EXPERT INSIGHT

Learning OpenCV 5 Computer Vision with Python

Build practical computer vision and machine learning applications with OpenCV 5 and Python 3



Fourth Edition

Joseph Howse
Joe Minichino

‹packt›

Learning OpenCV 5 Computer Vision with Python

Fourth Edition

Build practical computer vision and machine learning applications with OpenCV 5 and Python 3

Joseph Howse
Joe Minichino

packt

Learning OpenCV 5 Computer Vision with Python

Fourth Edition

Copyright © 2026 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 authors 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.
This book was written by Joseph Howse and Joe Minichino. Generative AI tools were used only to assist with phrasing, and all technical content and code were created, verified, and tested by the authors and Packt's editorial team. Packt does not accept AI-generated content that replaces expert authorship.

Portfolio Director: Sunith Shetty
Relationship Lead: Deepesh Patel
Project Manager: Yash Basil
Content Engineer: Aditi Chatterjee
Technical Editor: Seemanjay Ameriya
Indexer: Pratik Shirodkar
Production Designer: Ajay Patule
Growth Lead: Ankur Mulasi

First published: April 2013
Second edition: September 2015
Third edition: February 2020
Fourth edition: October 2026

Production reference: 1071026

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80323-022-1
www.packtpub.com

I dedicate my work to Sam, Jan, Bob, Bunny, and the cats, who have been my lifelong guides and companions.

– Joseph Howse

Contributors

About the authors

Joseph Howse lives in a Canadian fishing village, where he chats with his cats, crafts his books, and nurtures an orchard of hardy fruit trees. He is President of Nummist Media, a publishing and consulting company with specialties including computer vision and medical imaging. Since 2015, he has written a large part of Packt's library of technical books on OpenCV. Joseph is also the author of an award-winning novel series, *Next Year's Snow*, which follows a Soviet/Ukrainian family through several generations to the present day.

I want to thank all the people who have shaped this book's four editions: the readers, my coauthor Joe Minichino, and the teams of editors, including Denim Pinto, Rebecca Anne Robinson, and Namrata Katara, technical reviewers, and marketers. Above all, my family makes my work possible, and I dedicate this book to them.

Joe Minichino is a Senior Cloud Architect at fourTheorem. Joe specializes in AWS architecture, ML/AI, and Data Analytics. Born and raised in Varese, Lombardy, Italy, and coming from a humanistic background in philosophy (at Università Statale in Milan), Joe has lived in Cork, Ireland, since 2004. There, he graduated in computer science at the Cork Institute of Technology.

I want to thank my partner, Rowena, and my daughters, Livia and Siabhra, for all the joy and support they bring into my life; my co-author, Joe Howse; my employer, fourTheorem, for being truly special; all my colleagues there; and the team at Packt for the great help and support.

About the reviewers

Süleyman Türkmen is a self-taught hobbyist programmer who lives in Turkey and works as an assistant administrator at a high school. He has been an active participant in the OpenCV community since 2015, contributing numerous bug fixes and improvements to the OpenCV library.

I would like to express my sincere gratitude to the Packt editors for the opportunity to be part of this project. I would also like to thank the members of the OpenCV Forum, especially Berak, Steven Puttemans, and Laurent Berger.

Deepak Ghimire, Ph.D., is a Senior Research Engineer at the Korea Electronics Technology Institute (KETI), South Korea. He earned his bachelor's degree in Nepal, followed by an M.Sc. and Ph.D. in Computer Science and Engineering from Jeonbuk National University in South Korea. He has more than 15 years of experience in computer vision and AI, specializing in edge AI, autonomous systems, ADAS, robotics, real-time inference, sensor fusion, and model optimization. Previously, he served as a Research Professor at Soongsil University and a postdoctoral researcher at KETI. He has authored and co-authored more than 50 peer-reviewed publications.

Table of Contents

Preface **xvii**

Free benefits with your book ..... **xxii**

Part 1: Exploring the Fundamentals of OpenCV **1**

Chapter 1: Setting Up OpenCV **3**

Technical requirements **5**
What's new in OpenCV 5 ..... **5**
Optimizing OpenCV for specific hardware **6**
Choosing and using the right setup tools **7**
Installation on Windows • 8
Using a ready-made OpenCV package • 8
Building OpenCV from source • 9
Installation on macOS • 12
Using Homebrew with ready-made packages • 13
Using Homebrew with your own custom packages • 14
Installation on Debian, Ubuntu, Linux Mint, and similar systems • 15
Using a ready-made OpenCV package • 15
Building OpenCV from source • 16
Installation on Arch Linux, Manjaro, and similar systems • 18
Using a ready-made OpenCV package • 19
Building OpenCV from source • 19
Installation on other Unix-like systems • 22
Running the official sample code **22**
Finding documentation, help, and updates **23**
Finding this book's sample code **25**
Summary **25**

Chapter 2: Handling Files, Cameras, and GUIs **27**

Technical requirements **28**

Basic I/O scripts 28
Reading/writing an image file • 28
Converting between an image and raw bytes • 33
Accessing image data with numpy.ndarray • 35
Converting between numpy.ndarray and cv2.UMat • 38
Reading/writing a video file • 39
Enabling hardware acceleration for video I/O • 40
Capturing camera frames • 42
Sending a video stream over a network • 45
Displaying an image in a window • 50
Displaying camera frames in a window • 51
Project Cameo (face tracking and image manipulation) 53
Cameo – an object-oriented design 54
Abstracting a video stream with managers.CaptureManager • 55
Abstracting a window and keyboard with managers.WindowManager • 61
Applying everything in Cameo • 62
Summary 65

Chapter 3: Processing Images with OpenCV 67

Technical requirements 67
Converting images between different color models 68
Light is not paint • 69
Exploring the Fourier transform 69
HPFs and LPFs • 70
Creating modules 74
Edge detection 74
Custom kernels – getting convolutions 76
Modifying the application 78
Edge detection with Canny 80
Contour detection 82
Bounding box, minimum area rectangle, and minimum enclosing circle • 83
Convex contours and the Douglas-Peucker algorithm • 87
Detecting lines, circles, and other shapes 90
Detecting lines • 91
Detecting circles • 94

Detecting other shapes • 99
Summary **100**

Chapter 4: Depth Estimation and Segmentation 101

Technical requirements **102**
Creating modules **103**
Capturing frames from an OpenNI-compatible depth camera **104**
Converting 10-bit images to 8-bit **107**
Creating a mask from a disparity map **109**
Modifying the application for OpenNI-compatible cameras **110**
Depth estimation with a normal camera **115**
Stereo imaging with an OAK-D camera **122**
Foreground detection with the GrabCut algorithm **128**
Image segmentation with the Watershed algorithm **133**
Image segmentation with HFS • 136
Summary **138**

Part 2: Finding and Tracking People, Patterns, and Objects 141

Chapter 5: Detecting and Recognizing Faces 143

Technical requirements **144**
Conceptualizing Haar cascades **144**
Getting Haar cascade data **146**
Using OpenCV to perform face detection **147**
Performing face detection on a still image • 147
Performing face detection on a video • 149
Performing face recognition **153**
Generating the data for face recognition • 153
Choosing a face recognition algorithm • 155
Loading the training data for face recognition • 156
Performing face recognition with Eigenfaces • 158
Performing face recognition with Fisherfaces • 160
Performing face recognition with LBPH • 161
Discarding results based on the confidence score • 161
Swapping faces in infrared **162**

Modifying the application's loop • 164
Masking a copy operation • 167
Summary 169

Chapter 6: Retrieving Images and Searching Using Image Descriptors 171

Technical requirements 172
Understanding types of feature detection and matching 172
Defining features • 173
Detecting Harris corners 173
Detecting DoG features and extracting SIFT descriptors 176
Anatomy of a keypoint • 179
Detecting Fast Hessian features and extracting SURF descriptors 180
Using ORB with FAST features and BRIEF descriptors 182
FAST • 183
BRIEF • 184
Brute-force matching • 184
Matching a logo in two images • 185
Filtering matches using K-nearest neighbors and the ratio test 189
Matching with FLANN 194
Finding homography with FLANN-based matches 199
A sample application – tattoo forensics 203
Saving image descriptors to a file • 203
Scanning for matches • 205
Summary 208

Chapter 7: Building Custom Object Detectors 211

Technical requirements 212
Understanding HOG descriptors 212
Visualizing HOG • 213
Using HOG to describe regions of an image • 215
Understanding NMS 216
Understanding SVMs 217
Detecting people with HOG descriptors 218
Creating and training an object detector 223

Understanding BoW • 224
Applying BoW to computer vision • 225
k-means clustering • 226
Detecting cars 227
Combining an SVM with a sliding window • 235
Detecting a car in a scene • 236
Saving and loading a trained SVM • 243
Training a custom HOG detector • 243
Summary 249

Chapter 8: Tracking Objects 251

Technical requirements..... 252
Detecting moving objects with background subtraction..... 252
Implementing a basic background subtractor • 254
Using a MOG background subtractor • 257
Using a KNN background subtractor • 262
Using GMG and other background subtractors • 264
Tracking colorful objects using MeanShift and CamShift 267
Planning our MeanShift sample • 268
Calculating and back-projecting color histograms • 269
Understanding the parameters of cv2.calcHist • 272
Understanding the parameters of cv2.calcBackProject • 273
Implementing the MeanShift example • 274
Using CamShift • 277
Finding trends in motion using the Kalman filter 279
Understanding the predict and update phases • 280
Tracking a mouse cursor • 281
Tracking pedestrians..... 284
Planning the flow of the application • 285
Comparing the object-oriented and functional paradigms • 285
Implementing the Pedestrian class • 287
Implementing the main function • 291
Considering the next steps • 295
Modifying the pedestrian tracker to use MIL • 295
Summary 299

Part 3: Tackling Deeper Problems: 6DOF Tracking, DNNs, and Scaling Up **301**

Chapter 9: Camera Models and Augmented Reality **303**

Technical requirements ... **304**
Understanding 3D image tracking and augmented reality ... **304**
Understanding camera and lens parameters • 311
Understanding cv2.solvePnPRansac • 316
Implementing the demo application ... **319**
Importing modules • 319
Performing grayscale conversion • 320
Performing 2D-to-3D spatial conversions • 321
Implementing the application class • 323
Initializing the tracker • 324
Implementing the main loop • 333
Tracking the image in 3D • 334
Initializing and applying the Kalman filter • 338
Converting between Rodrigues vectors and Euler vectors • 344
Drawing the tracking results and masking the tracked object • 346
Running and testing the application • 349
Improving the 3D tracking algorithm • 354
Summary ... **355**

Chapter 10: Introduction to 3D Navigation, Odometry, and Point Clouds **357**

Technical requirements ... **358**
Understanding visual odometry ... **359**
Building a visual odometry application for OAK-D ... **361**
Defining the camera matrix and other odometry parameters • 361
Aligning depth frames to match RGB frames • 365
Preparing odometry frames and computing the odometry • 368
Accumulating the odometry result over multiple frames • 371
Visualizing the frames and odometry results • 373
Extending the odometry application • 377

Understanding point clouds **378**
Loading and showing point clouds **380**
Segmenting point clouds into multiple objects **383**
Using BIRCH for segmentation • 385
Using DBSCAN for segmentation • 387
Summary **390**

Chapter 11: Introduction to Neural Networks with OpenCV 391

Technical requirements **392**
Understanding ANNs **393**
Understanding neurons and perceptrons • 395
Understanding the layers of a neural network • 396
Choosing the size of the input layer • 397
Choosing the size of the output layer • 397
Choosing the size of the hidden layer • 397
Training a basic ANN in OpenCV **398**
Training an ANN classifier in multiple epochs **401**
Recognizing handwritten digits with an ANN **406**
Understanding the MNIST database of handwritten digits • 407
Choosing training parameters for the MNIST database • 408
Implementing a module to train the ANN • 409
Implementing a minimal test module • 414
Implementing the main module • 414
Trying to improve the ANN's training • 421
Finding other potential applications • 423
Using DNNs from other frameworks in OpenCV **424**
Detecting and classifying objects with third-party DNNs **425**
Detecting and classifying faces with third-party DNNs **429**
Using OpenCV with MediaPipe and TensorFlow to classify gestures and actions **433**
What is MediaPipe? • 434
TensorFlow • 434
Hand recognition • 435
Hand gesture recognition • 437
Generating labeled training data • 438
Building and training the classifier with Jupyter Notebooks • 443

Testing the model on camera input • 447

Summary 452

Chapter 12: OpenCV Applications at Scale 453

Technical requirements 453

Understanding containers and Docker basics 454

The 12-Factor app • 455

Docker basics • 456

Docker version • 456

Docker run • 457

Docker pull • 458

Dockerfile • 459

A note on Linux distributions in Docker images • 463

AWS Serverless (Fargate, Lambda) and AWS SAM 464

AWS Lambda • 464

AWS SAM • 465

Is this really necessary? • 476

Example application: Face detection • 477

A note on AWS Fargate • 481

Summary 481

Chapter 13: Unlock the Code Bundle and the PDF Version 483

Unlock this book's free benefits in three easy steps 484

Appendix A: Bending Color Space with the Curves Filter 487

Formulating a curve 487

Caching and applying a curve 489

Designing object-oriented curve filters 491

Emulating photo films 493

Emulating Kodak Portra • 494

Emulating Fuji Provia • 495

Emulating Fuji Velvia • 495

Emulating cross-processing • 496

Summary 496

Other Books You May Enjoy **498**

Index **501**

Preface

Now in its fourth edition, this is the original book on OpenCV's Python bindings. Readers will learn a great range of techniques and algorithms, from the classics to the state-of-the-art, and from geometry to machine learning. All of this is in aid of solving practical computer vision problems in well-built applications. Using OpenCV 5 and Python 3, we adopt an approach that is accessible to computer vision novices, yet also informative for experts who want to expand and update their skills.

We start with an introduction to OpenCV 5 and explain how to set it up with Python 3 on various platforms. Next, you'll learn how to perform basic operations such as reading, writing, manipulating, and displaying still images, videos, and camera feeds. You'll learn about image processing and video analysis, along with depth estimation and segmentation, and you'll gain practice by building a simple GUI application. Next, you'll tackle two popular problems: face detection and face recognition.

As we advance, we'll explore concepts of object classification and machine learning, enabling you to create and use object detectors and classifiers, and even track objects in movies or video camera feeds. Then, we'll extend our work into 3D tracking and augmented reality. Finally, we'll learn about **artificial neural networks** (**ANNs**) and **deep neural networks** (**DNNs**) as we develop applications to recognize handwritten digits and to classify a person's age.

By the end of this book, you will have acquired the right knowledge and skills to embark on your own real-world computer vision projects.

Who this book is for

This book is intended for people interested in learning computer vision, machine learning, and OpenCV in the context of practical real-world applications. The book will appeal to computer vision novices as well as experts who want to get up to date with OpenCV 5 and Python 3. Readers should be familiar with basic Python programming, but no prior knowledge of image processing, computer vision, or machine learning is required.

What this book covers

Chapter 1, Setting Up OpenCV, explains how to set up OpenCV 5 with Python 3 on various platforms. It also provides troubleshooting steps for common problems.

Chapter 2, Handling Files, Cameras, and GUIs, introduces OpenCV's I/O functionalities. It also discusses an object-oriented design for a GUI project that we will develop further in other chapters.

Chapter 3, Processing Images with OpenCV, presents some techniques required to alter images, such as manipulating colors, sharpening an image, marking contours of objects, and detecting geometric shapes.

Chapter 4, Depth Estimation and Segmentation, shows you how to use data from a depth camera to identify foreground and background regions, so that we can limit an effect to only the foreground or background.

Chapter 5, Detecting and Recognizing Faces, introduces some of OpenCV's functionality for face detection and recognition, along with the data files that define particular types of detectable objects.

Chapter 6, Retrieving Images and Searching Using Image Descriptors, shows how to describe the features of an image with the help of OpenCV, and how to make use of features to match and search for images.

Chapter 7, Building Custom Object Detectors, applies a combination of computer vision and machine learning algorithms to locate and classify objects in an image. It shows how to implement this combination of algorithms with OpenCV.

Chapter 8, Tracking Objects, demonstrates ways to track and predict the motion of people and objects in a video or live camera feed.

Chapter 9, Camera Models and Augmented Reality, enables you to build an augmented reality application that uses information about cameras, objects, and motion to superimpose 3D graphics atop tracked objects in real time.

Chapter 10, Introduction to 3D Navigation, Odometry, and Point Clouds, shows how to measure motion using input from a depth camera. It also introduces the use of point clouds and segmentation techniques to represent objects in 3D scenes.

Chapter 11, Introduction to Neural Networks with OpenCV, introduces you to **artificial neural networks** (**ANNs**) and **deep neural networks** (**DNNs**) in OpenCV, and illustrates their usage in real-world applications.

Chapter 12, OpenCV Applications at Scale, shows how to package OpenCV applications and deploy them to a server, making them available to the public even at a large scale.

Appendix A, Bending Color Space with the Curves Filter, describes the concept of color curves and our implementation of them using SciPy.

To get the most out of this book

The reader is expected to have at least basic proficiency in the Python programming language.

A Windows, macOS, or Linux development machine is recommended. You can refer to *Chapter 1, Setting Up OpenCV*, for instructions about setting up OpenCV 5, Python 3, and other dependencies.

This book takes a hands-on approach to learning and includes 104 sample scripts, along with sample data. Working through these examples as you read the book will help reinforce the concepts.

The code for this book is released under the BSD 3-Clause open source license. The reader is encouraged to use, modify, improve, and even publish their changes to these example programs. However, be aware that our third-party dependencies such as libraries, datasets, and AI/ML models may have different licenses than our sample code itself.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Disclaimer on images

Some images in this title are presented for contextual purposes, and the readability of the graphic is not crucial to the discussion. Please refer to our free graphic bundle to download the images.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781803230221`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Note that we allow kernel sizes to be specified as arguments for `strokeEdges`. The `blurKsize` argument is used as `ksize` for `medianBlur`, while `edgeKsize` is used as `ksize` for `Laplacian`."

A block of code is set as follows:

```
import cv2
import os

output_folder = '../data/at/my_name'
if not os.path.exists(output_folder):
    os.makedirs(output_folder)
```

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

```
> pip install opencv-contrib-python
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "NumPy has a **fast Fourier transform (FFT)** package, which contains the `fft2` method. This method allows us to compute a **discrete Fourier transform (DFT)** of the image."

> **Note**
>
> ***Warnings*** or important notes appear like this.

> **Tip**
>
> Tips and tricks 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 or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

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 reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

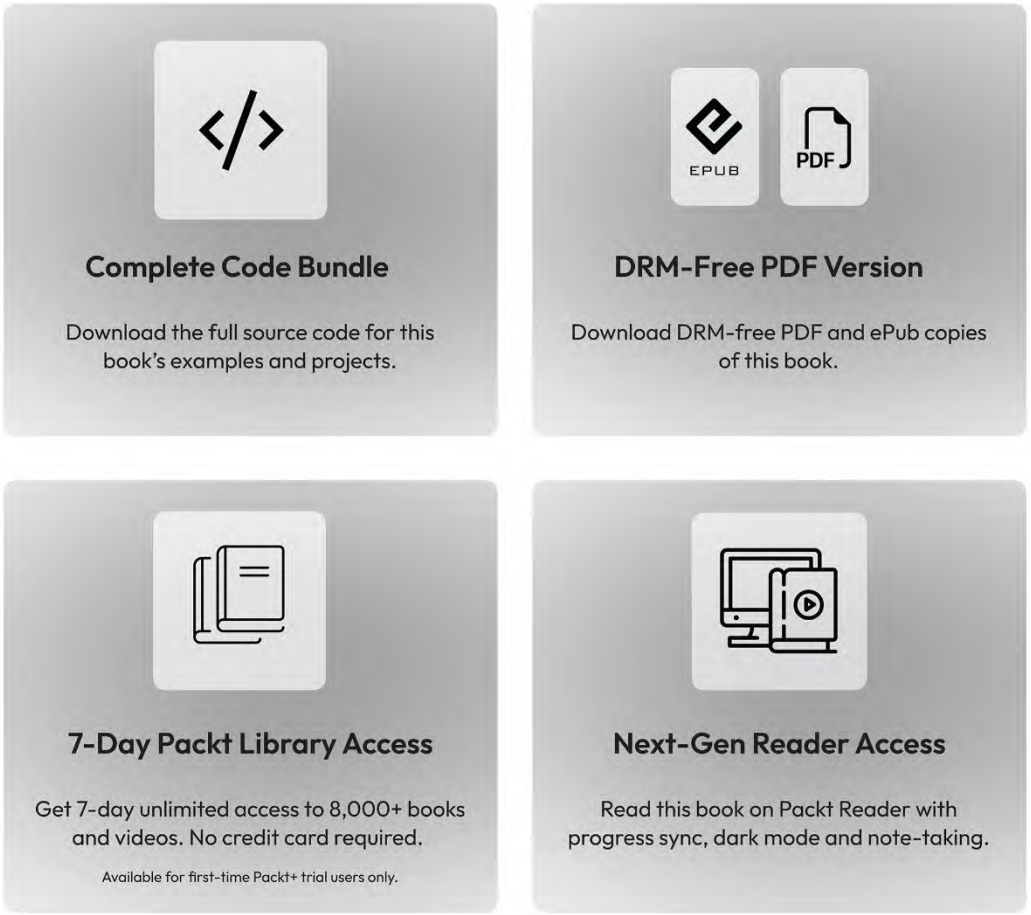
Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

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 `http://authors.packt.com/`.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *Learning OpenCV 5 Computer Vision with Python, Fourth Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



`https://packt.link/r/1803230223`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1

Exploring the Fundamentals of OpenCV

Not surprisingly, most computer vision projects involve setting up and programming a computer system to capture, process, display, and save visual data. *Part 1* deals with these fundamentals: how to configure and install OpenCV; how to work with video files, webcams, and depth cameras; how to build GUI applications; and what information we can glean from analyzing the pixel data and geometry of typical images.

This part contains the following chapters:

- *Chapter 1*, *Setting Up OpenCV*
- *Chapter 2*, *Handling Files, Cameras, and GUIs*
- *Chapter 3*, *Processing Images with OpenCV*
- *Chapter 4*, *Depth Estimation and Segmentation*

1

Setting Up OpenCV

You've picked up this book, so you may already have an idea of what OpenCV is. Maybe you have heard of capabilities that seem to come straight out of science fiction, such as training an artificial intelligence model to recognize anything that it sees through a camera. If this is your interest, you will not be disappointed! **OpenCV** stands for **Open Source Computer Vision**. It is a free computer vision library that allows you to manipulate images and videos to accomplish a variety of tasks, from displaying frames from a webcam to teaching a robot to recognize real-life objects.

In this book, you will learn how to leverage the immense potential of OpenCV with the Python programming language. Python is an elegant language with a relatively shallow learning curve and very powerful features. This chapter is a quick guide to setting up Python 3, OpenCV 5, and other dependencies. As part of OpenCV, we will set up the `opencv_contrib` modules, which offer additional functionality that is maintained by the OpenCV community rather than the core development team. After the setup, we will also look at OpenCV's Python sample scripts and documentation.

The following related libraries are covered in this chapter:

- **NumPy**: This library is a dependency of OpenCV's Python bindings. It provides numeric computing functionality, including efficient arrays.
- **SciPy**: This library is a scientific computing library that is closely related to NumPy. It is not required by OpenCV, but it is useful if you wish to manipulate data in OpenCV images.
- **OpenNI 2**: This library is an optional dependency of OpenCV. It adds support for certain depth cameras, such as the Asus Xtion PRO.

For this book's purposes, OpenNI 2 can be considered optional. It is used throughout *Chapter 4, Depth Estimation and Segmentation*, but is not used in the other chapters or appendices.

Optionally, if we build OpenCV with a custom configuration, we can include support for OpenNI 2, and we can even include OpenCV's implementations of some patented algorithms, which OpenCV calls *non-free*. Notably, one of our code samples in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*, relies on a patented algorithm called SURF. However, the rest of the book does not use any patented algorithms, so this non-free content can be considered optional; in any case, patent restrictions may make it unsuitable for redistribution in your future projects.

Note

This book focuses on OpenCV 5, the new major release of the OpenCV library. Additional information about OpenCV is available at `https://opencv.org/`, and the official documentation is available at`https://docs.opencv.org/5.x`.

We will cover the following topics in this chapter:

- What's new in OpenCV 5
- Choosing and using the right setup tools
- Running samples
- Finding documentation, help, and updates

Note

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Again, it is worth mentioning that this chapter includes optional instructions on how to build OpenCV from source code, with custom configurations that are relevant to a few projects in this book. However, configuring and customizing OpenCV is a large topic, and the configuration options sometimes change between minor releases (such as OpenCV 5.1 versus 5.0), so advanced users may wish to refer to additional tutorials online.

Note

One of the book's authors, Joseph Howse, maintains a GitHub repository of additional OpenCV build recipes at `https://github.com/JoeHowse/OpenCVBuildKit`.

Technical requirements

This chapter assumes that you are using one of the following operating systems:

- Windows 10 or a later version
- macOS 10.15 (Catalina) or a later version
- Debian 10 (Buster) or a later version, or a derivative such as the following:
    - Ubuntu 18.04 (Bionic Beaver) or a later version
    - Linux Mint 19 or a later version
- Arch Linux, or a derivative such as Manjaro

For editing Python scripts and other text files, this book's authors simply recommend that you have a good text editor. **Visual Studio** (**VS**) Code is the most suitable editor because of its simplicity, and almost all OS distributions support this editor. Other examples include the following:

- Notepad++ for Windows
- BBEdit (free version) for macOS
- GEdit for the GNOME desktop environment on Linux
- Kate for the KDE Plasma desktop environment on Linux

We also recommend that you have a Git client installed since this will be the easiest way to obtain the book's latest sample code, which is hosted on GitHub. You can download the example project and code for this book by following the instructions in the ***Download the example code files*** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Besides the operating system, there are no other prerequisites for this setup chapter.

What's new in OpenCV 5

If you are an OpenCV veteran, you might want to know more about OpenCV 5's changes before you decide to install it. Here are some of the highlights:

- The C++ implementation of OpenCV has been updated to C++14. OpenCV's Python bindings wrap the C++ implementation, so as Python users, we may gain some performance advantages from this update, even though we are not using C++ directly.
- Support for Python 2 has been dropped because this Python version is obsolete and no longer maintained. If you have any old Python 2 projects, you should upgrade them to Python 3.

- For many algorithms, support for named parameters has been added in order to make the initialization code more concise and readable.
- For the `pip` package installer, the `opencv-python` and `opencv-contrib-python` packages are now maintained by the core OpenCV team rather than a third party. The `opencv-contrib-python-nonfree` package has been discontinued due to patent restrictions.
- The SIFT algorithm's patent has expired and SIFT is now included in standard builds of OpenCV.
- Many new optimizations have been implemented. Existing OpenCV 4 projects can take advantage of many of these optimizations without further changes beyond updating the OpenCV version.
- OpenCV's DNN module, in particular, is better optimized. Also, the loading and caching of DNN models are better automated.
- Support for RISC-V processors has been added.

A new module has been added for 2D tracking, including some functionality that was previously located in `opencv_contrib`.

Whether or not you have used a previous version of OpenCV, this book will serve as a general guide to OpenCV 5, and some of the new features will receive special attention in subsequent chapters.

Optimizing OpenCV for specific hardware

Generally, the setup instructions in this chapter are compatible with any hardware architecture that is supported by OpenCV and the operating system. For example, the setup instructions that target Linux operating systems will work equally well on x64, x86, ARM, or RISC-V hardware (except that the optional OpenNI package is unavailable for RISC-V).

That said, OpenCV supports many optional optimizations for specific hardware. By following the instructions in this chapter, you will get an OpenCV installation that uses a default set of optimizations for your hardware. The defaults are generally good, but for some applications, you could get even better performance with a more highly customized installation of OpenCV. Here are a few examples of use cases:

- For x64, x86, or ARM processors, OpenCV can optionally integrate with Intel **Threading Building Blocks** (**TBB**), a multithreading library. OpenCV can use TBB to speed up certain parallel algorithms, notably the Haar cascade classifiers that we will study in *Chapter 5, Detecting and Recognizing Faces*.

- For NVIDIA graphics processors, OpenCV can optionally integrate with CUDA, another multiprocessing library. OpenCV can use CUDA to speed up various deep learning algorithms, as well as any kind of operation on very large blocks of data, such as very large images.
- For AMD graphics processors, OpenCV can optionally integrate with clBLAS and clFFT. These AMD libraries provide fine-tuned versions of certain OpenCL operations. OpenCL is another multiprocessing framework. Already, by default, OpenCV enables OpenCL optimizations on any compatible hardware, yet clBLAS and clFFT may make these optimizations perform even better on AMD hardware. Specifically, OpenCV can use clBLAS and clFFT to speed up two kinds of operations that are relevant to many signal processing applications: **General Matrix Multiply** (**GEMM**) and **Fast Fourier Transform** (**FFT**). For example, GEMM is the fundamental operation in the Kalman filter, which we will use for motion stabilization in *Chapter 8, Tracking Objects*, and *Chapter 9, Camera Models and Augmented Reality*.

> **Note**
>
> For examples of how to enable specific optimizations, see Joseph Howse's repository of OpenCV build recipes at `https://github.com/JoeHowse/OpenCVBuildKit/tree/opencv5_python`. Note that optimized OpenCV builds may have additional runtime dependencies, so you may need to copy additional `DLLs` into your Python `site_packages` folder.

For now, let's proceed using default optimizations, which are fine for the purpose of running this book's sample projects on a wide range of hardware. However, bear in mind that many other optimization options are available to you for your future projects, where you may have specific performance requirements on specific hardware.

Choosing and using the right setup tools

We are free to choose various setup tools depending on our operating system and how much configuration we want to do.

Regardless of the choice of operating system, Python offers some built-in tools that are useful for setting up a development environment. These tools include a package manager called `pip` and a virtual environment manager called `venv`. Some of this chapter's instructions will cover `pip` specifically, but if you would like to learn about `venv`, please refer to the official Python documentation at `https://docs.python.org/3/library/venv.html`.

Tip

You should consider using `venv` if you plan to maintain a variety of Python projects that might have conflicting dependencies – for example, projects that depend on different versions of OpenCV. Each of `venv`'s virtual environments has its own set of installed libraries, and we can switch between these environments without reinstalling anything. Within a given virtual environment, libraries can be installed using `pip` or, in some cases, other tools.

Let's take an overview of the setup tools available for Windows, macOS, Ubuntu, Arch Linux, and other Unix-like systems.

Installation on Windows

Windows does not come with Python preinstalled. However, an installation wizard is available for Python, and Python provides a package manager called `pip`, which lets us easily install ready-made builds of NumPy, SciPy, and OpenCV. Alternatively, we can build OpenCV from source in order to enable nonstandard features, such as support for depth cameras via OpenNI 2 or support for patented algorithms. OpenCV's build system uses CMake for configuring the system and Visual Studio for compilation.

Before anything else, let's install Python. Go to `https://www.python.org/getit/` and download and run the most recent installer for Python 3.14. You probably want an installer for 64-bit Python, though OpenCV can work with 32-bit Python too.

Once Python has been installed, we can use `pip` to install NumPy and SciPy. Open the Command Prompt and run the following command:

```
> pip install numpy scipy
```

Now, we must decide whether we want a ready-made build of OpenCV (without support for depth cameras) or a custom build (with support for depth cameras). The next two subsections cover these alternatives.

Using a ready-made OpenCV package

OpenCV, including the `opencv_contrib` modules, can be installed as a `pip` package. This is as simple as running the following command:

```
> pip install opencv-contrib-python
```

You might find that this `pip` package offers all the OpenCV features you currently want. On the other hand, if you intend to use depth cameras or patented algorithms, or if you just want to learn about the general process of making a custom build of OpenCV, you should *not* install the OpenCV `pip` package; you should proceed to the next subsection instead.

Building OpenCV from source

If you want support for depth cameras, you should also install OpenNI 2, which is available as a set of precompiled binaries with an installation wizard. Then, we must build OpenCV from source using CMake and Visual Studio.

To obtain OpenNI 2, go to `https://structure.io/openni` and download the latest ZIP for Windows and your system's architecture (x64 or x86). Unzip it to get an installer file, such as `OpenNI-Windows-x64-2.2.msi`. Run the installer. After the installation is complete, log out and log back in (or, alternatively, reboot) so that changes to the environment variables can take effect.

Now, let's set up Visual Studio. To build OpenCV 5, we need Visual Studio 2017 or a later version. If you do not already have a suitable version, go to `https://visualstudio.microsoft.com/downloads/` and download and run one of the installers for one of the following:

- The free Visual Studio 2026 Community edition
- Any of the paid Visual Studio 2026 editions, which have a 30-day trial period

During installation, ensure that any optional C++ components have been selected. After the installation finishes, reboot.

For OpenCV 5, the build configuration process requires CMake 3 or a later version. Go to `https://cmake.org/download/`, download the installer for the latest version of CMake for your architecture (x64 or x86), and run it. During installation, select either **Add CMake to the system** `PATH` **for all users** or **Add CMake to the system** `PATH` **for the current user**.

At this stage, we have set up the dependencies and build environment for our custom build of OpenCV. Now, we need to obtain the OpenCV source code and configure and build it. We can do this by following these steps:

1. Go to `https://opencv.org/releases/` and get the latest OpenCV download for Windows. It is a self-extracting ZIP. Run it and, when prompted, enter any destination folder, which we will refer to as `<opencv_unzip_destination>`. During extraction, a subfolder is created at `<opencv_unzip_destination>\opencv`.

2. Go to `https://github.com/opencv/opencv_contrib/tags` and download the latest ZIP of the `opencv_contrib` modules. Unzip this file to any destination folder, which we will refer to as `<opencv_contrib_unzip_destination>`.
3. Open the Command Prompt and run the following command to make another folder where our build will go:

```
> mkdir <build_folder>
```

Change the directory to the `build` folder:

```
> cd <build_folder>
```

4. Now, we are ready to configure our build with CMake's command-line interface. To understand all the options, we can read the code in `<opencv_unzip_destination>\opencv\CMakeLists.txt`. However, for this book's purposes, we only need to use the options that will give us a release build with Python bindings, `opencv_contrib` modules, non-free content, and depth camera support via OpenNI 2. Some options differ slightly, depending on the Visual Studio version and target architecture (`x64` or `x86`). To create a `64`-bit (`x64`) solution for Visual Studio 2026, run the following command (but replace `<opencv_contrib_unzip_destination>` and `<opencv_unzip_destination>` with the actual paths):

```
> cmake -DCMAKE_BUILD_TYPE=RELEASE DWITH_OPENNI2=ON
-DOPENCV_EXTRA_MODULES_PATH="<opencv_contrib_unzip_destination>
/modules" -DOPENCV_ENABLE_NONFREE=ON -G "Visual Studio 18 2026" -A x64
"<opencv_unzip_destination>/opencv/sources"
```

Alternatively, to create a 32-bit (x86) solution for Visual Studio 2026, run the following command (but replace `<opencv_contrib_unzip_destination>` and `<opencv_unzip_destination>` with the actual paths):

```
> cmake -DCMAKE_BUILD_TYPE=RELEASE DWITH_OPENNI2=ON
-DOPENCV_EXTRA_MODULES_PATH="<opencv_contrib_unzip_destination>
/modules" -DOPENCV_ENABLE_NONFREE=ON -G "Visual Studio 18 2026" -A Win32
"<opencv_unzip_destination>/opencv/sources"
```

As the preceding command runs, it prints information about dependencies that are either found or missing. OpenCV has many optional dependencies, so do not panic (yet) about missing dependencies. However, if the build does not finish successfully, try installing the missing dependencies. (Many are available as prebuilt binaries.) Then, repeat this step.

5. CMake will have generated a Visual Studio solution file in `<opencv_build_folder>/OpenCV.sln`. Open it in Visual Studio. Ensure that the **Release** configuration (not the **Debug** configuration) is selected in the drop-down list in the toolbar near the top of the Visual Studio window. (OpenCV's Python bindings will probably fail to build in **Debug** configuration because most Python distributions do not contain debug libraries.) Go to the **BUILD** menu and select **Build Solution**. Watch the build messages in the **Output** pane at the bottom of the window, and wait for the build to finish.
6. By this stage, OpenCV has been built, but it hasn't been installed at a location where Python can find it. Before proceeding further, let's ensure that our Python environment does not already contain a conflicting build of OpenCV. Find and delete any OpenCV files in Python's `DLLs` folder and `site_packages` folder. For example, these files might match the following patterns:
   - `C:\Program Files\Python314\DLLs\opencv_*.dll`
   - `C:\Program Files\Python314\Lib\site-packages\opencv`
   - `C:\Program Files\Python314\Lib\site-packages\cv2.pyd`
7. Finally, let's install our custom build of OpenCV. For a 64-bit build, copy the following files into Python's `site_packages` folder:
   - `<build_folder>\lib\python3\Release\*.pyd`
   - `<build_folder>\bin\Release\*.dll`
   - `C:\Program Files\OpenNI2\Redist\OpenNI2.dll`
   - `C:\Program Files\OpenNI2\Redist\OpenNI2`: Copy the whole folder so as to create a subfolder, `site_packages\OpenNI2`

   Alternatively, for a 32-bit build, copy the following files into Python's `site_packages` folder:
   - `<build_folder>\lib\python3\Release\*.pyd`
   - `<build_folder>\bin\Release\*.dll`

- `C:\Program Files (x86)\OpenNI2\Redist\OpenNI2.dll`
- `C:\Program Files (x86)\OpenNI2\Redist\OpenNI2`: Copy the whole folder so as to create a subfolder, `site_packages\OpenNI2`.

Now, we have completed the OpenCV build process on Windows, and we have a custom build that is suitable for all of this book's Python projects.

Tip

In the future, if you want to update to a new version of the OpenCV source code, repeat all the preceding steps, starting from downloading OpenCV.

Installation on macOS

macOS comes with a preinstalled Python distribution that has been customized by Apple for the system's internal needs. To develop our own projects, we should make a separate Python installation to ensure that we do not conflict with the system's Python needs.

For macOS, there are several possible approaches for obtaining standard versions of Python 3, NumPy, SciPy, and OpenCV. All approaches ultimately require OpenCV to be compiled from source using the Xcode command-line tools. However, depending on the approach, this task is automated for us in various ways by third-party tools. We will look at this kind of approach using a package manager called Homebrew. A package manager can potentially do everything that CMake can, plus it helps us resolve dependencies and separate our development libraries from system libraries.

Tip

MacPorts is another popular package manager for macOS. However, at the time of writing, MacPorts does not offer packages for OpenCV 5 or OpenNI 2, so we will not use it in this book.

Before proceeding, let's make sure that the Xcode command-line tools are set up properly. Open a Terminal and run the following command:

```
$ xcode-select --install
```

Agree to the license agreement and any other prompts. The installation should run to completion. Now, we have the compilers that Homebrew requires.

Using Homebrew with ready-made packages

Starting on a system where Xcode and its command-line tools are already set up, the following steps will give us an OpenCV installation via Homebrew:

1. Open a Terminal and run the following command to install Homebrew:

```
$ /bin/bash -c "$(curl -fsSL )"
```

2. Homebrew does not automatically put its executables in `PATH`. To do so, create or edit the `~/.profile` file and add the following line at the top of the code if you are using an Apple Silicon Mac:

```
export PATH=/opt/homebrew/bin:/opt/homebrew/sbin:$PATH
```

   Alternatively, if you are using an Intel Mac, add the following line:

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

   Save the file and run this command to refresh `PATH`:

```
$ source ~/.profile
```

   Note that executables installed by Homebrew now take precedence over executables installed by the system.

3. For Homebrew's self-diagnostic report, run the following command:

```
$ brew doctor
```

   Follow any troubleshooting advice it gives.

4. Now, update Homebrew:

```
$ brew update
```

5. Run the following command to install Python 3.14:

```
$ brew install python
```

6. Now, we want to install OpenCV with the `opencv_contrib` modules. At the same time, we want to install dependencies such as NumPy. To accomplish this, run the following command:

```
$ brew install opencv
```

> **Note**
>
> Homebrew does not provide an option to install OpenCV with OpenNI 2 support. Homebrew always installs OpenCV with the `opencv_contrib` modules, including non-free content such as the patented SURF algorithm, which we will cover in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. If you intend to distribute software that depends on OpenCV's non-free content, you should do your own investigation of how the patent and licensing issues might apply in specific countries and to specific use cases.

7. Similarly, run the following command to install SciPy:

```
$ brew install scipy
```

Now, we have all we need to develop OpenCV projects with Python on macOS.

Using Homebrew with your own custom packages

Just in case you ever need to customize a package, Homebrew makes it easy to edit existing package definitions:

```
$ brew edit opencv
```

The package definitions are actually scripts in the Ruby programming language. Tips on editing them can be found in the Homebrew docs at `https://docs.brew.sh/Formula-Cookbook`. A script may specify Make or CMake configuration flags, among other things.

> **Tip**
>
> To see which CMake configuration flags are relevant to OpenCV, refer to `https://github.com/opencv/opencv/blob/5.x/CMakeLists.txt` in the official OpenCV repository on GitHub.

After making edits to the Ruby script, save it.

The customized package can be treated as normal. For example, it can be installed as follows:

```
$ brew install opencv
```

Installation on Debian, Ubuntu, Linux Mint, and similar systems

Debian, Ubuntu, Linux Mint, and related Linux distributions use the `apt` package manager. On these systems, it is easy to install packages for Python 3 and many Python modules, including NumPy and SciPy. An OpenCV package is also available via `apt`, but at the time of writing, this package has not been updated to OpenCV 5. Instead, we can obtain OpenCV 5 (without support for depth cameras) from Python's standard package manager, `pip`. Alternatively, we can build OpenCV 5 from source. When built from source, OpenCV can support depth cameras via OpenNI 2, which is available as a set of precompiled binaries with an installation script.

Regardless of our approach to obtaining OpenCV, let's begin by updating `apt` so that we can obtain the latest packages. Open a Terminal and run the following command:

```
$ sudo apt-get update
```

Having updated `apt`, let's run the following command to install NumPy and SciPy for Python 3:

```
$ sudo apt-get install python3-numpy python3-scipy
```

Note

Equivalently, we could have used the Ubuntu Software Center, which is the `apt` package manager's graphical frontend.

Now, we must decide whether we want a ready-made build of OpenCV (without support for depth cameras or patented algorithms) or a custom build (with support for depth cameras and patented algorithms). The next two subsections cover these alternatives.

Using a ready-made OpenCV package

OpenCV, including the `opencv_contrib` modules, can be installed as a `pip` package. This is as simple as running the following command:

```
$ pip3 install opencv-contrib-python
```

Tip

On some Linux systems, `pip` is only supported within a Python `venv` virtual environment, and you will receive an error message to this effect when you try to run `pip`. (This restriction ensures separation between the system's Python packages and the user's Python packages.) If this is the case on your system, first use `venv` to set up a virtual environment and then, in your virtual environment, install the necessary packages with `pip` as follows:

```
$ pip3 install numpy scipy opencv-contrib-python
```

You might find that this `pip` package offers all the OpenCV features you currently want. On the other hand, if you intend to use depth cameras or patented algorithms, or if you just want to learn about the general process of making a custom build of OpenCV, you should *not* install the OpenCV `pip` package; you should proceed to the next subsection instead.

Building OpenCV from source

To build OpenCV from source, we need a C++ build environment and the CMake build configuration system. Specifically, we need CMake 3. Run the following commands to ensure that the necessary versions of CMake and other build tools are installed:

```
$ sudo apt-get install build-essential cmake pkg-config
```

As part of the build process for OpenCV, CMake will need to access the internet to download additional dependencies. If your system uses a proxy server, ensure that your environment variables for the proxy server have been configured properly. Specifically, CMake relies on the `http_proxy` and `https_proxy` environment variables. To define them, you can edit your `~/.bash_profile` script and add lines such as the following (but modify them so that they match your own proxy URLs and port numbers):

```
export http_proxy=http://myproxy.com:8080
export https_proxy=http://myproxy.com:8081
```

Note

If you are unsure whether your system uses a proxy server, it probably doesn't, so you can ignore this step.

To build OpenCV's Python bindings, we need an installation of the Python 3 development headers. To install them, run the following command:

```
$ sudo apt-get install python3-dev
```

To capture frames from typical USB webcams, OpenCV depends on **Video for Linux** (**V4L**). On most systems, V4L comes preinstalled, but just in case it is missing, run the following command:

```
$ sudo apt-get install libv4l-dev
```

As we mentioned previously, to support depth cameras, OpenCV depends on OpenNI 2. Go to `https://structure.io/openni` and download the latest ZIP of OpenNI 2 for Linux and your system's architecture (`x64`, `x86`, or ARM). Unzip it to any destination, which we will refer to as `<openni2_unzip_destination>`. Run the following commands:

```
$ cd <openni2_unzip_destination>
$ sudo ./install.sh
```

The preceding installation script configures the system so that it supports depth cameras as USB devices.

Now that we have the build environment and dependencies installed, we can obtain and build the OpenCV source code. To do so, follow these steps:

1. Go to `https://opencv.org/releases/` and download the latest source package. Unzip it to any destination folder, which we will refer to as `<opencv_unzip_destination>`.
2. Go to `https://github.com/opencv/opencv_contrib/tags` and download the latest source package for the `opencv_contrib` modules. Unzip it to any destination folder, which we will refer to as `<opencv_contrib_unzip_destination>`.
3. Open a Terminal. Run the following commands to create a directory where we will put our OpenCV build:

   ```
   $ mkdir <build_folder>
   ```

   Change into the newly created directory:

   ```
   $ cd <build_folder>
   ```

4. Run the following commands to define temporary environment variables that will help CMake locate the OpenNI 2 headers and binaries:

```
$ export OPENNI2_INCLUDE=<openni2_unzip_destination>/Include
$ export OPENNI2_REDIST=<openni2_unzip_destination>/Redist
```

5. Now, we can use CMake to generate a build configuration for OpenCV. The output of this configuration process will be a set of Makefiles, which are scripts we can use to build and install OpenCV. A complete set of CMake configuration options for OpenCV is defined in the `<opencv_unzip_destination>/opencv/sources/CMakeLists.txt` file. For our purposes, we care about the options that relate to OpenNI 2 support, Python bindings, `opencv_contrib` modules, and non-free content. Configure OpenCV by running the following command:

```
$ cmake -D CMAKE_BUILD_TYPE=RELEASE WITH_OPENNI2=ON
OPENCV_EXTRA_MODULES_PATH=<opencv_contrib_unzip_destination> -D
OPENCV_ENABLE_NONFREE=ON <opencv_unzip_destination>
```

6. Finally, run the following commands to interpret our newly generated Makefiles and thereby build and install OpenCV:

```
$ make -j8
$ sudo make install
```

So far, we have completed the OpenCV build process on Debian, Ubuntu, or a similar system, and we have a custom build that is suitable for all of this book's Python projects.

Installation on Arch Linux, Manjaro, and similar systems

Arch Linux, Manjaro, and related Linux distributions use the `pacman` package manager. On these systems, it is easy to install packages for Python 3 and many Python modules, including NumPy and SciPy. An OpenCV package is also available via `pacman`, and it is up-to-date with OpenCV 5. As an alternative to using the `pacman` OpenCV package, we can build OpenCV 5 from source. When built from source, OpenCV can support depth cameras via OpenNI 2, which is available as a set of precompiled binaries with an installation script.

Regardless of our approach to obtaining OpenCV, let's begin by updating pacman and the whole system. Open a Terminal and run the following command:

```
$ sudo pacman -Syyu
```

Having updated the system, let's run the following command to install NumPy and SciPy for Python 3:

```
$ sudo pacman -S python-numpy python-scipy
```

> **Note**
>
> Equivalently, on Manjaro, we could have used Pamac (also known as Add/Remove Software), which is a package manager with a graphical frontend.

Now, we must decide whether we want a ready-made build of OpenCV (without support for depth cameras or patented algorithms) or a custom build (with support for depth cameras and patented algorithms). The next two subsections cover these alternatives.

Using a ready-made OpenCV package

OpenCV, including the `opencv_contrib` modules, can be installed as a `pacman` package. This is as simple as running the following command:

```
$ sudo pacman -S opencv
```

You might find that this `pacman` package offers all the OpenCV features you currently want. On the other hand, if you intend to use depth cameras or patented algorithms, or if you just want to learn about the general process of making a custom build of OpenCV, you should *not* install the OpenCV `pacman` package; you should proceed to the next subsection instead.

Building OpenCV from source

To build OpenCV from source, we need a C++ build environment and the CMake build configuration system. Specifically, we need CMake 3. Run the following commands to ensure that the necessary versions of CMake and other build tools are installed:

```
$ sudo pacman -S base-devel cmake pkgconfig
```

As part of the build process for OpenCV, CMake will need to access the internet to download additional dependencies. If your system uses a proxy server, ensure that your environment variables for the proxy server have been configured properly. Specifically, CMake relies on the `http_proxy` and `https_proxy` environment variables. To define them, you can edit your

`~/.bash_profile` script and add lines such as the following (but modify them so that they match your own proxy URLs and port numbers):

```
export http_proxy=http://myproxy.com:8080
export https_proxy=http://myproxy.com:8081
```

Note

If you are unsure whether your system uses a proxy server, it probably doesn't, so you can ignore this step.

To capture frames from typical USB webcams, OpenCV depends on **Video for Linux** (**V4L**). On most systems, V4L comes preinstalled, but just in case it is missing, run the following command:

```
$ sudo pacman -S v4l-utils
```

As we mentioned previously, to support depth cameras, OpenCV depends on OpenNI 2. Go to `https://structure.io/openni` and download the latest ZIP of OpenNI 2 for Linux and your system's architecture (`x64`, `x86`, or ARM). Unzip it to any destination, which we will refer to as `<openni2_unzip_destination>`. Run the following commands:

```
$ cd <openni2_unzip_destination>
$ sudo ./install.sh
```

The preceding installation script configures the system so that it supports depth cameras as USB devices.

Now that we have the build environment and dependencies installed, we can obtain and build the OpenCV source code. To do so, follow these steps:

1. Go to `https://opencv.org/releases/` and download the latest source package. Unzip it to any destination folder, which we will refer to as `<opencv_unzip_destination>`.
2. Go to `https://github.com/opencv/opencv_contrib/tags` and download the latest source package for the `opencv_contrib` modules. Unzip it to any destination folder, which we will refer to as `<opencv_contrib_unzip_destination>`.

3. Open a Terminal. Run the following commands to create a directory where we will put our OpenCV build:

```
$ mkdir <build_folder>
```

   Change into the newly created directory:

```
$ cd <build_folder>
```

4. Run the following commands to define temporary environment variables that will help CMake locate the OpenNI 2 headers and binaries:

```
$ export OPENNI2_INCLUDE=<openni2_unzip_destination>/Include
$ export OPENNI2_REDIST=<openni2_unzip_destination>/Redist
```

5. Now, we can use CMake to generate a build configuration for OpenCV. The output of this configuration process will be a set of Makefiles, which are scripts we can use to build and install OpenCV. A complete set of CMake configuration options for OpenCV is defined in the `<opencv_unzip_destination>/opencv/sources/CMakeLists.txt` file. For our purposes, we care about the options that relate to OpenNI 2 support, Python bindings, `opencv_contrib` modules, and non-free content. Configure OpenCV by running the following command:

```
$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D WITH_OPENNI2=ON -D
OPENCV_EXTRA_MODULES_PATH=<opencv_contrib_unzip_destination> -D
OPENCV_ENABLE_NONFREE=ON <opencv_unzip_destination>
```

6. Now, run the following commands to interpret our newly generated Makefiles, and thereby build and install OpenCV:

```
$ make -j8
$ sudo make install
```

7. A final step is necessary because OpenCV's installer uses a different Python `site-packages` path than the standard one for Arch Linux or Manjaro. To resolve this difference, create a link as follows:

```
$ sudo ln -s /usr/local/lib/python3.14/site-packages/cv2 /usr/lib/
python3.14/site-packages/cv2
```

At this point, we have completed the OpenCV build process on Arch Linux, Manjaro, or a similar system, and we have a custom build that is suitable for all of this book's Python projects.

Installation on other Unix-like systems

On other Unix-like systems, the package manager and available packages may differ. Consult your package manager's documentation and search for packages with `opencv` in their names. Remember that OpenCV and its Python bindings might be split into multiple packages.

Also, look for any installation notes that have been published by the system provider, the repository maintainer, or the community. Since OpenCV uses camera drivers and media codecs, getting all of its functionality to work can be tricky on systems with poor multimedia support. Under some circumstances, system packages might need to be reconfigured or reinstalled for compatibility.

If packages are available for OpenCV, check their version number. OpenCV 5 is recommended for this book's purposes. Also, check whether the packages offer Python bindings, depth camera support via OpenNI 2, and implementations of the non-free algorithms. Finally, check whether anyone in the developer community has reported success or failure when using the packages.

If, instead, you want to do a custom build of OpenCV from source, it might be helpful to refer to the previous sections' steps for Debian or Arch Linux and adapt these steps to the package manager and packages that are present on another system.

Running the official sample code

Running a few sample scripts is a good way to test whether OpenCV has been set up correctly. Some samples are included in OpenCV's source code archive. If you have not already obtained the source code, go to `https://opencv.org/releases/` and download one of the following archives:

- For Windows, download the latest archive, labeled **Windows**. It is a self-extracting ZIP. Run it and, when prompted, enter any destination folder, which we will refer to as `<opencv_unzip_destination>`. Find the Python samples in `<opencv_unzip_destination>/opencv/sources/samples/python`.
- For other systems, download the latest archive, labeled **Sources**. It is a ZIP file. Unzip it to any destination folder, which we will refer to as `<opencv_unzip_destination>`. Find the Python samples in `<opencv_unzip_destination>/samples/python`.

Some of the sample scripts require command-line arguments. However, the following scripts (among others) should work without any arguments:

- `hist.py`: This script displays a photo. Press `A`, `B`, `C`, `D`, or `E` to see variations of the photo, along with a corresponding histogram of color or grayscale values.
- `opt_flow.py`: This script displays a webcam feed with a superimposed visualization of the optical flow or, in other words, the direction of motion. Slowly wave your hand at the webcam to see the effect. Press `1` or `2` for alternative visualizations.

To exit a script, press `Esc` (not the window's close button).

If we encounter the `ImportError: No module named cv2` message, then this means that we are running the script from a Python installation that does not know anything about OpenCV. There are two possible explanations for this:

- Some steps in the OpenCV installation might have failed or been missed. Go back and review the steps.
- If we have multiple Python installations on the machine, we might be using the wrong version of Python to launch the script. For example, on macOS, it might be the case that OpenCV has been installed for Homebrew Python, but we are running the script with the system's version of Python. Go back and review the installation steps about editing the system's `PATH` variable. Also, try launching the script manually from the command line using commands such as the following:

  ```
  $ python hist.py
  ```

  You can also try the following command:

  ```
  $ python3.14 python/camera.py
  ```

As another possible means of selecting a different Python installation, try editing the sample script to remove the `#!` lines. These lines might explicitly associate the script with the wrong Python installation (for our particular setup).

Finding documentation, help, and updates

OpenCV's documentation can be found at `http://docs.opencv.org/5.x`, where you can either read it online or download it for offline reading. If you write code on airplanes or other places without internet access, you will definitely want to keep offline copies of the documentation.

The documentation includes a combined API reference for OpenCV's C++ API and its Python API. When you look up a class or function, be sure to read the section under the heading **Python**.

Note

OpenCV's Python module is named `cv2`. The **2** in `cv2` has nothing to do with the version number of OpenCV; we really are using OpenCV 5. Historically, OpenCV had a `cv` Python module that wrapped a now-obsolete C version of the library. The `cv` module was removed in OpenCV 4.

Still, in the official OpenCV 5 samples and in some other authors' samples, you might see valid Python code such as the following, which imports the `cv2` module and gives it a local alias, `cv`:

```
import cv2 as cv
```

Here, there is nothing special about the alias `cv`; countless other aliases would also be valid, and the choosing of an alias will have no effect on the module's version or functionality. For example, the alias `george_eliot_42` would be valid, though not sensible:

```
import cv2 as george_eliot_42
```

You might even observe that the OpenCV 5 documentation follows an unstated convention of using `cv` as an alias for `cv2`. Despite all of this, remember that in OpenCV 5, the Python module's real name is `cv2`.

If the documentation does not seem to answer your questions, try talking to the OpenCV community. Here are some sites where you will find helpful people:

- The OpenCV forum: `https://forum.opencv.org/`
- Adrian Rosebrock's website: `https://www.pyimagesearch.com/`
- Joseph Howse's website for his books and presentations: `https://nummist.com/opencv/`

Lastly, if you are an advanced user who wants to try new features, bug fixes, and sample scripts from the latest (unstable) OpenCV source code, have a look at the project's repository at `https://github.com/opencv/opencv/`.

Finding this book's sample code

We will walk through a set of relevant code samples in each chapter of this book. For now, let's just take note of the samples' location and format.

All of this book's samples can be found online in the Git repository. The contents are organized into various folders by chapter. For some chapters, an interactively editable version of the code is provided as a Jupyter notebook. For example, `Chapter 5` has a Jupyter notebook located in the repository at `chapter05/chapter05.ipynb`.

> **Note**
>
> If you are new to Git or want to expand your Git skills, take a look at the free ebook *Pro Git*, by Scott Chacon and Ben Straub. You can find it online at `https://git-scm.com/book/`. The book's second chapter, *Git Basics*, is especially helpful for beginners.

Typically, when you start studying a chapter, you might want to pull down the latest code from GitHub, just in case the authors have made any recent corrections or updates. Also, if the chapter has a Jupyter notebook, you might want to open it online via the Google Colab website, which enables you to experiment with your own modifications to the code in a ready-made environment, without the need to set up or save anything on your local machine. Where applicable, the chapter's *Technical requirements* section will specify a Google Colab URL where you can open the chapter's Jupyter notebook.

To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Advanced readers might want to experiment with some of the book's sample code right away. Otherwise, the rest of this book will be your guide!

Summary

By now, we should have an OpenCV installation that will serve our needs for the diverse projects described in this book. Depending on which approach we took, we may also have a set of tools and scripts that can be used to reconfigure and rebuild OpenCV for our future needs.

We also know where to find this book's sample code and OpenCV's official Python samples. The official samples cover a different range of functionalities outside this book's scope, but they are useful as additional learning aids.

In the next chapter, we will familiarize ourselves with the most basic functions of the OpenCV API, namely displaying images and videos, capturing videos through a webcam, and handling basic keyboard and mouse inputs.

2

Handling Files, Cameras, and GUIs

Installing OpenCV and running samples is fun, but at this stage, we want to try things out in our own way. This chapter introduces OpenCV's I/O functionality. We also discuss the concept of a project and the beginnings of an object-oriented design for this project, which we will flesh out in subsequent chapters.

By starting with a look at I/O capabilities and design patterns, we will build our project in the same way we would make a sandwich: from the outside in. Bread slices and spread, or endpoints and glue, come before fillings or algorithms. We choose this approach because computer vision is mostly extroverted; it contemplates the real world outside our computer, and we want to apply all of our subsequent algorithmic work to the real world through a common interface.

Specifically, in this chapter, our code samples and discussions will cover the following tasks:

- Reading images from image files, video files, camera devices, or raw bytes of data in memory
- Writing images to image files or video files
- Manipulating image data in NumPy arrays
- Converting between NumPy arrays and OpenCV's `UMat` format, which supports OpenCL acceleration
- Enabling hardware acceleration of video I/O
- Sending a video stream over a network
- Displaying images in windows

- Handling keyboard and mouse input
- Implementing an application with an object-oriented design

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Please refer to `Chapter 1, Setting Up OpenCV`, for installation instructions.

The complete code for this chapter can be found in this book's GitHub repository in the `Chapter02` folder. You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Basic I/O scripts

Most CV applications need to get images as input. Most also produce images as output. An interactive CV application might require a camera as an input source and a window as an output destination. However, other possible sources and destinations include image files, video files, and raw bytes. For example, raw bytes might be transmitted via a network connection, or they might be generated by an algorithm if we incorporate procedural graphics into our application. Let's look at each of these possibilities.

Reading/writing an image file

OpenCV provides the `imread` function to load an image from a file and the `imwrite` function to write an image to a file. These functions support various file formats for still images (not videos). The supported formats vary, as formats can be added or removed in a custom build of OpenCV, but normally BMP, PNG, JPEG, and TIFF are among the supported formats.

Let's explore the anatomy of the representation of an image in OpenCV and NumPy. An image is a multidimensional array; it has columns and rows of pixels, and each pixel has a value. For different kinds of image data, the pixel value may be formatted in different ways. For example, we can create a 3x3 square black image from scratch by simply creating a 2D NumPy array:

```
img = numpy.zeros((3, 3), dtype=numpy.uint8)
```

If we print this image to a console, we obtain the following result:

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]], dtype=uint8)
```

Here, each pixel is represented by a single 8-bit integer, which means that the values for each pixel are in the 0-255 range, where 0 is black, 255 is white, and the in-between values are shades of gray. This is a grayscale image.

Let's now convert this image into **blue-green-red (BGR)** format using the `cv2.cvtColor` function:

```
img = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
```

Let's observe how the image has changed:

```
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=uint8)
```

As you can see, each pixel is now represented by a three-element array, with each integer representing one of the three color channels: B, G, and R. Other common color models, such as **hue-saturation-value (HSV)**, will be represented in the same way, albeit with different value ranges. For example, the hue value of the HSV color model has a range of 0-180.

> **Note**
>
> For more information about color models, refer to `Chapter 3, Processing Images with OpenCV`, specifically the `Converting between different color models` section.

You can check the structure of an image by inspecting the `shape` property, which returns rows, columns, and the number of channels (if there is more than one).

Consider this example:

```
img = numpy.zeros((5, 3), dtype=numpy.uint8)
print(img.shape)
```

The preceding code will print `(5, 3)`; in other words, we have a grayscale image with 5 rows and 3 columns. If you then converted the image into BGR, the shape would be `(5, 3, 3)`, which indicates the presence of three channels per pixel.

Images can be loaded from one file format and saved to another. For example, let's convert an image from PNG into JPEG:

```
import cv2
import sys

image = cv2.imread('MyPic.png')
if image is None:
    print('Failed to read image from file')
    sys.exit(1)

success = cv2.imwrite('MyPic.jpg', image)
if not success:
    print('Failed to write image to file')
    sys.exit(1)
```

Note

OpenCV's Python module is called `cv2` even though we are using OpenCV 5.x and not OpenCV 2.x. Historically, OpenCV had two Python modules: `cv2` and `cv`. The latter wrapped a legacy version of OpenCV implemented in C. Nowadays, OpenCV has only the `cv2` Python module, which wraps the current version of OpenCV implemented in C++.

