finished installation and click on the **Open Containing Folder** option, as follows:

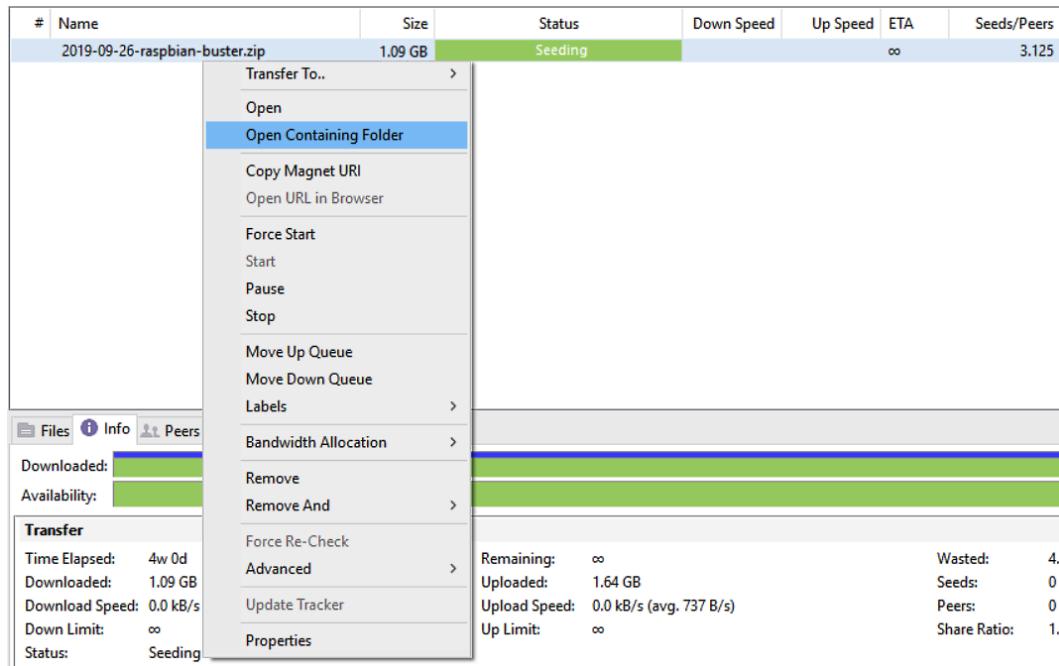


Figure 1.24 – Opening the location of the downloaded image

This will open the folder that has the ZIP file for the Raspbian OS image.

5. We require software for unzipping the file. **7-Zip** is the free and open source software for this. We can download the appropriate installable file (32-bit x86 or 64-bit x64) and install it. Once the installation is complete, open the ZIP file using the software. The following is a screenshot of 7-Zip:

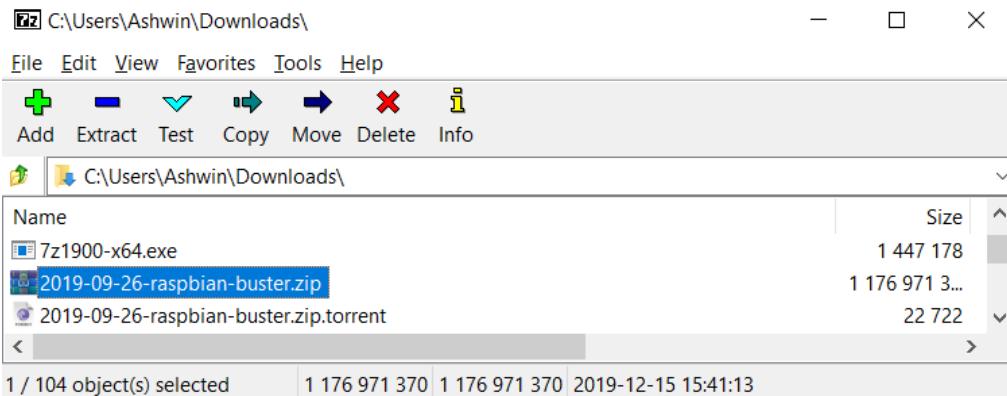


Figure 1.25 – 7-Zip application window

Double-click on the ZIP file and then click on the **Extract** button in the menu. This will extract the file. The extracted file has the `.img` extension.

6. We require software to write this image to the microSD card and **Win32DiskImager** is the perfect software for the task. Download it from <http://sourceforge.net/projects/win32diskimager/files/latest/>. Run the installation file and install it.

Preparing the microSD card manually

The best way of installing an OS on a microSD card is to do it manually. This allows us to prepare the SD card manually so that we have easier access to the `/boot/config.txt` configuration file, which must be modified, in a few cases, before booting up the RPi. We will discuss this in detail later. The default Raspbian image has only two partitions—**boot** and **system**. I recommend choosing, at a minimum, a 16 GB class 10 microSD card. Then, follow these steps:

1. Unpack the fresh microSD card and insert it into the card reader. Plug the card reader into your Windows laptop or computer. Many laptops and computers come with an SD card reader. For these, insert the microSD card into the microSD to SD card adapter, and insert the adapter into the slot for the SD card reader of the computer or laptop.

2. Then, a new drive will appear in the left-hand panel of **Windows File Explorer**. Right-click on the drive and choose **Format**. Here is a screenshot of the **Format** window:

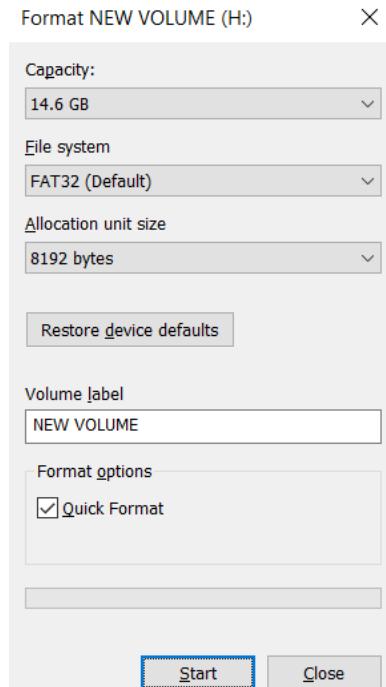


Figure 1.26 – Formatting the microSD card

3. Make sure that you check the **Quick Format** checkbox. Then, click on the **Start** button. It will show a warning message, as follows:

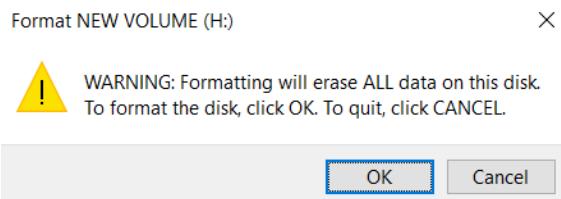


Figure 1.27 – Dialogue box for confirmation

4. Click on the **OK** button to finish formatting.

- Once the formatting is complete, we need to write the Raspbian OS image file to the microSD card. Open Win32DiskImager and choose the Raspbian OS image file, as shown in the following screenshot:

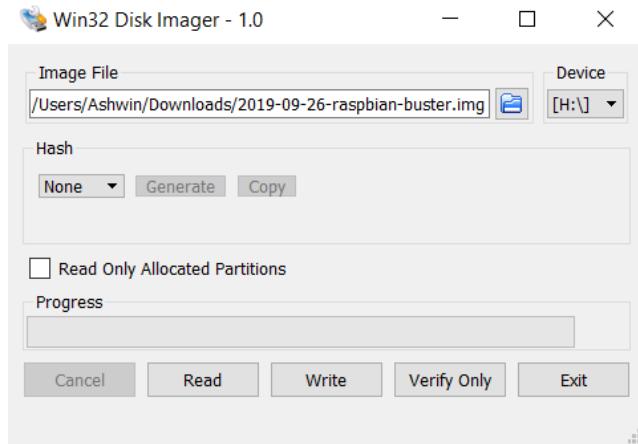


Figure 1.28 – The Win32 Disk Imager application window

- Then, click on the **Write** button. It will show the following warning box. Simply click on the **OK** button:

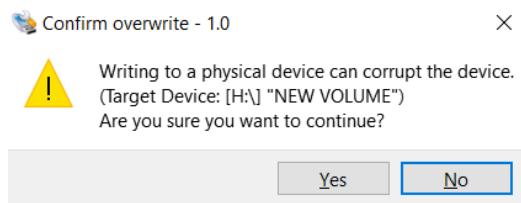


Figure 1.29 – Dialogue box to confirm writing of the image to the microSD card

- Once the OS is successfully written to the SD card, it shows the following message box:

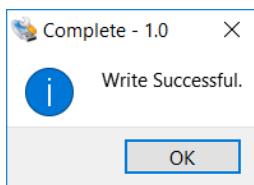


Figure 1.30 – Confirmation message box

This means that the image has been successfully written to the microSD card. Now we can use it to boot up the RPi.

8. Now, this step is necessary only if you are using a VGA monitor and not the HDMI monitor. Readers using the HDMI monitor can safely ignore this step. The microSD card's **BOOT** partition can be accessed using **Windows File Explorer**. It has the config.txt file. Double-click and open the file. We must edit the settings in the /boot/config.txt file, as follows, to enable a proper display on the VGA monitor:
 - a) Change #disable_overscan=1 to disable_overscan=1.
 - b) Change #hdmi_force_hotplug=1 to hdmi_force_hotplug=1.
 - c) Change #hdmi_group=1 to hdmi_group=2.
 - d) Change #hdmi_mode=1 to hdmi_mode=16.
 - e) Change #hdmi_drive=2 to hdmi_drive=2.
 - f) Change #config_hdmi_boost=4 to config_hdmi_boost=4.
 - g) Save the file.

The commented lines (that have # at the beginning) are disabled. We must enable these lines by uncommenting them. This can be done by removing # at the beginning of these commented lines.

Note

If you are using Linux or macOS, then you will find the instructions to install the Raspbian OS on your microSD card for these OSes at <https://www.raspberrypi.org/documentation/installation/installing-images/>.

Booting up the Raspberry Pi for the first time

Let's boot up our Pi for the first time with the microSD card using the following steps:

1. Insert the microSD card into the microSD card slot of Pi. RPi 1 Model A and RPi 1 Model B do not have slots for an SD card. So, for these board models, we must use a microSD to SD card converter.
2. Connect the Pi to the HDMI monitor. As discussed earlier, in case you have a VGA monitor, connect it using the HDMI/mini-HDMI/micro-HDMI to VGA converter.

3. Connect the USB mouse and USB keyboard. It is recommended that you have a single keyboard with a mousepad. For RPi Zero and RPi Zero W, you need to first connect it to a USB OTG cable, and then connect the USB OTG cable to the board.
4. Connect the RPi board to an appropriate power supply. Connect the monitor to the power supply. We need to make sure that the power is switched off at this point.
5. Ensure you verify all the connections once. Then, turn on the power supply of the monitor. Finally, turn on the power supply for the RPi.

Now, our RPi board will start booting up. The green LED on the board will start blinking. Congratulations! The RPi board is booting for the first time.

Note

If your HDMI monitor is showing no signal, then power down the RPi and change `#hdmi_force_hotplug=1` to `hdmi_force_hotplug=1` in `/boot/config.txt` on the microSD card. Boot up the RPi with this changed setting and the HDMI monitor will show the signal.

Once the RPi boots up, the Raspbian desktop and a guided setup window appear, as follows:



Figure 1.31 – Welcome window on Raspbian

Click on the **Next** button, and the following window will appear:



Figure 1.32 – Window for setting the country

In the preceding window, set **Country:** and **Language:**. It will automatically select the time zone according to the country you selected. You can change that too if you wish. Click on the **Next** button, and the following window will appear:

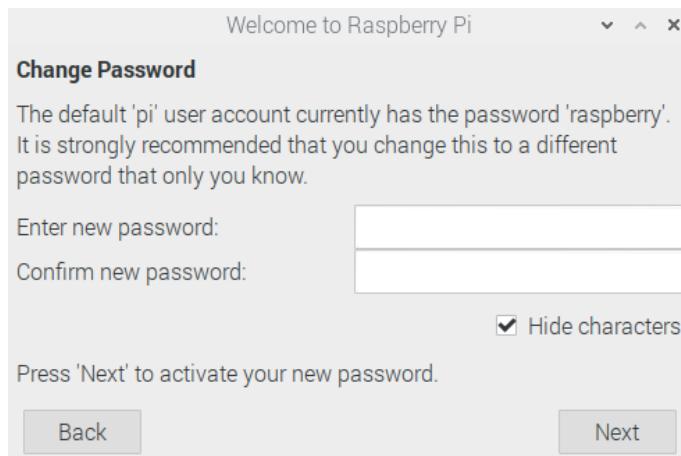


Figure 1.33 – Window for setting a new password

You can choose to set a new password for the default pi user. If you leave it blank, then it will retain the default password. The following is the next window that appears:

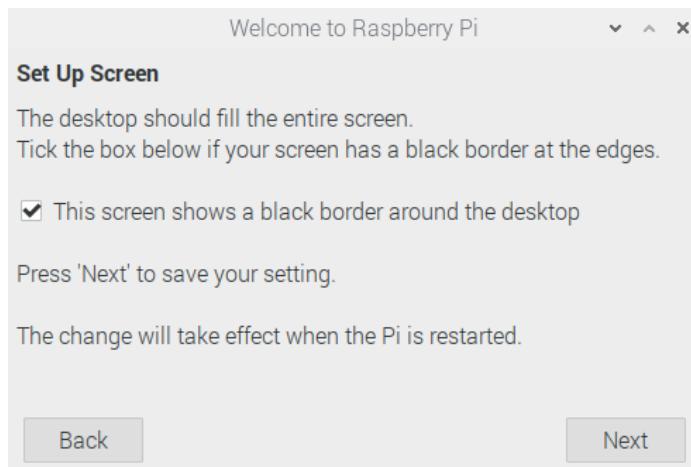


Figure 1.34 – Window for setting up the screen

Check the checkbox if there are black borders on the edges of the desktop view. The Raspbian OS will rectify it upon the next boot. The following window will appear after you click on the **Next** button, but only if the board model has Wi-Fi:

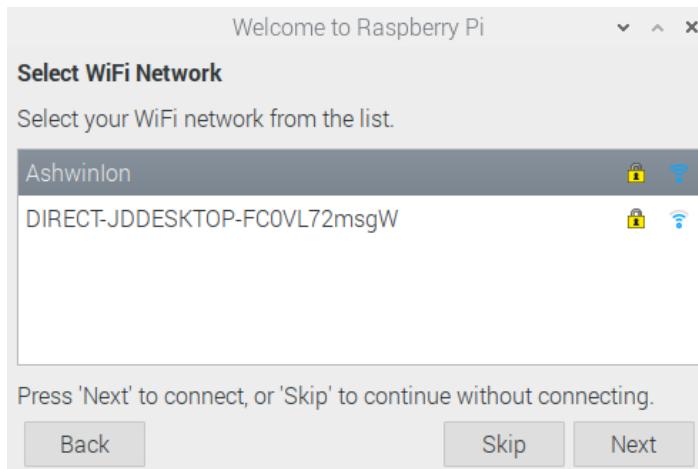


Figure 1.35 – Wi-Fi connections

Choose the network that you know the credentials for, and click on the **Next** button. The following window will appear:

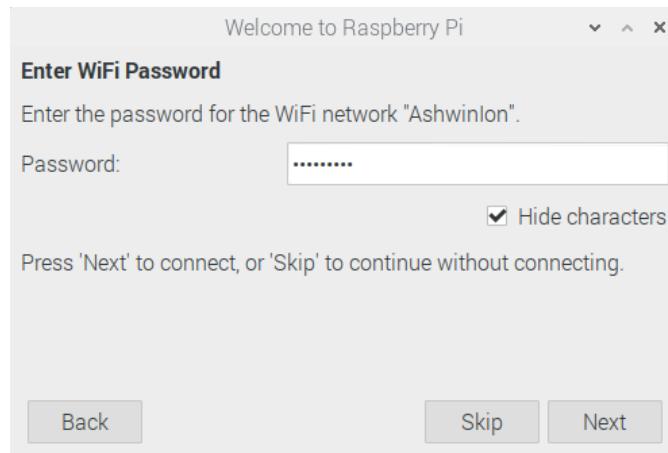


Figure 1.36 – Connecting to the Wi-Fi at my home

Key in your Wi-Fi password here, and click on the **Next** button. The following window will appear:

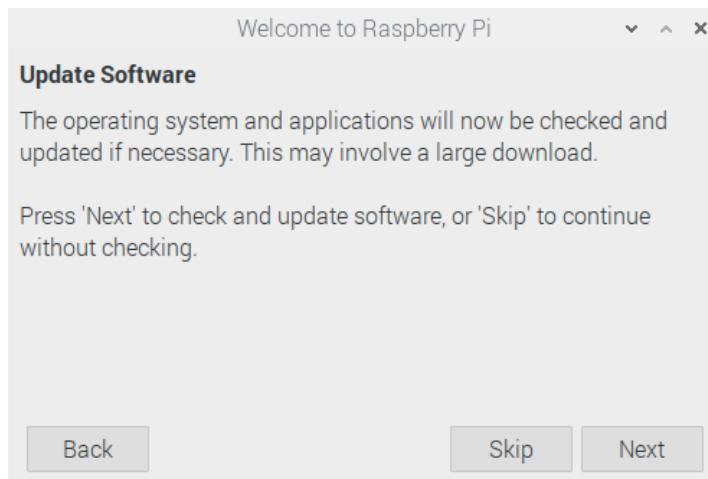


Figure 1.37 – Update Software

We can update the Raspbian OS and installed software here. We are going to learn how to do it manually in the latter part of this chapter. Click on the **Skip** or **Next** button, and the following window will appear:

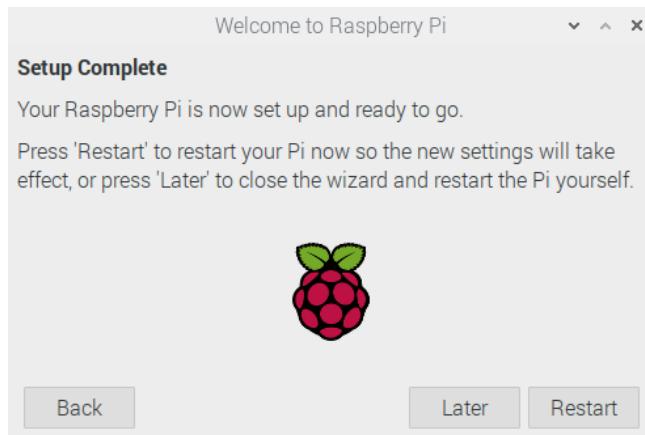


Figure 1.38 – Confirmation of completing the initial setup

We have finished most of the setup. Now, there are a few more things to do before we reboot our RPi, so click on the **Later** button.

Now, in the top-left corner of the desktop, you should see a Raspberry icon. It is the menu for Raspbian and functions in a similar way to the Windows logo on Microsoft Windows. Click on the logo and navigate to **Preferences | Raspberry Pi Configuration**:



Figure 1.39 – Raspberry Pi Configuration in the Raspbian menu

This is the **Raspberry Pi Configuration** tool. It will open a window as follows, and we can change the settings of the Raspberry Pi board:

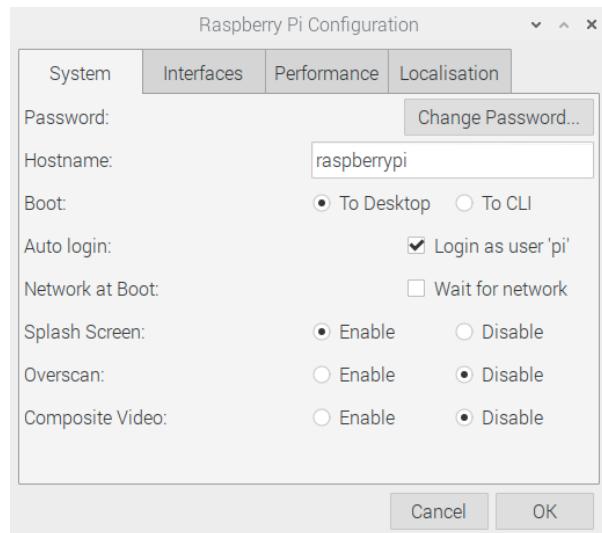


Figure 1.40 – Configuring the system

The preceding screenshot is the **System** tab. As of now, there is no need to change anything here. The following is the **Interfaces** tab:

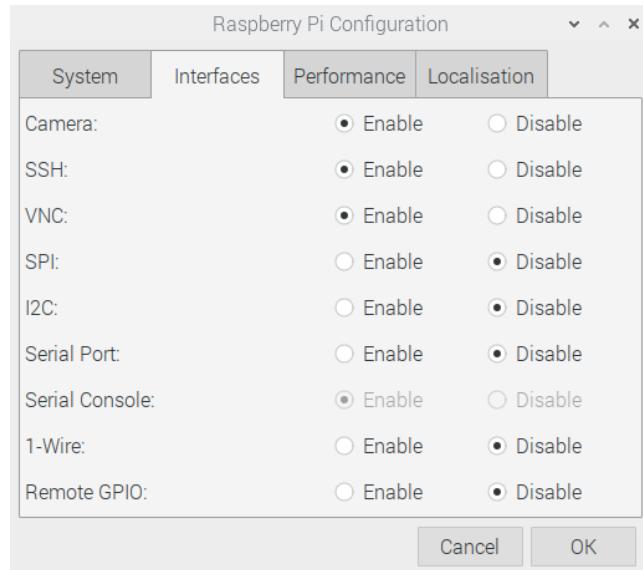


Figure 1.41 – Configuring interfaces

Enable the camera, SSH, and VNC. The following is the **Performance** tab:

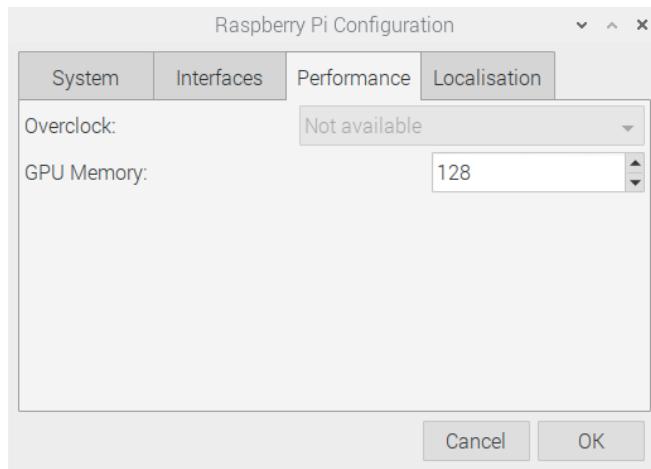


Figure 1.42 – Memory and Overclock options

This menu has an option for overclocking and GPU memory. For the RPi 4B, overclocking is disabled. We will learn how to overclock an RPi 4B board manually in the next chapter.

The **Localisation** tab is as follows:

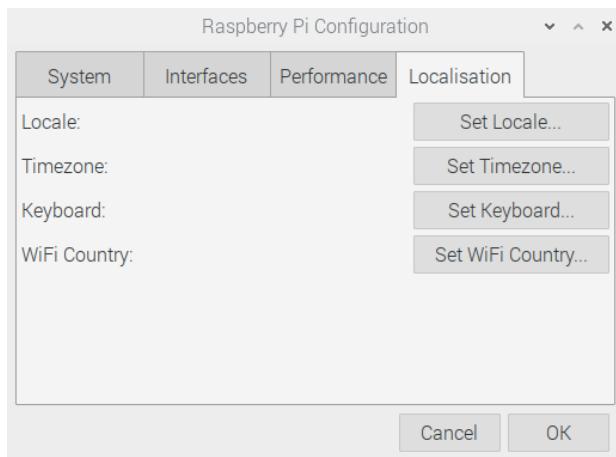


Figure 1.43 – Localisation options

You might want to change these settings as per your region of residence.

Once all these settings have been changed as per our choice, we can restart the RPi board by clicking on the **Shutdown** button in the **Raspbian** menu:



Figure 1.44 – Rebooting the Pi

Here, we find the option to reboot the RPi. Once we reboot, and if we have chosen to retain the original password for the default user, `pi`, the following warning message window will appear when booting up:

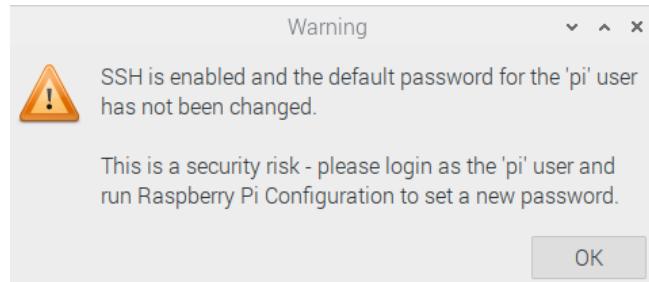


Figure 1.45 – Message after rebooting if the default password has not been changed

And this will keep on appearing after every boot as long as we choose to retain the default password.

Connecting various RPi board models to the internet

We can directly plug in the Ethernet cable to the RJ45 Ethernet port Pi boards. This will automatically detect the connection and connect to the internet.

Note

Make sure that **DHCP (Dynamic Host Configuration Protocol)** is enabled at the Wi-Fi router, the managed switch, or the internet gateway.

PRi 1 A, PRi 1 A+, RPi Zero, RPi Zero W, and RPi 3 A+ do not have Ethernet ports. However, RPi Zero W and RPi 3 A+ have built-in Wi-Fi. We can use a USB Wi-Fi dongle for the remaining models:



Figure 1.46 – USB Wi-Fi adapter

Plug this Wi-Fi adapter into the USB port. If the USB ports are not enough, then use a powered USB hub. For Raspberry Pi Zero, we need to use an additional USB OTG cable, as discussed earlier.

After plugging in the USB Wi-Fi adapter, we need to open LXterminal. This is the command-line utility. We can find it as a small black icon in Raspbian's taskbar and under **Accessories** in the Raspbian menu. Once we click on it, the following window will appear:

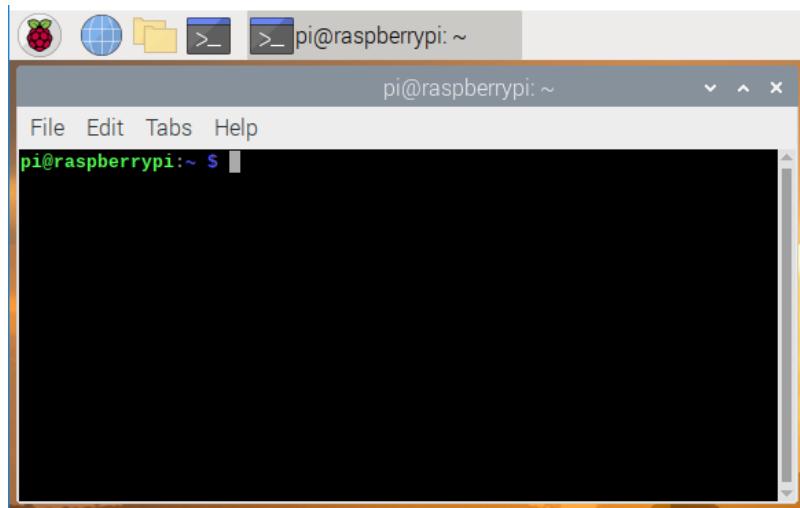


Figure 1.47 – Raspberry Pi LXterminal window

We can type in the Linux commands here. After typing them in, press *Enter* to execute the command. We have opened this so that we can manually configure the network interface of Raspbian. It is quite easy. All the network-related information is stored in the `/etc/network/interfaces` file. To connect to the Wi-Fi after plugging in the USB Wi-Fi dongle, we need to add a few entries to this file. First, take the backup of the original file by executing the following command:

```
mv /etc/network/interfaces /etc/network/interfaces.bkp
```

Then, we can create the `interfaces` file from scratch by running the following command:

```
sudo nano /etc/network/interfaces
```

The preceding command will open the network interface's file with a plain text editor known as `nano`. It is a simple WYSIWYG editor. Enter the following lines there:

```
source-directory /etc/network/interfaces.d
auto lo
iface lo inet loopback

auto wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
wpa-ssid "AshwinIon"
wpa-psk "internet1"
```

After entering the lines, press *Ctrl + X* and then press *Y*. In the preceding settings, substitute `AshwinIon` with your own SSID and `internet1` with a password for the same. Then, run the following command in Command Prompt:

```
sudo service networking restart
```

This restarts the networking service and connects to the Wi-Fi. In any case (Ethernet or Wi-Fi), the RPi is assigned with a unique IP address. We can find it out by running the `ifconfig` command at `lxterminal`. The output of the command will have the `Ipv4` address listed under `inet`.

Another way to know the IP address of the RPi is to check the active client tables in the router or the managed switch that the RPi board is connected to. The following is a screenshot of my router's active client table where we can see an entry for RPi:

	Setup	Wireless	Advanced	Maintenance	Status
Device Info					
Active Client Table					
This table shows IP address, MAC address for each client.					
Active Wired Client Table					
	Name	IP Address	MAC Address		
	realme-2-Pro	192.168.2.2	50:29:f5:9d:bf:c1		
	DESKTOP-FC0VL72	192.168.2.5	d4:6e:0e:11:b2:ea		
	raspberrypi	192.168.2.6	7c:dd:90:00:e2:1e		
Active Wireless Client Table					
	Name	IP Address	MAC Address		
	realme-2-Pro	192.168.2.2	50:29:f5:9d:bf:c1		
	DESKTOP-FC0VL72	192.168.2.5	d4:6e:0e:11:b2:ea		
	raspberrypi	192.168.2.6	7c:dd:90:00:e2:1e		
<input type="button" value="Refresh"/>					

Figure 1.48 – Active client table of a home Wi-Fi router

Updating the RPi

Advanced Package Tool (APT) is a package management utility in Debian, Ubuntu, Raspbian, and their derivatives. APT is used to install, upgrade, and remove software. We will learn how to use it to update the OS and the software on the RPi board.

Run the following command:

```
sudo apt-get update
```

This command synchronizes the package list from the online source repository of software. Indexes of all the packages are refreshed. This updates all the repositories of the apps to all the latest update lists. This command must be executed before we execute the upgrade command.

Then, run the following command:

```
sudo apt-get dist-upgrade -fix-missing -y
```

This downloads and installs all the packages. It also removes obsolete packages. Depending on the speed of the internet, it takes some time. Finally, update the firmware by running the following command:

```
sudo rpi-update
```

This will update the firmware. Following this, the RPi board will be up to date in all aspects. Finally, we can run the following command to shut down the RPi:

```
sudo shutdown -h now
```

And the following command reboots it:

```
sudo reboot
```

This will update the firmware. Following this, the RPi board will be up to date in all aspects.

Summary

In this chapter, we learned important terms such as computer vision, OpenCV, SBCs, and Raspberry Pi. We learned how to set up a Raspbian OS on Raspberry Pi and how to configure a Pi to access the internet. We also learned how to update a Pi.

With the completion of this chapter, you can go ahead and set up the Raspbian OS on your Raspberry Pi. Additionally, you can connect your RPi board to the internet using Wi-Fi or Ethernet. This will make you ready for the computer vision adventure that will soon follow.

In the next chapter, you will learn how to remotely access a RPi, how to overclock it, and the installation of OpenCV 4 for Python 3 on an RPi.

2

Preparing the Raspberry Pi for Computer Vision

In the previous chapter, we learned the fundamentals of single-board computers, computer vision, and OpenCV. We learned the detailed specifications of the **Raspberry Pi (RPi) 4B** and the RPi Zero W. We also learned how to set up Raspbian OS on all the RPi boards models in detail.

In this chapter, we will learn how to prepare our RPi board for computer vision. Continuing from where we left off in the previous chapter, we will start by installing the OpenCV library for computer vision and the other necessary software for remotely accessing the desktop, as well as Command Prompt. We will learn how to transfer files between an RPi and a Windows PC. We will also learn how to exploit the computation power of the RPi by overclocking it and installing a heatsink on it to reduce the temperature of the processor.

The topics that we'll cover in this chapter are as follows:

- Remotely logging into the RPi with SSH
- Remote desktop access
- Installing OpenCV on an RPi board
- Heatsinks and overclocking the RPi 4B

Remotely logging into the RPi with SSH

We can remotely access the Command Prompt of the RPi board using various software from Windows. We can run all the Linux commands that do not involve the GUI remotely from Windows. As you may recall, we discussed how to enable SSH with the Raspberry Pi Configuration tool in *Chapter 1, Introduction to Computer Vision and Raspberry Pi*. It enables remote login through SSH.

In order to get started, follow these steps:

1. First, we need to install any SSH software available for free. The most popular is PuTTY (<https://www.putty.org/>). I prefer to use another popular SSH client that comes with SFTP known as the Bitvise SSH client. You can download the installation file for Windows from <https://www.bitvise.com/ssh-client-download> and install it. After doing that, open the Bitvise SSH client. The following window will appear:

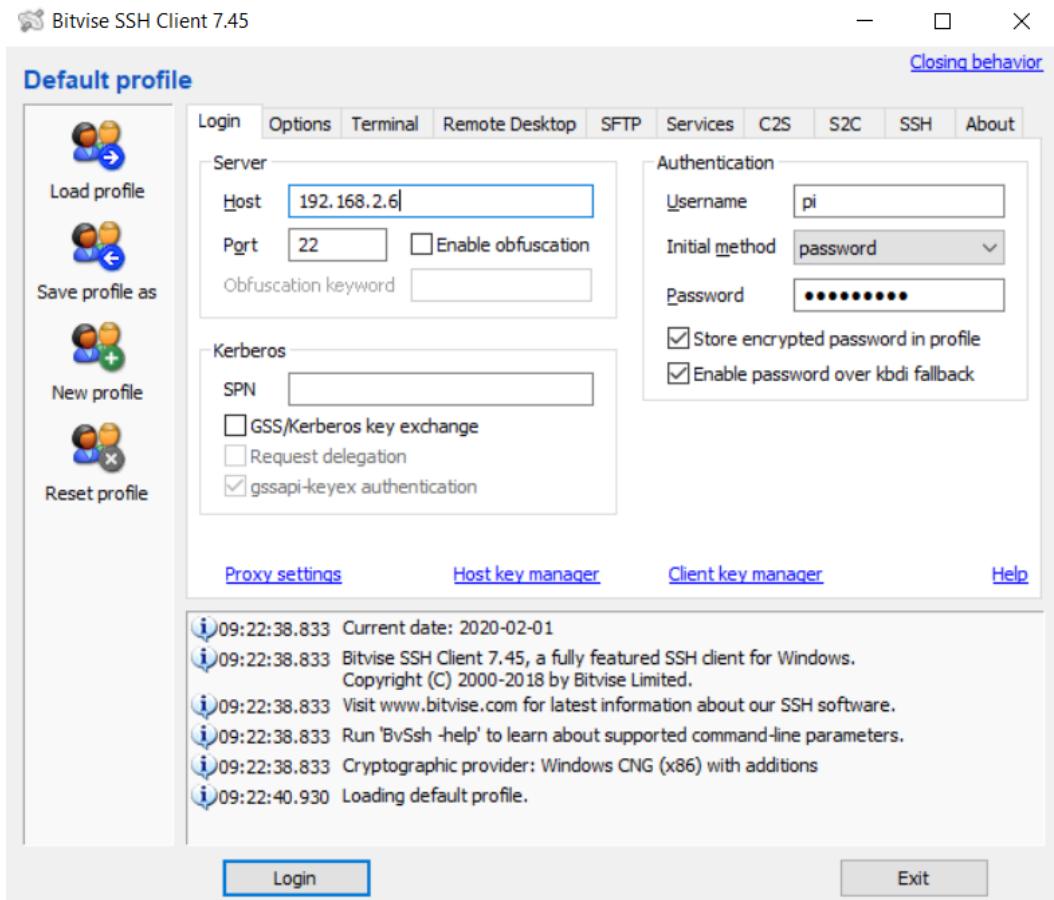


Figure 2.1 – Bitwise Connection window

Enter a hostname, username, and password. The hostname is nothing but the IPv4 address of our RPi board, which we learned how to find in *Chapter 1, Introduction to Computer Vision and Raspberry Pi*.

2. After entering all the necessary information, click the **Login** button. This will start the RSA key exchange and display the following message box:

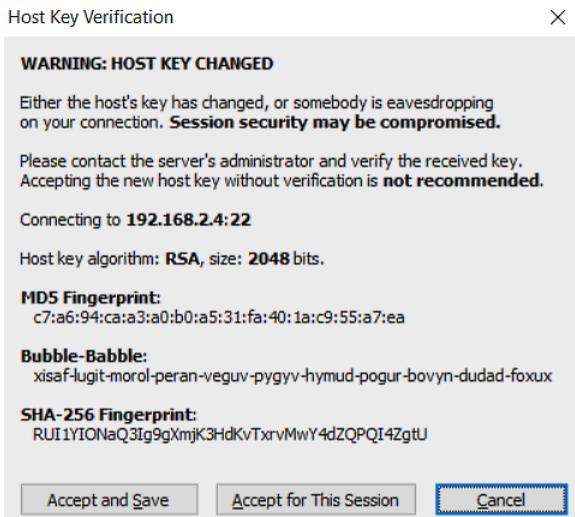


Figure 2.2 – Message window for the first-time connection

3. Click the **Accept and Save** button. This will save the exchanged RSA keys. Note that this message box won't be displayed if we try to connect to Raspberry Pi again with the same Windows computer. After that, two separate windows will appear. The first is the Command Prompt of the Raspberry Pi. Just like LXTerminal, we can run Linux commands from here too:

A screenshot of a Bitvise xterm window titled 'pi@192.168.2.4:22 - Bitvise xterm - pi@raspberrypi: ~'. The window contains the following text:

```
Last login: Sat Feb 1 09:45:06  
  
SSH is enabled and the default  
This is a security risk - pleas  
  
pi@raspberrypi:~ $
```

The text 'pi@raspberrypi:~ \$' is highlighted in green, indicating the prompt for input.

Figure 2.3 – Bitwise SSH window

4. We can change the font and size of the text that appears here by changing the properties, which can be found by right-clicking the title bar. The following is the file transfer window:

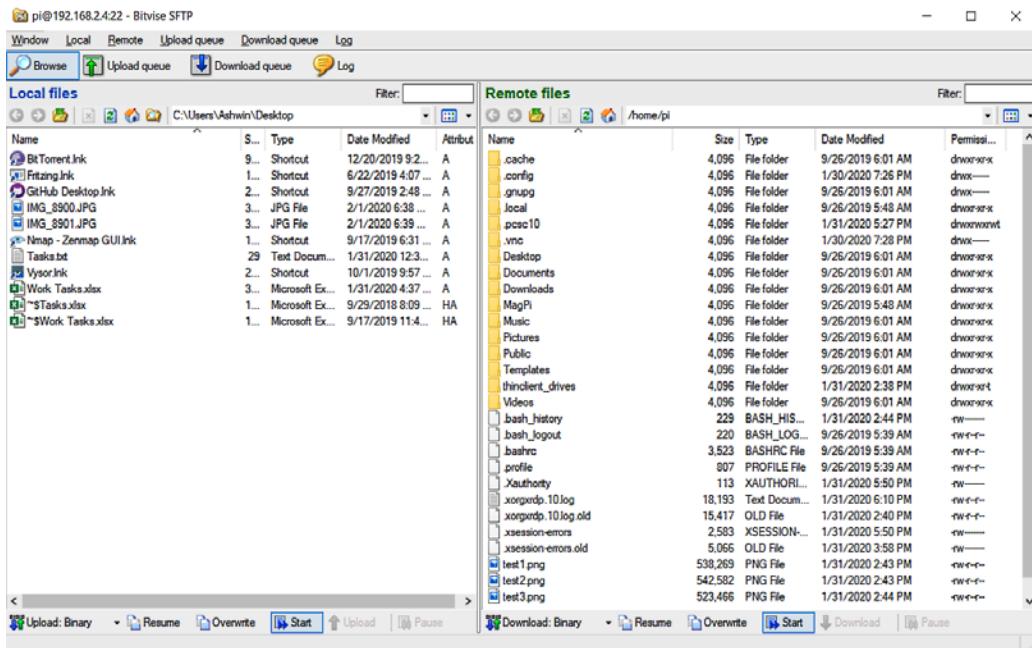


Figure 2.4 – Bitwise FTP file transfer window

On the left-hand pane, we have the Windows desktop and on the right-hand pane, we have `/home/pi`, the home directory of the `pi` user. We can transfer files between Windows and the RPi just by dragging and dropping them between these panes.

NOTE

We can access the Raspberry Pi Configuration Tool from the Command Prompt using the `sudo raspi-config` command. This is the command-line version of the tool.

This is how we can connect to the Command Prompt of the Raspbian OS remotely and transfer files. Next, we will learn how to remotely access the Raspbian OS desktop.

Remote desktop access

The Bitvise SSH client is great for file transfers and accessing the Command Prompt terminal of RPi. However, we need to use another piece of software to access the desktop of RPi remotely. There are two methods we can follow. The first one is VNC (we learned how to enable it in *Chapter 1, Introduction to Computer Vision and Raspberry Pi*, using the Raspberry Pi Configuration tool), while the other is using Windows' built-in **Remote Desktop Connection** utility. We can find it in the Windows search bar, as follows:

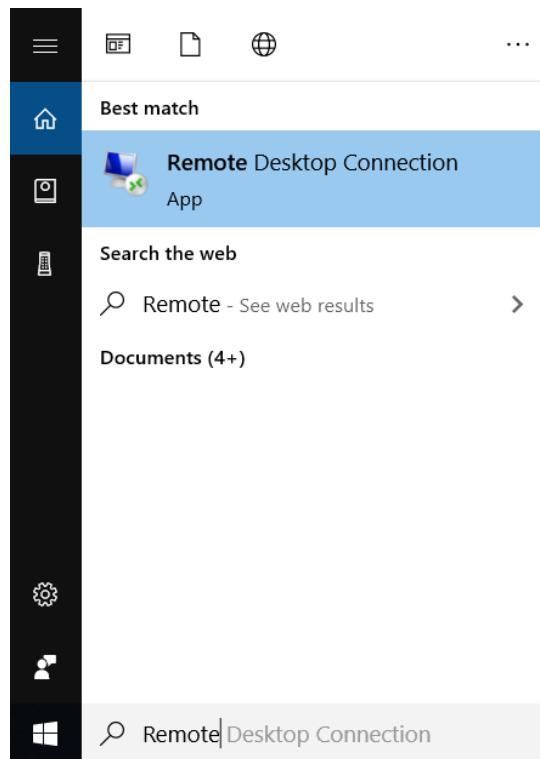


Figure 2.5 – Remote Desktop Connection option in the Windows search bar

But before we can use it, we need to install `xrdp` on the RPi. Installing it is very easy. We just need to run the following command at LXTerminal on the RPi:

```
sudo apt-get install xrdp -y
```

Information

You might want to read more about `xrdp` at <http://xrdp.org/>.

Once xrdp has been installed on the RPi, you need to follow these steps:

1. Open the **Remote Desktop Connection** application on your Windows PC:

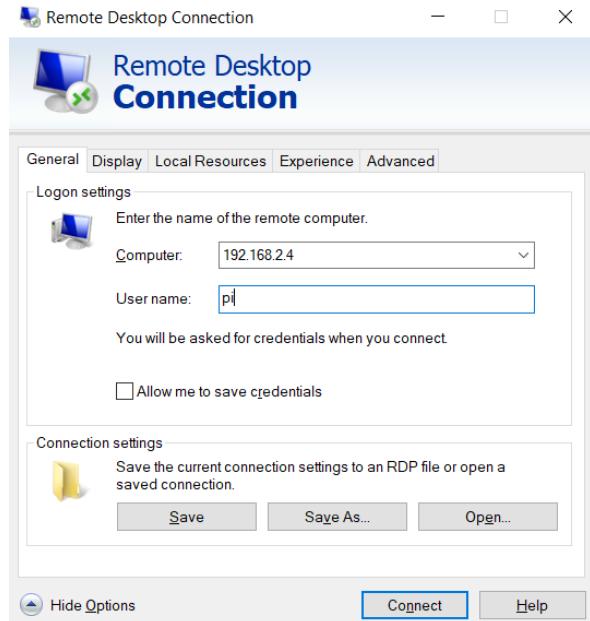


Figure 2.6 – Remote Desktop Connection

2. Enter the IP address and pi in the textboxes labeled **Computer** and **User name**. You might want to check the checkbox for **Allow me to save credentials** and save the connection settings too. Once we click the **Connect** button, the following window will appear:

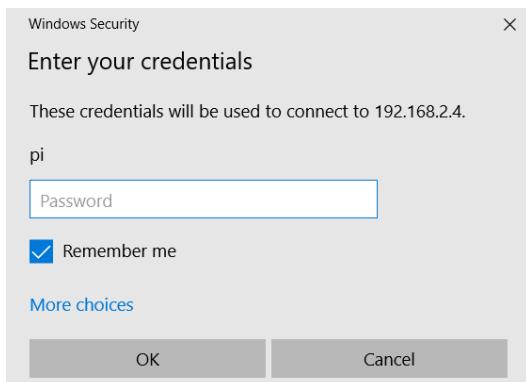


Figure 2.7 – Credentials for the Raspbian OS for Remote Desktop Connection

3. Enter the password and check the checkbox if you want to save the password for this connection. Click the **OK** button; the RPi remote desktop window will appear after a few moments. If you have less traffic on your LAN, then the working of Remote Desktop will be smooth. The following is a screenshot of the Remote Desktop window:

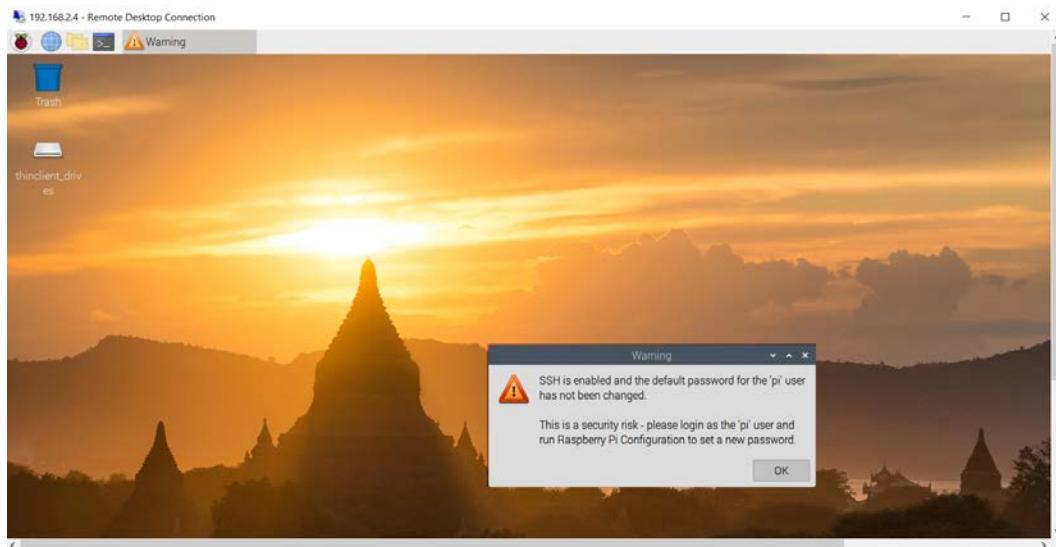


Figure 2.8 – Raspbian OS Remote Desktop

We can perform all the tasks related to the GUI from here. This means we don't need a separate display for the RPi board if we use Remote Desktop.

Installing OpenCV on an RPi board

Follow these steps to install OpenCV on the RPi:

1. First, we need to install a few dependencies. Run the following command to install all these dependencies:

```
sudo apt-get install -y libhdf5-dev libhdf5-serial-dev  
libatlas-base-dev libjasper-dev libqtgui4 libqt4-test
```

2. Once the installation is successful, we can install OpenCV on the RPi:

```
pip3 install opencv-python==4.0.1.24
```

- Once the installation of OpenCV is successful, we can verify it by running the following command:

```
python3 -c "import cv2; print(cv2.__version__)"
```

The following should be the output:

```
4.0.1
```

This means that the installation is completed and that we can import OpenCV in our Python 3 programs.

Next, we will learn how to overclock the RPi 4B and how to install heatsink on it.

Heatsinks and overclocking RPi 4B

Overclocking means running the processors at higher speeds than those that are intended. When we overclock the processors, their temperature tends to rise and they radiate more heat. Raspberry Pi board models do not come with any built-in coolers. You can buy passive heatsinks from many online shops such as Amazon. The following is an example of a heatsink with a fan:



Figure 2.9 – Small heatsink for RPi

The heatsink fan can be powered by connecting it to a 5V or 3.3V power supply. The speed of the fan depends on the voltage, and we can connect it to the RPi power pins. We will learn more about the GPIO and the power pins of RPi in the next chapter. The best and the most effective heatsink that I found was the ICE Tower fan for the RPi 4B (<https://www.seeedstudio.com/ICE-Tower-CPU-Cooling-Fan-for-Raspberry-pi-Support-Pi-4-p-4097.html>).

The following is my own Pi with the ICE Tower mounted on it:



Figure 2.10 – ICE Tower installed on Raspberry Pi

It comes with a booklet with easy installation instructions.

NOTE:

It is necessary to install an actively cooled heatsink and fan on the RPi's processor to overclock it. Overclocking any processor without adequate cooling may damage it.

We can overclock the CPU, GPU, and RAM of an RPi board. In this section, we will discuss how to overclock an RPi 4B board.

Make sure that you update the firmware with the following command:

```
sudo rpi-update
```

It is necessary to update the firmware before overclocking the Pi. Once you've done that, run the following command:

```
sudo nano /boot/config.txt
```

This will open `/boot/config.txt` using the **nano** text editor. At the end of the file, add the following lines:

```
over_voltage=6  
arm_freq=2147
```

In the first line, we are setting the overvoltage as overclocking requires additional power. In the next line, we are overriding the default clock frequencies of the CPU. Save the changes and reboot the RPi.

Often, the RPi may not boot back up. In that case, you might want to change the `/boot/config.txt` settings for overclocking (using a Windows PC) to `over_voltage=2` and `arm_freq=1750`, respectively,

In the case that these setting too fail to boot the RPi, then comment both the lines and the RPi will boot up. Overclocking does not work stably with every processor.

When we run a computationally heavy process on the RPi board, all these additional megahertz will manifest themselves. We can monitor the clock in real time using the following command:

```
watch -n1 vcgencmd measure_clock arm
```

The output will cross the speed range of 2 billion (2 GHz) once we launch any heavy program on the RPi.

All this additional processing power we obtained by overclocking the RPi board will help us with our computer vision experiments.

Summary

In this chapter, we learned how to remotely log into the RPi and how to access the RPi desktop remotely with RDP. We also learned how to install OpenCV and how to verify it. Also, we learned how to overclock an RPi board.

We will be using all the skills we learned in this chapter throughout this book for accessing the Command Prompt and desktop of Raspbian OS remotely while writing programs for computer vision. We will also use file transfer quite a few times, as well as the OpenCV library in most programs.

In the next chapter, we will learn the basics of Python, NumPy, matplotlib, and the RPi GPIO library. We will also learn about the SciPy ecosystem.

3

Introduction to Python Programming

In the previous chapter, we learned how to remotely access the Command Prompt and the desktop of a **Raspberry Pi (RPi)** board. We also installed OpenCV for Python 3. Finally, we learned how to overclock the RPi and examined the various heatsinks for the RPi.