If `imread` fails to read the specified image file (for example, if the file does not exist or does not match any supported image file format), then `imread` returns `None`; otherwise, it returns the image as a NumPy array. If `imwrite` fails to write the specified image file, `imwrite` returns `False`; otherwise, it returns `True`.

By default, `imread` returns an image in the BGR color format even if the file uses a grayscale format. BGR represents the same color model as the widely used **red-green-blue (RGB)**, but the byte order is reversed. If you are using OpenCV functions for all your I/O needs, you do not need to worry about byte order differences (OpenCV will handle these internally for you), but bear in mind that other file I/O libraries or GUI libraries may expect image data to be in RGB

order instead of BGR. Then, to reverse the byte order, you can use `cv2.cvtColor` with the `cv2.COLOR_BGR2RGB` flag or the `cv2.COLOR_RGB2BGR` flag.

Optionally, we may specify the mode of `imread`. The supported options include the following:

- `cv2.IMREAD_COLOR`: This is the default option, providing a 3-channel BGR image with an 8-bit value (0-255) for each channel.
- `cv2.IMREAD_GRAYSCALE`: This provides an 8-bit grayscale image.
- `cv2.IMREAD_ANYCOLOR`: This provides either an 8-bit-per-channel BGR image or an 8-bit grayscale image, depending on the metadata in the file.
- `cv2.IMREAD_UNCHANGED`: This reads all of the image data, including the alpha or transparency channel (if there is one) as a fourth channel.
- `cv2.IMREAD_ANYDEPTH`: This loads an image in grayscale at its original bit depth. For example, it provides a 16-bit-per-channel grayscale image if the file represents an image in this format.
- `cv2.IMREAD_ANYDEPTH | cv2.IMREAD_COLOR`: This combination loads an image in BGR color at its original bit depth.
- `cv2.IMREAD_REDUCED_GRAYSCALE_2`: This loads an image in grayscale at half its original resolution. For example, if the file contains a 640 x 480 image, it is loaded as a 320 x 240 image.
- `cv2.IMREAD_REDUCED_COLOR_2`: This loads an image in 8-bit-per-channel BGR color at half its original resolution.
- `cv2.IMREAD_REDUCED_GRAYSCALE_4`: This loads an image in grayscale at one-quarter of its original resolution.
- `cv2.IMREAD_REDUCED_COLOR_4`: This loads an image in 8-bit-per-channel color at one-quarter of its original resolution.
- `cv2.IMREAD_REDUCED_GRAYSCALE_8`: This loads an image in grayscale at one-eighth of its original resolution.
- `cv2.IMREAD_REDUCED_COLOR_8`: This loads an image in 8-bit-per-channel color at one-eighth of its original resolution.

As an example, let's load a PNG file as a grayscale image (losing any color information in the process), and then save it as a grayscale PNG image:

```
import cv2
import sys
```

```
grayImage = cv2.imread('MyPic.png', cv2.IMREAD_GRAYSCALE)
if grayImage is None:
    print('Failed to read image from file')
    sys.exit(1)

success = cv2.imwrite('MyPicGray.png', grayImage)
if not success:
    print('Failed to write image to file')
    sys.exit(1)
```

The path of an image, unless absolute, is relative to the working directory (which is usually the path from which the Python script is run), so, in the preceding example, `MyPic.png` would have to be in the working directory or the image would not be found. If you prefer to avoid assumptions about the working directory, you can use absolute paths, such as `"C:\\Users\\Joe\\Pictures\\MyPic.png"` on Windows, `"/Users/Joe/Pictures/MyPic.png"` on Mac, or `"/home/joe/pictures/MyPic.png"` on Linux.

The `imwrite` function requires an image to be in the BGR or grayscale format with a certain number of bits per channel that the output format can support. For example, the BMP file format requires 8 bits per channel, while PNG allows either 8 or 16 bits per channel.

Some image file formats, notably **multipage TIFF**, can contain multiple images in one file. OpenCV can work with these kinds of files via the `imreadmulti` and `imwritemulti` functions, as seen in the following example:

```
import cv2
import sys

success, grayImages = cv2.imreadmulti(
    'MyMultiPics.tiff', flags=cv2.IMREAD_GRAYSCALE)
if not success:
    print('Failed to read images from file')
    sys.exit(1)
print('Number of images:', len(grayImages))

success = cv2.imwritemulti('MyMultiPicsGray.tiff', grayImages)
if not success:
    print('Failed to write images to file')
    sys.exit(1)
```

Here, note that `grayImages` is a tuple of NumPy arrays; each element in the tuple represents a single image.

Converting between an image and raw bytes

Conceptually, a byte is an integer ranging from 0 to 255. Throughout real-time graphic applications today, a pixel is typically represented by one byte per channel, though other representations are also possible.

An OpenCV image is a 2D or 3D array of the `numpy.ndarray` type. Typically, in NumPy code, a `numpy.ndarray` object is created with the `numpy.array` function. An 8-bit grayscale image is a 2D array containing byte values. A 24-bit BGR image is a 3D array, which also contains byte values. We may access these values by using an expression such as `image[0, 0]` or `image[0, 0, 0]`. The first index is the pixel's `y` coordinate or row, `0` being the top. The second index is the pixel's `x` coordinate or column, `0` being the leftmost. The third index (if applicable) represents a color channel. The array's three dimensions can be visualized in the following Cartesian coordinate system:

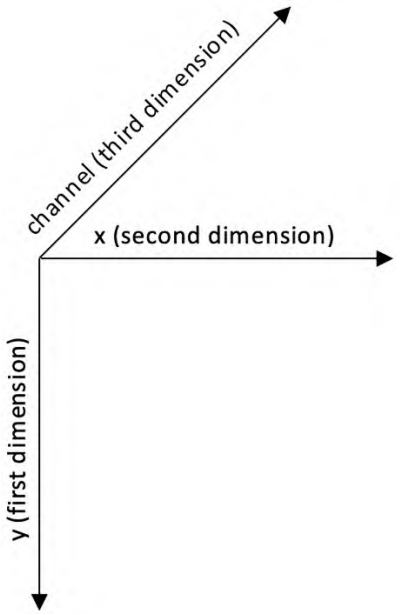


Figure 2.1: The dimensions of an image as a NumPy array

For example, in an 8-bit grayscale image with a white pixel in the upper-left corner, `image[0, 0]` is `255`. For a 24-bit (8-bit-per-channel) BGR image with a blue pixel in the upper-left corner, `image[0, 0]` is `[255, 0, 0]`.

Provided that an image has 8 bits per channel, we can cast it to a standard Python `bytearray` object, which is one-dimensional:

```
byteArray = bytearray(image)
```

Conversely, provided that `bytearray` contains bytes in an appropriate order, we can cast and then reshape it to get a `numpy.ndarray` type that is an image:

```
grayImage = numpy.array(grayByteArray).reshape(height, width)
bgrImage = numpy.array(bgrByteArray).reshape(height, width, 3)
```

As a more complete example, let's convert a `bytearray` that contains random bytes into a grayscale image and a BGR image:

```
import cv2
import numpy
import os

# Make an array of 120,000 random bytes.
randomByteArray = bytearray(os.urandom(120000))
flatNumpyArray = numpy.array(randomByteArray)

# Convert the array to make a 400x300 grayscale image.
grayImage = flatNumpyArray.reshape(300, 400)
cv2.imwrite('RandomGray.png', grayImage)

# Convert the array to make a 400x100 color image.
bgrImage = flatNumpyArray.reshape(100, 400, 3)
cv2.imwrite('RandomColor.png', bgrImage)
```

Note

Here, we use Python's standard `os.urandom` function to generate random raw bytes, which we then convert into a NumPy array. Note that it is also possible to generate a random NumPy array directly (and more efficiently) using a statement such as `numpy.random.randint(0, 256, 120000).reshape(300, 400)`. The only reason we use `os.urandom` is to help to demonstrate conversion from raw bytes.

After running this script, we should have a pair of randomly generated images, `RandomGray.png` and `RandomColor.png`, in the script's directory.

Here is an example of `RandomGray.png` (though yours will almost certainly differ since it is random):

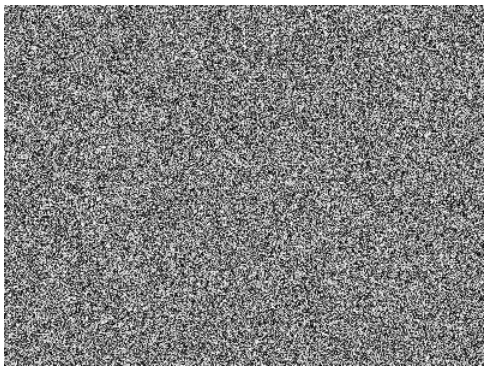


Figure 2.2a: The result of interpreting 120,000 random bytes as a 400x300 grayscale image

Similarly, here is an example of `RandomColor.png`:

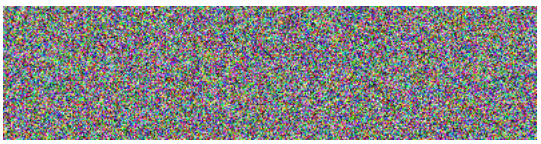


Figure 2.2b: The result of interpreting the same 120,000 random bytes as a 400x100 BGR color image

Now that we have a better understanding of how an image is formed from data, we can start performing basic operations on it.

Accessing image data with numpy.ndarray

We already know that the easiest (and most common) way to load an image in OpenCV is to use the `imread` function. We also know that this will return an image, which is really an array (either a 2D or 3D one, depending on the parameters you passed to `imread`).

The `numpy.ndarray` class is greatly optimized for array operations, and it allows certain kinds of bulk manipulations that are not available in a plain Python list. These kinds of `numpy.ndarray` type-specific operations come in handy for image manipulations in OpenCV. However, let's explore image manipulations step by step, starting with a basic example. Say

you want to manipulate a pixel at coordinates (0, 0) in a BGR image and turn it into a white pixel:

```
import cv2

img = cv2.imread('MyPic.png')
img[0, 0] = [255, 255, 255]
```

If you then save the modified image to file and view it, you will see a white dot in the top-left corner of the image. Naturally, this modification is not very useful, but it begins to show the possibilities. Now, let's leverage the capabilities of `numpy.ndarray` to perform transformations on an array much faster than we could do with a plain Python list.

Let's say that you want to change the blue value of a particular pixel, say, the pixel at coordinates (150, 120). The `numpy.ndarray` type provides a handy method, `item`, that takes three parameters: the `x` (or left) position, the `y` (or top) position, and the index within the array at the (`x`, `y`) position (remember that in a BGR image, the data at a certain position is a three-element array containing the B, G, and R values in this order) and returns the value at the index position. Another method, `itemset`, sets the value of a particular channel of a particular pixel to a specified value. `itemset` takes two arguments: a three-element tuple (`x`, `y`, and index) and the new value.

In the following example, we change the value of the blue channel at (150, 120) from its current value to an arbitrary `255`:

```
import cv2

img = cv2.imread('MyPic.png')
img.itemset((150, 120, 0), 255)  # Sets the value of a pixel's blue channel
print(img.item(150, 120, 0))  # Prints the value of a pixel's blue channel
```

For modifying a single element in an array, the `itemset` method is somewhat faster than the indexing syntax that we saw in the first example in this section.

Again, modifying an element of an array does not do much in itself, but it does open a world of possibilities. However, for performance reasons, this is only suitable for small regions of interest. When you need to manipulate an entire image or a large region of interest, it is advisable that you utilize either OpenCV's functions or NumPy's **array slicing**. The latter allows you to specify a range of indices. Let's consider an example of using array slicing to

manipulate color channels. Setting all G (green) values of an image to `0` is as simple as the following code:

```
import cv2

img = cv2.imread('MyPic.png')
img[:, :, 1] = 0
```

This piece of code performs a fairly significant operation and is easy to understand. The relevant line is the last one, which basically instructs the program to take all pixels from all rows and columns and set the green value (at index one of the three-element BGR array) to `0`. If you display this image, you will notice a complete absence of green.

There are several interesting things we can do by accessing raw pixels with NumPy's array slicing; one of them is defining a **region of interest** (**ROI**). An ROI is simply a selected rectangular subsection of an image; we may process or modify this subsection without affecting the rest of the image. Once the region is defined, we can perform a number of operations. For example, we can bind this region to a variable, then define a second region, and finally assign the value of the first region to the second (hence copying a portion of the image over to another position in the image):

```
import cv2

img = cv2.imread('MyPic.png')
my_roi = img[0:100, 0:100]
img[300:400, 300:400] = my_roi
```

It is important to make sure that the two regions correspond in terms of size. If not, NumPy will (rightly) complain that the two shapes are mismatched.

Finally, we can access the properties of `numpy.ndarray`, as shown in the following code:

```
import cv2

img = cv2.imread('MyPic.png')
print(img.shape)
print(img.size)
print(img.dtype)
```

These three properties are defined as follows:

- `shape`: This is a tuple describing the `shape` of the array. For an image, it contains (in order) the height, width, and—if the image is in color—the number of channels. The length of the `shape` tuple is a useful way to determine whether an image is grayscale or color. For a grayscale image, we have `len(shape) == 2`, and for a color image, `len(shape) == 3`.
- `size`: This is the number of elements in the array. In the case of a grayscale image, this is the same as the number of pixels. In the case of a BGR image, it is three times the number of pixels because each pixel is represented by three elements (B, G, and R).
- `dtype`: This is the datatype of the array's elements. For an 8-bit-per-channel image, the datatype is `numpy.uint8`.

All in all, it is strongly advised that you familiarize yourself with NumPy in general, and `numpy.ndarray` in particular, when working with OpenCV. This class is the foundation of OpenCV's Python interface.

Converting between numpy.ndarray and cv2.UMat

A NumPy array keeps its data in main memory – never in GPU memory. This means that OpenCV cannot apply its optional GPU optimizations to images that are represented as NumPy arrays. If we want to allow OpenCV to move data into GPU memory and apply GPU optimizations (where supported), then we can convert a NumPy array to a different format called an **OpenCV Universal Matrix** or `cv2.UMat`.

> **Note**
>
> Currently, `cv2.UMat` supports GPU acceleration on systems that have OpenCL-compliant GPU drivers, with a minimum version of OpenCL 1.2. Most modern desktop or laptop GPUs, as well as some high-end smartphone or tablet GPUs, have this level of OpenCL support.

The following code initializes a `UMat` by copying data from a NumPy array:

```
my_umat = cv2.UMat(my_numpy_array)
```

Most OpenCV functions will accept a `UMat` argument in place of a NumPy array and, then, will return a `UMat` instead of a NumPy array. Here is an example:

```
my_converted_umat = cv2.cvtColor(my_umat, cv2.COLOR_GRAY2BGR)
```

NumPy functions will not accept a `UMat` argument, a `UMat` does not have the same methods as a `numpy.ndarray`, and we cannot perform slicing or other Pythonic operations on `UMat` data. However, we can convert a `UMat` back to a NumPy array, as follows:

```
my_converted_numpy_array = my_converted_umat.get()
```

You might wonder, should you start using `UMat` everywhere in order to speed up your applications? Probably not. Copying data into and out of GPU memory is an expensive operation and, in many cases, this overhead cost outweighs the benefit of greater parallelism on the GPU. On the other hand, if you can minimize the number of copies and if you can apply a series of OpenCV functions to a large amount of data on the GPU, then the benefit may outweigh the cost.

Throughout this book, we will mainly use `numpy.ndarray` instead of `cv2.UMat`, but bear in mind that you may want to test the performance of `cv2.UMat` as an alternative for your own practical applications.

Reading/writing a video file

OpenCV provides the `VideoCapture` and `VideoWriter` classes, which support various video file formats. The supported formats vary depending on the operating system and the build configuration of OpenCV, but normally it is safe to assume that the AVI format is supported. Via its `read` method, a `VideoCapture` object may be polled for new frames until it reaches the end of its video file. Each frame is an image in the BGR format.

Conversely, an image may be passed to the `write` method of the `VideoWriter` class, which appends the image to a file using `VideoWriter`. Let's look at an example that reads frames from one AVI file and writes them to another with a YUV encoding:

```
import cv2

videoCapture = cv2.VideoCapture('MyInputVid.avi')
fps = videoCapture.get(cv2.CAP_PROP_FPS)
size = (
    int(videoCapture.get(cv2.CAP_PROP_FRAME_WIDTH)),
    int(videoCapture.get(cv2.CAP_PROP_FRAME_HEIGHT))
)
videoWriter = cv2.VideoWriter(
    'MyOutputVid.avi', cv2.VideoWriter_fourcc('I','4','2','0'), fps, size
)
```

```
success, frame = videoCapture.read()
while success:  # Loop until there are no more frames.
    videoWriter.write(frame)
    success, frame = videoCapture.read()
```

The arguments to the constructor of the `VideoWriter` class deserve special attention. A video's filename must be specified. Any preexisting file with this name is overwritten. A video codec must also be specified. The available codecs may vary from system to system. The supported options may include the following:

- `0`: This option is an uncompressed raw video file. The file extension should be `.avi`.
- `cv2.VideoWriter_fourcc('I','4','2','0')`: This option is an uncompressed YUV encoding, 4:2:0 chroma subsampled. This encoding is widely compatible but produces large files. The file extension should be `.avi`.
- `cv2.VideoWriter_fourcc('P','I','M','1')`: This option is MPEG-1. The file extension should be `.avi`.
- `cv2.VideoWriter_fourcc('X','V','I','D')`: This option is a relatively old MPEG-4 encoding. It is a good option if you want to limit the size of the resulting video. The file extension should be `.avi`.
- `cv2.VideoWriter_fourcc('M','P','4','V')`: This option is another relatively old MPEG-4 encoding. It is a good option if you want to limit the size of the resulting video. The file extension should be `.mp4`.
- `cv2.VideoWriter_fourcc('X','2','6','4')`: This option is a relatively new MPEG-4 encoding. It may be the best option if you want to limit the size of the resulting video. The file extension should be `.mp4`.
- `cv2.VideoWriter_fourcc('T','H','E','O')`: This option is **Ogg Vorbis**. The file extension should be `.ogv`.

A frame rate and frame size must be specified too. Since we are copying from another video, these properties can be read from the `get` method of the `VideoCapture` class.

Enabling hardware acceleration for video I/O

Often, modern GPUs are capable of accelerating the algorithms that decode (read) and encode (write) some of the common video formats. By default, OpenCV's `VideoCapture` and `VideoWriter` classes do not attempt to use GPU acceleration; they run the decoding and encoding algorithms on the CPU. However, starting in OpenCV 4.5.2 and continuing in OpenCV 5, the support for GPU-accelerated encoding/decoding has been improving. If we

modify our video I/O example in the ways highlighted below, OpenCV will try to use GPU acceleration on supported systems:

```
import cv2

videoCapture = cv2.VideoCapture(
    'MyInputVid.avi', cv2.CAP_ANY,
    [cv2.CAP_PROP_HW_ACCELERATION, cv2.VIDEO_ACCELERATION_ANY]
)
fps = videoCapture.get(cv2.CAP_PROP_FPS)
size = (
    int(videoCapture.get(cv2.CAP_PROP_FRAME_WIDTH)),
    int(videoCapture.get(cv2.CAP_PROP_FRAME_HEIGHT))
)
videoWriter = cv2.VideoWriter(
    'MyOutputVid.avi', cv2.VideoWriter_fourcc('I','4','2','0'), fps, size,
    [cv2.VIDEOWRITER_PROP_HW_ACCELERATION, cv2.VIDEO_ACCELERATION_ANY]
)

success, frame = videoCapture.read()
while success: # Loop until there are no more frames.
    videoWriter.write(frame)
    success, frame = videoCapture.read()
```

Here, the parameters `cv2.CAP_ANY, [cv2.CAP_PROP_HW_ACCELERATION, cv2.VIDEO_ACCELERATION_ANY]` tell the `VideoCapture` instance to use any available capture back-end with any available acceleration back-end. Similarly, the parameter `[cv2.VIDEOWRITER_PROP_HW_ACCELERATION, cv2.VIDEO_ACCELERATION_ANY]` tells the `VideoWriter` instance to use any available acceleration back-end.

Currently, Windows is the best-supported platform in terms of OpenCV's use of GPU-accelerated encoding/decoding. Some Linux configurations are also supported. Broader support may be available in future releases of OpenCV 5.x. On systems where OpenCV does not support GPU-accelerated encoding/decoding, it should fall back safely to CPU encoding/decoding. Note that the support for GPU acceleration may vary according to the video format, as well as the operating system and hardware.

> **Note**
>
> For more information on the ongoing efforts to support GPU-accelerated encoding/decoding, see the OpenCV Wiki article at `https://github.com/opencv/opencv/wiki/Video-IO-hardware-acceleration`.

Bear in mind that there is no guarantee that enabling hardware acceleration will actually speed up your application; in some cases, the opposite could be true. Typically, high-resolution videos might be a better use case for hardware acceleration than low-resolution videos. Regardless, for any practical application where time is critical, you should conduct tests using the types of data and hardware that your users may have.

Next, let's turn our attention to the task of capturing video input from a camera instead of from a pre-recorded video file.

Capturing camera frames

A stream of camera frames is represented by a `VideoCapture` object too. However, for a camera, we construct a `VideoCapture` object by passing the camera's device index instead of a video's filename. Let's consider the following example, which captures 10 seconds of video from a camera and writes it to an AVI file. The code is similar to the previous section's sample (which captured frames from a video file instead of a camera) but changes are `highlighted in bold`:

```
import cv2

cameraCapture = cv2.VideoCapture(0)
fps = 30  # An assumption
size = (
    int(cameraCapture.get(cv2.CAP_PROP_FRAME_WIDTH)),
    int(cameraCapture.get(cv2.CAP_PROP_FRAME_HEIGHT))
)
videoWriter = cv2.VideoWriter(
    'MyOutputVid.avi', cv2.VideoWriter_fourcc('I','4','2','0'),
    fps, size
)

success, frame = cameraCapture.read()
numFramesRemaining = 10 * fps - 1 # 10 seconds of frames
while success and numFramesRemaining > 0:
    videoWriter.write(frame)
```

```
    success, frame = cameraCapture.read()
    numFramesRemaining -= 1
```

For some cameras on certain systems, `cameraCapture.get(cv2.CAP_PROP_FRAME_WIDTH)` and `cameraCapture.get(cv2.CAP_PROP_FRAME_HEIGHT)` may return inaccurate results. To be more certain of the actual image dimensions, you can first capture a frame and then get its height and width with code such as `h, w = frame.shape[:2]`. Occasionally, you might even encounter a camera that yields a few bad frames with unstable dimensions before it starts yielding good frames with stable dimensions. If you are concerned about guarding against this kind of quirk, you may want to read and ignore a few frames at the start of a capture session.

We could, in principle, call `cameraCapture.get(cv2.CAP_PROP_FPS)` to get the camera's frame rate; however, for most cameras on most systems, this does not return an accurate value; it typically returns either `nan` (not a number) or `0`. The official documentation at warns of the following:

> *"Reading / writing properties involves many layers. Some unexpected result might happen along this chain.*
>
> `VideoCapture -> API Backend -> Operating System -> Device Driver -> Device Hardware`
>
> *The returned value might be different from what really is used by the device or it could be encoded using device-dependent rules (for example, steps or percentage). Effective behavior depends on device driver and the API backend."*

To create an appropriate `VideoWriter` instance for the camera, we have to either make an assumption about the frame rate (as we did in the preceding code) or measure it using a timer. The latter approach is better and we will cover it later in this chapter, in the *Abstracting a video stream with managers.CaptureManager* section.

The number of cameras and their order is, of course, system dependent. Unfortunately, OpenCV does not provide any means of querying the number of cameras or their properties. If an invalid index is used to construct a `VideoCapture` class, the `VideoCapture` class will not yield any frames; its `read` method will return `(False, None)`. To avoid trying to retrieve frames from a `VideoCapture` object that was not opened correctly, you may want to first call the `VideoCapture.isOpened` method, which returns a Boolean.

The `read` method is inappropriate when we need to synchronize either a set of cameras or a **multihead camera** such as a stereo camera. Then, we use the `grab` and `retrieve` methods instead. For a set of two cameras, we can use code similar to the following:

```
success0 = cameraCapture0.grab()
success1 = cameraCapture1.grab()
if success0 and success1:
    frame0 = cameraCapture0.retrieve()
    frame1 = cameraCapture1.retrieve()
```

Finally, bear in mind that the `VideoCapture` class is an abstraction, in the sense that it may use various underlying video capture APIs or camera APIs, depending on the operating system and the way we have configured OpenCV. For example, depending on the configuration choices that you made in `Chapter 1, Setting Up OpenCV`, your build of OpenCV may or may not support the use of `VideoCapture` with OpenNI-compatible depth cameras.

Optionally, when we construct a `VideoCapture` instance, we can specify which camera API we want to use as a back-end. For example, on Windows, the default back-end for most camera hardware is **Microsoft Media Foundation (MSMF)**, and thus our previous code sample uses this default; however, if we want to use the older and more widely compatible **DirectShow** back-end, we can request it as follows:

```
videoCapture = cv2.VideoCapture(0, cv2.CAP_DSHOW)
```

Alternatively, if we want to change the way OpenCV selects a default camera back-end on a particular system, we can do so via an environment variable. For example, if we want to ensure that OpenCV avoids the MSMF back-end, we can define an environment variable with the name `OPENCV_VIDEOIO_PRIORITY_MSMF` and the value `0`. This means MSMF becomes OpenCV's least-preferred (lowest priority) camera back-end.

Tip

Sometimes, a particular camera proves to be incompatible with a particular back-end. Then, you may want to take steps to change the back-end, as described above. On Windows, OpenCV may give errors such as the following if a camera is not working with the MSMF back-end:

```
[ WARN:0@25.936] global C:\opencv\modules\videoio\src\cap_msmf.cpp
(539) `anonymous-namespace'::SourceReaderCB::~SourceReaderCB
terminating async callback
```

> Then, instead of MSMF, you could try DirectShow, as described above.

For a list of other back-end flags besides `cv2.CAP_DSHOW`, see the documentation at `https://docs.opencv.org/5.0/main_modules/videoio_flags_base.html`.

Sending a video stream over a network

Perhaps you have heard of a type of camera called an **IP camera**, which continuously sends a stream of images via a specified IP address on a network. The video stream from an IP camera can potentially be viewed in any web browser on any computer connected to the same network. Of course, the network should be secured to prevent unwanted access. IP cameras are used in a wide range of practical applications, including security systems, baby monitors, and livestock monitors.

Using OpenCV and a bit of networking code, we can turn any webcam or any computer vision application into the equivalent of an IP camera. For prototyping purposes, this is a great way to present a demo to multiple users, or to debug an application that is running on a **headless system** (a system with no display capabilities of its own). For example, if your computer vision application runs on a small, mobile robot, then you and other users may find it convenient to access the robot's video stream in a web browser on other local systems, rather than trying to build display capabilities into the robot itself.

> **Note**
>
> The following sample assumes some basic familiarity with concepts relating to web applications, but let's review a few key terms. A **client** application, such as a web browser, typically sends a **request** to a **server** application, such as a streaming video service. The server application sends a **response** to the client; the response may contain content for the client to use. The client and server applications may be running on the same machine or different machines. On a network, a machine is identified by an **IP address**. On a given machine, a particular server application is identified by a **port number**. Thus, a client app can find a server app by the combination of IP address and port number.

Let's build a basic sample of a video streaming server. To capture images from a camera, we will use OpenCV. To serve the images via a network, we will use Flask, a minimalistic web

application framework for Python. We can install Flask by running one of the following commands (depending on the environment):

- For any Python environment, using `pip` as a package manager, run:

```
$ pip install flask
```

- Alternatively, on Debian or Ubuntu, using `apt` as a package manager, run:

```
$ sudo apt-get install python3-flask
```

- Alternatively, on Arch Linux or Manjaro, using `pacman` as a package manager, run:

```
$ sudo pacman -S python-flask
```

Having installed Flask, we can begin to write our sample Python script. Import OpenCV and Flask as follows:

```
import cv2
import flask
```

Next, we will create an instance of a Flask application. For this purpose, Flask provides a class that is simply called `flask.Flask`. We instantiate it with a module name, which is normally just the name of our script:

```
app = flask.Flask(__name__)
```

Our Flask application has one role: to serve a stream of images. We will use a streaming format called **Motion JPEG** (**MJPEG**), which is supported by all popular web browsers and many other stream-reading applications. The stream will consist of an indeterminate number of frames, captured from a camera and converted to JPEG format. The stream will start when a client connects to our server, and it will end when the client disconnects.

Before a function that uses Flask to handle client requests, let's first write a helper function that uses OpenCV to capture and convert frames. Specifically, the helper will be a Python **generator** function, which is a convenient way to encapsulate logic about getting the next element in an indefinitely long series (of frames, in our case). Our helper begins with familiar code to start capturing low-resolution camera frames:

```
def mjpeg_generator():
```

```
    video_capture = cv2.VideoCapture(0)
    video_capture.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    video_capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    success, frame = video_capture.read()
```

We will capture and convert frames in a loop until we fail to read a frame from the camera, or until we fail to convert a frame to JPEG format, or until the rest of our application stops calling our generator function. To convert frames, we will use an OpenCV function called `imencode`, which is very similar to `imwrite`, except that `imencode` keeps the encoded image in memory instead of writing it to file. Every time we successfully encode a JPEG image, we will parcel the JPEG data with a header, in a format defined by the MJPEG standard. Also, each packet in the MJPEG stream is separated from the next by a boundary marker, in this case `'--frame'`. Here is the implementation of the loop:

```
    while success:
        success, jpeg = cv2.imencode('.jpg', frame)
        if not success:
            break
        yield (b'--frame\r\n'
               b'Content-Type: image/jpeg\r\n'
               b'Content-Length: %d\r\n'
               b'\r\n'
               b'%s\r\n' % (len(jpeg), jpeg.tobytes()))
        success, frame = video_capture.read()
```

Note the use of Python's `yield` keyword, which is a defining characteristic of a generator function. The `yield` keyword is akin to the `return` keyword, but the next time the generator function is called, it will start from the last `yield` statement instead of from the beginning of the function.

That concludes the implementation of our `mjpeg_generator` helper function. Now, let's see how we can use such a function to make Flask serve a continuous stream of images in response to a client's request. Flask enables us to use a **decorator** pattern to associate certain functions with certain web requests. Here is a function that handles any requests directed at the root (`'/'`) path of the streaming video server:

```
@app.route('/')
def stream_from_camera():
    return flask.Response(
```

```
        mjpeg_generator(),
        mimetype='multipart/x-mixed-replace;boundary=frame')
```

Note

The `@` annotation (as in `@app.route`) denotes a Python **decorator**, a built-in feature of the language that allows the wrapping of a function by another function, normally used to apply user-defined or library-defined behavior in several places in an application. You can find relevant documentation at `https://docs.python.org/3/reference/compound_stmts.html`.

Note that the `app` in `@app.route` is the variable name we gave to our `flask.Flask` instance, near the start of our script. The function name `stream_from_camera` has no special significance, as far as Flask is concerned; we could have chosen any function name for this request handler. Finally, note that the role of this function is to return a `flask.Response` object, which represents our server's response to the client's request. Specifically, the response is an MJPEG stream. We initialize the `flask.Response` instance by calling our generator function, which provides the stream's content, and by specifying a standard media type or **mimetype**, which tells the client what kind of content to expect. Here, the mimetype `'multipart/x-mixed-replace;boundary=frame'` tells the client to expect a series of packets, where each new packet's content (in this case, each JPEG frame) is intended to replace the last, and where the boundary marker between packets is `'--frame'`.

The only thing left to do in this script is to tell our server application to start running. We can do this, with some optional arguments, as follows:

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080, threaded=True)
```

The `host` argument specifies an IP address where the application should accept incoming requests; the special value `'0.0.0.0'` means that Flask should use any and all IP addresses available, including any and all IP addresses that your router has assigned to this computer. (Alternatively, we can manually specify an IP address, but it has to be a valid address for this computer on this network, or else our app will fail to run.) The `port` argument is the port number; we use `8080`, which is a conventional port number for HTTP server apps, other than a public website (which is conventionally port 80). Lastly, with `threaded=True`, we specify that this server app should be multithreaded; this makes it possible for multiple clients to connect at once.

Tip

Here, our networking code is a rudimentary sample, it has no security features, and the resulting video stream is accessible to anybody on the same network. Do not use this code on public networks – for instance, in airports or cafés. For practical video streaming applications, always consider using HTTPS, password protection, and other security measures as appropriate.

Let's run our script. Your operating system or firewall software might ask you whether Python can have permission to access the network. Say yes. Now, you should see command line output similar to the following:

```
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://192.168.2.10:8080
Press CTRL+C to quit
```

Take note of the two URLs. The IP address starting with `127.0` will only work on the same machine. The IP address starting with `192.168` will also work on other machines on the same local network.

Now, on the same computer, open a web browser and enter the first URL (such as `http://127.0.0.1:8080`) in the browser's address bar. The camera should start capturing frames (and its recording light should turn on, if it has one), and you should see the resulting video stream in the browser window.

Now, on a different computer connected to the same local network, open a web browser and navigate to the second URL (such as `http://192.168.2.10:8080`). Again, you should see a stream of frames from the camera attached to the first computer. The video streams in the two web browsers, on the two machines, should appear to be synchronized or nearly synchronized, unless your local network is very slow.

The following photo shows the camera stream in two web browsers, on two machines. The laptop is capturing the camera frames and streaming them to the web browsers on the same laptop and on the desktop:



Figure 2.3: A photo of an MJPEG streaming application. A server application on the laptop is capturing video frames from the laptop's camera, converting them to JPEG format, and streaming them to two client applications: a web browser on the same laptop and a web browser on the desktop

When you have finished looking at the streams, close the two web browsers. The camera should stop capturing frames (and its recording light should turn off, if it has one). As the command line output says, you can press *Ctrl+C* to quit the server app.

Now, let's move on to writing our own basic GUI applications, to show a camera stream on a local machine, without any networking code or web browsers.

Displaying an image in a window

One of the most basic operations in OpenCV is displaying an image in a window. This can be done with the `imshow` function. If you come from any other GUI framework background, you might think it sufficient to call `imshow` to display an image. However, in OpenCV, the window is drawn (or re-drawn) only when you call another function, `waitKey`. The latter function pumps the window's event queue (allowing various events such as drawing to be handled), and it returns the keycode of any key that the user may have typed within a specified timeout. To some extent, this rudimentary design simplifies the task of developing demos that use video or webcam input; at least the developer has manual control over the capture and display of new frames.

Here is a very simple sample script to read an image from a file and display it:

```
import cv2
import numpy as np

img = cv2.imread('my-image.png')
cv2.imshow('my image', img)
cv2.waitKey()
cv2.destroyAllWindows()
```

The `imshow` function takes two parameters: the name of the window in which we want to display the image and the image itself. We will talk about `waitKey` in more detail in the next section, `Displaying camera frames in a window`.

The aptly named `destroyAllWindows` function disposes of all of the windows created by OpenCV.

Displaying camera frames in a window

OpenCV allows named windows to be created, redrawn, and destroyed using the `namedWindow`, `imshow`, and `destroyWindow` functions. Also, any window may capture keyboard input via the `waitKey` function and mouse input via the `setMouseCallback` function. Let's look at an example where we show the frames captured from a live camera:

```
import cv2

clicked = False
def onMouse(event, x, y, flags, param):
    global clicked
    if event == cv2.EVENT_LBUTTONUP:
        clicked = True

cameraCapture = cv2.VideoCapture(0)
cv2.namedWindow('MyWindow')
cv2.setMouseCallback('MyWindow', onMouse)

print('Showing camera feed. Click window or press any key to stop.')

success, frame = cameraCapture.read()

while success and cv2.waitKey(1) == -1 and not clicked:
```

```
    cv2.imshow('MyWindow', frame)
    success, frame = cameraCapture.read()

cv2.destroyWindow('MyWindow')
cameraCapture.release()
```

The argument for `waitKey` is the number of milliseconds to wait for keyboard input. By default, it is `0`, which is a special value meaning infinity. The return value is either `-1` (meaning that no key has been pressed) or an ASCII keycode, such as `27` for *Esc*. For a list of ASCII keycodes, refer to `https://www.asciitable.com/`. Also, note that Python provides a standard function, `ord`, which can convert a character into its ASCII keycode. For example, `ord('a')` returns `97`.

Again, note that OpenCV's window functions and `waitKey` are interdependent. OpenCV windows are only updated when `waitKey` is called. Conversely, `waitKey` only captures input when an OpenCV window has focus.

The mouse callback passed to `setMouseCallback` should take five arguments, as seen in our code sample. The callback's `param` argument is set as an optional third argument to `setMouseCallback`. By default, it is `0`. The callback's event argument is one of the following actions:

- `cv2.EVENT_MOUSEMOVE`: This event refers to mouse movement.
- `cv2.EVENT_LBUTTONDOWN`: This event refers to the left button going down when it is pressed.
- `cv2.EVENT_RBUTTONDOWN`: This event refers to the right button going down when it is pressed.
- `cv2.EVENT_MBUTTONDOWN`: This event refers to the middle button going down when it is pressed.
- `cv2.EVENT_LBUTTONUP`: This event refers to the left button coming back up when it is released.
- `cv2.EVENT_RBUTTONUP`: This event refers to the right button coming back up when it is released.
- `cv2.EVENT_MBUTTONUP`: This event refers to the middle button coming back up when it is released.
- `cv2.EVENT_LBUTTONDBLCLK`: This event refers to the left button being double-clicked.

- `cv2.EVENT_RBUTTONDBLCLK`: This event refers to the right button being double-clicked.
- `cv2.EVENT_MBUTTONDBLCLK`: This event refers to the middle button being double-clicked.

The mouse callback's flags argument may be some bitwise combination of the following events:

- `cv2.EVENT_FLAG_LBUTTON`: This event refers to the left button being pressed.
- `cv2.EVENT_FLAG_RBUTTON`: This event refers to the right button being pressed.
- `cv2.EVENT_FLAG_MBUTTON`: This event refers to the middle button being pressed.
- `cv2.EVENT_FLAG_CTRLKEY`: This event refers to the `Ctrl` key being pressed.
- `cv2.EVENT_FLAG_SHIFTKEY`: This event refers to the `Shift` key being pressed.
- `cv2.EVENT_FLAG_ALTKEY`: This event refers to the `Alt` key being pressed.

Unfortunately, OpenCV does not provide any means of manually handling window events. For example, we cannot stop our application when a window's close button is clicked. Due to OpenCV's limited event handling and GUI capabilities, many developers prefer to integrate it with other application frameworks. Later in this chapter, in the *Cameo – an object-oriented design* section, we will design an abstraction layer to help to integrate OpenCV with any application framework.

Project Cameo (face tracking and image manipulation)

OpenCV is often studied through a cookbook approach that covers a lot of algorithms, but nothing about high-level application development. To an extent, this approach is understandable because OpenCV's potential applications are so diverse. To name just a few examples, OpenCV can be used in photo/video editors, motion-controlled games, medical imaging devices, a robot's AI, or psychology experiments where we log participants' eye movements. Across these varied use cases, can we truly study a useful set of abstractions?

The book's authors believe we can, and the sooner we start creating abstractions, the better. Throughout `Chapters 2` to `5`, we will structure several of our OpenCV examples around a single application, but, at each step, we will design a component of this application to be extensible and reusable.

We will develop an interactive application that performs face tracking and image manipulations on camera input in real time. This type of application covers a broad range of OpenCV's functionality and challenges us to create an efficient, effective implementation.

Specifically, our application will merge faces in real time. Given two streams of camera input (or, optionally, prerecorded video input), the application will superimpose faces from one stream atop faces in the other. Filters and distortions will be applied to give this blended scene a unified look and feel. Users should have the experience of being engaged in a live performance where they enter another environment and persona. This type of user experience is popular in amusement parks such as Disneyland.

In such an application, users would immediately notice flaws, such as a low frame rate or inaccurate tracking. To get the best results, we will try several approaches using conventional imaging and depth imaging.

We will call our application Cameo. A cameo (in jewelry) is a small portrait of a person or (in film) a very brief role played by a celebrity.

Cameo – an object-oriented design

Python applications can be written in a purely procedural style. This is often done with small applications, such as our basic I/O scripts, discussed previously. However, from now on, we will often use an object-oriented style because it promotes modularity and extensibility.

> **Note**
>
> Microsoft Learn offers tutorials on object-oriented programming in "More Python for Beginners" at . If you are still relatively new to the topic or need a refresher, spend an hour studying this excellent tutorial.

From our overview of OpenCV's I/O functionality, we know that all images are similar, regardless of their source or destination. No matter how we obtain a stream of images or where we send it as output, we can apply the same application-specific logic to each frame in this stream. Separation of I/O code and application code becomes especially convenient in an application, such as Cameo, that uses multiple I/O streams.

We will create classes called `CaptureManager` and `WindowManager` as high-level interfaces to I/O streams. Our application code may use `CaptureManager` to read new frames and, optionally, to dispatch each frame to one or more outputs, including a still image file, a video file, and a window (via a `WindowManager` class). A `WindowManager` class lets our application code handle a window and events in an object-oriented style.

Both `CaptureManager` and `WindowManager` are extensible. We could make implementations that did not rely on OpenCV for I/O.

Abstracting a video stream with managers.CaptureManager

As we have seen, OpenCV can capture, show, and record a stream of images from either a video file or camera, but there are some special considerations in each case. Our `CaptureManager` class abstracts some of the differences and provides a higher-level interface to dispatch images from the capture stream to one or more outputs—a still image file, video file, or window.

A `CaptureManager` object is initialized with a `VideoCapture` object and has `enterFrame` and `exitFrame` methods that should typically be called on every iteration of an application's main loop. Between a call to `enterFrame` and `exitFrame`, the application may (any number of times) set a `channel` property and get a `frame` property. The `channel` property is initially `0` and only multihead cameras use other values. The `frame` property is an image corresponding to the current channel's state when `enterFrame` was called.

A `CaptureManager` class also has the `writeImage`, `startWritingVideo`, and `stopWritingVideo` methods that may be called at any time. Actual file writing is postponed until `exitFrame`. Also, during the `exitFrame` method, `frame` may be shown in a window, depending on whether the application code provides a `WindowManager` class either as an argument to the constructor of `CaptureManager` or by setting the `previewWindowManager` property.

If the application code manipulates `frame`, the manipulations are reflected in recorded files and in the window. A `CaptureManager` class has a constructor argument and property called `shouldMirrorPreview`, which should be `True` if we want `frame` to be mirrored (horizontally flipped) in the window but not in recorded files. Typically, when facing a camera, users prefer a live camera feed to be mirrored.

Recall that a `VideoWriter` object needs a frame rate, but OpenCV does not provide any reliable way to get an accurate frame rate for a camera. The `CaptureManager` class works around this limitation by using a frame counter and Python's standard `time.perf_counter` function to estimate the frame rate if necessary. This approach is not foolproof. Depending on frame rate fluctuations and the system-dependent implementation of `time.perf_counter`, the accuracy of the estimate might still be poor in some cases. However, if we deploy to unknown hardware, it is better than just assuming that the user's camera has a particular frame rate.

Let's create a file called `managers.py`, which will contain our implementation of `CaptureManager`. This implementation turns out to be quite long, so we will look at it in several pieces:

1. First, let's add imports and a constructor, as follows:

```
import cv2
import numpy
import time


class CaptureManager(object):
    def __init__(
        self, capture, previewWindowManager = None,
        shouldMirrorPreview = False
    ):
        self.previewWindowManager = previewWindowManager
        self.shouldMirrorPreview = shouldMirrorPreview
        self._capture = capture
        self._channel = 0
        self._enteredFrame = False
        self._frame = None
        self._imageFilename = None
        self._videoFilename = None
        self._videoEncoding = None
        self._videoWriter = None
        self._startTime = None
        self._framesElapsed = 0
        self._fpsEstimate = None
```

2. Next, let's add the following getter and setter methods for the properties of `CaptureManager`:

```
    @property
    def channel(self):
        return self._channel

    @channel.setter
    def channel(self, value):
        if self._channel != value:
            self._channel = value
```

```
            self._frame = None

    @property
    def frame(self):
        if self._enteredFrame and self._frame is None:
            _, self._frame = self._capture.retrieve(
                self._frame, self.channel)
        return self._frame

    @property
    def isWritingImage(self):
        return self._imageFilename is not None

    @property
    def isWritingVideo(self):
        return self._videoFilename is not None
```

Note that most of the member variables are nonpublic, as denoted by the underscore prefix in variable names, such as `self._enteredFrame`. These nonpublic variables relate to the state of the current frame and any file-writing operations. As discussed previously, the application code only needs to configure a few things, which are implemented as constructor arguments and settable public properties: the camera channel, the window manager, and the option to mirror the camera preview.

Note

Python does not enforce the concept of nonpublic member variables, but in cases where the developer intends a variable to be treated as nonpublic, you will often see the single-underscore prefix (_) or double-underscore prefix (__). The single-underscore prefix is just a convention, indicating that the variable should be treated as protected (accessed only within the class and its subclasses). The double-underscore prefix actually causes the Python interpreter to rename the variable, such that `MyClass.__myVariable` becomes `MyClass._MyClass__myVariable`. This is called **name mangling** (quite appropriately). By convention, such a variable should be treated as private (accessed only within the class, and not its subclasses). The same prefixes, with the same significance, can be applied to methods as well as variables.

3. Continuing with our implementation, let's add the `enterFrame` method to `managers.py`:

```
    def enterFrame(self):
        """Capture the next frame, if any."""

        # But first, check that any previous frame was exited.
        assert not self._enteredFrame, \
            'previous enterFrame() had no matching exitFrame()'

        if self._capture is not None:
            self._enteredFrame = self._capture.grab()
```

Note that the implementation of `enterFrame` only grabs (synchronizes) a frame, whereas actual retrieval from a channel is postponed to a subsequent reading of the `frame` variable.

4. Next, let's add the `exitFrame` method to `managers.py`:

```
    def exitFrame(self):
        """Draw to the window. Write to files. Release the frame."""

        # Check whether any grabbed frame is retrievable.
        # The getter may retrieve and cache the frame.
        if self.frame is None:
            self._enteredFrame = False
            return

        # Update the FPS estimate and related variables.
        if self._framesElapsed == 0:
            self._startTime = time.perf_counter()
        else:
            timeElapsed = time.perf_counter() - self._startTime
            self._fpsEstimate = self._framesElapsed / timeElapsed
        self._framesElapsed += 1

        # Draw to the window, if any.
        if self.previewWindowManager is not None:
            if self.shouldMirrorPreview:
                mirroredFrame = numpy.fliplr(self._frame)
                self.previewWindowManager.show(mirroredFrame)
```

```
            else:
                self.previewWindowManager.show(self._frame)

        # Write to the image file, if any.
        if self.isWritingImage:
            cv2.imwrite(self._imageFilename, self._frame)
            self._imageFilename = None

        # Write to the video file, if any.
        self._writeVideoFrame()

        # Release the frame.
        self._frame = None
        self._enteredFrame = False
```

The implementation of `exitFrame` takes the image from the current channel, estimates a frame rate, shows the image via the window manager (if any), and fulfills any pending requests to write the image to files.

5. Several other methods also pertain to file writing. Let's add the following implementations of public methods named `writeImage`, `startWritingVideo`, and `stopWritingVideo` to `managers.py`:

```
    def writeImage(self, filename):
        """Write the next exited frame to an image file."""
        self._imageFilename = filename

    def startWritingVideo(
        self, filename,
        encoding = cv2.VideoWriter_fourcc('M','J','P','G')
    ):
        """Start writing exited frames to a video file."""
        self._videoFilename = filename
        self._videoEncoding = encoding

    def stopWritingVideo(self):
        """Stop writing exited frames to a video file."""
        self._videoFilename = None
        self._videoEncoding = None
        self._videoWriter = None
```

The preceding methods simply update the parameters for file-writing operations, whereas the actual writing operations are postponed to the next call of `exitFrame`.

6. Earlier in this section, we saw that `exitFrame` calls a helper method named `_writeVideoFrame`. Let's add the following implementation of `_writeVideoFrame` to `managers.py`:

```
    def _writeVideoFrame(self):

        if not self.isWritingVideo:
            return

        if self._videoWriter is None:
            fps = self._capture.get(cv2.CAP_PROP_FPS)
            if numpy.isnan(fps) or fps <= 0.0:
                # The capture's FPS is unknown so use an estimate.
                if self._framesElapsed < 20:
                    # Wait until more frames elapse so that the
                    # estimate is more stable.
                    return
                else:
                    fps = self._fpsEstimate
            size = (int(self._capture.get(
                        cv2.CAP_PROP_FRAME_WIDTH)),
                    int(self._capture.get(
                        cv2.CAP_PROP_FRAME_HEIGHT)))
            self._videoWriter = cv2.VideoWriter(
                self._videoFilename, self._videoEncoding, fps, size)

        self._videoWriter.write(self._frame)
```

The preceding method creates or appends to a video file in a manner that should be familiar from our earlier scripts (refer to the *Reading/writing a video file section*, earlier in this chapter). However, in situations where the frame rate is unknown, we skip some frames at the start of the capture session so that we have time to build up an estimate of the frame rate.

This concludes our implementation of `CaptureManager`. Although it relies on `VideoCapture`, we could make other implementations that do not use OpenCV for input. For example, we could make a subclass that is instantiated with a socket connection, whose byte stream could be parsed as a stream of images. Also, we could make a subclass that uses a third-party camera

library with different hardware support than what OpenCV provides. However, for Cameo, our current implementation is sufficient.

Abstracting a window and keyboard with managers.WindowManager

As we have seen, OpenCV provides functions that cause a window to be created, destroyed, show an image, and process events. Rather than being methods of a window class, these functions require a window's name to be passed as an argument. Since this interface is not object-oriented, it is arguably inconsistent with OpenCV's general style. Also, it is unlikely to be compatible with other window- or event-handling interfaces that we might eventually want to use instead of OpenCV's.

For the sake of object orientation and adaptability, we abstract this functionality into a `WindowManager` class with the `createWindow`, `destroyWindow`, `show`, and `processEvents` methods. As a property, `WindowManager` has a function object called `keypressCallback`, which (if it is not `None`) is called from `processEvents` in response to any keypress. The `keypressCallback` object must be a function that takes a single argument, specifically an ASCII keycode.

Let's add an implementation of `WindowManager` to `managers.py`. The implementation begins with the following class declaration and `__init__` method:

```
class WindowManager(object):

    def __init__(self, windowName, keypressCallback = None):
        self.keypressCallback = keypressCallback

        self._windowName = windowName
        self._isWindowCreated = False
```

The implementation continues with the following methods to manage the life cycle of the window and its events:

```
    @property
    def isWindowCreated(self):
        return self._isWindowCreated

    def createWindow(self):
        cv2.namedWindow(self._windowName)
        self._isWindowCreated = True
```

```
    def show(self, frame):
        cv2.imshow(self._windowName, frame)

    def destroyWindow(self):
        cv2.destroyWindow(self._windowName)
        self._isWindowCreated = False

    def processEvents(self):
        keycode = cv2.waitKey(1)
        if self.keypressCallback is not None and keycode != -1:
            self.keypressCallback(keycode)
```

Our current implementation only supports keyboard events, which will be sufficient for Cameo. However, we could modify `WindowManager` to support mouse events, too. For example, the class interface could be expanded to include a `mouseCallback` property (and an optional constructor argument) but could otherwise remain the same. With an event framework other than OpenCV's, we could support additional event types in the same way by adding callback properties.

Applying everything in Cameo

Our application is represented by the `Cameo` class with two methods: `run` and `onKeypress`. On initialization, a `Cameo` object creates a `WindowManager` object with `onKeypress` as a callback, as well as a `CaptureManager` object using a camera (specifically, a `cv2.VideoCapture` object) and the same `WindowManager` object. When `run` is called, the application executes a main loop in which frames and events are processed.

As a result of event processing, `onKeypress` may be called. The spacebar causes a screenshot to be taken, *Tab* causes a screencast (a video recording) to start/stop, and *Esc* causes the application to quit.

In the same directory as `managers.py`, let's create a file called `cameo.py`, where we will implement the `Cameo` class:

1. The implementation begins with the following `import` statements and `__init__` method:

```
import cv2
from managers import WindowManager, CaptureManager

class Cameo(object):
```

```
    def __init__(self):
        self._windowManager = WindowManager(
            'Cameo',self.onKeypress)
        self._captureManager = CaptureManager(
            cv2.VideoCapture(0), self._windowManager, True)
```

2. Next, let's add the following implementation of the `run` method:

```
    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            frame = self._captureManager.frame

            if frame is not None:
                # TODO: Filter the frame (Chapter 3).
                pass

            self._captureManager.exitFrame()
            self._windowManager.processEvents()
```

3. To complete the `Cameo` class implementation, here is the `onKeypress` method:

```
    def onKeypress(self, keycode):
        """Handle a keypress.

        space -> Take a screenshot.
        tab -> Start/stop recording a screencast.
        escape -> Quit.

        """
        if keycode == 32: # space
            self._captureManager.writeImage('screenshot.png')
        elif keycode == 9: # tab
            if not self._captureManager.isWritingVideo:
                self._captureManager.startWritingVideo(
                    'screencast.avi')
            else:
                self._captureManager.stopWritingVideo()
```

```
            elif keycode == 27: # escape
                self._windowManager.destroyWindow()
```

4. Finally, let's add a `__main__` block that instantiates and runs `Cameo`, as follows:

```
if __name__=="__main__":
    Cameo().run()
```

When running the application, note that the live camera feed is mirrored, while screenshots and screencasts are not. This is the intended behavior, as we pass `True` for `shouldMirrorPreview` when initializing the `CaptureManager` class.

Here is a screenshot of Cameo, showing a window (with the title **Cameo**) and the current frame from a camera:

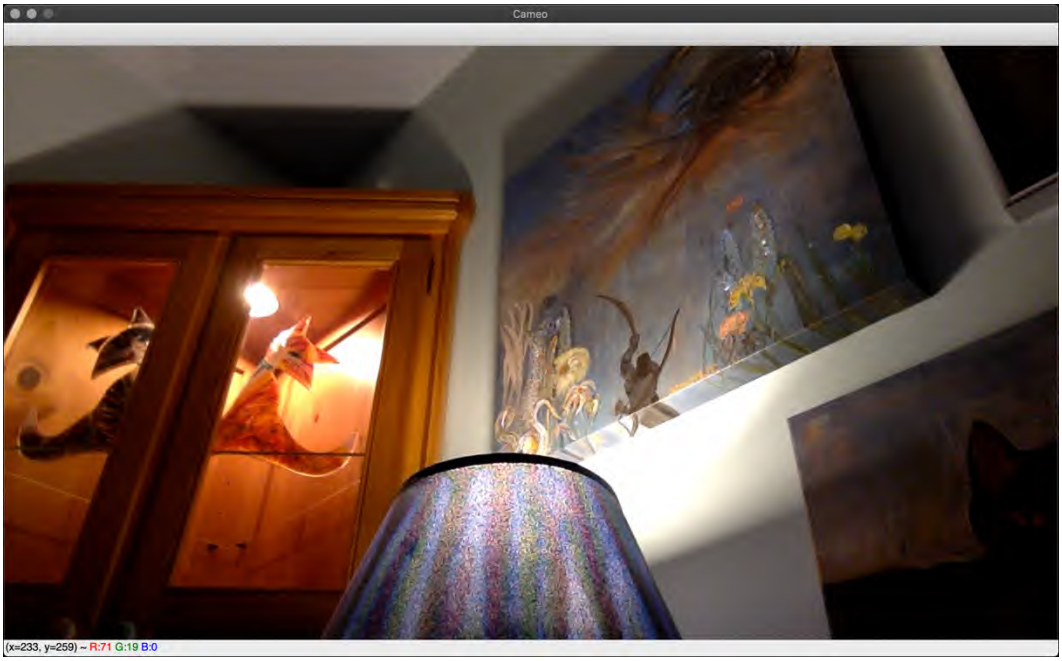


Figure 2.4: A screenshot of the Cameo application, showing a camera frame in a window

So far, we do not manipulate the frames in any way except to mirror them for preview. We will start to add more interesting effects in *Chapter 3, Processing Images with OpenCV*.

Summary

By now, we should have an application that displays a camera feed, listens for keyboard input, and (on command) records a screenshot or screencast. We are ready to extend the application by inserting some image-filtering code (*Chapter 3, Processing Images with OpenCV*) between the start and end of each frame. Optionally, we are also ready to integrate other camera drivers or application frameworks besides the ones supported by OpenCV.

We also possess the knowledge to manipulate images as NumPy arrays, UMat objects, or data that we can stream over a network. An understanding of image data and especially of NumPy arrays forms the perfect foundation for our next topic, filtering images.

3

Processing Images with OpenCV

Sooner or later, when working with images, you will find that you need to alter them: be it by applying artistic filters, extrapolating certain sections, blending two images, or whatever else your mind can conjure. This chapter presents some techniques that you can use to alter images. By the end of this chapter, you should be able to perform tasks such as sharpening an image, marking the contours of subjects, and detecting crosswalks using a line segment detector. Specifically, our discussion and code samples will cover the following topics:

- Converting images between different color models
- Understanding the importance of frequencies and the Fourier transform in image processing
- Applying **high-pass filters (HPFs)**, **low-pass filters (LPFs)**, edge detection filters, and custom convolution filters
- Detecting and analyzing contours, lines, circles, and other geometric shapes
- Writing classes and functions that encapsulate the implementation of a filter

> **Note**
>
> **Download the code bundle and the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

This chapter uses Python, OpenCV, NumPy, and SciPy. Please refer to *Chapter 1*, *Setting Up OpenCV*, for installation instructions.

The complete code for this chapter can be found in this book's GitHub repository in the `chapter03` folder. The sample images are also in this book's GitHub repository, in the `images` folder.

You can download the example project and code for this book by following the instructions in the ***Download the example code files*** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

A subset of the chapter's sample code can be edited and run interactively in Google Colab.

Converting images between different color models

OpenCV implements literally hundreds of formulas that pertain to the conversion of color models. Some color models are commonly used by input devices such as cameras, while other models are commonly used for output devices such as televisions, computer displays, and printers. In between input and output, when we apply computer vision techniques to images, we will typically work with three kinds of color models: **grayscale**, **blue-green-red (BGR)**, and **hue-saturation-value (HSV)**. Let's go over these briefly:

- **Grayscale** is a model that reduces color information by translating it into shades of gray or brightness. This model is extremely useful for the intermediate processing of images in problems where brightness information alone is sufficient, such as face detection. Typically, each pixel in a grayscale image is represented by a single 8-bit value, ranging from 0 for black to 255 for white.
- **BGR** is the blue-green-red color model, in which each pixel has a triplet of values representing the blue, green, and red components or **channels** of the pixel's color. Web developers, and anyone who works with computer graphics, will be familiar with a similar definition of colors, except with the reverse channel order, **red-green-blue (RGB)**. Typically, each pixel in a BGR image is represented by a triplet of 8-bit values, such as `[0, 0, 0]` for black, `[255, 0, 0]` for blue, `[0, 255, 0]` for green, `[0, 0, 255]` for red, and `[255, 255, 255]` for white.
- The **HSV** model uses a different triplet of channels. Hue is the color's tone, saturation is its intensity, and value represents its brightness. For an object of a known color, the *H* and *S* channel values tend to be relatively stable under a variety of lighting conditions. (Conversely, in grayscale, BGR, or RGB images, all channel values may vary greatly depending on whether the given object is viewed in light or in shadow.) Thus, HSV is often used to describe a colorful object that we want to track under varying lighting conditions, as we will do in *Chapter 8, Tracking Objects*.

By default, OpenCV uses the BGR color model (with 8 bits per channel) to represent any image that it loads from a file or captures from a camera.

Now that we have defined the color models we will use, let's consider how the default model might differ from our intuitive understanding of color.

Light is not paint

For newcomers to the BGR color space, it might seem that things do not add up properly: for example, the `[0, 255, 255]` triplet (no blue, full green, and full red) produces the color yellow. If you have an artistic background, you won't even need to pick up paints and brushes to know that green and red paint mix together into a muddy shade of brown. However, the color models that are used in computing are called **additive** models, and they deal with lights. Lights behave differently from paints (which follow a **subtractive** color model), and since software runs on a computer whose medium is a monitor that emits light, the color model of reference is an additive one.

Differences in color are the way that we perceive differences in wavelengths of light. A coat of paint changes an object's color by absorbing or reflecting certain wavelengths of light. Fundamentally, computer vision involves the analysis of waves of light (or sometimes waves of radiation or ultrasound) as they travel through space and time and get reflected or transformed by things in their path. Thus, we arrive at our next topic: waveforms.

Exploring the Fourier transform

Much of the processing you apply to images and videos in OpenCV involves the concept of the Fourier transform in some capacity. Joseph Fourier was an 18th-century French mathematician who discovered and popularized many mathematical concepts. He studied the physics of heat, and the mathematics of all things that can be represented by waveform functions. In particular, he observed that all waveforms are just the sum of simple sinusoids of different frequencies.

In other words, the waveforms you observe all around you are the sum of other waveforms. This concept is incredibly useful when manipulating images because it allows us to identify regions in images where a signal (such as the values of image pixels) changes a lot, and also regions where the change is less dramatic. We can then apply domain-specific logic to mark these regions as noise or regions of interest, background or foreground, and so on. These are the frequencies that make up the original image, and we have the power to separate them to make sense of the image and extrapolate interesting data.

Note

OpenCV implements a number of algorithms that enable us to process images and make sense of the data contained in them, and these are also reimplemented in NumPy to make our life even easier. NumPy has a **fast Fourier transform (FFT)** package, which contains the `fft2` method. This method allows us to compute a **discrete Fourier transform (DFT)** of the image. For more about the Fourier transform and its advanced usage in image analysis, see Joseph Howse's book *OpenCV 4 for Secret Agents*, specifically *Chapter 7, Seeing a Heartbeat with a Motion-Amplifying Camera*.

The Fourier transform can be used to find the **magnitude spectrum** of an image (or other data). The magnitude spectrum represents the original image (or other data) in terms of its changes. Think of it as taking an image and dragging all the brightest pixels to the center. Then, you gradually work your way out to the border where all the darkest pixels have been pushed. Immediately, you will be able to see how many light and dark pixels are contained in your image and the percentage of their distribution.

The Fourier transform is the conceptual basis of many algorithms that are used for common image processing operations, such as edge detection or line and shape detection. However, the Fourier transform is an expensive operation, so, in practice, many algorithms replace it with simpler functions that process a small region or neighborhood of pixels, instead of computing waveforms or frequencies across the entire image.

Before examining line and shape detection in detail, let's take a look at two kinds of filters that form the foundation of edge detection: HPFs and LPFs.

HPFs and LPFs

An HPF is a filter that examines a region of an image and boosts the intensity of certain pixels based on the difference in the intensity of the surrounding pixels.

Take, for example, the following kernel:

```
[[    0,   -0.25,      0],
 [ -0.25,       1,  -0.25],
 [    0,   -0.25,      0]]
```

Note

A **kernel** is a set of weights that are applied to a region in a source image to generate a single pixel in the destination image. For example, if we call an OpenCV function with a parameter to specify a kernel size or `ksize` of 7, this implies that 49 (7 x 7) source pixels

are considered when generating each destination pixel. We can think of a kernel as a piece of frosted glass moving over the source image and letting a diffused blend of the source's light pass through.

The preceding kernel gives us the average difference in intensity between the central pixel and all its immediate horizontal and vertical neighbors. If a pixel stands out from the surrounding pixels, the resulting value will be high. This type of kernel represents a so-called high-boost filter, which is a type of HPF, and it is particularly effective in edge detection.

Note

Note that the values in an edge detection kernel typically sum up to `0`. We will cover this in the *Custom kernels – getting convolutions* section of this chapter.

Let's go through an example of applying an HPF to an image:

```
import cv2
import numpy as np
from scipy import ndimage

kernel_3x3 = np.array([[-1, -1, -1],
                       [-1,  8, -1],
                       [-1, -1, -1]])

kernel_5x5 = np.array([[-1, -1, -1, -1, -1],
                       [-1,  1,  2,  1, -1],
                       [-1,  2,  4,  2, -1],
                       [-1,  1,  2,  1, -1],
                       [-1, -1, -1, -1, -1]])

img = cv2.imread("../images/statue_small.jpg",
                 cv2.IMREAD_GRAYSCALE)

k3 = ndimage.convolve(img, kernel_3x3)
k5 = ndimage.convolve(img, kernel_5x5)

blurred = cv2.GaussianBlur(img, (17, 17), 0)
g_hpf = img - blurred
```

```
cv2.imshow("3x3", k3)
cv2.imshow("5x5", k5)
cv2.imshow("blurred", blurred)
cv2.imshow("g_hpf", g_hpf)
cv2.waitKey()
cv2.destroyAllWindows()
```

After the initial imports, we define a `3x3` kernel and a `5x5` kernel, and then we load the image in grayscale. After that, we want to convolve the image with each of the kernels. There are several library functions available for such a purpose. NumPy provides the `convolve` function; however, it only accepts one-dimensional arrays. Although the convolution of multidimensional arrays can be achieved with NumPy, it would be a bit complex. SciPy's `ndimage` module provides another `convolve` function, which supports multidimensional arrays. Finally, OpenCV provides a `filter2D` function (for convolution with 2D arrays) and a `sepFilter2D` function (for the special case of a 2D kernel that can be decomposed into two one-dimensional kernels). The preceding code sample illustrates the `ndimage.convolve` function. We will use the `cv2.filter2D` function in other samples in the *Custom kernels – getting convolutions* section of this chapter.

Our script proceeds by applying two HPFs with the two convolution kernels we defined. Finally, we also implement a different method of obtaining an HPF by applying an LPF and calculating the difference with the original image. Let's see how each filter looks. As input, we start with the following photograph:



Figure 3.1a: A photograph of a statue of an angel, with a clear sky in the background. We will perform high-pass (edge detection) and low-pass (blur) filtering on this image

Now, here is a screenshot of the output:

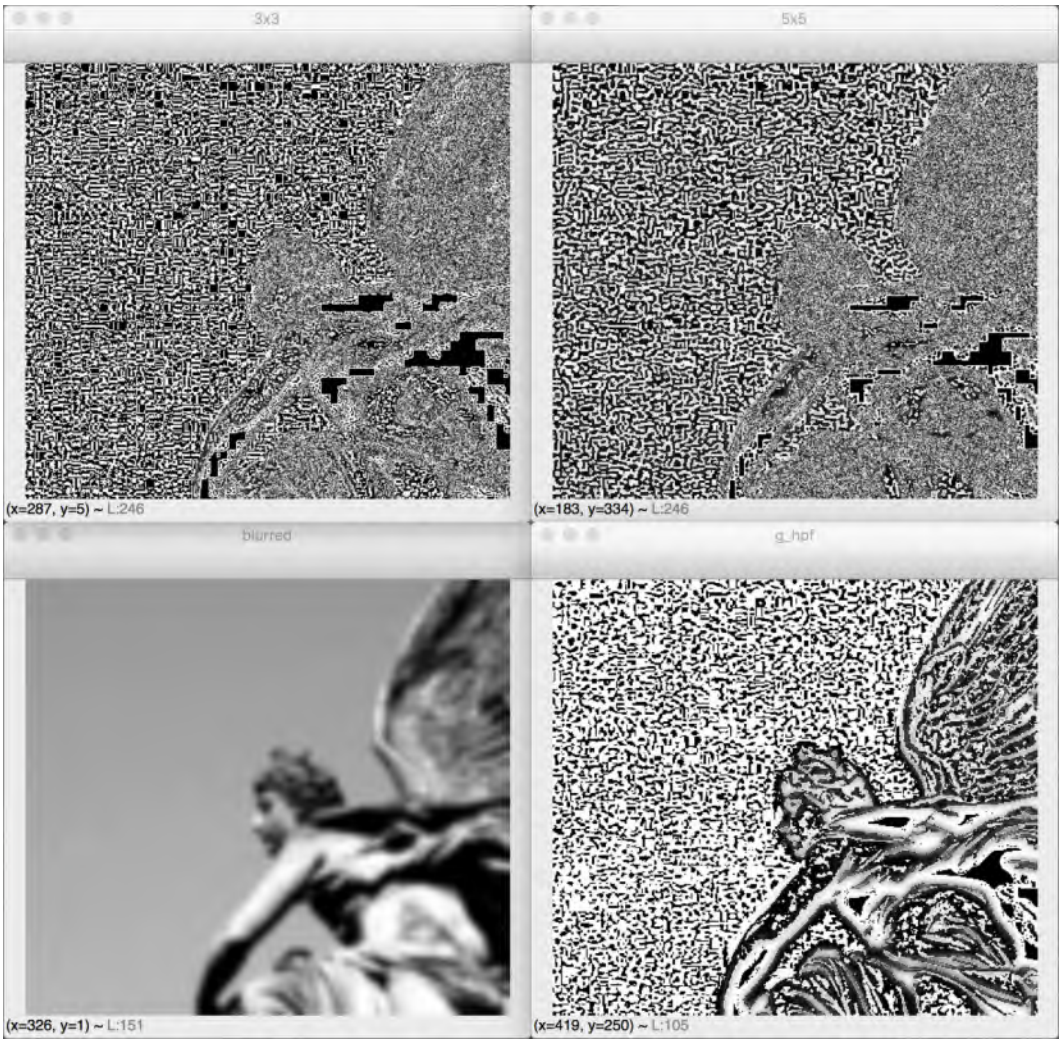


Figure 3.1b: Results of applying high-pass and low-pass filters. Top left: A basic 3x3 high-pass kernel. Top right: A basic 5x5 high-pass kernel. Bottom left: A Gaussian blur low-pass kernel. Bottom right: A differential high-pass filter, based on the preceding Gaussian blur result

You will notice that the differential HPF, as shown in the bottom-right photograph, yields the best edge-finding result. Since this differential method involves a low-pass filter, let's elaborate on that type of filter. If an HPF boosts the intensity of a pixel, given its difference with neighbors, an LPF will smooth or flatten the pixel intensity if the difference from surrounding pixels is lower than a certain threshold. This is used in denoising and blurring. For example,

one of the most popular blurring/smoothing filters, the Gaussian blur, is a low-pass filter that attenuates the intensity of high-frequency signals. The result of the Gaussian blur is shown in the lower-left photograph.

Now that we have tried these filters in a basic example, let's consider how to integrate them into a larger, more interactive application.

Creating modules

Let's revisit the Cameo project that we started in, *Chapter 2*, *Handling Files, Cameras, and GUIs*. We can modify Cameo so that it applies filters to the captured images in real time. As in the case of our `CaptureManager` and `WindowManager` classes, our filters should be reusable outside of Cameo. Thus, we should separate the filters into their own Python module or file.

Let's create a file called `filters.py` in the same directory as `cameo.py`. We need the following `import` statements in `filters.py`:

```
import cv2
import numpy
import utils
```

Let's also create a file called `utils.py` in the same directory. It should contain the following `import` statements:

```
import cv2
import numpy
import scipy.interpolate
```

We will be adding filter functions and classes to `filters.py`, while more general-purpose math functions will go in `utils.py`.

Edge detection

Edges play a major role in both human and computer vision. We, as humans, can easily recognize many object types and their pose just by seeing a backlit silhouette or a rough sketch. Indeed, when art emphasizes edges and poses, it often seems to convey the idea of an archetype, such as Auguste Rodin's *The Thinker* or Joe Shuster's *Superman*. Software, too, can reason about edges, poses, and archetypes. We will discuss these kinds of reasoning in later chapters.

OpenCV provides many edge-finding filters, including `Laplacian`, `Sobel`, and `Scharr`. These filters are supposed to turn non-edge regions into black and turn edge regions into white or saturated colors. However, they are prone to misidentifying noise as edges. This flaw can be mitigated by blurring an image before trying to find its edges. OpenCV also provides many blurring filters, including `blur` (a simple average), `medianBlur`, and `GaussianBlur`. The arguments for the edge-finding and blurring filters vary but always include `ksize`, an odd whole number that represents the width and height (in pixels) of a filter's kernel.

For blurring, let's use `medianBlur`, which is effective in removing digital video noise, especially in color images. For edge-finding, let's use `Laplacian`, which produces bold edge lines, especially in grayscale images. After applying `medianBlur`, but before applying `Laplacian`, we should convert the image from BGR into grayscale.

Once we have the result of `Laplacian`, we can invert it to get black edges on a white background. Then, we can normalize it (so that its values range from 0 to 1) and then multiply it with the source image to darken the edges. Let's implement this approach in `filters.py`:

```
def strokeEdges(src, dst, blurKsize=7, edgeKsize=5):
    if blurKsize >= 3:
        blurredSrc = cv2.medianBlur(src, blurKsize)
        graySrc = cv2.cvtColor(blurredSrc, cv2.COLOR_BGR2GRAY)
    else:
        graySrc = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
    cv2.Laplacian(graySrc, cv2.CV_8U, graySrc, ksize=edgeKsize)
    normalizedInverseAlpha = (1.0 / 255) * (255 - graySrc)
    channels = cv2.split(src)
    for channel in channels:
        channel[:] = channel * normalizedInverseAlpha
    cv2.merge(channels, dst)
```

Note that we allow kernel sizes to be specified as arguments for `strokeEdges`. The `blurKsize` argument is used as `ksize` for `medianBlur`, while `edgeKsize` is used as `ksize` for `Laplacian`. With a typical webcam, a `blurKsize` value of 7 and an `edgeKsize` value of 5 might produce the most pleasing effect. Unfortunately, `medianBlur` is expensive with a large `ksize` argument, such as 7.

Tip

If you encounter performance problems when running `strokeEdges`, try decreasing the `blurKsize` value. To turn off the blur effect, set it to a value of less than 3.

We will see the effect of this filter a little later in this chapter, after we have integrated it into Cameo in the *Modifying the application* section.

Custom kernels – getting convolutions

As we have just seen, many of OpenCV's predefined filters use a kernel. Remember that a kernel is a set of weights that determines how each output pixel is calculated from a neighborhood of input pixels. Another term for a kernel is a **convolution matrix**. It mixes up or convolves the pixels in a region. Similarly, a kernel-based filter may be called a convolution filter.

OpenCV provides a very versatile `filter2D` function, which applies any kernel or convolution matrix that we specify. To understand how to use this function, let's learn about the format of a convolution matrix. It is a 2D array with an odd number of rows and columns. The central element corresponds to a pixel of interest, while the other elements correspond to the neighbors of this pixel. Each element contains an integer or floating-point value, which is a weight that gets applied to an input pixel's value. Consider this example:

```
kernel = numpy.array([[-1, -1, -1],
                      [-1,  9, -1],
                      [-1, -1, -1]])
```

Here, the pixel of interest has a weight of `9` and its immediate neighbors each have a weight of `-1`. For the pixel of interest, the output color will be nine times its input color, minus the input colors of all eight adjacent pixels. If the pixel of interest is already a bit different from its neighbors, this difference becomes intensified. The effect is that the output image looks *sharper*, as the contrast between the neighbors is increased.

Continuing with our example, we can apply this convolution matrix to a source and destination image, respectively, as follows:

```
cv2.filter2D(src, -1, kernel, dst)
```

The second argument specifies the per-channel depth of the destination image (such as `cv2.CV_8U` for 8 bits per channel). A negative value (such as the one being used here) means that the destination image has the same depth as the source image.

Note

For color images, note that `filter2D` applies the kernel equally to each channel. To use different kernels on different channels, we would also have to use OpenCV's `split` and `merge` functions.

Based on this simple example, let's add two classes to `filters.py`. One class, `VConvolutionFilter`, will represent a convolution filter in general. A subclass, `SharpenFilter`, will represent our sharpening filter specifically. Let's edit `filters.py` so that we can implement these two new classes, as follows:

```
class VConvolutionFilter(object):
    """A filter that applies a convolution to V (or all of BGR)."""

    def __init__(self, kernel):
        self._kernel = kernel

    def apply(self, src, dst):
        """Apply the filter with a BGR or gray source/destination."""
        cv2.filter2D(src, -1, self._kernel, dst)

class SharpenFilter(VConvolutionFilter):
    """A sharpen filter with a 1-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[-1, -1, -1],
                              [-1,  9, -1],
                              [-1, -1, -1]])
        VConvolutionFilter.__init__(self, kernel)
```

Note that the weights sum up to `1`. This should be the case whenever we want to leave the image's overall brightness unchanged. If we modify a sharpening kernel slightly so that its weights sum up to `0` instead, we'll have an edge detection kernel that turns edges white and non-edges black. For example, let's add the following edge detection filter to `filters.py`:

```
class FindEdgesFilter(VConvolutionFilter):
    """An edge-finding filter with a 1-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[-1, -1, -1],
                              [-1,  8, -1],
                              [-1, -1, -1]])
        VConvolutionFilter.__init__(self, kernel)
```

Next, let's make a blur filter. Generally, for a blur effect, the weights should sum up to 1 and should be positive throughout the neighborhood. For example, we can take a simple average of the neighborhood as follows:

```
class BlurFilter(VConvolutionFilter):
    """A blur filter with a 2-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[0.04, 0.04, 0.04, 0.04, 0.04],
                              [0.04, 0.04, 0.04, 0.04, 0.04],
                              [0.04, 0.04, 0.04, 0.04, 0.04],
                              [0.04, 0.04, 0.04, 0.04, 0.04],
                              [0.04, 0.04, 0.04, 0.04, 0.04]])
        VConvolutionFilter.__init__(self, kernel)
```

Our sharpening, edge detection, and blur filters use kernels that are highly symmetric. Sometimes, though, kernels with less symmetry produce an interesting effect. Let's consider a kernel that blurs on one side (with positive weights) and sharpens on the other (with negative weights). It will produce a ridged or *embossed* effect, which might be useful for artistic purposes, or for detecting edges on one side of an object but not the opposite side. Here is an implementation that we can add to `filters.py`:

```
class EmbossFilter(VConvolutionFilter):
    """An emboss filter with a 1-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[-2, -1, 0],
                              [-1,  1, 1],
                              [ 0,  1, 2]])
        VConvolutionFilter.__init__(self, kernel)
```

This set of custom convolution filters is very basic. Indeed, it is more basic than OpenCV's ready-made set of filters. However, with a bit of experimentation, you should be able to write your own kernels that produce a unique look.

Modifying the application

Now that we have high-level functions and classes for several filters, it is trivial to apply any of them to the captured frames in Cameo. Let's edit `cameo.py` and add the lines that appear

`highlighted in bold` in the following excerpts. First, we need to add our `filters` module to our list of imports, as follows:

```
import cv2
import filters
from managers import WindowManager, CaptureManager
```

Now, we need to initialize any filter objects we will use. An example of this can be seen in the following modified `__init__` method:

```
class Cameo(object):

    def __init__(self):
        self._windowManager = WindowManager('Cameo', self.onKeypress)
        self._captureManager = CaptureManager(
            cv2.VideoCapture(0), self._windowManager, True)
        self._curveFilter = filters.BGRPortraCurveFilter()
```

Finally, we need to modify the `run` method in order to apply our choice of filters. Refer to the following example:

```
    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            frame = self._captureManager.frame

            if frame is not None:

                filters.strokeEdges(frame, frame)
                self._curveFilter.apply(frame, frame)

            self._captureManager.exitFrame()
            self._windowManager.processEvents()

    # ... The rest is the same as in Chapter 2
```

Here, we have applied two effects: stroking the edges and emulating the colors of a brand of photo film called Kodak Portra. Feel free to modify the code to apply any filters you like.

> **Note**
>
> For details about how to implement the Portra film emulation effect, refer to *Appendix A, Bending Color Space with a Curves Filter*.

Here is a screenshot from Cameo with stroked edges and Portra-like colors:



Figure 3.2: A screenshot from Cameo, with two filters applied: stroked edges and Portra-like colors

Now that we have sampled some of the visual effects that we can achieve with simple filters, let's consider how we can use other simple functions for analytical purposes – specifically, the detection of edges and shapes.

Edge detection with Canny

OpenCV offers a handy function called `Canny` (after the algorithm's inventor, John F. Canny), which is very popular not only because of its effectiveness, but also because of the simplicity of its implementation in an OpenCV program since it is a one-liner:

```
import cv2
import numpy as np

img = cv2.imread("../images/statue_small.jpg",
                 cv2.IMREAD_GRAYSCALE)
canny_img = cv2.Canny(img, 200, 300)
cv2.imshow("canny", canny_img)
```

```
cv2.waitKey()
cv2.destroyAllWindows()
```

The result is a very clear identification of the edges:



Figure 3.3: Results of applying Canny edge detection

The Canny edge detection algorithm is complex but also quite interesting. It is a three-step process:

1. Denoise the image and calculate the gradients using a Sobel filter.
2. Apply **non-maximum suppression** (**NMS**) on the edges. Basically, this means that the algorithm selects the best edges from a set of overlapping edges. We'll discuss the concept of NMS in detail in *Chapter 7, Building Custom Object Detectors*.
3. Apply a double threshold to all the detected edges to eliminate any false positives. If an edge pixel passes a high (stringent) threshold, it is immediately accepted. Otherwise, if it passes a low (less stringent) threshold and is connected to another accepted edge pixel, it is also accepted.

After finding Canny edges, we can do further analysis of the edges in order to determine whether they match a common shape, such as a line or a circle. The Hough transform is one algorithm that uses Canny edges in this way. We will experiment with it later in this chapter, in the *Detecting lines, circles, or other shapes* section.

For now, we will examine other ways of analyzing shapes, not based on edge detection but rather on the concept of finding a blob of similar pixels.

Contour detection

A vital task in computer vision is contour detection. We want to detect contours or outlines of subjects contained in an image or video frame, not only as an end in itself but also as a step toward other operations. These operations are, namely, computing bounding polygons, approximating shapes, and generally calculating **regions of interest (ROIs)**.
ROIs considerably simplify interaction with image data because a rectangular region in NumPy is easily defined with an array slice. We will be using contour detection and ROIs a lot when we explore the concepts of object detection (including face detection) and object tracking in later chapters.

Let's familiarize ourselves with the contour detection API via an example:

```
import cv2
import numpy as np

# Create a black image.
img = np.zeros((300, 300), dtype=np.uint8)

# Draw a square in two shades of gray.
img[50:150, 50:150] = 160
img[70:150, 70:150] = 128

# Apply a threshold so that the square becomes uniformly white.
ret, thresh = cv2.threshold(img, 127, 255, 0)

# Find the contour of the thresholded square.
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
                                       cv2.CHAIN_APPROX_SIMPLE)
color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
img = cv2.drawContours(color, contours, -1, (0,255,0), 2)
cv2.imshow("contours", color)
cv2.waitKey()
cv2.destroyAllWindows()
```

First, we create an empty black image that is 300 x 300 pixels in size. Then, in the upper-left part of this image, we draw a two-toned gray square by utilizing the ability of `arrays` to assign values on a slice.

Then, we threshold the image and call the `findContours` function. This function has three parameters: the input image, the hierarchy type, and the contour approximation method. The

second parameter specifies the type of hierarchy tree returned by the function. One of the supported values is `cv2.RETR_TREE`, which tells the function to retrieve the entire hierarchy of external and internal contours. These relationships may matter if we are searching for smaller objects (or smaller regions) inside larger objects (or larger regions). If you only want to retrieve the most external contours, use `cv2.RETR_EXTERNAL`. This may be a good choice in cases where the objects appear on a plain background and we do not care about finding objects within objects.

Referring back to the code sample, note that the `findContours` function returns two elements: the contours and their hierarchy. We use the contours to draw green outlines on the color version of the image. Finally, we display the image.

The result is a gray square with its contour drawn in green – a spartan scene, but effective in demonstrating the concept! Here it is:

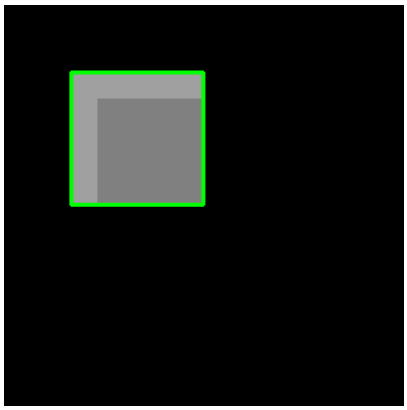


Figure 3.4: The result of contour detection using a simple image of a gray square on a black background. The detected contour is drawn in green

Let's move on to more meaningful examples.

Bounding box, minimum area rectangle, and minimum enclosing circle

Finding the contours of a square is a simple task; irregular, skewed, and rotated shapes bring out the full potential of OpenCV's `cv2.findContours` function. Let's take a look at the following image, which is an illustration of a mythical object called Mjölnir or Thor's Hammer:



Figure 3.5a: A drawing of Mjölnir, or Thor's Hammer, a prominent symbol in Norse mythology. We will apply contour detection to this image

For many applications, we might want to find a simple geometric shape that approximates the bounds of the subject. Three commonly used bounding shapes are the bounding box, the minimum enclosing rectangle, and the enclosing circle. (According to OpenCV's terminology, a bounding box is an upright rectangle, but a minimum enclosing rectangle may be rotated in order to fit the subject more snugly.) The `cv2.findContours` function, in conjunction with a few other OpenCV utilities, makes it very easy to find these bounding shapes. First, the following code reads an image from a file, converts it into grayscale, applies a threshold to the grayscale image, and finds the contours in the thresholded image:

```
import cv2
import numpy as np

img = cv2.pyrDown(cv2.imread("../images/hammer.jpg"))

ret, thresh = cv2.threshold(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY),
                            127, 255, cv2.THRESH_BINARY)
contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
                                  cv2.CHAIN_APPROX_SIMPLE)
```

Now, for each contour, we can find and draw the bounding box, the minimum enclosing rectangle, and the minimum enclosing circle, as shown in the following code:

```
for c in contours:
    # find bounding box coordinates
    x, y, w, h = cv2.boundingRect(c)
    cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)

    # find minimum area
    rect = cv2.minAreaRect(c)
    # calculate coordinates of the minimum area rectangle
    box = cv2.boxPoints(rect)
    # normalize coordinates to integers
    box = box.astype(int)
    # draw contours
    cv2.drawContours(img, [box], 0, (0, 0, 255), 3)

    # calculate center and radius of minimum enclosing circle
    (x, y), radius = cv2.minEnclosingCircle(c)
    # cast to integers
    center = (int(x), int(y))
    radius = int(radius)
    # draw the circle
    img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

Finally, we can use the following code to draw the contours and show the image in a window until the user presses a key:

```
cv2.drawContours(img, contours, -1, (255, 0, 0), 1)
cv2.imshow("contours", img)

cv2.waitKey()
cv2.destroyAllWindows()
```

Note that contour detection is performed on a thresholded image, so color information is already lost at this stage, but we draw on the original color image and then show the results in color.

Let's go back and look more closely at the steps we performed in the preceding `for` loop, where we process each detected contour. First, we calculate a simple bounding box:

```
    x, y, w, h = cv2.boundingRect(c)
```

This is a pretty straightforward conversion of contour information into the (x, y) coordinates, width, and height of the rectangle. Drawing this rectangle is an easy task and can be done using the following code:

```
    cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)
```

Next, we calculate the minimum-area rectangle enclosing the subject:

```
    rect = cv2.minAreaRect(c)
    box = cv2.boxPoints(rect)
    box = box.astype(int)
```

The mechanism being used here is particularly interesting: OpenCV does not have a function to calculate the coordinates of the minimum rectangle vertices directly from the contour information. Instead, we calculate the minimum rectangle area, and then calculate the vertices of this rectangle. Note that the calculated vertices are floats, but pixels are accessed with integers (you can't access a *portion* of a pixel for the purposes of OpenCV's drawing functions), so we need to perform this conversion. Next, we draw the box, which gives us the perfect opportunity to introduce the `cv2.drawContours` function:

```
    cv2.drawContours(img, [box], 0, (0,0, 255), 3)
```

This function – like all of OpenCV's drawing functions – modifies the original image. Note that it takes an array of contours in its second parameter so that you can draw a number of contours in a single operation. Therefore, if you have a single set of points representing a contour polygon, you need to wrap these points in an array, exactly like we did with our box in the preceding example. The third parameter of this function specifies the index of the `contours` array that we want to draw: a value of `-1` will draw all contours; otherwise, a contour at the specified index in the `contours` array (the second parameter) will be drawn.

Most drawing functions take the color of the drawing (as a BGR tuple) and its thickness (in pixels) as the last two parameters.

The last bounding contour we're going to examine is the minimum enclosing circle:

```
    (x, y), radius = cv2.minEnclosingCircle(c)
    center = (int(x), int(y))
    radius = int(radius)
    img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

The only peculiarity of the `cv2.minEnclosingCircle` function is that it returns a two-element tuple, of which the first element is a tuple itself, representing the coordinates of the circle's center, and the second element is the radius of this circle. After converting all these values into integers, drawing the circle is quite a trivial operation.

When we apply the preceding code to the original image, the final result looks like this:

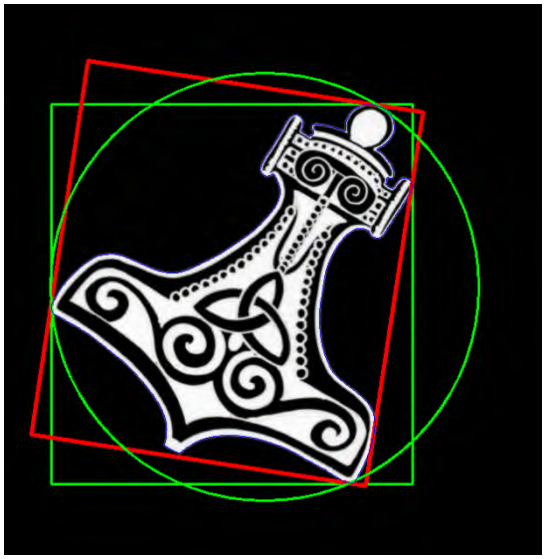


Figure 3.5b: Results of contour detection followed by three kinds of enclosing shape detection: bounding box, minimum area rectangle, and minimum enclosing circle

This is a good result insofar as the circle and rectangles fit tightly around the object. Obviously, though, the object is not circular or rectangular, so we could achieve a tighter fit with various other shapes. Let's do that next.

Convex contours and the Douglas-Peucker algorithm

When working with contours, we may encounter subjects with diverse shapes, including convex ones. A convex shape is one where there are no two points within this shape whose connecting line goes outside the perimeter of the shape itself.

The first facility that OpenCV offers to calculate the approximate bounding polygon of a shape is `cv2.approxPolyDP`. The `DP` in the function name stands for the **Douglas-Peucker** algorithm. This function takes three parameters:

- A contour.
- An epsilon value representing the maximum discrepancy between the original contour and the approximated polygon (the lower the value, the closer the approximated polygon will be to the original contour).
- A Boolean flag. If it is `True`, it signifies that the polygon is closed.

The epsilon value is of vital importance to obtain a useful contour, so let's understand what it represents. Epsilon is the maximum difference between the approximated polygon's perimeter and the original contour's perimeter. The smaller this difference is, the more similar the approximated polygon will be to the original contour.

You may ask yourself why we need an approximate polygon when we have a contour that is already a precise representation. The answer to this is that a polygon is a set of straight lines, and many computer vision tasks become simpler if we can define polygons so that they delimit regions for further manipulation and processing.

Now that we know what an epsilon is, we need to obtain contour perimeter information as a reference value. This can be obtained with the `cv2.arcLength` function of OpenCV:

```
epsilon = 0.01 * cv2.arcLength(cnt, True)
approx = cv2.approxPolyDP(cnt, epsilon, True)
```

Effectively, we're instructing OpenCV to calculate an approximated polygon whose perimeter can only differ from the original contour by an epsilon ratio – specifically, 1% of the original arc length.

OpenCV also offers a `cv2.convexHull` function for obtaining processed contour information for convex shapes. This is a straightforward one-line expression:

```
hull = cv2.convexHull(cnt)
```

Let's combine the original contour (as a green outline), approximated polygon contour (cyan), and the convex hull (red) into one image to observe the differences between them. To simplify things, we will draw the contours on top of a black background so that the original subject is not visible, but its contours are:



Figure 3.5c: Results of contour detection followed by convex hull detection and polygon approximation

As you can see, the convex hull (red) surrounds the entire subject, the approximated polygon (cyan) is the innermost polygon shape, and in between the two is the original contour (green), mainly composed of arcs.

By combining all the preceding steps into one script that loads a file, finds contours, approximates the contours as a polygon, finds a convex hull, and displays a visualization, we have the following code:

```
import cv2
import numpy as np

img = cv2.pyrDown(cv2.imread("../images/hammer.jpg"))

ret, thresh = cv2.threshold(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY),
                            127, 255, cv2.THRESH_BINARY)

contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
                                  cv2.CHAIN_APPROX_SIMPLE)

black = np.zeros_like(img)
for cnt in contours:
    epsilon = 0.01 * cv2.arcLength(cnt,True)
```

```
        approx = cv2.approxPolyDP(cnt,epsilon,True)
        hull = cv2.convexHull(cnt)
        cv2.drawContours(black, [cnt], -1, (0, 255, 0), 2)
        cv2.drawContours(black, [approx], -1, (255, 255, 0), 2)
        cv2.drawContours(black, [hull], -1, (0, 0, 255), 2)

    cv2.imshow("hull", black)
    cv2.waitKey()
    cv2.destroyAllWindows()
```

Code like this works well on simple images, in which we have only one or a few objects, and only a few colors that are easily separated by thresholds. Unfortunately, color thresholding and contour detection are less effective on complex images that contain several objects or multicolored objects. For these more challenging cases, we will have to look at more complex algorithms.

Detecting lines, circles, and other shapes

Detecting edges and finding contours are not only common and important tasks in their own right; they also form the basis of other complex operations. Line and shape detection go hand in hand with edge and contour detection, so let's examine how OpenCV implements these.

The traditional theory behind line and shape detection has its foundation in a technique called the Hough transform, invented by Richard Duda and Peter Hart, who extended and generalized the work that was done by Paul Hough in the early 1960s. We will take a look at OpenCV's API for Hough transforms.

We will also compare the results of Hough transforms with results from a more recent set of algorithms called Edge Drawing (ED). The latter are implemented in `opencv_contrib` (specifically, in the `ximgproc` module) as a class called `EdgeDrawing`, which is capable of detecting and drawing edge segments, lines, circles, and ellipses.

> **Note**
>
> The **Edge Drawing** (**ED**) algorithms are the work of Cuneyt Akinlar and Cihan Topal. Their implementation in `opencv_contrib` is the work of Suleyman Turkmen, who is also one of our technical reviewers for this book, and he has kindly contributed this chapter's sample code for `EdgeDrawing`.

Let's see these algorithms in action, first in an example of line detection and then in an *out-of-this-world* example featuring circles and ellipses.

Detecting lines

First of all, let's detect some lines. We can do this with either the `HoughLines` function or the `HoughLinesP` function. The former uses the standard Hough transform, while the latter uses the probabilistic Hough transform (hence the `P` in the name). The probabilistic version is so-called because it only analyzes a subset of the image's points and estimates the probability that these points all belong to the same line. This implementation is an optimized version of the standard Hough transform; it is less computationally intensive and executes faster. `HoughLinesP` is implemented so that it returns the two endpoints of each detected line segment, whereas `HoughLines` is implemented so that it returns a representation of each line as a single point and an angle, without information about endpoints.

Let's take a look at a very simple example:

```
import cv2
import numpy as np

img = cv2.imread('../images/houghlines5.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 120)

lines = cv2.HoughLinesP(edges, rho=1,
                        theta=np.pi/180.0,
                        threshold=20,
                        minLineLength=40,
                        maxLineGap=5)
for line in lines:
    line = line.squeeze()
    x1, y1, x2, y2 = line
    cv2.line(img, (x1, y1), (x2, y2), (0, 255, 0), 2)

cv2.imshow("edges", edges)
cv2.imshow("lines", img)
cv2.waitKey()
cv2.destroyAllWindows()
```

A crucial part of this simple script, as part of the `HoughLines` function call, is setting the minimum line length (shorter lines will be discarded) and the maximum line gap, which is the maximum size of a gap in a line before the two segments start being considered as separate lines.

Also, note that the `HoughLinesP` function takes a single channel binary image, which is processed through the Canny edge detection filter. Canny is not a strict requirement, but an image that has been denoised and only represents edges is the ideal source for a Hough transform, so you will find this to be a common practice.

The parameters of `HoughLinesP` are as follows:

- The binary image from Canny or another edge detection filter.
- The resolution or step size to use when searching for lines. `rho` is the positional step size in pixels, while `theta` is the rotational step size in radians. For example, if we specify `rho=1` and `theta=np.pi/180.0`, we search for lines that are separated by as little as 1 pixel and 1 degree.
- The `threshold`, which represents the threshold below which a line is discarded. The Hough transform works with a system of bins and votes, with each bin representing a line, so if a candidate line has at least the `threshold` number of votes, it is retained; otherwise, it is discarded.
- An optional argument, `lines`, which we do not use here and which is not really useful in the function's Python version. The `lines` argument can be used to provide a list or array in which `HoughLinesP` will put the resulting lines; otherwise, it will return a new list.
- `minLineLength` and `maxLineGap`, which we mentioned previously.

The following screenshot shows the output of our line detection script, using an image of a train platform as input:

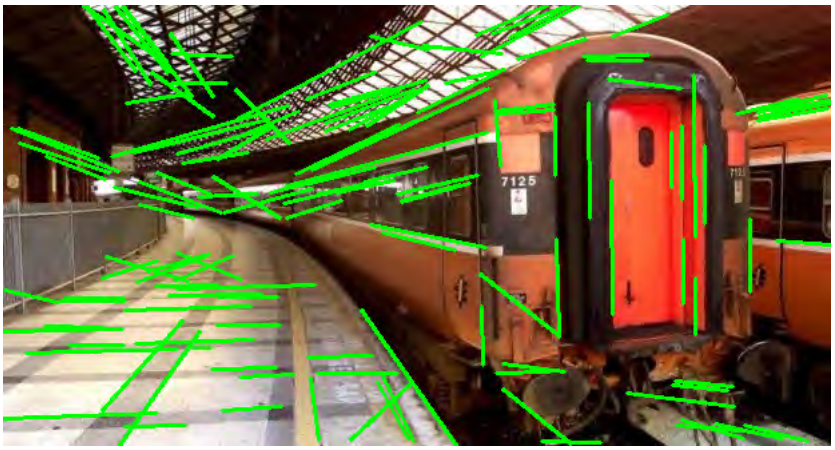


Figure 3.6a: Results of Hough line detection on an image of a train platform

Among other things, a segment of the platform's edge is detected as a line. Here, we begin to get an idea of how a seemingly basic concept such as line detection can have important practical applications.

Can `EdgeDrawing` do better? Let's see in a new script.

First, we load the input image as before:

```
import cv2
import numpy as np

img = cv2.imread('../images/houghlines5.jpg')
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Next, we create an instance of the `EdgeDrawing` class and set its parameters, specifically the minimum line length:

```
edge_drawing = cv2.ximgproc.createEdgeDrawing()

edge_drawing_params = cv2.ximgproc_EdgeDrawing_Params()
edge_drawing_params.MinLineLength = 20

edge_drawing.setParams(edge_drawing_params)
```

We proceed to detect edge segments and then continuous lines. The order matters: with `EdgeDrawing`, detection of edge segments is a prerequisite for any kind of shape detection. Here is the relevant code:

```
# Detect edges.
edge_drawing.detectEdges(gray_img)

# Detect lines based on the edges and the specified parameters.
lines = edge_drawing.detectLines()
```

We iterate over the detected lines and draw each one in green:

```
# Draw the detected lines.
if lines is not None:
    lines = np.uint16(np.around(lines))
    for line in lines:
        line = line.squeeze()
```

```
        cv2.line(img, (line[0], line[1]),
                 (line[2], line[3]), (0, 255, 0), 2, cv2.LINE_AA)
```

Finally, we show the resulting visualization of the detected lines:

```
cv2.imshow("Detected lines", img)
cv2.waitKey()
cv2.destroyAllWindows()
```

Running this script, we get the following image as output:



Figure 3.6b: Results of ED line detection on an image of a train platform

This result with `EdgeDrawing` is noticeably better than our previous result with `HoughLinesP`. For example, with `EdgeDrawing`, the entire edge of the platform is detected as a set of lines.

Now, what about other shapes?

Detecting circles

OpenCV also has a function for detecting circles, called `HoughCircles`. It works in a very similar fashion to `HoughLines`, but where `minLineLength` and `maxLineGap` were the parameters used to discard or retain lines, `HoughCircles` instead lets us specify a minimum distance between a circle's centers as well as minimum and maximum values for a circle's radius. Here is the obligatory example:

```
import cv2
import numpy as np

planets = cv2.imread('../images/planet_glow.jpg')
gray_img = cv2.cvtColor(planets, cv2.COLOR_BGR2GRAY)
gray_img = cv2.medianBlur(gray_img, 5)

circles = cv2.HoughCircles(gray_img, cv2.HOUGH_GRADIENT,
                           1, 120, param1=90, param2=40,
                           minRadius=0, maxRadius=0)

if circles is not None:
    circles = np.uint16(np.around(circles))

    for i in circles[0,:]:
        # draw the outer circle
        cv2.circle(planets, (i[0], i[1]), i[2], (0, 255, 0), 2)

        # draw the center of the circle
        cv2.circle(planets,(i[0], i[1]), 2, (0, 0, 255), 3)

cv2.imwrite("planets_circles.jpg", planets)
cv2.imshow("HoughCirlces", planets)
cv2.waitKey()
cv2.destroyAllWindows()
```

Note that we do not call the `Canny` function here because, when we call `HoughCircles` with the `cv2.HOUGH_GRADIENT` or `cv2.HOUGH_GRADIENT_ALT` option, the latter function already applies the Canny algorithm internally, using the value of `param1` as the first Canny threshold and half this value as the second Canny threshold. Also, with `cv2.HOUGH_GRADIENT`, the value of `param2` is a threshold for weeding out overlapping circle detections; a higher value leads to fewer overlapping detections. With `cv2.HOUGH_GRADIENT_ALT`, the value of `param2` is a threshold for circle *perfectness*, where 1.0 means that only an ideal circle will pass, 0.9 is a good value for general use, and lower values may help in the detection of smaller circles.

Here is a visual representation of the result:

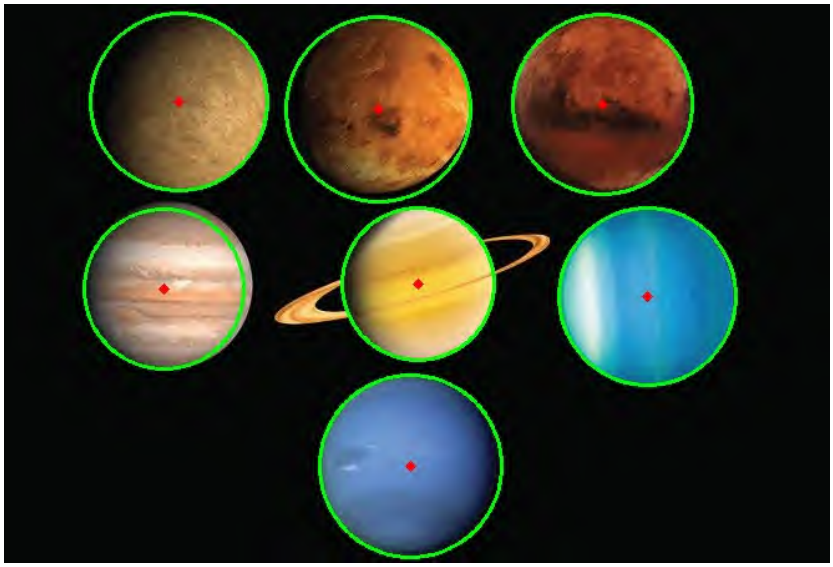


Figure 3.7a: Results of Hough circle detection on a stylized image of the planets

The detected circles' perimeters are drawn in green, and the center points are drawn in red. All planets are detected, though in some cases the circle is slightly too small, too large, or off-center. `HoughCircles` is unable to detect Saturn's rings, since these are not circles; they are, however, ellipses.

Let's see what `EdgeDrawing` can do with the same input image. First, we load and blur the image in the same manner as we previously did:

```
import random as rng
import cv2
import numpy as np

planets = cv2.imread("../images/planet_glow.jpg")
gray_img = cv2.cvtColor(planets, cv2.COLOR_BGR2GRAY)
gray_img = cv2.medianBlur(gray_img, 5)
```

We are going to draw two visualizations: first, a visualization of detected edge segments; and second, a visualization of detected ellipses based on the edge segments. As a starting point for these visualizations, we create two more copies of the input image:

```
segments_viz = planets.copy()
ellipses_viz = planets.copy()
```

We create an instance of `EdgeDrawing` and use it to detect edge segments (a necessary first step) and then ellipses:

```
edge_drawing = cv2.ximgproc.createEdgeDrawing()

# Detect edges and get the resulting edge segments.
edge_drawing.detectEdges(gray_img)
segments = edge_drawing.getSegments()

# Detect circles and ellipses based on the detected edges.
ellipses = edge_drawing.detectEllipses()
```

We iterate over the edge segments and draw each one in a random color:

```
# Draw the detected edge segments.
for segment in segments:
    color = (rng.randint(16, 256),
             rng.randint(16, 256),
             rng.randint(16, 256))
    cv2.polylines(segments_viz, [segment], False, color, 1, cv2.LINE_8)
```

Similarly, we iterate over the ellipses. If an ellipse is a circle, we draw it in green; otherwise, we draw it in red:

```
# Draw the detected circles and ellipses.
if ellipses is not None:
    for ellipse in ellipses:
        ellipse = ellipse.squeeze()
        center = (int(ellipse[0]), int(ellipse[1]))
        axes = (int(ellipse[2] + ellipse[3]),
                int(ellipse[2] + ellipse[4]))
        angle = ellipse[5]

        if ellipse[0][2] == 0:  # Ellipse
            color = (0, 0, 255)
        else:  # Circle
            color = (0, 255, 0)
        cv2.ellipse(ellipses_viz, center, axes, angle, 0, 360,
                    color, 2, cv2.LINE_AA)
```

Finally, we save and show our two visualizations:

```
cv2.imwrite("planets_edge_drawing_segments.jpg", segments_viz)
cv2.imwrite("planets_edge_drawing_ellipses.jpg", ellipses_viz)

cv2.imshow("Detected edge segments", segments_viz)
cv2.imshow("Detected circles (green) and ellipses (red)", ellipses_viz)

cv2.waitKey()
cv2.destroyAllWindows()
```

Looking at the visualization of edge segments, we can see many semi-elliptical and semi-circular segments:

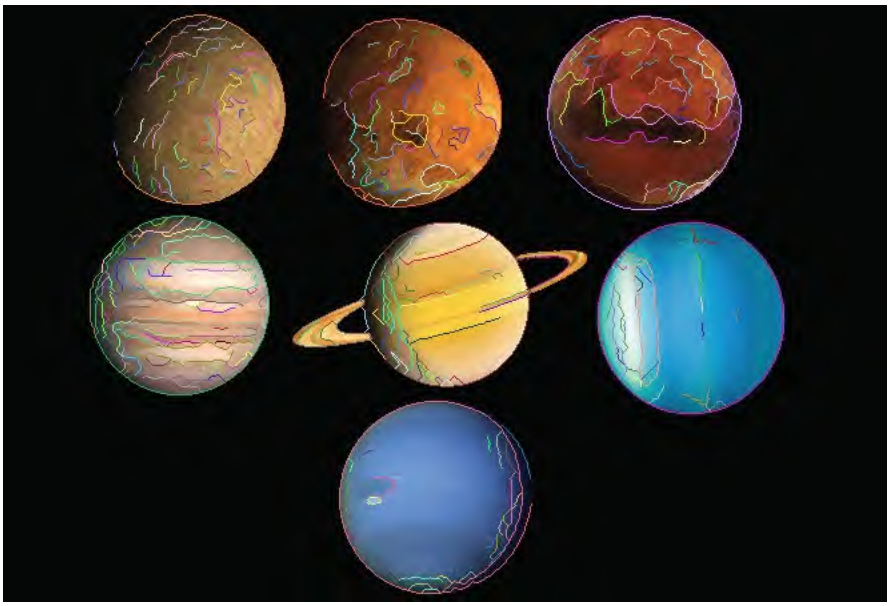


Figure 3.7b: Results of ED edge segment detection on a stylized image of the planets

The final result fills in the gaps to produce full ellipses and full circles:

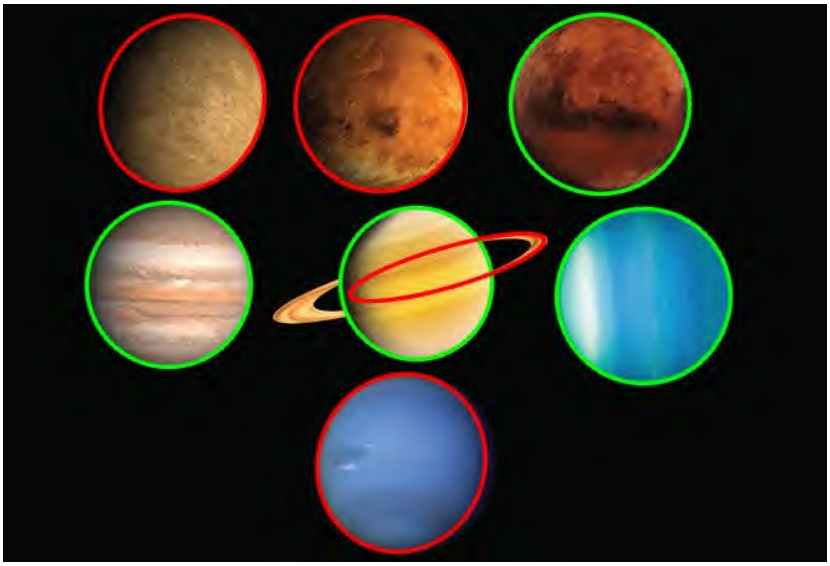


Figure 3.7c: Results of ED ellipse detection on a stylized image of the planets. Detected circles are drawn in green and other detected ellipses are drawn in red

Compared to `HoughCircles`, `EdgeDrawing` fits the circles or ellipses more tightly to the planets' visible bounds. Also, note that `EdgeDrawing` has partly detected Saturn's rings as a rotated ellipse. Overall, `EdgeDrawing` is the winner.

Detecting other shapes

OpenCV's implementations of the Hough transform are limited to detecting lines and circles. Similarly, the `EdgeDrawing` class specializes in lines and ellipses. However, we already implicitly explored shape detection in general when we talked about `approxPolyDP`. This function allows for the approximation of polygons, so if your image contains polygons, they will be accurately detected through the combined use of `cv2.findContours` and `cv2.approxPolyDP`.

Summary

At this point, you should have gained a good understanding of color models, the Fourier transform, and several kinds of filters that have been made available by OpenCV to process images.

You should also be proficient in detecting edges, lines, circles, and shapes in general. Additionally, you should be able to find contours and exploit the information they provide about the subjects contained in an image. These concepts are complementary to the next chapter's topics – namely, the segmentation of an image according to depth and estimating the distance of a subject in an image.

4

Depth Estimation and Segmentation

This chapter begins by showing you how to use data from a depth camera to identify foreground and background regions, so that we can limit an effect to only the foreground or only the background.

After covering depth cameras, the chapter proceeds with other techniques for depth estimation, namely, **stereo imaging** and **structure from motion** (**SfM**). The latter techniques do not necessarily require a specialized depth camera; instead, they rely on capturing images of a subject from multiple perspectives with one or more ordinary cameras.

Finally, the chapter covers segmentation techniques that allow us to extract foreground objects from a single image. By the end of the chapter, you will learn several ways to segment an image into multiple depths or multiple objects. Specifically, we will cover the following topics:

- Using an OpenNI-compatible depth camera to capture depth maps, point cloud maps, disparity maps, images based on visible light, and images based on infrared light
- Understanding an infrared depth camera as a type of night-vision camera
- Converting 10-bit images to 8-bit images
- Converting a disparity map into a mask that differentiates between foreground and background regions
- Using either stereo imaging or SfM to create a disparity map, based on the **semiglobal block matching** (**SGBM**) algorithm
- Using the `depthai` Python module for stereo imaging and SGBM computation on an OAK-D device

- Using the GrabCut algorithm to segment an image into foreground and background regions
- Using the Watershed algorithm to segment an image into multiple regions based on colors and ridges. These regions might be different objects.
- Using **Hierarchical Feature Selection (HFS)** to segment an image into multiple regions based on colors and higher-level features. These regions might be different objects.

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Some parts of the chapter use a depth camera. Two different kinds of depth cameras are covered:

1. OpenNI-compatible depth cameras, such as Asus Xtion PRO, are supported via OpenCV's optional support for OpenNI 2. Please refer back to *Chapter 1, Setting Up OpenCV*, for instructions on how to build and install OpenCV with OpenNI 2 support.
2. The OAK-D line of depth cameras from Luxonis are supported via the `depthai` Python module. To install `depthai` and other dependencies, run `$ pip install blobconverter depthai==2.32.0.0` (or `$ pip3 install blobconverter depthai==2.32.0.0`, depending on your Python environment). Note that we are specifically installing `depthai` version 2.32.0.0 because our book's scripts have been tested with this version, and `depthai` has relatively frequent API changes, which can break compatibility.

> **Note**
>
> On Linux, to use OAK-D cameras, you need to configure the system's USB device policies in order to whitelist this family of devices. To do so, run the following commands:
>
> ```
> $ echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="03e7",
> MODE="0666"' | sudo tee /etc/udev/rules.d/80-movidius.rules
> $ sudo udevadm control --reload-rules && sudo udevadm trigger
> ```
>
> Then, if the camera is already plugged into the computer's USB port, disconnect and reconnect it. You only need to perform these configuration steps once.

For OAK-D devices, you need a USB3 connection. The device and the `depthai` module are quite fussy about the quality of the connection. If you encounter the error `X_LINK_ERROR` when

attempting to capture frames, this may mean that the connection is poor. To potentially resolve the problem, try a different USB3 port or cable, and try removing any external USB3 hub that you may be using as an intermediary between the OAK-D device and the host computer.

The chapter also uses `matplotlib` to make charts. To install `matplotlib`, run `$ pip install matplotlib` (or `$ pip3 install matplotlib`, depending on your Python environment).

The completed code for this chapter can be found in this book's GitHub

repository in the `chapter04` folder. Sample images are in the repository in the `images` folder.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Creating modules

To help us build an interactive demo with an OpenNI-compatible depth camera, we will reuse much of the Cameo project that we developed in *Chapter 2*, *Handling Files, Cameras, and GUIs*, and *Chapter 3*, *Processing Images with OpenCV*. As you will recall, we designed Cameo to support various kinds of input, so we can easily adapt it to support OpenNI-compatible depth cameras in particular. We will add code that analyzes the depth layers in an image in order to find the main region, such as the face of a person sitting in front of the camera. Having found this region, we will paint everything else black. This type of effect is sometimes used in chat applications to hide the background so that users have more privacy.

Some of the code to manipulate depth-camera data will be reusable outside `Cameo.py`, so we should separate it into a new module. Let's create a `depth.py` file in the same directory as `Cameo.py`. We need the following `import` statement in `depth.py`:

```
import numpy
```

Our application will use depth-related functionality, so let's add the following `import` statement to `Cameo.py`:

```
import depth
```

We will also modify `CaptureManager.py`, but we do not need to add any new `import` statements to it.

Now that we have taken a brief look at the modules that we will create or modify, let's go deeper into the subject of depth.

Capturing frames from an OpenNI-compatible depth camera

Back in *Chapter 2*, *Handling Files, Cameras, and GUIs*, we discussed the concept that a computer can have multiple video capture devices and each device can have multiple channels. Suppose a given device is a depth camera. Each channel might correspond to a different lens and sensor, or perhaps a different kind of data such as a normal color image versus a depth map. OpenCV, via its optional support for OpenNI 2, allows us to request any of the following channels from a depth camera (though a given camera might support only some of these channels):

- `cv2.CAP_OPENNI_DEPTH_MAP`: This is a **depth map**—a grayscale image in which each pixel value is the estimated distance from the camera to a surface. Specifically, each pixel value is a 16-bit unsigned integer representing a depth measurement in millimeters.
- `cv2.CAP_OPENNI_POINT_CLOUD_MAP`: This is a **point cloud map**—a color image in which each color corresponds to an *x*, *y*, or *z* spatial dimension. Specifically, the channel yields a BGR image, where B is *x* (blue is right), G is *y* (green is up), and R is *z* (red is deep), from the camera's perspective. Each value is a 32-bit float representing a measurement in meters.
- `cv2.CAP_OPENNI_DISPARITY_MAP` or `cv2.CAP_OPENNI_DISPARITY_MAP_32F`: These are **disparity maps**—grayscale images in which each pixel value is the stereo disparity of a surface. To conceptualize stereo disparity, let's suppose we overlay two images of a scene, shot from different viewpoints. The result would be similar to seeing double. For points on any pair of twin objects in the scene, we can measure the distance in pixels. This measurement is the stereo disparity. Nearby objects exhibit greater stereo disparity than far-off objects. Thus, nearby objects appear brighter in a disparity map. `cv2.CAP_OPENNI_DISPARITY_MAP` is a disparity map with 8-bit unsigned integer values and `cv2.CAP_OPENNI_DISPARITY_MAP_32F` is a disparity map with 32-bit floating-point values.
- `cv2.CAP_OPENNI_VALID_DEPTH_MASK`: This is a **valid depth mask** that shows whether the depth information at a given pixel is believed to be valid (shown by a non-zero value) or invalid (shown by a value of zero). For example, if the depth camera depends on an infrared illuminator (an infrared light or flash), depth information is invalid in regions that are occluded (shadowed) from this light.

- `cv2.CAP_OPENNI_BGR_IMAGE`: This is an ordinary BGR image from a camera that captures visible light. Each pixel's B, G, and R values are unsigned 8-bit integers.
- `cv2.CAP_OPENNI_GRAY_IMAGE`: This is an ordinary monochrome image from a camera that captures visible light. Each pixel value is an unsigned 8-bit integer.
- `cv2.CAP_OPENNI_IR_IMAGE`: This is a monochrome image from a camera that captures **infrared (IR)** light, specifically the **near infrared (NIR)** part of the spectrum. Each pixel value is an unsigned 16-bit integer. Typically, the camera will not actually use this entire 16-bit range but, instead, just a portion of it such as a 10-bit range; still, the data type is a 16-bit integer. Although NIR light is invisible to the human eye, it is physically quite similar to red light. Thus, NIR images from a camera do not necessarily look strange to a human being. However, a typical depth camera not only captures NIR light, but also projects a grid-like pattern of NIR lights for the benefit of the depth-finding algorithm. For example, looking at the depth camera's NIR image, we might see a recognizable face, yet the face might appear to be dotted with bright white lights. This is how the NIR lights appear to the camera, though they are invisible to the unaided human eye. Indeed, thanks to its onboard NIR lighting, the depth camera can see even in a room that appears entirely dark to us. Conversely, in a sunlit outdoor scene, the depth camera may function poorly because of interference from the strong NIR component of sunlight. Just as we cannot easily see a flashlight beam on a sunny day, the depth camera cannot easily see its NIR pattern under these conditions.

Note

For OAK-D depth cameras, the `depthai` Python module supports some of the same image types, though with a different API. We will look at the specifics of `depthai` usage later in this chapter, in the section *Stereo imaging with an OAK-D camera*.

Let's consider samples of some of these image types. The following screenshot shows a point cloud map of a man sitting behind a sculpture of a cat:

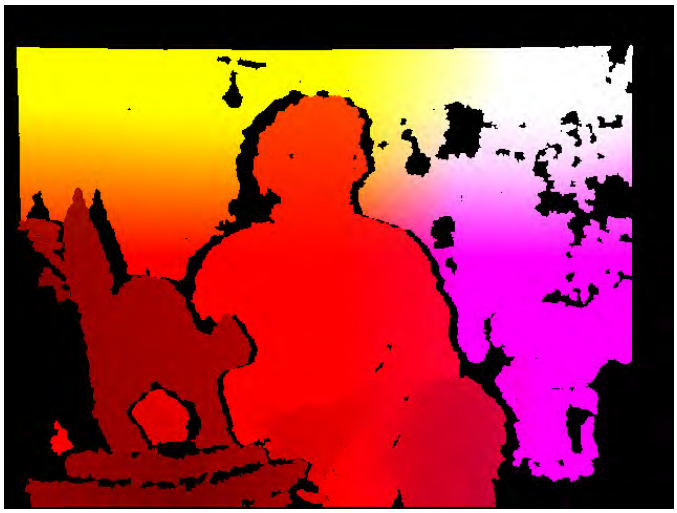


Figure 4.1a: A point cloud map, showing a man sitting behind a sculpture of a cat

Here is a disparity map of the same scene:



Figure 4.1b: A disparity map, showing a man sitting behind a sculpture of a cat

Finally, here is a valid depth mask of the now-familiar cat sculpture and man:



Figure 4.1c: A valid depth mask, associated with the preceding depth images in Figures 4.1a and 4.1b

Next, let's consider how to use some of the channels from a depth camera in an interactive application such as Cameo.

Converting 10-bit images to 8-bit

As we noted in the previous section, some of the channels of a depth camera use a range larger than 8 bits for their data. A large range tends to be useful for computations but inconvenient for display, since most computer monitors are only capable of using an 8-bit range, [0, 255], per channel.

OpenCV's `cv2.imshow` function rescales and truncates the given input data in order to convert the image for display. Specifically, if the input image's data type is unsigned 16-bit or signed 32-bit integers, `cv2.imshow` divides the data by 256 and truncates it to the 8-bit unsigned integer range, [0, 255]. If the input image's data type is 32-bit or 64-bit floating-point numbers, `cv2.imshow` assumes that the data's range is [0.0, 1.0], so it multiplies the data by 255 and truncates it to the 8-bit unsigned integer range, [0, 255]. By re-scaling the data, `cv2.imshow` is relying on its naive assumptions about the original scale. These assumptions will be wrong in some cases. For example, if an image's data type is 16-bit unsigned integers, but the actual data range is 10-bit unsigned integers, [0, 1023], then the image would look very dark if we relied on `cv2.imshow` to convert it.

Consider the following example of an image of an eye, captured with a 10-bit grayscale camera. On the left-hand side, we see the result of a correct conversion from the 10-bit scale to the 8-bit scale. On the right-hand side, we see the result of an incorrect conversion, based on a faulty assumption that the image uses a 16-bit scale:



Figure 4.2: A pair of 8-bit grayscale images, both converted from the same 10-bit grayscale image. The version on the left is correctly converted. The version on the right is incorrectly converted, based on an erroneous assumption of 16-bit input instead of 10-bit

The improperly converted image looks all black because our assumption about the scale was off by a huge amount: 6 bits or a factor of 64. Such a mistake is possible if we rely on `cv2.imshow` to perform conversions to the 8-bit scale automatically for us.

Of course, to avoid such problems, we can do our own image conversions and then pass the resulting 8-bit image to `cv2.imshow`. Let's modify `managers.py` (one of our existing scripts in the Cameo project) in order to provide an option to convert 10-bit images to 8-bit. We will provide a `shouldConvertBitDepth10To8` variable, which the developer can set to `True` or `False`. The following code block (with changes `highlighted in bold`) shows how we initialize this variable:

```
class CaptureManager(object):

    def __init__(self, capture, previewWindowManager = None,
                 shouldMirrorPreview = False,
                 shouldConvertBitDepth10To8 = True):

        self.previewWindowManager = previewWindowManager
        self.shouldMirrorPreview = shouldMirrorPreview
        self.shouldConvertBitDepth10To8 = shouldConvertBitDepth10To8
```

```
        # ... The rest of the method is unchanged ...
```

Next, we will modify the `frame` property's getter to support the conversion. If `shouldConvertBitDepth10To8` is `True`, and the frame's datatype is 16-bit unsigned integers, then we will assume the frame actually has a 10-bit range, which we will convert to 8-bit. As part of the conversion, we will apply a right-bit shift operation, `>> 2`, which truncates the two least significant bits. This is equivalent to integer division by 4. Here is the relevant code:

```
    @property
    def frame(self):
        if self._enteredFrame and self._frame is None:
            _, self._frame = self._capture.retrieve(
                    self._frame, self.channel)
            if self.shouldConvertBitDepth10To8 and \
                    self._frame is not None and \
                    self._frame.dtype == numpy.uint16:
                self._frame = (self._frame >> 2).astype(
                        numpy.uint8)
        return self._frame
```

With these modifications in place, we will be able to more easily manipulate and display frames from some channels, notably `cv2.CAP_OPENNI_IR_IMAGE`. Next, though, let's look at an example of a function that manipulates frames from the `cv2.CAP_OPENNI_DISPARITY_MAP` and `cv2.CAP_OPENNI_VALID_DEPTH_MASK` channels in order to create a mask that isolates one thing, such as the user's face. Afterward, we will consider how to use all these channels together in Cameo.