Continuing from where we left off at the end of the previous chapter, in this chapter, we will start by looking at Python 3 programming on the RPi. We will have a brief look at the **Scientific Python (SciPy)** ecosystem and all the libraries in it. Then, we will write basic programs for numerical computation with NumPy N-Dimensional Arrays (**ndarrays**). We will also learn how to visualize data with Matplotlib. Finally, we will explore the hardware aspects of the RPi with the **General Purpose Input Output (GPIO)** library of Python for the RPi.

In short, we will cover the following topics:

- Understanding Python 3
- The SciPy ecosystem

- Programming with NumPy and Matplotlib
- RPi GPIO programming

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter03/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/37UVwmO>.

Understanding Python 3

Python is a high-level, interpreted, general-purpose programming language. It was created by Guido van Rossum and was started as a personal hobby project but has since grown into what it is today. The following is a timeline of the major milestones in the development of the Python programming language:

Month	Year	Milestone
December	1989	Python began as a hobby programming project.
February	1991	Code was published to alt.sources.
January	1994	Version 1.0.
January	1994	The comp.lang.python newsgroup was formed.
October	2000	Python 2.0 released.
December	2008	Python 3.0 released.
December	2019	Python 2's Sunset.
January	2020	Python 3 continues....

Figure 3.1 – Timeline of Python development milestones

Guido van Rossum held the title of **benevolent dictator for life** for the Python project for most of its life cycle. He stepped down from the role in July 2018 and has been part of the **Python Steering Council** ever since.

You can read more about Python on its home page at www.python.org.

The Python programming language has two major versions—Python 2 and Python 3. They are mostly incompatible with one another. As the preceding timeline shows, Python 2's sunset happened on 31st December 2019. This means that there is no further development of Python 2. Official support has also ceased to exist. The only Python version under active development and with continued support is Python 3. A lot of code (in fact, billions of lines of code) that is in production for many organizations is still in Python 2. So, the migration from Python 2 to Python 3 requires a major effort.

Python on RPi and Raspberry Pi OS

Python comes pre-installed on the Raspberry Pi OS image that we downloaded. Both versions of Python—Python 2 and Python 3—come with the Raspberry Pi OS image. We will look at Python 3 in detail as we will write all of our programs with Python 3.

Open lxterminal or log in remotely to the RPi and run the following command:

```
python -v
```

This produces the following output:

```
Python 2.7.16
```

The -v option returns the version of the Python interpreter. So, the python command refers to the Python 2 interpreter. However, we need Python 3. So, run the following command in the Command Prompt:

```
Python3 -v
```

This produces the following output:

```
Python 3.7.3
```

This is the Python 3 interpreter and we will use it for all of our programming exercises throughout this book. To find out the location of the interpreter on your disc (in our case, our microSD card), run the following command:

```
which python3
```

This produces the following output:

```
/usr/bin/python3
```

This is where the executable for the Python 3 interpreter is located.

Python 3 IDEs on Raspberry Pi OS

Before we get started with Python 3 programming, we will learn which **Integrated Development Environments (IDEs)** can be used to write programs with Python. Raspberry Pi OS, as of now, comes with two IDEs. Both can be accessed from the **Programming** option in the Raspbian menu, as shown:

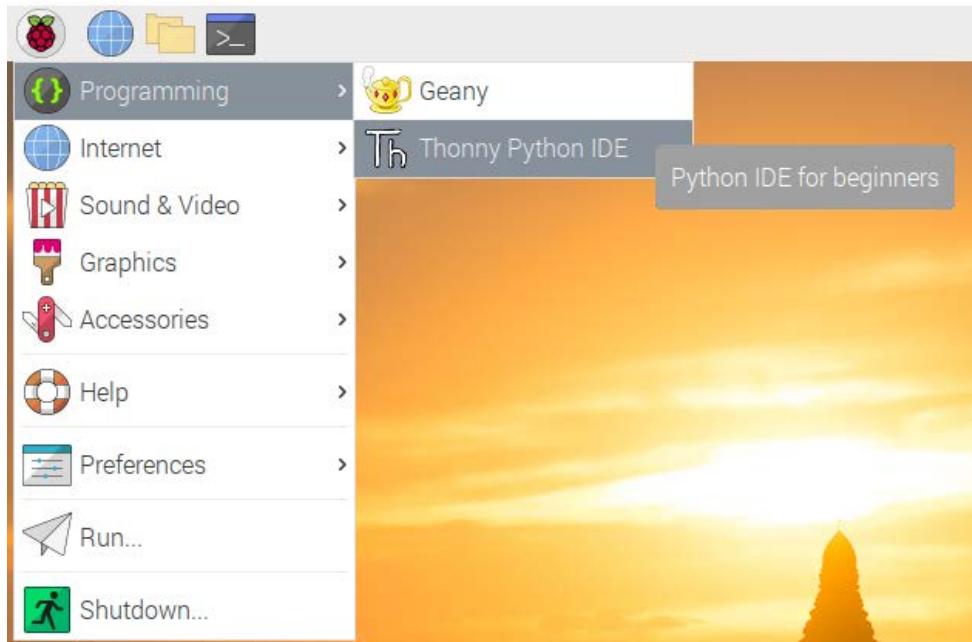


Figure 3.2 – The Thonny and Geany Python IDEs in the Raspbian menu

The first option is the **Geany** IDE, which can be used with many programming and markup languages, including Python 2 and Python 3. You can read more about it at <https://geany.org/>. The second option is **Thonny Python IDE**, which supports Python 3 and the MicroPython variants.

I personally prefer to use **Integrated Development and Learning Environment (IDLE)**, which is developed and maintained by the Python Foundation. You can read more about it at <https://docs.python.org/3/library/idle.html>. Earlier versions of Raspberry Pi OS used to come with IDLE. However, it is no longer present in the latest version of Raspberry Pi OS. Instead, we have Geany and Thonny. However, we can download IDLE with the following command:

```
sudo apt-get install idle3 -y
```

Once installed, we can find it in the **Programming** menu option under the Raspbian menu, as shown in the following screenshot:



Figure 3.3 – The option for IDLE for Python 3

Click on it to open it. Alternatively, we can launch it from the Command Prompt with the following command:

```
idle
```

Note that this command will not work and will throw an error if we have remotely connected to the Command Prompt of the RPi (using an SSH client such as PuTTY or Bitvise) as the command invokes the GUI. It will work if a visual display is directly connected to the RPi or if we access the RPi desktop remotely. This invokes a new window, as follows:

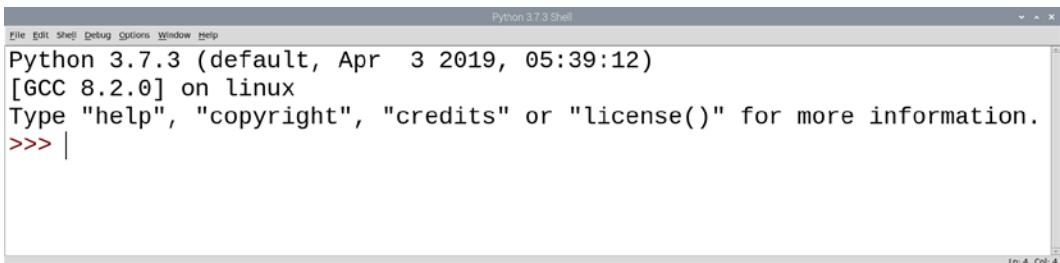


Figure 3.4 – Python 3 interactive mode in IDLE

This is the Python 3 interpreter prompt, or Python 3 shell. We will discuss this concept in detail later in this chapter.

Now, go to **File | New File** from the top menu. This will open a new code editor window, as follows:

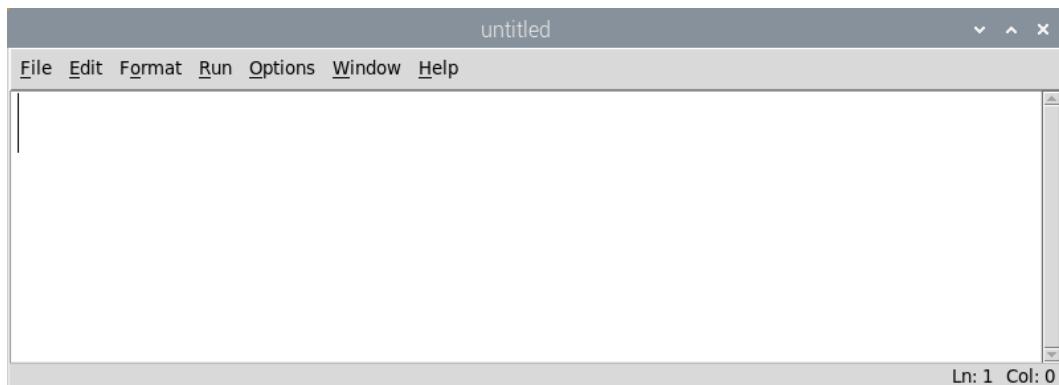


Figure 3.5 – A new blank Python program

The interpreter window will also stay open when this happens. You can either close or minimize it. If you find it difficult to read the text in the interpreter or code editor in IDLE due to the size of the font, you can go to **Options | Configure IDLE** from the menu to set the font and size of the text. The configuration window looks as follows:



Figure 3.6 – IDLE configuration

Let's write a customary **Hello World!** program. Type the following text into the window:

```
print('Hello World!')
```

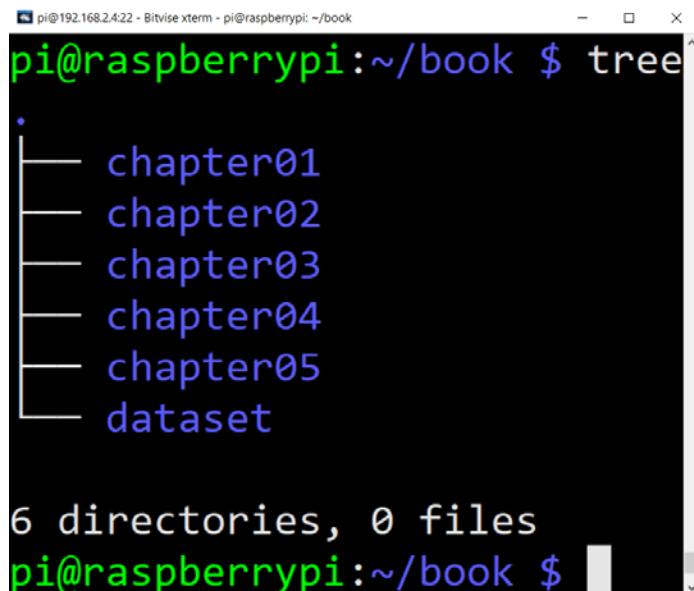
Then, from the menu, click **Run | Run Module**. It will ask you to save it. Click the **OK** button and it will take you to the **Save** dialog box. I prefer to save the code for this book by chapter in a directory, with sub-directories for each chapter. You can make the directory structure by running the following commands in the home directory of the `pi` user:

```
mkdir book
mkdir book/dataset
mkdir book/chapter01
```

We can make a separate directory for each chapter like this. Also, a separate `dataset` directory to store our data is needed. After creating the prescribed directory structure, run the following sequence of commands:

```
cd book
tree
```

We can see the directory structure in the following output of the `tree` command:



The screenshot shows a terminal window titled "pi@192.168.2.4:22 - Bitvise xterm - pi@raspberrypi: ~/book". The command `tree` is run, displaying the following directory structure:

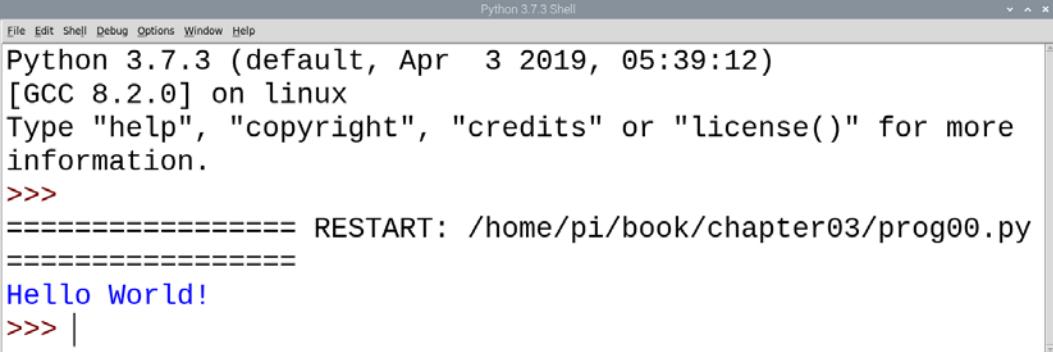
```
pi@raspberrypi:~/book $ tree
.
├── chapter01
├── chapter02
├── chapter03
├── chapter04
├── chapter05
└── dataset

6 directories, 0 files
pi@raspberrypi:~/book $
```

Figure 3.7 – Directory structure for saving programs for this book

We can create the same directory structure by using the **Save** dialog box of IDLE or the **File Manager** application of Raspberry Pi OS.

Once the directory corresponding to the current chapter is created, save the file there as `prog00.py`. You just need to enter the filename; IDLE will automatically assign the `.py` extension to the file. Then, the file will be executed by the Python 3 interpreter and the output will be visible in the interpreter shell, as follows:



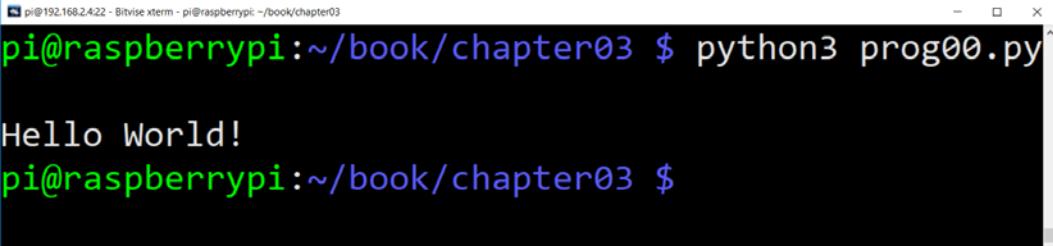
```
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
=====
RESTART: /home/pi/book/chapter03/prog00.py
=====
Hello World!
>>> |
```

Figure 3.8 – Execution of a Python 3 program in IDLE

We can also write the same code with the **Nano** editor. The only difference is that we also need to provide an extension when saving it. We can navigate to the directory that has the `prog00.py` file and run the following command to feed the file to the Python 3 interpreter:

```
python3 prog00.py
```

The Python 3 interpreter will execute the program and print the output, as follows:



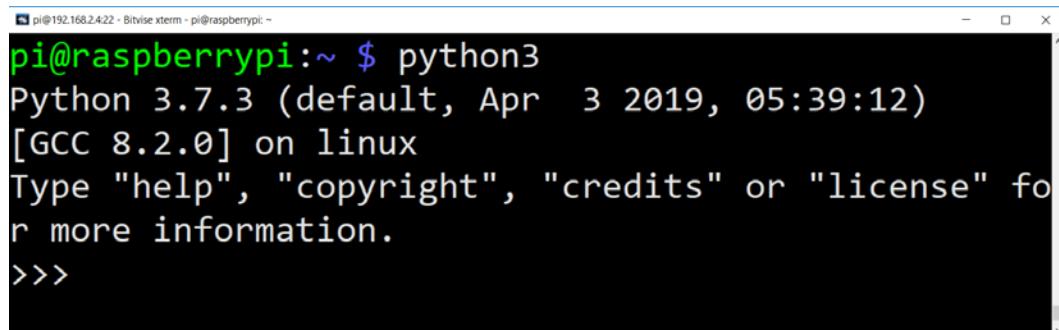
```
pi@192.168.2.422 - Bitvise xterm - pi@raspberrypi: ~/book/chapter03 $ python3 prog00.py
Hello World!
pi@raspberrypi:~/book/chapter03 $
```

Figure 3.9 – Execution of a Python 3 program in LXTerminal

Working with Python 3 in interactive mode

We have seen how to write a Python 3 program using the IDLE and Nano editors. We have also seen how to launch the program using IDLE and from the Command Prompt of Raspberry Pi OS. Running a Python 3 program in this fashion is known as script mode.

There is also another mode—interactive mode. In interactive mode, we launch the Python interpreter and it acts as a command-line interpreter. When we enter and run a statement, we get immediate feedback from the interpreter. We can launch the interactive mode in two ways. We have already seen the first way. When we launch IDLE, it opens the interpreter and we can use it to run the Python 3 statements. The other way is to run the `python3` command in the Command Prompt. This will invoke the Python 3 interpreter in the Command Prompt, as follows:

A screenshot of a terminal window titled "pi@192.168.2.4:22 - Bitvise xterm - pi@raspberrypi: ~". The window displays the output of the command `python3`. It shows the Python version (3.7.3), the build date (Apr 3 2019, 05:39:12), the compiler (GCC 8.2.0), and the operating system (linux). It also provides help information for "help", "copyright", "credits", and "license". The prompt shows the user has typed `>>>` and is awaiting further input.

```
pi@raspberrypi:~ $ 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.
>>>
```

Figure 3.10 – Python 3 in interactive mode on the Command Prompt

Type the following statement into the prompt:

```
>>> print('Hello World!')
```

Then, press *Enter*. It will be executed and the output will be shown on the next line. This way, we can execute single-line statements and small code snippets like this. We will be using interactive mode extensively in this chapter. From the next chapter onward, we will use script mode—that is, we will save the programs in files and launch them from the command prompt or IDLE.

The basics of Python 3 programming

Let's start by learning the basics of Python 3 programming. Open the Python 3 interactive prompt. Type in the following statements:

```
>>> pi = 3.14
>>> print(pi)
```

This will show the value of the `pi` variable. Run the following statement:

```
>>> print(type(3.14))
```

This shows the following output:

```
<class 'float'>
```

You might have noticed that we have not declared the data type of the variable here. That is because Python is a **dynamically typed** programming language. We also say that the variable belongs to a class type. This means that it is an object, which is true for all variables and other constructs in Python. Everything is an object in Python. This makes Python a truly object-oriented programming language. Almost everything has attributes and methods.

In order to exit the Command Prompt, press `Ctrl + D` or run the `exit()` statement.

Let's create our own class and an object of that class. Save the file and name it `prog01.py`, then add the following code to it:

```
class Person:

    def __init__(self, name='', age=0):
        self.name = name
        self.age = age

    def show(self):
        print(self.name)
        print(self.age)
```

In the preceding code, we defined `Person`. The `__init__()` class is the initializer function and it is called automatically whenever an object of the `Person` class is created. The `self` parameter is a reference to the current instance of the class and is used to access variables that belong to the class within the class definition.

Let's add some more code to `prog01.py`. We will create a class object, as follows:

```
p1 = Person('Ashwin', 25)
p1.show()
```

We created the `p1` class and then showed the properties of the object with the `show()` function call. Here, we are assigning the values to class member variables at the time of the creation of the class object.

Let's look at another way to create an object and assign values to the member variables. Add the following code to the file:

```
p2 = Person()
p2.name = 'Jane'
p2.age = 20
print(p2.name)
print(p2.age)
```

In the preceding code, we are creating an object and the initializer function is called with the default arguments. Then, we are assigning the values to the class variables and accessing them directly using the class object. Run the program and see the output.

Now, open the Python 3 interpreter and run the following statements:

```
>>> import sys
>>> print(sys.platform)
```

This will return the name of the current OS (Linux). The first statement imports `sys`, which is a Python standard library. It comes with the Python interpreter as part of its **batteries included** motto. This means the Python interpreter comes with a large set of useful libraries. `sys.platform` returns the current OS name string.

Let's try another example. In the previous chapter, we installed the OpenCV library. Let's import that again now. We have already tested it directly from the Raspberry Pi OS Command Prompt. Let's try to do the same in interactive mode:

```
>>> import cv2
>>> print(cv2.__version__)
```

The first statement imports the OpenCV library to the current session. The second statement returns a string that contains the version number of the installed OpenCV library.

There are many topics in the basics of Python 3 programming, but it is very difficult to cover all of them. Also, to do so is beyond the scope of this book. However, we will use the topics we have just learned quite frequently in this book.

In the next section, we will explore the SciPy ecosystem libraries.

The SciPy ecosystem

The SciPy ecosystem is a collection of libraries for programming science, mathematics, and engineering functionalities. It has the following libraries as core components:

- NumPy
- SciPy
- Matplotlib
- IPython
- SymPy
- pandas

In this book, we will use all of these libraries except SymPy and pandas. In this section, we will have a look at the NumPy and Matplotlib libraries. We will learn the useful aspects of the other two libraries in the later chapters of this book.

The basics of NumPy

NumPy is a fundamental package that can be used for numerical computation with Python. It is a matrix library for linear algebra. NumPy ndarrays can also be used as an efficient multi-dimensional container of generic data. Arbitrary data types can also be defined and used. NumPy is an extension of the Python programming language. It adds support for large multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions that can be used to operate on these arrays. We will use NumPy arrays throughout this book to represent images and carry out complex mathematical operations. NumPy comes with many built-in functions for all of these operations. So, we do not have to worry about basic array operations. We can focus directly on the concepts and code for computer vision. All of the OpenCV array structures are converted to and from NumPy arrays. So, whatever operations you perform in NumPy, you can always combine NumPy with OpenCV.

We will use NumPy with OpenCV a lot in this book. Let's start with some simple example programs that will demonstrate the real power of NumPy.

NumPy comes pre-installed on Raspberry Pi OS. So, we do not have to install it separately.

Open the Python 3 interpreter and try the following examples.

Creation of ndarrays

Let's see some examples on ndarray creation. The `array()` method will be used very frequently in this book. There are many ways to create different types of arrays. We will explore these methods as and when they are needed in this book. Follow these commands for ndarray creation:

```
>>> import numpy as np
>>> x=np.array([1,2,3])
>>> x
array([1, 2, 3])
>>> y=arange(10)
>>> y=np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Basic operations on ndarrays

We are going to learn about the `linspace()` function now. It takes three parameters—`start_num`, `end_num`, and `count`. This creates an array with equally spaced points, starting from `start_num` and ending with `end_num`. You can try out the following example:

```
>>> y=np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a=np.array([1,3,6,9])
>>> a
array([1, 3, 6, 9])
>>> b=np.linspace(0,15,4)
>>> b
array([ 0.,  5., 10., 15.])
>>> c = a - b
>>> c
array([ 1., -2., -4., -6.])
```

The following is the code that can be used to calculate the square of every element in an array:

```
>>> a**2  
array([ 1,  9, 36, 81], dtype=int32)
```

Linear algebra with ndarrays

Let's explore some examples relating to linear algebra. You will learn how to use the `transpose()`, `inv()`, `solve()`, and `dot()` functions, which are useful when performing operations related to linear algebra:

```
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])  
>>> a.transpose()  
array([[1, 4, 7],  
       [2, 5, 8],  
       [3, 6, 9]])  
>>> np.linalg.inv(a)  
array([[-4.50359963e+15,  9.00719925e+15, -4.50359963e+15],  
       [ 9.00719925e+15, -1.80143985e+16,  9.00719925e+15],  
       [-4.50359963e+15,  9.00719925e+15, -4.50359963e+15]])  
>>> b = np.array([3, 2, 1])  
>>> np.linalg.solve(a, b)  
array([-9.66666667, 15.33333333, -6.          ])  
>>> c = np.random.rand(3, 3)  
>>> c  
array([[0.91827923, 0.75063499, 0.40049332],  
       [0.09520566, 0.16718726, 0.6577751 ],  
       [0.95343917, 0.50972786, 0.65649385]])  
>>> np.dot(a, c)  
array([[ 3.96900804,  2.61419309,  3.68552507],  
       [ 9.86978019,  6.89684344,  8.82981189],  
       [15.77055234, 11.17949379, 13.9740987 ]])
```

Note:

You can explore NumPy in more detail at <http://www.numpy.org>.

Matplotlib

Matplotlib is a plotting library for Python and it produces publication-quality figures. It can produce various types of visualizations, such as plots, three-dimensional visualizations, and images. It is important to understand the basics of Matplotlib to work with any computer vision library, such as OpenCV.

Matplotlib was developed by John D. Hunter and it is continually developed by the open source community. It is an integral part of the SciPy ecosystem. All the other libraries in the SciPy ecosystem use Matplotlib for the visualization of data. `pyplot` is a module in Matplotlib that provides a MATLAB-like interface for the visualization of data.

Before we begin programming with Matplotlib, we need to install it as it does not come pre-installed on Raspberry Pi OS.

We can install it with the `pip3` utility. We have already seen how to use this utility when installing OpenCV. Let's look at it in more detail. `pip` means **Pip Installs Packages** or **Pip Installs Python**. It is a recursive acronym (meaning the acronym is itself part of the acronym). It is a command-line utility that comes with the Python interpreter and is used for installing libraries. `pip3` is the Python 3 version of this utility. It first connects to the Python Package Index, which is a repository of Python libraries. Then, it downloads and installs the library we need.

Note:

You can read more about the Python Package Index and `pip` at <https://pypi.org/project/pip/> and <https://pypi.org/>.

We can install Matplotlib by running the following command:

```
pip3 install matplotlib
```

Matplotlib is a big library and has many prerequisite libraries. All of these prerequisite libraries are automatically installed by `pip3` and then Matplotlib is installed. It will take some time to do so, depending on the speed of your internet connection.

Once the installation is complete, we can write a few sample programs. We will use Python 3 in script mode and IDLE or the Nano editor to write programs. Create a new file, `prog02.py`, and add the following code to it:

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
x = np.array([1, 2, 3, 4], dtype=np.uint8)
y = x**2 + 1
plt.plot(x, y)
plt.grid('on')
plt.show()
```

In the first line of the preceding code, we import Matplotlib's `pyplot` module with the `plt` alias. Then, we import NumPy. We use the `array()` function call to create a linear ndarray by passing it a list of 8-bit unsigned integers (`uint8` refers to the data type). Then, we define $y = x^2 + 1$. The `plot()` function plots y versus x . We can set the grid on or off by passing '`on`' or '`off`' to the `grid()` function call. The `show()` function call starts an event loop, looks for all the currently active visualization objects, and opens a visualization window to show the plots or the other visualization. The following is the output:

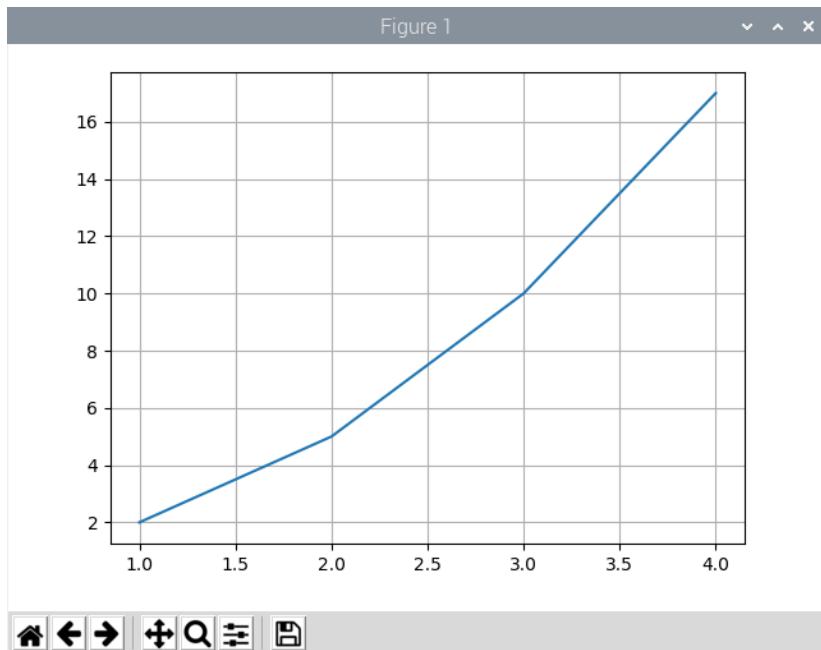


Figure 3.11 – Visualization with Matplotlib

As we can see, this shows us the visualization. The grid is visible as we turned it on. We also have image controls at the bottom where we can save, zoom, and perform other operations on the visualization. Note that we need to run the program directly on the RPi by using IDLE on the Command Prompt or by using the remote desktop. Running the program from a remote SSH command line will not throw any error but it won't show any output either.

Save the `prog02.py` code file as `prog03.py`. After the `plot()` function call and before `grid()` call, add the following lines:

```
y = x + 1  
plt.plot(x, y)  
plt.title('Graph')  
plt.xlabel('X-Axis')  
plt.ylabel('Y-Axis')
```

The rest of the code remains as it is. Save the program. Here, we are demonstrating the visualization of multiple plots in the same window. We are also adding a title and labels to the graph. Run the `prog03.py` file and the following is the output:

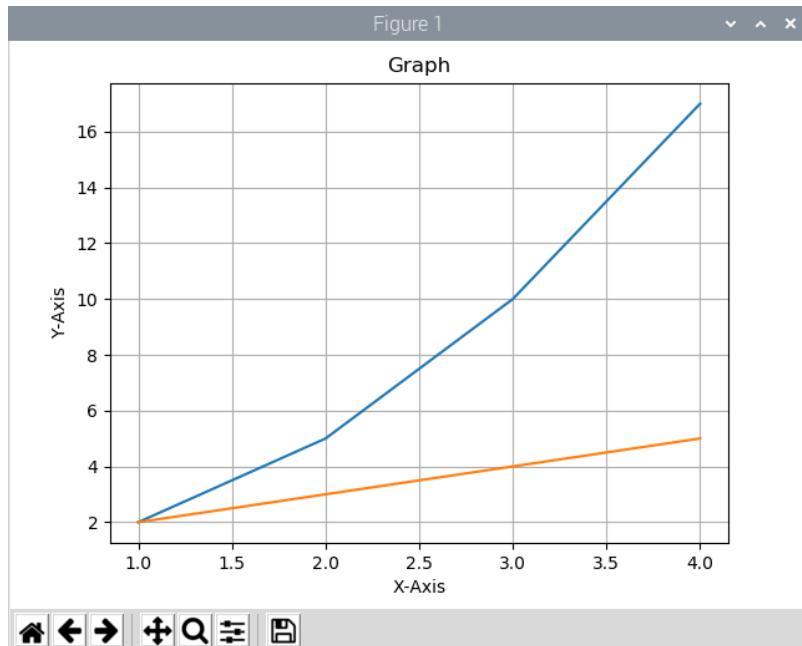


Figure 3.12 – Multiple graphs with Matplotlib

We can add the following line just before `plt.show()` to save the visualization on the disk:

```
plt.savefig('test1.png', dpi=300, bbox_inches='tight')
```

This will save the visualization in the current directory and name it `test1.png`.

Let's move on to the more interesting part. We can visualize ndarrays as images with the `imshow()` function. Let's see an example. Create a new file named `prog04.py` and add the following code to it:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype = np.uint8 )

plt.imshow(x)

plt.show()
```

In the preceding code, we are creating a two-dimensional array (with a size of 5x2) and visualizing it as an image with the `imshow()` call. The following is the output:

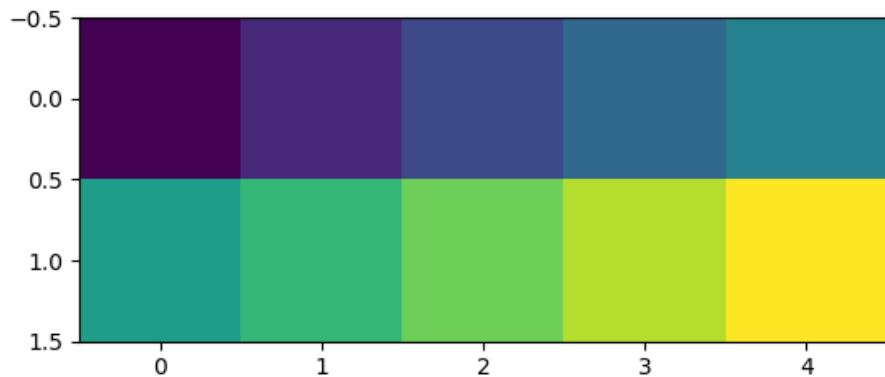


Figure 3.13 – Visualizing numbers as an image

As this is an image, we really do not need grid and axis ticks. We can add the following two lines to the code before `plt.show()` to turn them off:

```
plt.axis('off')
plt.grid('off')
```

Run the modified code and observe the output.

The image has been rendered with what we call as default colormap. A colormap is a color scheme for visualizations. If we change `plt.imshow(x)` to `plt.imshow(x, cmap='gray')`, then the following is the output:



Figure 3.14 – The image in greyscale mode

There are a lot of colormaps. We can even create our own custom colormaps; however, for the demonstration of computer vision algorithms, that is not needed as the existing colormaps suffice. If you are curious about the available colormap names that we can use, you can find them out as follows. Open the Python 3 interpreter in the interactive mode and import the `pyplot` module to Matplotlib:

```
>>> import matplotlib.pyplot as plt
```

The `plt.colormaps()` list has names of all the colormaps. First, we check how many colormaps are there, which is easy. Run the following statement:

```
>>> print(len(plt.colormaps()))
```

This will print the number of colormaps. Finally, run the following statement to see a list of all the available colormaps:

```
>>> print(plt.colormaps())
```

The list is quite long and we will be using only a few of the colormaps from the list for our demonstrations. In the `prog04.py` file, change `plt.imshow(x)` to `plt.imshow(x, cmap='Accent')`, and the following will be the output:

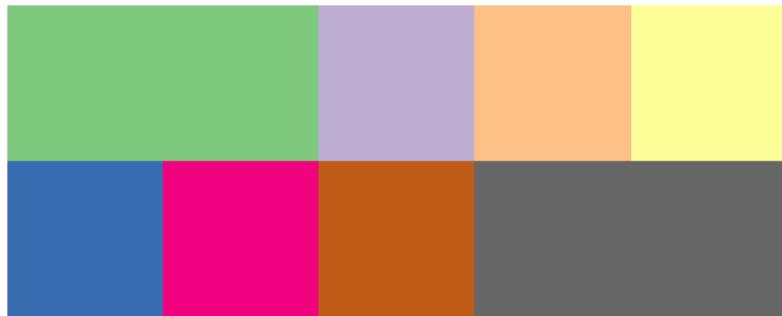


Figure 3.14 – The image with the Accent colormap

This much knowledge of Matplotlib is more than enough to get started with OpenCV and computer vision.

Up to now, we have seen examples of visualization of one-dimensional and two-dimensional ndarrays. Now, let's see how we can create a random three-dimensional ndarray and how to visualize it. Observe the following code:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.rand(3, 3, 3)
plt.imshow(x)
plt.axis('off')
plt.grid('off')
plt.show()
```

In the preceding code, we are using `np.random.rand()` to create a random three-dimensional array. We just need to pass to it the size of every dimension. In the preceding example, we are creating a three-dimensional matrix with a size of $3 \times 3 \times 3$. Run the preceding code and see the output for yourself. All the images we will work with throughout this book are represented as either two-dimensional or three-dimensional ndarrays. This knowledge of data visualization will be very helpful once we start working with images.

RPi GPIO programming with Python 3

One of the main unique selling points of the RPi and similar single-board computers is the onboard GPIO pins. A few early models of the RPi boards have 26 pins. Most recent models have 40 pins for GPIO. We can obtain the details of the pins on a board by running the `pinout` command on the Command Prompt. The following is the output of the command for my RPi 4B board:

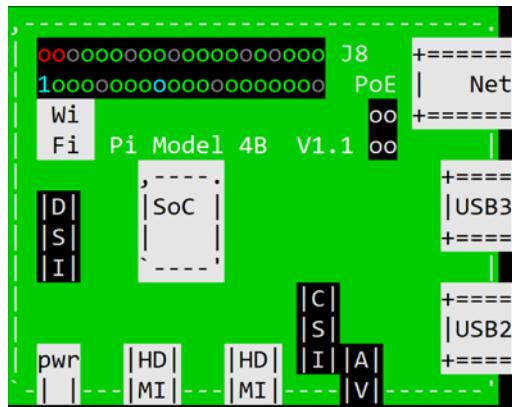


Figure 3.16 – Part 1 of the command pinout

In the top left, we can see the 40 pins for GPIO. Pin number 1 is labeled there. The red circle above it is pin number 2. The pin adjacent to pin number 1 is pin number 3, and so on. The following part of the output shows the numbering of all the pins:

3V3	(1)	(2)	5V
GPIO2	(3)	(4)	5V
GPIO3	(5)	(6)	GND
GPIO4	(7)	(8)	GPIO14
GND	(9)	(10)	GPIO15
GPIO17	(11)	(12)	GPIO18
GPIO27	(13)	(14)	GND
GPIO22	(15)	(16)	GPIO23
3V3	(17)	(18)	GPIO24
GPIO10	(19)	(20)	GND
GPIO9	(21)	(22)	GPIO25
GPIO11	(23)	(24)	GPIO8
GND	(25)	(26)	GPIO7
GPIO0	(27)	(28)	GPIO1
GPIO5	(29)	(30)	GND
GPIO6	(31)	(32)	GPIO12
GPIO13	(33)	(34)	GND
GPIO19	(35)	(36)	GPIO16
GPIO26	(37)	(38)	GPIO20
GND	(39)	(40)	GPIO21

Figure 3.17 – Part 2 of the command pinout

As we can see in the preceding output, we have power pins (3V3, 5V, and GND) and digital I/O pins, marked as GPIO_{xx}.

LED programming with GPIO

Now, we will see how to program LEDs with GPIO pins as output pins. Let's prepare a simple circuit for blinking an LED, first.

For that, we need jumper cables, an LED, and a 220-ohm resistor. Prepare your circuit as in the following diagram:

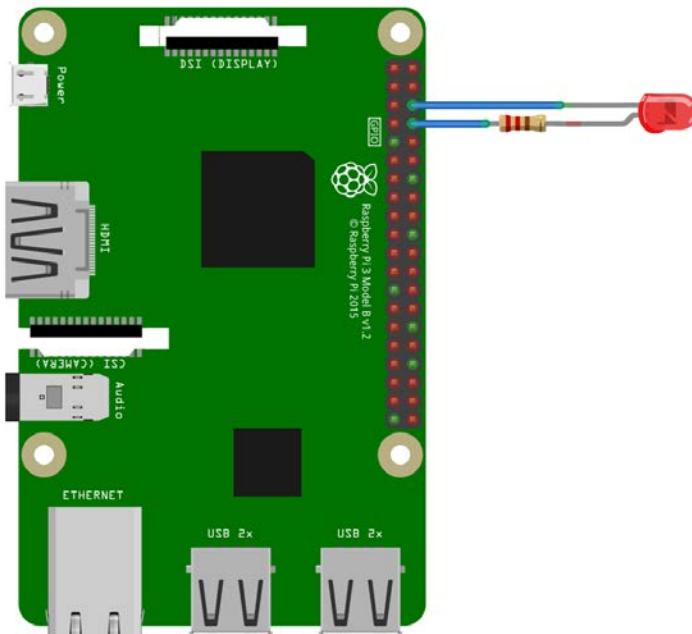


Figure 3.18 – LED-resistor circuit

As we can see in the preceding circuit diagram, we are connecting the anode of the LED to physical pin 8 through a 220-ohm resistor, and the cathode of the LED is connected to physical pin 6, which is a **Ground (GND)** pin.

Note:

You will find many beautiful circuit diagrams like this throughout this book. I used an open source software called Fritzing to generate them. You can access Fritzing's home page at <https://fritzing.org/>. Fritzing files have the *.fzz extension. These files are part of the downloadable code bundle for this book.

Now, let's get into the code. For that, we need to install the GPIO library. The latest version of Raspberry Pi OS comes with the GPIO library already installed. However, if it is not there, we can install it by running the following command:

```
sudo apt-get install python3-rpi.gpio -y
```

Now, create a new file, prog05.py, in the same directory and add the following code to it:

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)
GPIO.setup(8, GPIO.OUT, initial=GPIO.LOW)

while True:
    GPIO.output(8, GPIO.HIGH)
    sleep(1)
    GPIO.output(8, GPIO.LOW)
    sleep(1)
```

In the preceding code, the first two lines import the required libraries.

`setwarnings(False)` disables all the warnings and `setmode()` is used to set the pin addressing mode. There are two modes, `GPIO.BUILD` and `GPIO.BCM`. In the `GPIO.BUILD` mode, we refer to the pins by their physical location numbers. In the `GPIO.BCM` mode, we refer to the pins by their Broadcom SOC channel number. I prefer the `GPIO.BUILD` mode because it is easy to remember the pins by their physical location number. `setup()` is used to set each GPIO pin as an input or output.

In the preceding code, the first argument is the pin number, the second argument is the mode, and the third one is the initial state of the pin. `output()` is used to send either HIGH or LOW signals to the pin. `sleep()` is imported from the `time` library and it produces a delay of a given number of seconds. Run the preceding program to make the LED blink. In order to terminate the program, press `Ctrl + C` on the keyboard.

Similarly, we can write the following code for the same circuit to flash the LED to convey a **Save our Souls (SOS)** message visually:

```
import RPi.GPIO as GPIO
from time import sleep
```

```
GPIO.setwarnings(False)      # Ignore Warnings
GPIO.setmode(GPIO.BOARD)    # Use Physical Pin Numbering
GPIO.setup(8, GPIO.OUT, initial=GPIO.LOW)

def flash(led, duration):
    GPIO.output(led, GPIO.HIGH)
    sleep(duration)
    GPIO.output(led, GPIO.LOW)
    sleep(duration)

while True:
    flash(8, 0.2)
    flash(8, 0.2)
    flash(8, 0.2)
    sleep(0.3)

    flash(8, 0.5)
    flash(8, 0.5)
    flash(8, 0.5)

    flash(8, 0.2)
    flash(8, 0.2)
    flash(8, 0.2)
    sleep(1)
```

In the preceding program, we defined a custom `flash()` function that accepts the pin number and the duration of the flash. Then, we set the provided pin to `HIGH` for the given duration and then set it to `LOW` for the given duration. So, the LED connected to the pin is alternately turned on and off for the given duration. When this happens in the pattern of . . . - - - . . . (triple dots followed by a triple dash followed by triple dots), which is Morse code for SOS, it is called a **distress signal**. For each `.` (dot) character, we flash the LED for 0.2 seconds, and for each `-` (dash) character, we flash it for half a second. We have added all of this to the preceding infinite `while` loop. When we run the program, it starts flashing the SOS message until we terminate it by pressing `Ctrl + C` on the keyboard.

Let's look at some more GPIO and Python 3 programming. Prepare a circuit as in the following diagram:

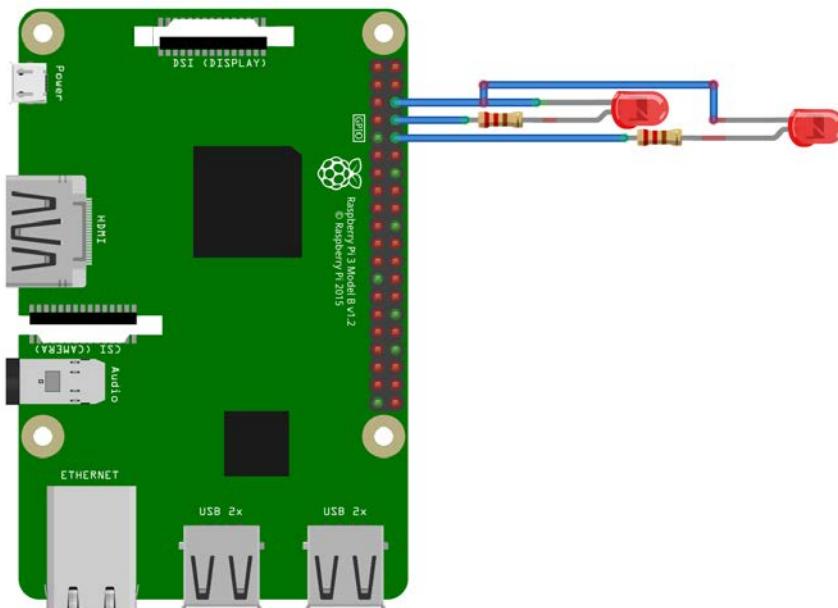


Figure 3.19 – A circuit diagram with two LEDs

As we can see here, we just need to connect the anode of an additional LED to pin 10 through a 220-ohm resistor and a cathode of the same LED to the GND pin. We will make both the LEDs blink alternately. The following is the code for this:

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)
GPIO.setup(8, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(10, GPIO.OUT, initial=GPIO.LOW)

while True:
    GPIO.output(8, GPIO.HIGH)
    GPIO.output(10, GPIO.LOW)
    sleep(1)
```

```
GPIO.output(8, GPIO.LOW)
GPIO.output(10, GPIO.HIGH)
sleep(1)
```

You should now be familiar with all of the functions in the preceding code as we discussed them in the earlier two examples. This code, upon execution, makes the LEDs blink for 1 second alternately.

Now, there is another way to produce the same output. Take a look at the following program:

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)
GPIO.setup(8, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(10, GPIO.OUT, initial=GPIO.LOW)
counter = 0

while True:
    if counter % 2 == 0:
        led1 = 8
        led2 = 10
    else:
        led1 = 10
        led2 = 8

    GPIO.output(led1, GPIO.HIGH)
    GPIO.output(led2, GPIO.LOW)
    sleep(1)
    counter = counter + 1
```

In the preceding program, we are using slightly different logic to demonstrate the usage of `if` statement in Python 3. We have a variable named `counter`, which is set to 0 at the beginning. In the `while` loop, we check whether the `counter` value is even or odd, and depending on that, we set which LED is to be turned on and which is to be turned off. At the end of the loop, we increment `counter` by 1. The output of this program is the same as the earlier one and it can be terminated by pressing *Ctrl + C*.

Now, let's experiment with numerous LEDs. We will need a breadboard for this. Prepare a circuit as in the following diagram:

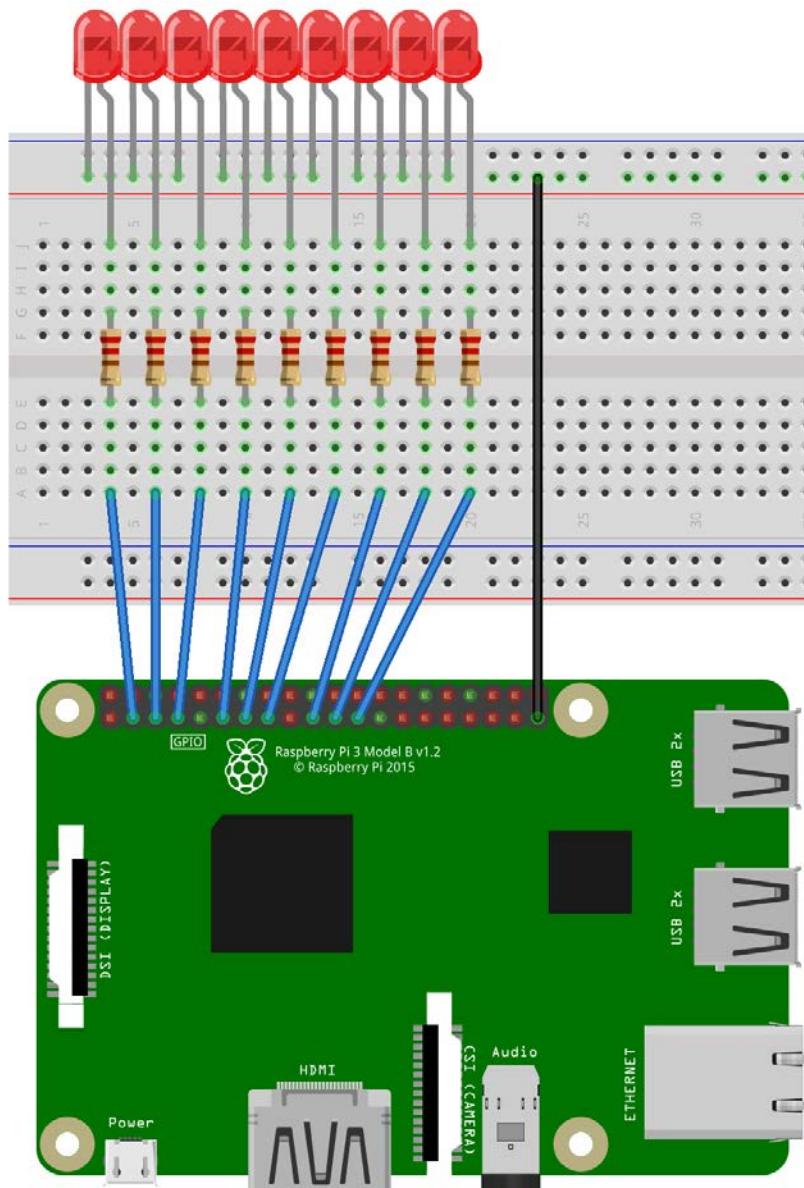


Figure 3.20 – A diagram for the chaser circuit

For programming, we will try a bit of a different approach. We will use the `gpiozero` library of Python 3. If it is not installed by default on your Raspbian distribution, it can be installed with the following command:

```
pip3 install gpiozero
```

This uses the BCM numbering system when addressing the pins. Save the following code in a Python file and run it to see a beautiful chaser effect:

```
from gpiozero import LED
from time import sleep

led1 = LED(2)
led2 = LED(3)
led3 = LED(4)
led4 = LED(17)
led5 = LED(27)
led6 = LED(22)
led7 = LED(10)
led8 = LED(9)
led9 = LED(11)

sleeptime = 0.2

while True:
    led1.on()
    sleep(sleeptime)
    led1.off()
    led2.on()
    sleep(sleeptime)
    led2.off()
    led3.on()
    sleep(sleeptime)
    led3.off()
```

```
led4.on()
sleep(sleepTime)
led4.off()
led5.on()
sleep(sleepTime)
led5.off()
led6.on()
sleep(sleepTime)
led6.off()
led7.on()
sleep(sleepTime)
led7.off()
led8.on()
sleep(sleepTime)
led8.off()
led9.on()
sleep(sleepTime)
led9.off()
```

All of the preceding code is self-explanatory and should be easy for you to understand by now. In the first line, we imported `LED`. We can pass to it a BCM pin number as an argument. It can be assigned to a variable and the variable then can call the `on()` and `off()` functions to turn the LED associated with it on and off, respectively. We also called `sleep()` between `on()` and `off()`.

Push-button programming with GPIO

Now, we are going to see how we can connect a push button to a RPi board with an internal pull-up resistor. Prepare a circuit as in the following diagram:

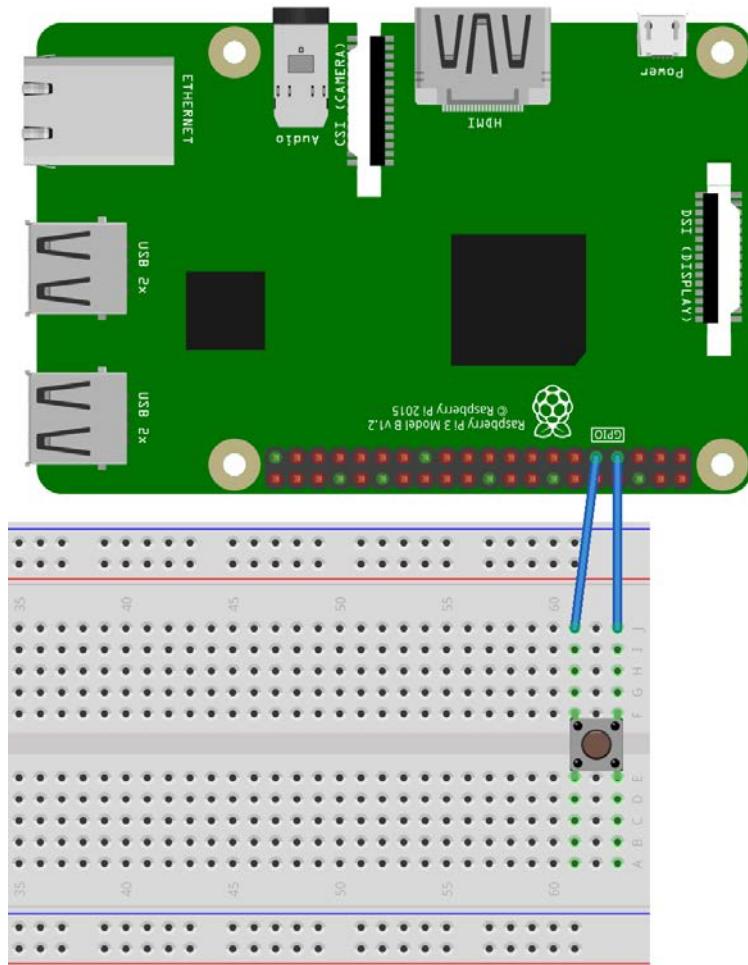


Figure 3.21 – A diagram for interfacing a push button

In the preceding circuit, we connect one end of the push button to pin number 7 and another to GND. Save the following code to a Python file:

```
from time import sleep
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
```

```
GPIO.setwarnings(False)
button = 7
GPIO.setup(button, GPIO.IN, GPIO.PUD_UP)
while True:
    button_state = GPIO.input(button)
    if button_state == GPIO.HIGH:
        print ("HIGH")
    else:
        print ("LOW")
    sleep(0.5)
```

In the preceding code, we are initializing pin 7 as an input pin. In `setup()`, the second argument decides the mode of the GPIO pin (`IN` or `OUT`). The third argument, `GPIO.PUD_UP`, decides whether it should be connected to an internal pull-up resistor. If we connect the button to an internal pull-up resistor, the GPIO pin to which the button is connected to is set to `HIGH` when the button is not pressed. If we press the button, it sets to `LOW`. `GPIO.input()` and returns the button state. Launch the program and the output will show `HIGH` if the button is open and `LOW` if the button is pressed. The following is the output:

```
File Edit Shell Debug Options Window Help
HIGH
HIGH
HIGH
HIGH
LOW
LOW
LOW
HIGH
HIGH
HIGH
```

Figure 3.22 – The output of the push button program

So, this is how we can detect a key press. The program can be terminated by pressing *Ctrl + C*.

We can also try a slightly different circuit and code. Prepare a circuit, as follows:

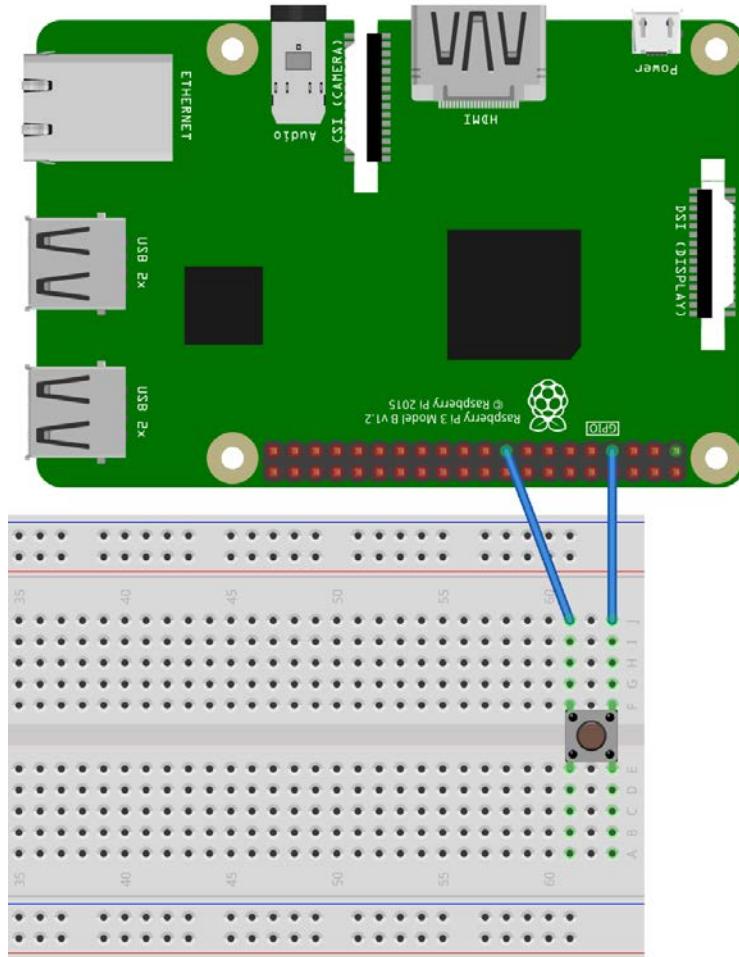


Figure 3.23 – Another circuit for the push button

In the preceding circuit, we connected one end of the push button to pin 7 and the other to a 3V3 pin. Do not connect this end to the 5V pin because when we push the button, it will connect to pin 7 and the GPIO pins can only handle up to 3V3 (3.3 V). Connecting them to a 5V source will damage the pins and the board. Prepare the circuit and make a copy of the code with the following command:

```
cp prog06.py prog07.py
```

In the new `prog07.py` file, we just have to make a small change in the `setup()` function call, as follows:

```
GPIO.setup(button, GPIO.IN, GPIO.PUD_DOWN)
```

This will connect the push button to the internal pull-down resistor. The pin connected to the push button remains set to `LOW` when the button is open and set to `HIGH` when the button is pushed. Run the program and the output looks as follows:

```
File Edit Shell Debug Options Window Help
LOW
LOW
LOW
HIGH
HIGH
HIGH
HIGH
HIGH
LOW
LOW
LOW
LOW
```

Figure 3.24 – The output of the second push button program

The program will show `LOW` in the beginning. If we push the button, it will become `HIGH`. The program can be terminated by pressing `CTRL + C`. This is another way of detecting a key press.

Summary

We learned the basics of Python 3 programming in this chapter. We also learned about the SciPy ecosystem and experimented with the NumPy and Matplotlib libraries. Finally, we saw how to use the GPIO pins of the RPi with LEDs and push buttons.

In the next chapter, we will get started with Python 3 and OpenCV programming. We will also try out a lot of hands-on exercises to learn about programming using a webcam and a RPi camera module.

4

Getting Started with Computer Vision

In the previous chapter, we learned the basics of programming of Python 3, NumPy, matplotlib, and **General Purpose Input Output (GPIO)**. In this chapter, we will focus on the acquisition of images and videos. This chapter has a lot of coding examples that we will be using throughout the book.

In this chapter, we will cover the following topics:

- Exploring image datasets
- Working with images using OpenCV
- Using matplotlib to visualize images
- Drawing geometric shapes with OpenCV and NumPy
- Working with a GUI
- Event handling and a primitive paint application
- Working with a USB webcam
- The Pi camera module

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter04/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/3dtrA2t>.

Exploring image datasets

We will need sample images for our computer vision programs with Python and OpenCV. We can find a lot of images online. However, many of those images are under copyright. Most computer vision researchers and professionals use standard image datasets. We prefer to use the following image datasets all the time:

- <http://sipi.usc.edu/database/>
- http://www.imageprocessingplace.com/root_files_V3/image_databases.htm
- <http://www.cvlab.cs.tsukuba.ac.jp/dataset/tsukubastereo.php>

Download these datasets. They will be in compressed `zip` format. Extract them into the `~/book/dataset` directory. From this chapter onward, we will write a lot of programs for computer vision that will require images, and we will use the images from these datasets for all our needs. The other option for images is to use a web camera and RPi camera module to capture images, which we will learn about later in this chapter.

Working with images using OpenCV

In this section, we will learn to read and store images using the OpenCV API and Python. All the programs in this book will use the OpenCV library. It can be imported with the following Python 3 statement:

```
import cv2
```

The `cv2.imread()` function reads an image from the disk and stores it in a NumPy `ndarray`. It accepts two arguments. The first argument is the name of the image file on the disk. The image should either be in the same directory where we are saving the current Python 3 script, or we must pass the absolute path of the image file as an argument to the `cv2.imread()` function.

The second argument is a flag that specifies the mode in which the image should be read. The flag can have one of the following values:

- `cv2.IMREAD_GRAYSCALE`: This reads an image from the disk in grayscale mode. The numerical value corresponding to this flag is 0.
- `cv2.IMREAD_UNCHANGED`: This reads an image from the disk as it is. The numerical value corresponding to this flag is -1.
- `cv2.IMREAD_COLOR`: This reads the image in color mode, and it is the default value for the argument of the parameter. The numerical value corresponding to this flag is 1. This is the default value of the argument.

The following is the code for reading an image in color mode:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', cv2.
IMREAD_COLOR)
```

We can rewrite the last line with the flags as follows:

```
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
```

The preceding style of writing code for reading the source images with numerical flags is very simple. So, we will use it throughout the book:

```
cv2.imshow('Mandrill', img)
cv2.waitKey(0)
cv2.destroyAllWindows('Mandrill')
```

The `cv2.imshow()` function displays an image in a window on the screen. It accepts two arguments. The string that is the name of the window is the first argument, and the NumPy ndarray variable that has the image to be displayed is the second variable.

The `cv2.waitKey()` function is a function used for binding the events for the keyboard. It accepts an argument, which is the number of milliseconds the function needs to wait to detect the keypress of the keyboard. If we pass it 0, it waits for the press of a key on the keyboard indefinitely. It is the only function in the OpenCV library that can handle the events of the keyboard. We must call it immediately after the call of the `cv2.imshow()` function. If we do not call it that way, no window for the image will be displayed on the screen as `cv2.waitKey()` is the only function that fetches and handles events.

The `cv2.destroyAllWindows()` function accepts the name of the windows to be destroyed as an argument. When all the windows the current program displays must be destroyed, we use the `cv2.destroyAllWindows()` function to do this. We will use these functions in almost all of the OpenCV programs throughout this book.

We can also create a window in advance that has a specific name, and then associate an image with that window later in the program when we need it. It is recommended that we create a window in advance before we process an image. The following code snippet demonstrates this:

```
cv2.namedWindow('Lena', cv2.WINDOW_AUTOSIZE)
cv2.imshow('Mandrill', img)
cv2.waitKey(0)
cv2.destroyWindow('Mandrill')
```

Let's put it all together to get the following script:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
cv2.imshow('Mandrill', img)
cv2.waitKey(0)
cv2.destroyWindow('Mandrill')
```

The preceding code imports an image, displays it on the screen, and then waits for a keystroke on the keyboard to close the image window. The following is a screenshot of the output of the preceding code:

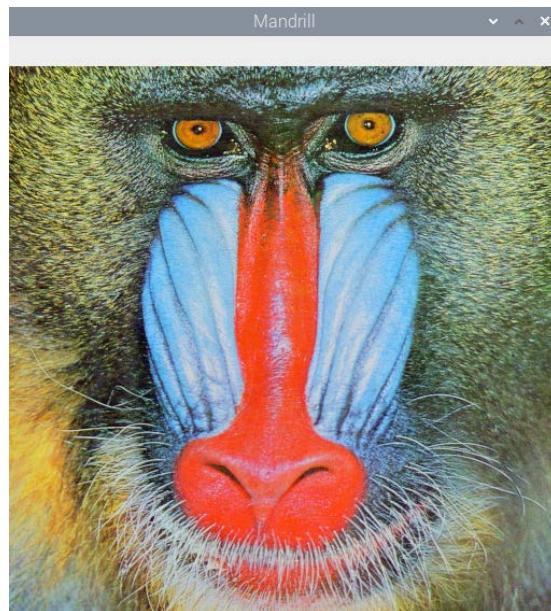


Figure 1: Reading and visualizing a color image with OpenCV

The `cv2.imwrite()` function saves a NumPy ndarray to a specific path on the disk. The first argument is the string that is the name of the file with which we want to save the image, and the second argument is the name of the NumPy array that has the image. Additionally, the `cv2.waitKey()` function can detect specific keystrokes on the keyboard. Let's look at a demonstration of both of the functions as follows:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
cv2.imshow('Mandrill', img)
keyPress = cv2.waitKey(0)
if keyPress == ord('q'):
    cv2.destroyAllWindows('Mandrill')
elif keyPress == ord('s'):
    cv2.imwrite('test.jpg', img)
    cv2.destroyAllWindows('Mandrill')
```

Here, the line `keyPress = cv2.waitKey(0)` saves the value of the keystroke on the keyboard in the `keyPress` variable. The `ord()` function accepts a single character and returns an integer that represents the Unicode of the character if it is a Unicode object. Based on the value of the `keyPress` variable, we can either exit straight away, or exit after saving the image to the disk. For example, if we press the *Esc* key, the `cv2.waitKey()` function returns the value of 27.

Using matplotlib to visualize images

Matplotlib is a very robust data visualization library for the Python 3 programming language. It is also capable of visualizing images. It also offers a wide range of options for plotting, and we will learn many of its capabilities in the later chapters of this book. Let's write a program that displays an image with matplotlib that we read in grayscale mode using the OpenCV `cv2.imread()` function:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 0)

import matplotlib.pyplot as plt
plt.imshow(img)
plt.title('Mandrill')
plt.axis('off')
plt.show()
```

Note

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

In the preceding example, we are reading an image in grayscale mode and displaying it with the matplotlib `plt.imshow()` function. The following is the output of the preceding program:

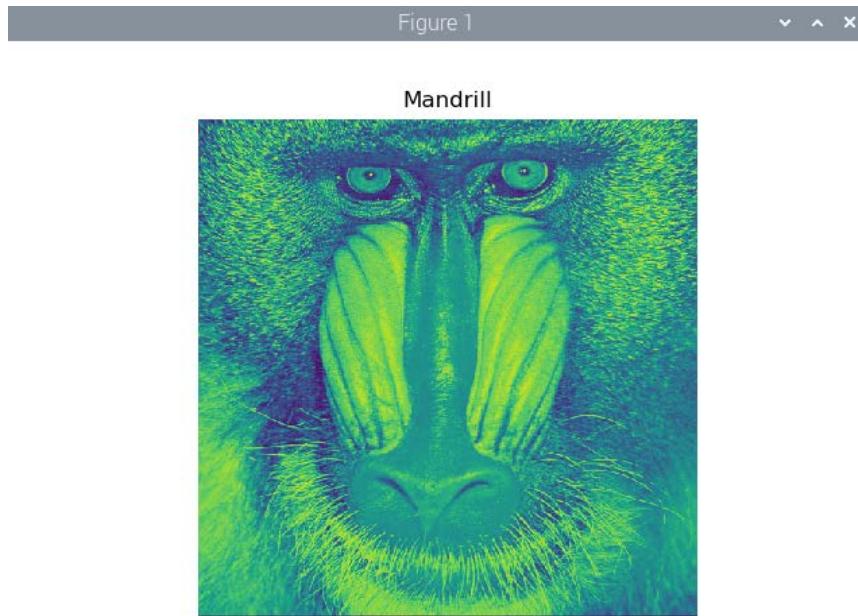


Figure 2: Visualizing a BGR image as an RGB image with matplotlib

I know that the image does not look natural. This is because we are reading the image in grayscale mode and visualizing it with the default colormap. Make the following changes in `plt.imshow()`, and we will find the output more palatable to our eyes. The following is the output:

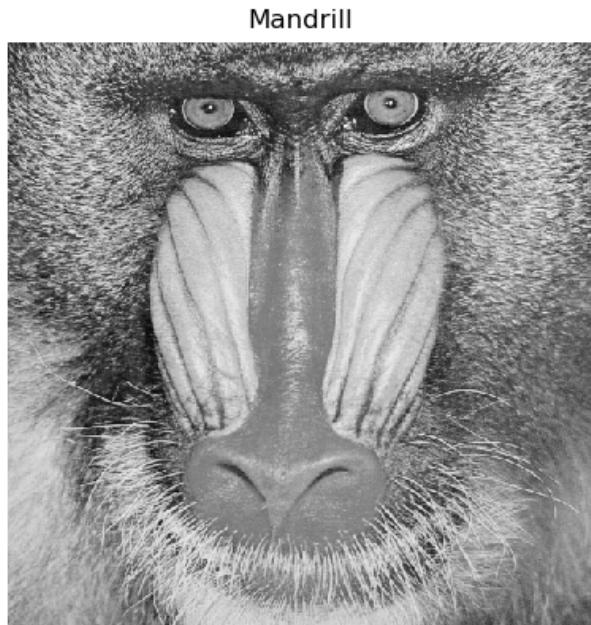


Figure 3: Visualizing a grayscale image

This was all about grayscale images.

The `cv2.imread()` function also works with color images. It reads and saves them as a three-dimensional ndarray of **Blue, Green, and Red (BGR)** pixels.

However, the `matplotlib plt.imshow()` function displays NumPy ndarrays as images in the RGB colorspace. If we read an image with the `cv2.imread()` function in the default BGR format of OpenCV and show it with the `plt.imshow()` function, the `plt.imshow()` function will treat the value for the intensity of the blue color as the value of the intensity of the red color and vice versa. This will make the image appear with distorted colors. Make the following change to the preceding code in the respective lines and run the program again:

```
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
plt.imshow(img)
```

Make the changes and run the code to see the color image with distorted colors. To solve this issue, we must convert the image read in the BGR colorspace by the `cv2.imread()` function to the RGB colorspace so that the `plt.imshow()` function will be able to render it in a fashion that makes sense to the human eyes and brain. We will use the `cv2.cvtColor()` function for this task, and we will learn about this in more detail later on in this book.

Drawing geometric shapes with OpenCV and NumPy

Let's learn how to draw various geometric shapes using the OpenCV drawing functions. We will also use NumPy here.

The following code imports all the required libraries for this demonstration:

```
import cv2  
import numpy as np
```

The following code creates an RGB ndarray of all zeros. It is an image in which all the pixels are black:

```
image = np.zeros((200, 200, 3), np.uint8)
```

We are using the `np.zeros()` function to create an ndarray of all zero elements.

We'll start by drawing a line, as it is a simple geometric shape. With the help of the following code, we'll draw a line with coordinates (0, 199) and (199, 0), with red color [(0, 0, 255) in BGR], and with a thickness of 2 pixels:

```
cv2.line(image, (0, 199), (199, 0), (0, 0, 255), 2)
```

All the OpenCV functions for drawing have the following common parameters:

- `img`: This is the image where we need to draw the geometric shapes.
 - `color`: This is passed as a tuple of **(B, G, R)** to express the colors where the value of the intensity of each color is in between 0 and 255.
 - `thickness`: The default value of the argument for this parameter is 1. For all the shapes that are geometrically closed, such as an ellipse, a circle, and a rectangle, -1 completely fills in the shape with the color specified as an argument.
 - `LineType`: This can have any one of the following three values:
 - 8: Eight-connected lines (this is the default value of the argument for this parameter).
 - 4: Four-connected lines.
- `cv2.LINE_AA`: This stands for **anti-aliasing** (it is usually used with geometric shapes that have curves such as an ellipse or circle).

The following line of code will help us to draw a rectangle with (20, 20) and (60, 60) as diagonally opposite vertices and the color blue:

```
cv2.rectangle(image, (20, 20), (60, 60), (255, 0, 0), 1)
```

The following line of code will help us to draw a circle with the center located at (80, 80), 10 pixels as the radius, and green as the fill color:

```
cv2.circle(image, (80, 80), 10, (0, 255, 0), -1)
```

The following line of code will help us to draw a full ellipse with no rotations, a center located at pixels (99, 99), and the lengths of the major and minor axes as 40 pixels and 20 pixels, respectively:

```
cv2.ellipse(image, (99, 99), (40, 20), 0, 0, 360, (128, 128, 128), -1)
```

The following code plots a polygon that has four points. It is defined as follows:

```
points = np.array([[100, 5], [125, 30], [175, 20], [185, 10]], np.int32)
points = points.reshape((-1, 1, 2))
cv2.polyline(image, [points], True, (255, 255, 0))
```

If we pass `False` as the value for the third argument in the call of the `polyline()` function, it joins all the points with the line segments and will not plot a closed shape.

We can also print text in the image using the `cv2.putText()` function. The following code adds the text to the image with (80, 180) as the bottom-left corner of the text, `HERSHEY_DUPLEX` as the font with size of text 1, and the color of the text as pink:

```
cv2.putText(image, 'Test', (80, 180), cv2.FONT_HERSHEY_DUPLEX, 1, (255, 0, 255))
```

The `cv2.putText()` function accepts one of the following fonts as an argument:

- `FONT_HERSHEY_DUPLEX`
- `FONT_HERSHEY_COMPLEX`
- `FONT_HERSHEY_SIMPLEX`

- FONT_HERSHEY_PLAIN
- FONT_HERSHEY_SCRIPT_SIMPLEX
- FONT_HERSHEY_SCRIPT_COMPLEX
- FONT_HERSHEY_TRIPLEX
- FONT_HERSHEY_COMPLEX_SMALL

The output image is shown using this familiar snippet of code:

```
cv2.imshow('Shapes', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The output of the preceding code is as follows:

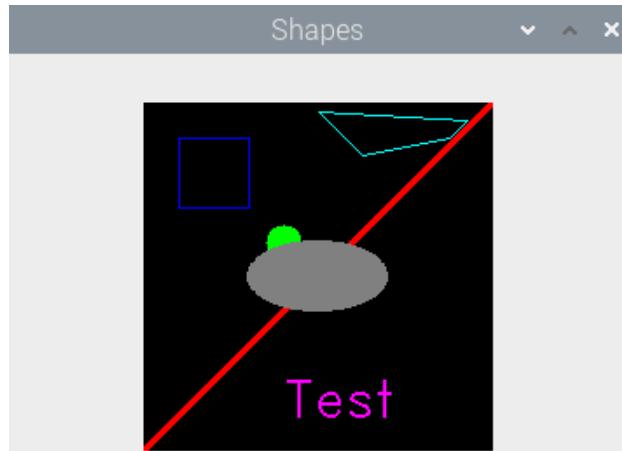


Figure 4: Drawing geometric shapes

Note

If the pixels of the geometric shapes overlap, then those pixels will always have the value that is assigned by the latest geometric function. For example, the ellipse overlaps the line and the circle in the preceding figure.

As an exercise, change the values of the arguments passed to all the geometric functions and run the code again to understand the functionality better.

Working with a GUI

By now we are aware of how to create a named window using the call of the OpenCV `cv2.namedWindow()` function. We will now demonstrate how to create trackbars using the `cv2.CreateTrackbar()` function, how to associate it with a named window, and how to use those trackbars to choose the value of the color channels in the RGB colorspace. Let's get started with the following code:

```
import numpy as np
import cv2
def empty(z):
    pass
image = np.zeros((300, 512, 3), np.uint8)
cv2.namedWindow('Palette')
cv2.createTrackbar('B', 'Palette', 0, 255, empty)
cv2.createTrackbar('G', 'Palette', 0, 255, empty)
cv2.createTrackbar('R', 'Palette', 0, 255, empty)
while(True):
    cv2.imshow('Palette', image)
    if cv2.waitKey(1) == 27 :
        break
    blue = cv2.getTrackbarPos('B', 'Palette')
    green = cv2.getTrackbarPos('G', 'Palette')
    red = cv2.getTrackbarPos('R', 'Palette')
    image[:] = [blue, green, red]
cv2.destroyAllWindows()
```

In the preceding code, we first create an image with all the pixels colored black and a named window with the name of `Palette`. The `cv2.createTrackbar()` function creates a trackbar. The following is the list of arguments accepted by this function:

- **Name:** The name of the trackbar.
- **Window_name:** The name of the output window the trackbar is to be associated with.
- **Value:** The initial value of the slider of the trackbar when it is created.
- **Count:** The maximum value of the slider of the trackbar (the minimum value of the slider is always 0).
- **Onchange ():** This function is called when we change the position of the slider of the trackbar.

We have created a function and named it `empty()`. We do not intend to perform any activity when we change the slider of the trackbar. We are just passing the call of this function to the `cv2.createTrackbar()` function. The call of the `cv2.getTrackbarPos()` function returns the most recent position of the slider of the trackbar. Based on the positions of the sliders of all three trackbars, we set the color of the palette. The application closes when we press the *Esc* key on the keyboard. The application we created should look like this:

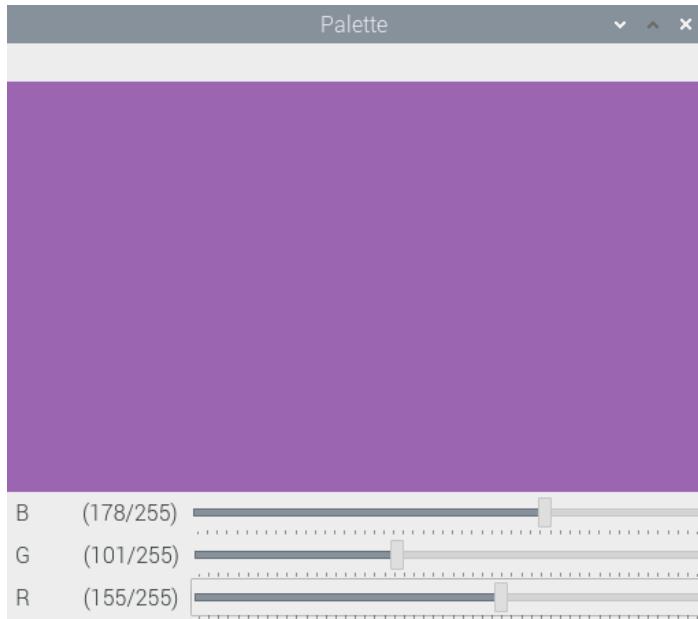


Figure 5: A BGR color palette

OpenCV also offers a lot of functionality to handle events. We will explore that next.

Event handling and a primitive paint application

A variety of keyboard and mouse events are recognized by OpenCV. We can view the list of events by following these instructions. Open the Python 3 Interpreter in interactive mode by running the `python3` command on Command Prompt, and then run the following statements:

```
>>> import cv2  
>>> events = [i for i in dir(cv2) if 'EVENT' in i]  
>>> print(events)
```

It will show the following output:

```
[ 'EVENT_FLAG_ALTKEY', 'EVENT_FLAG_CTRLKEY', 'EVENT_FLAG_LBUTTON',
  'EVENT_FLAG_MBUTTON', 'EVENT_FLAG_RBUTTON', 'EVENT_FLAG_SHIFTKEY',
  'EVENT_LBUTTONDOWNDBLCLK', 'EVENT_LBUTTONDOWN',
  'EVENT_LBUTTONUP', 'EVENT_MBUTTONDOWNDBLCLK', 'EVENT_MBUTTONDOWN',
  'EVENT_MBUTTONUP', 'EVENT_MOUSEWHEEL', 'EVENT_MOUSEMOVE',
  'EVENT_MOUSEWHEEL', 'EVENT_RBUTTONDOWNDBLCLK', 'EVENT_RBUTTONDOWN',
  'EVENT_RBUTTONUP' ]
```

We can write code to handle a couple of these events and create a simple and primitive paint application. Let's import the required libraries using the following code:

```
import cv2
import numpy as np
```

Create a black background and a named window:

```
windowName = 'Drawing'
img = np.zeros((512, 512, 3), np.uint8)
cv2.namedWindow(windowName)
```

Define a custom function, called `draw_circle()`:

```
def draw_circle(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWNDBLCLK:
        cv2.circle(img, (x, y), 40, (0, 255, 0), -1)
    if event == cv2.EVENT_MBUTTONDOWN:
        cv2.circle(img, (x, y), 20, (0, 0, 255), -1)
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(img, (x, y), 30, (255, 0, 0), -1)
```

In the preceding definition, we are drawing circles using various properties on the mouse events. Now, let's call the `setMouseCallback()` function and pass it the name of the window and the `draw_circle()` function as arguments:

```
cv2.setMouseCallback(windowName, draw_circle)
```

This call will bind the `draw_circle()` function with the given window's mouse events. Finally, we write the loop for displaying the image window and exit when the *Esc* key is pressed:

```
while(True):
    cv2.imshow(windowName, img)
    if cv2.waitKey(20) == 27:
        break

cv2.destroyAllWindows()
```

Run the entire code and you will see the following output:

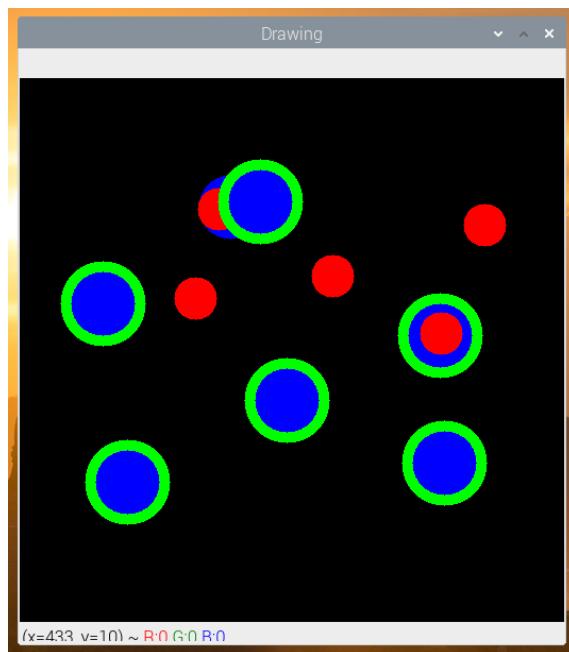


Figure 6: A simple paint application

As we have programmed the left, middle, and down button's double-click events, depending on these events and the location of the cursor, your output will be different.

We will use the drawing API in OpenCV sparingly in this book. The functionality that we will use the most throughout this book is related to the webcam. The next section is dedicated to the interfacing and use of the webcam with OpenCV and the Raspberry Pi.

Working with a USB webcam

Cameras are image sensors. That said, analog cameras and motion film cameras record images on films. Digital cameras have digital sensors to capture the image and these are stored in electronic formats on various types of storage mediums. A subset of digital cameras is USB webcams. These webcams, as their name indicates, can be interfaced to a computer via USB, hence the name, USB webcam. In this section, we will learn about the interfacing of USB webcams with the Raspberry Pi and programming using shell scripts, Python 3, and OpenCV in detail.

Note

All these webcams work with Raspberry Pi boards. However, a few webcams may have issues. The https://elinux.org/RPi_USB_Webcams URL has a list of many webcams and details regarding compatibility.

All the programs in this book are tested with the RPi 4B and a Logitech C310 webcam. You can view its product page at <https://www.logitech.com/en-in/product/hd-webcam-c310>.

Attach the USB webcam to the RPi using the USB port on the board and run the following command in the Terminal:

```
lsusb
```

The output of this command shows the list of all the USB devices connected to the Linux computer. The following is the output shown on my RPi board:

```
pi@raspberrypi:~/book/chapter04 $ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 005: ID 046d:081b Logitech, Inc. Webcam C310
Bus 001 Device 004: ID 1c4f:0002 SiGma Micro Keyboard TRACER
Gamma Ivory
Bus 001 Device 003: ID 046d:c077 Logitech, Inc. M105 Optical
Mouse
Bus 001 Device 002: ID 2109:3431 VIA Labs, Inc. Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

As you can see in the preceding output, the second entry corresponds to the USB webcam that is connected to the RPi board. We can also see a USB mouse and a USB keyboard connected to the RPi board.

Capturing images with the webcam

Let's now demonstrate how to capture images with the USB webcam attached to the RPi. We can install the `fswebcam` utility by running the following command on the terminal:

```
sudo apt-get install fswebcam
```

Once we install it, we can run the following command to capture a photograph with the USB webcam:

```
fswebcam -r 1280x960 --no-banner ~/book/chapter04/camtest.png
```

This command captures an image with a resolution of 1280 x 960 pixels using the USB webcam connected to the RPi. The command-line `--no-banner` argument passed to the command disables the banner for the timestamp. The image is saved with the filename passed as the last argument. If we run this command repeatedly, the new photograph that is captured will be overwritten to the same file. So, next time we run the command, we must pass a different filename as a parameter to the command if we do not want to overwrite the earlier file.

Note

If you want to read more about `fswebcam` (or any Linux command for that matter), you can run the `man fswebcam` command on Command Prompt.

Timelapse photography

Capturing photographs at regular intervals with a camera and playing them back at a higher frame rate than they were captured at is known as **timelapse photography**. For example, if we capture photographs at the rate of one photograph per minute for 10 hours, we will have 600 photographs. If we stitch all of them into a video and play it back with a frame rate of 30 photos per second, we will have a 20-second video. This video is known as a **timelapse video**. We can use the RPi board with a USB webcam for this. We have already learned how to use a USB webcam with an RPi board to do this. We have also learned the usage of the `fswebcam` utility. We will write a script that captures images with timestamps in the filename. Then, we will add this script to the crontab to execute it at regular intervals. Cron is a job scheduler for Unix-like OSes. It is driven by a file named `crontab` (cron table). It is a Unix configuration file that specifies the scripts or programs to be run at a particular time or interval.

Let's create a shell script with the name of `timelapse.sh` and save it in a location of our choice on the disk. We must add the following code to the script file and save it:

```
#!/bin/bash  
DATE=$(date +"%Y-%m-%d %H%M")  
fswebcam -r 1280x960 --no-banner Image_$DATE.png
```

Let's make the mode of the script executable by running the following command:

```
chmod +x timelapse.sh
```

This script takes a photograph using the USB webcam and then saves it to a location in the disk. The captured image has a new filename every time because the filename has a timestamp when the image is captured. We must execute this script manually once to make sure it works without any problem and that it captures an image in the filename format of `Image_<timestamp>.png`.

Once the script is checked for any issues, it must be executed at regular intervals to capture images for the timelapse sequence. For that to happen, we must add it to the crontab. The syntax for an entry in the crontab is as follows:

```
1 2 3 4 5 /location/command
```

Let's check the meaning of the terms in the syntax:

- 1: Position of minutes (can range from 0-59)
- 2: Position of hours (can range from 0-23)
- 3: Position of the day of the month (can range from 0-31)
- 4: Position of the month (can range from 0-12 [1 for January])
- 5: Position of the day of the week (can range from 0-7 [7 or 0 for Sunday])
- /location/command: Script or command name to schedule

Therefore, the entry for the crontab to run the `timelapse.sh` script once every minute is as follows:

```
* * * * * /home/pi/book/chapter04/timelapse.sh 2>&1
```

Open the crontab of user pi using the following command:

```
crontab -e
```

This will open the crontab. When we execute this for the very first time on our RPi, it will ask which text editor to choose. Choose the **Nano** option by entering 1. Add the preceding line to the crontab as an entry. Then save and exit it.

Once we exit the crontab, it will show us the following message:

```
crontab: installing new crontab
```

Once we do this, our setup for the timelapse is live. We can change the settings in the entry of the crontab to the settings of our choice. To run the script every 5 minutes, use the following:

```
*/5 * * * * /home/pi/book/chapter04/timelapse.sh 2>&1
```

To run the script every 2 hours, use the following:

```
* */2 * * * /home/pi/book/chapter04/timelapse.sh 2>&1
```

Once we capture all the images for our timelapse, we must encode them in a video that has a frame rate of 24, 25, 29, or 30 **frames per second (FPS)**. These are all the standard frame rates. I prefer to encode the video using 30 FPS. Raspberry Pi is a slow computer for video editing. It is recommended that you copy the images to a faster computer to encode the video. For Linux computers, I prefer to use the command-line **MEncoder** utility. We can use the other utilities or video editing tools for this task too. The following are the steps needed to create a timelapse video with **MEncoder** on the Raspberry Pi or any other Linux computer:

1. Install MEncoder using the following command on Command Prompt:

```
sudo apt-get install mencoder -y
```

2. Navigate to the output directory by issuing the following command:

```
cd /home/pi/book/chapter04
```

3. Create a list of the images to be used in our timelapse sequence using the following command:

```
ls Image_*.png > timelapse.txt
```

- Finally, we can use the following command to create a nice timelapse video:

```
mencoder -nosound -oov lavc -lavcopts  
vcodec=mpeg4:aspect=16/9:vbitrate=8000000 -vf  
scale=1280:960 -o timelapse.avi -mf type=jpeg:fps=30  
mf://@timelapse.txt
```

This creates the video with the `timelapse.avi` filename in the current directory where we are running the command (also known as the **present working directory**). The frame rate of the video will be 30 FPS. Very soon, we will learn how to play this video file.

Webcam video recording

We can use the USB webcam connected the RPi to record live videos using the command-line `ffmpeg` utility. We can install the `ffmpeg` utility using the following command:

```
sudo apt-get install ffmpeg
```

We can use the following command to record a video:

```
ffmpeg -f video4linux2 -r 25 -s 544x288 -i /dev/video0 test.avi
```

We can terminate the operation of recording the video by pressing *Ctrl + C* on the keyboard.

We can play the video using the command-line `omxplayer` utility. It comes preinstalled with the latest release of Raspbian, so we do not have to install it separately. To play a file with the `timelapse.avi` filename, navigate to the location of the video file using Command Prompt and run the following command:

```
omxplayer timelapse.avi
```

We can even double-click on the video files in the Raspbian GUI to play them with VLC media player.

Capturing images with the webcam using Python and OpenCV

Let's learn how to capture images with the webcam connected to the RPi using Python 3 and OpenCV:

```
import cv2  
import matplotlib.pyplot as plt  
cap = cv2.VideoCapture(0)
```

```
if cap.isOpened():
    ret, frame = cap.read()
else:
    ret = False
print(ret)
print(type(frame))
cv2.imshow('Frame from Webcam', frame)
cv2.waitKey(0)
cap.release()
cv2.destroyAllWindows()
```

In the previous code snippet, the `cv2.VideoCapture()` function creates an object to capture the video using the webcam connected to the RPi. The argument for it could either be the index of the video device or a video file. In this case, we are passing the index of the video device, which is 0. If we have more cameras connected to the RPi board, we can pass the appropriate device index based on which camera is chosen. If we have connected only one camera, then we just pass 0.

We can find out the number of cameras and device indexes associated with those cameras by running the following command:

```
ls -l /dev/video*
```

The `cap.read()` function returns a Boolean `ret` value and a NumPy `frame` ndarray that contains the image it captured. If the operation of capturing the image is successful, then `ret` will have a Boolean value of `True`; otherwise, it will have a Boolean value of `False`. The preceding code captures an image using the USB camera identified by `/dev/video0`, displays it on the screen, and then finally saves it to the disk with the filename `test.png`. The `cap.release()` function releases the video capture device.

Live videos with the webcam using Python and OpenCV

We can use the previous code with a few modifications to display a live video stream from a USB webcam:

```
import cv2
windowName = "Live Video Feed"
cv2.namedWindow(windowName)
cap = cv2.VideoCapture(0)
```

```
if cap.isOpened():
    ret, frame = cap.read()
else:
    ret = False
while ret:
    ret, frame = cap.read()
    cv2.imshow(windowName, frame)
    if cv2.waitKey(1) == 27:
        break

cv2.destroyAllWindows()
cap.release()
```

The previous code shows the live video captured by the webcam until we press the *Esc* key on the keyboard. The preceding code example is the template for all the code examples for the processing of live videos captured using the USB webcam connected to the RPi board.

Webcam resolution

We can read the properties of the webcam using `cap.get()`. We must pass 3 to get the width and 4 to get the height. We can also set the properties with `cap.set()` in the same way. The following code demonstrates this:

```
import cv2
windowName = "Live Video Feed"
cv2.namedWindow(windowName)
cap = cv2.VideoCapture(0)
print('Width : ' + str(cap.get(3)))
print('Height : ' + str(cap.get(4)))
cap.set(3, 5000)
cap.set(4, 5000)
print('Width : ' + str(cap.get(3)))
print('Height : ' + str(cap.get(4)))
if cap.isOpened():
    ret, frame = cap.read()
else:
    ret = False
```

```
while ret:  
    ret, frame = cap.read()  
    cv2.imshow(windowName, frame)  
    if cv2.waitKey(1) == 27:  
        break  
cv2.destroyAllWindows()  
cap.release()
```

In the preceding code, we are setting both the height and width to 5000. The webcam does not support this resolution, so the height and width will both be set to the maximum resolution supported by webcam. Run the preceding code and observe the output printed in Command Prompt.

FPS of the webcam

We can retrieve the FPS of the webcam we are using, and we can also calculate the actual FPS ourselves. The FPS that we retrieve as a property of the webcam and the calculated FPS could be different. Let's check this. Import all the required libraries:

```
import time  
import cv2
```

Next, initiate an object for the video capture:

```
cap = cv2.VideoCapture(0)
```

We can fetch the camera resolution using `cap.get()`, as follows:

```
fps = cap.get(cv2.CAP_PROP_FPS)  
print("FPS with CAP_PROP_FPS : {}".format(fps))
```

Then, we will capture 120 frames continuously. We record the time before and after the operation as follows:

```
num_frames = 120  
print("Capturing {} frames".format(num_frames))  
start = time.time()  
for i in range(0, num_frames):  
    ret, frame = cap.read()  
end = time.time()
```

Then, finally, we compute the actual time required to capture the frames, and we can calculate the FPS using the following formula:

The code is as follows:

```
seconds = end - start
print("Time taken : {0} seconds".format(seconds))
fps = num_frames / seconds
print("Actual FPS calculated : {0}".format(fps))
cap.release()
```

Run the entire program, and the output should be like this:

```
FPS with CAP_PROP_FPS : 30.0
Capturing 120 frames
Time taken : 9.86509919166565 seconds
Actual FPS calculated : 12.164094619685105
```

We usually never get the FPS retrieved from the properties due to hardware limitations.

Saving webcam videos

We use the OpenCV `cv2.VideoWriter()` function to save the live USB webcam stream to a video file on the disk. The following code demonstrates this:

```
import cv2

windowName = "Live Video Feed"
cv2.namedWindow(windowName)
cap = cv2.VideoCapture(0)
filename = 'output.avi'
codec = cv2.VideoWriter_fourcc('W', 'M', 'V', '2')
framerate = 30
resolution = (640, 480)
Output = cv2.VideoWriter(filename, codec,
                        framerate, resolution)
```

```
if cap.isOpened():
    ret, frame = cap.read()
else:
    ret = False

while ret:
    ret, frame = cap.read()
    Output.write(frame)
    cv2.imshow(windowName, frame)
    if cv2.waitKey(1) == 27:
        break

cv2.destroyAllWindows()
cap.release()
```

In the preceding code, the call of the `cv2.VideoWriter()` function accepts the arguments for the following parameters:

- `filename`: This is the name of the video file to be written on the disk.
- `fourcc`: This means the **four-character code**. We use the `cv2.VideoWriter_fourcc()` function for this. This function accepts a four-character code as an argument. A few supported four-character code formats are WMV2, MJPG, H264, WMV1, DIVX, and XVID. You can read more about four-character codes at <http://www.fourcc.org/codecs.php>.
- `framerate`: This refers to the FPS of the video to be captured.
- `resolution`: This is the resolution in pixels with which the video is to be captured and saved on the disk.

The preceding code records the video until the *Esc* key on the keyboard is pressed, and then saves it on the disk with the filename specified in the argument of the `cv2.VideoWriter()` function.

Playing back the video with OpenCV

We can easily play back the video using OpenCV. We just need to pass the name of the video file to the `VideoCapture()` function in place of the index of the webcam (which is 0, in our case). In order to decide the FPS for the playback, we need to pass the appropriate argument to the call of the `waitKey()` function. Suppose we want to play back the video at 25 FPS, then the argument to be passed can be calculated with the $1000/25 = 40$ formula. We know that `waitKey()` waits for the number of milliseconds we pass to it as an argument. And, a second has 1,000 milliseconds, hence the formula. For 30 FPS, this will be 33.3. Let's take a look at the following code:

```
import cv2

windowName = "OpenCV Video Player"
cv2.namedWindow(windowName)
filename = 'output.avi'
cap = cv2.VideoCapture(filename)

while(cap.isOpened()):
    ret, frame = cap.read()
    if ret:
        cv2.imshow(windowName, frame)
        if cv2.waitKey(33) == 27:
            break
    else:
        break

cv2.destroyAllWindows()
cap.release()
```

The preceding program plays the video file with a frame rate of 30 FPS and terminates after the last frame or when we press the *Esc* key on the keyboard. You might want to play with the program and try to change the output frame rate by changing the value of the argument to the call of the `cv2.waitKey()` function.

In the next section, we will study the Pi camera module in more detail.

The Pi camera module

A webcam uses a USB port for interfacing with a computer. That is why we can use it with any computer that has a USB port. The Pi camera modules (also known as **Pi camera boards**) are sensors that are specifically manufactured for RPi boards. The Raspberry Pi Foundation and many other third-party manufacturers produce them. Basically, they are PCBs with a specialized image sensor on them (that is why they are known as Pi camera boards).

The Pi camera board does not have a USB port. It connects to Raspberry Pi through a **Camera Serial Interface (CSI)** interface strip. Because of the dedicated connection that uses CSI, the performance of a Pi camera board is much better than a USB webcam. We can use Python 3 with a Pi camera module connected to the RPi to capture videos and still images programmatically. It is not possible to use the Pi camera board with any computer other than a Raspberry Pi (and a select few single-board computers that support connectivity with the camera module).

The camera modules are offered in two varieties—the camera module and the NoIR module. The camera module is great for daytime and well-illuminated scenes. The NoIR module is essentially a camera module without the **Infrared (IR)** filter. It does not produce impressive results during the daytime or in well-illuminated scenes. However, it is great in low light or in dark scenes when used with IR light.

You can find the latest versions of both of these modules at their product pages at <https://www.raspberrypi.org/products/camera-module-v2/> and <https://www.raspberrypi.org/products/pi-noir-camera-v2/>. There have been generations of these camera boards/modules, V1 and V2. The V1s are of 5 megapixels and are no longer produced. The V2s are the latest and they have an 8-megapixel sensor. You can read about the differences between them at <https://www.raspberrypi.org/documentation/hardware/camera/>.

All the cameras come with a detachable ribbon that can be used to connect the camera to the RPi boards using the **Camera Serial Interface (CSI)** port. The following is a photograph of a camera module and the ribbon:



Figure 7: The Pi camera board and the CSI interface ribbon

We must connect the blue end to the CSI port of the RPi board and the other end to the camera board.

The RPi Zero and RPi Zero W are equipped with smaller CSI ports. There are separate ribbons for them. The following is a photograph of such a ribbon:



Figure 8: Mini CSI ribbon

The following is a photograph of a Pi NoIR board connected to an RPi Zero board:

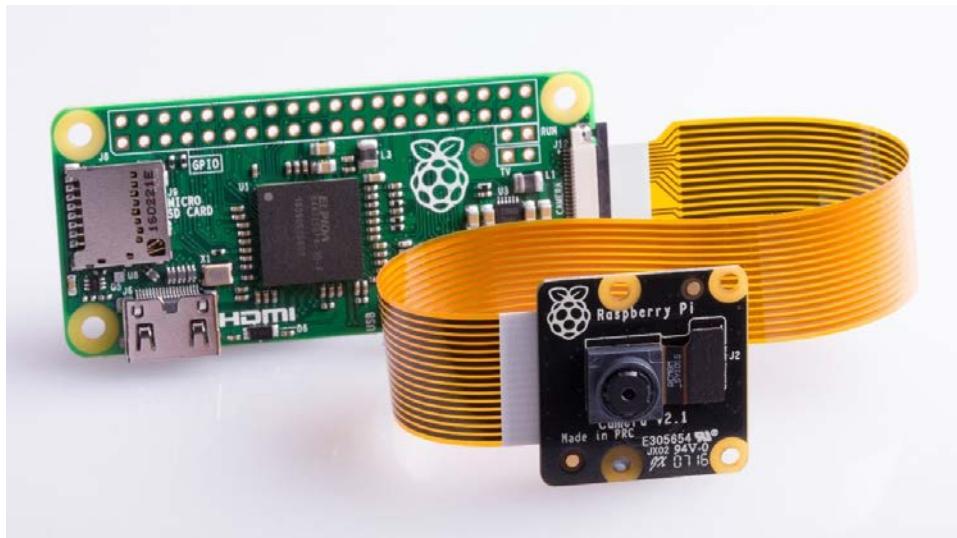


Figure 9: Pi NoIR with RPi Zero

I have already mentioned that the Pi Camera V1 is not in production anymore. You will find a lot of these V1 modules at low prices (from \$5 to \$7) online. Additionally, there are other manufacturers that produce similar boards that are compatible with RPi CSI ports. They are also available online for purchase.

Capturing images and videos with the raspistill and raspivid utilities

In order to capture still photographs and motion videos using the camera module of the RPi, we need to use the command-line `raspistill` and `raspivid` utilities. To capture an image, run the following command:

```
raspistill -o test.png
```

This command captures and saves an image in the current directory with the `test.png` filename.

To capture a 20-second video with the RPi camera module, run the following command in Command Prompt:

```
raspivid -o vid.h264 -t 20000
```

Unlike the `fswebcam` and `ffmpeg` utilities, the `raspistill` and `raspivid` utilities do not write anything to Command Prompt. So, we must check the current directory for any output. Additionally, we can run the following command after executing the `raspistill` and `raspivid` utilities to check whether these commands have been executed successfully:

```
echo $?
```

Many computers and OSes cannot play videos in the H.264 format directly. For that, we need to wrap them in the popular and widely supported MP4 format. To do this, we need a command-line utility known as MP4Box. We can install it by running the following command on Command Prompt:

```
sudo apt install -y gpac
```

Now, record an H.264 video:

```
raspivid -t 30000 -w 640 -h 480 -fps 25 -b 1200000 -p  
0,0,640,480 -o pivideo.h264
```

Wrap it in the MP4 format and remove the original file (if you want to), as follows:

```
MP4Box -add pivideo.h264 pivideo.mp4  
rm pivideo.h264
```

Just like the `fswebcam` utility, the `raspistill` utility can also be used to capture a timelapse sequence. In the `timelapse.sh` shell script that we prepared earlier, replace the line that calls the `fswebcam` utility with the appropriate `raspistill` command to record a photograph of a timelapse sequence. Then, use the `MEncoder` utility on the RPi or any other Linux computer to create a nice timelapse video.

Using picamera with Python 3

`picamera` is a Python package that provides a programming interface to the RPi camera module. The most recent version of Raspbian has `picamera` installed. If you do not have it installed, you can install it by running the following commands:

```
pip3 install picamera  
pip3 install "picamera[array]"
```

The following program quickly demonstrates the basic usage of the `picamera` module to capture a picture:

```
from time import sleep
from picamera import PiCamera
camera = PiCamera()
camera.resolution = (1024, 768)
camera.start_preview()
sleep(2)
camera.capture('test.png')
```

We are importing the `time` and `picamera` libraries in the first two lines. The call to the `start_preview()` function starts the preview of the scene to be captured. The `sleep(5)` function waits for 5 seconds before the `capture()` function captures and saves the photo to the file specified in its arguments.

The `picamera` module offers the `capture_continuous()` function for timelapse photography. Let's demonstrate how to use it in the following program:

```
camera = PiCamera()
camera.start_preview()
sleep(2)
for filename in camera.capture_continuous('img{counter:03d}.png'):
    print('Captured %s' % filename)
    sleep(1)
```

In the preceding code, the `capture_continuous()` function records the photographs for a timelapse sequence with the Pi camera board connected to the RPi. In this way, we do not have to depend on the `crontab` utility to continuously call the script because we can control it better programmatically.

We can record videos by using the `start_recording()`, `wait_recording()`, and `stop_recording()` functions, as follows:

```
import picamera
camera = picamera.PiCamera()
camera.resolution = (320, 240)
camera.start_recording('my_video.h264')
camera.wait_recording(5)
camera.stop_recording()
```

We can add text to the images as follows:

```
from time import sleep
from picamera import PiCamera
camera = PiCamera()
camera.resolution = (1024, 768)
camera.start_preview()
camera.annotate_text = 'Hello World!'
sleep(2)
camera.capture('test.png')
```

We can store an image in a three-dimensional NumPy array as follows:

```
import time, picamera
import numpy as np
with picamera.PiCamera() as camera:
    camera.resolution = (320, 240)
    camera framerate = 24
    time.sleep(2)
    output = np.empty((240, 320, 3), dtype=np.uint8)
    camera.capture(output, 'rgb')
    print(type(output))
```

We can also store the image captured in a NumPy array that is compatible with the OpenCV image format (BGR), as follows:

```
import time, picamera
import numpy as np
with picamera.PiCamera() as camera:
    camera.resolution = (320, 240)
    camera framerate = 24
    time.sleep(2)
    image = np.empty((240 * 320 * 3, ), dtype=np.uint8)
    camera.capture(image, 'bgr')
    image = image.reshape((240, 320, 3))
    print(type(image))
```

This stores the image in OpenCV's preferred BGR format. We can use the `cv2.imshow()` function to display this image.

Using the RPi camera module and Python 3 to record videos

We have already learned how to record a video with a USB webcam connected to RPi and the combination of Python 3 and OpenCV. I have noticed that the same code works for the RPi camera module too. We just need to connect the RPi camera module to the RPi and disconnect the USB webcam in order for the code to work with the RPi camera module and record videos using the code. Please go ahead and give it a try!

Summary

In this chapter, we learned how to work with images and videos. We also learned how to capture images with a USB webcam and RPi camera board. We also learned the basics of GUIs and the event handling functionality offered by OpenCV. We have gained good hands-on experience of shell and Python 3 programming. We will be using the image and video acquisition and handling techniques that we have learned here throughout the book.

In the next chapter, we will learn about the basics of image processing and how to write programs with NumPy and OpenCV.s

5

Basics of Image Processing

In the previous chapter, we learned about and demonstrated various ways to capture images and videos for image processing and computer vision applications. We learned how to use Command Prompt and Python 3 programming extensively to read images and to interface with the USB webcam and the Raspberry Pi camera module.

In this chapter, we will look at how to perform basic arithmetic and logical operations on images with NumPy, OpenCV, and matplotlib. We will also learn about different color channels and image properties in detail.

The following is a list of the topics that will be covered in this chapter:

- Retrieving image properties
- Basic operations on images
- Arithmetic operations on images
- Blending and transitioning images
- Multiplying images by constants and one another
- Creating a negative of an image
- Bitwise logical operations on images

This chapter has a lot of hands-on exercises that use Python 3 programming. We will use a lot of concepts, such as reading images from a disk and visualizing them, that we learned in earlier chapters when we demonstrate operations on images in this chapter.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter05/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/2V8vzev>.

Retrieving image properties

We can retrieve and use many properties, such as the data type, the dimensions, the shape, and the size of bytes of an image with NumPy. Open the Python 3 interpreter by running the `python3` command in the command prompt. Then, run the following statements one by one:

```
>>> import cv2  
>>> img = cv2.imread('/home/pi/book/dataset/4.1.01.tiff', 0)  
>>> print(type(img))
```

The following is the output of these statements:

```
<class 'numpy.ndarray'>
```

The preceding output confirms that the OpenCV `imread()` function read an image and stored it in NumPy's `ndarray` format. The following statement prints dimensions of the image it read:

```
>>> print(img.ndim)  
2
```

The image is read in grayscale mode, which is why it is a two-dimensional image. It just has a single channel composed of intensities of grayscale. Now, let's see its shape:

```
>>> print(img.shape)  
(256, 256)
```

The preceding statement prints the height and width in pixels. Let's see the size of the image:

```
>>> print(img.size)  
65536
```

If we multiply the height and width of the image, too, we get the preceding number. Let's see the data type of the NumPy ndarray:

```
>>> print(img.dtype)  
uint8
```

This is an unsigned integer of 8 bits for storing the grayscale intensity value of a pixel. The intensities vary from 0 to 255, which are the limits of the 8-bit unsigned data type. Each pixel consumes some bytes in memory. Let's see how to find out how many bytes it consumes in total, as follows:

```
>>> print(img.nbytes)  
65536
```

Now, let's repeat the same exercise for a color image. For that, let's read the same image in color mode:

```
>>> img = cv2.imread('/home/pi/book/dataset/4.1.01.tiff', 1)  
>>> print(type(img))
```

The following is the output of this:

```
<class 'numpy.ndarray'>
```

Let's check the number of dimensions:

```
>>> print(img.ndim)  
3
```

We have read the image in color mode and it is a three-dimensional NumPy ndarray. One of those two dimensions represents the height and width and one of the dimensions represents the color channels. Let's check out the shape now:

```
>>> print(img.shape)  
(256, 256, 3)
```

The first two values represent the width and height in pixels. The last value represents the number of channels. These channels represent the intensity values for the blue, green, and red colors of a pixel. Let's see the size of the image:

```
>>> print(img.size)  
196608
```

If we multiply all three numbers in the earlier output (256, 256, and 3), we get a value of 19,6608. The data type of the ndarray will be the same (uint8). Let's confirm this:

```
>>> print(img.dtype)  
uint8
```

Let's see how many bytes the image occupies in the main memory:

```
>>> print(img.nbytes)  
196608
```

In the next section, we will learn about basic operations on images.

Basic operations on images

Let's perform a few basic operations, such as splitting and combining the channels of a color image and adding a border to an image. We will continue this demonstration in interactive mode. Let's import OpenCV and read a color image, as follows:

```
>>> import cv2  
>>> img = cv2.imread('/home/pi/book/dataset/4.1.01.tiff', 1)
```

For any image, the origin—the (0, 0) pixel—is the pixel at the upper-left corner. We can retrieve the intensity values for all the channels by running the following statement:

```
>>> print(img[10, 10])  
[34 38 44]
```

These are the intensity values of the blue, green, and red channels, respectively, for pixel (10, 10). If you only want to access an individual channel for a pixel, then run the following statement:

```
>>> print(img[10, 10, 0])  
34
```

The preceding output, 34, is the intensity of the blue channel. Similarly, we can access the green and red channels with `img[10, 10, 0]` and `img[10, 10, 1]`, respectively.

Splitting the image into channels

Let's write a simple program to split an image into its constituent channels. There are multiple ways to do this. OpenCV offers the `split()` function to do this. Let's see a demonstration of this:

```
>>> import cv2  
>>> img = cv2.imread('/home/pi/book/dataset/4.1.01.tiff', 1)  
>>> b, g, r = cv2.split(img)
```

The last statement in the preceding list splits the color image into its constituent channels. We can also separate the channels with a bit of a faster method by using the NumPy `ndarray` indices, as follows:

```
>>> b = img[:, :, 0]  
>>> g = img[:, :, 1]  
>>> r = img[:, :, 2]
```

The `split()` function is a bit costlier (computationally) than the previous NumPy indexing method. We can also merge the channels, as follows:

```
>>> img1 = cv2.merge((b, g, r))
```

The preceding code merges all the constituent channels to form the original image. You may also want to create a Python 3 script file, add all the preceding code to that, and visualize the image with the `cv2.imshow()` function.

Next, we will learn how to add a border to images.

Adding a border to an image

We can add borders to an image with the `copyMakeBorder()` function. It accepts the following arguments:

- `Src`: The image
- `top, bottom, left, right`: The width of the border in terms of the number of pixels

- `borderType` : The type of border. This can be one of the following types:
 - a) `cv2.BORDER_REFLECT`
 - b) `cv2.BORDER_REFLECT_101` or `cv2.BORDER_DEFAULT`
 - c) `cv2.BORDER_REPLICATE`
 - d) `cv2.BORDER_WRAP`
 - e) `cv2.BORDER_CONSTANT`: Adds a border with a constant color. The value of the border color is the following argument.
- `Value`: The color of the border if the border type is `cv2.BORDER_CONSTANT`

Let's look at a few examples of borders around images. Consider the following program:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/4.1.01.tiff', 1)
b1 = cv2.copyMakeBorder(img, 10, 10, 10, 10, cv2.BORDER_WRAP)
b2 = cv2.copyMakeBorder(img, 10, 10, 10, 10, cv2.BORDER_
CONSTANT, value=[255, 0, 0])
cv2.imshow('Wrap', b1)
cv2.imshow('Constant', b2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The output of the preceding code is as follows:

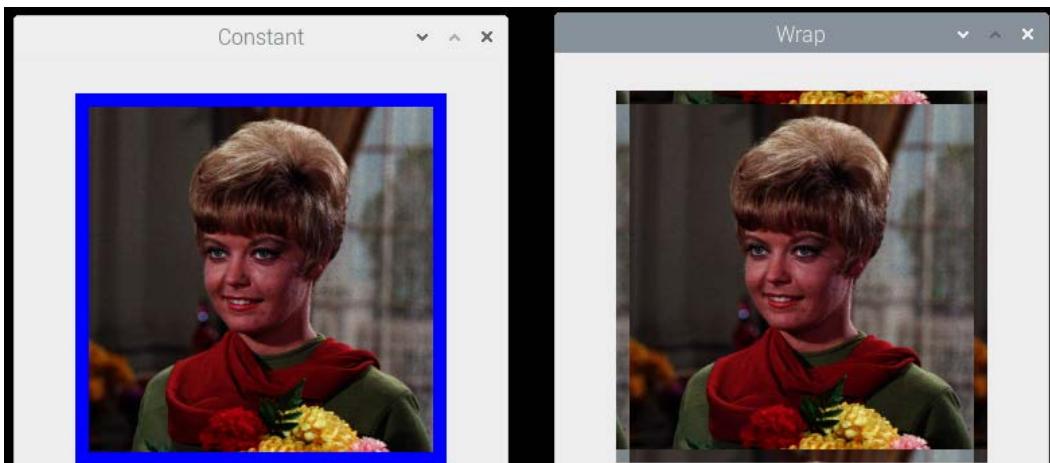


Figure 5.1 – Demonstration of borders

You might want to try some other border options. The following code creates a replicate-style border:

```
cv2.copyMakeBorder(img, 10, 10, 10, 10, cv2.BORDER_REPLICATE)
```

The following code creates a different replicate-style border:

```
cv2.copyMakeBorder(img, 10, 10, 10, 10, cv2.BORDER_REFLECT)
```

```
cv2.copyMakeBorder(img, 10, 10, 10, 10, cv2.BORDER_REFLECT_101)
```

This is how we create various types of borders for images. In the next section, we will look at carrying out arithmetic operations on images.

Arithmetic operations on images

We know that images are nothing but NumPy ndarrays and we can perform arithmetic operations on images just as we can perform them on ndarrays. If we know how to apply numerical or arithmetic operations to matrices, then we should not have any trouble doing the same when the operands for those operations are images. Images must be of the same size and must have the same number of channels for us to perform arithmetic operations on them, and these operations are performed on individual pixels. There are many arithmetic operations, such as addition and subtraction. The first is the addition operation. We can add two images by using either the NumPy `Addition` or the `add()` function in OpenCV, as follows:

```
import cv2
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
cv2.imshow('NumPy Addition', img1 + img2 )
cv2.imshow('OpenCV Addition', cv2.add(img1, img2))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The following is the output of the preceding code:

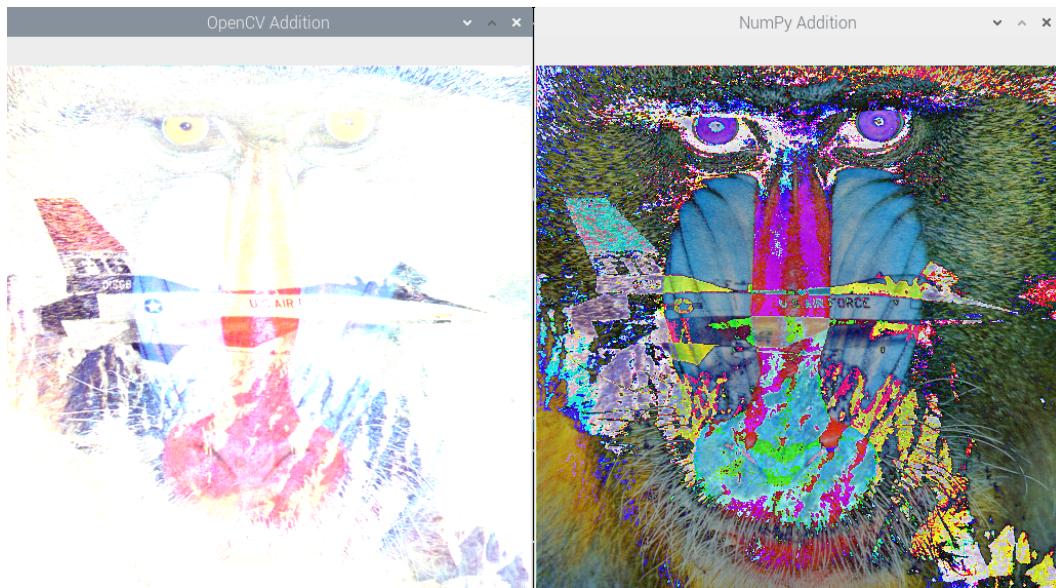


Figure 5.2 – Addition with OpenCV and NumPy

We can clearly see the difference between the two images that appear in the output. The reason for this is that OpenCV's `add()` function is a saturation operation and NumPy's addition operator is a modulo operation. Let's see in detail what that means. Open Python 3 in interactive mode and run the following statements:

```
>>> import numpy as np
>>> import cv2
>>> a = np.array([240], np.uint8)
>>> b = np.array([20], np.uint8)
>>> a + b
array([4], dtype=uint8)
```

We know that the maximum value that `uint8` can store is 255. Any value that exceeds 255 is then divided by 256 and the remainder is stored in the `uint8` data type:

```
>>> cv2.add(a, b)
array([[255]], dtype=uint8)
```

As you can see in the preceding code, in the case of `cv2.add()`, it just sets the value exceeding 255 to 255 for the `uint8` data type.

Similarly, we can compute subtraction with NumPy subtraction and `cv2.subtract()`. The following is an example of this:

```
import cv2
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
cv2.imshow('NumPy Subtract', img1 - img2)
cv2.imshow('OpenCV Subtract', cv2.subtract(img1, img2))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The result of the preceding code is as follows:

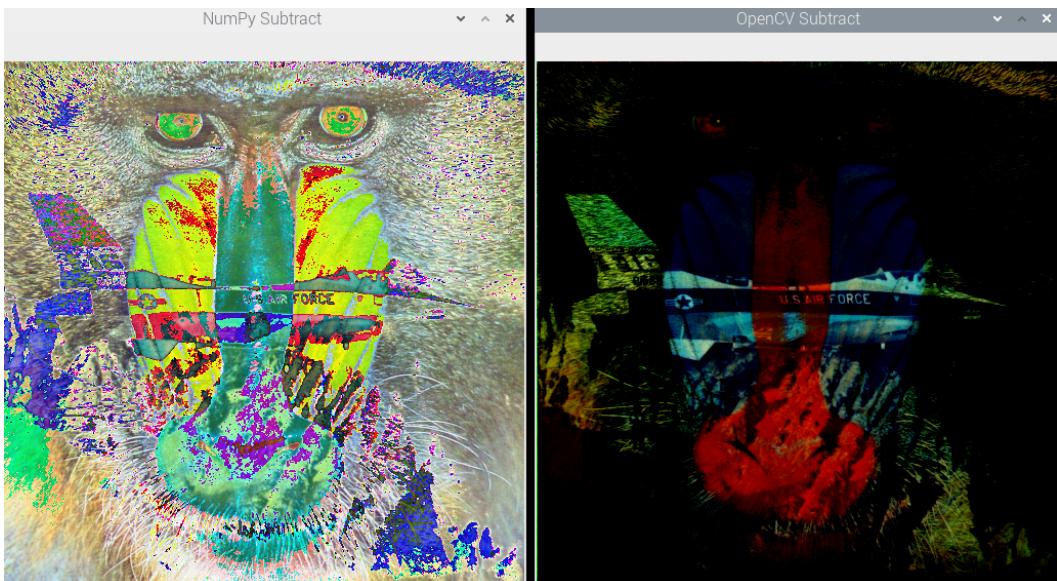


Figure 5.3 – Subtraction with NumPy and OpenCV

Let's try an exercise to understand the difference between the subtraction operation with NumPy and the subtraction operation with OpenCV in interactive mode, as follows:

```
>>> import cv2
>>> import numpy as np
>>> a = np.array([240], np.uint8)
>>> b = np.array([20], np.uint8)
>>> b - a
array([36], dtype=uint8)
```

We know that the lowest number that `uint8` can store is 0. If the number is negative, NumPy adds 256 to it for the `uint8` data type:

```
>>> cv2.subtract(b, a)
array([[0]], dtype=uint8)
```

As this shows, in the case of `cv2.subtract()`, the negative value is just rounded up to 0 for the `uint8` data type.

Note:

We are aware that the subtraction operation is not commutative. This means that $a - b$ is not equal to $b - a$ in most cases. So, if both of the images are of the same size and type, then `cv2.subtract(img1, img2)` and `cv2.subtract(img2, img1)` produce different results. However, the addition operation is commutative. So, `cv2.add(img1, img2)` and `cv2.add(img2, img1)` always produce the same result.

Blending and transitioning images

The `cv2.addWeighted()` function computes the weighted sum of the two images that we pass in as arguments. This causes them to blend. The following is some code that demonstrates this concept of blending:

```
import cv2
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
cv2.imshow('Blended Image',
           cv2.addWeighted(img1, 0.5, img2, 0.5, 0))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In the preceding code, we are passing the following five arguments to the OpenCV `cv2.addWeighted()` function:

- `img1`: The first image
- `alpha`: The coefficient for the first image (0.5 in the preceding example)
- `img2`: The second image

- beta: The coefficient for the second image (0.5 in the preceding example)
- gamma: The scalar value (0 in the preceding example)

OpenCV uses the following formula to compute the output image:

$$\text{output image} = (\text{alpha} * \text{img1}) + (\text{beta} * \text{img2}) + \text{gamma}$$

Every pixel of the output image is computed with this formula, and the following is the output of the preceding code:

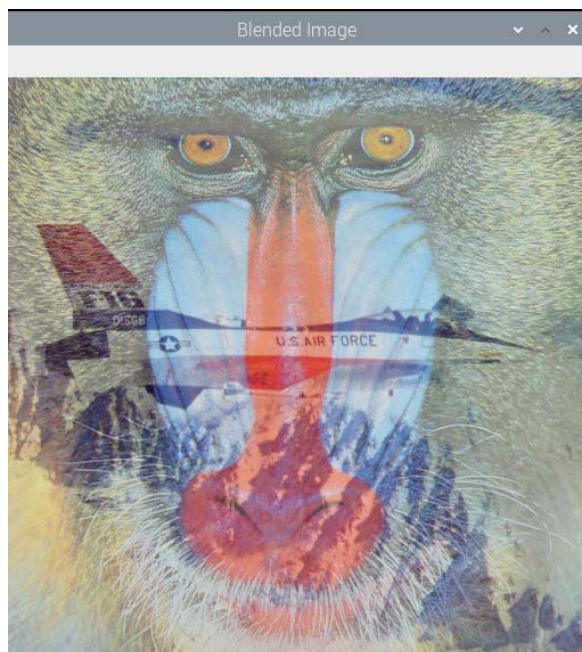


Figure 5.4 – Image blending

We can create a transition effect, which we see in films and video editing software, with the use of the same OpenCV function. The following code example creates a very smooth transition from one image to the other:

```
import cv2
import time
import numpy as np
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
for i in np.linspace(0, 1, 100):
```

```
alpha = i
beta = 1-alpha
print('ALPHA =' + str(alpha) + ' BETA =' + str(beta))
cv2.imshow('Image Transition',
cv2.addWeighted(img1, alpha, img2, beta, 0))
time.sleep(0.05)
if cv2.waitKey(1) == 27 :
    break
cv2.destroyAllWindows()
```

The output of the preceding code creates a transition effect.

We can also create a nice app with a trackbar, as follows:

```
import cv2
import time
import numpy as np
def emptyFunction():
    pass
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
output = cv2.addWeighted(img1, 0.5, img2, 0.5, 0)
windowName = "Transition Demo"
cv2.namedWindow(windowName)
cv2.createTrackbar('Alpha', windowName, 0,
                  1000, emptyFunction)
while(True):
    cv2.imshow(windowName, output)
    if cv2.waitKey(1) == 27:
        break
    alpha = cv2.getTrackbarPos('Alpha', windowName) / 1000
    beta = 1 - alpha
    output = cv2.addWeighted(img1, alpha, img2, beta, 0)
    print(alpha, beta)
cv2.destroyAllWindows()
```

The output of the preceding code creates a nice transitioning app. We can even connect two push buttons to the GPIO of Raspberry Pi in the pull-up configuration, as follows:

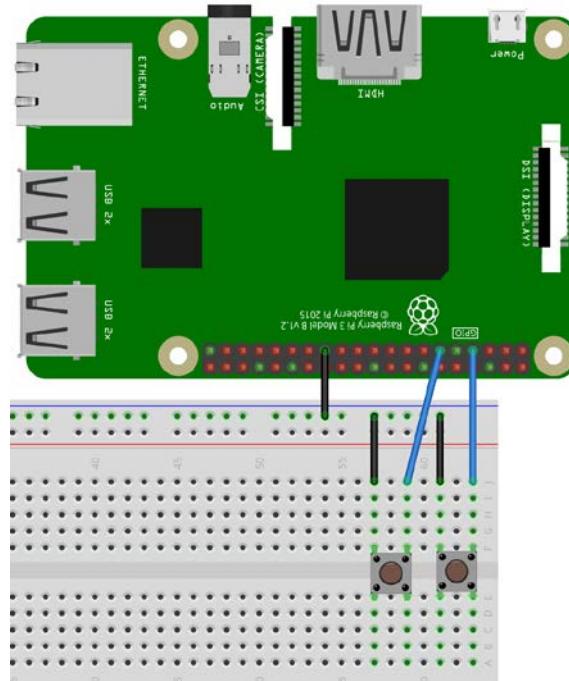


Figure 5.5 – A circuit with push buttons

We can write the following code to integrate the buttons with the image transitioning functionality:

```
import time
import RPi.GPIO as GPIO
import cv2
alpha = 0
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
button1 = 7
button2 = 11
GPIO.setup(button1, GPIO.IN, GPIO.PUD_UP)
GPIO.setup(button2, GPIO.IN, GPIO.PUD_UP)
while True:
```

```
button1_state = GPIO.input(button1)
if button1_state == GPIO.LOW and alpha < 1:
    alpha = alpha + 0.2
button2_state = GPIO.input(button2)
if button2_state == GPIO.LOW:
    if (alpha > 0):
        alpha = alpha - 0.2
    if (alpha < 0):
        alpha = 0
beta = 1 - alpha
output = cv2.addWeighted(img1, alpha, img2, beta, 0)
cv2.imshow('Transition App', output)
if cv2.waitKey(1) == 27:
    break
time.sleep(0.5)
print(alpha)
cv2.destroyAllWindows()
```

The preceding code, on the press of the buttons, changes the value of the alpha variable and the blending proportion of the images. Run the preceding program and press the buttons to see the action. We will use the preceding circuit and program as a template for many programs in this book.

In the next section, we will understand how to multiply images with one another and with a constant.

Multiplying images by a constant and one another

Just like normal matrices or NumPy ndarrays, images can be multiplied by a constant and with one another. We can multiply an image by a constant, as follows:

```
import cv2
img1 = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img2 = cv2.imread('/home/pi/book/dataset/4.2.05.tiff', 1)
cv2.imshow('Image1', img1 * 2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In the preceding code, every element of the ndarray representing the image is multiplied by 2. Run the preceding program and see the output. We can also multiply images with one another, as follows:

```
cv2.imshow('Image1', img1 * 2)
```

The result is likely to look like noise.

Creating a negative of an image

In terms of pure mathematics, when we invert the colors of an image, it creates a negative of the image. This inversion operation can be computed by subtracting the color of a pixel from 255. If it is a color image, we invert the color of all the planes. For a grayscale image, we can directly compute the inversion by subtracting it from 255, as follows:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/4.2.07.tiff', 0)
negative = abs(255 - img)
cv2.imshow('Grayscale', img)
cv2.imshow('Negative', negative)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The following is the output of this:



Figure 5.6 – A negative of an image

Try to find the negative of a color image, we just need to read the image in color mode in the preceding program.

Note:

The negative of a negative will be the original grayscale image. Try this on your own by computing the negative of the negative again for our color and grayscale images.

Bitwise logical operations on images

The OpenCV library has many functions for computing bitwise logical operations on images. We can compute bitwise logical AND, OR, XOR (exclusive OR), and NOT (inversion) operations. The best way to demonstrate how these functions work is to use them with binary (black and white) images:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
a = [0, 255, 0]
img1 = np.array([a, a, a], dtype=np.uint8)
img2 = np.transpose(img1)
not_out = cv2.bitwise_not(img1 )
and_out = cv2.bitwise_and(img1, img2)
or_out = cv2.bitwise_or(img1, img2)
xor_out = cv2.bitwise_xor(img1, img2)
titles = ['Image 1', 'Image 2', 'Image 1 NOT', 'AND', 'OR',
'XOR']
images = [img1, img2, not_out, and_out, or_out, xor_out]
for i in range(6):
    plt.subplot(2, 3, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

We created our own custom binary image to better demonstrate the functionality of the bitwise logical NOT, AND, OR, and XOR operations, respectively. We will use the `matplotlib` library `plt.subplot()` function to visualize multiple images at the same time.

In the preceding example, we created a grid of two rows and three columns to show the original input images and the computed outputs of the bitwise logical operations with OpenCV functions. Each image is displayed in one part of the grid. The first position is the top left, the second position is adjacent to that, and so on. We can change the line and make it `plt.subplot(3, 2, i+1)` to create a grid of three rows and two columns. We will use this technique later on in this book, too. We will use it to display images side by side in a single output window.

We can also use the `plt.subplot()` function without a loop. For each image, we must write the following set of statements. I am writing the code block for one image. Write the same for the other images:

```
plt.subplot(2, 3, 1)
plt.imshow(img1, cmap='gray')
plt.title('Image 1')
plt.axis('off')
```

Finally, we use the call of the `plt.show()` function to display everything on the screen. We use this technique to display two or three images. If we have more images than that, then we can use the loop technique to display multiple images in the same output window. The following is our output:

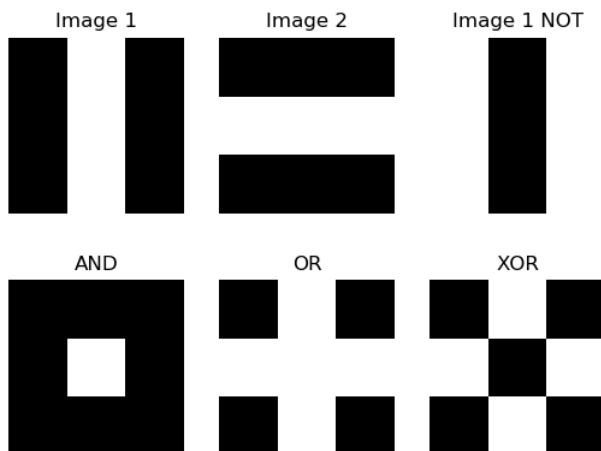


Figure 5.7 – Logical operations on images

You might want to implement the code for bitwise logical operations on the grayscale and color images.

Note:

We can also achieve the same result by using NumPy's logical operations.

Summary

In this chapter, we started by looking at image processing with OpenCV and NumPy. We learned about some important concepts, such as image channels, arithmetic and logical operations, and the negative of an image. Along the way, we also learned to use a bit more functionality in Python 3 and the NumPy library. The bitwise logical operations that we learned today will be very useful when writing programs for the functionality of object tracking by color in the next chapter.

In the next chapter, we will study colorspace transformations, and thresholding images.

6

Colorspaces, Transformations, and Thresholding

In the previous chapter, we learned how to perform basic mathematical and logical operations on images. In this chapter, we will continue to explore some more intriguing concepts in the area of computer vision and its applications in the real world. Just like in the earlier chapters of this book, we will have a lot of hands-on exercises with Python 3 and create many real-world apps. We will cover a very wide variety of advanced topics in the area of computer vision. The major topics we will learn about are related to colorspace, transformations, and thresholding images. After completing this chapter, you will be able to write programs for a few basic real-world applications, such as tracking an object that's a specific color. You will also be able to apply geometric and perspective transformations to images and live USB webcam feeds.

In this chapter, we will explore the following topics:

- Colorspace and converting them
- Performing transformation operations on images
- Perspective transformation of images
- Thresholding images

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter06/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/384oYqM>.

Colorspaces and converting them

Let's understand the concept of a colorspace. A **colorspace** is a mathematical model that is used to represent a set of colors. With colorspaces, we can represent colors with numbers. If you've ever have worked with web programming, then you must have come across various codes for colors since colors are represented in HTML with *Hexadecimal numbers*. This is a good example of representing colors with a colorspace and allows us to perform numerical and logical computations with them. Representing colors with colorspaces also allows us to reproduce the colors with ease in analog and digital forms.

We will frequently use **BGR**, **RGB**, **HSV**, and **grayscale** colorspace throughout this book. In BGR and RGB, B stands for blue, G stands for green, and R stands for red. OpenCV reads and stores a color image in the BGR colorspace. The HSV colorspace represents a set of colors with a component for hue, a component for saturation, and a component for value. It is a very commonly used colorspace in the areas of computer graphics and computer vision. OpenCV has a function, `cvtColor(img, conv_flag)`, that changes the colorspace of the image that's passed to it as an argument. The source and target colorspace are denoted by the argument that's passed to the `conv_flag` parameter. This function converts the numerical value of a color from the source colorspace into the target colorspace with the use of mathematical formulae used for colorspace conversion.

Note:

You can read more about colorspace and conversion at the following URL:
<http://colorizer.org>.

As you may recall, earlier, in *Chapter 4, Getting Started with Computer Vision*, we discussed that OpenCV loads images in BGR format and that Matplotlib uses the RGB format for images. So, when we display images read by OpenCV in BGR format with matplotlib in RGB format, the red and blue channels are interchanged in the visualization and the image looks funny. We should convert an image from BGR into RGB before displaying the image with matplotlib. There are two ways to do this.

Let's look at the first way. We can split the image into B, G, and R channels and merge them into an RGB image with the `split()` and `merge()` functions, as follows:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.2.07.tiff', 1)
b,g,r = cv2.split (img)
img = cv2.merge((r, g, b))
plt.imshow (img)
plt.title ('COLOR IMAGE')
plt.axis('off')
plt.show()
```

However, the `split` and `merge` operations are computationally expensive. A better approach is to use the `cv2.cvtColor()` function to change the colorspace of an image from BGR to RGB, as demonstrated in the following code:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.2.07.tiff', 1)
img = cv2.cvtColor (img, cv2.COLOR_BGR2RGB)
plt.imshow (img)
plt.title ('COLOR IMAGE')
plt.axis('off')
plt.show()
```

In the preceding code, we used the `cv2.COLOR_BGR2RGB` flag for color conversion. OpenCV has plenty of such flags for color conversion. We can run the following program to see the entire list:

```
import cv2
j=0
for filename in dir(cv2):
    if filename.startswith('COLOR_'):
        print(filename)
        j = j + 1
print('There are ' + str(j) +
      ' Colorspace Conversion flags in OpenCV ' +
      + cv2.__version__ + '.')
```

The last few lines of the output are shown in the following code block (I am not including the entire output due to space limitations):

```
 .
 .
 .
 .
 .
 COLOR_YUV420p2RGBA
 COLOR_YUV420sp2BGR
 COLOR_YUV420sp2BGRA
 COLOR_YUV420sp2GRAY
 COLOR_YUV420sp2RGB
 COLOR_YUV420sp2RGBA
 COLOR_mRGBA2RGBA
```

There are 274 colorspace conversion flags in OpenCV 4.0.1.

HSV colorspace

The term **HSV** stands for **hue, saturation, and value**. In this colorspace or color model, a color is represented by the hue (also known as the tint), the shade (which is the saturation scale or the amount of gray with white and black on extreme ends), and the brightness (the value or the luminescence). The intensities of the colors red, yellow, green, cyan, blue, and magenta are represented by the hue. The term saturation means the amount of gray component present in the color. The brightness or the intensity of the color is represented by the value component.

The following code converts a color from BGR into HSV and prints it:

```
import cv2
import numpy as np
c = cv2.cvtColor(np.array([[ [255, 0, 0] ]],
                         dtype=np.uint8),
                 cv2.COLOR_BGR2HSV)
print(c)
```

The preceding code snippet will print the HSV value of blue represented in BGR. The following is the output:

```
[[[120 255 255]]]
```

We will heavily use the HSV colorspace throughout this book. Before proceeding further, let's create a small app with a trackbar that adjusts the saturation of the color when the tracker moves:

```
import cv2
def emptyFunction():
    pass
img = cv2.imread('/home/pi/book/dataset/4.2.07.tiff', 1)
windowName = "Saturation Demo"
cv2.namedWindow(windowName)
cv2.createTrackbar('Saturation Level',
                  windowName, 0,
                  24, emptyFunction)
while(True):
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    saturation = cv2.getTrackbarPos('Saturation Level',
                                    windowName)
    s = s + saturation
    v = v + saturation
    img1 = cv2.cvtColor(cv2.merge((h, s, v)), cv2.COLOR_HSV2BGR)
    cv2.imshow(windowName, img1)
    if cv2.waitKey(1) == 27:
        break
cv2.destroyAllWindows()
```

In the preceding code, we first converted the image from BGR into HSV and split it into H, S, and V components. Then, we added a number to saturation (s), as well as a value (v), based on the position of the tracker in the trackbar. Then, we merged all the channels to create an HSV image and then converted it back into BGR to be displayed with the `cv2.imshow()` function. The following is a screenshot of the output window:

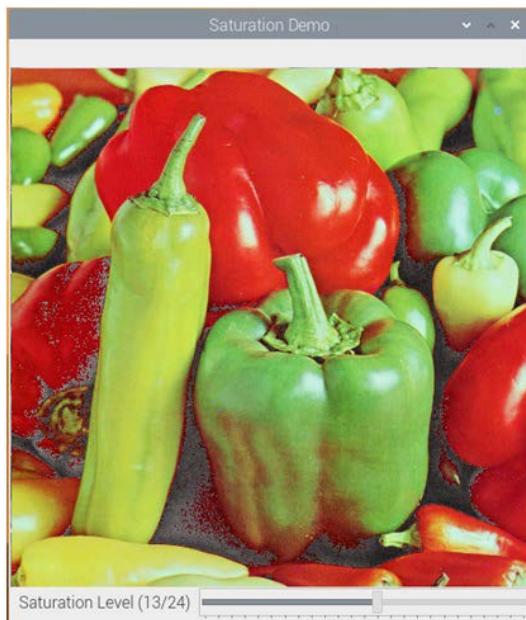


Figure 6.1 – App for adjusting the saturation of an image

Tracking in real time based on color

Now, let's learn how to demonstrate the concept of converting colorspaces to implement a real-life mini project. The HSV colorspace makes it easy for us to work with a range of a color. To track an object that can have colors in a specific range, we need to convert the image's colorspace into HSV and check if any part of the image falls within the specific range of the color we are interested in. OpenCV has a function, `cv2.inRange()`, that offers the functionality to define a color range.

This function accepts an image and the upper bound and the lower bound of the range of the color as arguments. Then, it checks if any pixel of the given image falls within the range of color (the upper bound and the lower bound). If the pixel value in the image lies in the given range of color, the corresponding pixel in the output image is set to the value 0; otherwise, it is set to the value 255. This creates a binary image that can be used as a mask for computing the logical operations that we will use for tracking the application.

The following example demonstrates this concept. We are using the logical `bitwise_and()` function to extract the range of the color we are interested in:

```
import numpy as np
import cv2
cap = cv2.VideoCapture(0)
while ( True ):
    ret, frame = cap.read()
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    image_mask = cv2.inRange(hsv, np.array([40, 50, 50]),
    np.array([80, 255, 255]))
    output = cv2.bitwise_and(frame, frame, mask=image_mask)
    cv2.imshow('Original', frame)
    cv2.imshow('Output', output)
    if cv2.waitKey(1) == 27:
        break
cv2.destroyAllWindows()
cap.release()
```

In this program, we are tracking the green-colored objects. The output should look like what's shown in the following screenshot. Here, I used the lid (cover) of a container, which is a greenish color:

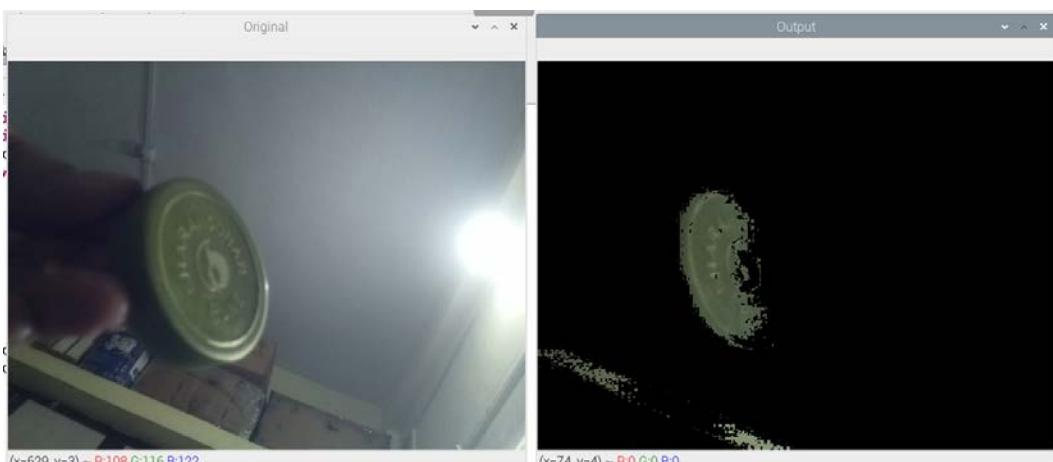


Figure 6.2 – Tracking an object by color in real time

The parts of the wall also have a greenish tint. Due to this, they are also visible in the output.

I have not included the intermediate mask image we computed in the preceding output. We can view it in a separate output window by adding the following line of code to the code we wrote earlier:

```
cv2.imshow('Image Mask', image_mask)
```

This mask is purely black and white, also known as a **binary image**. If we make modifications to the preceding code, we can track objects that have distinct colors. We must create another mask for the range of colors we are interested in. Then, we can combine both masks, as follows:

```
blue = cv2.inRange(hsv, np.array([100, 50, 50]), np.array([140, 255, 255]))
green = cv2.inRange(hsv, np.array([40, 50, 50]), np.array([80, 255, 255]))
image_mask = cv2.add(blue, green)
output = cv2.bitwise_and(frame, frame, mask=image_mask)
```

Run this code and check the output for yourself. We can add a trackbar to this code to select a range of blue or green colors. The following are the steps to do this:

1. First, import all the required libraries:

```
import numpy as np
import cv2
```

2. Then, we define an empty function:

```
def emptyFunction():
    pass
```

3. Let's initialize all the required objects and variables:

```
cap = cv2.VideoCapture(0)
windowName = 'Object Tracker'
trackbarName = 'Color Chooser'
cv2.namedWindow(windowName)
cv2.createTrackbar(trackbarName,
                  windowName, 0, 1,
                  emptyFunction)
color = 0
```

4. Here, we have the main loop:

```
while (True):
    ret, frame = cap.read()
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    color = cv2.getTrackbarPos(trackbarName, windowName)
    if color == 0:
        image_mask = cv2.inRange(hsv, np.array([40, 50,
50]), np.array([80, 255,
255]))
    else:
        image_mask = cv2.inRange(hsv, np.array([100, 50,
50]), np.array([140, 255,
255]))
    output = cv2.bitwise_and(frame, frame, mask=image_
mask)
    cv2.imshow(windowName, output)
    if cv2.waitKey(1) == 27:
        break
```

5. Finally, we destroy all the windows and release the camera:

```
cv2.destroyAllWindows()
cap.release()
```

Run the preceding code and see the output for yourself. By now we are aware of the GPIO interface and the push buttons. As an exercise, try to implement the same functionality with the push buttons so that there will be separate push buttons for tracking blue and green colors.

Performing transformation operations on images

In this section, we will learn how to perform various mathematical transformation operations on images with OpenCV and Python 3.

Scaling

Scaling means resizing an image. It is a geometric operation. OpenCV offers a function, `cv2.resize()`, for performing this operation. It accepts an image, a method for the interpolation of pixels, and the scaling factor as arguments and returns a scaled image. The following methods are used for the interpolation of the pixels in the output:

- `cv2.INTER_LANCZOS4`: This deals with the Lanczos interpolation method over a neighborhood of 8x8 pixels.
- `cv2.INTER_CUBIC`: This deals with the bicubic interpolation method over a neighborhood of 4x4 pixels and is preferred for performing the zooming operation on an image.
- `cv2.INTER_AREA`: This means resampling using pixel area relation. This is preferred for performing the shrinking operation on an image.
- `cv2.INTER_NEAREST`: This means the method of nearest-neighbor interpolation.
- `cv2.INTER_LINEAR`: This means the method of bilinear interpolation. This is the default argument for the parameter.

The following example demonstrates performing upscaling and downscaling on an image:

```
import cv2
img = cv2.imread('/home/pi/book/dataset/house.tiff', 1)
upscale = cv2.resize(img, None, fx=1.5, fy=1.5,
                     interpolation=cv2.INTER_CUBIC)
downscale = cv2.resize(img, None, fx=0.5, fy=0.5,
                      interpolation=cv2.INTER_AREA)
cv2.imshow('upscale', upscale)
cv2.imshow('downscale', downscale)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In the preceding code, we first upscale in both axes and then downscale in both axes by factors of 1.5 and 0.5, respectively. Run the preceding code to see the output. Also, as an exercise, try to pass different numbers as scaling factors.

The translation, rotation, and affine transformation of images

The `cv2.warpAffine()` function is used to compute operations such as translation, rotation, and affine transformations on input images. It accepts an input image, the matrix of the transformation, and the size of the output image as arguments, and then it returns the transformed image.

Note:

You can find out more about the mathematical aspects of affine transformations at <http://mathworld.wolfram.com/AffineTransform.html>.

The following examples demonstrate different types of mathematical transformations that can be applied to images with the `cv2.warpAffine()` function. The translation operation means changing (more precisely, shifting) the location of the image in the XY reference plane. The shifting factor in the x and y axes can be represented with a two-dimensional transformation matrix, T, as follows:

$$\begin{matrix} 1 & 0 & x \\ 0 & 1 & y \end{matrix}$$

The following code shifts the location of the image in the XY plane by a factor of (-50, 50):

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/house.tiff', 1)
input=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channel = img.shape
T = np.float32([[1, 0, -50], [0, 1, 50]])
output = cv2.warpAffine(input, T, (cols, rows))
plt.imshow(output)
plt.title('Shifted Image')
plt.show()
```

The output of the preceding code is as follows:

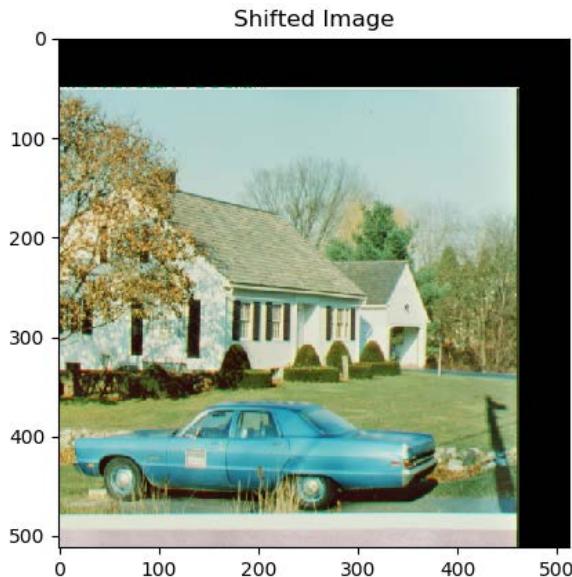


Figure 6.3 – Output of the translation operation

As shown in the preceding output, a part of the image in the output is cropped (or truncated), since the size of the output window is the same as the input window, and the original image has shifted beyond the first quadrant of the XY plane. Similarly, we can use the `cv2.warpAffine()` function to apply the operation of rotation with scaling to an input image. For this demonstration, we must define a matrix of the rotation using the `cv2.getRotationMatrix2D()` function.

This accepts the angle of anti-clockwise rotation in degrees, the center of the rotation, and the scaling factor as arguments. Then, it creates a matrix of the rotation operation that can be passed as an argument to the call of the `cv2.warpAffine()` function. The following example applies the rotation operation to an input image with 45 degrees as the angle of the rotation and the center of the image as the center of the rotation operation, and it also scales the output image down to half (50%) the size of the original input image:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/house.tiff', 1)
input = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channel = img.shape
R = cv2.getRotationMatrix2D((cols/2, rows/2), 45, 0.5)
```

```
output = cv2.warpAffine(input, R, (cols, rows))
plt.imshow(output)
plt.title('Rotated and Downscaled Image')
plt.show()
```

The output will be as follows:

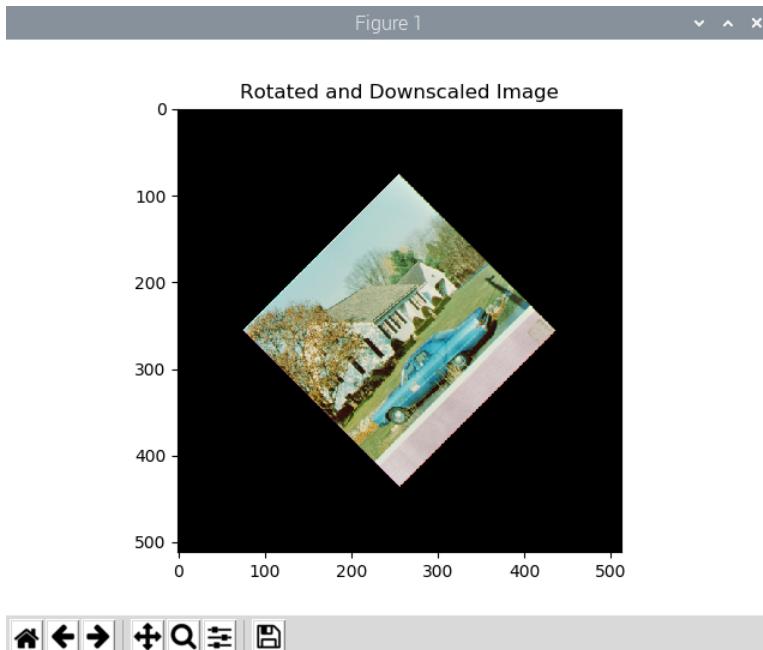


Figure 6.4 – Output of the rotation operation

We can also create a very nice animation by modifying the preceding program. The trick here is that, in the while loop, we must change the angle of rotation at a regular interval and show those frames successively to create the rotation effect on a still image. The following code example demonstrates this:

```
import cv2
from time import sleep
image = cv2.imread('/home/pi/book/dataset/house.tiff',1)
rows, cols, channels = image.shape
angle = 0
while(1):
    if angle == 360:
```

```
angle = 0
M = cv2.getRotationMatrix2D((cols/2, rows/2), angle, 1)
rotated = cv2.warpAffine(image, M, (cols, rows))
cv2.imshow('Rotating Image', rotated)
angle = angle +1
sleep(0.2)
if cv2.waitKey(1) == 27 :
    break
cv2.destroyAllWindows()
```

Run the preceding code and check the output for yourself. Now, let's try to implement this trick on a live webcam. Use the following code to do so:

```
import cv2
from time import sleep
cap = cv2.VideoCapture(0)
ret, frame = cap.read()
rows, cols, channels = frame.shape
angle = 0
while(1):
    ret, frame = cap.read()
    if angle == 360:
        angle = 0
    M = cv2.getRotationMatrix2D((cols/2, rows/2), angle, 1)
    rotated = cv2.warpAffine(frame, M, (cols, rows))
    cv2.imshow('Rotating Image', rotated)
    angle = angle +1
    sleep(0.2)
    if cv2.waitKey(1) == 27 :
        break
cv2.destroyAllWindows()
```

Run the preceding code and see it in action.

Now, let's learn about the concept of the affine mathematical transformation and demonstrate the same with OpenCV and Python 3. An affine transformation is a geometric mathematical transformation that ensures that the parallel lines in the original input image remain parallel in the output image. The usual inputs to the affine transformation operation are a set of three points that are not in the same line in the input image and the corresponding set of three points that are not in the same line in the output image. These sets of points are passed to the `cv2.getAffineTransform()` function to compute the transformation matrix, and that computed transformation matrix, in turn, is passed to the call of the `cv2.warpAffine()` function as an argument. The following example demonstrates this concept very well:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('/home/pi/book/dataset/4.2.06.tiff', 1)
input = cv2.cvtColor(image, cv2.COLOR_BGR2RGB )
rows, cols, channels = input.shape
points1 = np.float32([[100, 100], [300, 100], [100, 300]])
points2 = np.float32([[200, 150], [400, 150], [100, 300]])
A = cv2.getAffineTransform(points1, points2)
output = cv2.warpAffine(input, A, (cols, rows))
plt.subplot(121)
plt.imshow(input)
plt.title('Input')
plt.subplot(122)
plt.imshow(output)
plt.title('Affine Output')
plt.show()
```

The following is the output:

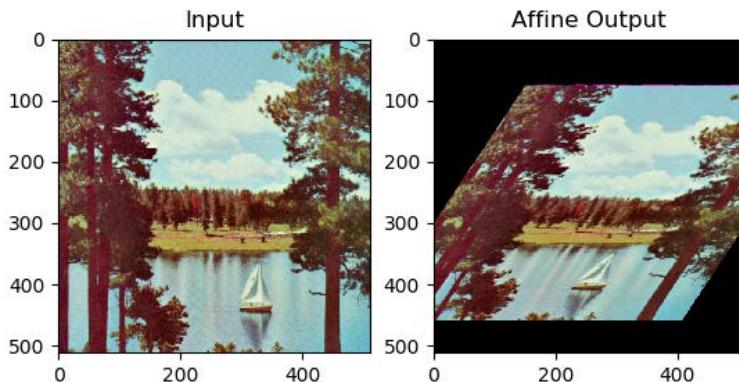


Figure 6.5 – Affine transformation

As we can see, the preceding code creates a shear-like effect on the input image.

Perspective transformation of images

In the mathematical operation of perspective transformation, a set of four points in the input image is mapped to a set of four points in the output image. The criteria for selecting the set of four points in the input and the output image is that any three points (in the input and the output image) must not be in the same line. Like affine mathematical transformation, in perspective transformation, the straight lines in the input images remain straight. However, there is no guarantee that the parallel lines in the input image remain parallel in the output image.

One of the most prominent real-life examples of this mathematical operation is the zoom and the angled zoom functions in image editing and viewing software tools. The amount of zoom and angle of zooming depend on the matrix of the transformation that is computed by the two sets of points that we discussed earlier. OpenCV provides the `cv2.getPerspectiveTransform()` function, which accepts two sets of four points from the input image and the output image and computes the matrix of the transformation. The `cv2.warpPerspective()` function accepts the computed matrix as an argument and applies it to the input image to compute the perspective transform of the input image. The following code demonstrates this aptly:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```

image = cv2.imread('/home/pi/book/dataset/ruler.512.tiff', 1)
input = cv2.cvtColor(image, cv2.COLOR_BGR2RGB )
rows, cols, channels = input.shape
points1 = np.float32([[0, 0], [400, 0], [0, 400], [400, 400]])
points2 = np.float32([[0,0], [300, 0], [0, 300], [300, 300]])
P = cv2.getPerspectiveTransform(points1, points2)
output = cv2.warpPerspective(input, P, (300, 300))
plt.subplot(121)
plt.imshow(input)
plt.title('Input Image')
plt.subplot(122)
plt.imshow(output)
plt.title('Perspective Transform')
plt.show()

```

The output will appear as follows:

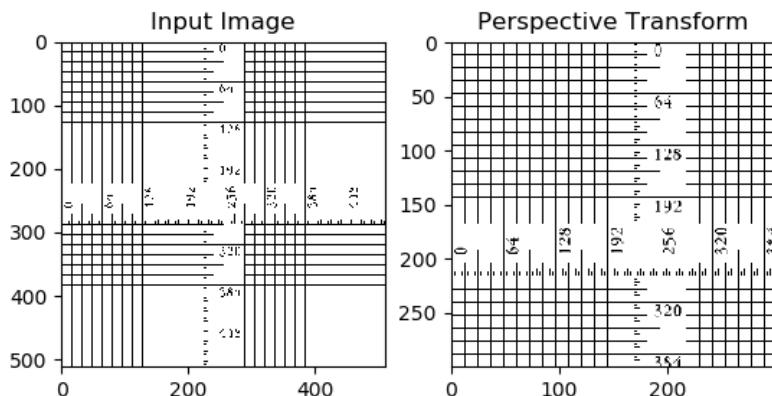


Figure 6.6 – Zoom operation with perspective transform

As an exercise for this section (and to improve your understanding of the operation of perspective transformation), pass various combinations of sets of points in the input and the output images to the program to see how the output changes when the input is changed. From the example we just discussed, you may get the impression that the parallelism between the lines in the input and the output image is preserved, but that is because of our choice of sets for the points in the input image and the output image. If we choose different sets of points, then the output will obviously be different.