Creating a mask from a disparity map

Let's assume that a user's face, or some other object of interest, occupies most of the depth camera's field of view. However, the image also contains some other content that is not of interest. By analyzing the disparity map, we can tell that some pixels within the rectangle are outliers—too near or too far to really be a part of the face or another object of interest. We can make a mask to exclude these outliers. However, we should only apply this test where the data is valid, as indicated by the valid depth mask.

Let's write a function to generate a mask whose values are `0` for the rejected regions of the image and `255` for the accepted regions. This function should take a disparity map, valid depth mask, and optionally a rectangle as arguments. If a rectangle is specified, we will make a mask

that is just the size of the specified region. This will be useful to us later in *Chapter 5, Detecting and Recognizing Faces*, where we will work with a face detector that finds bounding rectangles around faces. Let's name our function `createMedianMask` and implement it in `depth.py` as follows:

```
def createMedianMask(disparityMap, validDepthMask, rect = None):
    """Return a mask selecting the median layer, plus shadows."""
    if rect is not None:
        x, y, w, h = rect
        disparityMap = disparityMap[y:y+h, x:x+w]
        validDepthMask = validDepthMask[y:y+h, x:x+w]
    median = numpy.median(disparityMap)
    return numpy.where((validDepthMask == 0) | \
                       (abs(disparityMap - median) < 12),
                       255, 0).astype(numpy.uint8)
```

To identify outliers in the disparity map, we first find the median using `numpy.median`, which takes an array as an argument. If the array is of an odd length, the `median` returns the value that would lie in the middle of the array if the array were sorted. If the array is of an even length, the `median` returns the average of the two values that would be sorted nearest to the middle of the array.

To generate a mask based on per-pixel Boolean operations, we use `numpy.where` with three arguments. In the first argument, `where` takes an array whose elements are evaluated for truth or falsity. An output array of the same dimensions is returned. Wherever an element in the input array is `True`, the `where` function's second argument is assigned to the corresponding element in the output array. Conversely, wherever an element in the input array is `False`, the `where` function's third argument is assigned to the corresponding element in the output array.

Our implementation treats a pixel as an outlier when it has a valid disparity value that deviates from the median disparity value by 12 or more. We have chosen the value of 12 just for experimentation. Feel free to tweak this value later based on the results you encounter when running Cameo with your particular camera setup.

Modifying the application for OpenNI-compatible cameras

Let's open the `Cameo.py` file, which contains the `Cameo` class that we last modified in *Chapter 3, Processing Images with OpenCV*. This class implements an application that works well with regular cameras. We do not necessarily want to replace this class, but rather we want to create

a variant of it that changes the implementations of some methods in order to work with OpenNI-compatible depth cameras instead. For this purpose, we will make a subclass, which inherits some of the `Cameo` behaviors and overrides other behaviors. Let's call it the `CameoDepth` subclass.

Add the following line to `Cameo.py` (after the `Cameo` class and before the `__main__` code block) in order to declare `CameoDepth` as a subclass of `Cameo`:

```
class CameoDepth(Cameo):
```

We will override or reimplement the `__init__` method in `CameoDepth`. Whereas Cameo instantiates our `CaptureManager` class with a device index for a regular camera (specifically, device index `0` for the system's default camera), `CameoDepth` needs to use a device index for a depth camera. The latter can be `cv2.CAP_OPENNI2`, which represents a device index for Microsoft Kinect, or `cv2.CAP_OPENNI2_ASUS`, which represents a device index for Asus Xtion PRO or Occipital Structure. The following code block shows a sample implementation of the `__init__` method of `CameoDepth` (with differences from the `__init__` method of `Cameo` highlighted in bold), but you may want to modify it to uncomment the appropriate device index for your setup:

```
    def __init__(self):
        self._windowManager = WindowManager('Cameo', self.onKeypress)
        #device = cv2.CAP_OPENNI2 # uncomment for Kinect
        device = cv2.CAP_OPENNI2_ASUS # uncomment for Xtion or Structure
        self._captureManager = CaptureManager(
            cv2.VideoCapture(device), self._windowManager, True)
        self._curveFilter = filters.BGRPortraCurveFilter()
```

Similarly, we will override the `run` method in order to use several channels from the depth camera. First, we will try to retrieve a disparity map, then a valid depth mask, and finally, a BGR color image. If no BGR image can be retrieved, this probably means that the depth camera does not have any BGR sensor so in this case, we will proceed to retrieve an infrared grayscale image instead. The following code snippet shows the start of the run method of `CameoDepth` (with differences from the `run` method of `Cameo` highlighted in bold):

```
    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
```

```
            self._captureManager.enterFrame()
            self._captureManager.channel = cv2.CAP_OPENNI_DISPARITY_MAP
            disparityMap = self._captureManager.frame
            self._captureManager.channel = cv2.CAP_OPENNI_VALID_DEPTH_MASK
            validDepthMask = self._captureManager.frame
            self._captureManager.channel = cv2.CAP_OPENNI_BGR_IMAGE
            frame = self._captureManager.frame
            if frame is None:
                # Failed to capture a BGR frame.
                # Try to capture an infrared frame instead.
                self._captureManager.channel = cv2.CAP_OPENNI_IR_IMAGE
                frame = self._captureManager.frame
```

Having captured a disparity map, a valid depth mask, and either a BGR image or an infrared grayscale image, the `run` method continues by calling the `depth.createMedianMask` function that we implemented in the previous section, *Creating a mask from a disparity map*. We pass the disparity map and valid depth mask to the latter function and, in return, we receive a mask that is white in regions whose depth is close to the median depth, and black in other regions. Wherever the mask is black (`mask == 0`), we want to paint the BGR or infrared image black in order to obscure everything except the main object in the image. Finally, for a BGR image, we want to apply the artistic filters that we previously implemented in *Chapter 3*, *Processing Images with OpenCV*. The following code completes the implementation of the `CameoDepthrun` method:

```
            if frame is not None:

                # Make everything except the median layer black.
                mask = depth.createMedianMask(disparityMap, validDepthMask)
                frame[mask == 0] = 0

                if self._captureManager.channel == \
                        cv2.CAP_OPENNI_BGR_IMAGE:
                    # A BGR frame was captured.
                    # Apply filters to it.
                    filters.strokeEdges(frame, frame)
                    self._curveFilter.apply(frame, frame)

            self._captureManager.exitFrame()
            self._windowManager.processEvents()
```

`CameoDepth` does not need any other method implementations of its own; it inherits appropriate implementations from its parent class or `Cameo` superclass.

Now, we simply need to modify the `__main__` section of `Cameo.py` in order to run an instance of the `CameoDepth` class instead of the `Cameo` class. Here is the relevant code:

```
if __name__=="__main__":
    #Cameo().run() # uncomment for ordinary camera
    CameoDepth().run() # uncomment for depth camera
```

Plug in a depth camera and then run the script. Move closer to or farther from the camera until your face is visible, but the background goes black. The following screenshot was taken with `CameoDepth` and an Asus Xtion PRO camera. We can see an infrared image of one of the authors, Joseph Howse, who is brushing his teeth. The code has successfully blacked out the background so the image does not reveal whether he is brushing his teeth in a house, on a train, or in a tent. The mystery continues!!



Figure 4.3a: A screenshot of the `CameoDepth` *application. By creating a mask from a disparity map, our code has successfully blacked out the background*

This is a good opportunity to consider the output of the `createMedianMask` function, which we implemented in the previous section. If we visualize the regions where the mask is `0` as black, and the regions where the mask is `1` as white, then the mask of Joseph Howse brushing his teeth appears as follows:



Figure 4.3b: A visualization of the mask that was used to black out the background in Figure 4.3a

The result is good, but not perfect. For example, on the right-hand side of the image (from the viewer's perspective), the mask incorrectly includes a shadow region behind the hair and incorrectly excludes the shoulder. The latter problem could probably be solved by fine-tuning the criteria that we used with `numpy.where` in the implementation of `createMedianMask`.

If you are fortunate enough to have multiple depth cameras, try all of them to see how they differ in terms of their support for color images and their effectiveness in distinguishing near and far layers. Also, try various objects and lighting conditions to see how they affect (or do not affect) the infrared image. When you are satisfied with the result of your testing, let's move on to other techniques for depth estimation. (However, we will revisit `CameoDepth` in *Chapter 5*, *Detecting and Recognizing Faces*, specifically in the section *Swapping faces in the infrared*.)

Depth estimation with a normal camera

An infrared depth camera is an impressive device, but not every developer or user has one, and it has some limitations. Notably, such a camera does not work well outdoors because the infrared component of sunlight is much brighter than the camera's own infrared light source. Blinded by the sun, the camera cannot see the infrared pattern that it normally uses to estimate depth.

As an alternative, we can use one or more normal cameras, and we can estimate relative distances to objects based on triangulation from different camera perspectives. If we use two cameras simultaneously, this approach is called **stereo vision**. If we use one camera, but we move it over time to obtain different perspectives, this approach is called **structure from motion** (**SfM**). Broadly, techniques for stereo vision are also helpful in SfM, but in SfM we face additional problems if we are dealing with a moving subject. For this chapter's purposes, let's assume that we are dealing with a stationary subject.

As many philosophers would agree, geometry is fundamental to our understanding of the world. More to the point, **epipolar geometry** is the foundation of stereo vision. How does epipolar geometry work? Conceptually, it traces imaginary lines from the camera to each object in the image, then does the same on the second image, and calculates the distance to an object based on the intersection of the lines corresponding to the same object. Here is a representation of this concept:

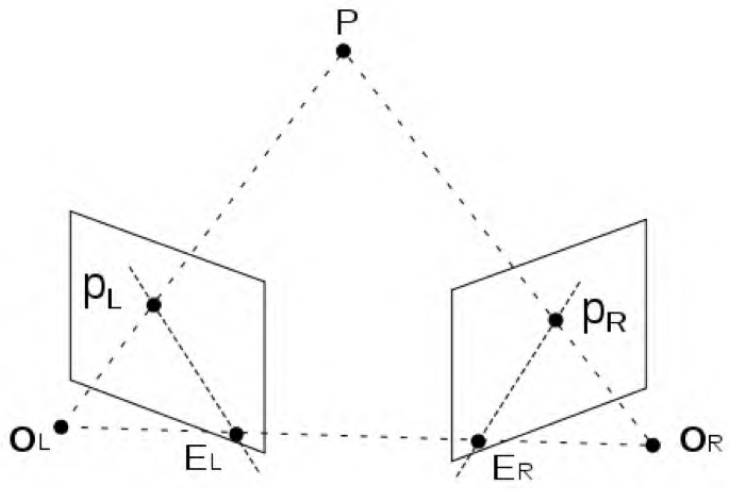


Figure 4.4: A diagram of the epipolar geometry of a stereo pair of images. The point P is projected as pL in the left image and pR in the right image.

Suppose we know the distance and direction between the two cameras – or, more precisely, between the two lenses' optical centers, *OL* and *OR*. Then, we can find the epipoles, *EL* and *ER*, which are the image points where *OR* and *OL* would be projected. (In practice, an epipole's image coordinates may lie beyond the image frame, but this is fine; we only need to calculate the epipole, not see it.) Then, by triangulation, we can find the distance from *OL* to *P* or from *OR* to *P*.

Let's see how OpenCV applies epipolar geometry to calculate a disparity map. This will enable us to segment the image into various layers of foreground and background. As input, we need two images of the same subject taken from different points of view.

Like so many of our scripts, this one begins by importing NumPy and OpenCV:

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

We define initial values for several parameters of a stereo algorithm, as seen in the following code:

```
minDisparity = 16
numDisparities = 192 - minDisparity
blockSize = 5
uniquenessRatio = 1
speckleWindowSize = 3
speckleRange = 3
disp12MaxDiff = 200
P1 = 600
P2 = 2400
```

With these parameters, we create an instance of OpenCV's `cv2.StereoSGBM` class. SGBM is an algorithm used for computing disparity maps. Here is the code that initializes the object:

```
stereo = cv2.StereoSGBM_create(
    minDisparity = minDisparity,
    numDisparities = numDisparities,
    blockSize = blockSize,
    uniquenessRatio = uniquenessRatio,
    speckleRange = speckleRange,
    speckleWindowSize = speckleWindowSize,
    disp12MaxDiff = disp12MaxDiff,
    P1 = P1,
```

```
    P2 = P2
)
```

We also load two images from the file:

```
imgL = cv2.imread('../images/color1_small.jpg')
imgR = cv2.imread('../images/color2_small.jpg')
```

We want to provide several sliders or trackbar widgets to enable a user to interactively adjust the parameters of the algorithm that calculates the disparity map. Whenever a user adjusts any of the sliders, we will update the parameters of the stereo algorithm by setting the properties of the `StereoSGBM` instance, and recalculate the disparity map by calling the `compute` method of the `StereoSGBM` instance. Let's take a look at the implementation of the `update` function, which is the callback function for the sliders:

```
def update(sliderValue=0):

    try:
        blockSize = cv2.getTrackbarPos(
            'blockSize', 'Disparity')
        uniquenessRatio = cv2.getTrackbarPos(
            'uniquenessRatio', 'Disparity')
        speckleWindowSize = cv2.getTrackbarPos(
            'speckleWindowSize', 'Disparity')
        speckleRange = cv2.getTrackbarPos(
            'speckleRange', 'Disparity')
        disp12MaxDiff = cv2.getTrackbarPos(
            'disp12MaxDiff', 'Disparity')
    except cv2.error:
        # One or more of the sliders has not been created yet.
        return

    stereo.setBlockSize(blockSize)
    stereo.setUniquenessRatio(uniquenessRatio)
    stereo.setSpeckleWindowSize(speckleWindowSize)
    stereo.setSpeckleRange(speckleRange)
    stereo.setDisp12MaxDiff(disp12MaxDiff)

    disparity = stereo.compute(
        imgL, imgR).astype(np.float32) / 16.0
```

```
    cv2.imshow('Left', imgL)
    cv2.imshow('Right', imgR)
    cv2.imshow('Disparity',
               (disparity - minDisparity) / numDisparities)
```

Now, let's look at the code that creates a window and sliders:

```
cv2.namedWindow('Disparity')
cv2.createTrackbar('blockSize', 'Disparity',
                   blockSize, 21, update)
cv2.createTrackbar('uniquenessRatio', 'Disparity',
                   uniquenessRatio, 50, update)
cv2.createTrackbar('speckleWindowSize', 'Disparity',
                   speckleWindowSize, 200, update)
cv2.createTrackbar('speckleRange', 'Disparity',
                   speckleRange, 50, update)
cv2.createTrackbar('disp12MaxDiff', 'Disparity',
                   disp12MaxDiff, 250, update)
```

Note that we provide the `update` function as an argument to the `cv2.createTrackbar` function so that `update` is called whenever a slider is adjusted. Next, we call `update` manually to initialize the disparity map:

```
# Initialize the disparity map. Show the disparity map and images.
update()
```

When the user presses any key, we will close the window:

```
# Wait for the user to press any key.
# Meanwhile, update() will be called anytime the user moves a slider.
cv2.waitKey()
```

Let's review the functionality of this example. We take two images of the same subject and calculate a disparity map, showing in brighter tones the points in the map that are closer to the camera (or, in other words, the points where the disparity is greater). Conversely, darker tones represent points that are farther from the camera. Also, black is used for regions where the two images have no overlap (and, thus, the disparity is undefined).

Here is the first image that we have used in this example:



Figure 4.5a: The left image in our `StereoSGBM` *example*

This is the second one:



Figure 4.5b: The right image in our `StereoSGBM` *example*

The user sees the original images, as well as a nice and quite easy-to-interpret disparity map:

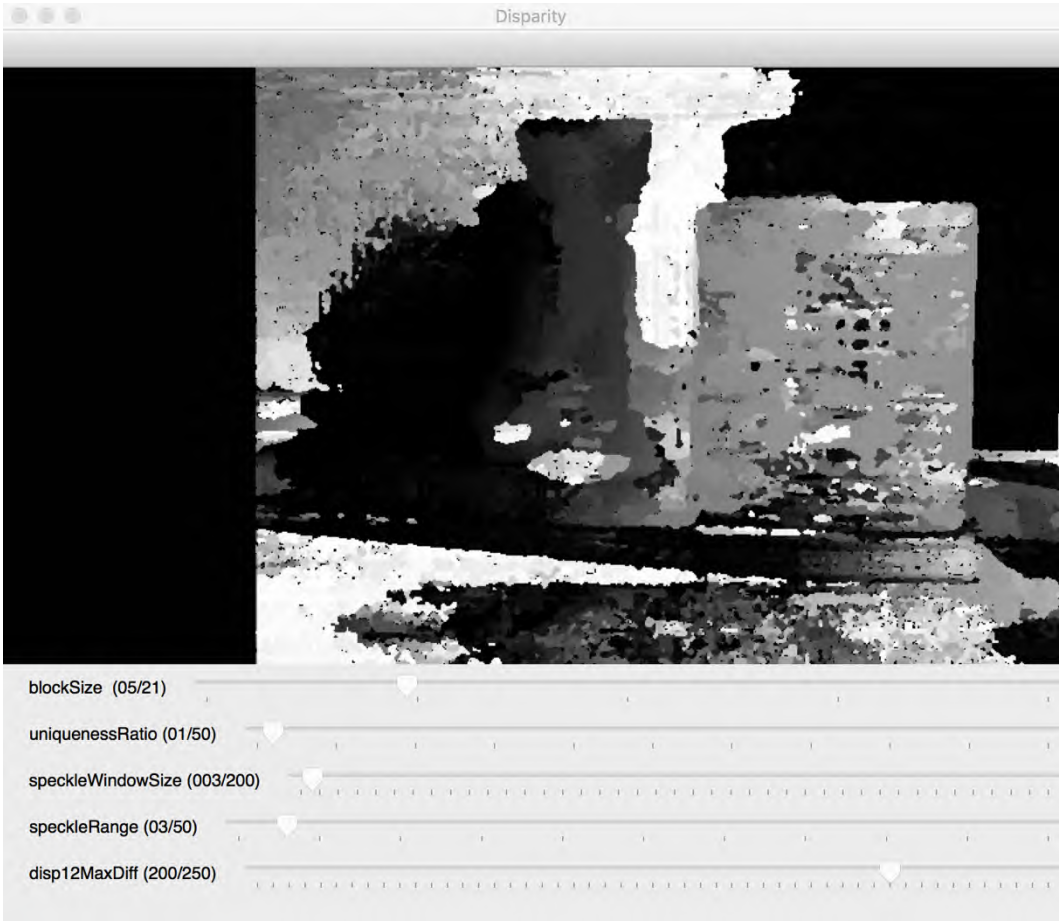


Figure 4.5c: A disparity map generated using `StereoSGBM`*, with a stereo pair of input images as shown in Figures 4.5a and 4.5b*

We have used many, but not all, of the parameters supported by `StereoSGBM`. The OpenCV documentation provides the following descriptions of all the parameters:

Parameter	**Description from OpenCV Documentation**
`minDisparity`	Minimum possible disparity value. Normally, it is zero, but sometimes rectification algorithms can shift images so this parameter needs to be adjusted accordingly.

`numDisparities`	Maximum disparity minus minimum disparity. The value is always greater than zero. In the current implementation, this parameter must be divisible by 16.
`blockSize`	Matched block size. It must be an odd number `>=1`. Normally, it should be somewhere in the 3–11 range.
`P1`	The first parameter controlling the disparity smoothness [see the description of `P2`].
`P2`	The second parameter controlling the disparity smoothness. The larger the values, the smoother the disparity. `P1` is the penalty on the disparity change by plus or minus 1 between neighbor pixels. `P2` is the penalty on the disparity change by more than 1 between neighbor pixels. The algorithm requires `P2 > P1`. See the `stereo_match.cpp` sample where some reasonably good `P1` and `P2` values are shown, such as `8*number_of_image_channels*SADWindowSize*SADWindowSize` and `32*number_of_image_channels*SADWindowSize*SADWindowSize`, respectively.
`disp12MaxDiff`	Maximum allowed difference (in integer pixel units) in the left-right disparity check. Set it to a non-positive value to disable the check.
`preFilterCap`	Truncation value for the prefiltered image pixels. The algorithm first computes the `x`-derivative at each pixel and clips its value by the `[-preFilterCap, preFilterCap]` interval. The resulting values are passed to the Birchfield-Tomasi pixel cost function.
`uniquenessRatio`	Margin in percentage by which the best (minimum) computed cost function value should *win* the second best value to consider the found match correct. Normally, a value within the 5–15 range is good enough.
`speckleWindowSize`	Maximum size of smooth disparity regions to consider their noise speckles and invalidate. Set it to 0 to disable speckle filtering. Otherwise, set it somewhere in the 50–200 range.

`speckleRange`	Maximum disparity variation within each connected component. If you do speckle filtering, set the parameter to a positive value; it will be implicitly multiplied by 16. Normally, 1 or 2 is good enough.
`mode`	Set it to `StereoSGBM::MODE_HH` to run the full-scale, two-pass dynamic programming algorithm. It will consume `O(W*H*numDisparities)` bytes, which is large for 640x480 stereo and huge for HD-size pictures. By default, it is set to `false`.

Table 4.1: Descriptions of all parameters supported by `StereoSGBM`

With the preceding script, you will be able to load images of your choice and play around with parameters until you are happy with the disparity map generated by `StereoSGBM`.

Stereo imaging with an OAK-D camera

Previously, when we discussed OpenNI-compatible depth cameras, we highlighted the role that NIR illumination and NIR images can play in depth estimation. Indeed, when OpenNI was first developed, NIR depth cameras were considered state-of-the-art. However, they have some shortcomings, such as their aforementioned poor performance in sunlit scenes, and the possibility that invisible NIR lights may cause eyestrain if unsuspecting users stare into them.

Arguably, the OAK-D cameras represent a new state of the art. Most OAK-D models eschew the NIR strategy, instead relying on high-speed (120 FPS) stereo imaging with a highly optimized implementation of SGBM. Indeed, every OAK-D device is capable of running a wide variety of computer vision and machine learning algorithms directly on the camera. Onboard, it has a specialized **Vision Processing Unit** (**VPU**) called the Intel Movidius Myriad X.

Note

As well as using high-speed stereo imaging, the top-of-the-line OAK-D Pro camera also includes an NIR laser dot projector. Thus, it is like a faster, smarter version of the classic OpenNI-compatible NIR depth cameras. However, in our sample code, we will focus purely on the stereo imaging features that are supported by the entire OAK-D product line. Other leading brands of depth cameras, such as Orbbec (`https://www.orbbec.com/`) and RealSense (`https://www.realsenseai.com/`), also offer a full range of stereo imaging and NIR projection capabilities, with advanced onboard processing in some models. Orbbec and RealSense have their own C++ and Python interfaces.

> Using the `depthai` Python module, let's build a basic demo of depth imaging with an OAK-D camera. All OAK-D models have a stereo pair of monochrome cameras, plus an RGB color camera in the middle. We will capture a stream of disparity maps, calculated by the OAK-D device itself based on the stereo pair. At the same time, we will capture a stream of RGB images from the color camera. Using OpenCV, we will display the disparity maps in one window and the RGB frames in another.

To begin our Python script, let's import the relevant modules:

```
import cv2
import depthai as dai
import numpy as np
```

Note that we have assigned an alias, `dai`, to the `depthai` module. The official `depthai` sample code uses the alias `dai`, so we are doing the same.

The concept of an imaging **pipeline** is fundamental to the design of `depthai`. Onboard the OAK-D device, streams of images may originate from multiple sources, such as multiple cameras. These streams may flow through processes, such as the process of computing a disparity map. Ultimately, the streams end in outputs, such as the disparity maps that are transferred from the OAK-D device to the host computer where our Python script is running. Each source, process, or output is called a **node** in the pipeline. We can define our pipeline's **topology** (its set of nodes and connections) via a `depthai.Pipeline` object, which we can instantiate as follows:

```
# Create the pipeline.
pipeline = dai.Pipeline()
```

As part of this pipeline's topology, let's define four nodes – two inputs, a process, and an output – pertaining to stereo imaging and the creation of disparity maps:

```
# Define the nodes in the mono -> stereo -> output chain.
monoLeft = pipeline.create(dai.node.MonoCamera)
monoRight = pipeline.create(dai.node.MonoCamera)
depth = pipeline.create(dai.node.StereoDepth)
disparityOut = pipeline.create(dai.node.XLinkOut)
disparityOut.setStreamName("disparity")
```

Note that we have given the disparity output node a stream name, `"disparity"`. Later, we will use this stream name as a key to fetch the output queue.

We will configure our two monochrome input nodes to use the left and right cameras, respectively, and to use a resolution of 400p (that is, 640x400):

```
# Configure the mono cameras.
monoResolution = dai.MonoCameraProperties.SensorResolution.THE_400_P
monoLeft.setResolution(monoResolution)
monoLeft.setBoardSocket(dai.CameraBoardSocket.CAM_B)
monoRight.setResolution(monoResolution)
monoRight.setBoardSocket(dai.CameraBoardSocket.CAM_C)
```

Next, we will configure the depth node to output a disparity map that is optimized for low noise and high accuracy at far depths:

```
# Configure the depth node for disparity output.

depth.setDefaultProfilePreset(
    dai.node.StereoDepth.PresetMode.DEFAULT)

# Median filter options:
#     MEDIAN_OFF
#     KERNEL_3x3
#     KERNEL_5x5
#     KERNEL_7x7 (default)
depth.initialConfig.setMedianFilter(dai.MedianFilter.KERNEL_7x7)

# True -> better occlusion handling
depth.setLeftRightCheck(True)

# True -> nearer minimum depth (max disparity = 190)
# False -> farther minimum depth (max disparity = 95)
depth.setExtendedDisparity(False)

# True -> better accuracy at far depth, but
#         incompatible with setExtendedDisparity(True)
depth.setSubpixel(False)
```

We will link the nodes such that the two monochrome inputs provide a stream of stereo images to the depth node, and the depth node provides a stream of disparity maps to our disparity output node:

```
# Link the nodes in the mono -> stereo -> output chain.
monoLeft.out.link(depth.left)
monoRight.out.link(depth.right)
depth.disparity.link(disparityOut.input)
```

Similarly, for RGB input and output, we can define, configure, and link the relevant nodes as follows:

```
# Define the nodes in the RGB chain.
rgb = pipeline.create(dai.node.ColorCamera)
rgbOut = pipeline.create(dai.node.XLinkOut)
rgbOut.setStreamName("rgb")

# Configure the RGB camera.
rgb.setBoardSocket(dai.CameraBoardSocket.CAM_A)
rgb.setResolution(
    dai.ColorCameraProperties.SensorResolution.THE_1080_P)
rgb.setVideoSize(1920, 1080)

# Link the nodes in the RGB -> output chain.
rgb.video.link(rgbOut.input)
```

Thus far, we defined the topology of an imaging pipeline, with two output streams named `"disparity"` and `"rgb"`. Now, with this pipeline definition, we can open a connection to the OAK-D device by instantiating the `depthai.Device` class, as follows:

```
# Connect to the device and start the pipeline.
with dai.Device(pipeline) as device:
```

Using this connection to the device, we can get our two output queues by name:

```
    # Get the output queues.
    disparityQ = device.getOutputQueue(
        name="disparity", maxSize=4, blocking=False)
    rgbQ = device.getOutputQueue(
        name="rgb", maxSize=4, blocking=False)
```

Next, let's implement our application's main loop. Every millisecond, we will poll for keyboard input and also poll the disparity queue and RGB queue for frames:

```
    while cv2.waitKey(1) == -1:

        # Grab the next disparity frame, if any is ready.
        disparityGrab = disparityQ.tryGet()

        # Grab the next RGB frame, if any is ready.
        rgbGrab = rgbQ.tryGet()
```

If a queue is empty, its `tryGet` method returns `None`. Note that in a typical OAK-D configuration, the stereo imaging runs at a higher frame rate than the RGB imaging, so it will often happen that the RGB queue is empty even when the disparity queue has a frame.

If a disparity frame is ready, let's get its data, normalize it to an 8-bit grayscale range, and display it:

```
        if disparityGrab is not None:
            disparityFrame = disparityGrab.getFrame()
            # Normalize the disparity map for better visualization.
            disparityFrame = (disparityFrame * \ (
                255.0 / depth.initialConfig.getMaxDisparity()
            )).astype(np.uint8)
            cv2.imshow("disparity", disparityFrame)
```

Similarly, if an RGB frame is ready, let's get its data, convert it to OpenCV's 8-bit-per-channel BGR format, and display it:

```
        if rgbGrab is not None:
            # Get the frame and convert it from its native NV12 encoding
            # to OpenCV's BGR format.
            bgrFrame = rgbGrab.getCvFrame()
            cv2.imshow("RGB", bgrFrame)
```

That's all! If we connect any OAK-D camera and run the script, we should see two windows: one with a continuous stream of disparity frames, and another with the corresponding RGB frames. Here is a screenshot of a disparity frame, captured with an OAK-D Lite device:



Figure 4.6a: A disparity frame captured with an OAK-D Lite device

If you guessed that the preceding disparity map shows two creatures lounging with a book, you are correct! Here is the corresponding RGB frame to prove it:



Figure 4.6b: An RGB frame captured with an OAK-D Lite device. This RGB frame corresponds to the disparity map seen above in Figure 4.6a. The pillow creatures in the image are called Fluffitarians; they are works of art by Janet Howse. The book in the image is a novel, The Girl in the Water, by Joseph Howse, with cover art by Janet Howse.

Of course, in a practical application, you might want to combine the data from the disparity map and the RGB image in some way. Remember to convert them to a common resolution first. For a complete example of this, see the official `depthai` tutorial, "RGB Depth alignment," at `https://docs.luxonis.com/software/depthai/examples/rgb_depth_aligned/`.

Depth estimation is only a small part of what OAK-D devices can do. Indeed, the most innovative thing about them is their multi-purpose VPU, which can efficiently solve many classification problems even if you do not use the camera's stereo capabilities at all. For a wide range of tutorials and code samples, see the official `depthai` documentation online at `https://docs.luxonis.com/software/depthai/examples/`.

Tip

You may find that in some cases, the official `depthai` code samples depend on a particular version of `depthai`. For example, when you run a script, it might output an error such as the following:

Version mismatch between installed depthai lib and the required one by the script.

```
    Required:  2.17.3.0
    Installed: 2.32.0.0
```

To work around this problem, you can downgrade your `depthai` installation to the required version using a `pip` command such as the following:

```
$ pip install depthai==2.17.3.0
```

(For some Python environments, you may need to substitute `pip3` for `pip`.)

Now, let's finish this chapter by exploring a segmentation algorithm that does not require any special hardware or any triangulation of camera perspectives.

Foreground detection with the GrabCut algorithm

Calculating a disparity map is a useful way to segment the foreground and background of an image, but `StereoSGBM` is not the only algorithm that can accomplish this and, in fact, `StereoSGBM` is more about gathering three-dimensional information from two-dimensional pictures than anything else. **GrabCut**, however, is a perfect tool for foreground/background segmentation. The GrabCut algorithm consists of the following steps:

1. A rectangle including the subject(s) of the picture is defined.

2. The area lying outside the rectangle is automatically defined as a background.
3. The data contained in the background is used as a reference to distinguish background areas from foreground areas within the user-defined rectangle.
4. A **Gaussian Mixture Model** (**GMM**) models the foreground and background, and labels undefined pixels as probable background and probable foreground.
5. Each pixel in the image is virtually connected to the surrounding pixels through virtual edges, and each edge is assigned a probability of being foreground or background, based on how similar it is in color to the pixels surrounding it.
6. Each pixel (or node as it is conceptualized in the algorithm) is connected to either the foreground node or the background node. You can visualize this as follows:

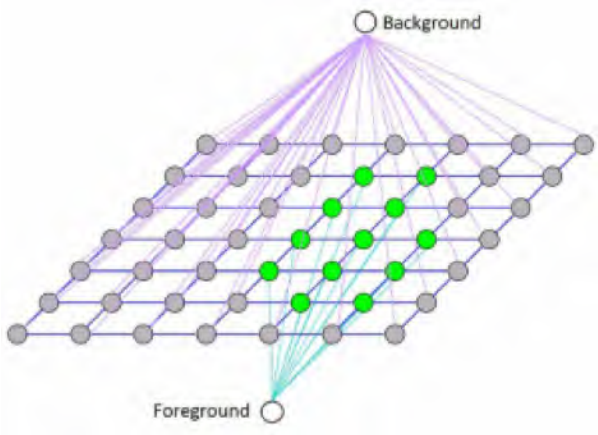


Figure 4.7a: A visualization of nodes in the GrabCut algorithm. Each pixel is connected to either the foreground or the background

7. After the nodes have been connected to either terminal (the background or foreground, also called the source or sink, respectively), the edges between nodes belonging to different terminals are cut (hence the name, GrabCut). Thus, the image is segmented into two parts. The following figure adequately represents the algorithm:

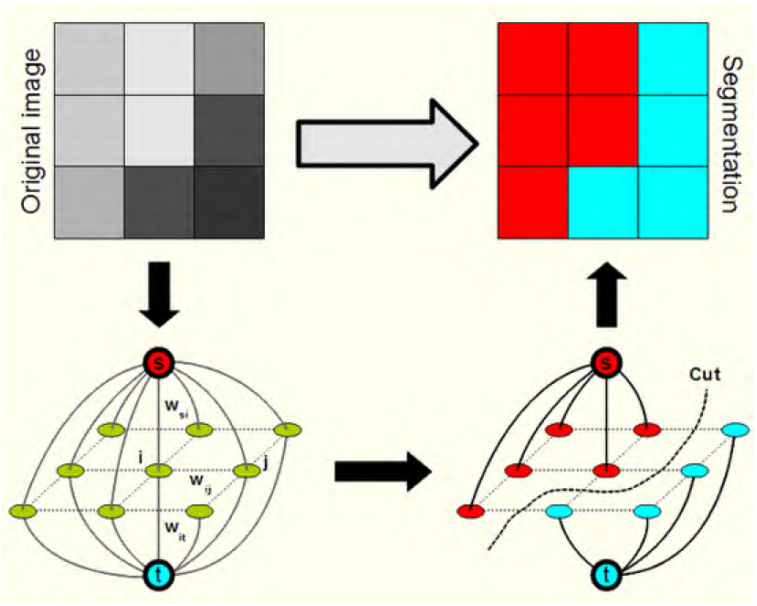


Figure 4.7b: A visualization of a cut in the GrabCut algorithm. Along the cutline, connections between nodes are severed, leaving two segments: the foreground and background

Let's look at an example. We start with the image of a beautiful statue of an angel:



Figure 4.8a: A photo showing a statue of an angel against a blue sky. We will use this as input for our `GrabCut` *example*

We want to grab our angel and discard the background. To do this, we will create a relatively short script that will segment the image using GrabCut, and then display the resulting foreground image side by side with the original. We will use `matplotlib`, a popular Python library, which makes displaying charts and images a trivial task.

The code is actually quite straightforward. First, we load the image we want to process, and then we create a mask populated with zeros with the same shape as the image we have loaded:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

original = cv2.imread('../images/statue_small.jpg')
img = original.copy()
mask = np.zeros(img.shape[:2], np.uint8)
```

We then create zero-filled background and foreground models:

```
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
```

We could have populated these models with data, but instead we are going to initialize the GrabCut algorithm with a rectangle that approximates the bounds of the foreground subject. Thus, background and foreground models are going to be determined based on the areas left out of the initial rectangle. This rectangle is defined in the next line:

```
rect = (100, 1, 421, 378)
```

Now to the interesting part! We run the GrabCut algorithm. As arguments, we specify the empty models, the mask, and the rectangle that we want to use to initialize the operation:

```
cv2.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)
```

Notice the integer argument 5. This is the number of iterations the algorithm is going to run on the image. You can increase it, but at some point, pixel classifications will converge so, effectively, you might just be adding iterations without any further improvements to the result.

After this, our mask will have changed to contain values between 0 and 3, inclusive. These values have the following meanings:

- `0` (also defined as `cv2.GC_BGD`) is an obvious background pixel.
- `1` (also defined as `cv2.GC_FGD`) is an obvious foreground pixel.
- `2` (also defined as `cv2.GC_PR_BGD`) is a probable background pixel.
- `3` (also defined as `cv2.GC_PR_FGD`) is a probable foreground pixel.

To visualize the result of the GrabCut, we want to paint the background black and leave the foreground unchanged. We can make another mask to help us do this. The values `0` and `2` (obvious and probable background) will be converted into 0s, and the values `1` and `3` (obvious and probable foreground) into 1s. The result will be stored in `mask2`. We will multiply the original image by `mask2` in order to make the background black (by multiplying by `0`) while leaving the foreground unchanged (by multiplying by `1`). Here is the relevant code:

```
mask2 = np.where((mask==2) | (mask==0), 0, 1).astype('uint8')
img = img*mask2[:,:,np.newaxis]
```

The final part of our script displays the images side by side:

```
plt.subplot(121)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title("grabcut")
plt.xticks([])
plt.yticks([])

plt.subplot(122)
plt.imshow(cv2.cvtColor(original, cv2.COLOR_BGR2RGB))
plt.title("original")
plt.xticks([])
plt.yticks([])

plt.show()
```

Here is the result:



Figure 4.8b: The output of our `GrabCut` *example. The left-hand image shows the background segment in black and the foreground segment in its original colors*

This is quite a satisfactory result. You will notice that a triangle of blue sky remains under the angel's arm; this area was incorrectly marked as foreground. It is possible to refine the GrabCut result by manually selecting more background regions and applying more iterations. This technique is quite well illustrated in the `grabcut.py` file in the `samples/python` folder of your OpenCV installation. (For details of where to find the OpenCV samples, refer back to *Chapter 1*, *Setting Up OpenCV*, specifically the section *Running the official sample code*.)

Image segmentation with the Watershed algorithm

Now, let's take a quick look at the Watershed algorithm. The algorithm is called Watershed because its conceptualization involves water. Imagine areas with low density (little to no change) in an image as valleys, and areas with high density (lots of change) as peaks. Start filling the valleys with water to the point where water from two different valleys is about to merge. To prevent the merging of water from different valleys, you build a barrier to keep them separated. The resulting barrier is the image segmentation.

Effectively, the Watershed algorithm uses a locally adaptive threshold to find each barrier. Thus, it can cope well with segmentation problems where some pairs of objects have a high-contrast boundary and others have a relatively low-contrast boundary. (For example, think of a basket of golf balls where most balls are white but a few balls are colored.) On the other hand, Watershed may struggle with segmentation problems where each object, internally, has a lot of contrasting textural details.

As an example, let's segment an image of a playing card. We want to separate the pips (the large, countable symbols) from the background:

1. Once more, we begin our script by importing `numpy`, `cv2`, and `matplotlib`. Then, we load our image of a playing card from the file:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('../images/5_of_diamonds.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

2. After converting the image from color to grayscale, we run a threshold on it. This operation helps by dividing the image into two regions, blacks and whites:

```
ret, thresh = cv2.threshold(
    gray, 0, 255,
    cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU
)
```

3. Next, we remove noise from the thresholded image by applying a morphological transformation to it. **Morphology** consists of **dilating** (expanding) or **eroding** (contracting) the white regions of the image in some series of steps. We will apply the morphological open operation, which consists of an erosion step followed by a dilation step. The open operation makes big white regions swallow up little black regions (noise), while leaving big black regions (real objects) relatively unchanged. The `cv2.morphologyEx` function, with the `cv2.MORPH_OPEN` argument, allows us to perform this operation:

```
# Remove noise.
kernel = np.ones((3,3), np.uint8)
opening = cv2.morphologyEx(
    thresh, cv2.MORPH_OPEN, kernel, iterations=2)
```

4. By dilating the result of the open transformation, we can obtain regions of the image that are most certainly background:

```
# Find the sure background region.
sure_bg = cv2.dilate(opening, kernel, iterations=3)
```

5. Conversely, we can obtain sure foreground regions by applying `distanceTransform`. In practical terms, we can be most confident that a point is really part of the foreground if it is far away from the nearest foreground-background edge. Once we have obtained the `distanceTransform` representation of the image, we apply a threshold to select regions that are most surely part of the foreground:

```
# Find the sure foreground region.
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(
    dist_transform, 0.7 * dist_transform.max(), 255, 0)
sure_fg = sure_fg.astype(np.uint8)
```

   At this stage, we have some sure foreground and background regions.

6. Now, what about the regions in between? We can find these unsure or unknown regions by subtracting the sure foreground from the background:

```
# Find the unknown region.
unknown = cv2.subtract(sure_bg, sure_fg)
```

7. Now that we have these regions, we can build our famous *barriers* to stop the water from merging. This is done with the `connectedComponents` function. We took a glimpse at graph theory when we analyzed the GrabCut algorithm and conceptualized an image as a set of nodes that are connected by edges. Given the sure foreground areas, some of these nodes will be connected together, but some will not. The disconnected nodes belong to different water valleys, and there should be a barrier between them:

```
# Label the foreground objects.
ret, markers = cv2.connectedComponents(sure_fg)
```

8. Next, we add 1 to the labels for all regions because we only want unknowns to stay at `0`:

```
# Add one to all labels so that sure background is not 0, but 1.
markers += 1

# Label the unknown region as 0.
markers[unknown==255] = 0
```

9. Finally, we open the gates! Let the water flow! The `cv2.watershed` function assigns the label `-1` to pixels that are edges between components. We color these edges blue in the original image:

```
markers = cv2.watershed(img, markers)
img[markers==-1] = [255, 0, 0]
```

10. Let's use `matplotlib` to show the result:

```
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.show()
```

The plot should look like the following image:



Figure 4.9: A segmentation result, using Watershed with an image of a playing card (the five of diamonds) as input. The blue outlines around the pips (the five large diamonds) show that they are correctly segmented

This type of segmentation could serve as part of a system for recognizing playing cards. Similarly, the Watershed algorithm can help us segment and count any kind of object on a plain background. For example, Watershed segmentation of coins on a sheet of paper is demonstrated in an official OpenCV tutorial at `https://docs.opencv.org/5.0/py_tutorials/py_imgproc/py_watershed/py_watershed.html`.

Image segmentation with HFS

Finally, let's consider a more recent segmentation algorithm called **Hierarchical Feature Selection** (**HFS**). As the name suggests, this algorithm considers multiple types of features in

order to relate image regions to each other hierarchically. Specifically, the hierarchy has three layers, corresponding to three stages of the algorithm:

1. First, the algorithm groups the pixels into irregularly shaped neighborhoods of similar color – rather like tiles in a mosaic. The minimum number of pixels per tile is defined by a parameter.
2. Next, the algorithm takes each pair of adjacent mosaic tiles and computes a feature vector to describe the differences between these two tiles. Effectively, these feature vectors are edge descriptors. Some of the weaker edges are discarded, such that neighborhoods of similar tiles are merged to make bigger tiles. The maximum number of pixels per bigger tile is, again, defined by a parameter.
3. Last, to obtain the finished segmentation, the algorithm merges some of the bigger tiles based on a process that is similar to step 2 above.

> **Note**
>
> To learn more about the nuances of HFS, see the algorithm's original definition in *HFS: Hierarchical Feature Selection for Efficient Image Segmentation*, by Ming-Ming Cheng, Yun Liu, et al. This research paper can be downloaded from `http://mftp.mmcheng.net/Papers/HFS.pdf`.

Compared to the Watershed algorithm, HFS is less reliant on ridge-like features and better able to cope with textured or multicolored objects. The hierarchical approach in HFS, with maximum sizes at steps 1 and 2, helps to ensure that this algorithm considers a variety of features before forming any large segments.

OpenCV provides an easy-to-use `cv2.hfs.HfsSegment` class. Let's see what we can do with it in just a few lines of code:

```
import cv2

img = cv2.imread('../images/statue_small.jpg')
hfsSegmenter = cv2.hfs.HfsSegment_create(
    height=img.shape[0], width=img.shape[1],
    minRegionSizeI=50, minRegionSizeII=100)
segmented_img = hfsSegmenter.performSegmentCpu(img)
cv2.imshow("HFS segmentation", segmented_img)
cv2.waitKey()
```

Note that we have defined the maximum region size as 50 pixels at stage 1 (`minRegionSizeI=50`) and then 100 pixels at stage 2 (`minRegionSizeII=100`). Also note that we are running the HFS algorithm on the CPU, as we have invoked the `performSegmentCpu` method. If we wanted to run the algorithm on the GPU instead, we could call `performSegmentGpu`, and in that case, OpenCV would transfer the image data to graphics memory, process it on the GPU, and then transfer it back to a NumPy array in the main memory.

Here is a screenshot of our script's output, which is the result of performing HFS segmentation on our famous stone angel:



Figure 4.10: A segmentation result, using HFS with an image of a stone angel as input

HFS has done a rather good job of reducing all the details and shadows to just a few segments. For this result, we did not even have to pre-process or post-process the image with any thresholding or morphology step. Now, try adjusting the parameters and try some other input images to see what else you can achieve. You can find more documentation on the `HfsSegment` class and its parameters at `https://docs.opencv.org/5.0/extra_modules/hfs.html`.

Summary

In this chapter, we learned how to analyze simple spatial relationships within images so that we can differentiate between multiple objects, or between a foreground and a background. Our techniques included the extraction of three-dimensional information from a two-dimensional input (a video frame or an image). First, we examined OpenNI-compatible depth cameras, then epipolar geometry, stereo images, and OAK-D devices, so we are now able to calculate

disparity maps. Finally, we looked at image segmentation with three popular methods: GrabCut, Watershed, and HFS.

As we progress through this book, we will continue to extract increasingly complex information from images. Next, we are ready to explore OpenCV's functionality for the detection and recognition of faces and other objects.

Part 2

Finding and Tracking People, Patterns, and Objects

What is it? Where did it go? These burning questions have given birth to many an algorithm in our beloved field of computer vision. *Part 2* explores techniques for detecting faces, recognizing specific people, searching for a particular pattern such as a logo, detecting any type of object such as a car, and tracking a car or other moving target across multiple frames of a video. These topics also serve as an introduction to machine learning.

This part contains the following chapters:

- *Chapter 5, Detecting and Recognizing Faces*
- *Chapter 6, Retrieving Images and Searching Using Image Descriptors*
- *Chapter 7, Building Custom Object Detectors*
- *Chapter 8, Tracking Objects*

5

Detecting and Recognizing Faces

Computer vision makes many *futuristic*-sounding tasks a reality. Two such tasks are face detection (locating faces in an image) and face recognition (identifying a face as belonging to a specific person). OpenCV implements several algorithms for face detection and recognition. These have applications in all sorts of real-world contexts, from security to entertainment.

This chapter introduces some of OpenCV's face detection and recognition functionality, along with data files that define particular types of trackable objects. Specifically, in this chapter, we'll look at Haar cascade classifiers, which analyze the contrast between adjacent image regions to determine whether or not a given image or sub-image matches a known type. We consider how to combine multiple Haar cascade classifiers in a hierarchy so that one classifier identifies a parent region (for our purposes, a face) and other classifiers identify child regions (such as eyes).

We also take a detour into the humble but important subject of rectangles. By drawing, copying, and resizing rectangular image regions, we can perform simple manipulations on image regions that we are tracking.

We will cover the following topics:

- Understanding Haar cascades
- Finding the pre-trained Haar cascades that come with OpenCV, including several face detectors
- Using Haar cascades to detect faces in still images and videos
- Gathering images to train and test a face recognizer
- Using several different face recognition algorithms: **Eigenfaces**, **Fisherfaces**, and **Local Binary Pattern Histograms** (**LBPHs**)
- Copying rectangular regions from one image to another, with or without a mask

- Using a depth camera to distinguish between a face and the background based on depth
- Swapping two people's faces in an interactive application

Note

While this chapter focuses on classic approaches to face detection and recognition, we will go on to explore advanced new models by using OpenCV with a variety of other libraries in *Chapter 11, Introduction to Neural Networks with OpenCV*.

By the end of this chapter, we will have integrated face tracking and rectangle manipulations into Cameo, the interactive application that we have developed in previous chapters. Finally, we will have some face-to-face interaction!

Technical requirements

This chapter uses Python, OpenCV, and NumPy. As part of OpenCV, it uses the optional `opencv_contrib` modules, which include functionality for face recognition. Some parts of this chapter use OpenCV's optional support for OpenNI 2 to capture images from depth cameras. Please refer back to *Chapter 1, Setting Up OpenCV*, for installation instructions.

The complete code for this chapter can be found in this book's GitHub repository in the `chapter05` folder. Sample images are in the repository's `images` folder.

A subset of the chapter's sample code can be edited and run interactively in Google Colab.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Conceptualizing Haar cascades

When we talk about classifying objects and tracking their location, what exactly are we hoping to pinpoint? What constitutes a recognizable part of an object?

Photographic images, even from a webcam, may contain a lot of detail for our (human) viewing pleasure. However, image detail tends to be unstable with respect to variations in lighting, viewing angle, viewing distance, camera shake, and digital noise. Moreover, even real differences in physical detail might not interest us for classification. Joseph Howse, one of this book's authors, was taught in school that no two snowflakes look alike under a microscope. Fortunately, as a Canadian child, he had already learned how to recognize snowflakes without a microscope, as the similarities are more obvious in bulk.

Hence, having some means of abstracting image detail is useful in producing stable classification and tracking results. The abstractions are called **features**, which are said to be **extracted** from the image data. There should be far fewer features than pixels, though any pixel might influence multiple features. A set of features is represented as a **vector** (conceptually, a set of coordinates in a multidimensional space), and the level of similarity between two images can be evaluated based on some measure of the distance between the images' corresponding feature vectors.

> **Note**
>
> Later, in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*, we will explore several kinds of features, as well as advanced ways of describing and matching sets of features.

Haar-like features are one type of feature that is often applied to real-time face detection. They were first used for this purpose in the paper *Robust Real-Time Face Detection,* by *Paul Viola and Michael Jones (International Journal of Computer Vision 57(2), 137–154, Kluwer Academic Publishers, 2001)*. An electronic version of this paper is available at `http://comp3204.ecs.soton.ac.uk/cw/viola04ijcv.pdf`.

Each Haar-like feature describes the pattern of contrast among adjacent image regions. For example, edges, vertices, and thin lines each generate a kind of feature. Some features are distinctive in the sense that they typically occur in a certain class of object (such as a face) but not in other objects. These distinctive features can be organized into a hierarchy, called a **cascade**, in which the highest layers contain features of greatest distinctiveness, enabling a classifier to quickly reject subjects that lack these features. If a subject is a good match for the higher-layer features, then the classifier considers the lower-layer features too in order to weed out more false positives.

For any given subject, the features may vary depending on the scale of the image and the size of the neighborhood (the region of nearby pixels) in which contrast is being evaluated. The neighborhood's size is called the **window size**. To make a Haar cascade classifier **scale-invariant**, or, in other words, robust to changes in scale, the window size is kept constant, but images are rescaled a number of times; hence, at some level of rescaling, the size of an object (such as a face) may match the window size. Together, the original image and the rescaled images are called an **image pyramid**, and each successive level in this pyramid is a smaller rescaled image. OpenCV provides a scale-invariant classifier that can load a Haar cascade from an XML file in a particular format. Internally, this classifier converts any given image into an image pyramid.

Haar cascades, as implemented in OpenCV, are not robust to changes in rotation or perspective. For example, an upside-down face is not considered similar to an upright face, and a face viewed in profile is not considered similar to a face viewed from the front. A more complex and resource-intensive implementation could improve a Haar cascade's robustness to rotation by considering multiple transformations of images as well as multiple window sizes. However, we will confine ourselves to the implementation in OpenCV.

Getting Haar cascade data

This book's repository contains a folder called `data/haarcascades`. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

In that folder, you will find XML files that can be loaded by an OpenCV class called `cv2.CascadeClassifier`. An instance of this class interprets a given XML file as a Haar cascade, which provides a detection model for a type of object such as a face. `cv2.CascadeClassifier` can detect this type of object in any image. As usual, we could obtain a still image from a file, or we could obtain a series of frames from a video file or a video camera.

> **Note**
>
> OpenCV 4 (and earlier versions) included these XML cascade files with any standard installation. Since OpenCV 5, they are no longer included, yet they still work with the current implementation of `cv2.CascadeClassifier`.

From the `data/haarcascades` folder, we will use the following cascade files:

- `haarcascade_frontalface_default.xml`
- `haarcascade_eye.xml`

As their names suggest, these cascades are for detecting faces and eyes. They require a frontal, upright view of the subject. We will use them later when building a face detector.

> **Note**
>
> If you are curious about how these cascade files are generated, you can find more information in Joseph Howse's book, *OpenCV 4 for Secret Agents* (Packt Publishing, 2019), specifically in *Chapter 3, Training a Smart Alarm to Recognize the Villain and His Cat*. With a lot of patience and a reasonably powerful computer, you can make your own cascades and train them for various types of objects.

Using OpenCV to perform face detection

With `cv2.CascadeClassifier`, it makes little difference whether we perform face detection on a still image or a video feed. The latter is just a sequential version of the former: face detection on a video is simply face detection applied to each frame. Naturally, with more advanced techniques, it would be possible to track a detected face continuously across multiple frames and determine that the face is the same one in each frame. However, it is good to know that a basic sequential approach also works.

Let's go ahead and detect some faces.

Performing face detection on a still image

The first and most basic way to perform face detection is to load an image and detect faces in it. To make the result visually meaningful, we will draw rectangles around faces in the original image. Remembering that the face detector is designed for upright, frontal faces, we will use an image of a row of people, specifically woodcutters, standing shoulder to shoulder and facing the photographer or viewer.

Let's go ahead and create the following basic script to perform face detection:

```
import cv2
face_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_frontalface_default.xml')
img = cv2.imread('../images/woodcutters.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.08, 5)
for (x, y, w, h) in faces:
    img = cv2.rectangle(img, (x, y), (x+w, y+h), (255, 255, 0), 2)

cv2.imshow('Woodcutters Detected!', img)
cv2.imwrite('./woodcutters_detected.png', img)
cv2.waitKey(0)
```

Let's walk through the preceding code in small steps. First, we use the obligatory `cv2` import that you will find in every script in this book. Then, we declare a `face_cascade` variable, which is a `CascadeClassifier` object that loads a cascade for face detection:

```
face_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_frontalface_default.xml')
```

We then load our image file with `cv2.imread` and convert it into grayscale because `CascadeClassifier`, like many of OpenCV's classifiers, expects grayscale images. (If we try to use a color image, `CascadeClassifier` will internally convert it to grayscale anyway.) The next step, `face_cascade.detectMultiScale`, is where we perform the actual face detection:

```
img = cv2.imread('../images/woodcutters.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.08, 5)
```

The parameters of `detectMultiScale` include `scaleFactor` and `minNeighbors`. The `scaleFactor` argument, which should be greater than 1.0, determines the downscaling ratio of the image at each iteration of the face detection process. As we discussed earlier in the *Conceptualizing Haar cascades* section, this downscaling is intended to achieve scale invariance by matching various faces to the window size. The `minNeighbors` argument is the minimum number of overlapping detections that are required in order to retain a detection result. Normally, we expect that a face may be detected in multiple overlapping windows, and a greater number of overlapping detections makes us more confident that the detected face is truly a face.

The value returned from the detection operation is a list of tuples that represent the face rectangles. OpenCV's `cv2.rectangle` function allows us to draw rectangles at the specified coordinates. `x` and `y` represent the left and top coordinates, while `w` and `h` represent the width and height of the face rectangle. We draw cyan rectangles around all of the faces we find by looping through the `faces` variable, making sure we use the original image for drawing, not the gray version:

```
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (255, 255, 0), 2)
```

Lastly, we call `cv2.imshow` to display the resulting processed image, and we call `cv2.imwrite` to save it. As usual, to prevent the image window from closing automatically, we insert a call to `waitKey`, which returns when the user presses any key:

```
cv2.imshow('Woodcutters Detected!', img)
cv2.imwrite('./woodcutters_detected.png', img)
cv2.waitKey(0)
```

And there we go, three members of the band of woodcutters have been detected in our image, as shown in the following screenshot:



Figure 5.1: Face detection results of a photograph of woodcutters (Image credit: Prokudin-Gorsky)

Note

The photograph in this example is the work of Sergey Prokudin-Gorsky (1863-1944), a pioneer of color photography. Tsar Nicholas II sponsored Prokudin-Gorsky to photograph people and places throughout the Russian Empire as a vast documentary project. Prokudin-Gorsky photographed these woodcutters near the Svir River, in northwestern Russia, in 1909.

Here, we have no false positive detections; all three rectangles really are woodcutters' faces. However, we do have two false negatives (woodcutters whose faces were not detected). Try adjusting the parameters of `face_cascade.detectMultiScale` to see how the results change. Then, let's proceed to a more interactive example.

Performing face detection on a video

We now understand how to perform face detection on a still image. As mentioned previously, we can repeat the process of face detection on each frame of a video (be it a camera feed or a pre-recorded video file).

The next script will open a camera feed, read a frame, examine that frame for faces, and scan for eyes within the detected faces. Finally, it will draw blue rectangles around the faces and green rectangles around the eyes. Here is the script in its entirety:

```
import cv2

face_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_eye.xml')

camera = cv2.VideoCapture(0)
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(
            gray, 1.3, 5, minSize=(120, 120))
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            roi_gray = gray[y:y+h, x:x+w]
            eyes = eye_cascade.detectMultiScale(
                roi_gray, 1.1, 5, minSize=(40, 40))
            for (ex, ey, ew, eh) in eyes:
                cv2.rectangle(frame, (x+ex, y+ey),
                              (x+ex+ew, y+ey+eh), (0, 255, 0), 2)
        cv2.imshow('Face Detection', frame)
```

Let's break up the preceding sample into smaller, more digestible chunks:

1. As usual, we import the `cv2` module. After that, we initialize two `CascadeClassifier` objects, one for faces and another for eyes:

   ```
   face_cascade = cv2.CascadeClassifier(
       f'../data/haarcascades/haarcascade_frontalface_default.xml')
   eye_cascade = cv2.CascadeClassifier(
       f'../data/haarcascades/haarcascade_eye.xml')
   ```

2. As in most of our interactive scripts, we open a camera feed and start iterating over frames. We continue until the user presses any key. Whenever we successfully capture a frame, we convert it into grayscale as our first step in processing it:

```
camera = cv2.VideoCapture(0)
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

3. We detect faces with the `detectMultiScale` method of our face detector. As we have previously done, we use the `scaleFactor` and `minNeighbors` arguments. We also use the `minSize` argument to specify a minimum size of a face, specifically 120x120. No attempt will be made to detect faces smaller than this. (Assuming that our user is sitting close to the camera, it is safe to say that the user's face will be larger than 120x120 pixels.) Here is the call to `detectMultiScale`:

```
        Faces = face_cascade.detectMultiScale(
            gray, 1.3, 5, minSize=(120, 120))
```

4. We iterate over the rectangles of the detected faces. We draw a blue border around each rectangle in the original color image. Then, within the same rectangular region of the grayscale image, we perform eye detection:

```
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            roi_gray = gray[y:y+h, x:x+w]
            eyes = eye_cascade.detectMultiScale(
                roi_gray, 1.1, 5, minSize=(40, 40))
```

Tip

The eye detector is a bit less accurate than the face detector. You might see shadows, parts of the frames of glasses, or other regions of the face falsely detected as eyes. To improve the results, you could try defining `roi_gray` as a smaller region of the face, since we can make a good guess about the eyes' location in an upright face. You could also try using a `maxSize` argument to avoid false positives that are too large to be eyes. Also, you could adjust `minSize` and `maxSize` so that the dimensions are proportional to `w` and `h`, the

> size of the detected face. As an exercise, feel free to experiment with changes to these and other parameters.

5. We loop through the resulting eye rectangles and draw green outlines around them:

```
            for (ex, ey, ew, eh) in eyes:
                cv2.rectangle(frame, (x+ex, y+ey),
                              (x+ex+ew, y+ey+eh), (0, 255, 0), 2)
```

6. Finally, we show the resulting frame in the window:

```
cv2.imshow('Face Detection', frame)
```

Run the script. If our detectors produce accurate results, and if any face is within the field of view of the camera, you should see a blue rectangle around the face and a green rectangle around each eye, as shown in this screenshot:

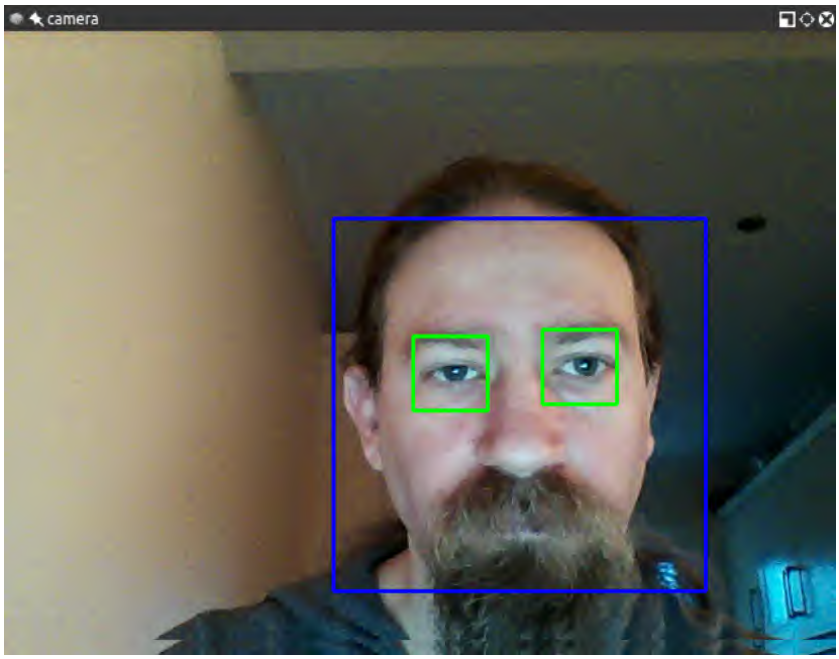


Figure 5.2: The resulting detection of a face and eyes in a photograph

Experiment with this script to see how the face and eye detectors perform under various conditions. Try a brighter or darker room. If you wear glasses, try removing them. Try various

people's faces and various expressions. Adjust the detection parameters in the script to see how they affect the results. When you are satisfied, let's consider what else we can do with faces in OpenCV.

Performing face recognition

Detecting faces is a fantastic feature of OpenCV and one that constitutes the basis for a more advanced operation: face recognition. What is face recognition? It is the ability of a program, given an image or a video feed containing a person's face, to identify that person. One of the ways to achieve this (and the approach adopted by OpenCV) is to *train* the program by feeding it a set of classified pictures (a facial database) and perform recognition based on the features of those pictures.

Another important feature of OpenCV's face recognition module is that each recognition has a confidence score, which allows us to set thresholds in real-life applications to limit the incidence of false identifications.

Let's start from the very beginning; to perform face recognition, we need faces to recognize. We fulfill this requirement in two ways: supply the images ourselves or obtain freely available face databases. A large directory of face databases is available online at `http://www.face-rec.org/databases/`.

If we trained a face recognizer on these samples, we would then have to run face recognition on an image that contains the face of one of the sampled people. This process might be educational, but perhaps not as satisfying as providing images of our own. You probably had the same thought that many computer vision learners have had: I wonder if I can write a program that recognizes my face with a certain degree of confidence. Indeed, you can, and soon you will!

Generating the data for face recognition

Let's go ahead and write a script that will generate those images for us. A few images containing different expressions are all that we need, but it is preferable that the training images are square and are all the same size. Our sample script uses a size of 200x200, but most freely available datasets have smaller images than this.

Here is the script itself:

```
import cv2
import os

output_folder = '../data/at/my_name'
if not os.path.exists(output_folder):
```

```
    os.makedirs(output_folder)

face_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_eye.xml')

camera = cv2.VideoCapture(0)
count = 0
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(
            gray, 1.3, 5, minSize=(120, 120))
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            face_img = cv2.resize(gray[y:y+h, x:x+w], (200, 200))
            face_filename = '%s/%d.pgm' % (output_folder, count)
            cv2.imwrite(face_filename, face_img)
            count += 1
        cv2.imshow('Capturing Faces...', frame)
```

Here, we are generating sample images by building on our newfound knowledge of how to detect a face in a video feed. We are detecting a face, cropping that region of the grayscale-converted frame, resizing it to be 200x200 pixels, and saving it as a PGM file with a name in a particular folder (in this case, `jm`, one of the authors' initials; you can use your own initials). Like many of our windowed applications, this one runs until the user presses any key.

The `count` variable is present because we needed progressive names for the images. Run the script for a few seconds, change your facial expression a few times, and check the destination folder you specified in the script. You will find a number of images of your face, grayed, resized, and named with the format `<count>.pgm`.

Modify the `output_folder` variable to make it match your name. For example, you might choose `'../data/at/my_name'`. Run the script, wait for it to detect your face in a number of frames (say, 20 or more), and then press any key to quit. Now, modify the `output_folder` variable again to make it match the name of a friend whom you also want to recognize. For example, you might choose `'../data/at/name_of_my_friend'`. Do not change the base part of the folder (in this case, `'../data/at'`) because later, in the *Loading the training data for face*

recognition section, we will write code that loads the training images from all of the subfolders of this base folder. Ask your friend to sit in front of the camera, run the script again, let it detect your friend's face in a number of frames, and then quit. Repeat this process for any additional people you might want to recognize.

Let's now move on to try and recognize the user's face in a video feed. This should be fun!

Choosing a face recognition algorithm

OpenCV 5 implements three different algorithms for recognizing faces: **Eigenfaces**, **Fisherfaces**, and **Local Binary Pattern Histograms** (**LBPHs**). Eigenfaces and Fisherfaces are derived from a more general-purpose algorithm called **Principal Component Analysis** (**PCA**). For a detailed description of the algorithms, refer to the following links:

- **PCA**: *A Tutorial on Principal Component Analysis* (2013), by Jonathon Shlens, is available at `http://arxiv.org/pdf/1404.1100v1.pdf`. This algorithm was invented in 1901 by Karl Pearson, and the original paper, *On Lines and Planes of Closest Fit to Systems of Points in Space*, is available at `http://pca.narod.ru/pearson1901.pdf`.
- **Eigenfaces**: The paper *Eigenfaces for Recognition* (1991), by *Matthew Turk and Alex Pentland*, is available at `https://www.face-rec.org/algorithms/PCA/jcn.pdf`.
- **Fisherfaces**: The seminal paper *The Use of Multiple Measurements in Taxonomic Problems (1936)*, by *R. A. Fisher*, is available at `http://onlinelibrary.wiley.com/doi/10.1111/j.1469-1809.1936.tb02137.x/pdf`.
- **Local Binary Pattern**: The first paper describing this algorithm is *Performance evaluation of texture measures with classification based on Kullback discrimination of distributions (1994)*, by *T. Ojala, M. Pietikainen, and D. Harwood.* It is available at `https://ieeexplore.ieee.org/document/576366`.

For this book's purposes, let's just take a high-level overview of the algorithms. First and foremost, they all follow a similar process; they take a set of classified observations (our face database, containing numerous samples per individual), train a model based on it, perform an analysis of face images (which may be face regions that we detected in an image or video), and determine two things: the subject's identity and a measure of confidence that this identification is correct. The latter is commonly known as the **confidence score**.

Eigenfaces performs PCA, which identifies principal components of a certain set of observations (again, your face database), calculates the divergence of the current observation (the face being detected in an image or frame) compared to the dataset, and produces a value. The smaller the value, the smaller the difference between the face database and the detected face; hence, a value of 0 is an exact match.

Fisherfaces also derives from PCA and evolves the concept, applying more complex logic. While computationally more intensive, it tends to yield more accurate results than Eigenfaces.

LBPH instead divides a detected face into small cells and, for each cell, builds a histogram that describes whether the brightness of the image is increasing when comparing neighboring pixels in a given direction. This cell's histogram can be compared to the histogram of the corresponding cell in the model, producing a measure of similarity. Of the face recognizers in OpenCV, the implementation of LBPH is the only one that allows the model sample faces and the detected faces to be of a different shape and size. Hence, it is a convenient option, and the authors of this book find that its accuracy compares favorably to the other two options.

Despite the algorithms' differences, OpenCV provides a similar interface for all three, as we shall soon see.

Loading the training data for face recognition

Regardless of our choice of face recognition algorithm, we can load the training images in the same way. Earlier, in the *Generating the data for face recognition* section, we generated training images and saved them in folders that were organized according to people's names or initials. For example, the following folder structure could contain sample face images of this book's authors, Joseph Howse (`jh`) and Joe Minichino (`jm`):

```
../
  data/
    at/
      jh/
      jm/
```

Let's write a script that loads these images and labels them in a way that OpenCV's face recognizers will understand. To work with the filesystem and the data, we will use the Python standard library's `os` module, as well as the `cv2` and `numpy` modules. Let's create a script that starts with the following `import` statements:

```
import os

import cv2
import numpy
```

Let's add the following `read_images` function, which walks through a directory's subdirectories, loads the images and shows each of them for user information purposes, resizes them to a specified size, and puts the resized images in a list. The order of this list has no significance in itself but, at the same time, our function builds two corresponding lists that

describe the images: first, a corresponding list of people's names or initials (based on the subfolder names), and second, a corresponding list of labels or numeric IDs associated with the loaded images. For example, `jh` could be a name, and `0` could be the label for all images that were loaded from the `jh` subfolder. Finally, the function converts the lists of images and labels into NumPy arrays, and it returns three variables: the list of names, the NumPy array of images, and the NumPy array of labels. Here is the function's implementation:

```
def read_images(path, image_size):
    names = []
    training_images, training_labels = [], []
    label = 0
    for dirname, subdirnames, filenames in os.walk(path):
        for subdirname in subdirnames:
            names.append(subdirname)
            subject_path = os.path.join(dirname, subdirname)
            for filename in os.listdir(subject_path):
                img = cv2.imread(os.path.join(subject_path, filename),
                                 cv2.IMREAD_GRAYSCALE)
                if img is None:
                    # The file cannot be loaded as an image.
                    # Skip it.
                    continue
                img = cv2.resize(img, image_size)
                cv2.imshow('training', img)
                cv2.waitKey(5)
                training_images.append(img)
                training_labels.append(label)
            label += 1
    training_images = numpy.asarray(training_images, numpy.uint8)
    training_labels = numpy.asarray(training_labels, numpy.int32)
    return names, training_images, training_labels
```

Let's call our `read_images` function by adding code such as the following:

```
path_to_training_images = '../data/at'
training_image_size = (200, 200)
names, training_images, training_labels = read_images(
    path_to_training_images, training_image_size)
```

Tip

Edit the `path_to_training_images` variable in the preceding code block to ensure that it matches the base folder of the `output_folder` variables you defined earlier in the code for the section *Generating the data for face recognition*.

So far, we have training data in a useful format, but we have not yet created a face recognizer or performed any training. We will do so in the next section, where we continue the implementation of the same script.

Performing face recognition with Eigenfaces

Now that we have an array of training images and an array of their labels, we can create and train a face recognizer with just two more lines of code:

```
model = cv2.face.EigenFaceRecognizer_create()
model.train(training_images, training_labels)
```

What have we done here? We created the Eigenfaces face recognizer with OpenCV's `cv2.EigenFaceRecognizer_create` function, and we trained the recognizer by passing the arrays of images and labels (numeric IDs). Optionally, we could have passed two arguments to `cv2.EigenFaceRecognizer_create`:

- `num_components`: This is the number of components to keep for the PCA.
- `threshold`: This is a floating-point value specifying a confidence threshold. Faces with a confidence score below the threshold will be discarded. By default, the threshold is the maximum floating-point value so that no faces are discarded.

Note

Having trained this model, we could save it to a file using code such as `model.write('my_model.xml')`. Later (for example, in another script), we could skip the training step and just load the pre-trained model from the file using code such as `model.read('my_model.xml')`. However, for the sake of our simple demo, we will just train and test the model in one script without saving the model to a file.

To test this recognizer, let's use a face detector and a video feed from a camera. As we have done in previous scripts, we can use the following line of code to initialize the face detector:

```
face_cascade = cv2.CascadeClassifier(
    f'../data/haarcascades/haarcascade_frontalface_default.xml')
```

The following code initializes the camera feed, iterates over frames (until the user presses any key), and performs face detection and recognition on each frame:

```
camera = cv2.VideoCapture(0)
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        faces = face_cascade.detectMultiScale(frame, 1.3, 5)
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            roi_gray = gray[x:x+w, y:y+h]
            if roi_gray.size == 0:
                # The ROI is empty. Maybe the face is at the image edge.
                # Skip it.
                continue
            roi_gray = cv2.resize(roi_gray, training_image_size)
            label, confidence = model.predict(roi_gray)
            text = '%s, confidence=%.2f' % (names[label], confidence)
            cv2.putText(frame, text, (x, y - 20),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
        cv2.imshow('Face Recognition', frame)
```

Let's walk through the most important functionality of the preceding block of code. For each detected face, we convert and resize it so that we have a grayscale version that matches the expected size (in this case, 200x200 pixels as defined by the `training_image_size` variable in the previous section, *Loading the training data for face recognition*). Then, we pass the resized, grayscale face to the face recognizer's `predict` function. This returns a label and confidence score. We look up the person's name corresponding to the numeric label of that face. (Remember that we created the `names` array in the previous section, *Loading the training data for face recognition*.) We draw the name and confidence score in blue text above the recognized face. After iterating over all detected faces, we display the annotated image.

> **Note**
>
> We have taken a simple approach to face detection and recognition, and it serves the purpose of enabling you to have a basic application running and understand the process of face recognition in OpenCV 5. To improve upon this approach and make it more robust, you could take further steps such as correctly aligning and rotating detected faces so that the accuracy of the recognition is maximized.

When you run the script, you should see something similar to the following screenshot:

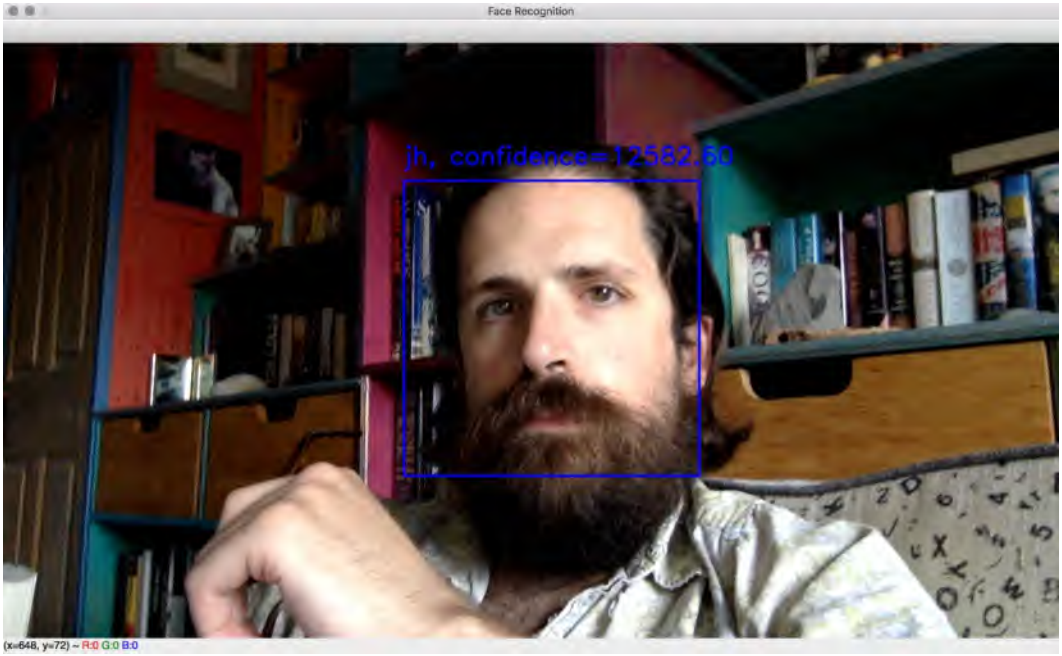


Figure 5.3: A face recognition result using live camera input

Next, let's consider how we would adapt this script to replace Eigenfaces with another face recognition algorithm.

Performing face recognition with Fisherfaces

What about Fisherfaces? The process does not change much; we simply need to instantiate a different algorithm. With default arguments, the declaration of our `model` variable would look like this:

```
model = cv2.face.FisherFaceRecognizer_create()
```

`cv2.face.FisherFaceRecognizer_create` takes the same two optional arguments as `cv2.createEigenFaceRecognizer_create`: the number of principal components to keep and the confidence threshold. As you might guess, these types of `*_create` functions are quite common in OpenCV for situations where several different algorithms share a common interface. Let's look at one more example.

Performing face recognition with LBPH

For the LBPH algorithm, again, the process is similar. However, the algorithm factory takes the following optional parameters (in order):

- `radius`: The pixel distance between the neighbors that are used to calculate a cell's histogram (by default, 1)
- `neighbors`: The number of neighbors used to calculate a cell's histogram (by default, 8)
- `grid_x`: The number of cells into which the face is divided horizontally (by default, 8)
- `grid_y`: The number of cells into which the face is divided vertically (by default, 8)
- `confidence`: The confidence threshold (by default, the highest possible floating-point value so that no results are discarded)

With default arguments, the model declaration would look like this:

```
model = cv2.face.LBPHFaceRecognizer_create()
```

Note

Note that, with LBPH, we do not need to resize images as the division into grids allows a comparison of patterns identified in each cell.

Having looked at the available face recognition algorithms, let's now consider how to evaluate a recognition result.

Discarding results based on the confidence score

The `predict` method returns a tuple, in which the first element is the label of the recognized individual and the second is the confidence score. All algorithms come with the option of setting a confidence score threshold, which measures the distance of the recognized face from the original model; therefore, a score of 0 signifies an exact match.

There may be cases in which you would rather retain all recognitions and then apply further processing, so you can come up with your own algorithms to estimate the confidence score of a recognition. For example, if you are trying to identify people in a video, you may want to

analyze the confidence score in subsequent frames to establish whether the recognition was successful or not. In this case, you can inspect the confidence score obtained by the algorithm and draw your own conclusions.

> **Note**
>
> The typical range of the confidence score depends on the algorithm. Eigenfaces and Fisherfaces produce values (roughly) in the range of 0 to 20,000, with any score below 4,000-5,000 indicating a fairly confident recognition. For LBPH, a good recognition scores, roughly below 50, and any value above 80 is considered a poor confidence score.

A normal custom approach would be to hold off drawing a rectangle around a recognized face until we have a number of frames with a good confidence score (where "good" is an arbitrary threshold we must choose, based on our algorithm and use case), but you have total freedom to use OpenCV's face recognition module to tailor your application to your needs. Next, let's see how face detection and recognition work in a specialized use case.

Swapping faces in infrared

Face detection and recognition are not limited to the visible spectrum of light. With a **near-infrared** (**NIR**) camera and an NIR light source, face detection and recognition are possible even when a scene appears totally dark to the human eye. This capability is quite useful in security and surveillance applications.

We studied the basic usage of NIR depth cameras, such as the Asus Xtion PRO, in *Chapter 4, Depth Estimation and Segmentation*. We extended the object-oriented code of our interactive application, Cameo. We captured frames from a depth camera. Based on depth, we segmented each frame into a main layer (such as the user's face) and other layers. We painted the other layers black. This achieved the effect of hiding the background so that only the main layer (the user's face) appeared on-screen in the interactive video feed.

Now, let's modify Cameo to do something that exercises our previous skills in depth segmentation and our new skills in face detection. Let's detect faces and, when we detect at least two faces in a frame, let's swap the faces so that one person's head appears atop another person's body. Rather than copying all pixels in a detected face rectangle, we will only copy the pixels that are part of the main depth layer for that rectangle. This should achieve the effect of swapping faces but not the background pixels surrounding the faces.

Once the changes are complete, Cameo will be able to produce output such as the following screenshot:



Figure 5.4: The result of swapping two faces in a frame from an infrared camera

Here, we see the face of Joseph Howse swapped with the face of Janet Howse, his mother. Although Cameo is copying pixels from rectangular regions (and this is clearly visible at the bottom of the swapped regions, in the foreground), some of the background pixels within the rectangles are masked based on depth and thus are not swapped, so we do not see rectangular edges everywhere.

You can find all of the relevant changes to the Cameo source code in this book's repository, specifically in the `chapter05/cameo` folder. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

For brevity, we will not discuss all of the changes here in this book, but we will cover some of the highlights in the next two subsections, *Modifying the application's loop* and *Masking a copy operation*.

Modifying the application's loop

To support face swapping, the Cameo project has two new modules called `rects` and `trackers`. The `rects` module contains functions for copying and swapping rectangles, with an optional mask to limit the copy or swap operation to particular pixels. The `trackers` module contains a class called `FaceTracker`, which adapts OpenCV's face detection functionality to an object-oriented style of programming.

As we have covered OpenCV's face detection functionality earlier in this chapter, and we have demonstrated an object-oriented programming style in previous chapters, we will not go into the `FaceTracker` implementation here. Instead, you may look at it in this book's repository.

Let's open `cameo.py` so that we can walk through the overall changes to the application:

1. Near the top of the file, we need to import our new modules, as highlighted in the following code block:

```
import cv2
import depth
import filters
from managers import WindowManager, CaptureManager
import rects
from trackers import FaceTracker
```

2. Now, let's turn our attention to changes in the `__init__` method of our `CameoDepth` class. Our updated application uses an instance of `FaceTracker`. As part of its functionality, `FaceTracker` can draw rectangles around detected faces. Let's give Cameo's user the option to enable or disable the drawing of face rectangles. We will keep track of the currently selected option via a Boolean variable. The following code block highlights the necessary changes to initialize the `FaceTracker` object and the Boolean variable:

```
class CameoDepth(Cameo):

    def __init__(self):
        self._windowManager = WindowManager(
            'Cameo', self.onKeypress)
        #device = cv2.CAP_OPENNI2 # uncomment for Kinect
        device = cv2.CAP_OPENNI2_ASUS # uncomment for Xtion
        self._captureManager = CaptureManager(
            cv2.VideoCapture(device), self._windowManager, True)
```

```
        self._faceTracker = FaceTracker()
        self._shouldDrawDebugRects = False
        self._curveFilter = filters.BGRPortraCurveFilter()
```

We make use of the `FaceTracker` object in the `run` method of `CameoDepth`, which contains the application's main loop that captures and processes frames. Every time we successfully capture a frame, we call methods of **FaceTracker** to update the face detection result and get the latest detected faces. Then, for each face, we create a mask based on the depth camera's disparity map. (Previously, in *Chapter 4, Depth Estimation and Segmentation*, we created such a mask for the entire image instead of a mask for each face rectangle.)

3. Then, we call a function, `rects.swapRects`, to perform a masked swap of the face rectangles. (We will look at the implementation of `swapRects` a little later, in the *Masking a copy operation* section.) Depending on the currently selected option, we might tell `FaceTracker` to draw rectangles around the faces. All of the relevant changes are `highlighted` in the following code block:

```
    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            # ... The logic for capturing a frame is unchanged ...
            if frame is not None:
                self._faceTracker.update(frame)
                faces = self._faceTracker.faces
                masks = [
                    depth.createMedianMask(
                        disparityMap, validDepthMask,
                        face.faceRect) \
                    for face in faces
                ]
                rects.swapRects(frame, frame,
                                [face.faceRect for face in faces],
                                masks)

                if self._captureManager.channel == \
                        cv2.CAP_OPENNI_BGR_IMAGE:
                    # A BGR frame was captured.
                    # Apply filters to it.
```

```
                    filters.strokeEdges(frame, frame)
                    self._curveFilter.apply(frame, frame)

                if self._shouldDrawDebugRects:
                    self._faceTracker.drawDebugRects(frame)

            self._captureManager.exitFrame()
            self._windowManager.processEvents()
```

4. Finally, let's modify the `onKeypress` method so that the user can hit the `X` key to start or stop displaying rectangles around detected faces. Again, the relevant changes are `highlighted` in the following code block:

```
    def onKeypress(self, keycode):
        """Handle a keypress.

        space -> Take a screenshot.
        tab -> Start/stop recording a screencast.
        x -> Start/stop drawing debug rectangles around faces.
        escape -> Quit.

        """
        if keycode == 32: # space
            self._captureManager.writeImage('screenshot.png')
        elif keycode == 9: # tab
            if not self._captureManager.isWritingVideo:
                self._captureManager.startWritingVideo(
                    'screencast.avi')
            else:
                self._captureManager.stopWritingVideo()
        elif keycode == 120: # x
            self._shouldDrawDebugRects = \
                not self._shouldDrawDebugRects
        elif keycode == 27: # escape
            self._windowManager.destroyWindow()
```

Next, let's look at the implementation of the `rects` module that we imported earlier in this section.

Masking a copy operation

The `rects` module is implemented in `rects.py`. We already saw a call to the `rects.swapRects` function in the previous section. However, before we consider the implementation of `swapRects`, we first need a more basic `copyRect` function.

As far back as *Chapter 2, Handling Files, Cameras, and GUIs*, we learned how to copy data from one rectangular **region of interest** (**ROI**) to another using NumPy's slicing syntax. Outside the ROIs, the source and destination images were unaffected. Now, we want to apply further limits to this copy operation. We want to use a given mask that has the same dimensions as the source rectangle.

We shall copy only those pixels in the source rectangle where the mask's value is not zero. Other pixels shall retain their old values from the destination image. This logic, with an array of conditions and two arrays of possible output values, can be expressed concisely with the `numpy.where` function.

With this approach in mind, let's consider our `copyRect` function. As arguments, it takes a source and destination image, a source and destination rectangle, and a mask. The latter may be `None`, in which case we simply resize the content of the source rectangle to match the destination rectangle and then assign the resulting resized content to the destination rectangle. Otherwise, we next ensure that the mask and the images have the same number of channels. We assume that the mask has one channel, but the images may have three channels (BGR). We can add duplicate channels to the mask using the `repeat` and `reshape` methods of `numpy.ndarray`. Finally, we perform the copy operation using `numpy.where`. The complete implementation is as follows:

```
def copyRect(src, dst, srcRect, dstRect, mask = None,
             interpolation = cv2.INTER_LINEAR):
    """Copy part of the source to part of the destination."""

    x0, y0, w0, h0 = srcRect
    x1, y1, w1, h1 = dstRect

    # Resize the contents of the source sub-rectangle.
    # Put the result in the destination sub-rectangle.
    if mask is None:
        dst[y1:y1+h1, x1:x1+w1] = \
            cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                       interpolation = interpolation)
    else:
```

```
        if not utils.isGray(src):
            # Convert the mask to 3 channels, like the image.
            mask = mask.repeat(3).reshape(h0, w0, 3)
        # Perform the copy, with the mask applied.
        dst[y1:y1+h1, x1:x1+w1] = \
            numpy.where(cv2.resize(mask, (w1, h1),
                                   interpolation = \
                                   cv2.INTER_NEAREST),
                        cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                                   interpolation = interpolation),
                        dst[y1:y1+h1, x1:x1+w1])
```

We also need to define a `swapRects` function, which uses `copyRect` to perform a circular swap of a list of rectangular regions. `swapRects` has a `masks` argument, which is a list of masks whose elements are passed to the respective `copyRect` calls. If the value of the `masks` argument is `None`, we pass `None` to every `copyRect` call. The following code shows the full implementation of `swapRects`:

```
def swapRects(src, dst, rects, masks = None,
              interpolation = cv2.INTER_LINEAR):
    """Copy the source with two or more sub-rectangles swapped."""

    if dst is not src:
        dst[:] = src
    numRects = len(rects)
    if numRects < 2:
        return
    if masks is None:
        masks = [None] * numRects

    # Copy the contents of the last rectangle into temporary storage.
    x, y, w, h = rects[numRects - 1]
    temp = src[y:y+h, x:x+w].copy()

    # Copy the contents of each rectangle into the next.
    i = numRects - 2
    while i >= 0:
        copyRect(src, dst, rects[i], rects[i+1], masks[i], interpolation)
        i -= 1
```

```
    # Copy the temporarily stored content into the first rectangle.
    copyRect(temp, dst, (0, 0, w, h), rects[0],
             masks[numRects - 1], interpolation)
```

Note that the `mask` argument in `copyRect` and the `masks` argument in `swapRects` both have a default value of `None`. If no mask is specified, these functions copy or swap the entire contents of the rectangle or rectangles.

Summary

By now, you should have a good understanding of how face detection and face recognition work and how to implement them in Python and OpenCV 5.

The accuracy of detection and recognition algorithms heavily depends on the quality of the training data, so make sure you provide your applications with a large number of training images covering a variety of expressions, poses, and lighting conditions. Later in this book, in *Chapter 11, Introduction to Neural Networks with OpenCV*, we will look at how to use several robust, pre-trained face detection models that build atop advanced algorithms and large sets of training data.

As human beings, we might be predisposed to think that human faces are particularly recognizable. We might even be overconfident in our own face recognition abilities. However, in computer vision, there is nothing very special about human faces, and we can just as readily use algorithms to find and identify other things. We will begin to do so next in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*.

6

Retrieving Images and Searching Using Image Descriptors

Similar to the human eyes and brain, OpenCV can detect the main features of an image and extract them into **image descriptors**. These features can then be used as a database, enabling image-based searches. Moreover, we can use keypoints to stitch images together and compose a bigger image. (Think of putting together many pictures to form a 360° panorama.)

This chapter will show you how to detect the features of an image with OpenCV and make use of them to match and search images. Over the course of this chapter, we will take sample images and detect their main features, and then try to find a region of another image that matches the sample image. We will also find the homography or spatial relationship between a sample image and a matching region of another image.

More specifically, we will cover the following tasks:

- Detecting keypoints and extracting local descriptors around the keypoints using any of the following algorithms: **Harris corners**, **SIFT**, **SURF**, or **ORB**
- Matching keypoints using **brute-force algorithms** or the **FLANN algorithm**
- Filtering out bad matches using **KNN** and the **ratio test**
- Finding the homography between two sets of matching keypoints
- Searching a set of images to determine which one contains the best match for a reference image

We will finish this chapter by building a proof-of-concept forensic application. Given a reference image of a tattoo, we will search through a set of images of people in order to find a person with a matching tattoo.

Technical requirements

This chapter uses Python, OpenCV, and NumPy. With regard to OpenCV, we use the optional `opencv_contrib` modules, which include additional algorithms for keypoint detection and matching. To enable the **SURF** algorithm (which is patented and *not* free for commercial use), we must configure the `opencv_contrib` modules with the `OPENCV_ENABLE_NONFREE` flag in CMake. Please refer to *Chapter 1, Setting Up OpenCV*, for installation instructions. Additionally, if you have not already installed `Matplotlib`, install it by running `$ pip install matplotlib` (or `$ pip3 install matplotlib`, depending on your environment).

The complete code for this chapter can be found in this book's GitHub repository in the `chapter06` folder. The sample images can be found in the `images` folder.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

A subset of the chapter's sample code can be edited and run interactively in Google Colab.

Understanding types of feature detection and matching

A number of algorithms can be used to detect and describe features, and we will explore several of them in this section. The most commonly used feature detection and descriptor extraction algorithms in OpenCV are as follows:

- **Harris**: This algorithm is useful for detecting corners
- **Scale-Invariant Feature Transform** (**SIFT**): This algorithm is useful for detecting blobs
- **Speeded Up Robust Features (SURF)**: This algorithm is useful for detecting blobs
- **Features from Accelerated Segment Test** (**FAST**): This algorithm is useful for detecting corners
- Binary Robust Independent Elementary Features (BRIEF): This algorithm is useful for detecting blobs
- **Oriented FAST and Rotated BRIEF (ORB)**: This algorithm is useful for detecting a combination of corners and blobs

Matching features can be performed with the following methods:

- Brute-force matching
- Fast Library for Approximate Nearest Neighbors (FLANN)-based matching

Spatial verification can then be performed with homography.

We have just introduced a lot of new terminology and algorithms. Now, we will go over their basic definitions.

Defining features

What is a feature, exactly? Why is a particular area of an image classifiable as a feature, while others are not? Broadly speaking, a feature is an area of interest in the image that is unique or easily recognizable. **Corners** and regions with a high density of textural detail are good features, while patterns that repeat themselves a lot and low-density regions (such as a blue sky) are not. **Edges** are good features as they tend to divide two regions with abrupt changes in the grayscale or color values. A **blob** (a region of an image that greatly differs from its surrounding areas) is also an interesting feature. For example, in a photo of a room (if it is daytime and the photographer is not using flash), a window may be a blob, as it is much brighter than the surrounding walls. Obviously, a window may have corners and edges too, yet even in a blurry photo where the corners and edges are unclear, we could still see the window as a bright blob.

Most feature detection algorithms revolve around the identification of corners, edges, and blobs, with some also focusing on the concept of a **ridge**, which you can conceptualize as the axis of symmetry of an elongated object. Think, for example, about identifying a two-lane road in an image. The road's dividing line is an axis of symmetry, and this clearly visible feature should help us distinguish a road from other elongated objects, such as a river.

Some algorithms are better at identifying and extracting features of a certain type, so it is important to know what your input image is; then, you can utilize the best tool in your OpenCV belt. Let's start by finding corners in an image containing a grid.

Detecting Harris corners

We can find corners using the Harris corner detection algorithm. We do not have to look far to find suitable test subjects. If you continue to study OpenCV beyond this book, you will discover that chessboards are a common subject of analysis in computer vision, partly because a checkered pattern is suited to many types of feature detection, and partly because chess is a popular pastime, especially in Russia, which is the country of origin of many of OpenCV's developers.

Here is our sample image of a chessboard and chess pieces:

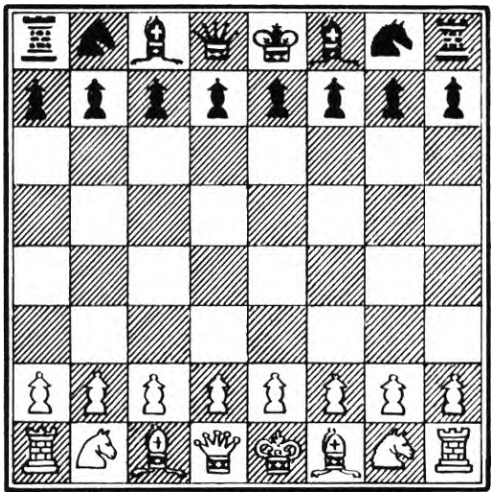


Figure 6.1: A chessboard, to be used for corner detection

OpenCV has a handy function called `cv2.cornerHarris`, which detects corners in an image. We can see this function at work in the following basic example:

```
import cv2

img = cv2.imread('../images/chess_board.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
dst = cv2.cornerHarris(gray, 2, 23, 0.04)
img[dst > 0.01 * dst.max()] = [0, 0, 255]
cv2.imshow('corners', img)
cv2.waitKey()
```

Let's analyze the code. After the usual imports, we load the chessboard image and convert it into grayscale. Then, we call the `cornerHarris` function:

```
dst = cv2.cornerHarris(gray, 2, 23, 0.04)
```

The most important parameter here is the third one, which defines the aperture or kernel size of the Sobel operator. The Sobel operator detects edges by measuring horizontal and vertical differences between pixel values in a neighborhood, and it does this using a kernel. The `cv2.cornerHarris` function uses a Sobel operator whose aperture is defined by this parameter. In plain English, the parameters define how sensitive corner detection is. It must be between 3

and 31 and be an odd value (so that the Sobel kernel is symmetrical around the central pixel, the possible corner). With a low (highly sensitive) value of 3, all those diagonal lines in the black squares of the chessboard will register as corners when they touch the border of the square. For a higher (less sensitive) value of 23, the detector will ignore most of the diagonal lines, but it will still register corners at the junctions of the horizontal and vertical lines (the chessboard's grid).

`cv2.cornerHarris` returns an image in floating-point format. Each value in this image represents a score for the corresponding pixel in the source image. A moderate or high score indicates that the pixel is likely to be a corner. Conversely, we can treat pixels with the lowest scores as non-corners. Consider the following line:

```
img[dst > 0.01 * dst.max()] = [0, 0, 255]
```

Here, we select pixels with scores that are at least 1% of the highest score, and we color these pixels red in the original image. Here is the result:

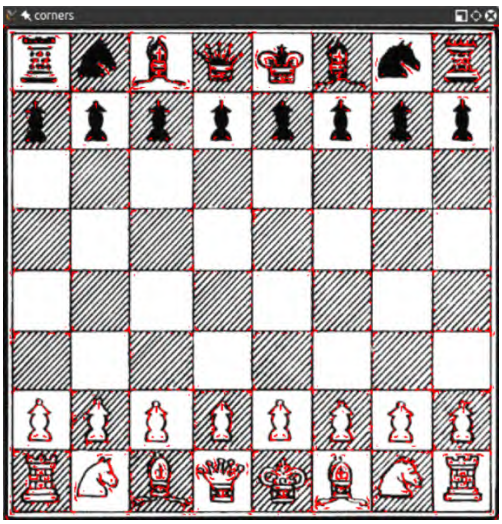


Figure 6.2: Corner detection in a chessboard

We can see that the detection results include nearly all corners in the grid. Some other corners, notably of the chess pieces, are detected too. Indeed, it is quite typical to see that feature detection (with any algorithm) produces noisy results, which would need further processing before we could make higher-level inferences about objects, such as "Here is square A2 of the chessboard" or "Here is a pawn."

Tip

If we tweak the second parameter in `cv2.cornerHarris`, we will see that smaller regions (for a smaller parameter value) or larger regions (for a larger parameter value) will be detected as corners. This parameter is called the block size.

Next, let's consider this corner detector's particular limitations, and how these are addressed in more advanced algorithms.

Detecting DoG features and extracting SIFT descriptors

The preceding technique, which uses `cv2.cornerHarris`, is great for detecting corners and has a distinct advantage because even if the image is rotated, corners are still corners. However, if we scale an image to a smaller or larger size, some parts of the image may lose or even gain corner quality.

For example, take a look at the following corner detections in an image of the F1 Italian Grand Prix track:



Figure 6.3: Corner detections in an image of the F1 Italian Grand Prix track

Here is the corner detection result with a smaller version of the same image:

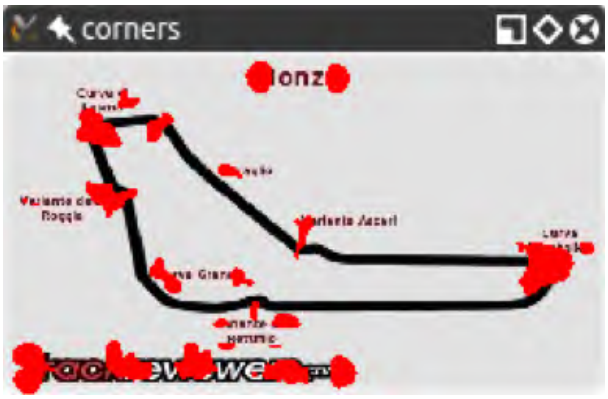


Figure 6.4: Corner detections in a smaller image of the F1 Italian Grand Prix track

You will notice how the corners are a lot more condensed; however, even though we gained some corners, we lost others! In particular, let's examine the **Variante Ascari** chicane, which looks like a squiggle at the end of the part of the track that runs straight from northwest to southeast. In the larger version of the image, both the entrance and the apex of the double bend were detected as corners. In the smaller image, the apex is not detected as such. If we further reduce the image, at some scale, we will lose the entrance to that chicane too.

This loss of features raises an issue: we need an algorithm that works regardless of the scale of the image. Enter **SIFT**. While the name may sound a bit mysterious, now that we know what problem we are trying to solve, it actually makes sense. We need a function (a transform) that will describe features (a feature transform) and will not output different results depending on the scale of the image (a scale-invariant feature transform). Note that SIFT does not *detect* keypoints. Keypoint *detection* is done with the **Difference of Gaussians** (**DoG**), while SIFT describes the region surrounding the keypoints by means of a feature vector.

A quick introduction to the DoG is in order. Previously, in *Chapter 3, Processing Images with OpenCV*, we talked about low-pass filters and blurring operations, specifically the `cv2.GaussianBlur` function. DoG is the result of applying different Gaussian filters to the same image. Previously, we applied this type of technique for edge detection, and the idea is the same here. The final result of a DoG operation contains areas of interest (keypoints), which are then going to be described through SIFT.

Let's see how DoG and SIFT behave in the following image, which is full of corners and features:



Figure 6.5: SIFT descriptors for an image of Varese, Lombardy, Italy

Here, the beautiful panorama of Varese (in Lombardy, Italy) gains a new type of fame as a subject of computer vision. Here is the code that produces this processed image:

```
import cv2

img = cv2.imread('../images/varese.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(gray, None)

cv2.drawKeypoints(img, keypoints, img, (51, 163, 236),
                  cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('sift_keypoints', img)
cv2.waitKey()
```

After the usual imports, we load the image we want to process. Then, we convert the image into grayscale. By now, you may have gathered that many methods in OpenCV expect a

grayscale image as input. The next step is to create a SIFT detection object and compute the features and descriptors of the grayscale image:

```
sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(gray, None)
```

Behind the scenes, these simple lines of code carry out an elaborate process. We create a `cv2.SIFT` object, which uses DoG to detect keypoints and then computes a feature vector for the surrounding region of each keypoint. As the name of the `detectAndCompute` method clearly suggests, two main operations are performed: feature detection and the computation of descriptors. The return value of the operation is a tuple containing a list of keypoints and another list of the keypoints' descriptors.

Finally, we process this image by drawing the keypoints on it with the `cv2.drawKeypoints` function and then displaying it with the usual `cv2.imshow` function. As one of its arguments, the `cv2.drawKeypoints` function accepts a flag that specifies the type of visualization we want. Here, we specify `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS` in order to draw a visualization of the scale and orientation of each keypoint.

Anatomy of a keypoint

Each keypoint is an instance of the `cv2.KeyPoint` class, which has the following properties:

- The `pt` (point) property contains the *x* and *y* coordinates of the keypoint in the image.
- The `size` property indicates the diameter of the feature.
- The `angle` property indicates the orientation of the feature, as shown by the radial lines (like a clock's second hand) in the preceding processed image.
- The `response` property indicates the strength of the keypoint. Some features are classified by SIFT as stronger than others, and `response` is the property you would check to evaluate the strength of a feature.
- The `octave` property indicates the layer in the image pyramid where the feature was found. Let's briefly review the concept of an image pyramid, which we discussed previously in *Chapter 5*, *Detecting and Recognizing Faces*, in the *Conceptualizing Haar cascades* section. The SIFT algorithm operates in a similar fashion to face detection algorithms in that it processes the same image iteratively but alters the input at each iteration. In particular, the scale of the image is a parameter that changes at each

iteration (`octave`) of the algorithm. Thus, the `octave` property is related to the image scale at which the keypoint was detected.

- Finally, the `class_id` property can be used to assign a custom identifier to a keypoint or a group of keypoints.

Let's proceed to examine some other approaches to feature detection and descriptor extraction, and see how they compare to SIFT in terms of their design and OpenCV's implementation of them.

Detecting Fast Hessian features and extracting SURF descriptors

Computer vision is a relatively young branch of computer science; so many famous algorithms and techniques have only been invented recently. SIFT is, in fact, only 27 years old, having been published by David Lowe in 1999.

SURF is a feature detection algorithm that was published in 2006 by Herbert Bay. SURF is several times faster than SIFT, and it is partially inspired by it.

> **Note**
>
> Note that SURF is a patented algorithm and, for this reason, is made available only in builds of `opencv_contrib` where the `OPENCV_ENABLE_NONFREE` CMake flag is used. SIFT was formerly a patented algorithm, but its patent expired, and now SIFT is available in standard builds of OpenCV.

It is not particularly relevant to this book to understand how SURF works under the hood, as long as we can use it in our applications and make the best of it. What is important to understand is that `cv2.SURF` is an OpenCV class that performs keypoint detection with the Fast Hessian algorithm and descriptor extraction with SURF, much like the `cv2.SIFT` class performs keypoint detection with DoG and descriptor extraction with SIFT.

Also, the good news is that OpenCV provides a standardized API for all its supported feature detection and descriptor extraction algorithms. Thus, with only trivial changes, we can adapt our previous code sample to use SURF instead of SIFT. Here is the modified code, with the changes `highlighted`:

```
import cv2

img = cv2.imread('../images/varese.jpg')
```

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

surf = cv2.xfeatures2d.SURF_create(8000)
keypoints, descriptors = surf.detectAndCompute(gray, None)

cv2.drawKeypoints(img, keypoints, img, (51, 163, 236),
                  cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('surf_keypoints', img)
cv2.waitKey()
```

The parameter to `cv2.xfeatures2d.SURF_create` is a threshold for the Fast Hessian algorithm. By increasing the threshold, we can reduce the number of features that will be retained. With a threshold of `8000`, we get the following result:



Figure 6.6: SURF descriptors for an image of Varese, Lombardy, Italy

Try adjusting the threshold to see how it affects the result. As an exercise, you may want to build a GUI application with a slider that controls the value of the threshold. This way, a user can adjust the threshold and see the number of features increase and decrease in an inversely proportional fashion. We built a GUI application with sliders in *Chapter 4*, *Depth Estimation and Segmentation*, in the *Depth estimation with a normal camera* section, so you may want to refer back to that section as a guide.

Note

OpenCV and `opencv_contrib`'s `xfeatures2d` module implement a large number of keypoint detectors and descriptor extractors with a similar interface to the `SIFT_create` and `SURF_create` functions that we have just seen. For example, you may wish to experiment with the following alternatives:

- `cv2.AgastFeatureDetector_create`
- `cv2.AKAZE_create`
- `cv2.BRISK_create`
- `cv2.FastFeatureDetector_create`
- `cv2.GFTTDetector_create`
- `cv2.KAZE_create`
- `cv2.MSER_create`
- `cv2.ORB_create`: We will look at ORB together very soon!
- `cv2.SimpleBlobDetector_create`
- `cv2.xfeatures2d.HarrisLaplaceFeatureDetector_create`
- `cv2.xfeatures2d.StarDetector_create`
- `cv2.xfeatures2d.TBMR_create`

Details about the parameters and the underlying algorithms can be found in the OpenCV documentation.

Next, we'll examine the FAST corner detector, the BRIEF keypoint descriptor, and ORB (which uses FAST and BRIEF together).

Using ORB with FAST features and BRIEF descriptors

If SIFT is young, and SURF is younger, ORB is in its infancy. ORB was first published in 2011 as a fast alternative to SIFT and SURF.

The algorithm was published in the paper *ORB: an efficient alternative to SIFT or SURF*, available in PDF format at `https://github.com/eYSIP-2016/Object-Tracking-Camera/blob/master/Research%20papers%20referred(Object%20Tracking)/orb_final.pdf`.

ORB mixes the techniques used in the FAST keypoint detector and the BRIEF keypoint descriptor, so it is worth taking a quick look at FAST and BRIEF first. Then, we will talk about brute-force matching, an algorithm used for feature matching, and look at an example of feature matching.

FAST

The **FAST** algorithm works by analyzing circular neighborhoods of 16 pixels. (This 16-pixel radius is fixed, but bear in mind that the algorithm can work at multiple scales using an image pyramid.) FAST marks each pixel in a neighborhood as brighter or darker than a particular threshold, which is defined relative to the center of the circle. A neighborhood is deemed to be a corner if it contains a number of contiguous pixels marked as brighter or darker.

FAST also uses a high-speed test, which can sometimes determine that a neighborhood is not a corner by checking just 2 or 4 pixels instead of 16. To understand how this test works, let's take a look at the following diagram, taken from the OpenCV documentation:

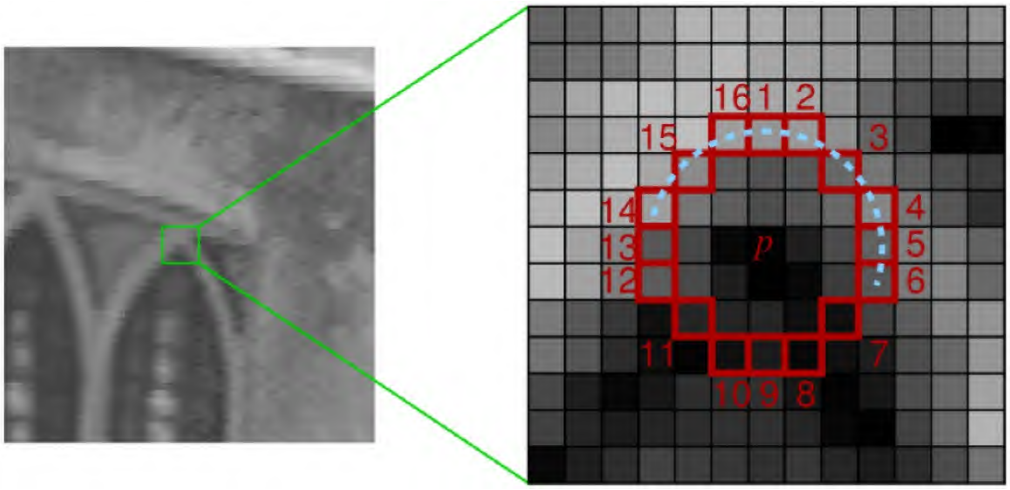


Figure 6.7: How the FAST algorithm analyzes a neighborhood around a corner

Here, we can see a 16-pixel neighborhood at two different magnifications. The pixels at positions 1, 5, 9, and 13 correspond to the four cardinal points at the edge of the circular neighborhood. If the neighborhood is a corner, we expect that out of these four pixels, exactly three *or* exactly one will be brighter than the threshold. (Another way of saying this is that exactly one *or* exactly three of them will be darker than the threshold.) If exactly two of them are brighter than the threshold, then we have an edge, not a corner. If exactly four or exactly zero of them are brighter than the threshold, then we have a relatively uniform neighborhood that is neither a corner nor an edge.

FAST is a clever algorithm, but it's not devoid of weaknesses, and to compensate for these weaknesses, developers analyzing images can implement a machine learning approach in order to feed a set of images (relevant to a given application) to the algorithm so that parameters such as the threshold are optimized. Whether the developer specifies parameters directly or provides a training set for a machine learning approach, FAST is an algorithm that is sensitive to the developer's input, perhaps more so than SIFT.

BRIEF

BRIEF, on the other hand, is not a feature detection algorithm, but a descriptor. Let's delve deeper into the concept of what a descriptor is, and then look at BRIEF.

When we previously analyzed images with SIFT and SURF, the heart of the entire process was the call to the `detectAndCompute` function. This function performs two different steps – detection and computation – and these steps return two different results, coupled in a tuple.

The result of detection is a set of keypoints; the result of the computation is a set of descriptors for those keypoints. This means that OpenCV's `cv2.SIFT` and `cv2.xfeatures2d.SURF` classes implement algorithms for both detection and description. Remember, though, that the original SIFT and SURF are *not* feature detection algorithms. OpenCV's `cv2.SIFT` implements DoG feature detection plus SIFT description, while OpenCV's `cv2.xfeatures2d.SURF` implements Fast Hessian feature detection plus SURF description.

Keypoint descriptors are a representation of the image that serves as the gateway to feature matching because you can compare the keypoint descriptors of two images and find commonalities.

BRIEF is one of the fastest descriptors currently available. The theory behind BRIEF is quite complicated, but suffice it to say that BRIEF adopts a series of optimizations that make it a very good choice for feature matching. As we will discuss in the next section, feature matching can sometimes be an exhaustive process, so choosing an efficient type of descriptor really matters here.

Brute-force matching

A brute-force matcher is a descriptor matcher that compares two sets of keypoint descriptors and generates a result that is a list of matches. It is called brute-force because little optimization is involved in the algorithm. For each keypoint descriptor in the first set, the matcher makes comparisons to every keypoint descriptor in the second set. Each comparison produces a distance value and the best match can be chosen on the basis of the least distance.

More generally, in computing, the term brute-force is associated with an approach that prioritizes the exhaustion of all possible combinations (for example, all the possible combinations of characters to crack a password of a known length). Conversely, an algorithm

that prioritizes speed might skip some possibilities and try to take a shortcut to the solution that seems the most plausible.

OpenCV provides a `cv2.BFMatcher` class that supports several approaches to brute-force feature matching. This matcher, along with OpenCV's implementation of ORB, is integral to our next example.

Matching a logo in two images

Now that we have a general idea of what FAST and BRIEF are, we can understand why the team behind ORB (composed of Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski) chose these two algorithms as a foundation for ORB.

In their paper, the authors set out to achieve the following results:

- The addition of a fast and accurate orientation component to FAST
- The efficient computation of oriented BRIEF features
- Analysis of variance and correlation of oriented BRIEF features
- A learning method to decorrelate BRIEF features under rotational invariance, leading to better performance in nearest-neighbor applications

The main points are quite clear: ORB aims to optimize and speed up operations, including the very important step of utilizing BRIEF in a rotation-aware fashion so that matching is improved, even in situations where a training image has a very different rotation from the query image.

At this stage, though, perhaps you have had enough of the theory and want to sink your teeth into some feature matching, so let's look at some code. The following script attempts to match features in a logo to the features in a photograph that contains the logo:

```
import cv2
from matplotlib import pyplot as plt

# Load the images.
img0 = cv2.imread('../images/nasa_logo.png',
                  cv2.IMREAD_GRAYSCALE)
img1 = cv2.imread('../images/kennedy_space_center.jpg',
                  cv2.IMREAD_GRAYSCALE)

# Perform ORB feature detection and description.
orb = cv2.ORB_create()
kp0, des0 = orb.detectAndCompute(img0, None)
kp1, des1 = orb.detectAndCompute(img1, None)
```

```
# Perform brute-force matching.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des0, des1)

# Sort the matches by distance.
matches = sorted(matches, key=lambda x:x.distance)

# Draw the best 25 matches.
img_matches = cv2.drawMatches(
    img0, kp0, img1, kp1, matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)

# Show the matches.
plt.imshow(img_matches)
plt.show()
```

Let's examine this code step by step. After the usual imports, we load two images (the query image and the scene) in grayscale format. Here is the query image, which is the NASA logo:



Figure 6.8: The NASA logo, to be used for keypoint descriptor matching

Here is the photo of the scene, which is the Kennedy Space Center:



Figure 6.9: The Kennedy Space Center, with the NASA logo on its wall

Now, we proceed to create the ORB feature detector and descriptor:

```
# Perform ORB feature detection and description.
orb = cv2.ORB_create()
kp0, des0 = orb.detectAndCompute(img0, None)
kp1, des1 = orb.detectAndCompute(img1, None)
```

In a similar fashion to what we did with SIFT and SURF, we detect and compute the keypoints and descriptors for both images.

From here, the concept is pretty simple: iterate through the descriptors and determine whether they are a match or not, and then calculate the quality of this match (distance) and sort the matches so that we can display the top *n* matches with a degree of confidence that they are, in fact, matching features on both images. `cv2.BFMatcher` does this for us:

```
# Perform brute-force matching.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des0, des1)

# Sort the matches by distance.
matches = sorted(matches, key=lambda x:x.distance)
```

At this stage, we already have all the information we need, but as computer vision enthusiasts, we place quite a bit of importance on visually representing data, so let's draw these matches in a `matplotlib` chart:

```
# Draw the best 25 matches.
img_matches = cv2.drawMatches(
    img0, kp0, img1, kp1, matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)

# Show the matches.
plt.imshow(img_matches)
plt.show()
```

Tip

Python's slicing syntax is quite robust. If the `matches` list contains fewer than 25 entries, the `matches[:25]` slicing command will run without problems and give us a list with just as many elements as the original.

The result is as follows:

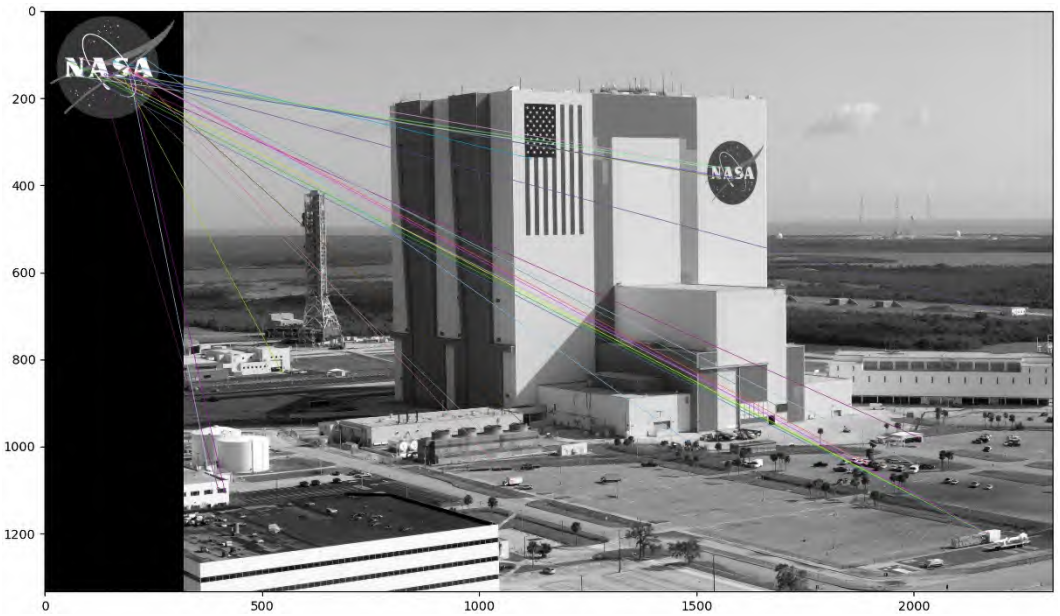


Figure 6.10: Matches between the NASA logo and the Kennedy Space Center, using brute-force matching

You might think that this is a disappointing result. Indeed, we can see that most of the matches are false matches. Unfortunately, this is quite typical. To improve our results, we need to apply additional techniques to filter out bad matches. We'll turn our attention to this task next.

Filtering matches using K-nearest neighbors and the ratio test

Imagine that a large group of renowned philosophers asks you to judge their debate on a question of great importance to life, the universe, and everything. You listen carefully as each philosopher speaks in turn. Finally, when all the philosophers have exhausted all their lines of argument, you review your notes and perceive two things, as follows:

- Every philosopher disagrees with every other
- No one philosopher is much more convincing than the others

From your first observation, you infer that *at most* one of the philosophers is right; however, it is possible that all the philosophers are wrong. Then, from your second observation, you begin to fear that you are at risk of picking a philosopher who is wrong, *even if* one of the philosophers is correct. Any way you look at it, these people have made you late for dinner. You call it a tie and say that the debate's all-important question remains unresolved.

We can compare our imaginary problem of judging the philosophers' debate to our practical problem of filtering out bad keypoint matches.

First, let's assume that each keypoint in our query image has, at most, one correct match in the scene. By implication, if our query image is the NASA logo, we assume that the other image – the scene – contains, at most, one NASA logo. Given that a query keypoint has, at most, one correct or good match, when we consider all *possible* matches, we are primarily observing bad matches.

Thus, a brute-force matcher, which computes a distance score for every possible match, can give us plenty of observations of the distance scores for bad matches. We expect that a good match will have a significantly better (lower) distance score than the numerous bad matches, so the scores for the bad matches can help us pick a threshold for a good match. Such a threshold does not necessarily generalize well across different query keypoints or different scenes, but at least it helps us on a case-by-case basis.

Now, let's consider the implementation of a modified brute-force matching algorithm that adaptively chooses a distance threshold in the manner we have described. In the previous section's code sample, we used the `match` method of the `cv2.BFMatcher` class in order to get a list containing the single best (least-distance) match for each query keypoint. By doing so, we discarded information about the distance scores of all the worse possible matches – the kind of information we need for our adaptive approach. Fortunately, `cv2.BFMatcher` also provides a `knnMatch` method, which accepts an argument, `k`, that specifies the maximum number of best (least-distance) matches that we want to retain for each query keypoint. (In some cases, we may get fewer matches than the maximum.) KNN stands for k-nearest neighbors.

We will use the `knnMatch` method to request a list of the two best matches for each query keypoint. Based on our assumption that each query keypoint has, at most, one correct match, we are confident that the second-best match is wrong. We multiply the second-best match's distance score by a value less than 1 in order to obtain the threshold.

Then, we accept the best match as a good match only if its distance score is less than the threshold. This approach is known as the ratio test, and it was first proposed by David Lowe, the author of the SIFT algorithm. He describes the ratio test in his paper *Distinctive Image Features from Scale-Invariant Keypoints*, which is available at `https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf`. Specifically, in the *Application to object recognition* section, he states the following:

> *"The probability that a match is correct can be determined by taking the ratio of the distance from the closest neighbor to the distance of the second closest."*

We can load the images, detect keypoints, and compute ORB descriptors in the same way as we did in the previous section's code sample. Then, we can perform brute-force KNN matching using the following two lines of code:

```
# Perform brute-force KNN matching.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
pairs_of_matches = bf.knnMatch(des0, des1, k=2)
```

`knnMatch` returns a list of lists; each inner list contains at least one match and no more than `k` matches, sorted from best (least distance) to worst. The outer list's length is the number of keypoints that had at least one match. The following line of code sorts the outer list based on the distance score of the best matches:

```
# Sort the pairs of matches by distance.
pairs_of_matches = sorted(pairs_of_matches, key=lambda x:x[0].distance)
```

Let's draw the top 25 best matches, along with any second-best matches that `knnMatch` may have paired with them. We can't use the `cv2.drawMatches` function because it only accepts a one-dimensional list of matches; instead, we must use `cv2.drawMatchesKnn`. The following code is used to select, draw, and show the matches:

```
# Draw the 25 best pairs of matches.
img_pairs_of_matches = cv2.drawMatchesKnn(
    img0, kp0, img1, kp1, pairs_of_matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS
)

# Show the pairs of matches.
plt.imshow(img_pairs_of_matches)
plt.show()
```

So far, we have not filtered out any bad matches – and, indeed, we have deliberately included the second-best matches, which we believe to be bad – so the result looks a mess. Here it is:

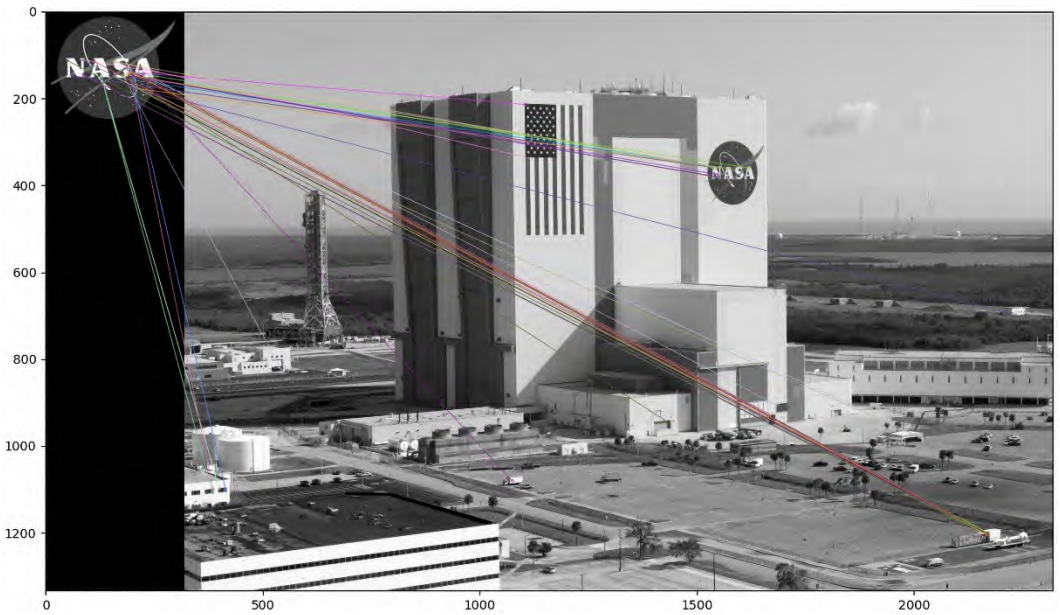


Figure 6.11: Matches between the NASA logo and the Kennedy Space Center, using brute-force KNN matching

Now, let's apply the ratio test. We will set the threshold at 0.8 times the distance score of the second-best match. If `knnMatch` has failed to provide a second-best match, we reject the best match anyway because we are unable to apply the test. The following code applies these conditions and provides us with a list of best matches that passed the test:

```
# Apply the ratio test.
matches = [x[0] for x in pairs_of_matches
           if len(x) > 1 and x[0].distance < 0.8 * x[1].distance]
```

Having applied the ratio test, we are now only dealing with best matches (not pairs of best and second-best matches), so we can draw them with `cv2.drawMatches` instead of `cv2.drawMatchesKnn`. Again, we will select the top 25 matches from the list. The following code is used to select, draw, and show the matches:

```
# Draw the best 25 matches.
img_matches = cv2.drawMatches(
    img0, kp0, img1, kp1, matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
```

```
# Show the matches.
plt.imshow(img_matches)
plt.show()
```

Here, we can see the matches that passed the ratio test:



Figure 6.12: Matches between the NASA logo and the Kennedy Space Center, using brute-force KNN matching and the ratio test

Comparing this output image to the one in the previous section, we can see that KNN and the ratio test have allowed us to filter out many bad matches. The remaining matches are not perfect but nearly all of them point to the correct region – the NASA logo on the side of the Kennedy Space Center.

We have made a promising start. Next, we will replace the brute-force matcher with a faster matcher called FLANN. After that, we will learn how to describe a set of matches in terms of homography, that is, a 2D transformation matrix that expresses the position, rotation, scale, and other geometric characteristics of the matched object.

Matching with FLANN

FLANN stands for Fast Library for Approximate Nearest Neighbors. It is an open-source library under the permissive two-clause BSD license, and it is available on GitHub at `https://github.com/flann-lib/flann`. There, we find the following description:

> *"FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for the nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. FLANN is written in C++ and contains bindings for the following languages: C, MATLAB, Python, and Ruby."*

In other words, FLANN has a big toolbox; it knows how to choose the right tools for the job, and it speaks several languages. These features make the library fast and convenient. Indeed, FLANN's authors claim that it is 10 times faster than other nearest neighbor search software for many datasets.

Although FLANN is also available as a standalone library, we will use it as part of OpenCV because OpenCV conveniently wraps FLANN in a simplified interface.

To begin our practical example of FLANN matching, let's import NumPy, OpenCV, and `Matplotlib`, and load two images from files. Here is the relevant code:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img0 = cv2.imread('../images/gauguin_entre_les_lys.jpg',
                  cv2.IMREAD_GRAYSCALE)
img1 = cv2.imread('../images/gauguin_paintings.png',
                  cv2.IMREAD_GRAYSCALE)
```

Here is the first image, the query image, that our script is loading:



Figure 6.13: Paul Gauguin's painting Entre les lys (Among the lilies), to be used for keypoint descriptor matching

This work of art is *Entre les lys* (*Among the lilies*), painted by Paul Gauguin in 1889. We will search for matching keypoints in a larger image that contains multiple works by Gauguin, alongside some haphazard shapes drawn by one of the authors of this book. Here is the larger image:

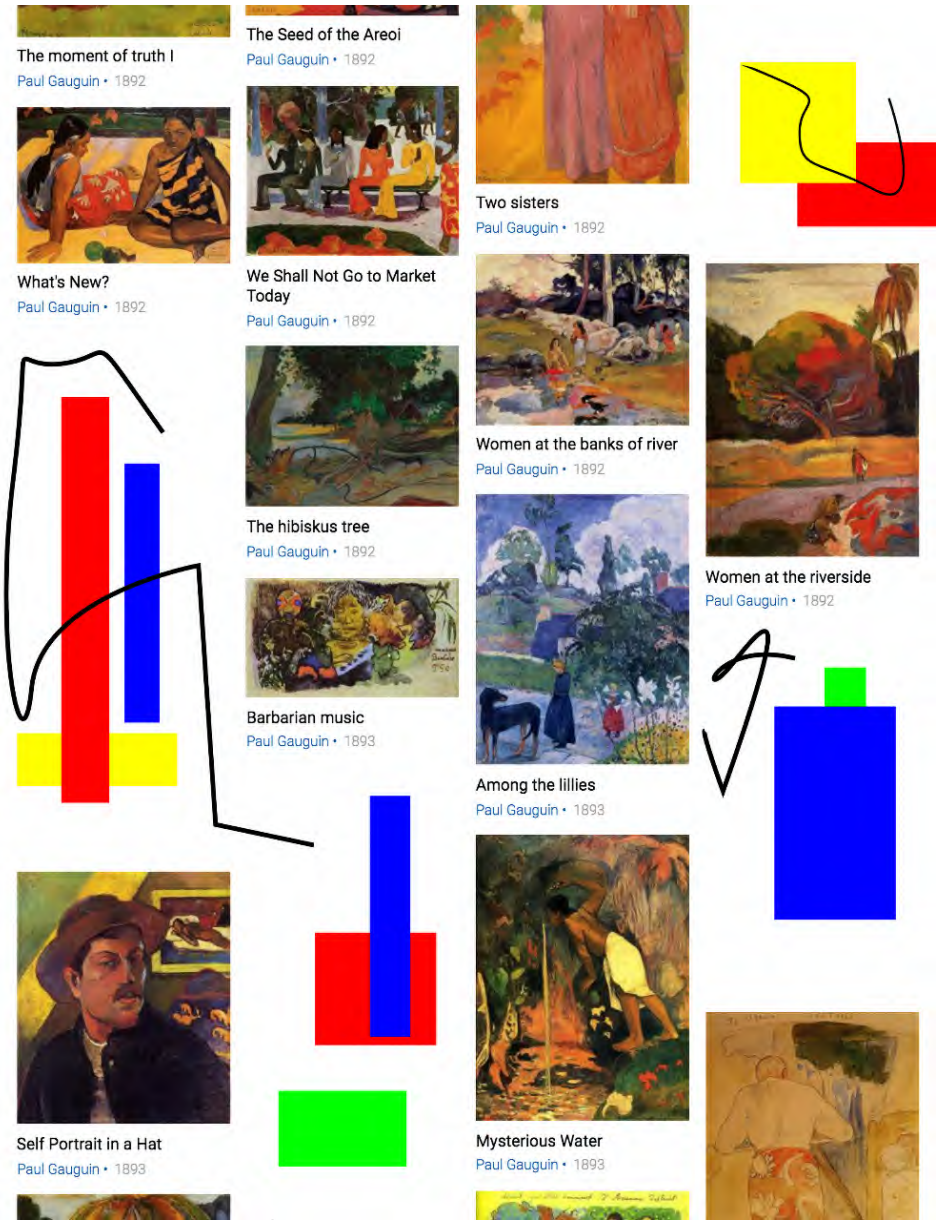


Figure 6.14: Haphazard shapes and various Gauguin paintings, to be used for keypoint descriptor matching

Within the larger image, *Entre les lys* appears in the third column, third row. The query image and the corresponding region of the larger image are not identical; they depict *Entre les lys* in

slightly different colors and at a different scale. Nonetheless, this should be an easy case for our matcher.