These are all the transformation operations we can perform on images with OpenCV. Next, we will see how to threshold images with OpenCV.

Thresholding images

Thresholding is the simplest way to divide images into various parts, which are known as **segments**. Thresholding is the simplest form of segmentation operation. If we apply the thresholding operation to a grayscale image, it is usually (but not all the time) transformed into a binary image. A binary image is a strictly black and white image and it can either have a 0 (black) or 255 (white) value for a pixel. Many segmentation algorithms, advanced image processing operations, and computer vision applications use thresholding as the first step for processing images.

Thresholding is perhaps the simplest image processing operation. First, we must define a value for the threshold. If a pixel has a value greater than the threshold, then we assign 255 (white) to that pixel; otherwise, we assign 0 (black) to the pixel. This is the simplest way we can implement the thresholding operation on an image. There are other thresholding techniques too, and we will learn about and demonstrate them in this section.

The OpenCV `cv2.threshold()` function applies thresholding to images. It accepts the image, the value of the threshold, the maximum value, and the technique of thresholding as arguments and returns the thresholded image as the output. This function assigns the value of the maximum value to a pixel if its value is greater than the value of the threshold. As we mentioned earlier, there are variations of this method. Let's take a look at all the thresholding techniques in detail.

Let's assume that (x, y) is the input pixel. Here, we can threshold an image in the following ways:

- `cv2.THRESH_BINARY`: If $\text{intensity}(x, y) > \text{thresh}$, then set $\text{intensity}(x, y) = \text{maxVal}$; otherwise, set $\text{intensity}(x, y) = 0$.
- `cv2.THRESH_BINARY_INV`: If $\text{intensity}(x, y) > \text{thresh}$, then set $\text{intensity}(x, y) = 0$; otherwise, set $\text{intensity}(x, y) = \text{maxVal}$.
- `cv2.THRESH_TRUNC`: If $\text{intensity}(x, y) > \text{thresh}$, then set $\text{intensity}(x, y) = \text{threshold}$; else leave $\text{intensity}(x, y)$ as it is.
- `cv2.THRESH_TOZERO`: If $\text{intensity}(x, y) > \text{thresh}$; then leave $\text{intensity}(x, y)$ as it is; otherwise, set $\text{intensity}(x, y) = 0$.
- `cv2.THRESH_TOZERO_INV`: If $\text{intensity}(x, y) > \text{thresh}$, then set $\text{intensity}(x, y) = 0$; otherwise, leave $\text{intensity}(x, y)$ as it is.

Grayscale images with gradients are excellent input for thresholding algorithms as we can visually see the thresholding in action. In the following example, we are using a grayscale gradient image as an input to demonstrate the thresholding operation. We have set the value of the threshold to 127, so the image is segmented into two or more parts, depending on the value of the intensity of pixels and thresholding technique that we are using:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread('/home/pi/book/dataset/gray21.512.tiff', 1)
th = 127
max_val = 255
ret, o1 = cv2.threshold(img, th, max_val,
                       cv2.THRESH_BINARY)
print(o1)
ret, o2 = cv2.threshold(img, th, max_val,
                       cv2.THRESH_BINARY_INV)
ret, o3 = cv2.threshold(img, th, max_val,
                       cv2.THRESH_TOZERO)
ret, o4 = cv2.threshold(img, th, max_val,
                       cv2.THRESH_TOZERO_INV)
ret, o5 = cv2.threshold(img, th, max_val,
                       cv2.THRESH_TRUNC)
titles = ['Input Image', 'BINARY', 'BINARY_INV',
          'TOZERO', 'TOZERO_INV', 'TRUNC']
output = [img, o1, o2, o3, o4, o5]
for i in range(6):
    plt.subplot(2, 3, i+1)
    plt.imshow(output[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

The following is the output:

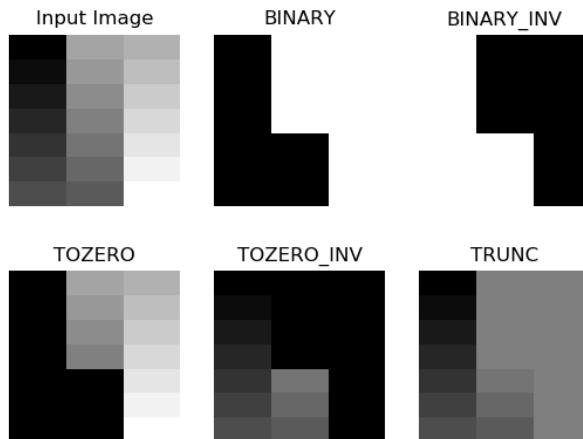


Figure 6.7 – Output of the thresholding operation

You might want to create an application with a trackbar. We can also interface two push buttons in the pull-up configuration and write some code to adjust the threshold on a live video with the help of those two push buttons:

```
import RPi.GPIO as GPIO
import cv2
thresh = 127
cap = cv2.VideoCapture(0)
GPIO.setmode(GPIO.BRD)
GPIO.setwarnings(False)
button1 = 7
button2 = 11
GPIO.setup(button1, GPIO.IN, GPIO.PUD_UP)
GPIO.setup(button2, GPIO.IN, GPIO.PUD_UP)

while True:
    ret, frame = cap.read()
    button1_state = GPIO.input(button1)
    if button1_state == GPIO.LOW and thresh < 256:
```

```
thresh = thresh + 1
button2_state = GPIO.input(button2)
if button2_state == GPIO.LOW and thresh > -1:
    thresh = thresh - 1
ret1, output = cv2.threshold(frame, thresh, 255,
                             cv2.THRESH_BINARY)
print(thresh)
cv2.imshow('Thresholding App', output)
if cv2.waitKey(1) == 27:
    break
cv2.destroyAllWindows()
```

Prepare a circuit by connecting two push buttons to pins 7 and 11. Connect a webcam to a USB or Pi Camera Module to the CSI port. Then, run the preceding code. The following will be the output:

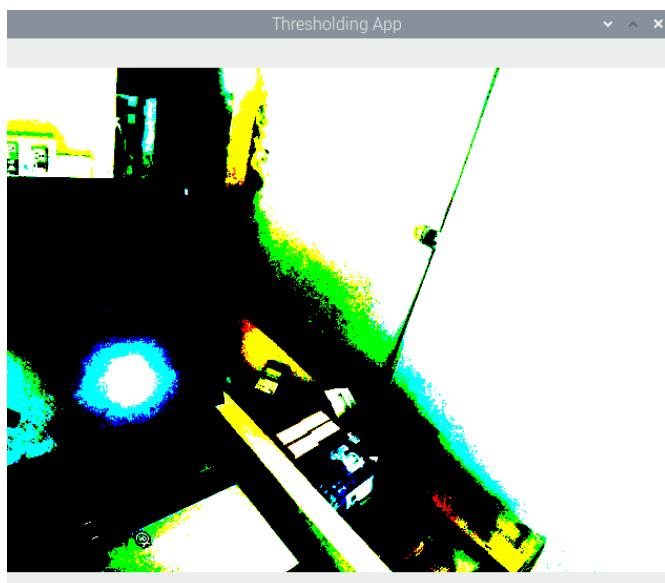


Figure 6.8 – Thresholding a live USB webcam feed

The output looks like this because we are applying thresholding to the live feed and the color image. OpenCV applies thresholding to all the channels. As an exercise, convert the input frame into grayscale and then apply different types of thresholds to it.

Otsu's binarization method

In our previous examples of thresholding, we chose the value of the thresholding argument. However, the value of the threshold for the input image is a technique that's automatically determined by Otsu's binarization method. However, this method does not work for all images. The prerequisite is that the input image must have two peaks in the histogram. Such images are known as **bimodal histogram images**. We will learn more about this concept and demonstrate how to use histograms and histograms of images later in this book. A bimodal histogram usually means that the image has a background and a foreground. Otsu's binarization works best with such images.

This method is not recommended for images other than those that have bimodal histograms as it will produce improper results. This method is always combined with other thresholding methods. While calling the `cv2.threshold()` function, we have to pass 0 as an argument to the `threshold` parameter, as shown in the following code snippet:

```
ret, output = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY +  
cv2.THRESH_OTSU )
```

Run the preceding code and see the output.

Adaptive thresholding

In the earlier examples (including Otsu's binarization), the threshold is the same for all the pixels in the entire image. That is why those techniques are known as global thresholding techniques. However, they do not produce good results for all types of images. For images where lighting is not uniform, global thresholding methods are not the best. We can use algorithms that compute the threshold values locally, depending on the value of the nearby pixel. Such techniques are known as local or adaptive thresholding techniques.

The `cv2.adaptiveThreshold()` method accepts a source image, maximum value, adaptive thresholding method, thresholding algorithm, block size, and a constant as inputs and produces a thresholded image as output. The following shows how to use the mean and Gaussian methods for deciding on the neighborhood in order to determine a threshold value:

```
import cv2  
import matplotlib.pyplot as plt  
img = cv2.imread('/home/pi/book/dataset/4.1.05.tiff', 0)  
block_size = 123  
constant = 6
```

```
th1 = cv2.adaptiveThreshold(img, 255,
                            cv2.ADAPTIVE_THRESH_MEAN_C,
                            cv2.THRESH_BINARY,
                            block_size, constant)
th2 = cv2.adaptiveThreshold (img, 255,
                            cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                            cv2.THRESH_BINARY,
                            block_size, constant)
output = [img, th1, th2]
titles = ['Original', 'Mean Adaptive', 'Gaussian Adaptive']
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(output[i], cmap='gray')
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

The following is the output of the preceding code:

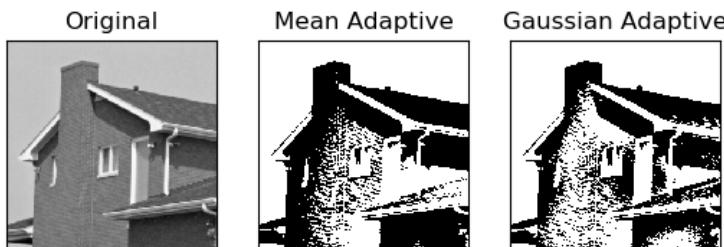


Figure 6.9 – Mean and Gaussian adaptive thresholding methods

As we can see in the preceding output image, the outputs produced by the mean and Gaussian adaptive threshold are different. We must choose the proper thresholding algorithm based on the input image to get the desired results. Often, a trial and error method is the best for choosing the thresholding algorithms and the value of the threshold.

Summary

This was an interesting chapter. We started by looking at colors and their application for object tracking by color. Then, we learned about transformations and thresholding. We also learned how to create a small app with push buttons for live thresholding. All the concepts we demonstrated, especially thresholding techniques, will be very useful for the advanced image processing applications we will learn about later in this book.

In the next chapter, we will learn about a few signal processing concepts and image noise. We will learn about techniques for filtering images and removing noise in images. We will also combine those concepts with the RPi's GPIO and create a few nice live image processing apps.

7

Let's Make Some Noise

In the previous chapter, we learned and demonstrated the concepts of colorspace and converting them, mathematical transformations, and thresholding operations.

In this chapter, we will learn and demonstrate the concepts related to noise and filtering. This entire chapter is dedicated to understanding the concept of noise in detail. First, we will learn how to simulate various types of noise pattern in depth. Then, we will learn and demonstrate how to use image kernels and the convolution operation. We will also learn how to use the convolution operation to apply various types of filters. Finally, we will learn the basics of low pass filters and demonstrate how to use them to perform blurring and noise removal operations.

We will also use GPIO for demonstrations. In this chapter, we will cover the following topics:

- Noise
- Working with kernels
- 2D convolution with the Signal Processing module in SciPy
- Filtering and blurring with OpenCV

After completing this chapter, you will be able to work with noisy images and reduce the amount of noise in them.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter07/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/3i7iagG>.

Noise

Let's understand the concept of noise in detail. In the field of signal processing, noise is simply just any unwanted signal mixed in with the expected signal. When we talk in terms of noise in images or videos, we can define noise as the undesired variation of intensity and color of pixels. This noise can come from multiple sources.

A few examples include dust on a camera lens, grains in the photo film (this one is desired in analog photography and filmmaking), errors in the CCD sensor and its storage, errors during transmission and reception, and errors while scanning the photograph. A very high amount of noise is not desired. This is because high noise reduces the useful and expected signal, affecting the quality of images.

We can mathematically represent the signal-to-noise ratio with the following formula:

$$\text{Signal to Noise Ratio} = (\text{Power of Signal}) / (\text{Power of Noise})$$

Note:

A higher signal-to-noise ratio means a better quality regarding the signal and the image.

Introducing noise to an image

As discussed in the previous section, there can be multiple sources where noise can originate. We can also introduce noise to a digital image by simulating various types of noise. In this section, we will learn how to simulate salt-and-pepper noise, Gaussian noise, Poisson noise, and random normal noise.

Salt-and-pepper noise

The random introduction of white (salt) and black (pepper) pixels to any image is known as **salt-and-pepper noise**. We can introduce it to any grayscale image like so:

```
import numpy as np
import cv2
import random
```

```
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.03.tiff', 0)
output = np.zeros(img.shape, np.uint8)
p = 0.05
for i in range (img.shape[0]):
    for j in range(img.shape[1]):
        r = random.random()
        if r < p/2:
            output[i][j] = 0
        elif r < p:
            output[i][j] = 255
        else:
            output[i][j] = img[i][j]
plt.imshow(output, cmap='gray')
plt.title('Salt and Pepper Sprinkled')
plt.axis('off')
plt.show()
```

In the preceding code, the noise density (denoted by p) is set to 0.05. We are generating a random number for each pixel and if it is less than $p/2$, we set the pixel to black. If it is between $p/2$ and p , then we set the pixel to white. Otherwise, the pixel is not modified. Since we are using the `random.random()` function to generate the noise, the generated noise is different each time we execute the program. The output with the introduced noise looks as follows:



Figure 7.1 – Salt and pepper noise

We can create a small app that adjusts the custom introduced noise with push buttons in the live webcam feed. Now, connect two push buttons to RPi's 7 and 11 GPIO pins in pull-up mode and write the following program:

```
import RPi.GPIO as GPIO
import cv2
import numpy as np
import random
p = 0.00
cap = cv2.VideoCapture(0)
ret, frame = cap.read()
output = np.zeros(frame.shape, np.uint8)
GPIO.setmode(GPIO.BRD)
GPIO.setwarnings(False)
button1 = 7
button2 = 11
GPIO.setup(button1, GPIO.IN, GPIO.PUD_UP)
GPIO.setup(button2, GPIO.IN, GPIO.PUD_UP)
```

In the preceding code, we are initializing the GPIO of the RPi and we are also creating the object for the USB webcam. Now, let's write the logic to adjust the amount of noise we get when pressing the push buttons:

```
while True:
    ret, frame = cap.read()
    button1_state = GPIO.input(button1)
    if button1_state == GPIO.LOW and p <= 0.1:
        p = p + 0.01
    if p > 0.1:
        p = 0.1
    button2_state = GPIO.input(button2)
    if button2_state == GPIO.LOW and p > 0:
        p = p - 0.01
    if p < 0:
        p = 0
    for i in range (frame.shape[0]):
        for j in range(frame.shape[1]):
```

```

        r = random.random()
        if r < p/2:
            output[i][j] = 0, 0, 0
        elif r < p:
            output[i][j] = 255, 255, 255
        else:
            output[i][j] = frame[i][j]
    print(p)
    cv2.imshow('Salt and pepper Noise App', output)
    if cv2.waitKey(1) == 27:
        break
cap.release()
cv2.destroyAllWindows()

```

The preceding program is computationally expensive because we are computing the noise and the output image continuously. If you are experiencing low frame rates, then reduce the resolution of the USB webcam connected to RPi. The output of the code will be like that shown in the preceding image.

Gaussian noise

This type of noise is named after the mathematician *Carl Friedrich Gauss* because the values of noise are normally distributed (also known as **gaussian distributed**). We can simulate this type of noise as follows:

```

import numpy as np
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.03.tiff', 0)
row, col = img.shape
img = img.astype(np.float32)
mean = 0
var = 0.1
sigma = var**0.5
gauss = np.random.normal(mean, sigma, (row, col))
gauss = gauss.reshape(row, col)
noisy = img + gauss

```

```
print(abs(noisy-img))
plt.imshow(noisy, cmap='gray')
plt.title('Gaussian (Normally distributed) Noise')
plt.axis('off')
plt.show()
```

The preceding code simulates Gaussian noise of a mean of 0 and 1, respectively, on a grayscale image. We are first converting the image from `uint8` into `float32` because the noise points can have floating values. We are using the `np.random.normal()` function to compute the data points for the noise. Note that the amount of noise it produces depends on the values of the mean and variance. For the values we used, the noise is not perceivable to us. Run the code and view the output. It will be as follows:



Figure 7.2 – Gaussian (normally distributed) noise

Poisson noise

The noise that is distributed according to the Poisson curve is known as **Poisson noise**. It is also known as **shot noise**. This phenomenon occurs because of the particle's nature in terms of light. Let's take a look at some example code where we'll introduce Poisson noise to an image:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.03.tiff', 0)
```

```
img = img.astype(np.float32)
vals = len(np.unique(img))
vals = 2 ** np.ceil(np.log2(vals))
noisy = np.random.poisson(img * vals) / float(vals)
print(abs(noisy-img))
plt.imshow(noisy, cmap='gray')
plt.title('Poisson Noise')
plt.axis('off')
plt.show()
```

The `np.random.poisson()` function produces random data points distributed along the Poisson curve. These data points are added to the image to create a noisy image with Poisson noise. Run the preceding code and view the output. It will be as follows:

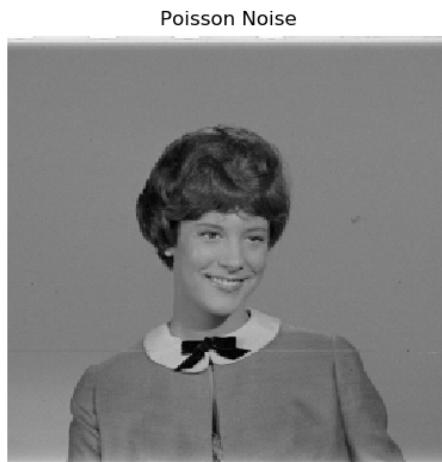


Figure 7.3 – Poisson noise

Random normal noise

We've already seen an example of Gaussian normal noise. We can also generate random normal noise, as follows:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.03.tiff', 0)
```

```
img = img.astype(np.float32)
row, col = img.shape
rand_noise = np.random.randn(row, col)
rand_noise = rand_noise.reshape(row, col)
noisy = img + img * rand_noise
print(abs(noisy-img))
plt.imshow(noisy, cmap='gray')
plt.title('Random Normal Noise')
plt.axis('off')
plt.show()
```

In the preceding code, the NumPy `np.random.randn()` function creates the data points for the random noise, which are then added to the image. This produces an image with the random noise applied. Run the preceding code and view the output. It will be as follows:

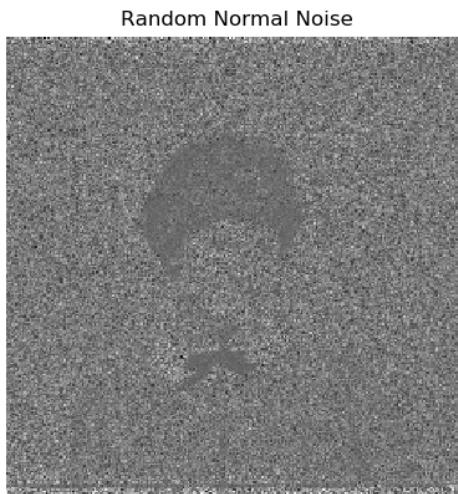


Figure 7.4 – Poisson noise

Working with kernels

Now, let's learn about kernels. We will learn how to use kernels for signal and image processing operations. Kernels are square numerical matrices. Depending on the size and components of the kernel, if we convolve the kernel with the image, we get blurred or sharpened output. Kernels are used for a variety of image processing operations.

Let's look at an example of a simple kernel used for averaging. It can be represented with the following formula:

$$k = (\text{Matrix of all ones}) / (\text{Number of rows} * \text{Number of columns})$$

By using the preceding formula, an averaging kernel that's 3x3 in size can be expressed as follows:

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 9$$

The value of the number of rows and the number of columns is always odd and always the same. They are all square matrices.

We can use the following NumPy code to create the preceding kernel:

```
K = np.ones((3, 3), np.uint8)/9
```

Now, we'll learn how to use the preceding kernel and other kernels to process the sample images from the dataset.

2D convolution with the signal processing module in SciPy

Now, let's take a look at the mathematical background of convolution. Convolution is understanding how the shape of a function is affected by another function. The process of computing it and the resultant function is known as a convolution. We can perform convolutions on 1D, 2D, and multidimensional data. Signals are multidimensional entities. Images are a type of signal. So, we can apply convolution to an image.

Note

You can read more about convolution at http://www.songho.ca/dsp/convolution/convolution2d_example.html.

We can perform convolution operations on images with various kernels to process images. For that, we will learn how to use the `signal` module from SciPy. Let's install the SciPy library with the following command:

```
pip3 install scipy
```

We can perform convolution operations on images with various kernels to process images. The function that performs convolution on 2D data is `signal.convolve2d()`. We must pass a grayscale image and a kernel as arguments to it, which then compute the convolution for the given data. The following is an example:

```
import scipy.signal
import numpy as np
import matplotlib.pyplot as plt
import cv2
img = cv2.imread('/home/pi/book/dataset/4.1.03.tiff', 0)
k1 = np.ones((7, 7), np.uint8)/49
blurred = scipy.signal.convolve2d(img, k1)
k2 = np.array([[0, -1, 0],
               [-1, 25, -1],
               [0, -1, 0]], dtype=np.int8)

sharpened = scipy.signal.convolve2d(img, k2)
plt.subplot(131)
plt.imshow(img, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(132)
plt.imshow(blurred, cmap='gray')
plt.title('Blurred Image')
plt.axis('off')
plt.subplot(133)
plt.imshow(sharpened, cmap='gray')
plt.title('Sharpened Image')
plt.axis('off')
```

The output is as follows:

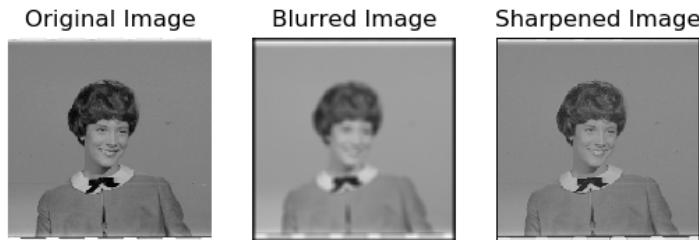


Figure 7.5 – Performing operations with kernels

As expected, the blur kernel produced a blurred output and the sharpening kernel produced a sharpened image. You may want to change the kernels and observe the effects on the image.

Filtering and blurring with OpenCV

OpenCV also has many filtering and convolution functions. These filtering functions are `cv2.filter2D()`, `cv2.boxFilter()`, `cv2.blur()`, `cv2.GaussianBlur()`, `cv2.medianBlur()`, `cv2.sepFilter2D()`, and `cv2.BilateralFilter()`. In this section, we will explore all these functions in detail.

2D convolution filtering

The `cv2.filter2D()` function, just like the `scipy.signal.convolve2d()` function, convolves a kernel with an image, thus applying a linear filter to the image. The advantage of the `cv2.filter2D()` function is that we can apply it to data that has more than two dimensions. We can apply this to color images, too.

This function accepts the input image, the depth of the output image (-1 means the input and the output have the same depth), and a kernel for the convolution operation as arguments. The following code demonstrates the usage of this function:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
input = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
output = cv2.filter2D(input, -1, np.ones((15, 15),
np.uint8)/225)
```

```
plt.subplot(121)
plt.imshow(input)
plt.title('Input')
plt.axis('off')
plt.subplot(122)
plt.imshow(output)
plt.title('Output')
plt.axis('off')
plt.show()
```

The following is the output:

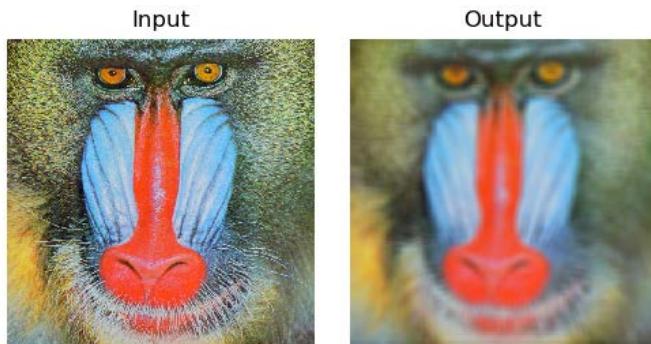


Figure 7.6 – Filtered and blurred versions of the same image

Note:

You can find interactive tutorials on convolution at the following URL: <http://micro.magnet.fsu.edu/primer/java/digitalimaging/processing/kernelmaskoperation/>

Low-pass filtering

As we discussed earlier, low-pass filters allow low-frequency components to pass through them. Edges and noise are usually high-frequency components. These are filtered out. So, low-pass filters are excellent for noise removal, blurring, and smoothing images.

The OpenCV library offers ready-made functions for performing low-pass filtering. We do not have to write programs from scratch to apply low-pass filters. These functions have code for the kernels written in their definition. We just have to pass arguments to the function and the function automatically creates the kernel and applies it to the image.

The `cv2.boxFilter()` function accepts the input source image, `ddepth`, and the size of the kernel as arguments, applies the kernel to the input image, and then returns the blurred image as output. The last parameter is `normalize`, which could be passed a Boolean value of `True` or `False`. This will decide whether the output is normalized. If normalization is passed the `True` value, the output is multiplied by $1/(number\ of\ rows * number\ of\ columns)$, which creates a normalized box filter effect, while if it is passed the `False` value, the output is multiplied by 1, which creates an unnormalized box filter effect.

The following line shows us an example of a normalized box filter:

```
output = cv2.boxFilter(input, -1, (3, 3), normalize=True)
```

The following line shows us an example of an unnormalized box filter:

```
output = cv2.boxFilter(input, -1, (3, 3), normalize=False)
```

The `cv2.blur()` function directly creates a normalized box filter and applies it to the image. We must pass the source input image and the size of the kernel as arguments. We do not have to specify if we want to have normalized output. This will produce the normalized output by default. The following two lines produce the same output:

```
output = cv2.blur(input, (3, 3))  
output = cv2.boxFilter(input, -1, (3, 3), normalize=True)
```

The OpenCV `cv2.GaussianBlur()` function applies a Gaussian kernel to the input image. We must pass the input source image and the size of the kernel as arguments to the call of this function. The third parameter is a standard deviation in the direction of the X axis. We are passing 0 as an argument for that. This function filters out all the Gaussian noise in the image. The following is the code example for this:

```
output = cv2.GaussianBlur(input, (3, 3), 0)
```

The OpenCV `cv2.medianBlur()` function applies a median filter and returns a blurred image. This filter is very effective against the images that have the salt-and-pepper type of noise. We need to pass the source input image and a number that defines the size of the square matrix as arguments for the call of this function, as follows:

```
output = cv2.medianBlur(img, 3)
```

This function computes the median of all the values of the members of the kernel. The value of the center of the kernel is replaced with the computed value of the median. This is a sliding window type of filter where the window of the matrix of the kernel slides over the matrix of the image and the pixel in the image that overlaps with the center of the kernel matrix is processed with a convolution operation using the computed value of the median.

The `cv2.sepFilter2D()` function applies a separable linear filter to an image. The following is a sample function call:

```
output = cv2.sepFilter2D(img, ddepth=-1, kernelX=1, kernelY=1,  
delta=1)
```

In the preceding function call, we have the following:

- `ddepth`: The depth of the output image (-1 if it is the same for the source and target images)
- `kernelX`: The coefficient for filtering each row
- `kernelY`: The coefficient for filtering each column
- `delta`: The constant value that's added to the filtered result

As an exercise for this chapter, you may want to use the `cv2.BilateralFilter()` function in one of your programs to filter an image.

Summary

In this chapter, we learned about noise and low-pass filtering techniques and how they are used to smooth images. The techniques we learned about in this chapter are very useful if we wish to remove various types of noise from images. You will use these techniques for removing, smoothing, and blurring noise while writing programs for real-life applications such as detecting movement in real time with a USB webcam.

In the next chapter, we will study high-pass filtering techniques and how to detect edges using various functions offered by OpenCV that implement various mathematical morphological operators.

8

High-Pass Filters and Feature Detection

In the previous chapter, we learned about kernels and low-pass filters and their applications. We learned about and demonstrated how to use low-pass filters in blurring, smoothing, and de-noising images.

In this chapter, we will learn about and demonstrate the uses of high-pass filters. This includes their application in image processing and computer vision. First, we will explore the Laplacian, Scharr, and Sobel high-pass filters. Then, we will learn about the Canny edge detection algorithm. We will also demonstrate Hough transforms for circles and lines. We will conclude by looking at corner detection with the Harris algorithm.

The following is a list of the topics we will cover in this chapter:

- Exploring high-pass filters
- Working with the Canny edge detector
- Finding circles and lines with Hough transforms
- Harris corner detection

After following this chapter, you will be able to use high-pass filters to detect the features in input images, such as edges, corners, lines, and circles.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter08/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/2CFnnpnD>.

Exploring high-pass filters

The concept of high-pass filters is exactly the opposite of low-pass filters. High-pass filters allow high-frequency components of information (such as signals and images) to pass through them. That is why they are known as **high-pass filters**. In an image, edges are high-frequency components. The kernels we use in high-pass filters boost the intense components in an image. That is why when we apply high-pass filters to images, we get the edges in the output.

Note:

You can read more about high-pass filters at https://difractionlimited.com/help/maximdl/High-Pass_Filtering.htm. Another type of signal filter is band-pass filters, which allow signals in a range (or band) of frequencies to pass through them. These filters allow us to highlight the edges in images and reduce the noise by using blurring at the same time. You can read more about them at <https://homepages.inf.ed.ac.uk/rbf/HIPR2/freqfilt.htm>.

OpenCV has a lot of library functions that implement high-pass filters. We will look at how to use the `Laplacian()`, `Sobel()`, and `Scharr()` functions.

Note:

You can learn about the mathematical aspects of high-pass filtering in more detail by referring to the following web pages:

https://www.tutorialspoint.com/dip/Sobel_operator.htm

https://www.tutorialspoint.com/dip/Laplacian_Operator.htm

The following is a list of parameters commonly used by all of the high-pass filtering functions and their meanings:

- **src:** This is the parameter for the source image in which edges are to be detected.
- **ddepth:** This is the parameter for deciding the depth of the target image. -1 means the source image and the target image have the same depth. The high-pass filtering functions offered by OpenCV support the following combinations of the depths of source and target images:

Source image depth	Target image depth
CV_8U	-1/CV_16S/CV_32F/CV_64F
CV_16U/CV_16S	-1/CV_32F/CV_64F
CV_32F	-1/CV_32F/CV_64F
CV_64F	-1/CV_64F

Figure 8.1 – A list of filter functions supported by OpenCV

- **dx:** This is the order of the derivative of X (this is not required for `Laplacian()`).
- **dy:** This is the order of the derivative of Y (this is not required for `Laplacian()`).
- **ksize:** This is the size of the matrix for the kernel (this can be 1, 3, 5, or 7 for the `Sobel()` function or a positive odd number for the `Laplacian()` function, and it is not required for the `Scharr()` function).
- **scale:** This is the scale, which is optional. This is the factor of the optional scale for the computed Laplacian values. Scaling is not applied by default.
- **delta:** This is the value of delta. This is an optional constant and is added to the final output.
- **borderType:** This is the method for the extrapolation of pixels for the pixels located at the boundary.

Let's write some code to demonstrate the functionality of the `Sobel()`, `Laplacian()`, and `Scharr()` functions. In the following code, we are computing the Laplacian and the first-order derivative of X of the input image using the `Scharr()` and `Sobel()` functions:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.05.tiff', 0)
laplacian = cv2.Laplacian(img, ddepth=cv2.CV_32F, ksize=17,
```

```
scale=1, delta=0,
borderType=cv2.BORDER_DEFAULT)
sobel = cv2.Sobel(img, ddepth=cv2.CV_32F, dx=1, dy=0,
ksize=11, scale=1, delta=0,
borderType=cv2.BORDER_DEFAULT)
scharr = cv2.Scharr(img, ddepth=cv2.CV_32F, dx=1, dy=0,
scale=1, delta=0,
borderType=cv2.BORDER_DEFAULT)
images=[img, laplacian, sobel, scharr]
titles=['Original', 'Laplacian', 'Sobel', 'Scharr']
for i in range(4):
    plt.subplot(2, 2, i+1)
    plt.imshow(images[i], cmap = 'gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

The computation of the derivative of X of the image with the `Laplacian()`, `Scharr()`, and `Sobel()` functions returns the vertical edges in the input image. The following screenshot shows the output of the preceding code:

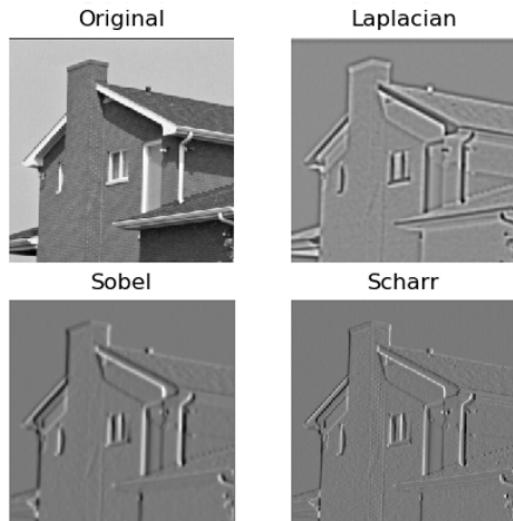


Figure 8.2 – The x derivative using a high-pass filter

We can connect two push buttons to the 7 and 11 GPIO pins in pull-up configuration and program them to adjust the values of dx and dy. The following is the code to do this:

```
import RPi.GPIO as GPIO
import cv2

x = 0
y = 1

cap = cv2.VideoCapture(0)

GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)

button1 = 7
button2 = 11

GPIO.setup(button1, GPIO.IN, GPIO.PUD_UP)
GPIO.setup(button2, GPIO.IN, GPIO.PUD_UP)

while True:
    print(x, y)
    ret, frame = cap.read()

    button1_state = GPIO.input(button1)

    if button1_state == GPIO.LOW:
        x = 0
        y = 1

    button2_state = GPIO.input(button2)

    if button2_state == GPIO.LOW:
        x = 1
        y = 0
```

Now, let's compute the output image with the `cv2.Scharr()` function:

```
output = cv2.Scharr(frame, ddepth=cv2.CV_32F,
                     dx=x, dy=y,
                     scale=1, delta=0,
                     borderType=cv2.BORDER_DEFAULT)

cv2.imshow('Salt and pepper Noise App', output)
if cv2.waitKey(1) == 27:
    break
cap.release()
cv2.destroyAllWindows()
```

Run the preceding program and observe the edge detection on the live video feed from the USB webcam connected to the Raspberry Pi board. We can also add the X derivative to the Y derivative (computed with Scharr) of the same live video feed, as follows:

```
import cv2
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    output1 = cv2.Scharr(frame, ddepth=cv2.CV_32F,
                         dx=0, dy=1,
                         scale=1, delta=0,
                         borderType=cv2.BORDER_DEFAULT)
```

The previous code segment computes the Scharr derivative of the Y axis. Now, let's write the code for the Scharr derivative of the X axis, as follows:

```
output2 = cv2.Scharr(frame, ddepth=cv2.CV_32F,
                     dx=1, dy=0,
                     scale=1, delta=0,
                     borderType=cv2.BORDER_DEFAULT)

cv2.imshow('Addition of Vertical and Horizontal',
           cv2.add(output1, output2))
if cv2.waitKey(1) == 27:
```

```
break  
cap.release()  
cv2.destroyAllWindows()
```

Run the preceding program and observe the added X and Y Scharr derivatives. You can implement similar programs with Sobel derivatives. All of these filters are used for detecting edges in the image.

In the next section, we will see how to use high-pass filters to detect edges in an image with the Canny edge detection algorithm.

Working with the Canny edge detector

The Canny edge detection algorithm was developed by John Canny. Canny's algorithm heavily uses the concept of high-pass filters. It has multiple steps.

Note:

You can read more about the Canny edge detection algorithm at <http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm>.

OpenCV has the `cv2.Canny()` function, which offers Canny's algorithm. The following are the steps of the algorithm:

1. A Gaussian kernel with a size of 5 x 5 pixels is applied to the input image to remove any noise.
2. Then, we compute the gradient of the intensity of the filtered image. We can use the L1 or the L2 norm for this step.
3. We then apply non-maximum suppression and identify the candidates for the possible sets of edges.
4. The final step is the operation of hysteresis. We finalize the edges depending on the thresholds passed to the images.

Note:

You can read more about the L1 and L2 norms and non-maximum suppression at <http://www.chioka.in/differences-between-the-l1-norm-and-the-l2-norm-least-absolute-deviations-and-least-squares/> and <https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c>.

The following is a list of parameters for the `cv2.Canny()` function:

- `img`: The input source image where we need to detect edges.
- `threshold1`: The lower bound for the threshold.
- `threshold2`: The upper bound for the threshold.
- `L2gradient`: If this value is `True`, the function uses the L2 norm to compute the set of edges, which is more accurate but computationally expensive. If it is `False`, then the L1 norm is used to compute the set of edges, which requires less computation but is less accurate.

This function computes and returns the set of detected edges in the source input image. The following code demonstrates this concept well:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.05.tiff', 0)
edges1 = cv2.Canny(img, 50, 300, L2gradient=False)
edges2 = cv2.Canny(img, 100, 150, L2gradient=True)
images = [img, edges1, edges2]
titles = ['Original', 'L1 Gradient', 'L2 Gradient']
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(images[i], cmap = 'gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

The output of the preceding code is as follows:



Figure 8.3 – The output of Canny edge detection

We can make the preceding program more interesting by computing the edges in real time, such that the thresholds are adjustable by OpenCV's trackbars:

```
import cv2
cv2.namedWindow('Canny')
img = cv2.imread('/home/pi/book/dataset/4.1.05.tiff', 0)
def empty(z):
    pass
cv2.createTrackbar('Threshold 1', 'Canny', 50, 100, empty)
cv2.createTrackbar('Threshold 2', 'Canny', 150, 300, empty)
while(True):
    l1 = cv2.getTrackbarPos('Threshold 1', 'Canny')
    l2 = cv2.getTrackbarPos('Threshold 2', 'Canny')
    output = cv2.Canny(img, l1, l2, L2gradient=False)
    cv2.imshow('Canny', output)
    if cv2.waitKey(1) == 27:
        break
cv2.destroyAllWindows()
```

In the previous code, we created two trackbars for the upper and lower thresholds of the Canny algorithm. We used the L1 norm to compute the edges. The output will be as follows:



Figure 8.4 – The output of the Canny edge detection algorithm with trackbars

We can apply this algorithm on real-life images, such as a live video feed from our webcam. In the next section, we will learn how to detect circles and lines with the Hough transform.

Finding circles and lines with Hough transforms

OpenCV offers a `cv2.HoughCircles()` function to detect circles in an image with Hough's method. This returns the centers and radii of the detected circles. It accepts an image, the (`cv2.HOUGH_GRADIENT`) method of detection, the inverse ratio of the resolution, the minimum distance between the centers of the circles to be detected, the highest threshold of the Canny method used internally, the threshold for the accumulator, and the maximum and minimum distances of the circles to be detected.

Note:

You can find more details about the mathematical aspects of Hough transforms for circles at https://www.cis.rit.edu/class/simg782/lectures/lecture_10/lec782_05_10.pdf.

In the following code, we accept the live video feed from a USB webcam as input. Then, we remove the noise by blurring the input frame, and then we pass the blurred frame to the call of the `cv2.HoughCircles()` function. Then, we visualize the detected circles with the `cv2.Circle()` function, as follows:

```
import cv2
cap = cv2.VideoCapture(0)
while (True):
    ret , frame = cap.read()
    grey = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    blur = cv2.blur(grey, (5, 5))
    circles = cv2.HoughCircles(blur,
                               method=cv2.HOUGH_GRADIENT,
                               dp=1, minDist=200,
                               param1=50, param2=13,
                               minRadius=30, maxRadius=175)
    if circles is not None:
        for i in circles [0,:]:
            cv2.circle(frame, (i[0], i[1]), i[2], (0, 255, 0),
2)
```

```
        cv2.circle(frame, (i[0], i[1]), 2, (0, 0, 255), 3)
cv2.imshow('Detected', frame)
if cv2.waitKey(1) == 27:
    break
cv2.destroyAllWindows()
cap.release()
```

Run the preceding program and observe the output. It should look as follows:

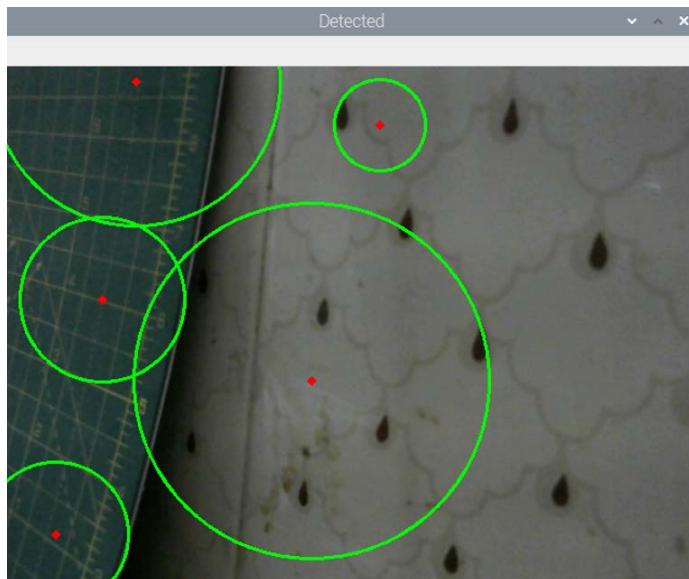


Figure 8.5 – The detected circles

The OpenCV `cv2.HoughLines()` function detects lines in an image. It accepts a grayscale image, the value of rho (the distance accuracy of an accumulator), theta (the angle accuracy of the accumulator), and the parameter of the threshold for the accumulator as arguments. We will demonstrate this with a live USB webcam video feed. The returned output is in polar format, which must be converted into the X/Y coordinate system before visualization:

```
import numpy as np
import cv2
cap = cv2.VideoCapture(0)
while True:
    ret, img = cap.read()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
edges = cv2.Canny(gray, 50, 250, apertureSize=5,
                   L2gradient=True)
lines = cv2.HoughLines(edges, 1, np.pi/180, 200)
if lines is not None:
    for rho,theta in lines[0]:
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a*rho
        y0 = b*rho
        pts1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
        pts2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))
        cv2.line(img, pts1, pts2, (0, 0, 255), 2)
cv2.imshow('Detected Lines', img)
if cv2.waitKey(1) == 27:
    break
cv2.destroyAllWindows()
cap.release()
```

Run the preceding code and observe its output. The output is as follows:

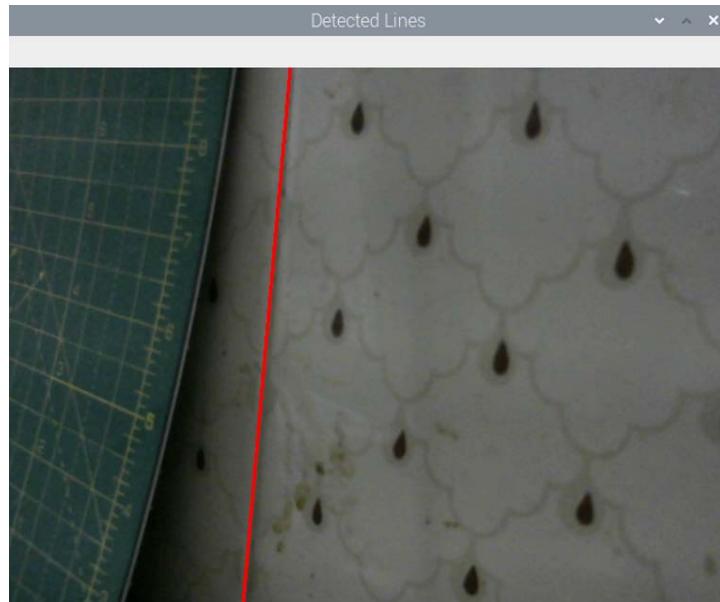


Figure 8.6 – The detected lines

The Hough transforms must be finely adjusted for the given input. This means that if we cannot see any lines or circles in the correct places, then we can try adjusting the value of the arguments passed to these Hough transform functions. Sometimes, it could produce false results, as in lines and circles will be visible even when there are none in the input frame. Again, for correct results, we must adjust the value of the arguments passed to these functions.

Harris corner detection

OpenCV has the `cv2.cornerHarris()` function for detecting corners. Its arguments are as follows:

- `img`: The input image, which must be grayscale and have the `float32` type.
- `blockSize`: This is the size of the neighborhood considered for corner detection.
- `ksize`: The aperture parameter of the Sobel derivative used.
- `k`: The free Harris detector parameter used in the equation.