Let's detect the necessary keypoints and extract our features using the `cv2.SIFT` class:

```
# Perform SIFT feature detection and description.
sift = cv2.SIFT_create()
kp0, des0 = sift.detectAndCompute(img0, None)
kp1, des1 = sift.detectAndCompute(img1, None)
```

So far, the code should seem familiar, since we have already dedicated several sections of this chapter to SIFT and other descriptors. In our previous examples, we fed the descriptors to `cv2.BFMatcher` for brute-force matching. This time, we will use `cv2.FlannBasedMatcher` instead. The following code performs FLANN-based matching with custom parameters:

```
# Define FLANN-based matching parameters.
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)

# Perform FLANN-based matching.
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des0, des1, k=2)
```

Here, we can see that the FLANN matcher takes two parameters: an `indexParams` object and a `searchParams` object. These parameters, passed in the form of dictionaries in Python (and structs in C++), determine the behavior of the index and search objects that are used internally by FLANN to compute the matches. We have chosen parameters that offer a reasonable balance between accuracy and processing speed. Specifically, we are using a **kernel density tree** (**kd-tree**) indexing algorithm with five trees, which FLANN can process in parallel. (The FLANN documentation recommends between 1 tree, which would offer no parallelism, and 16 trees, which would offer a high degree of parallelism if the system could exploit it.)

We are performing 50 checks or traversals of each tree. A greater number of checks can provide greater accuracy but at a greater computational cost.

After performing FLANN-based matching, we apply Lowe's ratio test with a multiplier of 0.7. To demonstrate a different coding style, we will use the result of the ratio test in a slightly different way compared to how we did in the previous section's code sample. Previously, we assembled a new list with just the good matches in it. This time, we will assemble a list called `mask_matches`, in which each element is a sublist of length `k` (the same `k` that we passed to

`knnMatch`). If a match is good, we set the corresponding element of the sublist to `1`; otherwise, we set it to `0`.

For example, if we have `mask_matches = [[0, 0], [1, 0]]`, this means that we have two matched keypoints; for the first keypoint, the best and second-best matches are both bad, while for the second keypoint, the best match is good but the second-best match is bad. Remember, we assume that all the second-best matches are bad. We use the following code to apply the ratio test and build the mask:

```
# Prepare an empty mask to draw good matches.
mask_matches = [[0, 0] for i in range(len(matches))]

# Populate the mask based on David G. Lowe's ratio test.
for i, (m, n) in enumerate(matches):
    if m.distance < 0.7 * n.distance:
        mask_matches[i]=[1, 0]
```

Now, it is time to draw and show the good matches. We can pass our `mask_matches` list to `cv2.drawMatchesKnn` as an optional argument, as `highlighted` in the following code segment:

```
# Draw the matches that passed the ratio test.
img_matches = cv2.drawMatchesKnn(
    img0, kp0, img1, kp1, matches, None,
    matchColor=(0, 255, 0), singlePointColor=(255, 0, 0),
    matchesMask=mask_matches, flags=0)

# Show the matches.
plt.imshow(img_matches)
plt.show()
```

`cv2.drawMatchesKnn` only draws the matches that we marked as good (with a value of `1`) in our mask. Let's unveil the result. Our script produces the following visualization of the FLANN-based matches:

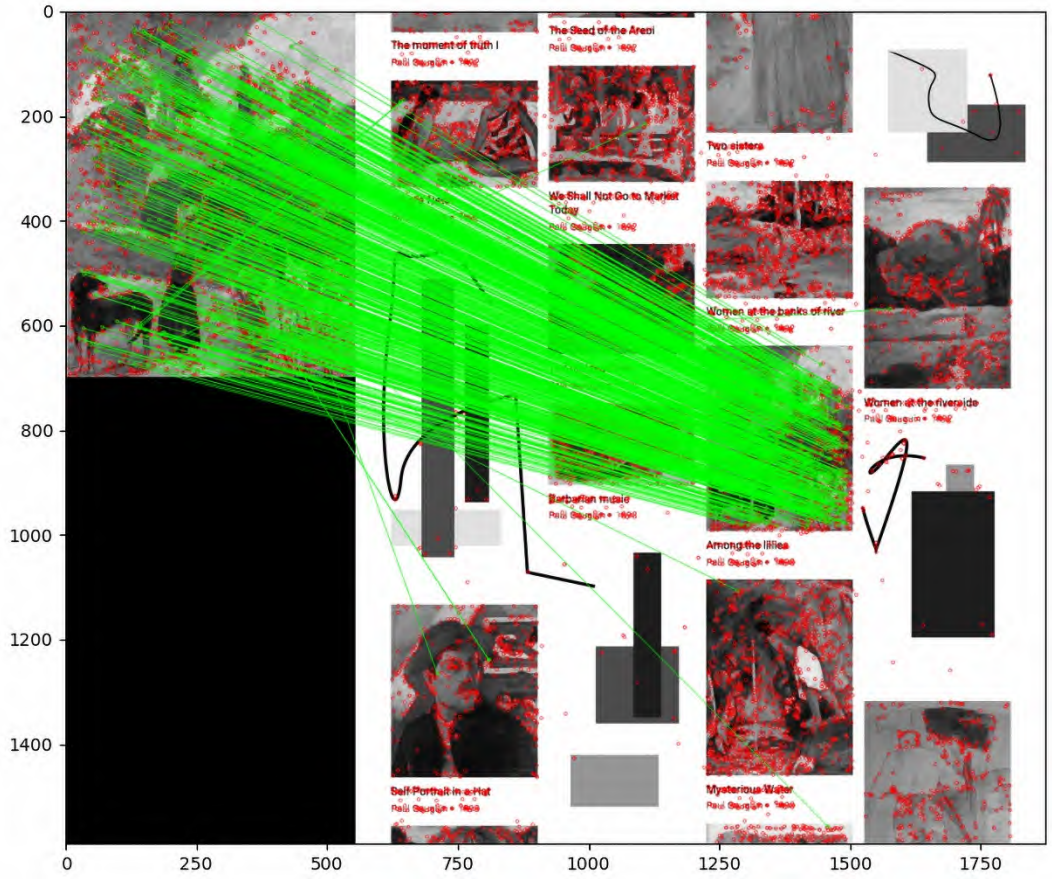


Figure 6.15: FLANN-based matching with Gauguin paintings

This is an encouraging picture: it appears that nearly all the matches fall in the right places. Next, let's try to reduce this type of result to a more succinct geometric representation – a homography – which would describe the pose of a whole matched object rather than a bunch of disconnected matched points.

Finding homography with FLANN-based matches

First of all, what is homography? Let's read a definition from the 1913 Webster's Dictionary:

> *"A relation between two figures, such that to any point of the one corresponds one and but one point in the other, and vice versa. Thus, a tangent line rolling on a circle cuts two fixed tangents of the circle in two sets of points that are homographic."*

If you, like the authors of this book, are none the wiser from the preceding definition, you will probably find the following explanation a bit clearer: homography is a condition in which two figures fit each other when one is a perspective distortion of the other.

First, let's take a look at what we want to achieve so that we can fully understand what homography is. Then, we will go through the code.

Imagine that we want to search for the following tattoo:

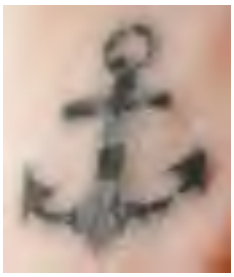


Figure 6.16: An anchor tattoo, to be used for keypoint matching and finding homography

We, as human beings, can easily locate the tattoo in the following image, despite there being a difference in rotation:



Figure 6.17: Fingers, with an anchor tattoo, to be used for keypoint matching and finding homography

As an exercise in computer vision, we want to write a script that produces the following visualization of keypoint matches and the homography:

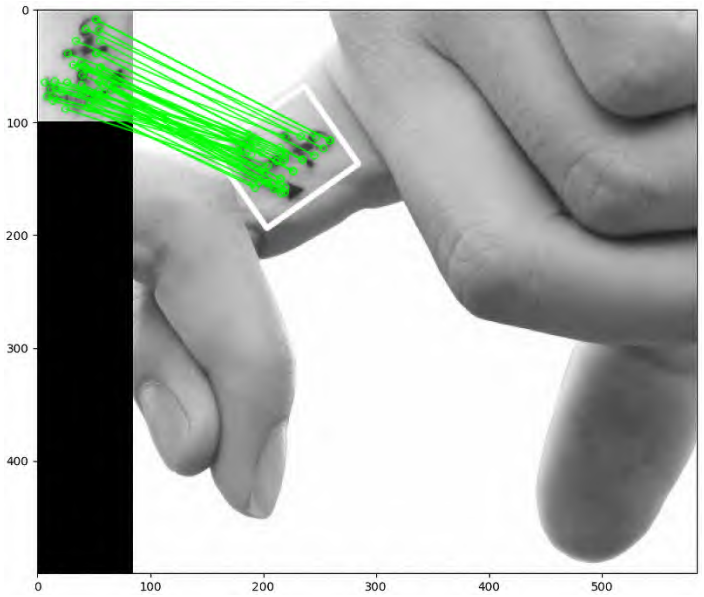


Figure 6.18: Keypoint matches and homography between two images of an anchor tattoo

As shown in the preceding screenshot, we took the subject in the first image, correctly identified it in the second image, drew matching lines between the keypoints, and even drew a white border showing the perspective deformation of the subject in the second image relative to the first image.

You might have guessed – correctly – that the script's implementation starts by importing libraries, reading images in grayscale format, detecting features, and computing SIFT descriptors. We did all of this in our previous examples, so we will omit that here. Let's take a look at what we do next:

1. We proceed by assembling a list of matches that pass Lowe's ratio test, as shown in the following code:

```
# Find all the good matches as per Lowe's ratio test.
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)
```

2. Technically, we can calculate the homography with as few as four matches. However, if any of these four matches is flawed, it will throw off the accuracy of the result. A more practical minimum is `10`. Given the extra matches, the homography-finding algorithm can discard some outliers in order to produce a result that closely fits a substantial subset of the matches. Thus, we proceed to check whether we have at least `10` good matches:

```
MIN_NUM_GOOD_MATCHES = 10

if len(good_matches) >= MIN_NUM_GOOD_MATCHES:
```

3. If this condition has been satisfied, we look up the 2D coordinates of the matched keypoints and place these coordinates in two lists of floating-point coordinate pairs. One list contains the keypoint coordinates in the query image, while the other list contains the matching keypoint coordinates in the scene:

```
    src_pts = np.float32(
        [kp0[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
    dst_pts = np.float32(
        [kp1[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
```

4. Now, we find the homography:

```
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    mask_matches = mask.ravel().tolist()
```

   Note that we create a `mask_matches` list, which will be used in the final drawing of the matches so that only points lying within the homography will have matching lines drawn.

5. At this stage, we have to perform a perspective transformation, which takes the rectangular corners of the query image and projects them into the scene so that we can draw the border:

```
    h, w = img0.shape
    src_corners = np.float32(
        [[0, 0], [0, h-1], [w-1, h-1], [w-1, 0]]).reshape(-1, 1, 2)
    dst_corners = cv2.perspectiveTransform(src_corners, M)
    dst_corners = dst_corners.astype(np.int32)
```

```
    # Draw the bounds of the matched region based on the homography.
    num_corners = len(dst_corners)
    for i in range(num_corners):
        x0, y0 = dst_corners[i][0]
        if i == num_corners - 1:
            next_i = 0
        else:
            next_i = i + 1
        x1, y1 = dst_corners[next_i][0]
        cv2.line(img1, (x0, y0), (x1, y1), 255, 3, cv2.LINE_AA)
```

Then, we proceed to draw the keypoints and show the visualization, as per our previous examples.

A sample application – tattoo forensics

Let's conclude this chapter with a real-life (or perhaps fantasy-life) example. Imagine you are working for the Gotham forensics department and you need to identify a tattoo. You have the original picture of a criminal's tattoo (perhaps captured in CCTV footage), but you don't know the identity of the person. However, you possess a database of tattoos, indexed with the name of the person that the tattoo belongs to.

Let's divide this task into two parts:

- Build a database by saving image descriptors to files
- Load the database and scan for matches between a query image's descriptors and the descriptors in the database

We will cover these tasks in the next two subsections.

Saving image descriptors to a file

The first thing we will do is save the image descriptors to an external file. This way, we don't have to recreate the descriptors every time we want to scan two images for matches.

For the purposes of our example, let's scan a folder for images and create the corresponding descriptor files so that we have them readily available for future searches. To create descriptors, we will use a process we have already used a number of times in this chapter: namely, load an image, create a feature detector, detect features, and compute descriptors. To save the descriptors to a file, we will use a handy method of NumPy arrays called `save`, which dumps array data into a file in an optimized way.

Tip

The `pickle` module, in the Python standard library, provides more general-purpose serialization functionality that supports any Python object and not just NumPy arrays. However, NumPy's array serialization is a good choice for numeric data.

Let's break our script up into functions. The main function will be named `create_descriptors` (plural, `descriptors`), and it will iterate over the files in a given folder. For each file, `create_descriptors` will call a helper function named `create_descriptor` (singular, `descriptor`), which will compute and save our descriptors for the given image file. Let's get started:

1. First, here is the implementation of `create_descriptors`:

```
import os

import numpy as np
import cv2

def create_descriptors(folder):
    feature_detector = cv2.SIFT_create()
    files = []
    for (dirpath, dirnames, filenames) in os.walk(folder):
        files.extend(filenames)
    for f in files:
        create_descriptor(folder, f, feature_detector)
```

   Note that `create_descriptors` creates the feature detector because we only need to do this once, not every time we load a file. The helper function, `create_descriptor`, receives the feature detector as an argument.

2. Now, let's look at the latter function's implementation:

```
def create_descriptor(folder, image_path, feature_detector):
    if not image_path.endswith('png'):
        print('skipping %s' % image_path)
        return
    print('reading %s' % image_path)
    img = cv2.imread(os.path.join(folder, image_path),
                     cv2.IMREAD_GRAYSCALE)
```

```
        keypoints, descriptors = feature_detector.detectAndCompute(
            img, None)
        descriptor_file = image_path.replace('png', 'npy')
        np.save(os.path.join(folder, descriptor_file), descriptors)
```

Note that we save the descriptor files in the same folder as the images. Moreover, we assume that the image files have the `png` extension. To make the script more robust, you could modify it so that it supports additional image file extensions such as `jpg`. If a file has an unexpected extension, we skip it because it might be a descriptor file (from a previous run of the script) or some other non-image file.

3. We have finished implementing the functions. To complete the script, we will call `create_descriptors` with a folder name as an argument:

```
folder = '../images/tattoos'
create_descriptors(folder)
```

When we run this script, it produces the necessary descriptor files in NumPy's array file format, with the file extension `npy`. These files constitute our database of tattoo descriptors, indexed by name. (Each filename is a person's name.) Next, we'll write a separate script so that we can run a query against this database.

Scanning for matches

Now that we have descriptors saved to files, we just need to perform matching against each set of descriptors to determine which set best matches our query image.

This is the process we will put in place:

1. Load a query image (`query.png`).
2. Scan the folder containing descriptor files. Print the names of the descriptor files.
3. Create SIFT descriptors for the query image.
4. For each descriptor file, load the SIFT descriptors and find FLANN-based matches. Filter the matches based on the ratio test. Print the person's name and the number of matches. If the number of matches exceeds an arbitrary threshold, print that the person is a suspect. (Remember, we are investigating a crime.)
5. Print the name of the prime suspect (the one with the most matches).

Let's consider the implementation:

1. First, the following code block loads the query image:

```
import os

import numpy as np
import cv2

# Read the query image.
folder = '../images/tattoos'
query = cv2.imread(os.path.join(folder, 'query.png'),
                   cv2.IMREAD_GRAYSCALE)
```

2. We proceed to assemble and print a list of the descriptor files:

```
# create files, images, descriptors globals
files = []
images = []
descriptors = []
for (dirpath, dirnames, filenames) in os.walk(folder):
    files.extend(filenames)
    for f in files:
        if f.endswith('npy') and f != 'query.npy':
            descriptors.append(f)
print(descriptors)
```

3. We set up our typical `cv2.SIFT` and `cv2.FlannBasedMatcher` objects, and we generate descriptors of the query image:

```
# Create the SIFT detector.
sift = cv2.SIFT_create()

# Perform SIFT feature detection and description on the
# query image.
query_kp, query_ds = sift.detectAndCompute(query, None)

# Define FLANN-based matching parameters.
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
```

```
# Create the FLANN matcher.
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

4. Now, we search for suspects, whom we define as people with at least 10 good matches for the query tattoo. Our search entails iterating over the descriptor files, loading the descriptors, performing FLANN-based matching, and filtering the matches based on the ratio test. We print a result for each person (each descriptor file):

```
# Define the minimum number of good matches for a suspect.
MIN_NUM_GOOD_MATCHES = 10

greatest_num_good_matches = 0
prime_suspect = None

print('>> Initiating picture scan...')
for d in descriptors:
    print('--------- analyzing %s for matches ------------' % d)
    matches = flann.knnMatch(
        query_ds, np.load(os.path.join(folder, d)), k=2)
    good_matches = []

    for m, n in matches:
        if m.distance < 0.7 * n.distance:
            good_matches.append(m)
    num_good_matches = len(good_matches)
    name = d.replace('.npy', '').upper()

    if num_good_matches >= MIN_NUM_GOOD_MATCHES:
        print('%s is a suspect! (%d matches)' % \
            (name, num_good_matches))
        if num_good_matches > greatest_num_good_matches:
            greatest_num_good_matches = num_good_matches
            prime_suspect = name
    else:
        print('%s is NOT a suspect. (%d matches)' % \
            (name, num_good_matches))
```

> **Note**
>
> Note the use of the `np.load` method, which loads a specified `npy` file into a NumPy array.

5. In the end, we print the name of the prime suspect (if we found a suspect, that is):

```
if prime_suspect is not None:
    print('Prime suspect is %s.' % prime_suspect)
else:
    print('There is no suspect.')
```

Running the preceding script produces the following output:

```
>> Initiating picture scan...
--------- analyzing anchor-woman.npy for matches ------------
ANCHOR-WOMAN is NOT a suspect. (2 matches)
--------- analyzing anchor-man.npy for matches ------------
ANCHOR-MAN is a suspect! (44 matches)
--------- analyzing lady-featherly.npy for matches ------------
LADY-FEATHERLY is NOT a suspect. (2 matches)
--------- analyzing steel-arm.npy for matches ------------
STEEL-ARM is NOT a suspect. (0 matches)
--------- analyzing circus-woman.npy for matches ------------
CIRCUS-WOMAN is NOT a suspect. (1 matches)
Prime suspect is ANCHOR-MAN.
```

Thus, the script has found that `ANCHOR-MAN` is the prime suspect. If, instead, the script had not found any good match, it would have finished by printing `There is no suspect.`

If we wanted, we could represent the matches and the homography graphically, as we did in the previous section.

Summary

In this chapter, we learned about detecting keypoints, computing keypoint descriptors, matching these descriptors, filtering out bad matches, and finding the homography between two sets of matching keypoints. We explored a number of algorithms that are available in OpenCV that can be used to accomplish these tasks, and we applied these algorithms to a variety of images and use cases. We saw that many parameters and thresholds are relevant to

these algorithms, and it is important to note that in any real-world application, a lot of experimentation is needed to determine appropriate parameter and threshold values for the particular use case.

If we combine our new knowledge of keypoints with additional knowledge about cameras and perspective, we can track objects in 3D space. This will be the topic of *Chapter 9, Camera Models and Augmented Reality*. You can skip ahead to that chapter if you are particularly keen to learn about the third dimension.

If, instead, you think the next logical step is to round off your knowledge of two-dimensional solutions for object detection, recognition, and tracking, you can continue sequentially with *Chapter 7, Building Custom Object Detectors*, and then *Chapter 8, Tracking Objects*. It is good to know a combination of 2D and 3D techniques so that you can choose an approach that offers the right kind of output and the right computational speed for a given application.

7

Building Custom Object Detectors

This chapter delves deeper into the concept of object detection, which is one of the most common challenges in computer vision. Having come this far in the book, you are perhaps wondering when you will be able to put computer vision into practice in the real world. Do you dream of building a system to detect cars and people? Well, you are not too far from your goal, actually.

We have already looked at some specific cases of object detection and recognition in previous chapters. We focused on upright, frontal human faces in *Chapter 5, Detecting and Recognizing Faces*, and on objects with corner-like or blob-like features in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. Now, in the current chapter, we will explore algorithms that have a good ability to generalize or extrapolate, in the sense that they can cope with the real-world diversity that exists within a given class of object. For example, different cars have different designs, and people can appear to be different shapes depending on the clothes they wear.

Specifically, we will pursue the following objectives:

- Learning about another kind of feature descriptor: the **histogram of oriented gradients** (**HOG**) descriptor.
- Understanding **non-maximum suppression**, also called **non-maxima suppression** (**NMS**), which helps us choose the best from an overlapping set of detection windows.
- Gaining a high-level understanding of **support vector machines** (**SVMs**). These general-purpose classifiers are based on supervised machine learning, in a way that is similar to linear regression.
- Detecting people with a pre-trained classifier based on HOG descriptors.

- Training a **bag-of-words** (**BoW**) classifier to detect a car. For this sample, we will work with a custom implementation of an image pyramid, a sliding window, and NMS so that we can better understand the inner workings of these techniques.
- Training a HOG classifier to detect a car.

Most of the techniques in this chapter are not mutually exclusive; rather, they work together as components of a detector. By the end of the chapter, you will know how to train and use classifiers that have practical applications on the streets!

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Please refer back to *Chapter 1, Setting Up OpenCV*, for installation instructions.

The complete code for this chapter can be found in this book's GitHub repository in the `chapter07` folder. The sample images are also in this book's GitHub repository, in the `images` folder.

You can download the example project and code for this book by following the instructions in the ***Download the example code files*** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

A subset of the chapter's sample code can be edited and run interactively in Google Colab.

Understanding HOG descriptors

HOG is a feature descriptor, so it belongs to the same family of algorithms as **scale-invariant feature transform** (**SIFT**), **speeded-up robust features** (**SURF**), and **Oriented FAST and rotated BRIEF** (**ORB**), which we covered in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. Like other feature descriptors, HOG is capable of delivering the type of information that is vital for feature matching, as well as for object detection and recognition. Most commonly, HOG is used for object detection. The algorithm – and, in particular, its use as a person detector – was popularized by Navneet Dalal and Bill Triggs in their paper *Histograms of Oriented Gradients for Human Detection* (INRIA, 2005), which is available online at `https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf`.

HOG's internal mechanism is really clever; an image is divided into cells and a set of gradients is calculated for each cell. Each gradient describes the change in pixel intensities in a given direction. Together, these gradients form a histogram representation of the cell. We encountered a similar approach when we studied face recognition with the **local binary pattern histogram** (**LBPH**) in *Chapter 5, Detecting and Recognizing Faces*.

Before diving into the technical details of how HOG works, let's first take a look at how HOG *sees* the world.

Visualizing HOG

Carl Vondrick, Aditya Khosla, Hamed Pirsiavash, Tomasz Malisiewicz, and Antonio Torralba have developed a HOG visualization technique called HOGgles (HOG goggles). For a summary of HOGgles, as well as links to code and publications, refer to Carl Vondrick's MIT web page at `http://www.cs.columbia.edu/~vondrick/ihog/index.html`. As one of their test images, Vondrick et al. use the following photograph of a truck:



Figure 7.1: A truck, to be used as a test image for HOG visualization

Vondrick et al. produce the following visualization of the HOG descriptors, based on an approach from Dalal and Triggs' earlier paper:

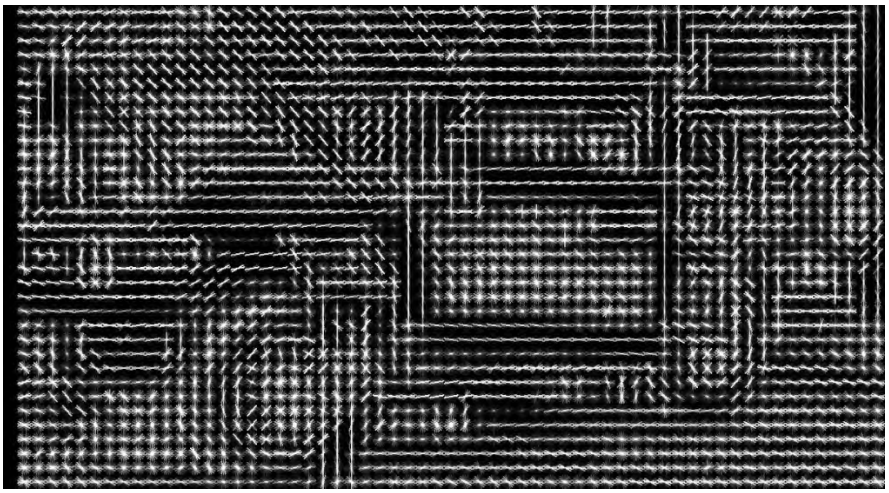


Figure 7.2: HOG cell gradients, visualized as lines

Then, applying HOGgles, Vondrick et al. invert the feature description algorithm to reconstruct the image of the truck as HOG sees it, as shown here:



Figure 7.3: HOG cell gradients, visualized as smooth grayscale transitions, based on the HOGgles technique

In both of these visualizations, you can see that HOG has divided the image into cells, and you can easily recognize the wheels and the main structure of the vehicle. In the first visualization, the calculated gradients for each cell are visualized as a set of crisscrossing lines that sometimes look like an elongated star; the star's longer axes represent stronger gradients. In the second visualization, using HOGgles, the gradients are visualized as a smooth transition in brightness, along various axes in a cell.

The importance of HOGgles is that it may help a developer understand cases where a HOG model misclassified part of an image. When we look closely at a HOGgles visualization, we may be able to see cloudy shapes that resemble objects of the positive class, even if no such object was really there. Then, we can consider how to retrain the model with relevant negative samples. For example, take a look at the paper by Vondrick et al.

Now, let's give further consideration to the way HOG works, and the way it can contribute to an object detection solution.

Using HOG to describe regions of an image

For each HOG cell, the histogram contains a number of bins equal to the number of gradients or, in other words, the number of axis directions that HOG considers. After calculating all the cells' histograms, HOG processes groups of histograms to produce higher-level descriptors. Specifically, the cells are grouped into larger regions, called blocks. These blocks can be made of any number of cells, but Dalal and Triggs found that 2x2 cell blocks yielded the best results when performing people detection. A block-wide vector is created so that it can be normalized, compensating for local variations in illumination and shadowing. (A single cell is too small a region to detect such variations.) This normalization improves a HOG-based detector's robustness, with respect to variations in lighting conditions.

Like other detectors, a HOG-based detector needs to cope with variations in objects' location and scale. The need to search in various locations is addressed by moving a fixed-size sliding window across an image. The need to search at various scales is addressed by scaling the image to various sizes, forming a so-called image pyramid. **We studied these techniques previously in** *Chapter 5, Detecting and Recognizing Faces***, specifically in the** *Conceptualizing Haar cascades* **section. However, let's elaborate on one difficulty: how to handle multiple detections in overlapping windows.**

Suppose we are using a sliding window to perform people detection on an image. We slide our window in small steps, just a few pixels at a time, so we expect that it will frame any given person multiple times. Assuming that overlapping detections are indeed one person, we do not want to report multiple locations but, rather, only one location that we believe to be correct. In other words, even if a detection at a given location has a *good* confidence score, we might reject

it if an overlapping detection has a *better* confidence score; thus, from a set of overlapping detections, we would choose the one with the *best* confidence score.

This is where NMS comes into play. Given a set of overlapping regions, we can suppress (or reject) all the regions for which our classifier did not produce a maximum score.

Understanding NMS

The concept of NMS might sound simple. From a set of overlapping solutions, just pick the best one! However, the implementation is more complex than you might initially think. Remember the image pyramid? Overlapping detections can occur at different scales. We must gather up all our positive detections and convert their bounds back to a common scale before we check for overlap. A typical implementation of NMS takes the following approach:

1. Construct an image pyramid.
2. Scan each level of the pyramid with the sliding window approach, for object detection. For each window that yields a positive detection (beyond a certain arbitrary confidence threshold), convert the window back to the original image's scale. Add the window and its confidence score to a list of positive detections.
3. Sort the list of positive detections by order of descending confidence score so that the best detections come first in the list.
4. For each window, *W*, in the list of positive detections, remove all subsequent windows that significantly overlap with *W*. We are left with a list of positive detections that satisfy the criterion of NMS.

Note

Besides NMS, another way to filter the positive detections is to eliminate any subwindows. When we speak of a **subwindow** (or subregion), we mean a window (or region in an image) that is entirely contained inside another window (or region). To check for subwindows, we simply need to compare the corner coordinates of various window rectangles. We will take this simple approach in our first practical example, in the *Detecting people with HOG descriptors* section. Optionally, NMS and suppression of subwindows can be combined.

Several of these steps are iterative, so we have an interesting optimization problem on our hands. A fast sample implementation in MATLAB is provided by Tomasz Malisiewicz at `http://www.computervisionblog.com/2011/08/blazing-fast-nmsm-from-exemplar-svm.html`. A port of this sample implementation to Python is provided by Adrian Rosebrock at

`https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/`. We will build atop the latter sample later in this chapter, in the *Detecting a car in a scene* section.

Now, how do we determine the confidence score for a window? We need a classification system that determines whether a certain feature is present or not, and a confidence score for this classification. This is where **SVMs** come into play.

Understanding SVMs

Without going into details of how an SVM works, let's just try to grasp what it can help us accomplish in the context of machine learning and computer vision. Given labeled training data, an SVM learns to classify the same kind of data by finding an optimal *hyperplane*, which, in plain English, is the plane that divides differently labeled data by the largest possible margin. To aid our understanding, let's consider the following diagram, which is provided by Zach Weinberg under the Creative Commons Attribution-Share Alike 3.0 Unported License:

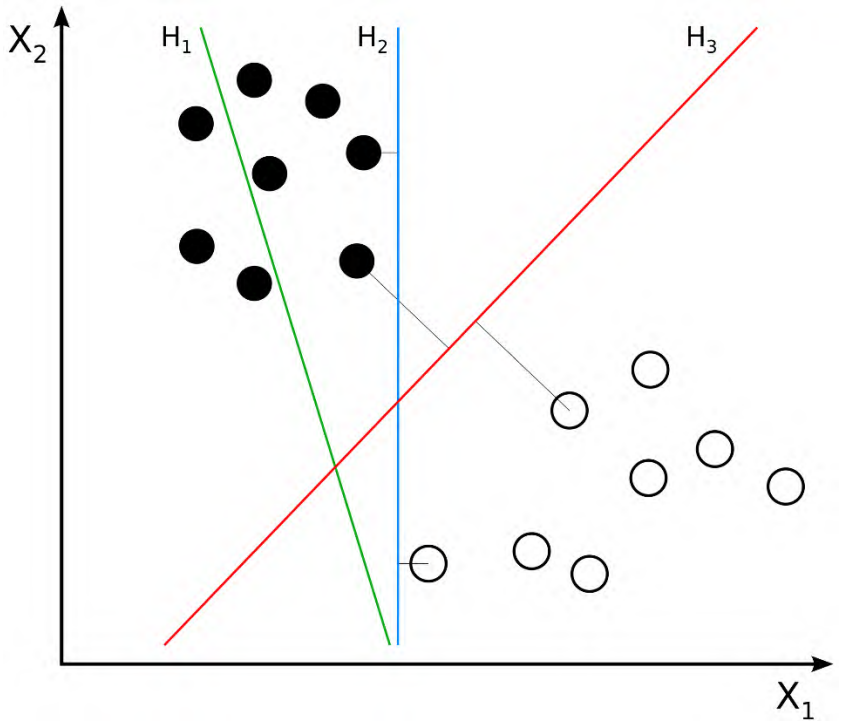


Figure 7.4: Clusters and hyperplanes. An SVM would identify $\mathbf{H}_3$ as the optimal hyperplane

Hyperplane $\mathbf{H}_1$ (shown as a green line) does not divide the two classes (the black dots versus the white dots). Hyperplanes $\mathbf{H}_2$ (shown as a blue line) and $\mathbf{H}_3$ (shown as a red line) both divide the classes; however, only hyperplane $\mathbf{H}_3$ divides the classes by a maximal margin.

Let's suppose we are training an SVM as a person detector. We have two classes, *person* and *non-person*. As training samples, we provide vectors of HOG descriptors of various windows that do or do not contain a person. These windows may come from various images. The SVM learns by finding the optimal hyperplane that maximally divides the multidimensional HOG descriptor space into people (on one side of the hyperplane) and non-people (on the other side). Thereafter, when we give the trained SVM a vector of HOG descriptors for any other window in any image, the SVM can judge whether the window contains a person or not. The SVM can even give us a confidence value that relates to the vector's distance from the optimal hyperplane.

The SVM model has been around since the early 1960s. However, it has undergone improvements since then, and the basis of modern SVM implementations can be found in the paper *Support-vector networks* (*Machine Learning*, 1995) by Corinna Cortes and Vladimir Vapnik. It is available at `http://link.springer.com/article/10.1007/BF00994018`.

Now that we have a conceptual understanding of the key components we can combine to make an object detector, we can start looking at a few examples. We will start with one of OpenCV's ready-made object detectors, and then we will progress to designing and training our own custom object detectors.

Detecting people with HOG descriptors

OpenCV comes with a class called `cv2.HOGDescriptor`, which is capable of performing people detection. The interface has some similarities to the `cv2.CascadeClassifier` class that we used in *Chapter 5, Detecting and Recognizing Faces*. However, unlike `cv2.CascadeClassifier`, the `cv2.HOGDescriptor` class sometimes returns nested detection rectangles. In other words, `cv2.HOGDescriptor` might tell us that it detected one person whose bounding rectangle is located completely inside another person's bounding rectangle. This situation really is possible; for example, a child could be standing in front of an adult, and the child's bounding rectangle could be completely inside the adult's bounding rectangle. However, in a typical situation, nested detections are probably errors, so `cv2.HOGDescriptor` is often used along with code to filter out any nested detections.

Let's begin our sample script by implementing a test to determine whether one rectangle is nested inside another. For this purpose, we will write a function, `is_inside(i, o)`, where `i` is the possible inner rectangle and `o` is the possible outer rectangle. The function will return `True` if `i` is inside `o`; otherwise, it will return `False`. Here is the start of the script:

```
import cv2

def is_inside(i, o):
    ix, iy, iw, ih = i
    ox, oy, ow, oh = o
    return ix > ox and ix + iw < ox + ow and \
        iy > oy and iy + ih < oy + oh
```

Now, we create an instance of `cv2.HOGDescriptor`, and we specify that it will use a default person detector that is built into OpenCV by running the following code:

```
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
```

Note that we specified the person detector with the `setSVMDetector` method. Hopefully, this makes sense based on the previous section of this chapter; an SVM is a classifier, so the choice of SVM determines the type of object our `cv2.HOGDescriptor` will detect.

Now, we proceed to load an image – in this case, an old photograph of women working in a hayfield – and we attempt to detect people in the image by running the following code:

```
img = cv2.imread('../images/haying.jpg')

found_rects, found_weights = hog.detectMultiScale(
    img, winStride=(4, 4), scale=1.02, groupThreshold=1.9)
```

Note that `cv2.HOGDescriptor` has a `detectMultiScale` method, which returns two lists:

1. A list of bounding rectangles for detected objects (in this case, detected people).
2. A list of weights or confidence scores for detected objects. A higher value indicates greater confidence that the detection result is correct.

`detectMultiScale` accepts several optional arguments, including the following:

- `winStride`: This tuple defines the *x* and *y* distance that the sliding window moves between successive detection attempts. HOG works well with overlapping windows, so the stride may be small relative to the window size. A smaller value produces more detections, at a higher computational cost. The default stride has no overlap; it is the same as the window size, which is `(64, 128)` for the default person detector.
- `scale`: This `scale` factor is applied between successive levels of the image pyramid. A smaller value produces more detections, at a higher computational cost. The value must be greater than `1.0`. The default is `1.5`.
- `groupThreshold`: This value determines how stringent our detection criteria are. A smaller value is less stringent, resulting in more detections. The default is `2.0`.

Tip

Prior to OpenCV 5, the `groupThreshold` parameter used to be named `finalThreshold`.

Now, we can filter the detection results to remove nested rectangles. To determine whether a rectangle is a nested rectangle, we potentially need to compare it to every other rectangle. Note the use of our `is_inside` function in the following nested loop:

```
found_rects_filtered = []
found_weights_filtered = []
for ri, r in enumerate(found_rects):
    for qi, q in enumerate(found_rects):
        if ri != qi and is_inside(r, q):
            break
    else:
        found_rects_filtered.append(r)
        found_weights_filtered.append(found_weights[ri])
```

Finally, let's draw the remaining rectangles and weights in order to highlight the detected people, and let's show and save this visualization, as follows:

```
for ri, r in enumerate(found_rects_filtered):
    x, y, w, h = r
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 255), 2)
    text = '%.2f' % found_weights_filtered[ri]
```

```
    cv2.putText(img, text, (x, y - 20),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)

cv2.imshow('Women in Hayfield Detected', img)
cv2.imwrite('./women_in_hayfield_detected.jpg', img)
cv2.waitKey(0)
```

If you run the script yourself, you will see rectangles around people in the image. Here is the result:



Figure 7.5: People detected using OpenCV's default HOG person detector

Note

The photograph is another example of the work of Sergey Prokudin-Gorsky (1863–1944), a pioneer of color photography. Here, the scene is a field at the Leushinskii Monastery, in northwestern Russia, in 1909.

Of the six women who are nearest to the camera, five have been successfully detected. At the same time, a tower in the background has been falsely detected as a person.

The default person detector, with its window size of `(64, 128)`, is ineffective at detecting the women who are distant from the camera. We can address this problem by choosing a different HOG person detector that is also built into OpenCV. This is simply a matter of changing the way we initialize our `HOGDescriptor`, as `highlighted` in the following code:

```
hog = cv2.HOGDescriptor((48, 96), (16, 16), (8, 8), (8, 8), 9)
hog.setSVMDetector(cv2.HOGDescriptor_getDaimlerPeopleDetector())
```

Here, we have loaded the Daimler person detector, which has a window size of `(48, 96)`. We have specified various numeric arguments relating to the dimensions and other training parameters of this model. If we did not specify these arguments, OpenCV would complain because it would expect a model with dimensions and training parameters matching the default person detector.

The Daimler person detector is more sensitive than the default person detector so, to reduce the false positive rate, let's adjust the detection parameters as `highlighted` below:

```
found_rects, found_weights = hog.detectMultiScale(
    img, winStride=(8, 8), scale=1.04, groupThreshold=6.0)
```

With this configuration, the Daimler model gives us slightly better detection results than the default model. The Daimler model's detection windows are more tightly fitted to the people, and one of the women in the distance is successfully detected:



Figure 7.5.2: People detected using OpenCV's Daimler HOG person detector. (Image credit: Prokudin-Gorsky)

In many real-world applications, people-detection results can be improved by analyzing a series of frames in a video. For example, imagine that we are looking at a surveillance video of the Leushinskii Monastery's hayfield, instead of a single photograph. We should be able to add code to determine that the tower cannot be a person because it does not move. Also, we should be able to detect additional people in other frames and track each person's movement from frame to frame. We will look at a people-tracking problem in *Chapter 8, Tracking Objects*.

Meanwhile, let's proceed to look at another kind of detector that we can train to detect a given class of objects.

Creating and training an object detector

Using a pre-trained detector makes it easy to build a quick prototype, and we are all very grateful to the OpenCV developers for making such useful capabilities as face detection and person detection readily available. However, whether you are a hobbyist or a computer vision professional, it is unlikely that you will only deal with people and faces.

Moreover, if you are like the authors of this book, you will wonder how the person detector was created in the first place and whether you can improve it. Furthermore, you may also wonder whether you can apply the same concepts to detect diverse objects, ranging from cars to goblins.

Indeed, in the industry, you may have to deal with problems of detecting very specific objects, such as registration plates, book covers, or whatever thing may be most important to your employer or client.

Thus, the question is, how do we come up with our own classifiers?

There are many popular approaches. Throughout the remainder of this chapter, we will see that one answer lies in SVMs and the BoW technique, which is a classic combination with convenient implementations in OpenCV.

We have already talked about SVMs and HOG. Let's now take a closer look at BoW.

Understanding BoW

BoW is a concept that was not initially intended for computer vision; rather, we use an evolved version of this concept in the context of computer vision. Let's first talk about its basic version, which – as you may have guessed – originally belongs to the field of language analysis and information retrieval.

> **Note**
>
> Sometimes, in the context of computer vision, BoW is called **bag of visual words (BoVW)**. However, we will simply use the term BoW, since this is the term used by OpenCV.

BoW is the technique by which we assign a weight or count to each word in a series of documents; we then represent these documents with vectors of these counts. Let's look at an example, as follows:

- Document 1: I like OpenCV and I like Python.
- Document 2: I like C++ and Python.
- Document 3: I don't like artichokes.

These three documents allow us to build a dictionary – also called a **codebook** or vocabulary – with these values, as follows:

```
{
    I: 4,
    like: 4,
```

```
    OpenCV: 1,
    and: 2,
    Python: 2,
    C++: 1,
    don't: 1,
    artichokes: 1
}
```

We have eight entries. Let's now represent the original documents using eight-entry vectors. Each vector contains values representing the counts of all words in the dictionary, in order, for a given document. The vector representation of the preceding three sentences is as follows:

```
[2, 2, 1, 1, 1, 0, 0, 0]
[1, 1, 0, 1, 1, 1, 0, 0]
[1, 1, 0, 0, 0, 0, 1, 1]
```

These vectors can be conceptualized as a histogram representation of documents or as a descriptor vector that can be used to train classifiers. For example, a document can be classified as *spam* or *not spam* based on such a representation. Indeed, spam filtering is one of the many real-world applications of BoW.

Now that we have a grasp of the basic concept of BoW, let's see how this applies to the world of computer vision.

Applying BoW to computer vision

We are by now familiar with the concepts of features and descriptors. We have used algorithms such as SIFT and SURF to extract descriptors from an image's features so that we can match these features in another image.

We have also recently familiarized ourselves with another kind of descriptor, based on a codebook or dictionary. We know about an SVM, a model that can accept labeled descriptor vectors as training data, can find an optimal division of the descriptor space in the given classes, and can predict the classes of new data.

Armed with this knowledge, we can take the following approach to build a classifier:

1. Take a sample dataset of images.
2. For each image in the dataset, extract descriptors (with SIFT, SURF, ORB, or a similar algorithm).

3. Add each descriptor vector to the BoW trainer.
4. Cluster the descriptors into *k* clusters whose centers (centroids) are our visual words. This last point probably sounds a bit obscure, but we will explore it further in the next section.

At the end of this process, we have a dictionary of visual words ready to be used. As you can imagine, a large dataset will help make our dictionary richer in visual words. Up to a point, the more words, the better!

Having trained a classifier, we should proceed to test it. The good news is that the test process is conceptually very similar to the training process outlined previously. Given a test image, we can extract descriptors and **quantize** them (or reduce their dimensionality) by calculating a histogram of their distances to the centroids. Based on this, we can attempt to recognize visual words, and locate them in the image.

This is the point in the chapter where you have built up an appetite for a deeper practical example and are raring to code. However, before proceeding, let's take a quick but necessary digression into the theory of *k*-means clustering so that you can fully understand how visual words are created. Thereby, you will gain a better understanding of the process of object detection using BoW and SVMs.

k-means clustering

k-means clustering is a method of quantization whereby we analyze a large number of vectors in order to find a small number of clusters. Given a dataset, *k* represents the number of clusters into which the dataset is going to be divided. The term *means* refers to the mathematical concept of the mean or the average; when visually represented, the mean of a cluster is its **centroid** or the geometric center of points in the cluster.

> **Note**
>
> **Clustering** refers to the process of grouping points in a dataset into clusters.

OpenCV provides a class called `cv2.xfeatures2d.BOWKMeansTrainer`, which we will use to help train our classifier. As you might expect, the OpenCV documentation gives the following summary of this class:

> *A "k means-based class to train a visual vocabulary using the bag of visual words approach."*

After this long theoretical introduction, we can look at an example and start training our custom classifier.

Detecting cars

To train any kind of classifier, we must begin by creating or acquiring a training dataset. We are going to train a car detector, so our dataset must contain positive samples that represent cars, as well as negative samples that represent other (non-car) things that the detector is likely to encounter while looking for cars. For example, if the detector is intended to search for cars on a street, then a picture of a curb, a crosswalk, a pedestrian, or a bicycle might be a more representative negative sample than a picture of the rings of Saturn. Besides representing the expected subject matter, ideally, the training samples should represent the way our particular camera and algorithm will see the subject matter.

Ultimately, in this chapter, we intend to use a sliding window of fixed size, so it is important that our training samples conform to a fixed size and that the positive samples are tightly cropped in order to frame a car without much background.

Up to a point, we expect that the classifier's accuracy will improve as we keep adding good training images. On the other hand, a large dataset can make the training slow, and it is possible to overtrain a classifier such that it fails to extrapolate beyond the training set. Later in this section, we will write our code in a way that allows us to easily modify the number of training images so that we find a good size experimentally.

Assembling a dataset of car images would be a time-consuming task if we were to do it all by ourselves (though it is entirely doable). To avoid reinventing the wheel – or the whole car – we can avail ourselves of ready-made datasets, such as the following:

- **UIUC Image Database for Car Detection:** `https://cogcomp.seas.upenn.edu/Data/Car/`
- **Stanford Cars Dataset**: `https://github.com/jhpohovey/StanfordCars-Dataset`

Let's use the UIUC dataset in our example. Several steps are involved in obtaining this dataset and using it in a script, so let's walk through them one by one, as follows:

1. Download the UIUC dataset from `https://github.com/gcr/arc-evaluator/raw/master/CarData.tar.gz`. Unzip it to some folder, which we will refer to as `<project_path>`. Now, the unzipped data should be located at `<project_path>/CarData`. Specifically, we will use some of the images in `<project_path>/CarData/TrainImages` and `<project_path>/CarData/TestImages`.

2. Also in `<project_path>`, let's create a Python script called `detect_car_bow_svm.py`. To begin the script's implementation, write the following code to check whether the `CarData` subfolder exists:

```
import cv2
import numpy as np
import os

if not os.path.isdir('CarData'):
    print(
        'CarData folder not found. Please download and unzip '
        'https://github.com/gcr/arc-evaluator/raw/master/CarData.tar.gz '
        'into the same folder as this script.')
    exit(1)
```

   If you can run this script and it does not print anything, this means everything is in its correct place.

3. Next, let's define the following constants in the script:

```
BOW_NUM_TRAINING_SAMPLES_PER_CLASS = 10
SVM_NUM_TRAINING_SAMPLES_PER_CLASS = 11

BOW_NUM_CLUSTERS = 40
```

   Note that our classifier will make use of two training stages: one stage for the BoW vocabulary, which will use a number of images as samples, and another stage for the SVM, which will use a number of BoW descriptor vectors as samples. Arbitrarily, we have defined a different number of training samples for each stage. At each stage, we could have also defined a different number of training samples for the two classes (*car* and *not car*), but instead, we will use the same number. We have also defined the number of BoW clusters. Note that the number of BoW clusters may be larger than the number of BoW training images, since each image has many descriptors.

4. We will use `cv2.SIFT` to extract descriptors and `cv2.FlannBasedMatcher` to match these descriptors. Let's initialize these algorithms with the following code:

```
sift = cv2.SIFT_create()

FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
```

```
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

Note that we have initialized SIFT and the **Fast Library for Approximate Nearest Neighbors** (**FLANN**) in the same way as we did in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. However, this time, descriptor matching is not our end goal; instead, it will be part of the functionality of our BoW.

5. OpenCV provides a class called `cv2.xfeatures2d.BOWKMeansTrainer` to train a BoW vocabulary, and a class called `cv2.xfeatures2d.BOWImgDescriptorExtractor` to convert some kind of lower-level descriptors – in our example, SIFT descriptors – into BoW descriptors. Let's initialize these objects with the following code:

```
bow_kmeans_trainer = cv2.xfeatures2d.BOWKMeansTrainer(BOW_NUM_CLUSTERS)
bow_extractor = cv2.xfeatures2d.BOWImgDescriptorExtractor(sift, flann)
```

When initializing `cv2.xfeatures2d.BOWKMeansTrainer`, we must specify the number of clusters – in our example, `40`, as defined earlier. When initializing `cv2.xfeatures2d.BOWImgDescriptorExtractor`, we must specify a descriptor extractor and a descriptor matcher – in our example, the `cv2.SIFT` and `cv2.FlannBasedMatcher` objects that we created earlier.

6. To train the BoW vocabulary, we will provide samples of SIFT descriptors for various *car* and *not car* images. We will load the images from the `CarData/TrainImages` subfolder, which contains positive (*car*) images with names such as `pos-x.pgm`, and negative (*not car*) images with names such as `neg-x.pgm`, where `x` is a number starting at `1`. Let's write the following utility function to return a pair of paths to the `i`th positive and negative training images, where `i` is a number starting at `0`:

```
def get_pos_and_neg_paths(i):
    pos_path = 'CarData/TrainImages/pos-%d.pgm' % (i+1)
    neg_path = 'CarData/TrainImages/neg-%d.pgm' % (i+1)
    return pos_path, neg_path
```

Later in this section, we will call the preceding function in a loop, with a varying value of `i`, when we need to acquire a number of training samples.

7. For each path to a training sample, we will need to load the image, extract SIFT descriptors, and add the descriptors to the BoW vocabulary trainer. Let's write another utility function to do precisely this, as follows:

```
def add_sample(path):
    img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    if descriptors is not None:
        bow_kmeans_trainer.add(descriptors)
```

> **Note**
>
> If no features are found in the image, then the `keypoints` and descriptor variables will be `None`.

8. At this stage, we have everything we need to start training the BoW vocabulary. Let's read a number of images for each class (*car* as the positive class and *not car* as the negative class) and add them to the training set, as follows:

```
for i in range(BOW_NUM_TRAINING_SAMPLES_PER_CLASS):
    pos_path, neg_path = get_pos_and_neg_paths(i)
    add_sample(pos_path)
    add_sample(neg_path)
```

9. Now that we have assembled the training set, we will call the vocabulary trainer's `cluster` method, which performs the *k*-means classification and returns the vocabulary. We will assign this vocabulary to the BoW descriptor extractor, as follows:

```
voc = bow_kmeans_trainer.cluster()
bow_extractor.setVocabulary(voc)
```

Remember that earlier, we initialized the BoW descriptor extractor with a SIFT descriptor extractor and FLANN matcher. Now, we have also given the BoW descriptor extractor a vocabulary that we trained with samples of SIFT descriptors. At this stage, our BoW descriptor extractor has everything it needs in order to extract BoW descriptors from **Difference of Gaussian** (**DoG**) features.

> **Note**
>
> Remember that `cv2.SIFT` detects DoG features and extracts SIFT descriptors, as we discussed in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*, specifically in the `Detecting DoG features and extracting SIFT descriptors` section.

10. Next, we will declare another utility function that takes an image and returns the descriptor vector, as computed by the BoW descriptor extractor. This involves extracting the image's DoG features and computing the BoW descriptor vector based on the DoG features, as follows:

```
def extract_bow_descriptors(img):
    features = sift.detect(img)
    return bow_extractor.compute(img, features)
```

11. We are ready to assemble another kind of training set, containing samples of BoW descriptors. Let's create two arrays to accommodate the training data and labels, and populate them with the descriptors generated by our BoW descriptor extractor. We will label each descriptor vector with `1` for a positive sample and `-1` for a negative sample, as shown in the following code block:

```
training_data = []
training_labels = []

for i in range(SVM_NUM_TRAINING_SAMPLES_PER_CLASS):
    pos_path, neg_path = get_pos_and_neg_paths(i)
    pos_img = cv2.imread(pos_path, cv2.IMREAD_GRAYSCALE)
    pos_descriptors = extract_bow_descriptors(pos_img)

    if pos_descriptors is not None:
        training_data.extend(pos_descriptors)
        training_labels.append(1)
    neg_img = cv2.imread(neg_path, cv2.IMREAD_GRAYSCALE)
    neg_descriptors = extract_bow_descriptors(neg_img)

    if neg_descriptors is not None:
```

```
        training_data.extend(neg_descriptors)
        training_labels.append(-1)
```

Note

Should you wish to train a classifier to distinguish between multiple positive classes, you can simply add other descriptors with other labels. For example, we could train a classifier that uses the label 1 for *car*, 2 for *person*, and -1 for *background*. You could even have multiple positive classes and no negative or background class but, in that case, your classifier will assume that everything belongs to one of the positive classes.

12. OpenCV provides a class called `cv2.ml_SVM`, representing an SVM. Let's create an SVM, and train it with the data and labels that we previously assembled, as follows:

```
svm = cv2.ml.SVM_create()
svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
```

Note

Note that we must convert the training data and labels from lists to NumPy arrays before we pass them to the `train` method of `cv2.ml_SVM`.

13. Finally, we are ready to test the SVM by classifying some images that were not part of the training set. We will iterate over a list of paths to test images. For each path, we will load the image, extract BoW descriptors, and get the SVM's **prediction** or classification result, which will be either 1.0 (*car*) or -1.0 (*not car*), based on the training labels we used earlier. We will draw text on the image to show the classification result, and we will show the image in a window. After showing all the images, we will wait for the user to hit any key, and then the script will end. All of this is achieved in the following block of code:

```
for test_img_path in ['CarData/TestImages/test-0.pgm',
                      'CarData/TestImages/test-1.pgm',
                      '../images/car.jpg',
                      '../images/haying.jpg',
                      '../images/statue.jpg',
                      '../images/woodcutters.jpg']:
```

```
    img = cv2.imread(test_img_path)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    descriptors = extract_bow_descriptors(gray_img)
    prediction = svm.predict(descriptors)

    if prediction[1][0][0] == 1.0:
        text = 'car'
        color = (0, 255, 0)
    else:
        text = 'not car'
        color = (0, 0, 255)

    cv2.putText(img, text, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1,
                color, 2, cv2.LINE_AA)
    cv2.imshow(test_img_path, img)

cv2.waitKey(0)
```

Save and run the script. You should see six windows with various classification results. Here is a screenshot of one of the true positive results:

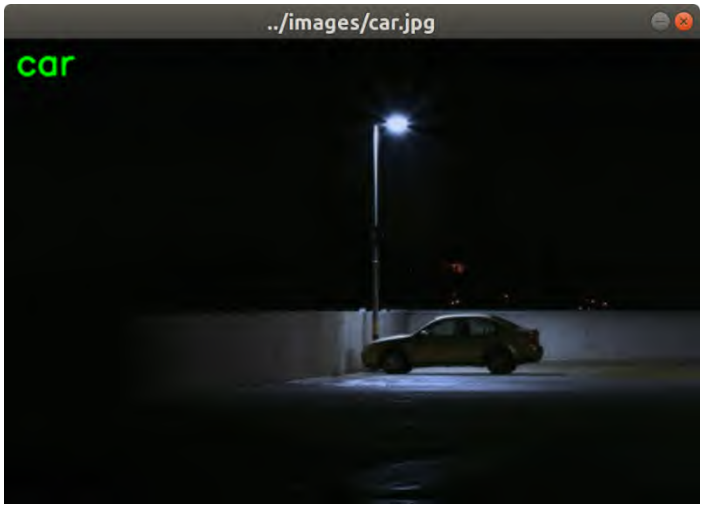


Figure 7.6: An image of a car, correctly classified by our SVM

The next screenshot shows one of the true negative results:

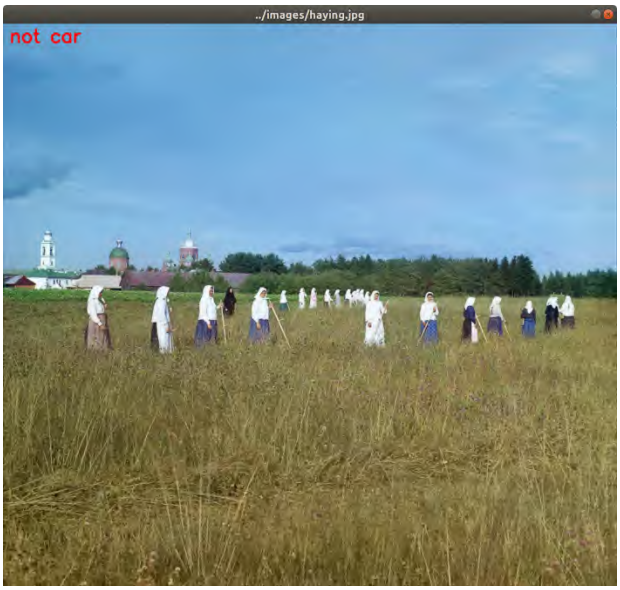


Figure 7.7: A non-car image, correctly classified by our SVM

Of the six images in our simple test, only the following one is incorrectly classified:



Figure 7.8: A non-car image, incorrectly classified as a car by our SVM

Ultimately, incorrect classification reflects a problem with the way the model was (or was not) trained for a given use case. We should consider whether the numbers of positive and negative training images were appropriate and whether the images contained relevant content for our use case. For example, if the car position in every image of our training dataset is in the center of the image, the test image with a car position at the corner of the image may be classified incorrectly. Also, as we gain proficiency in a greater variety of learning algorithms, we should always consider which algorithm is the most robust option within our computational budget.

We suggest that you experiment with adjusting the number of training samples and other parameters, and try testing the classifier on more images to see what results you can get.

Let's take stock of what we have done so far. We have used a mixture of SIFT, BoW, and SVMs to train a classifier to distinguish between two classes: *car* and *not car*. We have applied this classifier to whole images. The next logical step is to apply a sliding window technique so that we can narrow down our classification results to specific regions of an image.

Combining an SVM with a sliding window

By combining our SVM classifier with a sliding window technique and an image pyramid, we can achieve the following improvements:

- Detect multiple objects of the same kind (such as two cars) in an image.
- Determine the position and size of each detected object in an image.

We will adopt the following approach:

1. Take a region of the image, classify it, and then move this window to the right by a predefined step size. When we reach the rightmost end of the image, reset the x coordinate to **0**, move down a step, and repeat the entire process.
2. At each step, perform a classification with the SVM that was trained with BoW.
3. Keep track of all the windows that are positive detections, according to the SVM.
4. After classifying every window in the entire image, scale the image down, and repeat the entire process of using a sliding window. Thus, we are using an image pyramid. Continue rescaling and classifying until we get to a minimum size.

Tip

Choose a step size based on the level of precision you require. For example, if you need the estimate of the object's position to be accurate within 16 pixels, you should choose a step size of 16 pixels or less. Also, bear in mind that as the step size decreases, the number of overlapping windows and the computational cost increase.

When we reach the end of this process, we have collected important information about the content of the image. However, there is a problem: in all likelihood, we have found a number of overlapping blocks that each yield a positive detection with high confidence. That is to say, the image may contain one object that gets detected multiple times. If we reported these multiple detections, our report would be quite misleading, so we will filter our results using NMS.

> **Note**
>
> For a refresher, you may wish to refer back to the `Understanding NMS` section, earlier in this chapter.

Next, let's look at how to modify and extend our previous script, in order to implement the approach we have just described.

Detecting a car in a scene

We are now ready to apply all the concepts we have learned so far by creating a car detection script that scans an image and draws rectangles around cars. Let's create a new Python script, `detect_car_bow_svm_sliding_window.py`, by copying our previous script, `detect_car_bow_svm.py`. (We covered the implementation of `detect_car_bow_svm.py` earlier, in the *Detecting cars* section.) Much of the new script's implementation will remain unchanged because we still want to train a BoW descriptor extractor and an SVM in almost the same way as we did previously. However, after the training is complete, we will process the test images in a new way. Rather than classifying each image in its entirety, we will decompose each image into pyramid layers and windows, we will classify each window, and we will apply NMS to a list of windows that yielded positive detections.

For NMS, we will rely on Malisiewicz and Rosebrock's implementation, as described earlier in this chapter, in the *Understanding NMS* section. You can find a slightly modified copy of their implementation in this book's GitHub repository, specifically in the Python script at `chapter07/non_max_suppression.py`. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

This script provides a function with the following signature:

```
def non_max_suppression_fast(boxes, overlapThresh):
```

As its first argument, the function takes a *N*umPy array conta*i*ning rectangle coordinates and scores. If we have N rectangles, the shape of this array is *N*x5. For a given rectangle at index i, the values in the array have the following meanings:

- `boxes[i][0]` is the leftmost *x* coordinate.

- `boxes[i][1]` is the topmost y coordinate.
- `boxes[i][2]` is the rightmost x coordinate.
- `boxes[i][3]` is the bottommost y coordinate.
- `boxes[i][4]` is the score, where a higher score represents greater confidence that the rectangle is a correct detection result.

As its second argument, the function takes a threshold that represents the maximum proportion of overlap between rectangles. If two rectangles have a greater proportion of overlap than this, the one with the lower score will be filtered out. Ultimately, the function will return an array of the remaining rectangles.

Now, let's turn our attention to the modifications to the `detect_car_bow_svm_sliding_window.py` script, as follows:

1. First, we want to add a new import statement for the NMS function, as `highlighted` in the following code:

```
import cv2
import numpy as np
import os

from non_max_suppression import non_max_suppression_fast as nms
```

2. Let's define some additional parameters near the start of the script, as `highlighted` here:

```
BOW_NUM_TRAINING_SAMPLES_PER_CLASS = 10
SVM_NUM_TRAINING_SAMPLES_PER_CLASS = 11

BOW_NUM_CLUSERS = 12
SVM_SCORE_THRESHOLD = 2.22
NMS_OVERLAP_THRESHOLD = 0.44
```

   Note that we have reduced the number of k-means clusters from `40` to `12` (a number chosen arbitrarily based on experimentation). We will use `SVM_SCORE_THRESHOLD` as a threshold to distinguish between a positive window and a negative window. We will see how the score is obtained a little later in this section. We will use `NMS_OVERLAP_THRESHOLD` as the maximum acceptable proportion of overlap in the NMS step. Here, we have arbitrarily chosen 15%, so we will cull windows that overlap by more than this proportion. As you experiment with your SVMs, you may tweak these

parameters to your liking until you find values that yield the best results in your application.

3. We will also adjust the parameters of the SVM, as follows:

```
svm = cv2.ml.SVM_create()
svm.setType(cv2.ml.SVM_C_SVC)
svm.setC(50)
svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
```

With the preceding changes to the SVM, we are specifying the classifier's level of strictness or severity. As the value of the `C` parameter increases, the risk of false positives decreases but the risk of false negatives increases. In our application, a false positive would be a window detected as a car when it is really *not* a car, and a false negative would be a car detected as a window when it really *is* a car.

After the code that trains the SVM, we want to add two more helper functions. One of them will generate levels of the image pyramid, and the other will generate regions of interest, based on the sliding window technique. Besides adding these helper functions, we also need to handle the test images differently in order to make use of the sliding window and NMS. The following steps cover the changes:

1. First, let's look at the helper function that deals with the image pyramid. This function is shown in the following code block:

```
def pyramid(img, scale_factor=1.005, min_size=(100, 40),
            max_size=(600, 240)):
    h, w = img.shape
    min_w, min_h = min_size
    max_w, max_h = max_size
    while w >= min_w and h >= min_h:
        if w <= max_w and h <= max_h:
            yield img
        w /= scale_factor
        h /= scale_factor
        img = cv2.resize(img, (int(w), int(h)),
                         interpolation=cv2.INTER_AREA)
```

The preceding function takes an image and generates a series of resized versions of it. The series is bounded by a maximum and minimum image size.

> **Note**
>
> You will have noticed that the resized image is not returned with the `return` keyword but with the `yield` keyword. This is because this function is a so-called generator. It produces a series of images that we can easily use in a loop. If you are not familiar with generators, take a look at the official Python Wiki at `https://wiki.python.org/moin/Generators`.

2. Next up is the function to generate regions of interest, based on the sliding window technique. This function is shown in the following code block:

```
def sliding_window(img, step=20, window_size=(100, 40)):
    img_h, img_w = img.shape
    window_w, window_h = window_size
    for y in range(0, img_w, step):
        for x in range(0, img_h, step):
            roi = img[y:y+window_h, x:x+window_w]
            roi_h, roi_w = roi.shape
            if roi_w == window_w and roi_h == window_h:
                yield (x, y, roi)
```

Again, this is a generator. Although it is a bit deep-nested, the mechanism is very simple: given an image, return the upper-left coordinates and the sub-image representing the next window. Successive windows are shifted by an arbitrarily sized step from left to right until we reach the end of a row, and from the top to bottom until we reach the end of the image.

3. Now, let's consider the treatment of test images. As in the previous version of the script, we loop through a list of paths to test images, in order to load and process each one. The beginning of the loop is unchanged. For context, here it is:

```
for test_img_path in ['CarData/TestImages/test-0.pgm',
                      'CarData/TestImages/test-1.pgm',
                      '../images/car.jpg',
                      '../images/haying.jpg',
                      '../images/statue.jpg',
                      '../images/woodcutters.jpg']:
    img = cv2.imread(test_img_path)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

For each test image, we iterate over the pyramid levels, and for each pyramid level, we iterate over the sliding window positions. For each window or **region of interest (ROI)**, we extract BoW descriptors and classify them using the SVM. If the classification produces a positive result that passes a certain confidence threshold, we add the rectangle's corner coordinates and a confidence score to a list of positive detections. Continuing from the previous code block, we proceed to handle a given test image with the following code:

```
pos_rects = []
for resized in pyramid(gray_img):
    for x, y, roi in sliding_window(resized):
        descriptors = extract_bow_descriptors(roi)
        if descriptors is None:
            continue
        prediction = svm.predict(descriptors)
        if prediction[1][0][0] == 1.0:
            raw_prediction = svm.predict(
                descriptors,
                flags=cv2.ml.STAT_MODEL_RAW_OUTPUT)
            score = -raw_prediction[1][0][0]
            if score > SVM_SCORE_THRESHOLD:
                h, w = roi.shape
                scale = gray_img.shape[0] / \
                    float(resized.shape[0])
                pos_rects.append([int(x * scale),
                                  int(y * scale),
                                  int((x+w) * scale),
                                  int((y+h) * scale),
                                  score])
```

Let's take note of a couple of complexities in the preceding code, as follows:

- To obtain a confidence score for the SVM's prediction, we must run the `predict` method with an optional flag, `cv2.ml.STAT_MODEL_RAW_OUTPUT`. Then, instead of returning a label, the method returns a score as part of its output. This score may be negative, and a low value represents a high level of confidence. To make the score more intuitive – and to match the NMS function's assumption that a higher

score is better – we negate the score so that a high value represents a high level of confidence.

- Since we are working with multiple pyramid levels, the window coordinates do not have a common scale. We have converted them back to a common scale – the original image's scale – before adding them to our list of positive detections.

So far, we have performed car detection at various scales and positions; as a result, we have a list of detected car rectangles, including coordinates and scores. We expect a lot of overlap within this list of rectangles.

4. Now, let's call the NMS function, in order to cherry-pick the highest-scoring rectangles in the case of overlap, as follows:

```
    pos_rects = nms(np.array(pos_rects), NMS_OVERLAP_THRESHOLD)
```

Note

Note that we have converted our list of rectangle coordinates and scores to a NumPy array, which is the format expected by this function.

At this stage, we have an array of detected car rectangles and their scores, and we have ensured that these are the best non-overlapping detections we can select (within the parameters of our model).

5. Now, let's draw the rectangles and their scores by adding the following inner loop to the code:

```
    for x0, y0, x1, y1, score in pos_rects:
        cv2.rectangle(img, (int(x0), int(y0)), (int(x1), int(y1)),
                      (0, 255, 255), 2)
        text = '%.2f' % score
        cv2.putText(img, text, (int(x0), int(y0) - 20),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
```

As in the previous version of this script, the body of the outer loop ends by showing the current test image, including the annotations we have drawn on it. After the loop runs through all the test images, we wait for the user to press any key; then, the program ends, as shown here:

```
    cv2.imshow(test_img_path, img)
cv2.waitKey(0)
```

Let's run the modified script, and see how well it can answer the eternal question: *Dude, where's my car?*

The following screenshot shows a pair of successful detections:

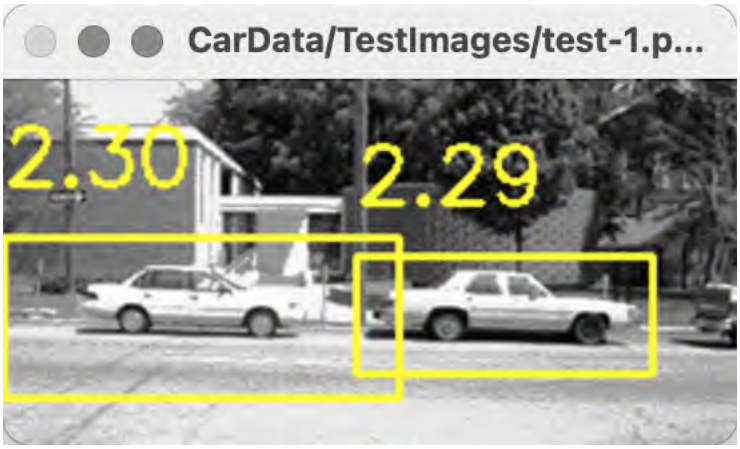


Figure 7.9: A pair of cars, correctly detected by our SVM with a sliding window approach

Among our other five test images, we have two false negatives (where a car is not detected as such) but no false positives.

Note

Remember that in this sample script, our training sets are small. Larger training sets, with more diverse backgrounds, could improve the results. Cropping the positive training samples more tightly around the car could help too. Also, remember that the image pyramid and sliding window are producing a large number of ROIs. When we consider this, we should realize that a low false *positive* rate is actually a significant accomplishment. Moreover, if we were performing detection on frames of a video, we could mitigate the problem of false *negatives* because we would have a chance to detect a car in multiple frames.

Feel free to experiment with the parameters and training sets of the preceding script. When you are ready, let's wrap up this chapter with a few closing notes.

Saving and loading a trained SVM

A final piece of advice on SVMs: you do not need to train a detector every time you want to use it – and, indeed, you should avoid doing so because training is slow. You can use code such as the following to save a trained SVM model to an XML file:

```
svm = cv2.ml.SVM_create()
svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
svm.save('my_svm.xml')
```

Subsequently, you can reload the trained SVM, using code such as the following:

```
svm = cv2.ml.SVM_create()
svm.load('my_svm.xml')
```

Typically, you might have one script that trains and saves your SVM model, and other scripts that load and use it for various detection problems.

Now that we have trained a custom SVM and used it as a detector with a sliding window approach, you may be wondering how to train custom models for OpenCV's `HOGDescriptor` class. After all, `HOGDescriptor` uses an SVM classifier and a sliding window just like we did, but with HOG descriptors instead of BoW descriptors. Next, let's see what revisions are required to replace BoW with HOG in our car detection script.

Training a custom HOG detector

With OpenCV, training and using a custom HOG detector is a five-step process:

1. Create an instance of `cv2.HOGDescriptor` and use its `compute` method to extract HOG descriptors for various training images.
2. Create an instance of `cv2.ml.SVM_create()` and train it as a classifier based on the training images' HOG descriptors (from the previous step) and labels (such as *car* or *non-car*).
3. Via the `HOGDescriptor`'s `setSVMDetector` method, tell it to use the custom SVM that we trained in the previous step.

4. Optionally, save the trained detection model as a file via `HOGDescriptor`'s `save` method, so that in the future we can load it via `HOGDescriptor`'s `load` method.
5. Detect objects in images by calling `HOGDescriptor`'s `detectMultiScale` method as usual. When we do this, the HOG detector internally uses our custom SVM to differentiate between the positive class and negative class for which we trained it.

Taking our previous samples of car detection as a starting point, we can keep much of the code that pertains to loading training images and training an SVM. However, our approach to generating descriptors will differ because, this time, we are going to use HOG descriptors instead of BOW descriptors. Also, this time, we will perform detection via the `HOGDescriptor` class instead of manually passing a series of sliding windows to the SVM classifier.

Let's make a copy of our car detection script and modify it. To begin, we can replace our definitions of various BoW, SVM, and NMS parameters with the following definitions, which are relevant to our use of HOG and SVM together:

```
HOG_WINDOW_SIZE = (96, 48)
HOG_WEIGHT_THRESHOLD = 0.45

SVM_NUM_TRAINING_SAMPLES_PER_CLASS = 300
```

Tip

OpenCV requires the HOG window size to be a multiple of `8` in each dimension. Subject to this constraint, choose a window size based on the smallest (or most distant) object you may want to detect. Here, we have chosen a small, horizontally elongated window size, `(96, 40)`, which should be appropriate for the cars in our training and test datasets. Our training images have an original size of `(100, 40)`, so our script will need to resize them just a little to match the window size.

We can proceed to initialize a `HOGDescriptor` with our chosen window size and various other parameters:

```
hog = cv2.HOGDescriptor(HOG_WINDOW_SIZE, (16, 16), (8, 8), (8, 8), 9)
```

All the training images will need to be resized to match the HOG window size. Thus, let's replace our old `extract_bow_descriptors` function with a new function to resize a training image and extract its HOG descriptors, as follows:

```
def extract_hog_descriptors(img):
    resized = cv2.resize(img, HOG_WINDOW_SIZE, cv2.INTER_CUBIC)
    return hog.compute(resized, (16, 16), (0, 0))
```

Apart from the preceding helper function, our process for assembling lists of descriptors and labels will not change much. However, note the minor updates highlighted in the following code block:

```
training_data = []
training_labels = []

for i in range(SVM_NUM_TRAINING_SAMPLES_PER_CLASS):
    pos_path, neg_path = get_pos_and_neg_paths(i)
    pos_img = cv2.imread(pos_path, cv2.IMREAD_GRAYSCALE)
    pos_descriptors = extract_hog_descriptors(pos_img)

    if pos_descriptors is not None:
        training_data.append(pos_descriptors)
        training_labels.append(1)
    neg_img = cv2.imread(neg_path, cv2.IMREAD_GRAYSCALE)
    neg_descriptors = extract_hog_descriptors(neg_img)

    if neg_descriptors is not None:
        training_data.append(neg_descriptors)
        training_labels.append(-1)
```

Previously, with BoW descriptors, we used Python's `extend` method to build the lists; now, with HOG descriptors, we use the `append` method. This is simply due to differences in the original shape of the two kinds of descriptor data.

Having assembled the lists of descriptors and labels, we can train an SVM as usual. However, we will customize the SVM's parameters as follows:

```
svm = cv2.ml.SVM_create()
svm.setDegree(3)
criteria = (cv2.TERM_CRITERIA_MAX_ITER + cv2.TERM_CRITERIA_EPS, 1000, 1e-3)
svm.setTermCriteria(criteria)
```

```
svm.setKernel(cv2.ml.SVM_LINEAR)
svm.setNu(0.5)
svm.setP(0.1)
svm.setC(0.01)
svm.setType(cv2.ml.SVM_EPS_SVR)

svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
```

Note

For details on the many parameters supported by OpenCV's `SVM` class, see the official API documentation at `https://docs.opencv.org/5.0/extra_modules/classcv_1_1ml_1_1SVM.html`.

So far, in our modified code, we have trained an SVM to classify sets of HOG descriptors. Note that we could use this trained SVM manually to classify the HOG descriptors of other images or image regions. For example, we might want to do this if we had a classification problem with multiple positive classes (such as a horse, donkey, and zebra) instead of a detection problem with one positive class and one negative class. However, for the latter case, it is more convenient to tell `HOGDescriptor` to use the SVM for detection, and then use the `detectMultiScale` method, which can handle other details such as the sliding window implementation.

With this in mind, let's configure our `HOGDescriptor` to use our SVM. OpenCV's API is, unfortunately, quite opaque in this area because the `SVM` class and the `HOGDescriptor` class have different ideas about how the model should be represented in matrix format. The following code performs a manual conversion between the two representations and then tells our HOG detector to use the SVM:

```
support_vectors = np.transpose(svm.getSupportVectors())
rho, _, _ = svm.getDecisionFunction(0)
svm_detector = np.append(support_vectors, [[-rho]], 0)
hog.setSVMDetector(svm_detector)
```

At this point, we have finished training our instance of `HOGDescriptor`, so we can use it to detect objects (in this case, cars) by calling its `detectMultiScale` method, exactly as we did in this chapter's earlier examples of person detection. We can filter out overlapping detections (as we did previously) and we can also reject results that have a low confidence score or low

"weights." There are no new methods to learn here, but let's look at how the familiar code for HOG-based detection fits into the framework of our car detection script:

```
def is_inside(i, o):
    ix, iy, iw, ih = i
    ox, oy, ow, oh = o
    return ix > ox and ix + iw < ox + ow and \
        iy > oy and iy + ih < oy + oh

for test_img_path in ['CarData/TestImages/test-0.pgm',
                      'CarData/TestImages/test-1.pgm',
                      '../images/car.jpg',
                      '../images/haying.jpg',
                      '../images/statue.jpg',
                      '../images/woodcutters.jpg']:
    img = cv2.imread(test_img_path)

    found_rects, found_weights = hog.detectMultiScale(
        img, winStride=(8, 8), scale=1.03, groupThreshold=2.0)

    found_rects_filtered = []
    found_weights_filtered = []
    for ri, r in enumerate(found_rects):
        if found_weights[ri] < HOG_WEIGHT_THRESHOLD:
            continue
        for qi, q in enumerate(found_rects):
            if found_weights[qi] < HOG_WEIGHT_THRESHOLD:
                continue
            if ri != qi and is_inside(r, q):
                break
        else:
            found_rects_filtered.append(r)
            found_weights_filtered.append(found_weights[ri])

    for ri, r in enumerate(found_rects_filtered):
        x, y, w, h = r
        cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 255), 2)
        text = '%.2f' % found_weights_filtered[ri]
        cv2.putText(img, text, (x, y - 20),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
```

Having made the `highlighted` changes, let's run the script and see the results. OpenCV's `HOGDescriptor` is a lot more robust than our homemade BoW descriptors, and we now have accurate results for all six of the test images listed above! Here is an example of a true positive detection result:

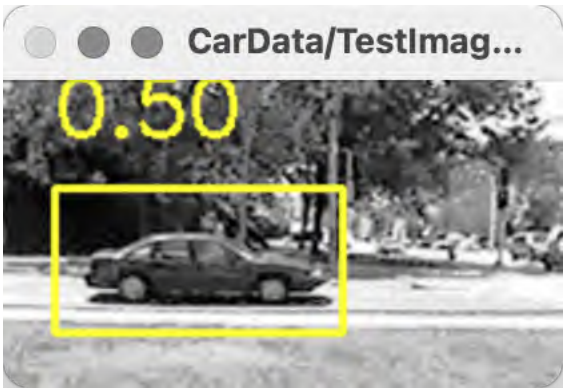


Figure 7.10: A car, correctly detected with our custom HOG detector

Go ahead and try the detector with some more images from the test dataset or other sources. Try adjusting parameters and see how the results change. For example, if you change the window size to `(192, 80)`, this will increase the confidence of the detection result in `car.jpg`; however, cars in some of the lower-resolution images will not be detected.

Tip

To improve the detector's training, you may wish to apply the technique of **hard negative mining**. This is a way to partially automate the selection of new negative training samples, in order to re-train a detector for cases that it finds "hard". Take a set of high-resolution background images, containing no positive samples (no cars). Run the detector on these images. Any resulting detections must be false positives, so crop out each detection window and add it to a set of negative training samples. At the end of this process, re-train the model with the old training samples plus the newly detected "hard negatives".

Lastly, note that we can save and load a trained HOG model in much the same way as we can save and load a trained SVM. Here is a trivial example:

```
# Save a trained model to file.
# We might do this at the end of a training script.
hog.save("my_model.xml")
```

```
# Load a trained model from file.
# We might do this at the start of an application script,
# before performing detections.
another_hog = cv2.HOGDescriptor()
another_hog.load("my_model.xml")
```

Now you can train a whole herd of HOGs for various purposes, and you can take them wherever you go!

Summary

In this chapter, we covered a wide range of concepts and techniques, including HOG, BoW, SVMs, image pyramids, sliding windows, and NMS. We learned that these techniques have applications in object detection, as well as other fields. We wrote a script that combined most of these techniques – BoW, SVMs, an image pyramid, a sliding window, and NMS – and we gained practical experience in machine learning through the exercise of training and testing a custom detector. We also repeated this exercise using HOG descriptors instead of BoW descriptors. Finally, we demonstrated that we can accurately detect cars!

Our new knowledge forms the foundation of the next chapter, in which we will utilize object detection and classification techniques on sequences of frames in videos. We will learn how to track objects and retain information about them – an important objective in many real-world applications.

8

Tracking Objects

In this chapter, we will explore a selection of techniques from the vast topic of object tracking, which is the process of locating a moving object in a movie or a video feed from a camera. Real-time object tracking is a critical task in many computer vision applications, such as surveillance, perceptual user interfaces, augmented reality, object-based video compression, and driver assistance.

Tracking objects can be accomplished in several ways, with the optimal technique being largely dependent on the task at hand. We will take the following route in our study of this topic:

- Detect moving objects based on differences between the current frame and a frame that represents the background. First, we will try a simple implementation of this approach. Then, we will use OpenCV's implementations of more advanced algorithms, namely, the **Mixture of Gaussians** (**MOG**) and **k-nearest neighbors** (**KNN**) background subtractors. We will also consider how to modify our scripts to use any other background subtractor that OpenCV supports, such as the **Godbehere-Matsukawa-Goldberg** (**GMG**) background subtractor.
- Track a moving object based on a color histogram of the object. This approach involves histogram back-projection, which is the process of computing the similarity between various image regions and a histogram. In other words, the histogram serves as a template of how we expect the object to look. We will use tracking algorithms called **MeanShift** and **CamShift**, which operate on the result of the histogram back-projection.
- Use a Kalman filter to find a trend in an object's motion and to predict where the object is going next.

- Review the manner in which OpenCV favors **object-oriented programming** (**OOP**) paradigms and consider how this differs from **functional programming** (**FP**) paradigms.
- Implement a pedestrian tracker that combines KNN background subtraction, MeanShift, and Kalman filtering.
- Modify the pedestrian tracker to use **Multiple Instance Learning Tracker** (**MILTrack**) or any of the other machine learning algorithms that OpenCV implements via its `cv2.Tracker` interface.

If you have been reading this book sequentially, then, by the end of this chapter, you will know a lot of ways to describe, detect, classify, and track objects in 2D. At that point, you should be ready to pursue 3D tracking in *Chapter 9, Camera Models and Augmented Reality*.

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Please refer to *Chapter 1, Setting Up OpenCV*, for installation instructions.

The complete code and sample videos for this chapter can be found in this book's GitHub

repository in the `chapter08` folder. The sample videos can be found in the `videos` folder.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Detecting moving objects with background subtraction

To track anything in a video, first, we must identify the regions of a video frame that correspond to moving objects. Many motion detection techniques are based on the simple concept of **background subtraction**. For example, suppose that we have a stationary camera viewing a scene that is also mostly stationary. In addition to this, suppose that the camera's exposure and the lighting conditions in the scene are stable so that frames do not vary much in terms of brightness. Under these conditions, we can easily capture a reference image that represents the background or, in other words, the stationary components of the scene. Then, any time the camera captures a new frame, we can subtract the frame from the reference image and take the absolute value of this difference in order to obtain a measurement of motion at each pixel location in the frame. If any region of the frame is very different from the reference image, we conclude that the given region is a moving object.

Background subtraction techniques, in general, have the following limitations:

- Any camera motion, change in exposure, or change in lighting conditions can cause a change in pixel values throughout the entire scene all at once; therefore, the entire background model (or reference image) becomes outdated.
- A piece of the background model can become outdated if an object enters the scene and then just stays there for a long period of time. For example, suppose our scene is a hallway. Someone enters the hallway, puts a poster on the wall, and leaves the poster there. For all practical purposes, the poster is now just another part of the stationary background; however, it was not part of our reference image, so our background model has become partly outdated.

These problems point to a need to dynamically update the background model based on a series of new frames. Advanced background subtraction techniques attempt to address this need in a variety of ways.

Another general limitation is that shadows and solid objects can affect a background subtractor in similar ways. For instance, we might get an inaccurate picture of a moving object's size and shape because we cannot differentiate the object from its shadow. However, advanced background subtraction techniques do attempt to distinguish between shadow regions and solid objects using various means.

Background subtractors generally have yet another limitation: they do not offer fine-grained control over the kind of motion that they detect. For example, if a scene shows a subway car that is continuously shaking as it travels on its track, this repetitive motion will affect the background subtractor. For practical purposes, we might consider the subway car's vibrations to be normal variations in a semi-stationary background. We might even know the frequency of these vibrations. However, a background subtractor does not embed any information about frequencies of motion, so it does not offer a convenient or precise way in which to filter out such predictable motions. To compensate for such shortcomings, we can apply preprocessing steps such as blurring the reference image and also blurring each new frame; in this way, certain frequencies are suppressed, albeit in a manner that is not very intuitive, efficient, or precise.

Note

Analyzing the frequencies of motion is beyond the scope of this book. However, for an introduction to this topic in a computer vision context, see Joseph Howse's book, *OpenCV 4 for Secret Agents* (Packt Publishing, 2019), specifically *Chapter 7, Seeing a Heartbeat with a Motion-Amplifying Camera*.

Now that we have had an overview of background subtraction and understood some of the obstacles that it faces, let's investigate how several implementations of it fare in action. We will start with a simple but not robust implementation that we can handcraft in a few lines of code and then progress to more sophisticated alternatives that OpenCV provides for us.

Implementing a basic background subtractor

To implement a basic background subtractor, let's take the following approach:

1. Start capturing frames from a camera.
2. Discard the first nine frames so that the camera has time to properly adjust its autoexposure to suit the lighting conditions in the scene.
3. Take the 10th frame, convert it to grayscale, blur it, and use this blurred image as the reference image for the background.
4. For each subsequent frame, blur the frame, convert it to grayscale, and compute the absolute difference between this blurred frame and the reference image of the background. Perform thresholding, smoothing, and contour detection on the differenced image. Draw and show the bounding boxes of the major contours.

> **Note**
>
> The use of a Gaussian blur should make our background subtractor less susceptible to small vibrations, as well as digital noise. We will also use morphological operations for the same purpose.

To blur images, we will use the Gaussian blur algorithm, which we originally discussed in *Chapter 3, Processing Images with OpenCV*, specifically in the *HPFs and LPFs* section. To smooth thresholded images, we will use morphological erosion and dilation, which we originally discussed in *Chapter 4, Depth Estimation and Segmentation*, specifically in the *Image segmentation with the Watershed algorithm* section. Contour detection and bounding boxes are also among the topics we introduced in *Chapter 3, Processing Images with OpenCV*, specifically in the *Contour detection* section.

Expanding the preceding list into smaller steps, we can consider our script's implementation in eight sequential blocks of code:

1. Let's begin by importing OpenCV and defining the size of our kernels for the `blur`, `erode`, and `dilate` operations:

```
import cv2
```

```
BLUR_RADIUS = 21
erode_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
dilate_kernel = cv2.getStructuringElement(
    cv2.MORPH_ELLIPSE, (9, 9))
```

2. Now, let's try to capture 10 frames from a camera:

```
cap = cv2.VideoCapture(0)

# Capture several frames to allow the camera's autoexposure to adjust.
for i in range(10):
    success, frame = cap.read()
if not success:
    exit(1)
```

3. If we were unable to capture 10 frames, we exit. Otherwise, we proceed to convert the 10^{th} frame to grayscale and blur it:

```
gray_background = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
gray_background = cv2.GaussianBlur(
    gray_background, (BLUR_RADIUS, BLUR_RADIUS), 0)
```

4. At this stage, we have our reference image of the background. Now, let's proceed to capture more frames, in which we may detect motion. Our processing of each frame begins with grayscale conversion and a Gaussian blur operation:

```
success, frame = cap.read()
while success:

    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    gray_frame = cv2.GaussianBlur(
        gray_frame, (BLUR_RADIUS, BLUR_RADIUS), 0)
```

5. Now, we can compare the blurred, grayscale version of the current frame to the blurred, grayscale version of the background image. Specifically, we will use OpenCV's `cv2.absdiff` function to find the absolute value (or the magnitude) of the difference between these two images. Then, we will apply a threshold to obtain a pure black-and-

white image, and morphological operations to smooth the thresholded image. Here is the relevant code:

```
diff = cv2.absdiff(gray_background, gray_frame)
_, thresh = cv2.threshold(diff, 40, 255, cv2.THRESH_BINARY)
cv2.erode(thresh, erode_kernel, thresh, iterations=2)
cv2.dilate(thresh, dilate_kernel, thresh, iterations=2)
```

6. At this point, if our technique has worked well, our thresholded image should contain white blobs wherever there is a moving object. Now, we want to find the contours of the white blobs and draw bounding boxes around them. As a further means of filtering out small changes that are probably not real objects, we will apply a threshold based on the area of the contour. If the contour is too small, we conclude that it is not a real moving object. (Of course, the definition of *too small* may vary depending on your camera's resolution and your application; in some circumstances, you might not wish to apply this test at all.) Here is the code to detect contours and draw bounding boxes:

```
_, contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
                                     cv2.CHAIN_APPROX_SIMPLE)

for c in contours:
    if cv2.contourArea(c) > 4000:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 255, 0), 2)
```

7. Now, let's show the differenced image, the thresholded image, and the detection result with the bounding rectangles:

```
cv2.imshow('diff', diff)
cv2.imshow('thresh', thresh)
cv2.imshow('detection', frame)
```

8. We will continue reading frames until the user presses the *Esc* key to quit:

```
k = cv2.waitKey(1)
if k == 27: # Escape
    break

success, frame = cap.read()
```

There you have it: a basic motion detector that draws rectangles around moving objects. The final result is something like this:

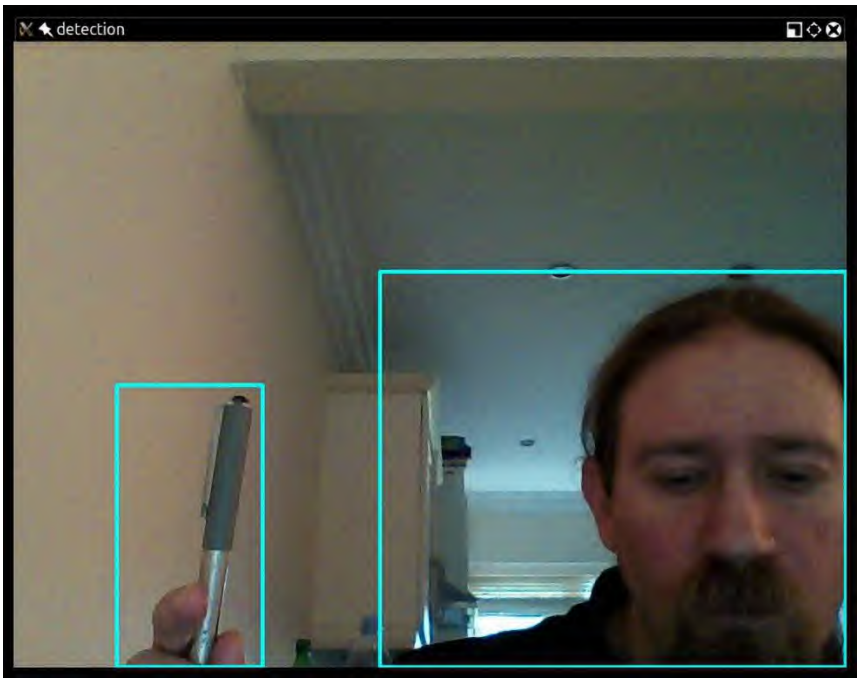


Figure 8.1: A moving pen and a moving person, detected using a basic background subtractor

Tip

To obtain good results with this script, make sure that you (and other moving objects) do not enter the camera's field of view until after the background image has been initialized.

For such a simple technique, this result is quite promising. However, our script makes no effort to update the background image dynamically, so it will quickly become outdated if the camera moves or the lighting changes. Thus, we should move on to more flexible and intelligent background subtractors. Fortunately, OpenCV provides several ready-made background subtractors for our use. We will start with the one that implements the MOG algorithm.

Using a MOG background subtractor

OpenCV provides a class called `cv2.BackgroundSubtractor`, which has various subclasses that implement various background subtraction algorithms.

You may recall that we previously used OpenCV's implementation of the GrabCut algorithm to perform foreground/background segmentation in *Chapter 4, Depth Estimation and Segmentation*, specifically in the *Foreground detection with the GrabCut algorithm* section. Like `cv2.grabCut`, the various subclass implementations of `cv2.BackgroundSubtractor` can produce a mask that assigns different values to different segments of the image. Specifically, a background subtractor can mark foreground segments as white (that is, an 8-bit grayscale value of 255), background segments as black (0), and (in some implementations) shadow segments as gray (127). Moreover, unlike GrabCut, the background subtractors update the foreground/background model over time, typically by applying machine learning to a series of frames. Many of the background subtractors are named after the statistical clustering technique on which they base their approach to machine learning. We will begin by looking at a background subtractor based on the MOG clustering technique.

OpenCV has two implementations of a MOG background subtractor. Perhaps not surprisingly, they are named `cv2.bgsegm.BackgroundSubtractorMOG` and `cv2.BackgroundSubtractorMOG2`. The latter is a more recent and improved implementation, which adds support for shadow detection, so we will use it.

As a starting point, let's take our basic background subtraction script from the previous section. We will make the following modifications to it:

1. Replace our basic background subtraction model with a MOG background subtractor.
2. As input, use a video file instead of a camera.
3. Remove the use of Gaussian blur.
4. Adjust the parameters used in the thresholding, morphology, and contour analysis steps.
5. Get and show the MOG subtractor's model of the background.

These modifications affect a few lines of code, which are scattered throughout the script. Near the top of the script, let's initialize the MOG background subtractor and modify the size of the morphology kernels, as `highlighted` in the following code block:

```
import cv2

bg_subtractor = cv2.createBackgroundSubtractorMOG2(detectShadows=True)

erode_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
dilate_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 7))
```

Note that OpenCV provides a function, `cv2.createBackgroundSubtractorMOG2`, to create an instance of `cv2.BackgroundSubtractorMOG2`. The function accepts a parameter, `detectShadows`, which we set to `True` so that the shadow regions will be marked as such and not marked as part of the foreground.

The remaining changes, including the use of the MOG background subtractor to obtain a foreground/shadow/background mask, are `highlighted` in the following code block:

```
cap = cv2.VideoCapture('../videos/hallway.mpg')
success, frame = cap.read()
while success:

    fg_mask = bg_subtractor.apply(frame)

    _, thresh = cv2.threshold(fg_mask, 244, 255, cv2.THRESH_BINARY)
    cv2.erode(thresh, erode_kernel, thresh, iterations=2)
    cv2.dilate(thresh, dilate_kernel, thresh, iterations=2)

    contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
                                      cv2.CHAIN_APPROX_SIMPLE)

    for c in contours:
        if cv2.contourArea(c) > 1000:
            x, y, w, h = cv2.boundingRect(c)
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 255, 0), 2)

    cv2.imshow('mog', fg_mask)
    cv2.imshow('thresh', thresh)
    cv2.imshow('background', bg_subtractor.getBackgroundImage())
    cv2.imshow('detection', frame)

    k = cv2.waitKey(30)
    if k == 27:  # Escape
        break

    success, frame = cap.read()
```

When we pass a frame to the background subtractor's `apply` method, the subtractor updates its internal model of the background and then returns a mask. As we previously discussed, the mask is white (`255`) for foreground segments, gray (`127`) for shadow segments, and black (`0`)

for background segments. For our purposes, we treat shadows as the background, so we apply a nearly white threshold (244) to the mask.

Note

To eliminate the thresholding step, we could configure the background subtractor to color the shadows black (instead of gray). To do this, we would use the subtractor's `setShadowValue` method, as `highlighted` below:

```
bg_subtractor = cv2.createBackgroundSubtractorMOG2(detectShadows=True)
bg_subtractor.setShadowValue(0)
```

However, for our learning purposes, it is good to see a visualization of the shadows in gray before we filter them out.

The following set of screenshots shows a mask from the MOG detector (the top-left panel), a thresholded and morphed version of this mask (top-right), the model of the background (bottom-left), and the detection result (bottom-right):



Figure 8.2: A moving person, detected using a MOG background subtractor

This scene contains not only shadows but also reflections due to the polished floor and wall. When shadow detection is enabled (as in the preceding screenshots), we are able to use a threshold to remove the shadows and reflections from our mask, leaving us with an accurate detection rectangle around the man in the hall. For comparison, if we disabled shadow detection by setting `detectShadows=False`, we would get results such as the following set of screenshots:



Figure 8.3: A moving person and moving reflections, detected using a MOG background subtractor

Now, with shadow detection disabled, we have two detections, which are both, arguably, inaccurate. One detection covers the man, his shadow, and his reflection on the floor. The second detection covers the man's reflection on the wall. These are, *arguably*, inaccurate detections because the man's shadow and reflections are not really moving objects, even though they are visual artifacts of a moving object.

So far, we have seen that a background subtraction script can be very concise and that a few small changes can drastically change the algorithm and the results, for better or worse. Continuing in the same vein, let's see how easily we can adapt our code to use another of OpenCV's advanced background subtractors to find another kind of moving object.

Using a KNN background subtractor

By modifying just five lines of code in our MOG background subtraction script, we can use a different background subtraction algorithm, different morphology parameters, and a different video as input. Thanks to the high-level interface that OpenCV provides, even such simple changes enable us to successfully handle a wide variety of background subtraction tasks.

Just by replacing `cv2.createBackgroundSubtractorMOG2` with `cv2.createBackgroundSubtractorKNN`, we can use a background subtractor based on KNN clustering instead of MOG clustering:

```
bg_subtractor = cv2.createBackgroundSubtractorKNN(detectShadows=True)
```

Note that despite the change in algorithm, the `detectShadows` parameter is still supported, as is the `setShadowValue` method. Additionally, the `apply` and `getBackgroundImage` methods are still supported, so we do not need to change anything related to the use of the background subtractor later in the script.

> **Note**
>
> Remember that `cv2.createBackgroundSubtractorMOG2` returns a new instance of the `cv2.BackgroundSubtractorMOG2` class. Similarly, `cv2.createBackgroundSubtractorKNN` returns a new instance of the `cv2.BackgroundSubtractorKNN` class. Both of these classes are subclasses of `cv2.BackgroundSubtractor`, which defines common methods such as `apply`.

With the following changes, we can use morphology kernels that are slightly better adapted to a horizontally elongated object (in this case, a car), and we can use a video of traffic as input:

```
erode_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 5))
dilate_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (17, 11))

cap = cv2.VideoCapture('../videos/traffic.flv')
```

To reflect the change in algorithm, let's change the title of the mask's window from `'mog'` to `'knn'`:

```
    cv2.imshow('knn', fg_mask)
```

The following set of screenshots shows the results of motion detection:

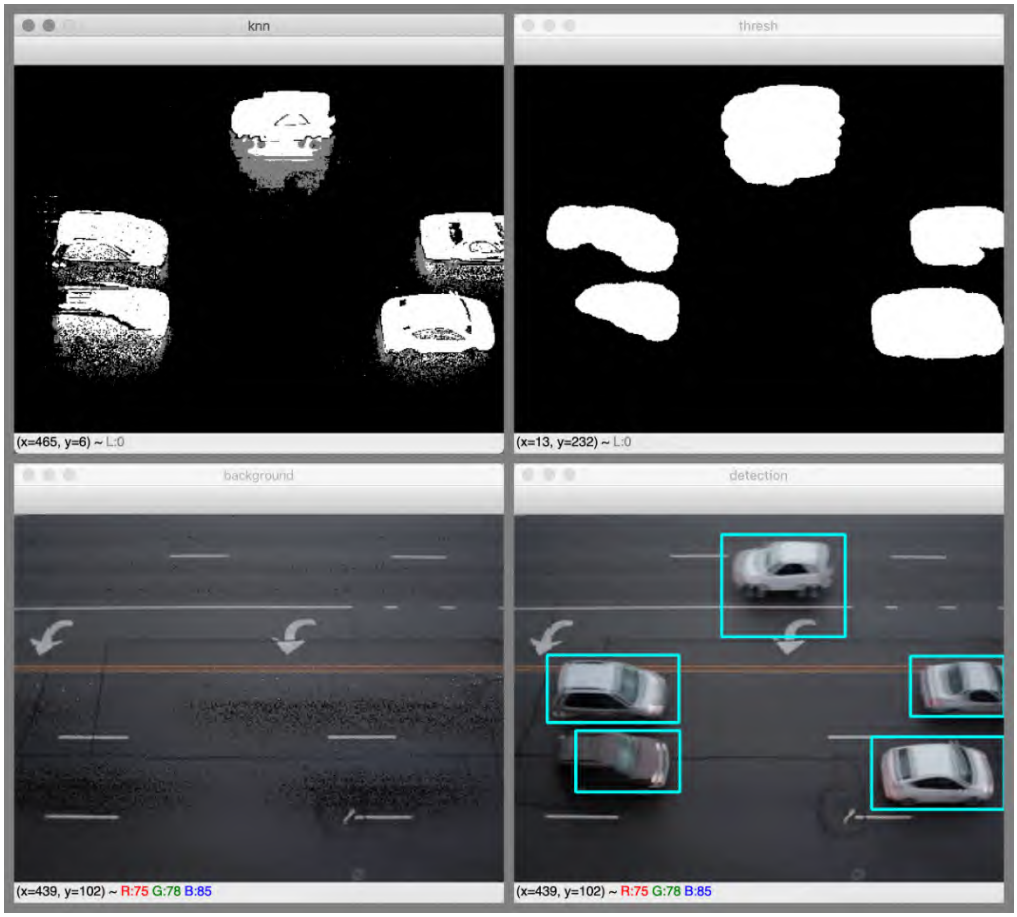


Figure 8.4: Moving cars, detected using a KNN background subtractor

The KNN background subtractor has worked quite well here, thanks in part to its ability to differentiate between objects and shadows. All cars have been individually detected; even though some cars are close to each other, they have not been merged into one detection. For three out of the five cars, the detection rectangles are accurate. For the dark car in the bottom-left part of the video frame, the background subtractor has failed to fully differentiate the rear of the car from the asphalt. For the white car at the top of the frame, the background subtractor has failed to fully differentiate the car and its shadow from the white markings on the road. Such problems could possibly be addressed by fine-tuning the morphology and threshold parameters, or by performing further analysis on the detected contours and surrounding regions based on the expected shape of a vehicle. Nonetheless, overall, this is a useful detection result that could enable us to count the number of cars traveling in each lane.

As we have seen, a few simple variations on a script can produce very different background subtraction results. Let's consider how we could further explore this observation.

Using GMG and other background subtractors

You are free to experiment with your own modifications to our background subtraction script. If you have obtained OpenCV with the optional `opencv_contrib` modules, as described in *Chapter 1, Setting Up OpenCV*, then several more background subtractors are available to you in the `cv2.bgsegm` module. They can be created using the following functions:

- `cv2.bgsegm.createBackgroundSubtractorCNT`
- `cv2.bgsegm.createBackgroundSubtractorGMG`
- `cv2.bgsegm.createBackgroundSubtractorGSOC`
- `cv2.bgsegm.createBackgroundSubtractorLSBP`
- `cv2.bgsegm.createBackgroundSubtractorMOG`

These functions do not support the `detectShadows` parameter, and they create background subtractors that do not support shadow detection. However, all the background subtractors support the `apply` method.

> **Note**
>
> As these types of background subtractors do not support shadow detection, they do not support the `setShadowValue` method. Moreover, some of them, including the GMG subtractor, do not support the `getBackgroundImage` method.

As an example of how to modify our background subtraction sample to use one of the `cv2.bgsegm` subtractors in the preceding list, let's use a GMG background subtractor. The relevant modifications are `highlighted` in the following block of code:

```
import cv2

bg_subtractor = cv2.bgsegm.createBackgroundSubtractorGMG()

erode_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (13, 9))
dilate_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (17, 11))

cap = cv2.VideoCapture('../videos/traffic.flv')
success, frame = cap.read()
while success:
```

```
    fg_mask = bg_subtractor.apply(frame)

    _, thresh = cv2.threshold(fg_mask, 244, 255, cv2.THRESH_BINARY)
    cv2.erode(thresh, erode_kernel, thresh, iterations=2)
    cv2.dilate(thresh, dilate_kernel, thresh, iterations=2)

    contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
                                      cv2.CHAIN_APPROX_SIMPLE)

    for c in contours:
        if cv2.contourArea(c) > 1000:
            x, y, w, h = cv2.boundingRect(c)
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 255, 0), 2)

    cv2.imshow('gmg', fg_mask)
    cv2.imshow('thresh', thresh)
    cv2.imshow('detection', frame)

    k = cv2.waitKey(30)
    if k == 27: # Escape
        break

    success, frame = cap.read()
```

Note the modifications are similar to the ones we saw in the previous section, *Using a KNN background subtractor*. We simply use a different function to create the GMG subtractor, we adjust the morphology kernels' sizes to values that work better for this algorithm, and we change one of the window titles to `'gmg'`.

Note

The GMG algorithm is named after its authors, Andrew B. Godbehere, Akihiro Matsukawa, and Ken Goldberg. They describe it in their paper, *Visual tracking of human visitors under variable-lighting conditions for a responsive audio art installation* (ACC, 2012), which is available at `https://ieeexplore.ieee.org/document/6315174`. The GMG background subtractor takes a few frames to initialize itself before it starts producing a mask with white (object) regions.

Compared to the KNN background subtractor, the GMG background subtractor produces worse results with our sample video of traffic. This is partly because OpenCV's implementation of GMG does not differentiate between shadows and solid objects, so the detection rectangles are elongated in the direction of the cars' shadows or reflections. Here is a sample of the output:

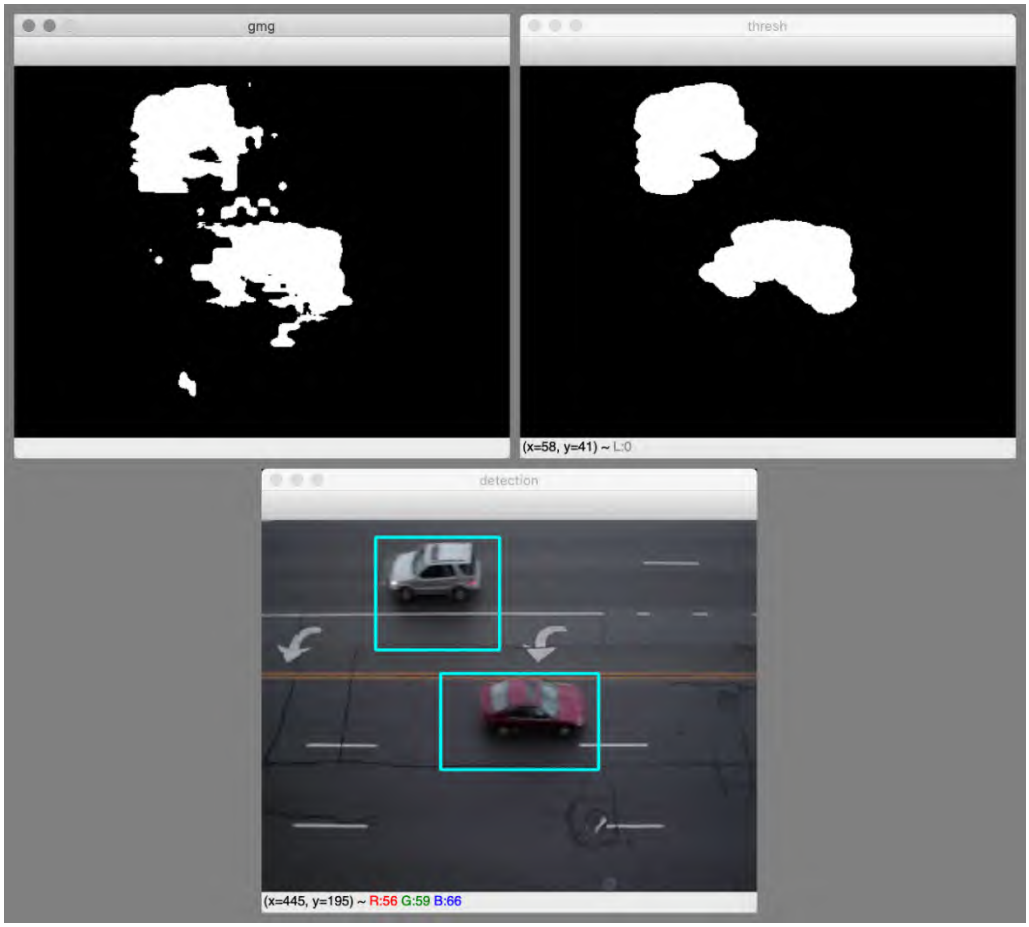


Figure 8.5: Moving cars, detected using a GMG background subtractor

When you are finished experimenting with background subtractors, let's proceed to examine other tracking techniques that rely on a template of an object we are trying to track, rather than a template of the background.

Tracking colorful objects using MeanShift and CamShift

We have seen that background subtraction can be an effective technique for detecting moving objects; however, we know that it has some inherent limitations. Notably, it assumes that the current background can be predicted based on past frames. This assumption is fragile. For example, if the camera moves, the entire background model could suddenly become outdated. Thus, in a robust tracking system, it is important to build some kind of model of foreground objects rather than just the background.

We saw various ways of detecting objects in *Chapter 5, Detecting and Recognizing Faces*, *Chapter 6, Retrieving Images and Searching Using Image Descriptors*, and *Chapter 7, Building Custom Object Detectors*. For object *detection*, we favored algorithms that could deal with a lot of variation within a class of objects, so that our car detector was not too particular about what shape or color of car it would detect. For object *tracking*, our needs are a bit different. If we were tracking cars, we would want a different model for each car in the scene so that a red car and a blue car would not get mixed up. We would want to track the motion of each car separately.

Once we have detected a moving object (by background subtraction or other means), we would like to describe the object in a way that distinguishes it from other moving objects. In this way, we can continue to identify and track the object even if it crosses paths with another moving object. A **color histogram** may serve as a sufficiently unique description. Essentially, an object's color histogram is an estimate of the probability distribution of pixel colors in the object. For example, the histogram could indicate that each pixel in the object is 10% likely to be blue. The histogram is based on the actual colors observed in the object's region of a reference image. For example, the reference image could be the video frame in which we first detected the moving object.

Compared to other ways of describing an object, a color histogram has some properties that are particularly appealing in the context of motion tracking. The histogram serves as a lookup table that directly maps pixel values to probabilities, so it enables us to use every pixel as a feature, at a low computational cost. In this way, we can afford to perform tracking with very fine spatial resolution in real time. To find the most likely location of an object that we are tracking, we just have to find the region of interest where the pixel values map to the maximum probability, according to the histogram.

Naturally, this approach is leveraged by an algorithm with a catchy name: MeanShift. For each frame in a video, the MeanShift algorithm performs tracking iteratively by computing a centroid based on probability values in the current tracking rectangle, shifting the rectangle's center to this centroid, recomputing the centroid based on values in the new rectangle, shifting the rectangle again, and so on. This process continues until **convergence** is achieved (meaning that the centroid ceases to move or nearly ceases to move) or until a maximum number of iterations is reached. Essentially, MeanShift is a clustering algorithm with applications that extend beyond computer vision. The algorithm was first described in a paper entitled *The estimation of the gradient of a density function, with applications in pattern recognition* (IEEE Transactions on Information Theory, 1975), by K. Fukunaga and L. Hostetler. The paper is available to IEEE subscribers at `https://ieeexplore.ieee.org/document/1055330`.

Before delving into a sample script, let's consider the type of tracking result we want to achieve with MeanShift, and let's learn more about OpenCV's functionality pertaining to color histograms.

Planning our MeanShift sample

For our first demonstration of MeanShift, we are not concerned with the approach to the initial detection of moving objects. We will use a naive approach that simply chooses the central part of the first video frame as our initial region of interest. (The user must ensure that an object of interest is initially located in the center of the video.) We will calculate a histogram of this initial region of interest. Then, in subsequent frames, we will use this histogram and the MeanShift algorithm to track the object.

Visually, the MeanShift demo will resemble many of the object detection samples that we have previously written. For every frame, we will draw a blue outline around the tracking rectangle, as shown here:



Figure 8.6: A toy phone with a distinctive lilac hue, tracked using histogram back-projection and MeanShift

Here, the toy phone has a lilac color that is not present in any other object in the scene. Thus, the phone has a distinctive histogram, making it easy to track. Next, let's consider how such a histogram is calculated and then used as a lookup table of probabilities.

Calculating and back-projecting color histograms

To calculate a color histogram, OpenCV provides a function called `cv2.calcHist`. To apply a histogram as a lookup table, OpenCV provides another function called `cv2.calcBackProject`. The latter operation is known as **histogram back-projection**, and it transforms a given image into a probability map based on a given histogram. Let's first visualize the output of these two functions, and then examine their parameters.

A histogram can use any color model, such as **blue-green-red** (**BGR**), **hue-saturation-value** (**HSV**), or grayscale. (For an introduction to color models, refer to *Chapter 3, Processing Images with OpenCV*, specifically the *Converting images between different color models* section.) For our samples, we will use the histograms of only the hue (H) channel of the HSV color model. The following diagram is a visualization of a hue histogram:

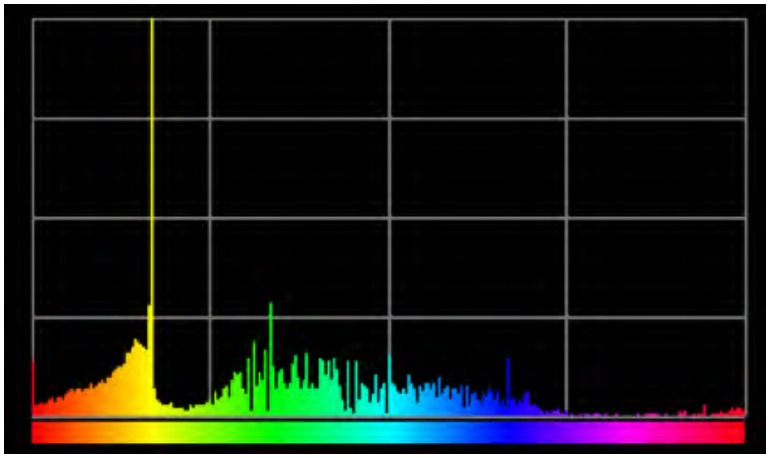


Figure 8.7: A visualization of a hue histogram. This is a sample of the output from an image-viewing application called DPEx (https://www.rysys.co.jp/dpex/en/)

On the x-axis of this plot, we have the hue, and on the y-axis, we have the hue's estimated probability or, in other words, the proportion of pixels in the image that have the given hue. If you are reading the e-book edition of this book, you will see that the plot is color-coded according to hue. From left to right, the plot progresses through the hues of the color wheel: red, yellow, green, cyan, blue, magenta, and, finally, back to red. This particular histogram seems to represent an object with a lot of yellow in it.

OpenCV represents H values with a range from 0 to 179, which, unfortunately, is a tad less precise than a typical HSV representation. Some other systems use a wider (more precise) range from 0 to 359 (like the degrees of a circle) or from 0 to 255 (to match the range of one byte).

Note

Some caution is required in interpreting hue histograms because pure black and pure white pixels do not have a meaningful hue; however, their hue is usually represented as 0 (red).

When we use `cv2.calcHist` to generate a hue histogram, it returns a 1D array that is conceptually similar to the preceding plot. Alternatively, depending on the parameters we provide, we could use `cv2.calcHist` to generate a histogram of a different channel or of two channels at once. In the latter case, `cv2.calcHist` would return a 2D array.

Once we have a histogram, we can back-project the histogram onto any image. `cv2.calcBackProject` produces a back-projection in the format of an 8-bit grayscale image, with pixel values that potentially range from 0 (indicating a low probability) to 255 (indicating a high probability), depending on how we scale the values. For example, consider the following pair of photographs, showing a back-projection and then a visualization of the MeanShift tracking result:

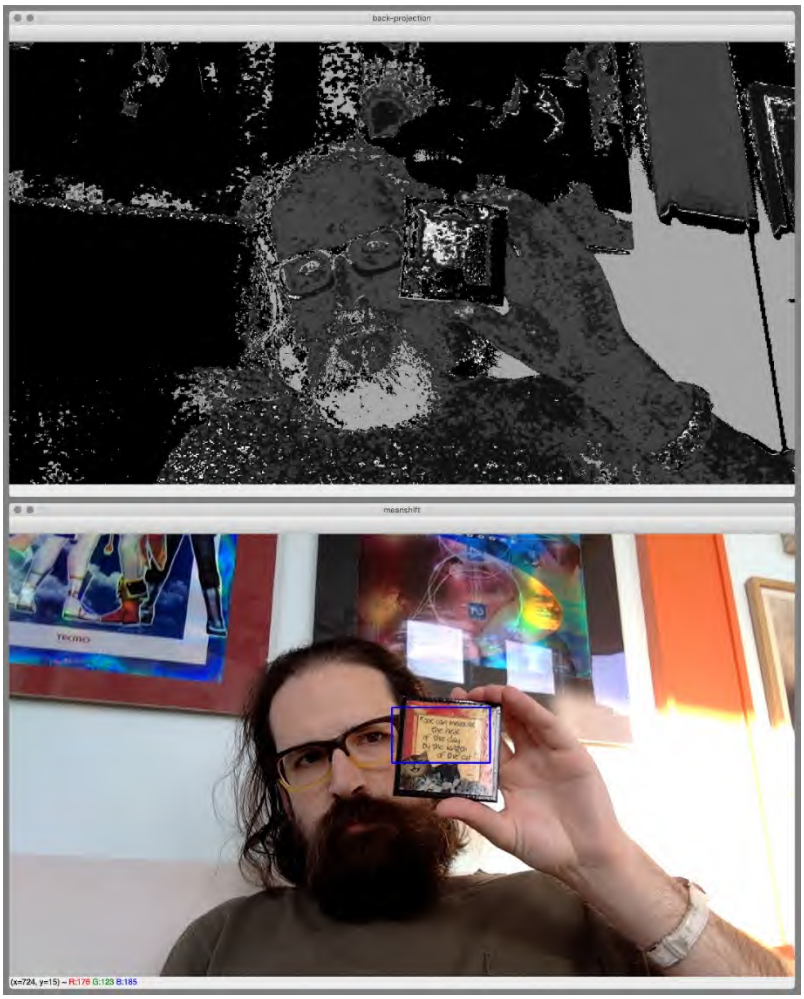


Figure 8.8: A coaster with distinctive yellow, red, and brown hues, tracked using histogram back-projection and MeanShift

Here, we are tracking a small object whose main colors are yellow, red, and brown. The back-projection is brightest in regions that are actually part of the object. The back-projection is also somewhat bright in other regions of similar color, such as Joseph Howse's brown beard, the yellow rims of his glasses, and the red border of one of the posters in the background.

Now that we have visualized the outputs of `cv2.calcHist` and `cv2.calcBackProject`, let's examine the parameters that these functions accept.

Understanding the parameters of cv2.calcHist

The `cv2.calcHist` function has the following signature:

```
calcHist(images, channels, mask, histSize,
         ranges[, hist=None[, accumulate=False]]) -> hist
```

The following table contains descriptions of the parameters (adapted from the official OpenCV documentation):

Parameter	Description
`Images`	This parameter is a list of one or more source images. They should all have the same bit depth (8-bit, 16-bit, or 32-bit) and the same size.
`Channels`	This parameter is a list of the indices of the channels used to compute the histogram. For example, `channels=[0]` means that only the first channel (that is, the channel with index `0`) is used to compute the histogram.
`Mask`	This parameter is a mask. If it is `None`, no masking is performed; every region of the images is used in the histogram calculation. If it is not `None`, then it must be an 8-bit array of the same size as each image in `images`. The mask's nonzero elements mark the regions of the images that should be used in the histogram calculation.
`histSize`	This parameter is a list of the number of histogram bins to use for each channel. The length of the `histSize` list must be the same as the length of the `channels` list. For example, if `channels=[0]` and `histSize=[180]`, the histogram has 180 bins for the first channel (and any other channels are unused).

`Ranges`	This parameter is a list that specifies the ranges of values to use for each channel. Each range is inclusive of the lower bound but exclusive of the upper bound. The length of the `ranges` list must be twice the length of the `channels` list. For example, if `channels=[0], histSize=[180]`, and `ranges=[0, 180]`, the histogram has 180 bins for the first channel, and these bins are based on values in the range from 0 to 179; in other words, there is one input value per bin.
`Hist`	This optional parameter is the output histogram. If it is `None` (the default), a new array will be returned as the output histogram.
`Accumulate`	This optional parameter is the `accumulate` flag. By default, it is `False`. If it is `True`, then the original content of `hist` is not cleared; instead, the new histogram is added to the original content of `hist`. This feature enables you to compute a single histogram from several lists of images, or to update the histogram over time.

Table 8.1: Parameters of cv2.calcHist

In our samples, we will calculate the hue histogram of a region of interest like so:

```
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])
```

Next, let's consider the parameters of `cv2.calcBackProject`.

Understanding the parameters of cv2.calcBackProject

The `cv2.calcBackProject` function has the following signature:

```
calcBackProject(images, channels, hist, ranges,
                scale[, dst=None]) -> dst
```

The following table contains descriptions of the parameters (adapted from the official OpenCV documentation):

Parameter	**Description**
`images`	This parameter is a list of one or more source images. They all should have the same bit depth (8-bit, 16-bit, or 32-bit) and the same size.

`channels`	This parameter must be the same as the `channels` parameter used in `calcHist`.
`hist`	This parameter is the histogram.
`ranges`	This parameter must be the same as the `ranges` parameter used in `calcHist`.
`scale`	This parameter is a scale factor. The back-projection is multiplied by this scale factor.
`dst`	This optional parameter is the output back-projection. If it is `None` (the default), a new array will be returned as the back-projection.

Table 8.2: Parameters of cv2.calcBackProject

In our samples, we will use code similar to the following line to back-project a hue histogram onto an HSV image:

```
back_proj = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)
```

Having examined the `cv2.calcHist` and `cv2.calcBackProject` functions in detail, let's now put them into action in a script that performs tracking with MeanShift.

Implementing the MeanShift example

Let's go sequentially through the implementation of our MeanShift example:

1. Like our basic background subtraction example, our MeanShift example begins by capturing (and discarding) several frames from a camera so that the autoexposure can adjust:

```
import cv2

cap = cv2.VideoCapture(0)

# Capture several frames to allow the camera's autoexposure to
# adjust.
for i in range(10):
    success, frame = cap.read()
if not success:
    exit(1)
```

2. By the 10^{th} frame, we assume that the exposure is good; therefore, we can extract an accurate histogram of a region of interest. The following code defines the bounds of the **region of interest** (**ROI**):

```
# Define an initial tracking window in the center of the frame.
frame_h, frame_w = frame.shape[:2]
w = frame_w//8
h = frame_h//8
x = frame_w//2 - w//2
y = frame_h//2 - h//2
track_window = (x, y, w, h)
```

3. Then, the following code selects the ROI's pixels and converts them to the HSV color space:

```
roi = frame[y:y+h, x:x+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
```

4. Next, we calculate the hue histogram of the ROI:

```
mask = None
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])
```

5. After the histogram is calculated, we normalize the values to a range from 0 to 255:

```
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
```

6. Remember that MeanShift performs a number of iterations before reaching convergence; however, this convergence is not assured. Thus, OpenCV allows us to specify the so-called termination criteria. Let's define the termination criteria as follows:

```
# Define the termination criteria:
# 10 iterations or convergence within 1-pixel radius.
term_crit = \
    (cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS, 10, 1)
```

Based on these criteria, MeanShift will stop calculating the centroid shift after 10 iterations (the **count** criterion) or when the shift is no longer larger than 1 pixel (the **epsilon** criterion). The combination of flags (`cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS`) indicates that we are using both of these criteria.

7. Now that we have calculated a histogram and defined MeanShift's termination criteria, let's start our usual loop in which we capture and process frames from the camera. With each frame, the first thing we do is convert it to the HSV color space:

```
success, frame = cap.read()
while success:

    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

8. Now that we have an HSV image, we can perform the long-awaited operation of histogram back-projection:

```
    back_proj = cv2.calcBackProject(
        [hsv], [0], roi_hist, [0, 180], 1)
```

9. The back-projection, the tracking window, and the termination criteria can be passed to `cv2.meanShift`, which is OpenCV's implementation of the MeanShift algorithm. Here is the function call:

```
    # Perform tracking with MeanShift.
    num_iters, track_window = cv2.meanShift(
        back_proj, track_window, term_crit)
```

Note

Note that MeanShift returns the number of iterations that it ran, as well as the new tracking window that it found. Optionally, we could compare the number of iterations to our termination criteria in order to determine whether the result converged. (If the actual number of iterations is less than the maximum, the result must have converged.)