The following is an example program that implements Harris corner detection:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.1.05.tiff', 0)
img = np.float32(img)
dst = cv2.cornerHarris(img, 2, 3, 0.04)
ret, dst = cv2.threshold(dst, 0.01*dst.max(), 255, 0)
dst = np.uint8(dst)
plt.imshow(dst, cmap='gray')
plt.axis('off')
plt.show()
```

In the preceding program, we converted the image into 32-bit float format and then we fed it to the corner detection function. Then, we thresholded the image. We used $0.01 * \text{dst}.\text{max}()$ as the value of the threshold to compute the binary image. Then, we converted the output into 8-bit integer format so that the output image could be displayed with `matplotlib`, as follows:



Figure 8.7 – The detected corners

We can use this corner detection method in industrial and robotics applications to detect the corners of regular and predictable objects. It is very useful in real-world automation.

Exercise

To practice what you have learned in this chapter, explore the `HoughLinesP()`, `goodFeaturesToTrack()`, and `FastFeatureDetector()` functions in OpenCV for detecting various features. Write programs using these functions to detect lines using probabilistic Hough transforms and other features.

Summary

In this chapter, we learned the concept and demonstration of high-pass filters. We applied high-pass filters on images to obtain various results. We also demonstrated the various techniques for detecting features, such as corners, lines, edges, and circles. All of these feature-detection algorithms rely on high-pass filtering. Canny's algorithm for edge detection uses Gaussian high-pass filters. The Harris corner detection algorithm uses Sobel spatial derivatives. All of these geometric feature-detection algorithms are routinely employed in real life in industrial automation, smart vehicles, and robotics.

In the next chapter of this book, we will learn the concepts and demonstrate the restoration of degraded images; the segmentation of images; k-means clustering of one-, two-, and multi-dimensional data; image quantization using k-means clustering; and the estimation of a depth map in detail.

9

Image Restoration, Segmentation, and Depth Maps

In the previous chapter, we demonstrated how to use high-pass filters and their applications in algorithms to detect edges.

In this chapter, we will learn about a few more advanced processing techniques regarding images. First, we will get started with the restoration of damaged or degraded images. Then, we will explore the fundamentals of various types of segmentation techniques. We have already seen that thresholding is a basic form of segmentation. We will explore this concept in more detail in this chapter. Finally, we will compute the disparity map and estimate the depths of objects in an image.

In this chapter, we will cover the following topics:

- Restoring damaged images using inpainting
- Segmenting images
- Disparity maps and depth estimation

By the end of this chapter, we will be able to restore damaged images, apply various segmentation algorithms to images, and estimate the depth of objects using disparity maps.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter09/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/2NsIzXY>.

Restoring damaged images using inpainting

The **restoration of an image** is the computational process of reconstructing damaged parts from existing parts of an image. If we capture an image on film with a photographic camera and develop it on paper, the photographic paper tends to degrade with the passage of time, leading to degradation of the photograph. Faulty sensors and imperfections such as dust and dirt on the camera lenses can introduce errors in the captured image. The process of transmission and reception can also introduce errors in the digital image. Image inpainting techniques can restore degraded and damaged images. Many algorithms are available to repair images. The OpenCV library implements two of the repairing methods using the `cv2.inpaint()` function.

This function accepts a degraded or damaged source image, a mask for image inpainting, the size of the inpainting neighborhood, and the inpainting method as arguments. The mask of inpainting is the damaged area represented by a grayscale image where white pixels refer to the area to be repaired or inpainted. The following code demonstrates both of the methods that we discussed above. The output produced by both methods is almost the same. We can create the damaged mask using free image editing software such as GIMP. Take a look at the following code:

```
import cv2
import matplotlib.pyplot as plt
image = cv2.imread('/home/pi/book/dataset/Damaged.tiff')
mask = cv2.imread('/home/pi/book/dataset/Mask.tiff', 0)
input = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
output_TELEA = cv2.inpaint(input, mask, 5, cv2.INPAINT_TELEA)
output_NS = cv2.inpaint(input, mask, 5, cv2.INPAINT_NS)
plt.subplot(221)
plt.imshow(input)
plt.title('Damaged Image')
plt.axis('off')
plt.subplot(222)
plt.imshow(mask, cmap='gray'),
```

```
plt.title('Mask')
plt.axis('off')
plt.subplot(223),
plt.imshow(output_TELEA)
plt.title('Telea Method')
plt.axis('off')
plt.subplot(224)
plt.imshow(output_NS)
plt.title('Navier Stokes Method')
plt.axis('off')
plt.show()
```

In the preceding code, we have used two techniques. The `cv2.INPAINT_TELEA` flag is based on a technique described in the paper named *An Image Inpainting Technique Based on the Fast Marching Method*, which was written and published in 2004 by Alexandru Telea.

The `cv2.INPAINT_NS` flag is based on a technique described in the paper *Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting*, which was written and published in 2001 by Bertalmio Marcelo, Andrea L. Bertozzi, and Guillermo Sapiro.

The following is the output:

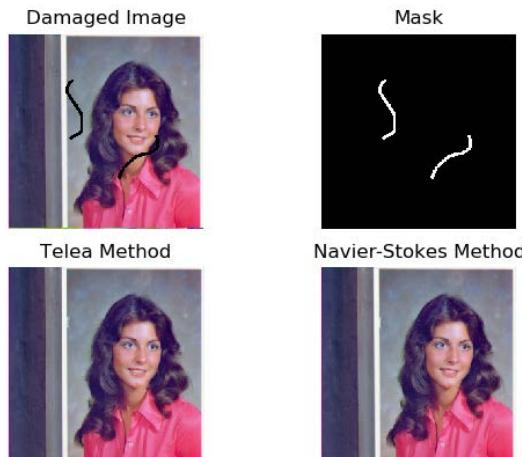


Figure 9.1 – The restoration of degraded images

In the preceding output, the first image is the damaged image. The second image is the binary mask corresponding to the damage. The images in the second row are the restored images using the **Telea** method and the **Navier-Stokes** method.

Note

You can find out more about image inpainting at <https://www.math.ucla.edu/~imagers/htmls/inp.html>.

Segmenting images

The segmentation of images is the process of dividing images into many sections or parts, also known as segments. This process is carried out using particular criteria. The simplest way in which we can divide images into segments is through thresholding. We have already learned about and demonstrated the techniques of thresholding in *Chapter 6, Colorspaces, Transformations, and Thresholding*. We will demonstrate two more methods of segmentation in this chapter. Those methods are the **Mean Shift** algorithm and **k-means clustering**.

Mean shift algorithm segmentation

Bogdan Georgescu and Chris M. Christoudias developed the mean shift algorithm and implemented it in C++. The Python implementation of the same algorithm is known as **PyMeanShift**. PyMeanShift uses ndarrays and NumPy for storing and processing images. That is why it is compatible with NumPy-based image processing libraries such as OpenCV, Mahotas, and scikit-image.

Note

You can find out more about this on the project GitHub page at <https://github.com/fjean/pymeanshift>.

There is no binary package for the installation of PyMeanShift on Linux, Unix, and other operating systems based on them. We must build it and install it from the source. Download the latest version of the source code from this URL: <https://github.com/fjean/pymeanshift>. The download will be a ZIP file. Copy it to the home directory of the pi user, /pi/home, and extract it. Navigate to the directory where we extracted it and run the following commands in order on LXTerminal:

```
cd ~  
cd pymeanshift-master/  
sudo python3 setup.py build  
sudo python3 setup.py install
```

Once the installation is complete, run the following command on Command Prompt to check whether it was successful:

```
python3 -c "import pymeanshift as pms"
```

The `pymeanshift` library offers the `pms.segment()` function, which segments the images represented by NumPy ndarrays. It accepts the source input image to be segmented, the spatial radius, the radius of the range, and the minimum density as arguments. Then, it returns a segmented image, a labeled color image, and a set of regions. The following is the code example to demonstrate the functionality:

```
import cv2
import pymeanshift as pms
from matplotlib import pyplot as plt
img = cv2.imread('/home/pi/book/dataset/house.tiff', 1)
input = cv2.cvtColor(img, cv2.COLOR_BGR2RGB )
(segmented_image, labels_image, number_regions) = pms.segment(
    input, spatial_radius=2, range_radius=2, min_density=300)
plt.subplot(131)
plt.imshow(input)
plt.title('Input')
plt.axis('off')
plt.subplot(132)
plt.imshow(segmented_image)
plt.title('Segmented Output')
plt.axis('off')
plt.subplot(133)
plt.imshow(labels_image)
plt.title('Labeled Output')
plt.axis('off')
plt.show()
```

The output of the preceding code is as follows:

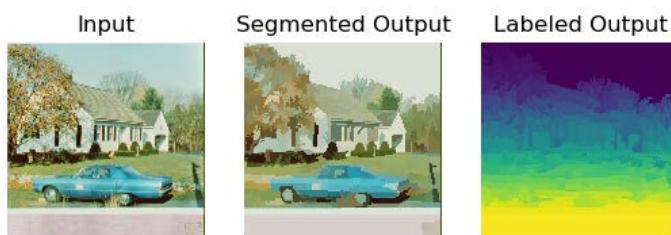


Figure 9.2 – Segmentation with PyMeanShift

As an exercise, pass different values of arguments to the function parameters and compare the output. We can apply this to a live feed video from a webcam as follows:

```
import cv2
import pymeanshift as pms
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    (segmented_image, labels_image, number_regions) = pms.
    segment(
        frame, spatial_radius=2, range_radius=2, min_
        density=50)
    cv2.imshow('Segmented', segmented_image)
    if cv2.waitKey(1) == 27:
        break
cv2.destroyAllWindows()
cap.release()
```

Usually, the segmentation is computationally a very expensive operation and, therefore, the **frames per second (FPS)** for a live video is very low. The output of this will be similar to the previous one (*Figure 2*).

K-means clustering and image quantization

The k-means clustering algorithm is a classification algorithm. Suppose that the input to the algorithm is a set of size n , and then the output divides that set into k number of partitions. That is why it is known as the k-means algorithm. Essentially, based on particular criteria, we are dividing or classifying the data into k number of classes or partitions. When this is applied to the data with two or more dimensions (that is, multidimensional data), it is called clustering. OpenCV has the `cv2.kmeans()` function that implements the k-means clustering algorithm for single-dimensional and multidimensional data. It accepts the arguments for the following parameters:

- **Data:** This is the input data to the k-means clustering algorithm. This data must be in the floating-point numerical format.
- **K:** The total number of partitions in the output of the algorithm. It must be known in advance (if the input is the color image, this will mean the number of colors in the output segmented image).
- **Criteria:** The termination criteria for the algorithm.

- **Attempts:** The number of times the algorithm is run with the different initial labels.
- **Flags:** This signifies the position of the initial centers for the clusters, which are passed in any one of the following values as arguments:

```
cv2.KMEANS_RANDOM_CENTERS  
cv2.KMEANS_PP_CENTERS  
cv2.KMEANS_USE_INITIAL_LABELS
```

Let's try to demonstrate this program for one-dimensional data first. We will create our own random data for this. Let's create and visualize the data:

```
import numpy as np  
import cv2  
from matplotlib import pyplot as plt  
x = np.random.randint(25, 100, 25)  
y = np.random.randint(175, 255, 25)  
z = np.hstack((x, y))  
z = z.reshape((50, 1))  
z = np.float32(z)  
plt.hist(z, 256, [0, 256])  
plt.show()
```

The sample random data will look like the following output:

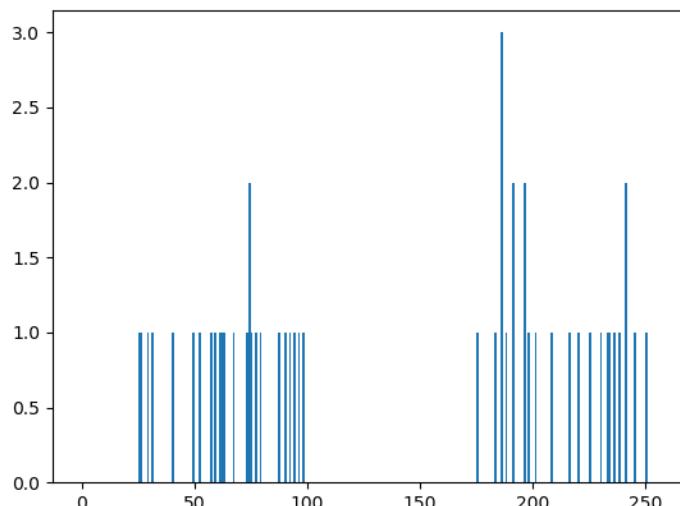


Figure 9.3 – One-dimensional data

We can clearly see the data divided into two groups. Now, let's make Raspberry Pi classify it and highlight the groups and their centers. Remove or comment out the last two lines of the preceding code and add the following lines:

```
criteria = (cv2.TERM_CRITERIA_EPS +
            cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
flags = cv2.KMEANS_RANDOM_CENTERS
compactness, labels, centers = cv2.kmeans(z, 2,
                                           None,
                                           criteria,
                                           10, flags)

A = z[labels==0]
B = z[labels==1]
plt.hist(A, 256, [0, 256], color = 'g')
plt.hist(B, 256, [0, 256], color = 'b')
plt.hist(centers, 32, [0, 256], color = 'r')
plt.show()
```

Let's run the preceding program. Note that we are rerunning the calls to the `np.random.randint()` functions so the dataset will be slightly different. Nevertheless, it will have two different groups, which are highlighted as follows:

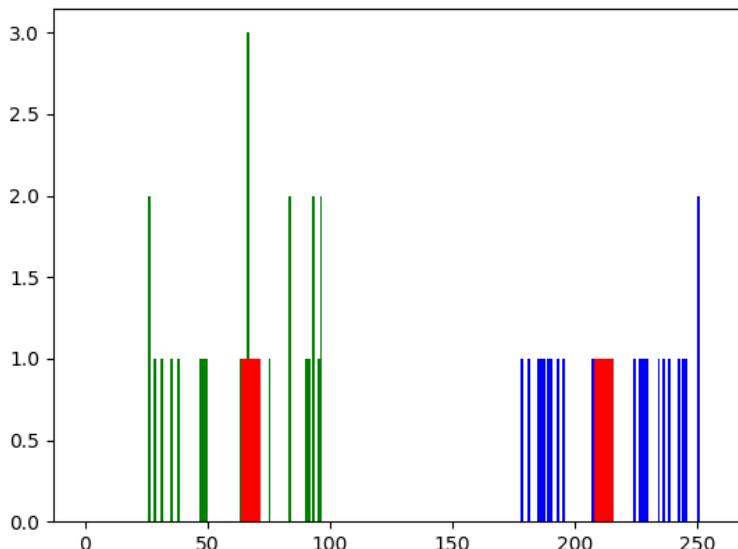


Figure 9.4 – K-means applied to one-dimensional data

We can implement this method in two-dimensional data as follows:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
X = np.random.randint(25, 50, (25, 2))
Y = np.random.randint(60, 85, (25, 2))
Z = np.vstack((X, Y))
Z = np.float32(Z)
criteria = (cv2.TERM_CRITERIA_EPS +
            cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret,label,center=cv2.kmeans(Z, 2, None, criteria,
                             10, cv2.KMEANS_RANDOM_CENTERS)

A = Z[label.ravel()==0]
B = Z[label.ravel()==1]
plt.scatter(A[:,0], A[:,1])
plt.scatter(B[:,0], B[:,1], c = 'g')
plt.scatter(center[:,0], center[:,1],
            s = 80, c = 'r', marker = 's')
plt.xlabel('X - Axis')
plt.ylabel('Y - Axis')
plt.show()
```

The output is as follows:

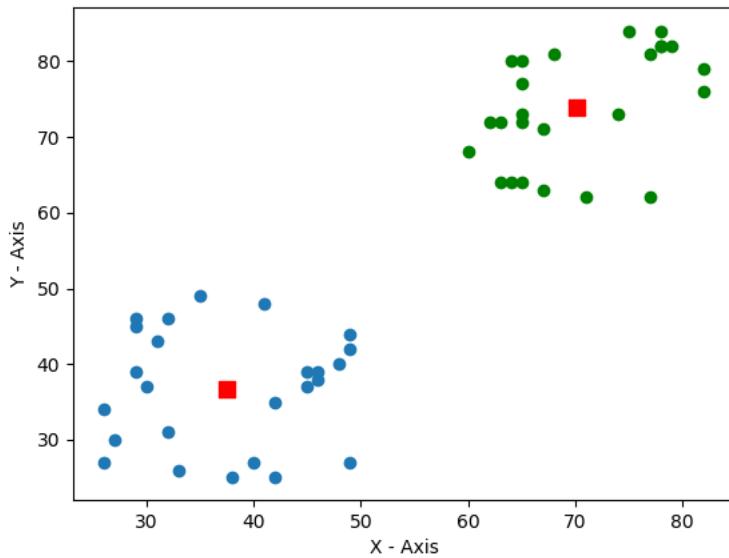


Figure 9.5 – K-means on two-dimensional data

In the preceding output, we can clearly see our data clustered into two groups. Let's write the code that applies the k-means clustering algorithm to a color image with the values for the sizes of k as 2, 4, and 12:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
Z = img.reshape((-1, 3))
Z = np.float32(Z)
criteria = (cv2.TERM_CRITERIA_EPS +
            cv2.TERM_CRITERIA_MAX_ITER,
            10, 1.0)
```

In the preceding code, we are reading the image in color mode and reshaping it. We are also converting it into 32-bit float format. Then, we are setting the criteria for clustering in the last line.

Let's compute the quantized image with the value of k as 2, as follows:

```
k = 2
ret, label1, center1 = cv2.kmeans(Z, k, None, criteria, 10,
                                    cv2.KMEANS_RANDOM_CENTERS)
center1=np.uint8(center1)
res1 = center1[label1.flatten()]
output1 = res1.reshape((img.shape))
```

In the previous code, we are computing the clusters with the `cv2.kmeans()` function and then flattening and reshaping them after converting the data into an 8-bit integer format.

Let's now compute the quantized image as follows with the value of k as 4:

```
k = 4
ret, label1, center1 = cv2.kmeans(Z, k, None, criteria, 10,
                                    cv2.KMEANS_RANDOM_CENTERS)
center1=np.uint8(center1)
res1 = center1[label1.flatten()]
output2 = res1.reshape((img.shape))
```

Let's now compute the quantized image as follows with the value of k as 12:

```
k = 12
ret, label1, center1 = cv2.kmeans(Z, k, None, criteria, 10,
                                    cv2.KMEANS_RANDOM_CENTERS)
center1=np.uint8(center1)
res1 = center1[label1.flatten()]
output3 = res1.reshape((img.shape))
```

Finally, let's display all the images in a grid using matplotlib:

```
output = [img, output1, output2, output3]
titles = ['Original Image', 'K=2', 'K=4', 'K=12']
for i in range(4):
    plt.subplot(2, 2, i+1)
    plt.imshow(output[i])
    plt.title(titles[i])
    plt.xticks([])
```

```
plt.yticks( [] )  
plt.show()
```

In the preceding code, initially, we are assigning random centers to all the clusters using the `cv2.KMEANS_RANDOM_CENTERS` flag. The following output of the program we wrote has the original image and the segmented images using quantization, with 2, 4, and 12 colors. The following is the output:

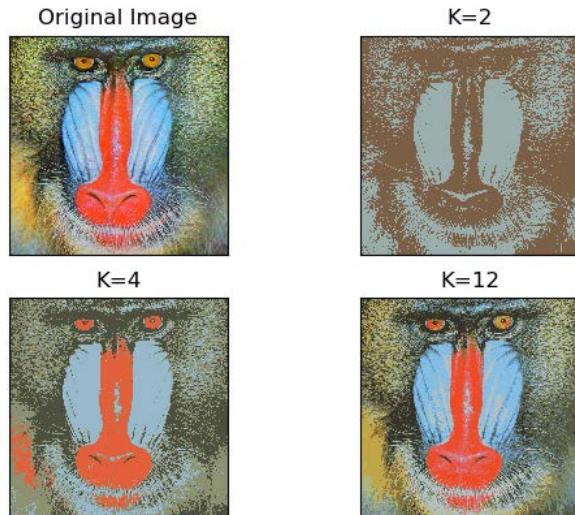


Figure 9.6 – K-means clustering and image quantization

As an exercise, run the preceding program with different values of the arguments to the functions and compare the outputs. It will be interesting to implement this on the live video from a webcam. Do not expect a high frame rate as it is computationally very expensive.

Comparison of k-means and the mean shift algorithm

The k-means algorithm has a time complexity of $O(n)$. The mean shift segmentation algorithm has a time complexity of $O(n^2)$. This difference of complexity is because the k-means algorithm provides the number of clusters through the argument at runtime. The mean shift segmentation algorithm must compute the number of clusters by itself at the time of execution. In applications where we do not know the number of clusters, we must use the mean shift algorithm. However, when we do know the number of clusters already, it is recommended that you use the k-means algorithm as it runs considerably faster when the number of clusters is known in advance.

Disparity maps and depth estimation

Disparity refers to the difference in the location of an object in the images captured by the left and right eyes or cameras. This difference or disparity is caused by parallax. Our brain uses this information regarding disparity to estimate the depth of objects (that is, their distance from us). We can compute the disparity between two images by applying this principle to every pixel in the pair of images captured by a webcam. This disparity information can be used to compute the estimated depth, thus mimicking the functionality of the brains of primates.

In terms of biology, this is known as **Stereoscopic Vision**, which enables us to see in three dimensions. OpenCV offers a `cv2.StereoBM`, `compute()` function that accepts the left image and the right image as an argument and returns a disparity map of the image pair. The `StereoBM_create()` function initializes the stereo state. It can have a number of disparities and block sizes as arguments. By default, they are 0 and 21, respectively. In the following example, we are calling this with default arguments. This stereo state is used to calculate the map of the disparities with the `cv2.StereoBM`, `compute()` functions. We need two images as inputs. One of these images corresponds to the input of the right camera, and the other corresponds to the input of the left camera. The following code demonstrates this concept using both functions:

```
import cv2
import matplotlib.pyplot as plt
Right= cv2.imread('/home/pi/book/dataset/imRsmall.jpg', 0)
Left = cv2.imread('/home/pi/book/dataset/imLsmall.jpg', 0)
stereo_BM_state=cv2.StereoBM_create()
output_map=stereo_BM_state.compute(Left, Right)
titles=['Left', 'Right', 'Depth Map']
output=[Left, Right, output_map]
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(output[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

The output of the preceding code will be as follows:



Figure 9.7 – Estimation of depth from the disparity map

In the preceding output, the brighter area in the output disparity map signifies more disparity. It means that objects in the source input image that correspond to the brighter areas in the output map of disparities are closer to the camera. Similarly, the darker colors in the output map of disparities signify that the object corresponding to those areas in the source input image is further away from the camera.

Summary

In this chapter, we learned about the concept of image inpainting and the restoration of damaged and degraded images. Then, we demonstrated many methods for the segmentation of images, including the mean shift algorithm and k-means clustering. Finally, we looked at how to estimate the depths of objects in images using disparity maps. All of these techniques are useful in many real-life applications. For example, whenever we want to send images over the network, we can use image quantization so that it consumes less bandwidth.

In the next chapter, we will learn about a few more advanced concepts such as histograms, the histograms of grayscale and color images, the detection of contours in images, and mathematical morphological operations.

10

Histograms, Contours, and Morphological Transformations

In the previous chapter, we learned about and demonstrated the basic- and intermediate-level concepts surrounding the areas of image processing and computer vision.

From this chapter onward, we will learn about and demonstrate advanced concepts that will prepare us for writing programs for applications in real life. First, we will look at the theoretical foundations of computing histograms with an `ndarray`. Then, we will learn how to compute it for grayscale and color image channels. We will also learn how to compute and visualize contours. Finally, we will learn about various mathematical morphological operations in detail and demonstrate how to use them with various structuring elements. We will learn about and demonstrate the following topics:

- Computing and visualizing histograms
- Visualizing image contours
- Applying morphological transformations to images

After completing this chapter, you will be able to comfortably work with all the concepts discussed throughout. These concepts are very useful when it comes to writing code for real-life applications in the area of computer vision.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter10/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/2NsLkZv>.

Computing and visualizing histograms

A histogram is a graphical representation of a distribution of data. Basically, it is the graphical depiction of a frequency distribution table. Let me explain this through an example. Suppose we have a dataset such as (1, 2, 1, 3, 4, 1, 2, 3, 4, 4, 2, 3, 4). Here, a frequency distribution looks as follows:

Element	Frequency of occurrence
1	3
2	3
3	3
4	4

Figure 10.1 – Frequency distribution

If we are to plot a bar graph so that the *x* axis represents elements and the *y* axis represents the frequency in which they occur, then this is known as a **histogram**. We can use `np.histogram()` to compute a histogram. `plt.hist()` can compute and directly plot it. Let's write some code that will use both functions to interpret the data in the preceding table:

```
import numpy as np
import matplotlib.pyplot as plt
a = np.array([1, 2, 1, 3, 4, 1, 2, 3, 4, 4, 2, 3, 4])
hist, bins = np.histogram(a)
```

```
print(hist)
print(bins)
plt.hist(a)
plt.show()
```

The output looks as follows:

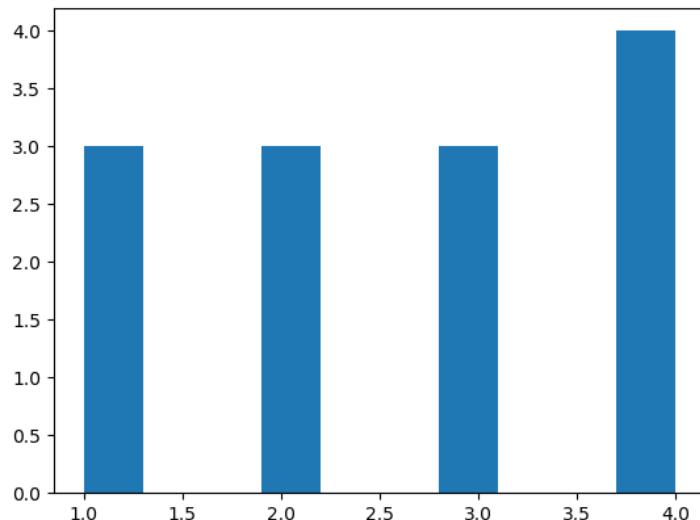


Figure 10.2 – Histogram of an ndarray

The graphical representation of the occurrences of shades of gray or the tones of colors is known as the histogram of an image. In the histogram of an image, we have the values of the shades of gray, or tones of colors, on the X-axis. The Y-axis represents the number of occurrences of those shades of gray for a grayscale image, or the tones of colors for a color image.

The values of the intensities of the gray or color image always range from 0 to 255 on the X-axis. The Y-axis shows the number of pixels. For a grayscale image, the histogram is computed for the complete image while for a color image, we compute the histograms of color channels separately.

For a color image, we can compute a channel-wise histogram. The following program visualizes the histogram for a grayscale image:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

```
img = cv2.imread('/home/pi/book/dataset/boat.512.tiff', 0)
plt.subplots_adjust(hspace=0.25, wspace=0.25)
plt.subplot(1, 2, 1)
plt.imshow(img, cmap='gray')
plt.axis('off')
plt.title('Original Image')
plt.subplot(1, 2, 2)
hist, bins = np.histogram(img.ravel(),
                           bins=256,
                           range=(0, 255))
plt.bar(bins[:-1], hist)
plt.title('Histogram')
plt.show()
```

In the preceding code, the `plt.subplots_adjust(hspace=0.25, wspace=0.25)` function call adjusts the horizontal and vertical spaces between the images in the subplots. We are using `np.histogram()` to compute the histogram of the image. We are using the `ravel()` function to flatten the image.

We know that the intensity levels of grayscale are between 0 and 255. Therefore, we are passing the relevant arguments for the bins and range. Finally, we are using `plt.bar()` to plot the histogram using the bar graph. The following is the output:

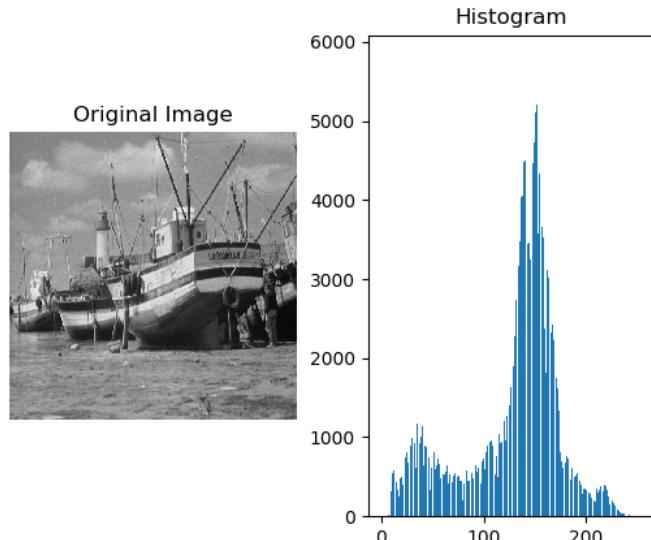


Figure 10.3 – Histogram of a grayscale image

We can even use the `plt.hist()` function to compute the same histogram. Just replace the lines containing `np.histogram()` and `plt.bar()` with the following line:

```
plt.hist(img.ravel(), bins=256, range=(0, 255))
```

The output is as follows:

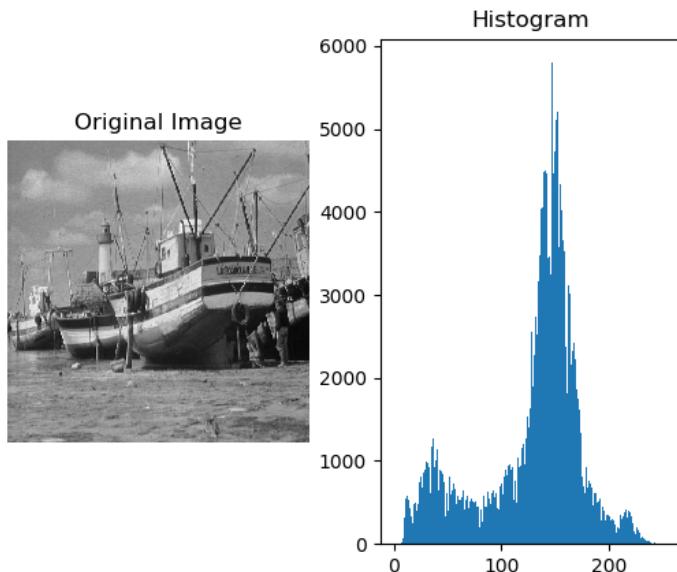


Figure 10.4 – Histogram of a grayscale image

As we can see, both methods produce the same output. We can compute the histogram for all the channels of a color image and show them with `plt.hist()`, as follows:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
img = cv2.imread('/home/pi/book/dataset/house.tiff', 1)
b = img[:, :, 0]
g = img[:, :, 1]
r = img[:, :, 2]
plt.subplots_adjust(hspace=0.5, wspace=0.25)
plt.subplot(2, 2, 1)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB),
cmap='gray')
```

```
plt.axis('off')
plt.title('Original Image')
plt.subplot(2, 2, 2)
plt.hist(r.ravel(), bins=256, range=(0, 255), color='r')
plt.title('Red Histogram')
plt.subplot(2, 2, 3)
plt.hist(g.ravel(), bins=256, range=(0, 255), color='g')
plt.title('Green Histogram')
plt.subplot(2, 2, 4)
plt.hist(b.ravel(), bins=256, range=(0, 255), color='b')
plt.title('Blue Histogram')
plt.show()
```

The output is as follows:

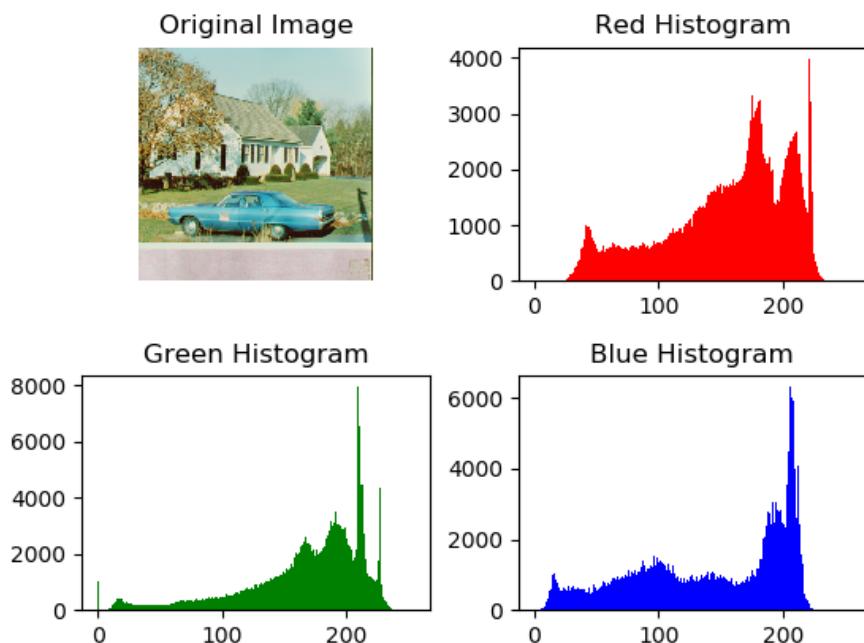


Figure 10.5 – Histogram of a color image

OpenCV offers the `cv2.calcHist()` function to compute and visualize histograms channels of color images separately. The `cv2.calcHist()` function accepts an image array, mask, channel index, range, and size as arguments to compute the histogram for a single channel of a color image. The following example demonstrates this by computing and visualizing a histogram for each color channel separately:

```
import cv2
from matplotlib import pyplot as plt
img = cv2.imread('/home/pi/book/dataset/house.tiff', 1)
input=cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
histr_RED = cv2.calcHist([input], [0], None, [256], [0, 255])
histr_GREEN = cv2.calcHist([input], [1], None, [256], [0, 255])
histr_BLUE = cv2.calcHist([input], [2], None, [256], [0, 255])
```

The preceding code computes the histograms for all the constituent channels of the input color image. Now, let's display the histogram with `matplotlib`, as follows:

```
plt.subplot(221)
plt.imshow(input)
plt.title('Original Image')
plt.axis('off')
plt.subplot(222)
plt.plot(histr_RED, color='r'),
plt.title('Red')
plt.xlim([0, 255])
plt.yticks([])
plt.subplot(223)
plt.plot(histr_GREEN, color='g')
plt.title('Green')
plt.xlim([0, 255])
plt.yticks([])
plt.subplot(224)
plt.plot(histr_BLUE, color='b')
plt.title('Blue')
plt.xlim([0, 255])
plt.yticks([])
plt.show()
```

The preceding code produces the following output. First, it shows the original image and then it will visualize the histograms of all the color channels:

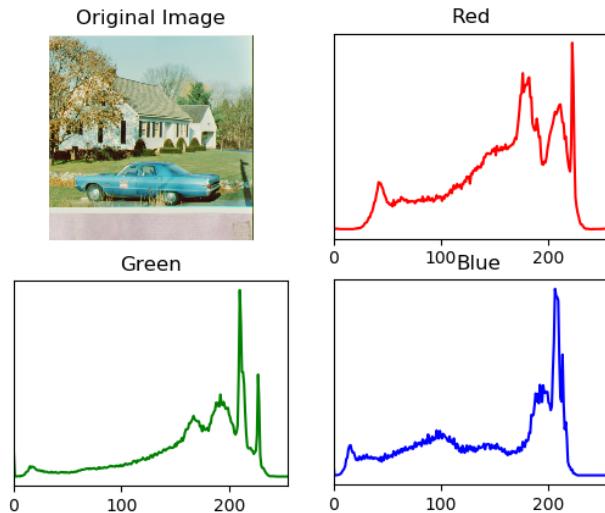


Figure 10.6 – Histogram of a color image

In the next subsection, we will learn about and demonstrate histogram equalization.

Histogram equalization

Histogram equalization is an image processing technique. It is used to improve the contrast in an image. It spreads out the most frequent intensity values. This means that the intensity range of the image is stretched out. This operation increases the contrast of the lower contrast areas, which enhances the images. There are multiple ways to equalize histograms. One option is that we can use the `cv2.equalizeHist()` function to equalize the histogram globally for an image. The other method we can use is called **contrast limited adaptive histogram equalization**. Unlike global histogram equalization, it computes several histograms for different regions of an image. This is also known as local histogram equalization:

1. In the following code, we are demonstrating how to equalize histograms for the individual channels of an RGB image and then merging them again to get a contrast enhanced output color image:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.2.03.tiff', 1)
```

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
R, G, B = cv2.split(img)
```

2. Let's equalize the color channels of the image separately and then merge the equalized channels to create the equalized image:

```
output1_R = cv2.equalizeHist(R)
output1_G = cv2.equalizeHist(G)
output1_B = cv2.equalizeHist(B)
output1 = cv2.merge((output1_R,
output1_G, output1_B))
```

3. Now, let's use the CLAHE method to equalize the color channels and then merge them to obtain the equalized image with CLAHE:

```
clahe = cv2.createCLAHE(clipLimit=2.0,
tileGridSize=(8, 8))
output2_R = clahe.apply(R)
output2_G = clahe.apply(G)
output2_B = clahe.apply(B)
output2 = cv2.merge((output2_R, output2_G,
output2_B))
output = [img, output1, output2]
titles = ['Original Image',
'Adjusted Histogram', 'CLAHE']
for i in range(3):
plt.subplot(1, 3, i+1)
plt.imshow(output[i])
plt.title(titles[i])
plt.axis('off')
plt.show()
```

The output of both methods is as follows. The first one is the original image, the second one is the histogram adjusted image, and the third image is a histogram equalized image that was produced with CLAHE:

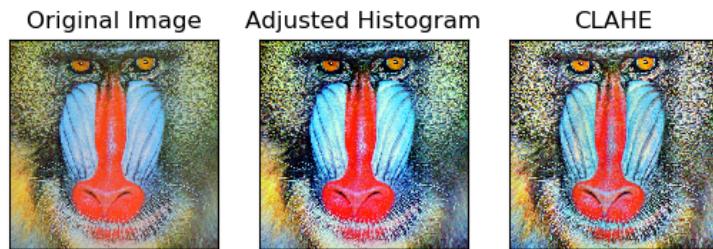


Figure 10.7 – Histogram equalization

In the next section, we will learn and demonstrate how to visualize image contours.

Visualizing image contours

A curve that joins all the points that lie continuously along the boundary that have the same value as the color of the pixels is known as a contour. Contours are used for detecting the boundaries in an image. Contours are also used for image segmentation. Contours are usually computed using edges in an image. However, contours are closed curves and that is their main distinction from the edges in an image. It is always a good idea to apply the thresholding operation to an image before we extract contours from an image. It will increase the accuracy of the computation of the contour operation.

The `cv2.findContours()` function is used to compute the contours in an image. This function accepts an image array, the mode of the retrieval of the contours, and the method for the approximation of contours as arguments. It then returns a list of computer contours in the image. The contour retrieval mode can be any of the following:

- `CV_RETR_CCOMP`
- `CV_RETR_TREE`
- `CV_RETR_EXTERNAL`
- `CV_RETR_LIST`

The method for the approximation of the contours can be any of the following:

- `CV_CHAIN_APPROX_TC89_L1`
- `CV_CHAIN_APPROX_TC89_KCOS`

- CV_CHAIN_APPROX_NONE
- CV_CHAIN_APPROX_SIMPLE

Once we've computed all the contours with the `cv2.findContours()` function, they can be visualized with the `cv2.drawContours()` function. We have already learned about and demonstrated the functions we can use to draw lines, circles, and other geometric shapes in *Chapter 4, Getting Started with Computer Vision*. The `cv2.drawContours()` function works in the same way. This function accepts the image array where contours are to be visualized, the list of detected contours using the `cv2.findContours()` function, the index of the contour to be drawn (we have to pass `-1` as an argument for this parameter in order to draw all the contours in the image), and the color and the thickness of the contour as arguments. The following program computes and visualizes all the contours in an image:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('/home/pi/book/dataset/4.2.07.tiff', 1)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray, 75, 255, 0)
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
                                         cv2.CHAIN_APPROX_SIMPLE)
cv2.drawContours(img, contours, -1, (0, 0, 255), 2)
original = cv2.imread('/home/pi/book/dataset/4.2.07.tiff', 1)
original = cv2.cvtColor(original, cv2.COLOR_BGR2RGB)
output = [original, img]
titles = ['Original', 'Contours']
for i in range(2):
    plt.subplot(1, 2, i+1)
    plt.imshow(output[i])
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

The output is as follows:

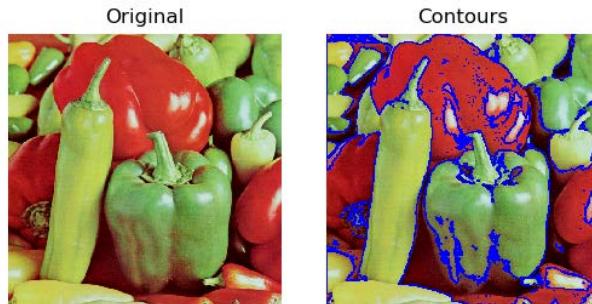


Figure 10.8 – Contours in a color image

In order to explore the concept of contours further and understand it better, write a few programs that use the `cv2.findContours()` and `cv2.drawContours()` functions with the different combinations of methods, colors, and modes. Then, compare all the output images with each other.

Applying morphological transformations to images

Morphological operations are mathematical in nature and they change the shape of an image. These operations can best be demonstrated visually with binary images. We can apply morphological operations to eliminate a lot of unnecessary information, such as noise, in an image. A morphological operation accepts an image and a kernel as inputs. We will create a custom binary image as a binary image since this is the most suitable way to visually demonstrate morphological operations.

The mathematical morphological operation of erosion contracts the boundaries in an image. In a binary image, the white part is considered the foreground and the black part is considered the background. The erosion operation sets all the pixels on the boundary of the background part that is white to black, thus effectively shrinking the white region. Morphological dilation is the exact opposite of the erosion operation. It adds the white pixels near the boundary of the foreground, so it effectively expands the white foreground in the image. The intensity of any morphological operation depends on the type and size of the kernel used in the operation and the number of times the operation is performed on an image. The morphological gradient operation is the computed difference between the dilation operation and the erosion operation.

Let's take a look at a few morphological operations in action. Now, let's import all the required libraries:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
Let us create a sample image, img = np.array([[0, 0, 0, 0, 0, 0, 0,
0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 255, 255, 255, 0, 0],
[0, 0, 255, 255, 255, 0, 0],
[0, 0, 255, 255, 255, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0]], dtype=np.uint8)
```

Let's create a kernel and compute the morphological operations:

```
kernel = np.ones((3, 3), np.uint8)
erosion = cv2.erode(img, kernel, iterations = 1)
dilation = cv2.dilate(img, kernel, iterations = 1)
gradient = cv2.morphologyEx(img,
                           cv2.MORPH_GRADIENT,
                           kernel)
titles=['Original', 'Erosion',
        'Dilation', 'Gradient']
output=[img, erosion, dilation, gradient]
```

Finally, let's visualize the computed outputs:

```
for i in range(4):
    plt.subplot(2, 2, i+1)
    plt.imshow(output[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```

The output of the preceding code is as follows:

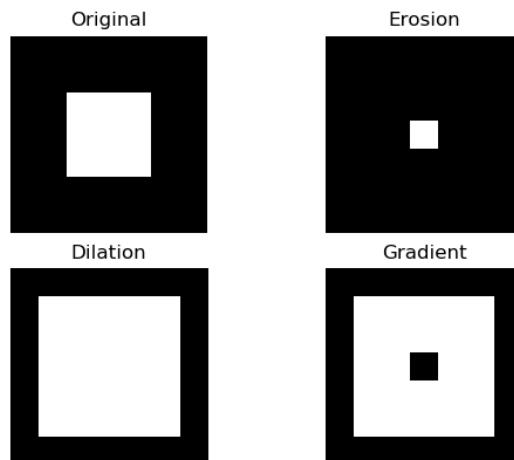


Figure 10.9 – Morphological operations

In the previous example, we first created a custom image to act as the source or the input. Then, we created a kernel that's 3x3 in size and applied it to the source image for all the mathematical morphological operations. OpenCV provides the `cv2.getStructuringElement()` function, which returns a custom kernel of the given shape and size in the arguments. The shape could be one of the values from `cv2.MORPH_CROSS`, `cv2.MORPH_RECT`, or `cv2.MORPH_ELLIPSE`. Also, the size that's passed must be an odd positive integer. You may want to print and see the values in the matrices that are used to represent the images in order to understand what exactly happens with numbers. Now, let's look at the various structuring elements:

1. Open Python 3 in interactive mode and run the following statements:

```
>>> import cv2  
>>> k = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))  
>>> k
```

The output is as follows:

```
array([[1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1]], dtype=uint8)
```

2. Let's look at an elliptical structural element:

```
>>> k = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))  
>>> k
```

The output is as follows:

```
array([[0, 0, 1, 0, 0],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [0, 0, 1, 0, 0]], dtype=uint8)
```

3. Let's look at a cross-structural element:

```
>>> k = cv2.getStructuringElement(cv2.MORPH_CROSS, (5, 5))  
>>> k
```

The output is as follows:

```
array([[0, 0, 1, 0, 0],  
       [0, 0, 1, 0, 0],  
       [1, 1, 1, 1, 1],  
       [0, 0, 1, 0, 0],  
       [0, 0, 1, 0, 0]], dtype=uint8)
```

Let's look at the remaining morphological operations by using a custom 3x3 cross kernel.

4. Let's import all the necessary libraries:

```
import numpy as np  
import cv2  
from matplotlib import pyplot as plt
```

5. The following lines create a sample binary image:

```
img = np.array([[0, 0, 0, 0, 0, 0, 0],  
               [0, 0, 0, 0, 0, 0, 0],  
               [0, 0, 255, 255, 255, 0, 0],  
               [0, 0, 255, 255, 255, 0, 0],  
               [0, 0, 255, 255, 255, 0, 0],
```

```
[0, 0, 0, 0, 0, 0, 0],  
[0, 0, 0, 0, 0, 0, 0]], dtype=np.uint8)
```

6. Let's now create the matrix for the structuring element:

```
kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,  
3))
```

7. Let's now apply the mathematical morphological operations to the sample binary image:

```
open = cv2.morphologyEx(img,  
cv2.MORPH_OPEN,  
kernel)  
close = cv2.morphologyEx(img,  
cv2.MORPH_CLOSE,  
kernel)  
tophat = cv2.morphologyEx(img,  
cv2.MORPH_TOPHAT,  
kernel)  
blackhat = cv2.morphologyEx(img,  
cv2.MORPH_BLACKHAT,  
kernel)  
hitmiss = cv2.morphologyEx(img,  
cv2.MORPH_HITMISS,  
kernel)
```

8. Let's now visualize the input and the output:

```
titles=['Original', 'Open',  
'Close', 'Top hat',  
'Black hat', 'Hit Miss']  
output=[img, open, close,  
tophat, blackhat,  
hitmiss]  
for i in range(6):  
    plt.subplot(2, 3, i+1)  
    plt.imshow(output[i], cmap='gray')
```

```

plt.title(titles[i])
plt.axis('off')
plt.show()

```

The output of the preceding code is as follows:

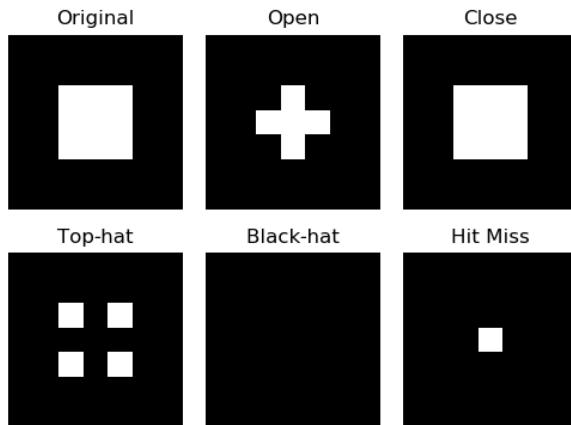


Figure 10.10 – More morphological operations

Let's understand the meaning of the operations we demonstrated here. Erosion followed by dilation is known as opening. Dilation followed by erosion is known as closing. Top hat extracts small elements and details from images. Top hat is the difference between the input image and the opening of the image. Black hat is the difference between the closing of the image and the image itself. Finally, hit-or-miss is an operation that detects a given configuration or pattern in a binary image.

Summary

In this chapter, we learned and demonstrated the concepts of histograms in general and saw how to create a simple histogram from a simple single-dimensional array. Then, we saw how to visualize a histogram for grayscale and color images. We also demonstrated how to use image contours. Finally, we visually demonstrated the operations that are performed in the area of mathematical morphology. These morphological operations will be extremely useful for real-life applications, some of which we will demonstrate in *Chapter 11, Real-Life Applications of Computer Vision*.

In the next chapter, we will demonstrate many of the concepts we learned in this and the earlier chapters by building real-life applications such as movement detectors, chroma keys with a green screen, and barcode detection in still images. It will be an exciting and interesting chapter that culminates all the knowledge we have gained so far.

11

Real-Life Applications of Computer Vision

In the previous chapter, we studied various advanced concepts in computer vision such as morphological operations and contours.

This chapter is the culmination of all the computer vision concepts we've learned and demonstrated in the earlier chapters. In this chapter, we will use the computer vision operation we learned about earlier in detail to implement a few real-life projects. We will also learn about a few new concepts such as background subtraction and the computation of optical flow and then demonstrate them for small applications. This chapter contains a lot of hands-on programming examples, as well as detailed explanations of the code and new functionality.

In this chapter, we will learn and demonstrate the code for the following topics:

- Implementing the Max RGB filter
- Implementing background subtraction
- Computing the optical flow
- Detecting and tracking motion
- Detecting barcodes in images
- Implementing the chroma key effect

After completing this chapter, you will be able to implement the concepts you've learned about to create real-life applications such as security systems and motion detection systems using the **Raspberry Pi (RPi)** and some camera sensors.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter11/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/2Z43syb>.

Implementing the Max RGB filter

We know that filters allow and block signals or data, depending on some criteria. Let's manually write the code for implementing a special filter based on the value of the intensity of the colors of pixels. This is known as the **Max RGB filter**. In a Max RGB filter, we compare the intensities of all the color channels of a color image for every pixel.

Then, we keep the intensity of the channel(s) with the maximum intensity and reduce the intensities of all the other channels to zero. This happens for every pixel in an image. Suppose, for a pixel, the intensities are (30, 200, 120). Then, after applying the Max RGB filter, it will be (0, 200, 0). Let's take a look at a program that will implement this with the NumPy and OpenCV functions:

```
import cv2
import numpy as np
def maxRGB(img):
    b = img[:, :, 0]
    g = img[:, :, 1]
    r = img[:, :, 2]
    M = np.maximum(np.maximum(b, g), r)
    b[b < M] = 0
    g[g < M] = 0
    r[r < M] = 0

    return cv2.merge((b, g, r))
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    cv2.imshow('Max RGB Filter', maxRGB(frame))
    if cv2.waitKey(1) == 27:
        break
cv2.destroyAllWindows()
cap.release()
```

Run the preceding program and view the output. It is interesting to see the filtered live feed. The output looks as follows:

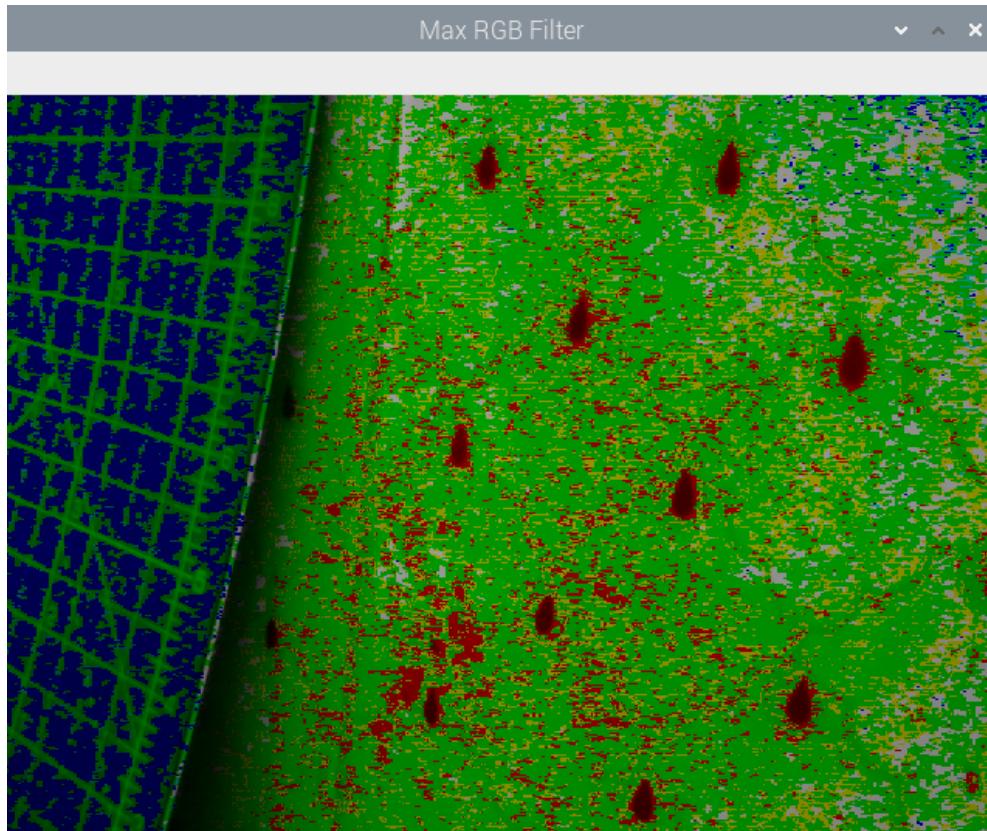


Figure 11.1 – Output of a Max RGB filter

In the next section, we will learn and demonstrate the concept of background subtraction.

Implementing background subtraction

Static cameras are used in many applications, such as security and monitoring. We can separate the background and moving objects by applying a process known as background subtraction. It usually returns a binary image with the background (the static part of the scene) in black pixels and the moving (changing or dynamic) parts in white pixels. OpenCV can implement this through two algorithms. The first is `createBackgroundSubtractorKNN()`. This creates a **K-Nearest Neighbour (KNN)** background subtractor object. Then, we can call the `apply()` function with the object to obtain the foreground mask. We can directly display the foreground mask in real time.

The following is a demonstration of how to use it:

```
import cv2
import numpy as np
cap = cv2.VideoCapture(0)
fgbg = cv2.createBackgroundSubtractorKNN()
while(True):
    ret, frame = cap.read()
    fgmask = fgbg.apply(frame)
    cv2.imshow('frame', fgmask)
    if cv2.waitKey(30) == 27:
        break
cap.release()
cv2.destroyAllWindows()
```

The output is a binary video stream, as shown in the following screenshot. I am waving my hand, which is highlighted by white pixels:



Figure 10.2 – Background subtraction with KNN

Note that if you keep your hand still for some time, OpenCV will consider it as a part of the background and will slowly dissolve it in the output.

Another similar function is `cv2.createBackgroundSubtractorMOG2()`. This also generates the foreground mask using the `apply()` function. The following is a sample program using it:

```
import cv2
import numpy as np
cap = cv2.VideoCapture(0)
fgbg = cv2.createBackgroundSubtractorMOG2()
while(True):
    ret, frame = cap.read()
    fgmask = fgbg.apply(frame)
    cv2.imshow('frame', fgmask)
    if cv2.waitKey(30) == 27:
        break
cap.release()
cv2.destroyAllWindows()
```

Run the preceding program and view the output. In both programs, we are creating the `fgbg` object and using the `apply()` function to compute the foreground mask, that is, `fgmask`. Then, we are just displaying the foreground mask in real time with the `imshow()` function. The expected output of this code can be seen in the preceding screenshot. Run the program and view the output for yourself.

Computing the optical flow

Optical flow (also known as optic flow) is the pattern that appears in the motion of objects in a video (live or recorded). Pay attention to the word *appearance* in the previous sentence. This means that if the observer (in our case, the camera) is in motion, then the objects in the scene are also considered to be moving, even if they are static. This is known as relative motion. In brief, the optical flow highlights the relative motion in the video. OpenCV has implementations for many of the functions that can compute the optical flow. The `cv2.calcOpticalFlowFarneback()` function computes the optical flow with the dense method. This means that it computes the flow for all the points. This function implements the Gunnar Farneback algorithm.

NOTE:

You can read more about the Gunner Farneback argument at the following URL:

[http://www.diva-portal.org/smash/get/diva2:273847/
FULLTEXT01.pdfTwo-Frame](http://www.diva-portal.org/smash/get/diva2:273847/FULLTEXT01.pdfTwo-Frame)

Let's see how we can compute the optical flow with OpenCV and Python 3 with the following code:

```
import cv2
import numpy as np
cap = cv2.VideoCapture(0)
ret, frame1 = cap.read()
prvs = cv2.cvtColor(frame1,
                    cv2.COLOR_BGR2GRAY)
hsv = np.zeros_like(frame1)
hsv[..., 1] = 255
while(cap):
    ret, frame2 = cap.read()
    next = cv2.cvtColor(frame2,
                        cv2.COLOR_BGR2GRAY)
    flow = cv2.calcOpticalFlowFarneback(prvs,
                                        next,
                                        None, 0.5,
                                        3, 15,
                                        3, 5,
                                        1.2, 0)
    mag, ang = cv2.cartToPolar(flow[..., 0],
                                flow[..., 1])
    hsv[..., 0] = ang * 180/np.pi/2
    hsv[..., 2] = cv2.normalize(mag, None, 0,
                               255, cv2.NORM_MINMAX)
    rgb = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
    cv2.imshow('Optical Flow', rgb)
    if cv2.waitKey(1) == 27:
        break
    prvs = next
```

```
cap.release()  
cv2.destroyAllWindows()
```

In the preceding program, `cv2.calcOpticalFlowFarneback()` returns the coordinates of flow in the XY (Cartesian) system. We then convert it into polar with the `cv2.cartToPolar()` function. Then, the hue shows the angle of the motion and the value shows the intensity of the motion in the final HSV frame, which is converted into BGR and shown as output. The output will look like what's shown in the preceding screenshot. The only difference will be that the optical flow will be denoted by various colors.

The concept of the optical flow has applications in the following areas:

- Object detection and tracking
- Movement detection and tracking
- Navigation of robots

Detecting and tracking motion

Let's build a system for detecting and tracking motion in real time with the RPi, OpenCV, and Python. We will use a very simple technique to detect motion. Basically, we will compute the difference between the successive frames of a video feed (a video file or a live feed from a USB webcam). Then, we will plot contours around the area of pixels where we wish to detect the difference between successive frames:

1. We will begin by importing OpenCV and NumPy. Also, initialize an object corresponding to the USB webcam:

```
import cv2  
import numpy as np  
cap = cv2.VideoCapture(0)
```

2. We will apply the dilation operation to the frames in the video. We need a kernel for that. We will define it before the video loop. Let's define it as follows:

```
k = np.ones((3, 3), np.uint8)
```

3. The following code captures and stores the successive frames in separate variables:

```
t0 = cap.read()[1]  
t1 = cap.read()[1]
```

- Now, let's write the block for the `while` loop. In this block, we compute the absolute difference between the frames we captured earlier. We are going to use the `cv2.absdiff()` function for this. Then, we will convert the computed absolute difference into grayscale to process it further:

```
while(True):  
    d=cv2.absdiff(t1, t0)  
    grey = cv2.cvtColor(d, cv2.COLOR_BGR2GRAY)
```

The following is the output of the preceding code. It shows the grayscale of the absolute difference between the successively captured frames:

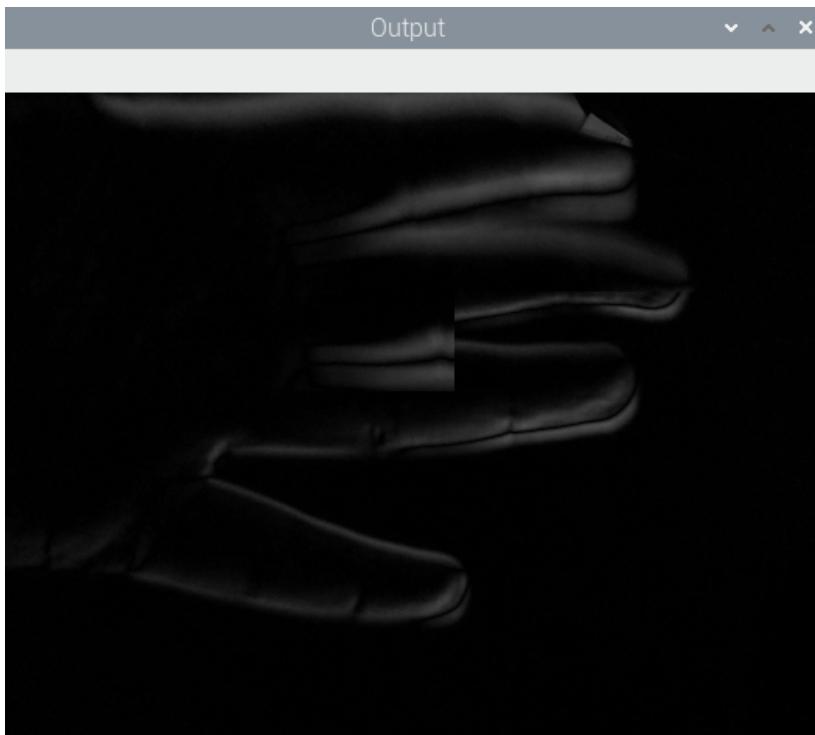


Figure 11.3 – Absolute difference between successive frames

- The output that we computed in the previous step has some noise. Due to this, we must blur it first with the Gaussian blurring technique to remove the noise:

```
blur = cv2.GaussianBlur(grey, (3, 3), 0)
```

6. We apply the technique of binary thresholding to transform the blurred output from the previous step into a binary image for further processing with the following code:

```
ret, th = cv2.threshold(blur, 15, 255, cv2.THRESH_BINARY)
```

7. Now, let's apply the dilation morphological operation to this binary image. This makes it easy to detect the boundaries in the thresholded image:

```
dilated = cv2.dilate(th, k, iterations=2)
```

The following is the output of the dilation operation:

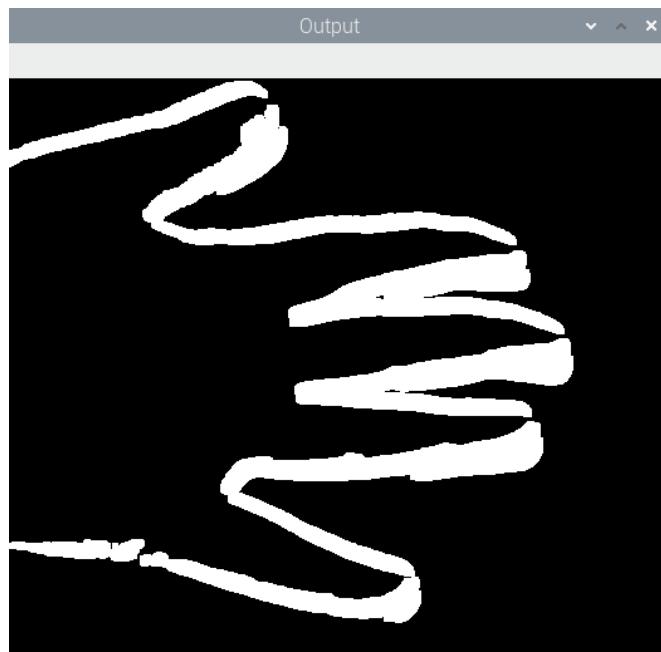


Figure 11.4 – Dilated output

8. Let's go ahead and find the contours in the dilated image:

```
contours, hierarchy = cv2.findContours(dilated,
                                         cv2.RETR_TREE,
                                         cv2.CHAIN_APPROX_SIMPLE)
t2=t0
cv2.drawContours(t2, contours, -1, (0, 255, 0), 2)
cv2.imshow('Output', t2)
```

- Now, let's copy the latest frame to the variable holding the older frame and then capture the next frame with the webcam:

```
t0=t1  
t1=cap.read() [1]
```

We end the while loop when the *Esc* key on the keyboard is pressed:

```
if cv2.waitKey(5) == 27 :  
    break
```

Once the while loop ends, we perform the usual cleaning-up tasks such as releasing the camera capture object and destroying the display window:

```
cap.release()  
cv2.destroyAllWindows()
```

The following is the output of executing the program:

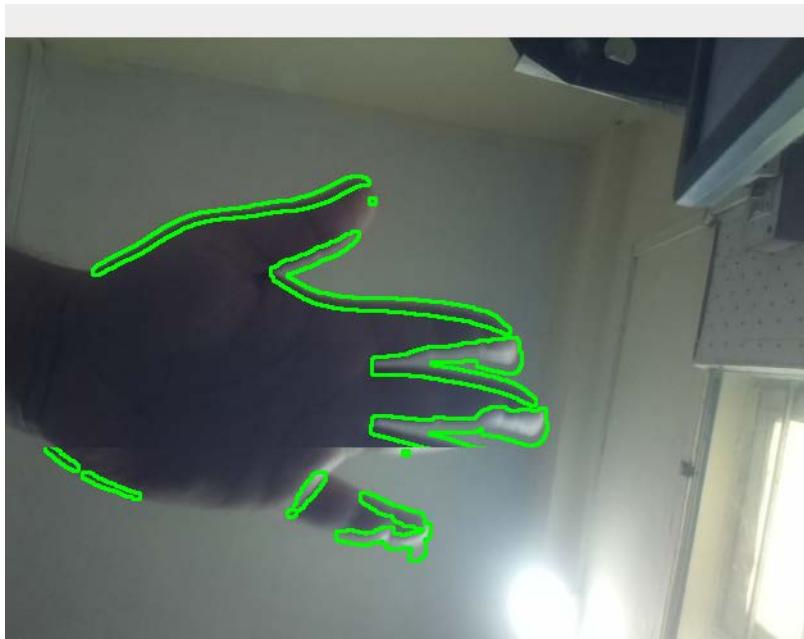


Figure 11.5 – Detected and highlighted movement

Keep in mind that this code is computationally expensive. Do not expect a very high framerate on the older and non-overclocked models of the RPi. As an exercise, draw contours of different colors. We can also compute the centroid with the help of the cv2.moments() function and represent those with small circles. This will make the output more interesting.

Detecting barcodes in images

A barcode is a way that information is represented visually and is easy to understand for purpose-made machines. There are many barcode formats. The usual format has parallel vertical lines of different thicknesses and different amounts of space in between them.

In this section, we will demonstrate how to detect a simple parallel-lines formatted barcode from a still image. We will use the following image of a soda can:



Figure 11.6 – The original source image

1. Let's read the source image of a soda can using the following code:

```
import numpy as np
import cv2
image=cv2.imread('/home/pi/book/dataset/barcode.jpeg', 1)
input = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

2. The horizontal image of a barcode has a low and a high vertical gradient. So, the candidate image must have the region that fits this criterion. We will use the `cv2.Sobel()` function to compute the horizontal and vertical derivatives and then compute the difference to find out the region that fits the criteria. Let's see how to do that:

```
hor_der = cv2.Sobel(input, ddepth=-1, dx=1, dy=0, ksize = 5)
ver_der = cv2.Sobel(input, ddepth=-1, dx=0, dy=1, ksize=5)
diff = cv2.subtract(hor_der, ver_der)
```

3. OpenCV provides the `cv2.convertScaleAbs()` function. It converts any numeric array into an array of **8-bit unsigned integers**. Let's use it, as follows:

```
diff = cv2.convertScaleAbs(diff)
```

The output is as follows:



Figure 11.7 – Difference between horizontal and vertical Sobel derivatives

4. The preceding output shows regions with very horizontal and very low vertical gradients. Let's apply a Gaussian blur to remove noise from the preceding output. Use the following code to do so:

```
blur = cv2.GaussianBlur(diff, (3, 3), 0)
```

The following is the output of the preceding code:



Figure 11.8 – After applying a Gaussian blur

5. Now, let's convert this image into a binary image by applying thresholding to it. The following is the code to do so:

```
ret, th = cv2.threshold(blur, 225, 255, cv2.THRESH_BINARY)
```

The following is the output binary image:

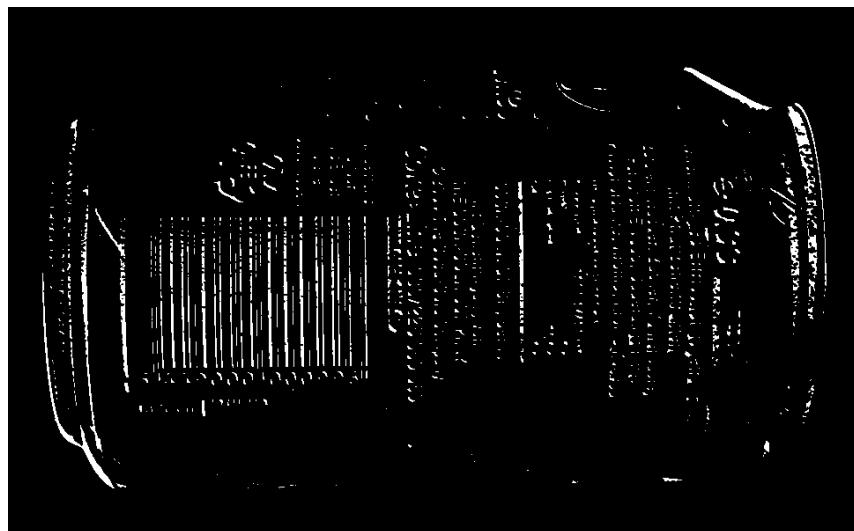


Figure 11.9 – Binary output

6. As shown in the preceding image, it is a binary image with the barcode and other high vertical gradient areas highlighted. We can dilate the image for further processing. It fills in the gap in between the vertical lines:

```
dilated = cv2.dilate(th, None, iterations = 10)
```

The output of the preceding code contains a lot of rectangle-shaped boxes corresponding to the barcode and other regions in the original image. We are interested in the region containing the barcode, not the other regions:



Figure 11.10 – Dilated binary output

7. The morphological erosion operation will eliminate most of the other regions that do not correspond to the barcode:

```
eroded = cv2.erode(dilated, None, iterations = 15)
```

The following is the output of the preceding code:



Fig.11.11 – Eroded image

8. Let's find the list of all the contours in this computed binary image. Use the following code to do so:

```
(contours, hierarchy) = cv2.findContours(eroded, cv2.  
RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

9. The biggest contour in this binary image is the contour corresponding to the region of the barcode. The following code finds the biggest contour in the image:

```
areas = [cv2.contourArea(temp) for temp in contours]  
max_index = np.argmax(areas)  
largest_contour = contours[max_index]
```

10. Let's retrieve the coordinates of the bounding rectangle of the biggest contour with the OpenCV `cv2.boundingRect()` function and then draw the rectangle in the image:

```
x, y, width, height = cv2.boundingRect(largest_contour)  
cv2.rectangle(image, (x, y), (x+width,  
y+height), (0,255,0), 2)  
cv2.imshow('Detected Barcode',image)  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

The preceding code draws the bounding rectangle over the area corresponding to the biggest contour in the image (which is the region of the barcode), as shown in the following output:



Figure 11.12 – Detected barcode

As shown in the previous screenshot, the approximate region with the barcode is outlined with a blue colored rectangle. This same code may not work with a lot of images, but it works with most images. We may have to tune the code to detect the regions with barcodes in the other images. You might want to change the following lines of code for the specific inputs:

```
blur = cv2.GaussianBlur(diff, (3, 3), 0)
dilated = cv2.dilate(th, None, iterations = 10)
eroded = cv2.erode(dilated, None, iterations = 15)
```

Based on this program, we can create many real-life applications. The first application is a barcode region detector for a live video feed from a USB webcam. The other application we can create is a generic program to detect barcodes. To tune the arguments that are passed to the functions, we can use trackbars.

In the next section, we will learn how to apply film-style chroma keying with OpenCV and Python 3 using the RPi and a USB webcam.

Implementing the chroma key effect

Chroma keying is also known as chroma key compositing. It is also colloquially known as the green screen or blue screen effect due to the green or blue background that we use while creating this effect. It is a post-production technique and can also be used on still images and live videos. In the chroma key effect, we place an object or a person in the foreground and capture an image or footage. The background is usually a green- or blue-colored fabric or wall. Then, we replace the green or blue color in the captured image or footage with another video or an image. This makes the viewers feel that the person or the object in the foreground is at a different location than the studio where they were filmed. This effect is one of the most used effects in film-making and live weather forecasts in news broadcasts:

1. Let's start by importing all the needed libraries and initiating a video capture object:

```
import numpy as np  
import cv2  
cap = cv2.VideoCapture(0)
```

2. To obtain the better frame rate or **frames per second (FPS)**, let's set the resolution of the USB webcam to 640x480 pixels. This will yield a better frame rate and the green screen effect will look natural:

```
cap.set(3, 640)  
cap.set(4, 480)
```

3. In the next step, we read the image that is to be used as the background. The image to be used as the background must have the same resolution as the resolution the webcam is set to. In this case, we have set it to 640x480. We know that all the arithmetic and logical operations that we will perform on NumPy arrays (in this case, the background image and the frames of the live feed from the USB webcam connected to the RPi) need the operand arrays to be of the same dimensions; otherwise, the Python 3 interpreter will throw an error. The following is the code for this:

```
bg = cv2.imread('/home/pi/book/dataset/bg.jpg', 1)
```

4. Let's write the familiar logic for the loop and read the frames of the live feed from the USB webcam, as follows:

```
while True:  
    ret, frame = cap.read()
```

We can use a green colored cloth (such as a curtain) or paper as the background for this demonstration. We will also use the box of the RPi camera module as the object in the foreground. The following is a photo of the original scene:



Figure 11.13 – Input video

5. As we discussed in *Chapter 6, Colorspaces, Transformations, and Thresholding*, when we need to work with color ranges, the HSV colorspace is the best way to represent colors. Let's convert the image into the HSV colorspace and then compute the mask for the green screen in the background with the following code:

```
hsv=cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
image_mask=cv2.inRange(hsv, np.array([40, 50, 50]),
np.array([80, 255, 255]))
```

This computes the mask for the image. The pixels in green (or any shade of it) are replaced with color and the remaining pixels are replaced with black:

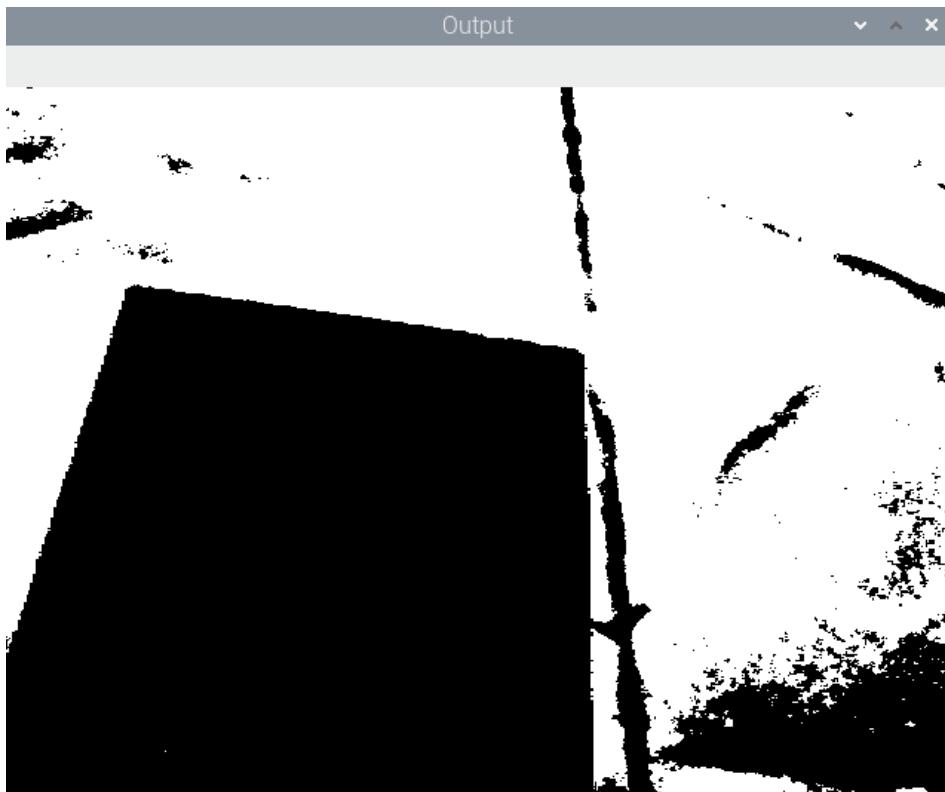


Figure 11.14 – Computing the mask

6. After computing the mask corresponding to the background image, we can apply the mask to the image in the background in order to hide the object in the foreground with the black pixels, as follows:

```
bg_mask=cv2.bitwise_and(bg, bg, mask=image_mask)
```

The preceding code replaces the white pixels with the pixels of the image we chose as the background. Also, the pixels corresponding to the foreground are in black, as shown in the following output:

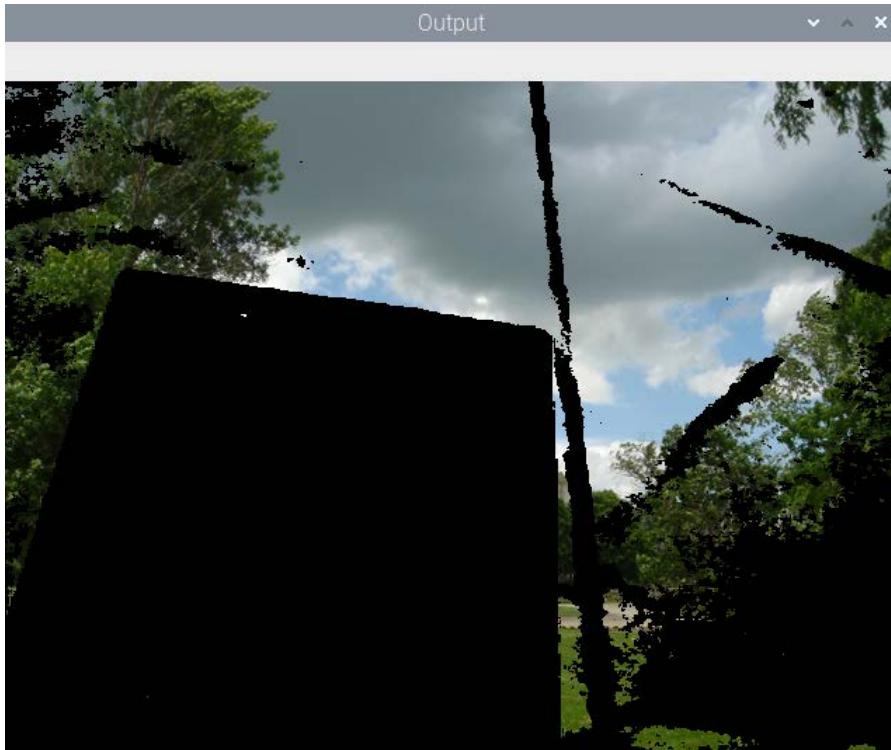


Figure 11.15 – White pixels in the mask replaced with the background

7. Now, we must extract the foreground from the live feed of the USB webcam. We can do this with the following code:

```
fg_mask=cv2.bitwise_and(frame, frame, mask=cv2.bitwise_
not(image_mask))
```

The preceding code extracts all the pixels that are not green (or shades of it). It also assigns the color black to the pixels that correspond to the background (the green screen):

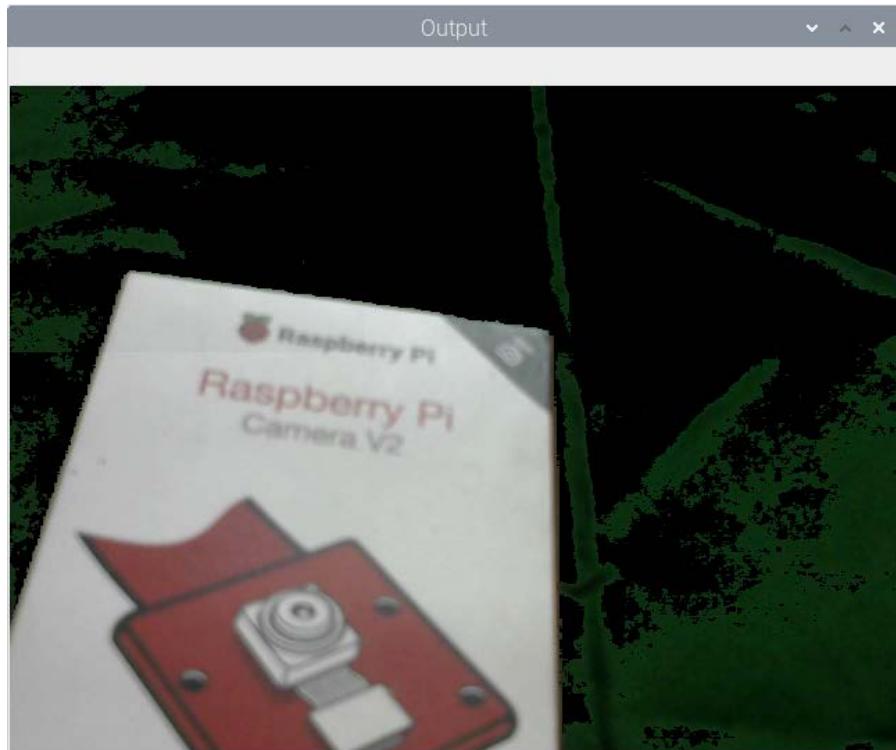


Figure 11.16 – Black pixels in the mask replaced with the foreground

8. Now, it is time for us to add the last two outputs we computed. This will replace the green background with the custom image and produce the chroma key effect:

```
cv2.imshow('Output', cv2.add(bg_mask, fg_mask))
if cv2.waitKey(1) == 27:
    break
cv2.destroyAllWindows()
cap.release()
```

The following is the final output:

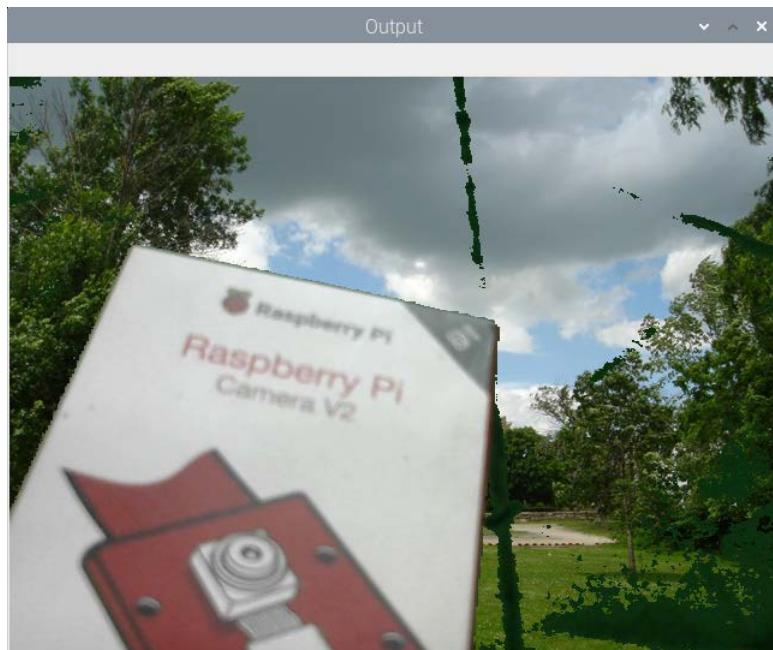


Figure 11.17 – Final output

Congratulations! We have managed to achieve a film-style chroma key effect with our RPi camera module box and a green cloth. We can see that the effect is imperfect. This is due to imperfect illumination. The webcam is not registering a few pixels as green pixels, but pixels of another color. The remedy to this is to have good and uniform illumination for the green background and foreground object.

The simple rule to be followed while implementing the chroma key effect is that the object we are chroma keying must not be the same color as the background screen. So, if we are using a green colored background, then neither the object nor any of its part can be green. The same is true for a blue background screen.

Summary

In this chapter, have learned how to demonstrate real-life applications using the concepts and techniques in computer vision we learned about in the previous chapters of this book. With the concepts we learned in this chapter, we can write a program for creating a simple security application.

From here on out, using the knowledge we've gained from the experiments in this book, we can explore the areas of image processing and computer vision with OpenCV in more detail. Our journey surrounding the OpenCV library concludes here.

In the next chapter, we will learn how to use another powerful, yet very easy to use, computer vision library for Python called Mahotas. We will also learn and demonstrate how to use Jupyter Notebook for scientific programming with Python 3.

12

Working with Mahotas and Jupyter

In the previous chapter, we learned about and demonstrated the use of real-life applications in the area of computer vision using Raspberry Pi with OpenCV and Python 3 programming.

In this chapter, we are going to learn the basics of another computer vision library—Mahotas. We are also going to have a look at a Jupyter project and understand how we can use the Jupyter Notebook for Python 3 programming. The topics we will learn in this chapter are as follows:

- Processing images with Mahotas
- Combining Mahotas and OpenCV
- Other popular image processing libraries
- Exploring the Jupyter Notebook for Python 3 programming

After following this chapter, you will be comfortable with using Mahotas for image processing. You will also be able to confidently run Python 3 programs with the Jupyter Notebook.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Chapter12/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/3fWRYDB>.

Processing images with Mahotas

Mahotas is a Python library for image processing and computer vision-related tasks. It was developed by Luis Pedro.

It implements many computer vision-related algorithms. It has been implemented in C++ and it operates on NumPy arrays. It also has a clean interface for Python 3.

Mahotas currently has over 100 functions for image processing and computer vision, and that number keeps growing with every release. This project is under active development and there is a new release every few months. Apart from the added functionality, every new release brings improvements in performance.

Note:

You can learn more about Mahotas by referring to <https://mahotas.readthedocs.io/en/latest/>.

We can install `mahotas` on Raspberry Pi with the following command:

```
pip3 install mahotas
```

A component of Mahotas will be installed in `/home/pi/.local/bin`. We can add this to the `PATH` variable permanently, as follows:

1. Open the `~/.profile` file in editing mode by running the following command:

```
nano ~/.profile
```

2. Add the following line to the end:

```
PATH='$PATH:/home/pi/.local/bin'
```

3. Reboot Raspberry Pi:

```
sudo reboot
```

We can verify whether `mahotas` has been installed successfully by running the following command on Command Prompt:

```
python3 -c 'import mahotas'
```

If this command doesn't return an error, then the installation is successful. Now, let's look at creating some programs with Mahotas.

Reading images and built-in images

Mahotas has many built-in images. Let's see how to use them. Look at the following code:

```
import matplotlib.pyplot as plt
import mahotas
photo = mahotas.demos.load('luispedro')
plt.imshow(photo)
plt.axis('off')
plt.show()
```

In the preceding code, `mahotas.demos.load()` is used for loading built-in images to a NumPy array. `luispedro` is an image of the author of the library. Unlike OpenCV, Mahotas reads and stores color images in RGB format. We can also load and display the image in grayscale mode, as follows:

```
photo = mahotas.demos.load('luispedro', as_grey=True)
plt.imshow(photo, cmap='gray')
```

We can load other library images, as follows:

```
photo = mahotas.demos.load('nuclear')
photo = mahotas.demos.load('lena')
photo = mahotas.demos.load('DepartmentStore')
```

We can also read the images stored on a disk, as follows:

```
photo= mahotas.imread('/home/pi/book/dataset/4.1.01.tiff')
```

This function works in the same way as the `cv2.imread()` OpenCV function.

Thresholding images

We already know the basics of thresholding. We can threshold a grayscale image by using a couple of functions available in `mahotas`. Let's demonstrate Otsu's binarization:

```
import matplotlib.pyplot as plt
import numpy as np
import mahotas
photo = mahotas.demos.load('luispedro', as_grey=True)
```

```
photo = photo.astype(np.uint8)
T_otsu = mahotas.otsu(photo)
plt.imshow(photo > T_otsu, cmap='gray')
plt.axis('off')
plt.show()
```

The `mahotas.otsu()` function accepts a grayscale image as an argument and returns the value of the threshold. The `photo > T_otsu` code returns the thresholded image. The following is the output:



Figure 12.1 – Otsu's binarization

We can perform thresholding with the Riddler-Calvard method, too, as follows:

```
T_rc = mahotas.rc(photo)
plt.imshow(photo > T_rc, cmap='gray')
```

The `mahotas.rc()` function accepts a grayscale image as an argument and returns the value of the threshold. The `photo > T_rc` code returns the thresholded image. Run this and check the output. It will show us a thresholded image with the Riddler-Calvard method.

The distance transform

The distance transform is a morphological operation. It is best visualized with binary (0 and 1) images. It transforms binary images into grayscale images in such a way that the grayscale intensity of a point visualizes its distance from the boundary in the image. The `mahotas.distance()` function accepts an image and computes the distance transform. Let's look at an example:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
import mahotas

f = np.ones((256, 256), bool)
f[64:191, 64:191] = False
plt.subplot(121)
plt.imshow(f, cmap='gray')
plt.title('Original Image')

dmap = mahotas.distance(f)
plt.subplot(122)
plt.imshow(dmap, cmap='gray')
plt.title('Distance Transform')
plt.show()
```

This creates a custom image of a square filled with the color black against a white background. Then, it computes the distance transform and visualizes it. This produces the following output:

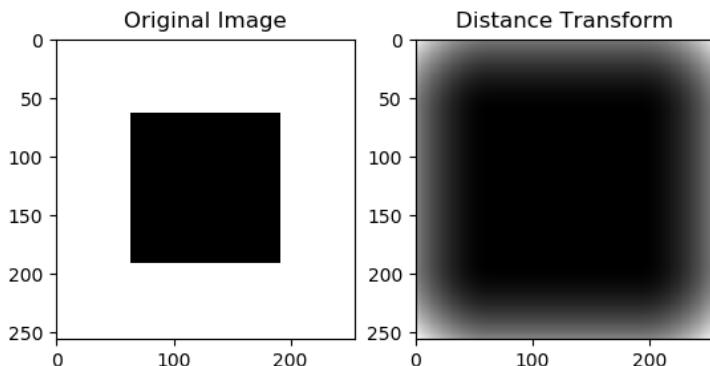


Figure 12.2 – Distance transform demonstration

Colorspace

We can convert an RGB image into sepia, as follows:

```
import matplotlib.pyplot as plt
import mahotas

photo = mahotas.demos.load('luispedro')
```

```
photo = mahotas.colors.rgb2sepia(photo)
plt.imshow(photo)
plt.axis('off')
plt.show()
```

The preceding code reads a grayscale image from the library and converts it into an image with sepia colorspace using the call of the `rgb2sepia()` function. It accepts an image as an argument and returns the converted image. The following is the output of the previous program:

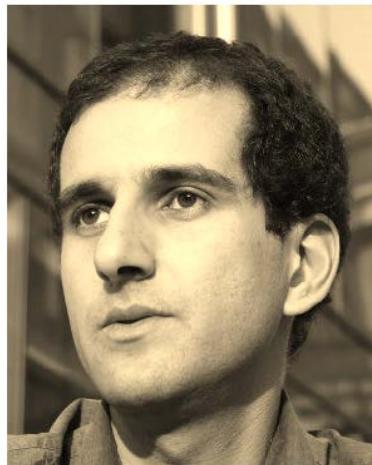


Figure 12.3 – A sepia image

In the next section, we will learn how we can combine the code for Mahotas and OpenCV.

Combining Mahotas and OpenCV

Just like OpenCV, Mahotas uses NumPy arrays to store and process images. We can also combine OpenCV and Mahotas. Let's see an example of this, as follows:

```
import cv2
import numpy as np
import mahotas as mh

cap = cv2.VideoCapture(0)

while True:
```

```
ret, frame = cap.read()
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
T_otsu = mh.otsu(frame)
output = frame > T_otsu
output = output.astype(np.uint8) * 255
cv2.imshow('Output', output)
if cv2.waitKey(1) == 27:
    break

cv2.destroyAllWindows()
cap.release()
```

In the preceding program, we converted a live frame into a grayscale version. Then, we applied a Mahotas implementation of Otsu's binarization, which converted the frame from the live video feed into a Boolean binary image. We need to convert this to the np.uint8 type and multiply it by 255 (all of which takes the form of ones in binary 8-bit) so that we can use it with cv2.imshow(). The output is as follows:

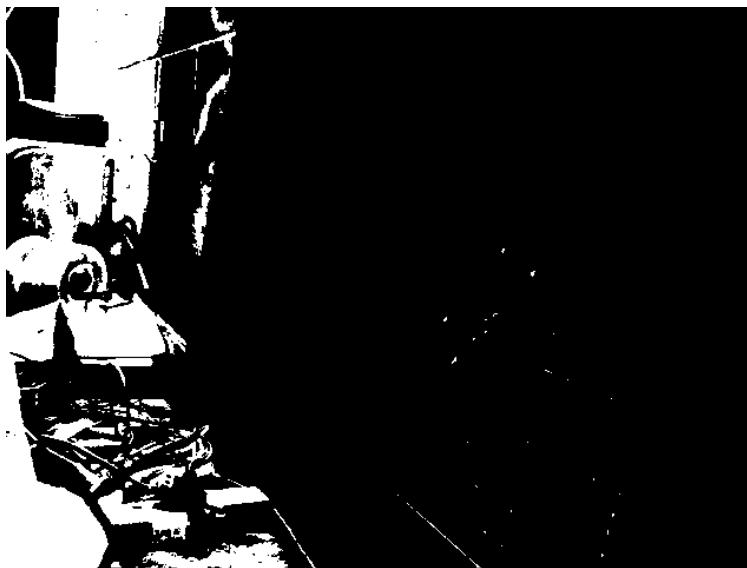


Figure 12.4 – A screenshot of the output window

We usually use the OpenCV functionality to read the live feed from the USB webcam. Then, we can use the functions from Mahotas, or any other image processing library, to process the frame. This way, we can combine the code from two different image processing libraries.

In the next section, we will learn the names and URLs of a few other Python image processing libraries.

Other popular image processing libraries

Python 3 has many third-party libraries. Many of these libraries use NumPy for processing images. Let's have a look at a list of the available libraries:

- skimage (<https://scikit-image.org/>)
- SimpleITK (<http://www.simpleitk.org/>)
- scipy.ndimage (<https://docs.scipy.org/doc/scipy/reference/ndimage.html>)

These are all NumPy-based image processing libraries. The Python imaging library and its well-maintained fork, pillow (<https://pillow.readthedocs.io/en/stable/>), are non-NumPy-based image manipulation libraries. They also have interfaces to convert images between the NumPy and PIL image formats.

We can combine code that uses various libraries to create various computer vision applications with the desired functionality.

In the next section, we will explore the Jupyter Notebook.

Exploring the Jupyter Notebook for Python 3 programming

The Jupyter Notebook is a web-based interactive interface that works like the interactive mode of Python 3. The Jupyter Notebook has 40 programming languages, including Python 3, R, Scala, and Julia. It provides an interactive environment for programming that can have visualizations, rich text, code, and other components, too.

Jupyter is a fork of the IPython project. All the language-agnostic parts of IPython were moved to Jupyter and the Python-related functionality of Jupyter is provided by an IPython kernel. Let's see how we can install Jupyter on Raspberry Pi:

1. Run the following commands one by one in Command Prompt:

```
sudo pip3 uninstall ipykernel
```

The previous command uninstalls the earlier version of the `ipykernel` utility.

2. The following commands install all the required libraries:

```
sudo pip3 install ipykernel==4.8.0
sudo pip3 install jupyter
sudo pip3 install prompt-toolkit==2.0.5
```

These commands will install Jupyter and the required components on Raspberry Pi. To launch the Jupyter Notebook, log in to Raspberry Pi's graphical environment (directly or using Remote Desktop) and run the following command in lxterminal:

```
jupyter notebook
```

This will launch the Jupyter Notebook server process and open a web browser window with a Jupyter Notebook interface, as follows:

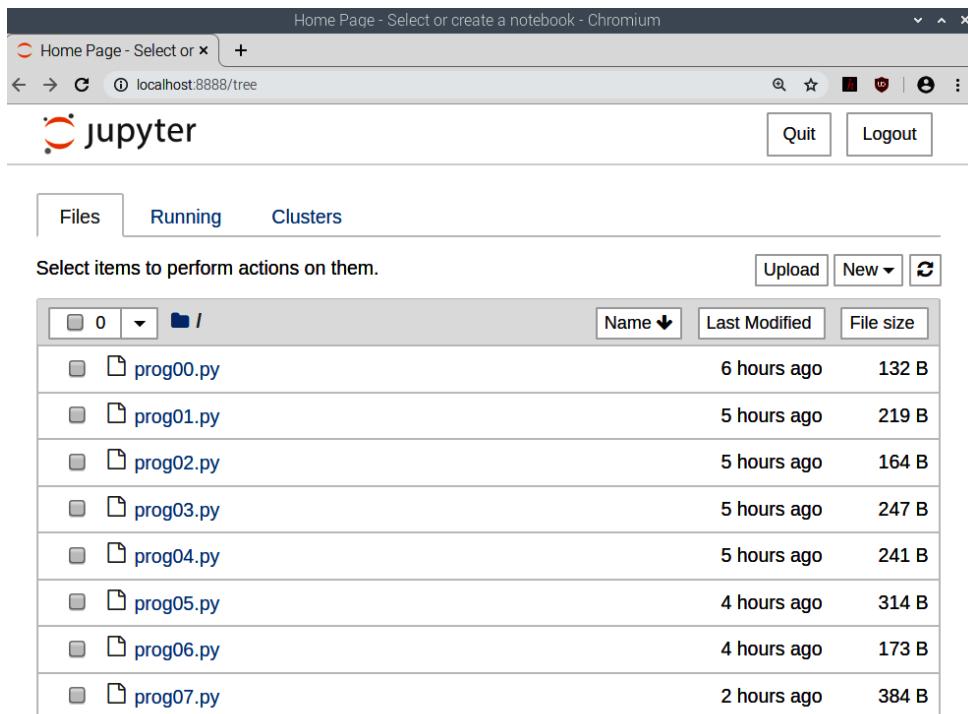
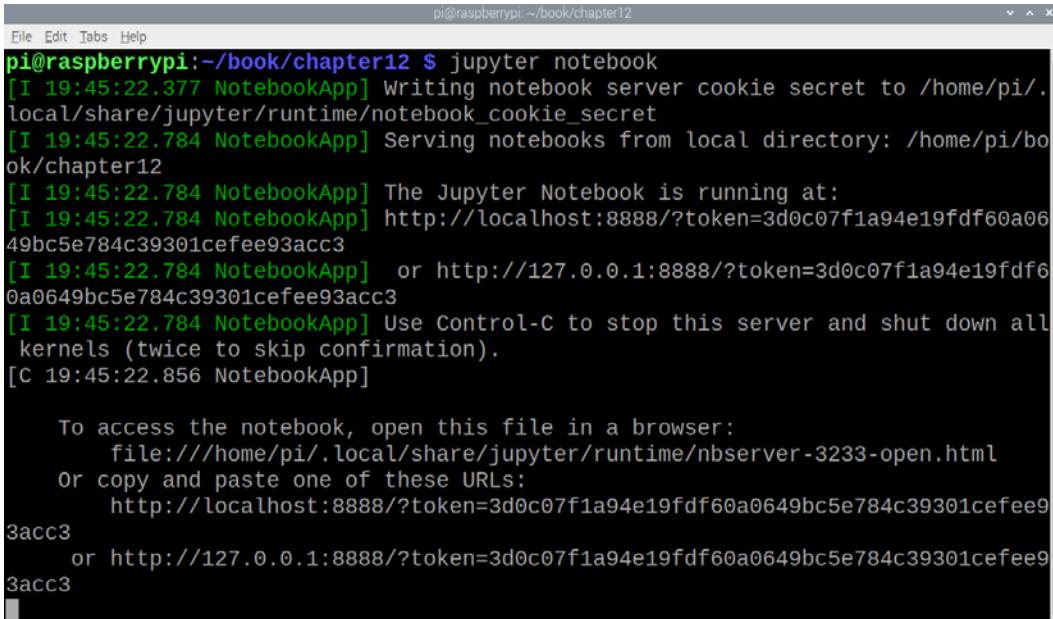


Figure 12.5 – The startup directory

The previous screenshot shows us the directory structure of the directory where we ran the command to launch. I ran it in the directory for the code for our current chapter, /home/pi/book/chapter12. The LXTerminal window where we ran the command shows the server log, as follows:



A screenshot of an LXTerminal window titled "pi@raspberrypi: ~book/chapter12". The window displays the output of the command "jupyter notebook". The log shows the server writing a cookie secret to a local file and serving notebooks from the specified directory. It provides two URLs for accessing the notebook: one via the local host IP (127.0.0.1) and another via the public IP (3d0c07f1a94e19fdf60a0649bc5e784c39301cefee9). It also indicates how to stop the server using Control-C.

```
pi@raspberrypi:~/book/chapter12 $ jupyter notebook
[I 19:45:22.377 NotebookApp] Writing notebook server cookie secret to /home/pi/.local/share/jupyter/runtime/notebook_cookie_secret
[I 19:45:22.784 NotebookApp] Serving notebooks from local directory: /home/pi/book/chapter12
[I 19:45:22.784 NotebookApp] The Jupyter Notebook is running at:
[I 19:45:22.784 NotebookApp] http://localhost:8888/?token=3d0c07f1a94e19fdf60a0649bc5e784c39301cefee93acc3
[I 19:45:22.784 NotebookApp] or http://127.0.0.1:8888/?token=3d0c07f1a94e19fdf60a0649bc5e784c39301cefee9
[I 19:45:22.784 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 19:45:22.856 NotebookApp]

To access the notebook, open this file in a browser:
  file:///home/pi/.local/share/jupyter/runtime/nbserver-3233-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=3d0c07f1a94e19fdf60a0649bc5e784c39301cefee9
3acc3
  or http://127.0.0.1:8888/?token=3d0c07f1a94e19fdf60a0649bc5e784c39301cefee9
3acc3
```

Figure 12.6 – The Jupyter Notebook server log

Now, let's get back to the browser window that is running Jupyter. In the top-right corner of the browser window, we have options to log out and quit. Below that, we can see the **Upload** button, the **New** drop-down menu, and the refresh symbol.

On the right-hand side, we can see three tabs. The first one, as we already saw, shows the directory structure from where we launched Jupyter in Command Prompt. The second tab shows the currently running processes.

Let's explore the **New** drop-down option on the right-hand side. The following screenshot shows the options available under this menu:

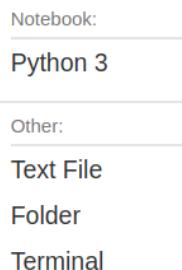


Figure 12.7 – The New menu dropdown

We can see an option for **Python 3** under the **Notebook** section. If you have any other programming languages that use Jupyter, then those languages will also be shown here. We will explore that shortly. The other options are **Text File**, **Folder**, and **Terminal**. The first two options under **Other** create a blank file and a blank directory, respectively. **Terminal**, when clicked, launches LXTerminal in a new tab of the browser window, shown in the following screenshot:

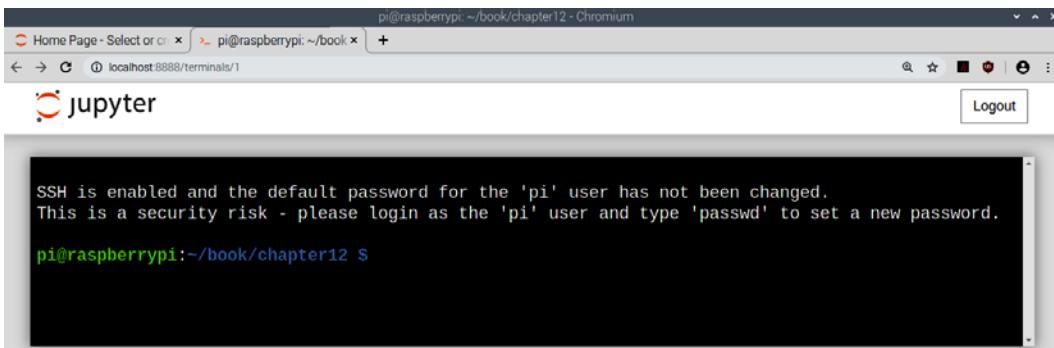


Figure 12.8 – Command Prompt running in a web browser tab

If we click on the original tab (listed as **Home Page** in the browser tabs) and check under the **Running** option, we can see an entry corresponding to the current terminal window tab, as shown in the following screenshot:

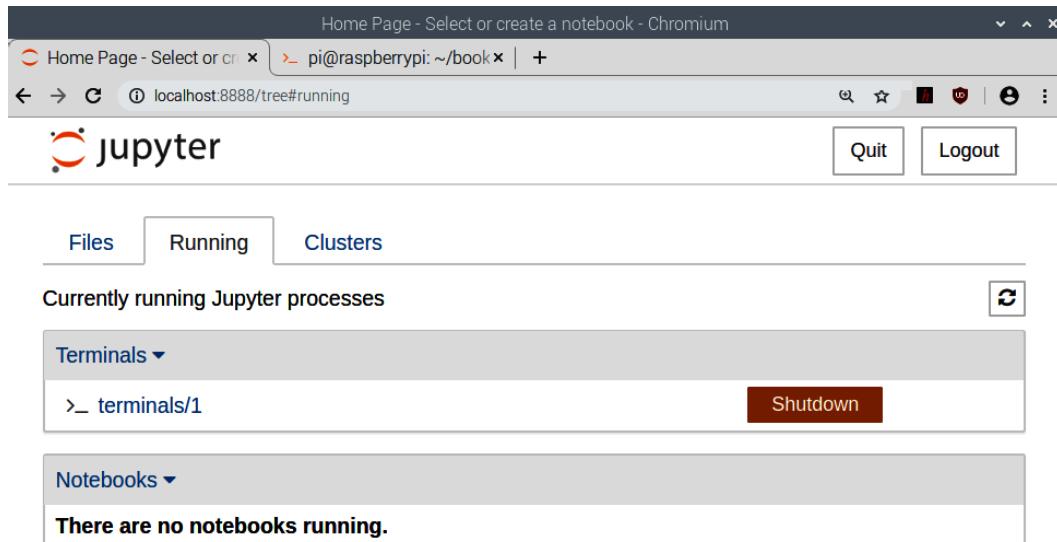


Figure 12.9 – A list of running subprocesses in Jupyter

As we can see, there are options to see the current notebooks and terminals launched under this server. We can shut them down from here. Go to **Files** and under the **New** dropdown, select **Python 3**. This will open a Python 3 notebook under a new tab in the same browser window:

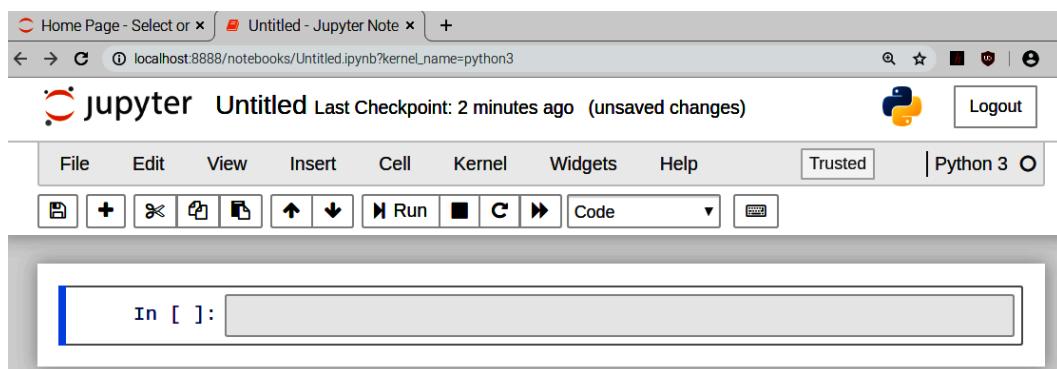


Figure 12.10 – A new Jupyter Notebook tab

We can see that the name of the notebook is Untitled. We can click on the name and it will open a modal dialog box to rename the notebook, as follows:

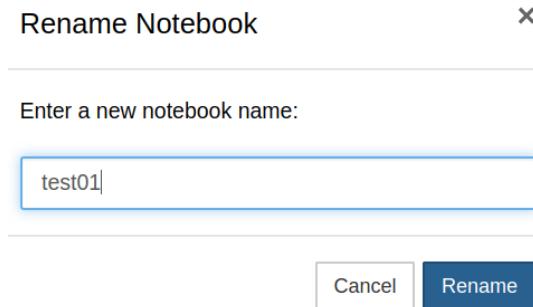


Figure 12.11 Renaming a notebook

Rename the notebook. After that, in the main **Home Page** tab, under **Files**, we can see the `test01.ipynb` file. Here, `ipynb` means an IPython notebook. You can see an entry in the **Running** tab, too. In the `/home/pi/book/chapter12` directory, we can find the `test01.ipynb` file. Now, let's see how we can use this file for Python 3 programming. Switch to the `test01` notebook tab in the browser again. Let's explore the interface in detail:

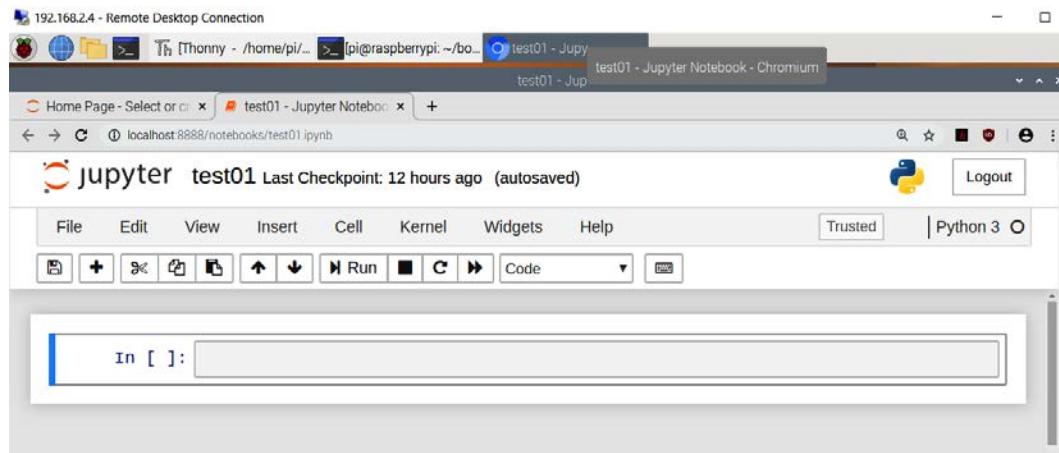


Figure 12.12 – A Jupyter notebook

We can see a long text area after the `In [] :` text. We can write code snippets here. Make sure that **Code** is selected from the dropdown of the menu. Then, add the following code to the text area:

```
print('Hello World')
```

We can run it by clicking on the **Run** button in the menu bar. The output will be printed here and a new text area will appear. The cursor will automatically set there:

```
In [1]: print('Hello World!')
```

Hello World!

```
In [ ]:
```

Figure 12.13 – Running the Hello World! program

The best thing about this notebook is that we can edit and re-execute the earlier cells. Let's try to understand the icons in the menu:



Figure 12.14: The icon buttons in the menu

Let's go from left to right. The first symbol (the floppy disk) is to save. The + symbol adds a text area cell after the currently highlighted cell. Then, we have the cut, copy, and paste options. The up and down arrows are used to shift the current text area cell up and down. Then, we have **Run** and the interrupt the kernel, restart the kernel, and restart and run the whole notebook buttons. The drop-down box decides the type of the cell. It has the following four options:

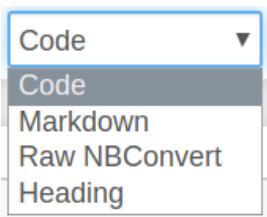


Figure 12.15 – Types of cells

If we want the cell to run the code, we choose **Code**. **Markdown** is a markup language used for rich text. Select an empty text area cell and change it to the **Markdown** type. Then, enter # Test into the cell and execute it. This will create a level-one heading, as follows:

Test

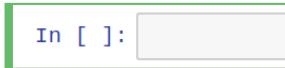


Figure 12.16 – A level 1 heading

We can use ## for level-two headings, ### for level-three headings, and so on.

One of the major features of the Jupyter Notebook is that we can even run the OS commands in a notebook. We need to precede the commands with the ! symbol and run them in the cell as **Code**. Let's look at a demonstration of this. Run the !ls -la command in the notebook and it should produce the following result:

```
In [4]: !ls -la
total 48
drwxr-xr-x  3 pi pi 4096 Feb 21 10:37 .
drwxr-xr-x 15 pi pi 4096 Feb 19 11:08 ..
drwxr-xr-x  2 pi pi 4096 Feb 20 21:36 .ipynb_checkpoints
-rw-r--r--  1 pi pi 132 Feb 20 14:17 prog00.py
-rw-r--r--  1 pi pi 219 Feb 20 14:28 prog01.py
-rw-r--r--  1 pi pi 164 Feb 20 14:38 prog02.py
-rw-r--r--  1 pi pi 247 Feb 20 15:07 prog03.py
-rw-r--r--  1 pi pi 241 Feb 20 15:08 prog04.py
-rw-r--r--  1 pi pi 314 Feb 20 15:40 prog05.py
-rw-r--r--  1 pi pi 173 Feb 20 16:12 prog06.py
-rw-r--r--  1 pi pi 384 Feb 20 18:16 prog07.py
-rw-r--r--  1 pi pi 1082 Feb 21 10:37 test01.ipynb
```

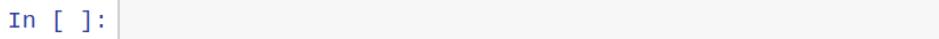


Figure 12.17 – Running OS commands in the Jupyter notebook

We can also show visualizations and images with matplotlib in the notebook. For that, we need to use the magic %matplotlib function. We can use this to set the backend of matplotlib to the inline backend of the Jupyter notebook, as follows:

```
%matplotlib inline
```

Let's see a short demonstration of this:

```
In [1]: %matplotlib inline
import cv2
import matplotlib.pyplot as plt

In [2]: img = cv2.imread('/home/pi/book/dataset/7.1.02.tiff', 0)
plt.imshow(img, cmap='gray')
plt.show()
```

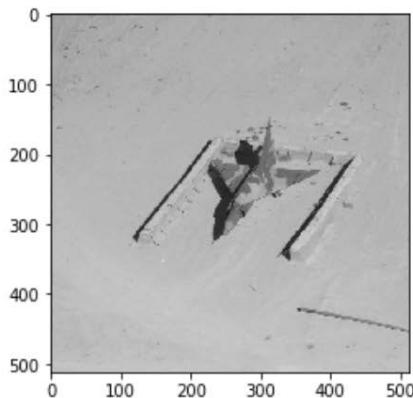


Figure 12.18 – Showing images in the Jupyter notebook

This is how we can show visualizations and images in the notebook itself.

This is a very useful concept. We can have rich text, OS commands, Python code, and output (including visualizations) in a single notebook. We can even share these ipynb notebook files electronically. Just like Python 3, we can use the Jupyter Notebook with a lot of languages, such as Julia, R, and Scala. The only limitation is that we cannot mix code of multiple programming languages in a single notebook.

The last thing I want to explain is how to clear the output. Click on the **Cell** menu. It looks as follows:

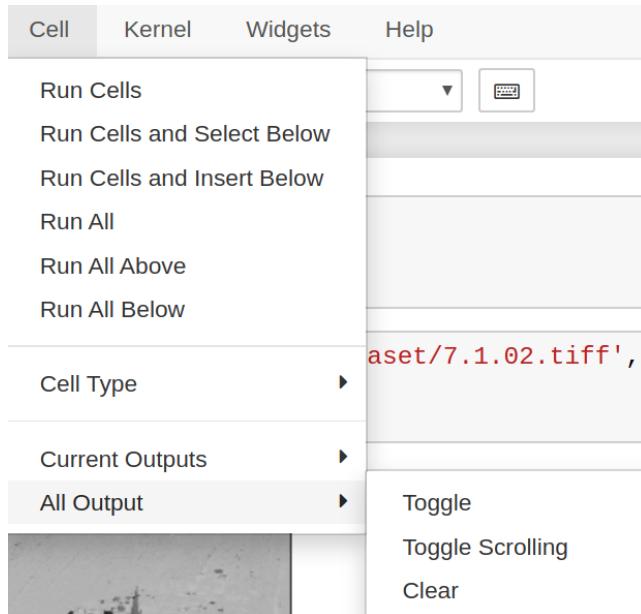


Figure 12.19 – Clearing all the output

We have the **Clear** option under **Current Outputs** and **All Output**. These clear the output of the current cell and the entire notebook, respectively.

I recommend exploring all the options in the menu bar yourself.

Summary

In this chapter, we explored the basics of Mahotas, which is a NumPy-based image processing library. We looked at a few image processing-related functions and learned how to combine Mahotas and OpenCV code for image processing. We also learned the names of other NumPy- and non-NumPy-based image processing libraries. You can explore those libraries further.

The last topic we learned about, the Jupyter Notebook, is very handy for quickly prototyping ideas and sharing code electronically. Many computer vision and data science professionals now use Jupyter notebooks for their Python programming projects.

In the *Appendix* section of this book, I have explained all the topics that I could not list under this chapter. These topics will be immensely useful to anyone who uses Raspberry Pi for various purposes.

13

Appendix

All the topics that could not be covered in this book's main chapters are instead covered here. This appendix is largely a collection of useful topics, including tips and tricks. So, let's look at a few tips and tricks relating to Raspberry Pi, Python 3, and OpenCV.

Technical requirements

The code files of this chapter can be found on GitHub at <https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming/tree/master/Appendix/programs>.

Check out the following video to see the Code in Action at <https://bit.ly/3ewQwHs>.

Performance measurement and the management of OpenCV

OpenCV has a lot of optimized and unoptimized code. The optimized code uses features of modern microprocessors, such as instruction pipelining and AVX.

We can check whether the optimization of OpenCV is enabled on the computer we are currently using with the `cv2.useOptimized()` function. We can also use the `cv2.setUseOptimized()` function to toggle the optimization. The `cv2.getTickCount()` function returns the number of clock ticks (also known as **clock cycles**) from the time that the computer was turned on. This function is called before and after the execution of the code snippet that we are interested in.

Then, we compute the difference between the clock cycles and it returns the number of clock cycles needed to execute the code snippet. The `cv2.getTickFrequency()` function returns the frequency of the clock cycles. Then, we can divide the difference between the clock cycles by the frequency of the clock cycles to obtain the time required for the execution of the code snippet:

```
import cv2
cv2.setUseOptimized(True)
print(cv2.useOptimized())
img = cv2.imread('/home/pi/book/dataset/4.1.01.tiff', 0)
e1 = cv2.getTickCount()
img1 = cv2.medianBlur(img, 23)
e2 = cv2.getTickCount()
t = (e2 - e1)/cv2.getTickFrequency()
print(t)
```

The output of the preceding code is as follows:

```
True
0.004361807
```

We can also use the functions in the `time` Python 3 library to establish the amount of time required to run any code snippet. Try this out as an exercise. Next, we will see how to reuse a Raspbian OS microSD card.

Reusing a Raspbian OS microSD card

We have learned to write the Raspbian OS to a microSD card using **Win32 Disk Imager**. Now, we are going to see how to reuse that microSD card for any other purpose. Insert the microSD card into the microSD card reader and connect it to a Windows PC. It will show two partitions. Only one of these is readable and it will be labeled `boot`. It should also have the `config.txt` file, which has a size of around 250 MB. The other partition is unreadable. We cannot use this microSD card as it is used for another purpose. So, we need to use a few tools to format this card before we can reuse it again for any other purpose.

Formatting the SD card using the SD card formatter

There is a free tool for formatting SD cards. We can download it from <https://www.sdcard.org/downloads/formatter/>. Install this tool and open it, and it will show the following window. The drive letters could be different depending on the number of drives on your computer. The following is a screenshot of the application:

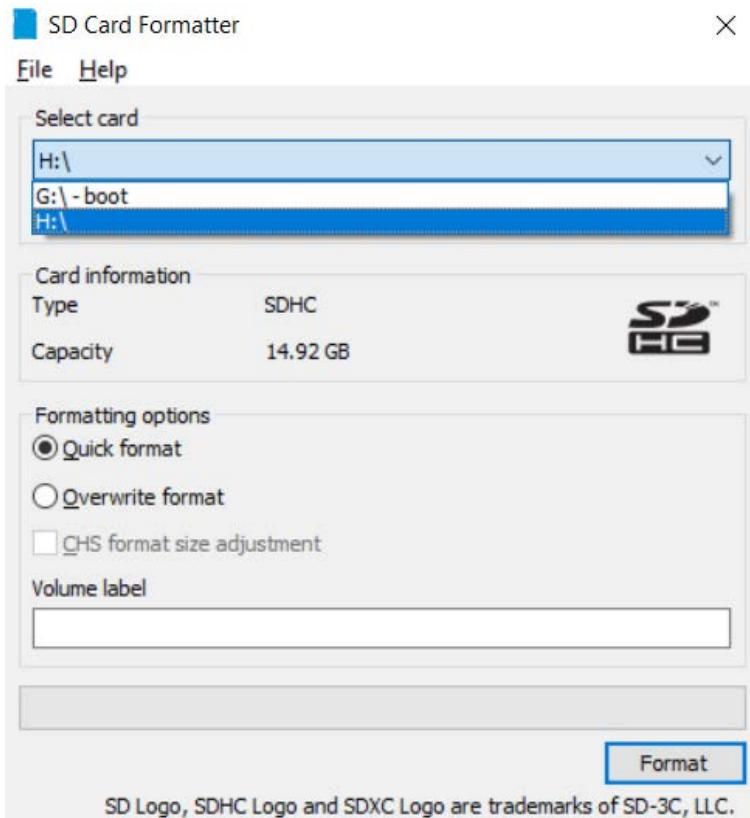


Figure 13.1 – The SD card formatter

Choose any drive (it will format the entire card anyway) and click on the **Format** button. This will show the following confirmation box:

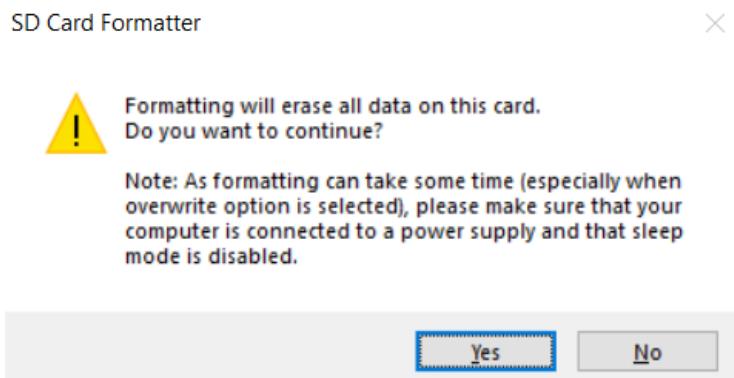


Figure 13.2 – Confirmation dialogue

Click the **Yes** button and it will format the card. After formatting, there will be only one drive letter corresponding to the card. The card is completely formatted now and we can use it as if it is fresh.

The Disk Management utility in Windows

We can even use the **Disk Management** utility in Windows to format the microSD card. In the search bar, type **Disk** and you will find the **Create and format Disk Partitions** option. You can also find this utility from the Windows Control Panel, too. Again, insert the Raspbian OS microSD card that you want to reuse into an SD card reader and connect it to a Windows computer. Open the **Disk Management** tool and you will see something as in the following screenshot:

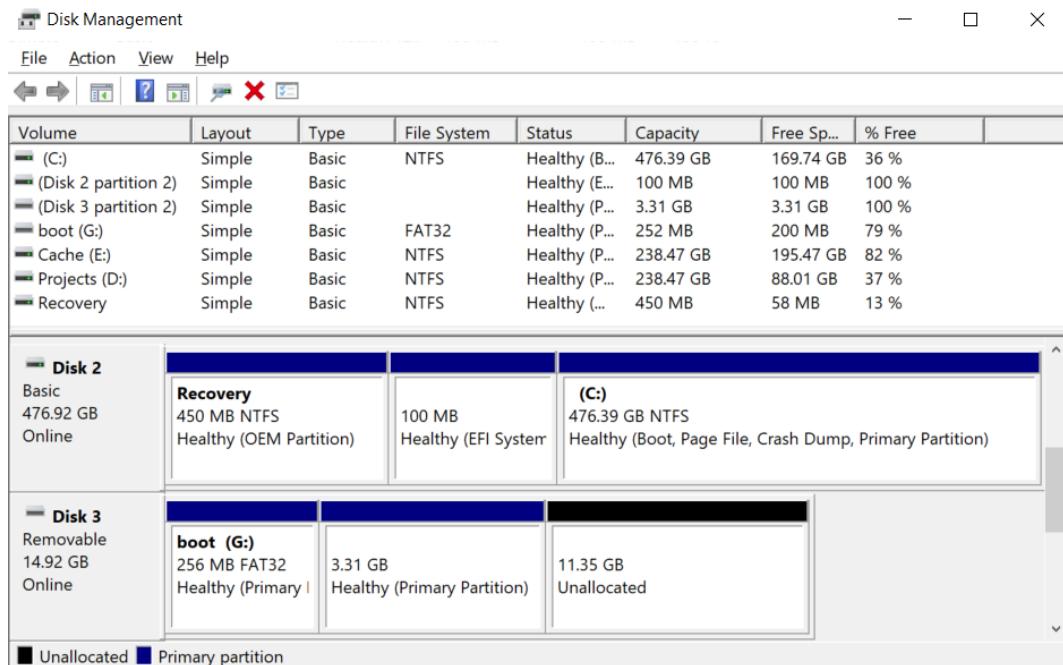


Figure 13.3 – The Disk Management utility window

This lists all the disks (removable and non-removable) attached to the system. Out of these, the one that is removable (with a boot partition of 256 MB) is the microSD card. As you can see in the preceding screenshot, I inserted the Raspbian OS microSD card without expanding the filesystem (I mean, I wrote the Raspbian OS to it, but did not use it to boot up the Raspberry Pi board). That is why it shows two allocated partitions and one unallocated partition. If you have used the card to boot up a Raspberry Pi board, it expands the filesystem and the second biggest partition occupies the unallocated part. So, used Raspbian OS microSD cards show two partitions only. Anyway, we can just right-click on an allocated partition and choose the **Delete Volume...** option. Do this for both of the allocated partitions:

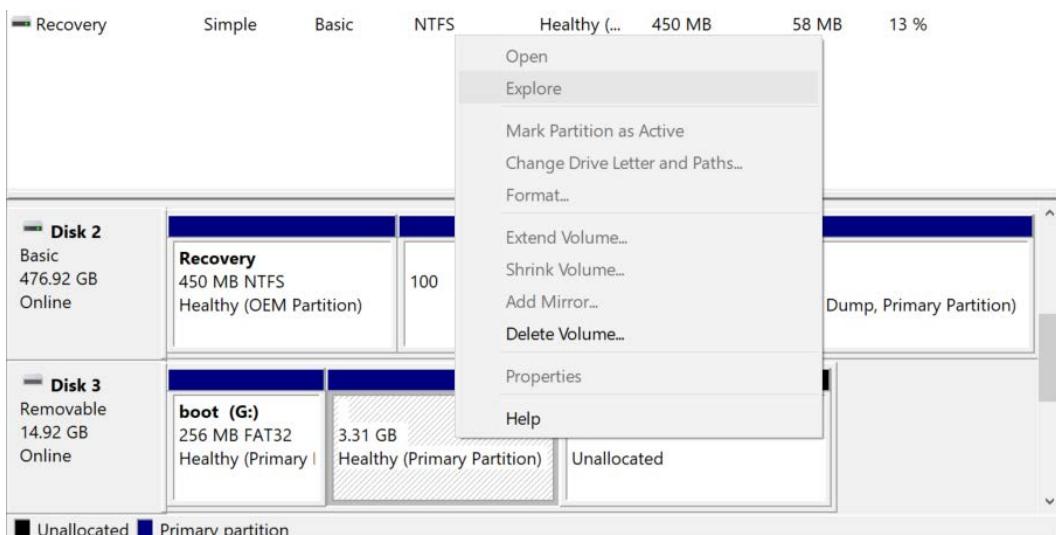


Figure 13.4 – Deleting partitions of the SD card

The disk will look as follows after deleting all the allocated parts:

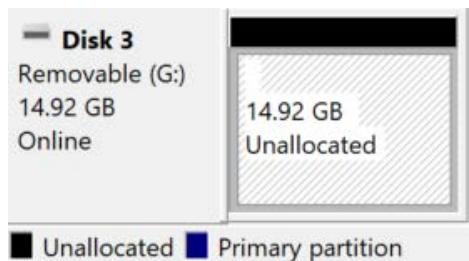


Figure 13.5 – Creating a new partition on the SD card

Just right-click on the disk corresponding to the microSD card and click on **New Simple Volume**. This will launch a wizard to a new volume. Complete the guided wizard with all the default options and you will get a fresh disk for reuse. You can rewrite the Raspbian OS to this or use it to store your favorite MP3 songs. This **Disk Management** tool gives us better control over the finer aspects of disk formatting and partitioning.

Tour of the raspi-config command-line utility

We can configure Raspberry Pi by using one of the following three methods:

- The Raspberry Pi configuration tool in the Raspbian OS menu
- By altering the content of /boot/config.txt
- With the `raspi-config` command-line utility

We will provide a detailed tour of the `raspi-config` tool in detail in this section. Open the Raspberry Pi command prompt and run the following command:

```
sudo raspi-config
```

This will open the Raspberry Pi configuration tool in Command Prompt, as in the following screenshot:

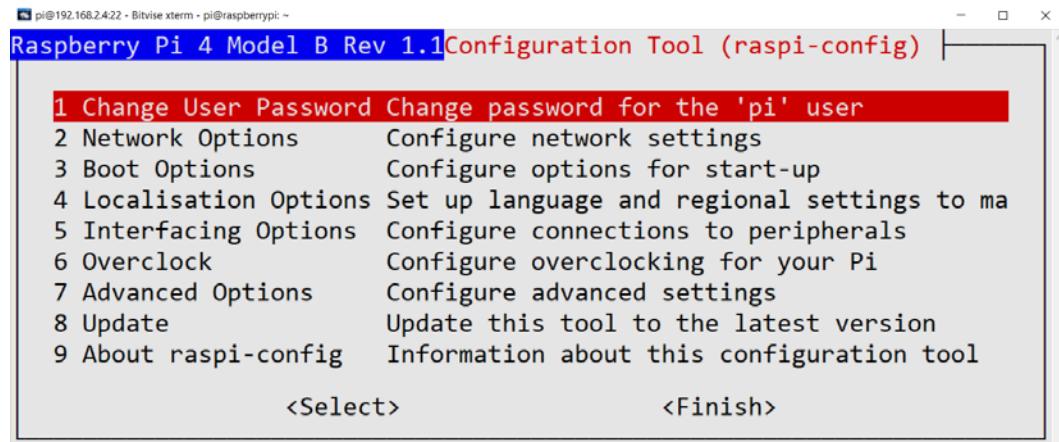


Figure 13.6 – The main menu of the `raspi-config` utility

The first option is used to change the password for the pi user. The second option in the main menu, **Network Options**, has the facility to change the way the Raspberry Pi board is connected to the network:

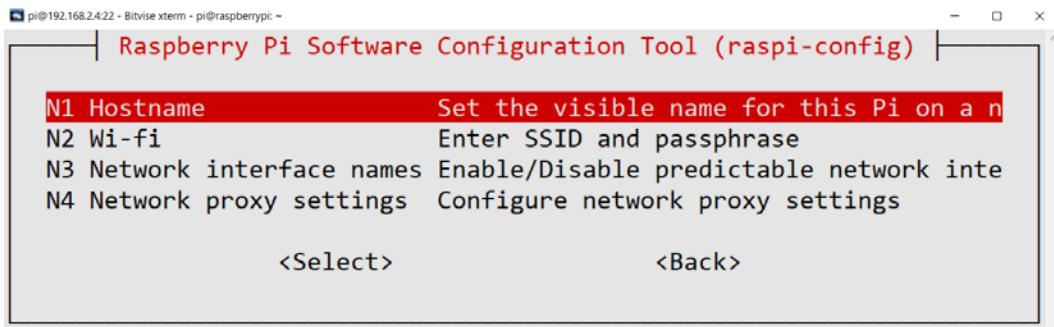


Figure 13.7 – Network Options

The third option in the main menu (**Boot Options**) details the booting options, as follows:

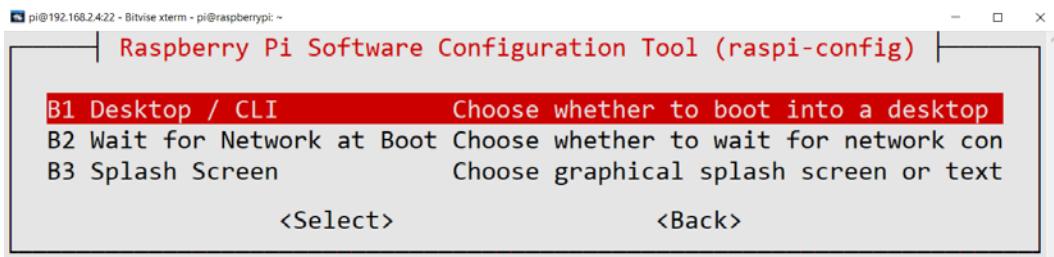


Figure 13.8 – Boot Options

The fourth option in the main menu (**Localization Options**) allows you to set the locale, time zone, keyboard layout, and Wi-Fi country, as follows:

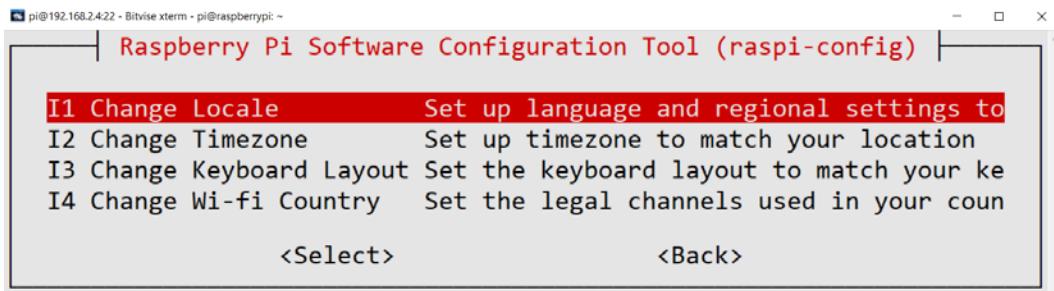


Figure 13.9 – Localization Options

The fifth option in the main menu is **Interfacing Options**, which appears as follows:

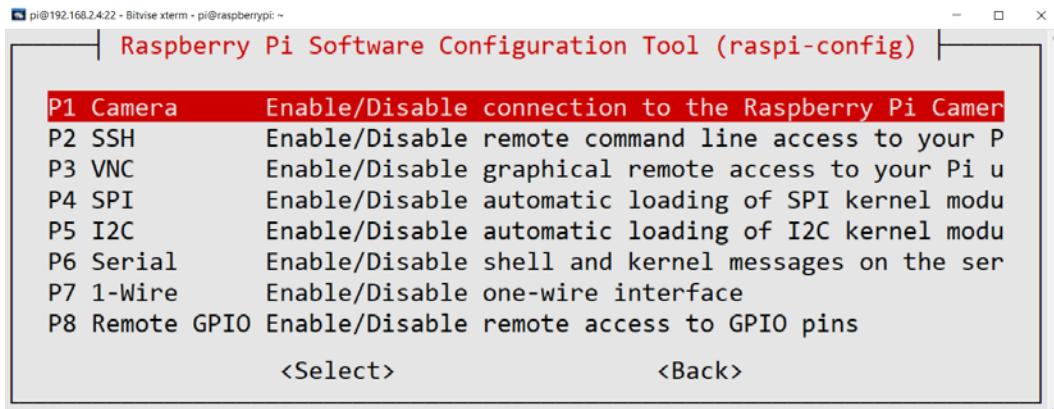


Figure 13.10 – Interfacing Options

Out of the preceding options, we have already enabled **P1 Camera**, **P2 SSH**, and **P3 VNC** for our demonstrations.

The sixth option in the main menu is for overclocking Raspberry Pi 1 and Raspberry Pi 2. Other models have to be overclocked manually.

The seventh option in the main menu is **Advanced Options**, as follows:

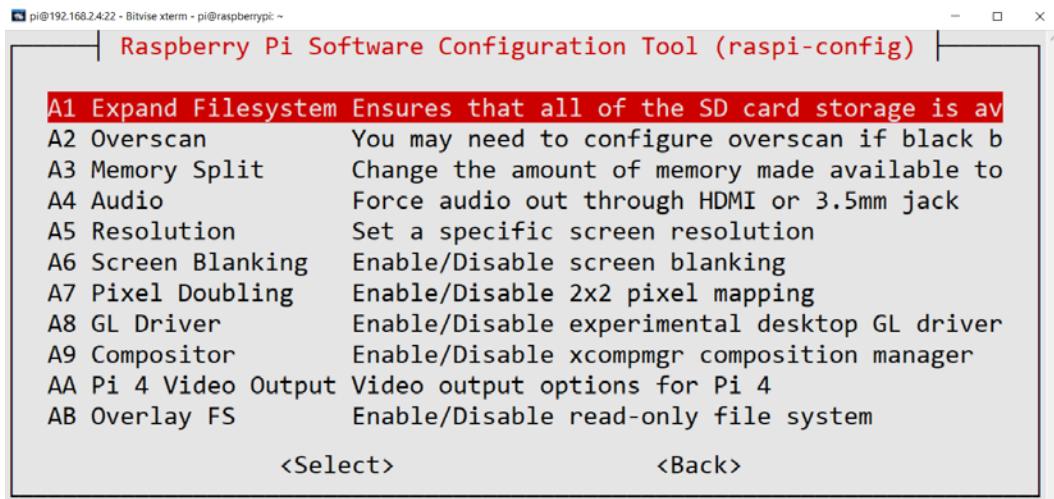


Figure 13.11 – Advanced Options

In the preceding screenshot, **A1 Expand Filesystem** expands the filesystem to make sure all the space in the microSD card is available for use. **A3 Memory Split** is used to allocate memory for the graphics.

The eighth option updates the `raspi-config` tool. If you are accessing the command prompt of the Raspberry Pi board, then this is the best way to configure Raspberry Pi.

Installation and the environment setup on Windows, Debian, and Ubuntu

We can demonstrate all the areas we have learned on other desktop computers with the Windows and Linux OSes. Only the part related to the Raspberry Pi camera module will not work with the other computers as desktop motherboards usually do not come with DSI ports. We can also run the code examples on other single-board computers that run Debian or Ubuntu.

The process to install the packages is the same on Ubuntu, Debian, and their derivatives. All the modern Linux distributions come with Python 3. We just need to use the `apt` and `pip3` tools for installation.

For a Windows PC, we need to install everything from scratch. Let's get started with understanding how to install Python 3 by taking the following steps:

1. Visit www.python.org and download the installation file for the latest Python 3 release:

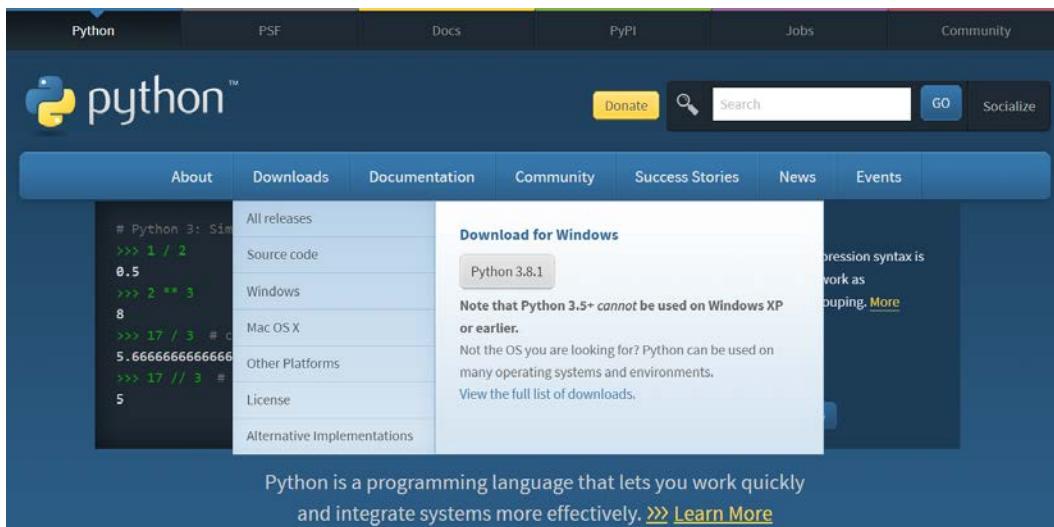


Figure 13.12 – The Python Foundation home page

Run the downloaded setup file. It will open an installation wizard, as follows:



Figure 13.13 – The Python 3 installation options

Be sure to check the **Add Python 3.8 to PATH** checkbox.

2. Then, click on **Customize installation**. The following window will appear:

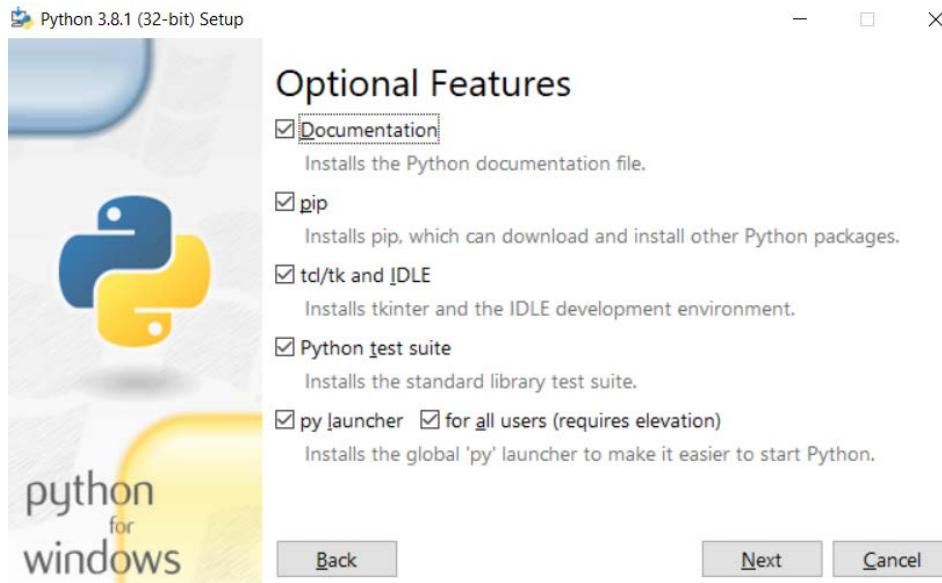


Figure 13.14 – Optional features for installation

Check all the checkboxes and then click the **Next** button. In the next window, keep all the options as they are and finish the installation.

3. Once the installation is completed, we can verify it by searching **IDLE** in the Windows search bar. Also, in cmd (Command Prompt of Windows), we can verify whether the `python` and `pip3` commands are working.

A Python 3 interpreter comes in the form of the binary executable `python.exe` file for Windows, and we can call it directly on Command Prompt if we have checked the appropriate option during the installation, as discussed previously. We can install all the packages used in the earlier chapters of this book with the `pip3` utility on Command Prompt.

Python implementations and Python distributions

A Python implementation is a program that acts as the Python programming language interpreter. The interpreter provided by <https://www.python.org/> and the one that comes with Linux is known as **CPython**. Other popular implementations include (but are not limited to) the following:

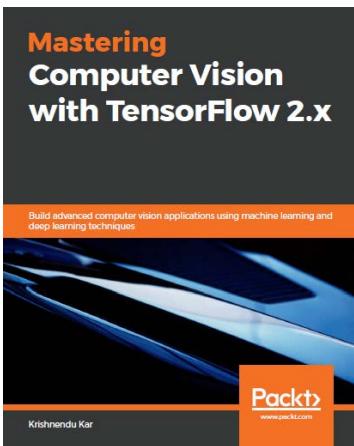
- MicroPython
- IronPython
- Stackless Python
- Jython
- PyPy
- CircuitPython

We can find a list of alternative implementations and their project URLs at <https://www.python.org/download/alternatives/>.

A Python distribution is a Python interpreter implementation and an additional set of packages bundled together. A few Python implementations are distributions themselves. Actually, there is no clear distinction between the terms *implementation* and *distribution*. We can find more information about distributions at <https://wiki.python.org/moin/PythonDistributions>.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

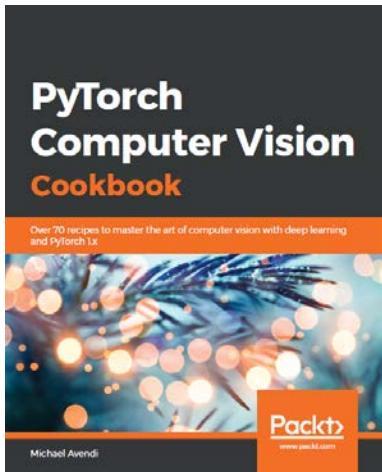


Mastering Computer Vision with TensorFlow 2.x

Krishnendu Kar

ISBN: 978-1-83882-706-9

- Explore methods of feature extraction and image retrieval and visualize different layers of the neural network model
- Use TensorFlow for various visual search methods for real-world scenarios
- Build neural networks or adjust parameters to optimize the performance of models
- Understand TensorFlow DeepLab to perform semantic segmentation on images and DCGAN for image inpainting
- Evaluate your model and optimize and integrate it into your application to operate at scale
- Get up to speed with techniques for performing manual and automated image annotation



PyTorch Computer Vision Cookbook

Michael Avendi

ISBN: 978-1-83864-483-3

- Develop, train and deploy deep learning algorithms using PyTorch 1.x
- Understand how to fine-tune and change hyperparameters to train deep learning algorithms
- Perform various CV tasks such as classification, detection, and segmentation
- Implement a neural style transfer network based on CNNs and pre-trained models
- Generate new images and implement adversarial attacks using GANs
- Implement video classification models based on RNN, LSTM, and 3D-CNN
- Discover best practices for training and deploying deep learning algorithms for CV applications

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!

Index

Symbols

2D convolution
 using, with Signal Processing
 module in SciPy 167-169
2D convolution filtering 169
7-Zip 22
8-bit unsigned integers 233

A

adaptive thresholding 156, 157
Advanced Package Tool (APT) 37
anti-aliasing 92
arithmetic operations
 performing, on images 123-125
ASUS Tinkerboard
 reference link 6

B

background subtraction
 implementing 224, 226
barcodes
 detecting, in images 232-237
batteries included motto 61

Beagleboard
 about 6
 URL 6
Berkley Software Distribution (BSD) 3
bimodal histogram images 156
binary image 142
Bitvise SSH client
 installation link 40
bitwise logical operations
 computing, on images 132-134

C

Camera Serial Interface (CSI) 110
Canny edge detection algorithm
 reference link 179
Canny edge detector
 about 179-182
 reference link 179
chroma key compositing 238
chroma key effect
 implementing 238-243
chroma keying 238
clock cycles 263
colorspace
 about 136-138, 249, 250
 converting 136-138

converting, to implement
real-life mini project 140-143
HSV colorspace 138-140
URL 136
compute module 8
computer vision
about 2
objective 3
contrast limited adaptive histogram
equalization 210
CPython 273

D

damaged images
restoring, with inpainting 190, 191
Debian
Python 3 installation 271
depth estimation 201, 202
Disk Management utility
in Windows 266-268
disparity maps 201
distance transform 248
distress signal 34, 74
Dynamic Host Configuration
Protocol (DHCP) 34
dyna-micro 5

E

events
handling 96-98

F

four-character code 108
frames per second (FPS) 102, 194, 238
retrieving, of USB webcam 106

Fritzing
reference link 73

G

gaussian distributed 163
Gaussian noise 163, 164
Geany IDE
reference link 54
General-Purpose Input/Output
(GPIO) pins 5
geometric shapes
drawing, with NumPy 92, 93
drawing, with OpenCV 92, 93
global thresholding techniques 156
GPIO
LED programming 72-79
push-button programming 80-83
Graphical User Interface (GUI)
about 3
working with 95, 96
Gunner Farneback argument
reference link 226

H

Harris Corner detection
about 185
implementing 185, 186
heatsinks
for RPi 47
high-pass filters
about 174
exploring 174-179
reference link 174
histogram
about 204
computing 204-210

-
- visualizing 204-210
 - histogram equalization 210, 212
 - Hough transforms
 - reference link 182
 - used, for finding circles and lines 182-185
 - HSV colorspace 138-140
 - hue, saturation, and value (HSV) 138

 - I**
 - ICE Tower fan
 - reference link 47
 - image contours
 - visualizing 212-214
 - image datasets
 - exploring 86
 - reference links 86
 - image inpainting
 - reference link 191
 - image processing
 - libraries, reference links 252
 - with OpenCV 86-89
 - image properties
 - retrieving 118-120
 - image quantization 199, 200
 - images
 - arithmetic operations, performing 123-125
 - barcodes, detecting 232-237
 - bitwise logical operations, computing 132-134
 - blending 126-130
 - border, adding 121-123
 - capturing, with raspistill and raspivid utilities 112, 113
 - capturing, with USB webcam 100
 - morphological transformations, applying to 214-219
 - multiplying, by constant 130
 - negative, creating 131
 - noise, adding 160
 - operations, performing 120
 - perspective transformation 150, 151
 - segmenting 192
 - splitting, into channels 121
 - transformation operations, performing 143
 - transitioning 126-130
 - visualizing, with Matplotlib 89-91
 - images processing, with Mahotas
 - about 246
 - colorspace 249, 250
 - distance transform 248
 - images, thresholding
 - about 152-155
 - adaptive thresholding 156, 157
 - Otsu's binarization method 156
 - Infrared (IR) filter 110
 - inpainting
 - damaged images, restoring 190, 191
 - Integrated Development and Learning Environment (IDLE)
 - reference link 54
 - Integrated Development Environments (IDEs) 54
 - intel boards
 - about 7
 - reference link 7
 - Intel Up Squared Kit
 - reference link 7
 - internet of things (IoT) 5

J

Jetson Nano Developer Kit
reference link 7
Jupyter Notebook
exploring, for Python 3
programming 252-261

K

kernels
working with 166, 167
k-means algorithm
versus mean shift algorithm 200
k-means clustering algorithm
using 194-198
K Nearest Neighbor (KNN) 224

L

L1 and L2 norms
reference link 179
LED programming
with GPIO 72-79
local histogram equalization 210
low-pass filtering 170-172

M

Mahotas
built-in images, reading with 247
combining, with OpenCV 250, 251
images processing 246
images, reading with 247
images, thresholding with 247, 248
reference link 246
mathematical aspects, of affine
transformations
reference link 145

mathematical transformation
operations, on images
affine transformation 145-150
performing 143
rotation 145-150
scaling 144
translation 145-150

Matplotlib
about 65-70
installing 65
using, to visualize images 89-91

Max RGB filter
about 222
implementing 222-224
mean shift algorithm
versus k-means algorithm 200
mean shift algorithm
segmentation 192-194

Mini-Micro Designer 1 (MMD-1) 5
morphological transformations
applying, to images 214-219
motion
detecting 228-232
MP4Box 113

N

Navier-Stokes method 191
ndarrays, NumPy
creating 63
operations, performing 63, 64
with linear algebra 64
non-maximum suppression
reference link 179
NumPy
reference link 64
used, for drawing geometric
shapes 92, 93

NumPy, SciPy ecosystem
 basics 62
 linear algebra, using with ndarrays 64
 ndarrays, creating 63
 operations, performing on
 ndarrays 63, 64
NVIDIA Jetson
 about 7
 reference link 7

O

Open Source Computer Vision (OpenCV)
 about 3
 blurring functions 169
 combining, with Mahotas 250, 251
 filtering functions 169
 installing, on RPi board 46, 47
 management 263, 264
 performance measurement 263, 264
 URL 4
 used, for capturing images with
 USB webcam 103, 104
 used, for capturing live videos
 with USB webcam 104, 105
 used, for drawing geometric
 shapes 92, 93
 used, for image processing 86-89
 used, for playing back USB
 webcam videos 109
optical flow (optic flow)
 application areas 228
 computing 226-228
OSes
 for Raspberry Pi 13, 14
 reference link 13
Otsu's binarization Method 156

P

PacketBeagle
 URL 6
perspective transformation
 of images 150, 151
picamera
 using, with Python 3 113-115
Pi camera boards 110
Pi camera module 110, 112
Pip Installs Python (pip)
 about 65
 reference link 65
Poisson noise 164, 165
present working directory 103
primitive paint application
 creating 96-98
printed circuit board (PCB) 4
Pulse Width Modulation (PWM) 5
push-button programming
 with GPIO 80-83
PuTTY
 URL 40
PyMeanShift 192
Python
 configuring, on Raspberry Pi 53
 configuring, on Raspbian OS 53
 URL 273
 used, for capturing images with
 USB webcam 103, 104
 used, for capturing live videos
 with USB webcam 104, 105
Python 3
 about 52, 53
 picamera, using with 113-115
 Raspberry Pi GPIO programming 71, 72
 reference link 53

- used, for capturing videos with RPi camera module 116
 - working with, in interactive mode 59
 - Python 3 IDEs
 - configuring, on Raspbian OS 54-58
 - Python 3 installation
 - on Debian 271
 - on Ubuntu 271
 - on Windows 271-273
 - Python 3 programming
 - basics 59-61
 - Jupyter Notebook, exploring for 252-261
 - Python distributions
 - about 273
 - reference link 273
 - Python implementations
 - about 273
 - reference link 273
 - Python Package Index
 - reference link 65
- ## R
- random normal noise 165
 - Raspberry Pi (RPi)
 - about 8
 - heatsinks 47
 - logging, remotely with SSH 40-43
 - OSes 13
 - Python, configuring 53
 - Raspbian, setting up 14
 - URL 8
 - Raspberry Pi configuration
 - methods 268
 - Raspberry Pi Foundation
 - reference link 8
 - Raspberry Pi GPIO programming
 - using, with Python 3 71, 72
 - Raspberry Pi Model 4B
 - about 9-11
 - product specifications 9, 10
 - reference link 9
 - Raspberry Pi models
 - about 8
 - buying, stores 12, 13
 - Raspberry Pi Model 4B 9
 - Raspberry Pi Zero W 12
 - Raspberry Pi Universal Power Supply
 - reference link 15
 - Raspberry Pi Zero W
 - about 12
 - reference link 11
 - Raspbian
 - URL 13
 - Raspbian image
 - boot 23
 - reference link 14
 - system 23
 - Raspbian OS
 - about 13
 - Python 3 IDEs, configuring 54-58
 - Python, configuring 53
 - Raspbian OS desktop
 - accessing remotely 44-46
 - Raspbian OS installation, on microSD card
 - for OSes, reference link 26
 - Raspbian OS microSD card
 - reusing 264
 - Raspbian, setting up on Raspberry Pi
 - about 14
 - booting up, with microSD card 26-34
 - component requisites 14-19
 - microSD card, preparing
 - manually 23-26
 - RPi board models, connecting to internet 34-37

software requisites, downloading 20-23
raspi-config command-line utility
 overview 268-271
raspistill and raspivid utilities
 used, for capturing images 112, 113
 used, for capturing videos 112, 113
relative motion 226
RPi 4B
 overclocking 47-49
RPi board
 OpenCV, installing 46, 47
 updating 37, 38
RPi camera module
 used, for capturing videos
 with Python 3 116
RPi USB Webcams
 reference link 99

S

salt and pepper noise 160-163
Save our Souls (SOS) 73
SciPy
 2D convolution, with Signal
 Processing module 167-169
SciPy ecosystem
 about 62
 libraries 62
 Matplotlib 65
 NumPy 62
script mode 59
SD card formatter
 used, for formatting SD card 264-266
SD cards
 formatting, with SD card
 formatter 264-266
SD cards, formatting tool
 download link 264

Secure Shell (SSH)
 used, for logging remotely
 into RPi 40-43
segments 152
Serial Peripheral Interface (SPI) 5
Signal Processing module
 using, with 2D convolution
 in SciPy 167-169
shot noise 164, 165
single-board computer (SBC)
 about 4
 advantages 5
 ASUS Tinkerboard 6
 Beagleboard 6
 disadvantages 5
 intel boards 7
 Microcomputer Trainer MMD-1 5
 NVIDIA Jetson 7
Stereoscopic Vision 201
sudo raspi-config command 43

T

Telea method 191
timelapse photography 100-103
timelapse video 100
torrent software
 download link 21
tracking
 detecting 228-232

U

Ubuntu
 Python 3 installation 271
USB webcam
 FPS, retrieving 106
 images, capturing with 100

resolution 105, 106
timelapse photography 100-103
used, for capturing images
 with OpenCV 103, 104
used, for capturing images
 with Python 103, 104
used, for capturing live videos
 with OpenCV 104, 105
used, for capturing live videos
 with Python 104, 105
used, for recording videos 103
video, playing back with OpenCV 109
videos, saving 107, 108
working with 99

V

videos
 capturing, with raspistill and
 raspivid utilities 112, 113

W

Win32DiskImager 23
Windows
 Disk Management utility 266-268
 Python 3 installation 271-273

X

xrdp
 URL 44