10. Finally, we draw and show the updated tracking rectangle:

```
    # Draw the tracking window.
    x, y, w, h = track_window
    cv2.rectangle(
        frame, (x, y), (x+w, y+h), (255, 0, 0), 2)

    cv2.imshow('back-projection', back_proj)
    cv2.imshow('meanshift', frame)
```

That is the whole example. If you run the program, it should produce an output that is similar to the screenshots we saw earlier, in the *Calculating and back-projecting color histograms* section.

By now, you should have a good idea of how color histograms, back-projections, and MeanShift work. However, the preceding program (and MeanShift in general) has a limitation: the size of the window does not change with the size of the object in the frames being tracked.

Gary Bradski – one of the founders of the OpenCV project – published a paper in 1998 to improve the accuracy of MeanShift. He described a new algorithm called **Continuously Adaptive MeanShift** (**CAMShift** or **CamShift**), which is very similar to MeanShift but also adapts the size of the tracking window when MeanShift reaches convergence. Let's take a look at an example of CamShift next.

Using CamShift

Although CamShift is a more complex algorithm than MeanShift, OpenCV provides a very similar interface for the two algorithms. The main difference is that a call to `cv2.CamShift` returns a rectangle with a particular rotation that follows the rotation of the object being tracked. With just a few modifications to the preceding MeanShift example, we can instead use CamShift and draw a rotated tracking rectangle. All of the necessary changes are `highlighted` in the following excerpt:

```
import cv2
import numpy as np

# ... Initialize the tracking window and histogram as previously ...

success, frame = cap.read()
while success:

    # Perform back-projection of the HSV histogram onto the frame.
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

```
    back_proj = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)

    # Perform tracking with CamShift.
    rotated_rect, track_window = cv2.CamShift(
        back_proj, track_window, term_crit)

    # Draw the tracking window.
    box_points = cv2.boxPoints(rotated_rect)
    box_points = box_points.astype(int)
    cv2.polylines(frame, [box_points], True, (255, 0, 0), 2)

    cv2.imshow('back-projection', back_proj)
    cv2.imshow('camshift', frame)

    k = cv2.waitKey(1)
    if k == 27: # Escape
        break

    success, frame = cap.read()
```

The arguments to `cv2.CamShift` are unchanged; they have the same meanings and the same values as the arguments to `cv2.meanShift` in our previous example.

We use the `cv2.boxPoints` function to find the vertices of the rotated tracking rectangle. Then, we use the `cv2.polylines` function to draw the lines connecting these vertices. The following screenshot shows the result:

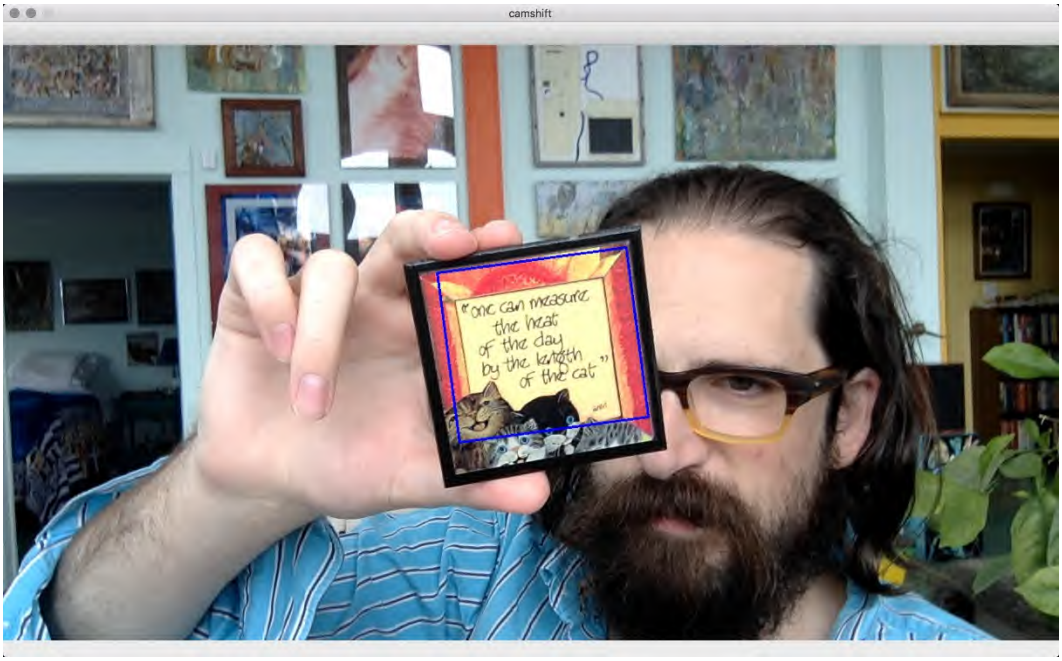


Figure 8.9: A coaster with distinctive yellow, red, and brown hues, tracked using histogram back-projection and CamShift

By now, you should be familiar with two families of tracking techniques. The first family uses background subtraction. The second uses histogram back-projection, combined with either MeanShift or CamShift. Now, let's meet the Kalman filter, which exemplifies a third family; it finds trends or, in other words, predicts future motion based on past motion.

Finding trends in motion using the Kalman filter

The Kalman filter is an algorithm developed mainly (but not exclusively) by Rudolf Kalman in the late 1950s. It has found practical applications in many fields, particularly navigation systems for all sorts of vehicles, from nuclear submarines to aircraft.

The Kalman filter operates recursively on a stream of noisy input data to produce a statistically optimal estimate of the underlying system state. In the context of computer vision, the Kalman filter can smooth the estimate of a tracked object's position.

Let's consider a simple example. Think of a small red ball on a table and imagine you have a camera pointing at the scene. You identify the ball as the subject to be tracked and flick it with your fingers. The ball will start rolling on the table in accordance with the laws of motion.

If the ball is rolling at a constant speed of 1 meter per second in a particular direction, it is easy to estimate where the ball will be in 1 second's time: it will be 1 meter away. The Kalman filter applies laws such as this to predict an object's position in the current video frame based on tracking results gathered in previous frames. The Kalman filter itself is not gathering these tracking results, but it is updating its model of the object's motion based on the tracking results derived from another algorithm – in our case, a visual tracking algorithm such as MeanShift. The tracking algorithm produces results that are (probably) noisy but (hopefully) **unbiased**; that is to say, its errors do not tend toward any particular direction. Conversely, it is the Kalman filter's role to smooth these results, to make them less noisy, at the risk of biasing the results by seeking a trend to fit an idealized model of motion.

Naturally, the Kalman filter cannot foresee new forces acting on the ball – such as a collision with a pencil lying on the table – but it can update its model of the ball's motion after the fact, when new tracking results deviate significantly from what the Kalman filter would have predicted. Through the use of the Kalman filter, we can obtain estimates that are more stable and more consistent with the laws of motion than the tracking results alone. This stabilization is usually a good thing, but be aware that the stabilized results may lag behind reality when a tracked object changes course.

Understanding the predict and update phases

From the preceding description, we gather that the Kalman filter's algorithm has two phases:

- Predict: In the first phase, the Kalman filter uses the covariance calculated up to the current point in time to estimate the object's new position.
- Update: In the second phase, the Kalman filter records the object's position and adjusts the covariance for the next cycle of calculations.

The update phase is – in OpenCV's terms – a **correction**. Thus, OpenCV provides a `cv2.KalmanFilter` class with the following methods:

```
predict([, control]) -> retval
correct(measurement) -> retval
```

For the purpose of smoothly tracking objects, we will call the `predict` method to estimate the position of an object, and then use the `correct` method to instruct the Kalman filter to adjust its calculations based on a new tracking result from another algorithm such as MeanShift.

However, before we combine the Kalman filter with a computer vision algorithm, let's examine how it performs with position data from a simple motion sensor.

Tracking a mouse cursor

Motion sensors have been commonplace in user interfaces for a long time. A computer's mouse senses its own motion relative to a surface such as a table. The mouse is a real, physical object, so it is reasonable to apply the laws of motion in order to predict changes in mouse coordinates. We are going to do exactly this as a demo of the Kalman filter.

Our demo will implement the following sequence of operations:

1. Start by initializing a black image and a Kalman filter. Show the black image in a window.
2. Every time the windowed application processes input events, use the Kalman filter to predict the mouse's position. Then, correct the Kalman filter's model based on the actual mouse coordinates. On top of the black image, draw a red line from the old predicted position to the new predicted position, and then draw a green line from the old actual position to the new actual position. Show the drawing in the window.
3. When the user hits the *Esc* key, exit and save the drawing to a file.

To begin the script, the following code initializes an 800 x 800 black image:

```
import cv2
import numpy as np

# Create a black image.
img = np.zeros((800, 800, 3), np.uint8)
```

Now, let's initialize the Kalman filter:

```
# Initialize the Kalman filter.
kalman = cv2.KalmanFilter(4, 2)
kalman.measurementMatrix = np.array(
    [[1, 0, 0, 0],
     [0, 1, 0, 0]], np.float32)
kalman.transitionMatrix = np.array(
    [[1, 0, 1, 0],
     [0, 1, 0, 1],
     [0, 0, 1, 0],
     [0, 0, 0, 1]], np.float32)
kalman.processNoiseCov = np.array(
```

```
    [[1, 0, 0, 0],
     [0, 1, 0, 0],
     [0, 0, 1, 0],
     [0, 0, 0, 1]], np.float32) * 0.03
```

Based on the preceding initialization, our Kalman filter will track a 2D object's position and velocity. We will take a deeper look at the process of initializing a Kalman filter in *Chapter 9, Camera Models and Augmented Reality*, where we will track a 3D object's position, velocity, acceleration, rotation, angular velocity, and angular acceleration. For now, let's just take note of the two parameters in `cv2.KalmanFilter(4, 2)`. The first parameter is the number of variables tracked (or predicted) by the Kalman filter, in this case, `4`: the *x* position, the *y* position, the *x* velocity, and the *y* velocity. The second parameter is the number of variables provided to the Kalman filter as measurements, in this case, `2`: the *x* position and the *y* position. We also initialize several matrices that describe the relationships among all these variables.

Having initialized the image and the Kalman filter, we also have to declare variables to hold the actual (measured) and predicted mouse coordinates. Initially, we have no coordinates, so we will assign `None` to these variables:

```
last_measurement = None
last_prediction = None
```

Then, we declare a callback function that handles mouse movement. This function is going to update the state of the Kalman filter, and draw a visualization of both the unfiltered mouse movement and the Kalman-filtered mouse movement. The first time we receive mouse coordinates, we initialize the Kalman filter's state so that its initial prediction is the same as the actual initial mouse coordinates. (If we did not do this, the Kalman filter would assume that the initial mouse position was `(0, 0)`.) Subsequently, whenever we receive new mouse coordinates, we correct the Kalman filter with the current measurement, calculate the Kalman prediction, and, finally, draw two lines: a green line from the last measurement to the current measurement and a red line from the last prediction to the current prediction. Here is the callback function's implementation:

```
def on_mouse_moved(event, x, y, flags, param):
    global img, kalman, last_measurement, last_prediction

    measurement = np.array([[x], [y]], np.float32)
    if last_measurement is None:
        # This is the first measurement.
        # Update the Kalman filter's state to match the measurement.
```

```
            kalman.statePre = np.array(
                [[x], [y], [0], [0]], np.float32)
            kalman.statePost = np.array(
                [[x], [y], [0], [0]], np.float32)
            measurement = measurement.squeeze()
            prediction = measurement
        else:
            kalman.correct(measurement)
            measurement = measurement.squeeze()

            prediction = kalman.predict()  # Gets a reference, not a copy
            prediction = prediction.squeeze()

            # Trace the path of the measurement in green.
            cv2.line(img,
                     (int(last_measurement[0]), int(last_measurement[1])),
                     (int(measurement[0]), int(measurement[1])), (0, 255, 0))

            # Trace the path of the prediction in red.
            cv2.line(img,
                     (int(last_prediction[0]), int(last_prediction[1])),
                     (int(prediction[0]), int(prediction[1])), (0, 0, 255))

        last_prediction = prediction.copy()
        last_measurement = measurement
```

The next step is to initialize the window and pass our callback function to the `cv2.setMouseCallback` function:

```
cv2.namedWindow('kalman_tracker')
cv2.setMouseCallback('kalman_tracker', on_mouse_moved)
```

Since most of the program's logic is in the mouse callback, the implementation of the main loop is simple. We just continually show the updated image until the user hits the *Esc* key:

```
while True:
    cv2.imshow('kalman_tracker', img)
    k = cv2.waitKey(1)
    if k == 27:  # Escape
```

```
            cv2.imwrite('kalman.png', img)
            break
```

Run the program and move your mouse around. If you make a sudden turn at high speed, you will notice that the prediction line (in red) will trace a wider curve than the measurement line (in green). This is because the prediction is following the momentum of the mouse's movement up to that time. Here is a sample result:

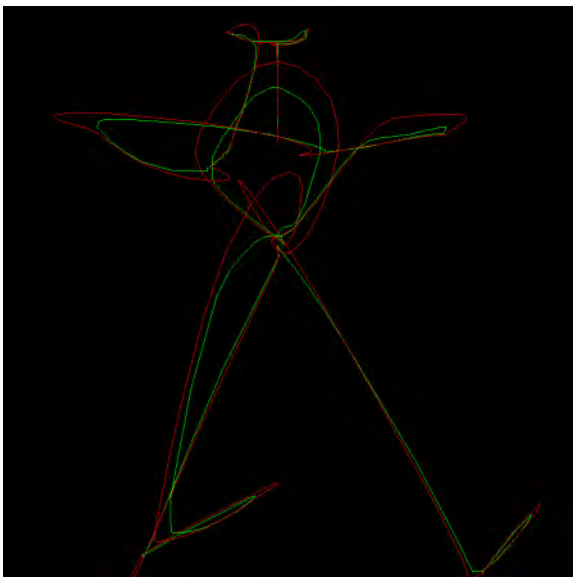


Figure 8.10: A drawing based on the movement of a computer mouse; the green line is actual movement, and the red line is a Kalman filter's prediction of the movement

Perhaps the preceding diagram will give us inspiration for our next sample application, in which we track pedestrians.

Tracking pedestrians

Up to this point, we have familiarized ourselves with the concepts of motion detection, object detection, and object tracking. You are probably anxious to put this newfound knowledge to good use in a real-life scenario. Let's do just that by tracking pedestrians in a video from a surveillance camera.

Note

You can find a surveillance video inside the OpenCV repository at `samples/data/vtest.avi`. A copy of this video is located inside this book's GitHub repository at `videos/pedestrians.avi`.

Let's lay out a plan and then implement the application!

Planning the flow of the application

The application will adhere to the following logic:

1. Capture frames from a video file.
2. Use the first 20 frames to populate the history of a background subtractor.
3. Based on background subtraction, use the 21st frame to identify moving foreground objects. We will treat these as pedestrians. For each pedestrian, assign an ID and an initial tracking window, and then calculate a histogram.
4. For each subsequent frame, track each pedestrian using a Kalman filter and MeanShift.

If this were a real-world application, you would probably store a record of each pedestrian's route through the scene so that a user could analyze it later. However, this type of record-keeping is beyond the remit of this example.

Additionally, in a real-world application, you would make sure to identify new pedestrians entering the scene; however, for now, we will focus on tracking only those objects that are in the scene near the start of the video.

You will find the code for this application inside the book's GitHub repository at `chapter08/track_pedestrians.py`. Before examining the implementation, let's briefly digress to consider programming paradigms and how they relate to our use of OpenCV.

Comparing the object-oriented and functional paradigms

Although most programmers are either familiar with (or work on a constant basis with) OOP, another paradigm called FP has, for many years, been gaining support among programmers who favor a purer mathematical foundation.

Note

The works of Samuel Howse demonstrate a specification of a programming language with a pure mathematical foundation. You can find his doctoral thesis, *NummSquared 2006a0 Explained*, at `https://nummist.com/poohbist/NummSquared2006a0Explained.pdf`, and his paper, *NummSquared: a New Foundation for Formal Methods*, at `https://nummist.com/poohbist/NummSquaredFormalMethods.pdf`.

FP treats programs as the evaluation of mathematical functions, allows functions to return functions, and permits functions to be passed as arguments to other functions. The strength of FP resides not only in what it can do, but also in what it can avoid, or aims at avoiding: for example, side effects and changing states. If the topic of FP has sparked an interest, make sure that you take a look at languages such as **Haskell**, **Clojure**, or **Meta Language** (**ML**).

Note

So, what is a side effect in programming terms? A function is said to have side effects if it produces any changes that are accessible outside its local scope, except for its return values. Python, like many other languages, is susceptible to side effects because it allows you to gain access to member variables and global variables – and, sometimes, this access can be accidental!

In languages that are not purely functional, a function's output can vary even when we pass it the same parameters repeatedly. For example, if a function takes an object as an argument, and the computation relies on the internal state of that object, the function will return different results according to changes in the object's state. This is a common occurrence in OOP with languages such as Python and C++.

So, why this digression? Well, it is a good occasion to consider the paradigms used in our own samples and in OpenCV, and how they differ from a purer mathematical approach. Throughout this book, we have often used global variables or object-oriented classes with member variables. The next program is another example of OOP. OpenCV, too, contains many functions with side effects and many object-oriented classes.

For example, any OpenCV drawing function, such as `cv2.rectangle` or `cv2.circle`, modifies the image that we pass to it as an argument. This approach contravenes one of FP's cardinal rules: avoid side effects and changes in state.

As a brief exercise, let's wrap `cv2.rectangle` in another Python function to perform a drawing in an FP style, without any side effects. The following implementation relies on making a copy of the input image, rather than modifying the original:

```
def draw_rect(img, top_left, bottom_right, color,
              thickness, fill=cv2.LINE_AA):
    new_img = img.copy()
    cv2.rectangle(new_img, top_left, bottom_right, color, thickness, fill)
    return new_img
```

This approach – while computationally more expensive due to the `copy` operation – allows for code such as the following to run without side effects:

```
frame = camera.read()
frame_with_rect = draw_rect(
    frame, (0, 0), (10, 10), (0, 255, 0), 1)
```

Here, `frame` and `frame_with_rect` are references to two different NumPy arrays containing different values. If we had used `cv2.rectangle` instead of our FP-inspired `draw_rect` wrapper, then `frame` and `frame_with_rect` would have been references to one and the same NumPy array (containing a drawing of a rectangle on top of the original image).

To conclude this digression, let's note that a variety of programming languages and paradigms can be applied successfully to computer vision problems. It is useful to know about multiple languages and paradigms so that you can choose the right tool for a given job.

Now, let's get back to our program and explore the implementation of a surveillance application, tracking moving objects in a video.

Implementing the Pedestrian class

The nature of the Kalman filter provides the main rationale for creating a `Pedestrian` class. The Kalman filter can predict the position of an object based on historical observations and can correct the prediction based on the actual data, but it can only do this for one object. As a consequence, we need one Kalman filter per object tracked.

Each `Pedestrian` object will act as a holder for a Kalman filter, a color histogram (calculated on the first detection of the object and used as a reference for the subsequent frames), and a tracking window, which will be used by the MeanShift algorithm. Furthermore, each pedestrian has an ID, which we will display so that we can easily distinguish between all of the pedestrians being tracked.

This time, instead of using just a hue histogram, we will use a 2D histogram of hue and value. Hue alone is not a great choice for tracking pedestrians because we do not necessarily expect to find a wide range of rainbow hues in the human form or in typical streetwear. Of course, a pedestrian with purple hair or a neon orange jacket might have a very distinctive hue histogram, yet many pedestrians would not. The addition of a value (or brightness) dimension may help to further distinguish pedestrians from each other and from typical backgrounds such as gray cement.

Let's proceed sequentially through the class's implementation:

1. As arguments, the `Pedestrian` class's constructor takes an ID, an initial frame in HSV format, and an initial tracking window. Here are the declarations of the class and its constructor:

```
import cv2
import numpy as np

class Pedestrian():
    """A tracked pedestrian with a state including an ID, tracking
    window, histogram, and Kalman filter.
    """

    def __init__(self, id, hsv_frame, track_window):
```

2. To begin the constructor's implementation, we define variables for the ID, the tracking window, and the MeanShift algorithm's termination criteria:

```
        self.id = id

        self.track_window = track_window
        self.term_crit = \
            (cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS, 10, 1)
```

3. We proceed by creating a normalized hue and value histogram of the region of interest in the initial HSV image:

```
        # Initialize the histogram.
        x, y, w, h = track_window
        roi = hsv_frame[y:y+h, x:x+w]
        roi_hist = cv2.calcHist([roi], [0, 2], None, [15, 16],
                                [0, 180, 0, 256])
```

```
        self.roi_hist = cv2.normalize(roi_hist, roi_hist, 0, 255,
                                      cv2.NORM_MINMAX)
```

4. Then, we initialize the Kalman filter:

```
        # Initialize the Kalman filter.
        self.kalman = cv2.KalmanFilter(4, 2)
        self.kalman.measurementMatrix = np.array(
            [[1, 0, 0, 0],
             [0, 1, 0, 0]], np.float32)
        self.kalman.transitionMatrix = np.array(
            [[1, 0, 1, 0],
             [0, 1, 0, 1],
             [0, 0, 1, 0],
             [0, 0, 0, 1]], np.float32)
        self.kalman.processNoiseCov = np.array(
            [[1, 0, 0, 0],
             [0, 1, 0, 0],
             [0, 0, 1, 0],
             [0, 0, 0, 1]], np.float32) * 0.03
        cx = x+w/2
        cy = y+h/2
        self.kalman.statePre = np.array(
            [[cx], [cy], [0], [0]], np.float32)
        self.kalman.statePost = np.array(
            [[cx], [cy], [0], [0]], np.float32)
```

Like in our mouse tracking example, we are configuring the Kalman filter to predict the motion of a 2D point. As the initial point, we use the center of the initial tracking window. This concludes the implementation of the constructor.

5. The `Pedestrian` class also has an `update` method, which we will call once per frame. As arguments, the `update` method takes a BGR frame (to use when we draw a visualization of the tracking result) and an HSV version of the same frame (to use for histogram back-projection). The implementation of the `update` method begins with familiar code for histogram back-projection and MeanShift, as displayed in the following lines:

```
def update(self, frame, hsv_frame):

    back_proj = cv2.calcBackProject(
```

```
            [hsv_frame], [0, 2], self.roi_hist, [0, 180, 0, 256], 1)

        ret, self.track_window = cv2.meanShift(
            back_proj, self.track_window, self.term_crit)
        x, y, w, h = self.track_window
        center = np.array("[[x+w/2], [y+h/2]]", np.float32)
```

6. Note that we have extracted the tracking window's center coordinates because we want to perform Kalman filtering on them. We proceed to do exactly this, and then we update the tracking window so that it is centered on the corrected coordinates:

```
        prediction = self.kalman.predict().squeeze()
        estimate = self.kalman.correct(center)
        center_offset = estimate[:,0][:2] - center
        self.track_window = (x + int(center_offset[0][0]),
                             y + int(center_offset[1][0]), w, h)
        x, y, w, h = self.track_window
```

7. To conclude the `update` method, we draw the Kalman filter's prediction as a blue circle, the corrected tracking window as a cyan rectangle, and the pedestrian's ID as blue text above the rectangle:

```
        # Draw the predicted center position as a circle.
        cv2.circle(frame, (int(prediction[0]), int(prediction[1])),
                   4, (255, 0, 0), -1)

        # Draw the corrected tracking window as a rectangle.
        cv2.rectangle(frame, (x,y), (x+w, y+h), (255, 255, 0), 2)

        # Draw the ID above the rectangle.
        cv2.putText(frame, 'ID: %d' % self.id, (x, y-5),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 0, 0),
                    1, cv2.LINE_AA)
```

This is all the functionality and data that we need to associate with an individual pedestrian. Next, we need to implement a program that provides the necessary video frames to create and update `Pedestrian` objects.

Implementing the main function

Now that we have a `Pedestrian` class to maintain data about the tracking of each pedestrian, let's implement our program's `main` function. We will look at the parts of the implementation sequentially:

1. We begin by loading a video file, initializing a background subtractor, and setting the background subtractor's history length (that is, the number of frames affecting the background model):

```
def main():

    cap = cv2.VideoCapture('../videos/pedestrians.avi')

    # Create the KNN background subtractor.
    bg_subtractor = cv2.createBackgroundSubtractorKNN()
    history_length = 20
    bg_subtractor.setHistory(history_length)
```

2. Then, we define morphology kernels:

```
    erode_kernel = cv2.getStructuringElement(
        cv2.MORPH_ELLIPSE, (3, 3))
    dilate_kernel = cv2.getStructuringElement(
        cv2.MORPH_ELLIPSE, (5, 7))
```

3. We define a list called `pedestrians`, which is initially empty. A little later, we will add `Pedestrian` objects to this list. We also set up a frame counter, which we will use to determine whether enough frames have elapsed to fill the background subtractor's history. Here are the relevant definitions of the variables:

```
    pedestrians = []
    num_history_frames_populated = 0
```

4. Now, we start a loop. At the start of each iteration, we try to read a video frame. If this fails (for instance, at the end of the video file), we exit the loop:

```
while True:
    grabbed, frame = cap.read()
    if not grabbed:
        break
```

5. Proceeding with the body of the loop, we update the background subtractor based on the newly captured frame. If the background subtractor's history is not yet full, we simply continue to the next iteration of the loop. Here is the relevant code:

```
    # Apply the KNN background subtractor.
    fg_mask = bg_subtractor.apply(frame)

    # Let the background subtractor build up a history.
    if num_history_frames_populated < history_length:
        num_history_frames_populated += 1
        continue
```

6. Once the background subtractor's history is full, we do more processing on each newly captured frame. Specifically, we apply the same approach we used with background subtractors earlier in this chapter: we perform thresholding, erosion, and dilation on the foreground mask, and then we detect contours, which might be moving objects:

```
    # Create the thresholded image.
    _, thresh = cv2.threshold(fg_mask, 127, 255,
                              cv2.THRESH_BINARY)
    cv2.erode(thresh, erode_kernel, thresh, iterations=2)
    cv2.dilate(thresh, dilate_kernel, thresh, iterations=2)

    # Detect contours in the thresholded image.
    contours, hier = cv2.findContours(
        thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

7. We also convert the frame to HSV format because we intend to use histograms in this format for MeanShift. The following line of code performs the conversion:

```
    hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

8. Once we have contours and an HSV version of the frame, we are ready to detect and track moving objects. We find and draw a bounding rectangle for each contour that is large enough to be a pedestrian. Moreover, if we have not yet populated the `pedestrians` list, we do so now by adding a new `Pedestrian` object based on each bounding rectangle (and the corresponding region of the HSV image). Here is the subloop that handles the contours in the manner we have just described:

```
        # Draw rectangles around large contours.
        # Also, if no pedestrians are being tracked yet, create some.
        should_initialize_pedestrians = len(pedestrians) == 0
        id = 0
        for c in contours:
            if cv2.contourArea(c) > 500:
                (x, y, w, h) = cv2.boundingRect(c)
                cv2.rectangle(frame, (x, y), (x+w, y+h),
                              (0, 255, 0), 1)
                if should_initialize_pedestrians:
                    pedestrians.append(
                        Pedestrian(id, hsv_frame,
                                   (x, y, w, h)))
            id += 1
```

9. By now, we have a list of pedestrians whom we are tracking. We call each `Pedestrian` object's `update` method, to which we pass the original BGR frame (for use in drawing) and the HSV frame (for use in tracking with MeanShift). Remember that each `Pedestrian` object is responsible for drawing its own information (text, the tracking rectangle, and the Kalman filter's prediction). Here is the subloop that updates the `pedestrians` list:

```
        # Update the tracking of each pedestrian.
        for pedestrian in pedestrians:
            pedestrian.update(frame, hsv_frame)
```

10. Finally, we display the tracking results in a window, and we allow the user to exit the program at any time by pressing the *Esc* key:

```
        cv2.imshow('Pedestrians Tracked', frame)

        k = cv2.waitKey(110)
```

```
            if k == 27:  # Escape
                break

if __name__ == "__main__":
    main()
```

There you have it: MeanShift working in tandem with the Kalman filter to track moving objects. All being well, you should see tracking results visualized in the following manner:



Figure 8.11: A visualization of contour detection results (thin green rectangle), Kalman-corrected MeanShift results (thick cyan rectangle), and the Kalman prediction (blue dot) in a complex tracker

In this cropped screenshot, the green rectangle with the thin border is the detected contour, the cyan rectangle with the thick border is the Kalman-corrected MeanShift tracking rectangle, and the blue dot is the center position predicted by the Kalman filter.

As usual, feel free to experiment with the script. You may want to adjust the parameters, try a MOG background subtractor instead of KNN, or try CamShift instead of MeanShift. These changes should affect just a few lines of code. When you have finished, we will consider other possible modifications that might have a larger effect on the structure of the script.

Considering the next steps

The preceding program can be expanded and improved in various ways, depending on the requirements of a particular application. Consider the following examples:

- You could remove a `Pedestrian` object from the `pedestrians` list (and thereby destroy the `Pedestrian` object) if the Kalman filter predicts the pedestrian's position to be outside the frame.
- You could check whether each detected moving object corresponds to an existing `Pedestrian` instance in the `pedestrians` list, and, if not, add a new object to the list so that it will be tracked in subsequent frames.
- You could train a **support vector machine** (**SVM**) and use it to classify each moving object. By these means, you could establish whether or not the moving object is something you intend to track. For instance, a dog might enter the scene but your application might require the tracking of humans only. For more information on training an SVM, refer to *Chapter 7, Building Custom Object Detectors*.
- You could adopt a tracking technique that is based on machine learning, instead of using MeanShift (or CamShift) and a Kalman filter. OpenCV provides easy-to-use implementations of several machine learning trackers, with a common interface called `cv2.Tracker`.

Let's delve a little deeper into this last point. For this chapter's final exercise, we are going to modify our code, especially the `Pedestrian` class, to use MILTrack as the tracking algorithm.

Modifying the pedestrian tracker to use MIL

By its nature, a video of a moving object shows us many variations of that object. For example, in some frames, we might see a person's face and hair, yet in other frames, we might see only the person's hair because she has turned her head away. Later, we might see the person's upper face but not her mouth or chin because she is drinking from a coffee mug. To track this person's head, it would be helpful to build a model of various segments rather than just the whole. Over a series of several frames, the tracker could learn that the features of the eyes, nose, mouth, and hair really are part of this woman's head, yet the coffee mug is not part of her head; it moves separately.

This type of learning is, essentially, what MILTrack tries to do. MILTrack breaks up the contents of the tracking window and nearby regions into several **patches** or sample subsections; it detects features in each patch, and it learns which of these local clusters of features do (or do not) move together over time. Specifically, the algorithm uses a variant of Haar-like features, which we previously encountered in *Chapter 5, Detecting and Recognizing Faces*. MILTrack was first described by B. Babenko, M. Yang, and S. Belongie in their paper, *Visual Tracking with*

Online Multiple Instance Learning (CVPR, 2009). A copy can be downloaded from `https://faculty.ucmerced.edu/mhyang/papers/cvpr09a.pdf`.

OpenCV implements MIL in a class called (not surprisingly) `cv2.TrackerMIL`. To create an instance of this class, we can call the `cv2.TrackerMIL_create` function, which has the following signature:

```
TrackerMIL_create([, parameters]) -> retval
```

The optional `parameters` argument is an instance of `cv2.TrackerMIL_Params`, which has the following properties:

- `featureSetNumFeatures`: The maximum number of Haar features to detect in each patch. The default is 250.
- `samplerInitInRadius`: The pixel radius, beyond the tracking window, to search for positive patches during initialization. The default is 3.0.
- `samplerInitMaxNegNum`: The maximum number of negative patches, at the end of initialization. The default is 65.
- `samplerSearchWinSize`: The pixel radius, beyond the tracking window, to search for negative patches. The same value is used during initialization and updates. The default is 25.0.
- `samplerTrackInRadius`: The pixel radius, beyond the tracking window, to search for positive patches during updates. The default is 4.0.
- `samplerTrackMaxNegNum`: The maximum number of negative patches, at the end of each update. The default is 65.
- `samplerTrackMaxPosNum`: The maximum number of positive patches, at the end of each update. The default is 100,000.

After creating an instance of `TrackerMIL`, we can treat it as a black box; it does not expose any further parameters that are specific to MIL. To use `TrackerMIL`, we simply pass an initial frame and tracking window to its `init` method, and subsequently we pass each new frame to its `update` method, which returns an updated tracking window. The `init` and `update` methods are part of the `cv2.Tracker` interface, which `TrackerMIL` implements.

Let's put MIL into action by modifying our pedestrian tracker. First, make a copy of the script (the one containing the `Pedestrian` class and `main` method) so that later, you can do a before-and-after comparison of the tracking results. Then, modify the constructor of our `Pedestrian` class to remove everything relating to histogram back-projection, CamShift, and the Kalman

filter; instead, just create and initialize a `TrackerMIL`. The changes are `highlighted` in the following block of code:

```
class Pedestrian():
    """A tracked pedestrian with a state including an ID and a
    MIL tracker.
    """

    def __init__(self, id, frame, track_window):
        self.id = id

        # Initialize the MIL tracker.
        tracker_params = cv2.TrackerMIL_Params()
        tracker_params.samplerInitMaxNegNum = 100
        tracker_params.samplerSearchWinSize = 30.0
        tracker_params.samplerTrackMaxNegNum = 100
        self.tracker = cv2.TrackerMIL_create(tracker_params)
        self.tracker.init(frame, track_window)
```

Note that we have modified the constructor's signature by removing the `hsv_frame` argument. HSV data was relevant to our previous example using histogram back-projection and CamShift; however, `TrackerMIL` expects the input to be either BGR or grayscale frames.

Similarly, the `Pedestrian` class's `update` function no longer has any reason to take an HSV-converted frame as an argument. The original BGR frame is the only thing we will need, and we will simply pass it on to the underlying `Tracker`'s `update` method, which returns an updated estimate of the pedestrian's bounding rectangle. The relevant changes to our code are `highlighted` in the following block:

```
    def update(self, frame):

        # Update the MIL tracker.
        ret, (x, y, w, h) = self.tracker.update(frame)

        # Draw the corrected tracking window as a cyan rectangle.
        cv2.rectangle(frame, (x,y), (x+w, y+h), (255, 255, 0), 2)

        # Draw the ID above the rectangle in blue text.
        cv2.putText(frame, 'ID: %d' % self.id, (x, y-5),
```

```
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 0, 0),
                    1, cv2.LINE_AA)
```

Note that OpenCV's design of a `Tracker` class, with an `init` function and an `update` function, fits very well into our existing object-oriented design of a class for a trackable object, the `Pedestrian`. By using these two similarly designed classes together, we have simplified the implementation of the `Pedestrian` class.

Next, the `main` function has very minimal changes. We no longer need to convert the frame to HSV, and a Pedestrian's constructor and `update` method no longer take an HSV frame as an argument. Note the `highlighted` changes in the following code:

```
        # Draw green rectangles around large contours.
        # Also, if no pedestrians are being tracked yet, create some.
        should_initialize_pedestrians = len(pedestrians) == 0
        id = 0
        for c in contours:
            if cv2.contourArea(c) > 500:
                (x, y, w, h) = cv2.boundingRect(c)
                cv2.rectangle(frame, (x, y), (x+w, y+h),
                              (0, 255, 0), 1)
                if should_initialize_pedestrians:
                    pedestrians.append(
                        Pedestrian(id, frame, (x, y, w, h)))
            id += 1

        # Update the tracking of each pedestrian.
        for pedestrian in pedestrians:
            pedestrian.update(frame)
```

Run the old and new versions of the script and see how the results compare. Afterward, you may wish to modify the `Pedestrian` class again to try some of the other machine learning trackers that are implemented by OpenCV. All you need to do is replace the call to `cv2.TrackerMIL_create` with one of the following alternatives:

- `cv2.TrackerCSRT_create`: This function returns a `Tracker` that implements the Channel and Spatial Reliability Tracker (CSRT) algorithm. For details of the algorithm, see the paper *Discriminative Correlation Filter with Channel and Spatial Reliability* (IJCV, 2018), by A. Lukežič, T. Vojíř, L. Zajc, J. Matas, and M. Kristan; it is available at `https://`

`openaccess.thecvf.com/content_cvpr_2017/papers/Lukezic_Discriminative_Correlation_Filter_CVPR_2017_paper.pdf`.

- `cv2.TrackerKCF_create`: This function returns a `Tracker` that implements the **Kernelized Correlation Filter** (**KCF**) algorithm. For details, see the paper *High-Speed Tracking with Kernelized Correlation Filters* (PAMI, 2015), by J. Henriques, R. Caseiro, P. Martins, and J. Batista; it is available at `https://www.robots.ox.ac.uk/~joao/publications/henriques_tpami2015.pdf`. Moreover, OpenCV's implementation extends the algorithm to support the kind of color features that are described in the paper Adaptive Color Attributes for Real-Time Visual Tracking (CVPR, 2014), by M. Danelljan, F. Khan, M. Felsberg and J. van de Weijer. The latter paper and its sample code are available at `http://www.cvl.isy.liu.se/research/objrec/visualtracking/colvistrack/index.html`.

> **Note**
>
> These other `Tracker*_create` functions also support an optional `parameters` argument, but with different fields than the ones in `TrackerMIL_Params`. Refer to the OpenCV documentation for details of the supported parameters in each case.

Whatever your needs, hopefully, this chapter has provided you with the necessary knowledge to build 2D tracking applications that satisfy your requirements.

Summary

This chapter has dealt with video analysis and, in particular, a selection of useful techniques for tracking objects.

We began by learning about background subtraction with a basic motion detection technique that calculates frame differences. Then, we moved on to more complex and efficient background subtraction algorithms – namely, MOG and KNN – which are implemented in OpenCV's `cv2.BackgroundSubtractor` class.

We then proceeded to explore the MeanShift and CamShift tracking algorithms, using color histograms and back-projections. We also familiarized ourselves with the Kalman filter and its usefulness in smoothing the results of a tracking algorithm. Then, we put all of our knowledge together in a sample surveillance application, which is capable of tracking pedestrians (or other moving objects) in a video.

Finally, we saw that we can easily replace one set of tracking techniques with another in our object-oriented implementation of a pedestrian tracker. We learned how to use OpenCV's

implementation of the MIL tracker and other advanced trackers that are based on machine learning.

By now, our foundation in OpenCV, computer vision, and machine learning is solidifying. We can look forward to several advanced topics in the next three chapters of this book. We will extend our knowledge of tracking into 3D space in *Chapter 9, Camera Models and Augmented Reality*, and *Chapter 10, Introduction to 3D Navigation, Odometry, and Point Clouds*. Then, we will tackle **artificial neural networks** (**ANNs**) and dive deeper into artificial intelligence in *Chapter 11, Introduction to Neural Networks with OpenCV*.

Part 3

Tackling Deeper Problems: 6DOF Tracking, DNNs, and Scaling Up

Throughout *Parts 1* and *2*, we focused on models and pipelines that present a relatively flat view of the world: objects have a position in 2D pixel coordinates and sometimes a depth measurement; we see these objects in a stream of images; we process them with models that have a clearly defined feature vector of a limited size. However, many real-world problems and modern computer vision techniques involve much higher dimensionality: objects moving and rotating with 6 degrees of freedom; parallel input streams from multiple devices or users; and neural networks with numerous layers having large feature vectors whose meaning becomes quite abstract. *Part 3* delves into the challenges of building and maintaining these deeper solutions. This part contains the following chapters:

- *Chapter 9*, *Camera Models and Augmented Reality*
- *Chapter 10*, *Introduction to 3D Navigation, Odometry, and Point Clouds*
- *Chapter 11*, *Introduction to Neural Networks with OpenCV*
- *Chapter 12*, *OpenCV Applications at Scale*

9

Camera Models and Augmented Reality

If you like geometry, photography, or 3D graphics, then this chapter's topics should especially appeal to you. We will learn about the relationship between 3D space and a 2D projection. We will model this relationship in terms of the basic optical parameters of a camera and lens. Finally, we will apply the same relationship to the task of drawing 3D shapes in an accurate perspective projection. Throughout all of this, we will integrate our previous knowledge of image matching and object tracking in order to track the 3D motion of a real-world object whose 2D projection is captured by a camera in real time.

On a practical level, we will build an augmented reality application that uses information about a camera, an object, and motion in order to superimpose 3D graphics on top of a tracked object in real time. To achieve this, we will conquer the following technical challenges:

- Modeling the parameters of a camera and lens
- Modeling a 3D object using 2D and 3D keypoints
- Detecting the object by matching keypoints
- Finding the object's 3D pose using the `cv2.solvePnPRansac` function
- Smoothing the 3D pose using a Kalman filter, while avoiding numeric problems (singularities) that are associated with some ways of representing rotations
- Drawing graphics on top of the object

Over the course of this chapter, you will acquire skills that will serve you well if you go on to build your own augmented reality engine or any other system that relies on 3D tracking, such as a robotic navigation system.

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Please refer back to *Chapter 1, Setting Up OpenCV,* for installation instructions.

The completed code and sample videos for this chapter can be found in this book's GitHub repository in the `chapter09` folder.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Note

This chapter's code contains excerpts from an open-source demo project called *Visualizing the Invisible*, by Joseph Howse (one of this book's authors). To learn more about this project, please visit its repository at `https://github.com/JoeHowse/VisualizingTheInvisible/`.

Understanding 3D image tracking and augmented reality

We have already solved problems involving image matching in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. Moreover, we have solved problems involving continuous tracking in *Chapter 8, Tracking Objects*. Therefore, we are familiar with many of the components of an image tracking system, though we have not yet tackled any 3D tracking problems.

So, what exactly is **3D tracking**? Well, it is the process of continually updating an estimate of an object's **pose** (its position and orientation) in a 3D space. Typically, the pose is expressed in terms of six variables: three variables to represent the object's 3D **translation** (that is, position) and the other three variables to represent its 3D rotation (that is, orientation).

Note

A more technical term for 3D tracking is **6DOF tracking** – that is, tracking with **6 degrees of freedom**, meaning the 6 variables we just mentioned. With any fewer than 6 variables, it would be impossible to adequately express the pose of a freely moving, freely turning object such as an airplane. Indeed, the practical problems of aeronautics and spaceflight have driven the adoption of 6DOF and higher-dimensional pose representations since the early 20th century.

There are several different ways of representing 3D rotation as three variables. Elsewhere, you might have encountered various kinds of Euler angle representations, which describe 3D rotation in terms of three separate 2D rotations around the *x*, *y*, and *z* axes in a particular order. OpenCV does not use Euler angles to represent 3D rotation; instead, it uses a representation called the **Rodrigues rotation vector**. Specifically, OpenCV uses the following six variables to represent the 6DOF pose:

1. **tx**: This is the object's translation along the *x* axis
2. **ty**: This is the object's translation along the *y* axis
3. **tz**: This is the object's translation along the *z* axis
4. **rx**: This is the first element of the object's Rodrigues rotation vector
5. **ry**: This is the second element of the object's Rodrigues rotation vector
6. **rz**: This is the third element of the object's Rodrigues rotation vector

Unfortunately, in the Rodrigues representation, there is no easy way to interpret *rx*, *ry*, and *rz* separately from each other. Taken together, as the vector *r*, they encode both an axis of rotation and an angle of rotation about this axis. Specifically, the following formulas define the relationship among the *r* vector; an angle, θ; a normalized axis vector, $\hat{r}$; and a 3x3 rotation matrix, *R*:

$$\theta = |r|$$

$$\hat{r} = \frac{r}{\theta}$$

$$R = (\cos\theta)I + (1 - \cos\theta)\hat{r}\hat{r}^T + (\sin\theta)\begin{bmatrix} 0 & -\hat{r}_z & \hat{r}_y \\ \hat{r}_z & 0 & -\hat{r}_x \\ -\hat{r}_y & \hat{r}_x & 0 \end{bmatrix}$$

For basic usage of OpenCV, we are not obliged to compute or interpret any of these variables directly. OpenCV provides functions that give us a Rodrigues rotation vector as a return value, and we can pass this rotation vector to other OpenCV functions as an argument – without ever needing to manipulate its contents for ourselves.

Unfortunately, if we want to perform custom filtering of the rotation, the Rodrigues vector is generally not an appropriate format because the vector's individual elements lack a meaningful scale. As shown in the formulas above, we need to look at all three elements together in order to understand the rotation's magnitude (the angle) and direction (the axis). On the other hand, if we look at a change in *rx* (or any element) alone, we cannot say whether this change might have resulted from a change in the rotation's magnitude or its direction. Thus, a Kalman filter cannot make reliable predictions about such elements individually.

Instead, we will convert the rotation into the more common Euler angle format, perform Kalman filtering on it, and then convert it back to the Rodrigues representation. We will cover the details of these conversions later. For now, let's just note that the **Euler rotation vector** represents a 3D rotation by the following three elements:

1. **Pitch or attitude**: This is the object's rotation around its local x axis
2. **Yaw or heading**: This is the object's rotation around its local y axis
3. **Roll or bank**: This is the object's rotation around its local z axis

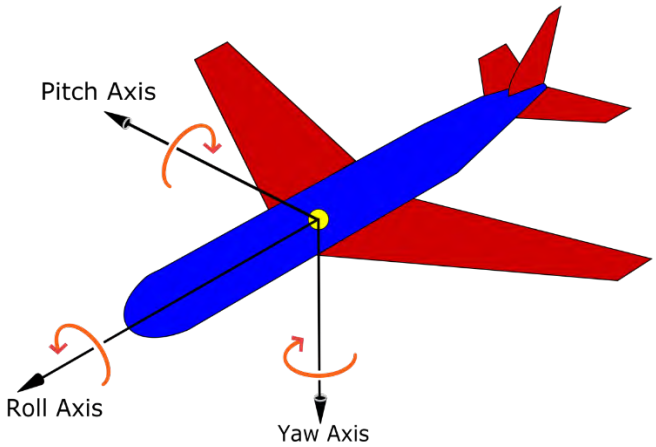


Figure 9.1: Pitch, yaw, and roll axes of an airplane, according to the DIN 9300 aerospace standard (Image credit: Wikimedia users Auawise and Jrvz. Licensed under CC-BY-SA-3.0.)

We will follow the **yaw-pitch-roll (YPR) Tait-Bryan convention**, whereby yaw is applied first, then pitch, and finally roll. The order matters because each element of the Euler rotation is relative to an object-local axis, which has already been transformed by any previously applied element of the Euler rotation. For example, in the YPR convention, the result of the yaw rotation determines the local axis of the pitch rotation, and the cumulative result of the yaw and pitch rotations determines the local axis of the roll rotation.

Note

The YPR convention refers to the order in which the rotations are applied, not the order in which they are listed in the vector. Despite the YPR convention, the order in the Euler vector is [*pitch*, *yaw*, *roll*].

Euler angles in general, and the YPR Tait-Bryan convention in particular, originate in the field of aeronautics. Imagine that we are describing the 3D rotation of an airplane flying within the atmosphere of Earth. Here are a few sample values that may help us understand the Euler vector format, [*pitch*, *yaw*, *roll*]:

- [0, 0, 0]: The airplane is level and its nose (its object-local *+z* axis) is pointing due North.
- [0, 90°, 0]: The airplane is level and its nose is pointing due East.
- [0, 90°, 90°]: The airplane's nose is pointing due East. Also, the airplane is banked 90° to its right; in other words, the right wing (its object-local *+x* axis) is pointing toward the Earth's surface.
- [45°, 90°, 0]: The airplane is heading East and climbing at a 45° angle.
- [90°, 0, 0]: The airplane is climbing vertically and its right wing is pointing due East.
- [90°, 90°, 0]: The airplane is climbing vertically and its right wing is pointing due South.
- [90°, 0, 270°]: Again, the airplane is climbing vertically and its right wing is pointing due South.
- [90°, 180°, 90°]: Yet again, the airplane is climbing vertically and its right wing is pointing due South!

As the last three examples show, there are cases where the same real-world rotation has multiple – indeed, infinite – possible representations in Euler angle format. These cases are called **singularities** and they occur where the second step of the rotation – pitch in the case of the YPR convention – is an angle of 90° or 270°. Here, with YPR, a pitch of 90° or 270° transforms the airplane's local *z* axis to be a vertical axis, so the roll will be a rotation around a vertical axis – but wait! The yaw's axis of rotation was vertical too! We are now at a singularity, where the distinction between two of our parameters has collapsed; a change in yaw would produce the same effect as a change in roll. Engineers use the term **gimbal lock** to describe a variety of practical problems that can arise from the poor handling of such singularities.

When we apply the Kalman filter to Euler angles, we will take care to steer the results away from divergent representations near singularities because such numeric instabilities would spoil our efforts to apply a smoothing filter!

Note

Some of the limitations of the Rodrigues vector and the Euler vector can be resolved by more complex, alternative formats that have at least four dimensions instead of three. The most famous of these is the **quaternion**, which uses imaginary numbers to represent a 3D rotation as a point on a four-dimensional hypersphere. For a discussion

> of how a Kalman filter can apply to quaternions and other alternative formats, see the article *Kalman Filtering for Attitude Estimation with Quaternions and Concepts from Manifold Theory*, by Pablo Bernal-Polo and Humberto Martínez-Barberá; it is available at `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6339217/`.

Now, let's say a few words about the relative nature of coordinates. For our purposes (and, indeed, for many problems in computer vision), the camera is the origin of the 3D coordinate system. Therefore, in any given frame, the camera's current *tx*, *ty*, *tz*, *rx*, *ry*, and *rz* values (using the Rodrigues vector representation of rotation) are defined to be [0, 0, 0, 0, 0, 1]. This means zero translation, and zero rotation around the *+z* axis (though any axis of rotation would produce the same result in this case, since the angle of rotation is 0). We will endeavor to track other objects relative to the camera's current pose.

Of course, for our edification, we will want to visualize the 3D tracking results. This brings us into the territory of **augmented reality** (**AR**). Broadly speaking, AR is the process of continually tracking relationships between real-world objects and applying these relationships to virtual objects, in such a way that a user perceives the virtual objects as being anchored to something in the real world. Typically, visual AR is based on relationships in terms of 3D space and perspective projection. Indeed, our case is typical; we want to visualize a 3D tracking result by drawing a projection of some 3D graphics atop the object we tracked in the frame.

We will return to the concept of perspective projection in a few moments. Meanwhile, let's see an overview of a typical set of steps involved in 3D image tracking and visual AR:

1. Define the parameters of the camera and lens. We will introduce this topic in this chapter.
2. Initialize a Kalman filter that we will use to stabilize the 6DOF tracking results. For more information about Kalman filtering, refer back to *Chapter 8, Tracking Objects*.
3. Choose a reference image, representing the surface of the object we want to track. For our demo, the object will be a plane, such as a piece of paper on which the image is printed.
4. Create a list of 3D points, representing the vertices of the object. The coordinates can be in any unit, such as meters, millimeters, or something arbitrary. For example, you could arbitrarily define 1 unit to be equal to the object's height.
5. Extract feature descriptors from the reference image. For 3D tracking applications, ORB is a popular choice of descriptor since it can be computed in real time, even on modest hardware such as a basic smartphone. Our demo will use ORB. For more information

about ORB, refer back to *Chapter 6, Retrieving Images and Searching Using Image Descriptors*.

6. Convert the feature descriptors from pixel coordinates to 3D coordinates, using the same mapping that we used in *step 4*.
7. Start capturing frames from the camera. For each frame, perform the following steps:
    1. Extract feature descriptors, and attempt to find good matches between the reference image and the frame. Our demo will use FLANN-based matching with a ratio test. For more information about these approaches for matching descriptors, refer back to *Chapter 6, Retrieving Images and Searching Using Image Descriptors*.
    2. If an insufficient number of good matches were found, continue to the next frame. Otherwise, proceed with the remaining steps.
    3. Attempt to find a good estimate of the tracked object's 6DOF pose based on the camera and lens parameters, the matches, and the 3D model of the reference object. For this, we will use the `cv2.solvePnPRansac` function.
    4. Apply the Kalman filter to stabilize the 6DOF pose so that it does not jitter too much from frame to frame.
    5. Based on the camera and lens parameters, and the 6DOF tracking results, draw a projection of some 3D graphics atop the tracked object in the frame.

> **Note**
>
> Here, our steps refer to a single reference image and a single object to track. However, the approach could be extended to deal with multiple reference images and multiple objects to track. For real-time use, optimization would be a part of the challenge.

Now, let's reorganize this overview a little, in order to get a more data-centric view of what we are going to do. We have two kinds of data: first, a reference image, and then a series of camera frames showing a scene. As usual, in OpenCV, our images start out as BGR data. The following diagram shows the pipeline of functions that we will apply to the reference image:

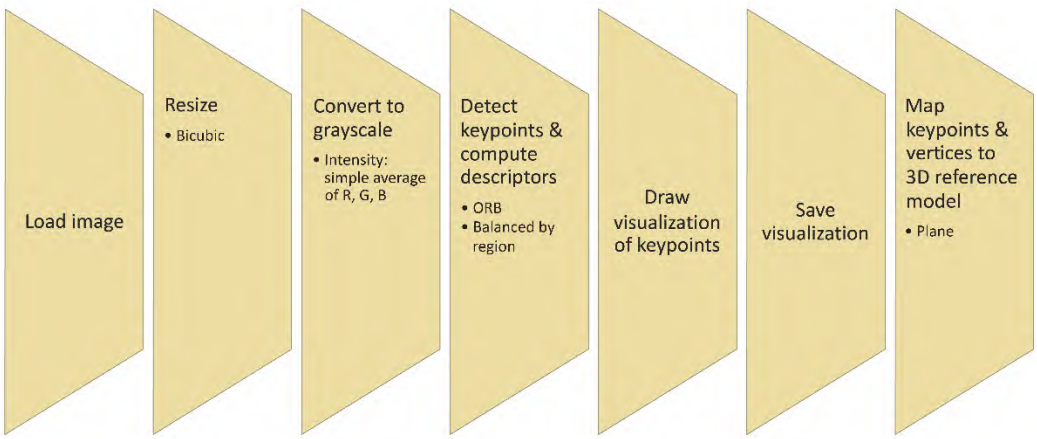


Figure 9.2a: The pipeline of functions that we will apply to the reference image. As we will use only one reference image, we will apply this pipeline only once.

Similarly, the following diagram shows the pipeline of functions that we will apply to the scene, that is, to each camera frame:



Figure 9.2b: The pipeline of functions that we will apply to the scene or, in other words, to the camera frames. As we will capture a stream of camera frames, we will apply this pipeline repeatedly.

We will take a complete walk through this pair of image processing pipelines later, in the section *Implementing the demo application*, but first, let's discuss two new concepts that appear in this outline: first, the parameters of the camera and lens, and second, the role of the mysterious function `cv2.solvePnPRansac`.

Understanding camera and lens parameters

Typically, when we capture an image, at least three objects are involved:

- The **subject** is something we want to capture in the image. Typically, it is an object that reflects light, and we want this object to appear in focus (sharp) in the image.
- The **lens** transmits light and focuses any reflected light from the **focal plane** onto the **image plane**. The focal plane is a circular slice of space that includes the subject (as defined previously). The image plane is a circular slice of space that includes the image sensor (as defined later). Typically, these planes are perpendicular to the lens's main (lengthwise) axis. The lens has an **optical center**, which is the point where incoming light from the focal plane converges before being projected back toward the image plane. The **focal distance** (that is, the distance between the optical center and the focal plane) varies depending on the distance between the optical center and the image plane. If we move the optical center closer to the image plane, the focal distance increases; conversely, if we move the optical center farther from the image plane, the focal distance decreases. Typically, in a camera system, the focus is adjusted by a mechanism that simply moves the lens back and forth. The **focal length** is defined as the distance between the optical center and the image plane when the focal distance is infinity.
- The **image sensor** is a photosensitive surface that receives light and records it as an image, in either an analog medium (such as film) or a digital medium. Typically, the image sensor is rectangular. Therefore, it does not cover all the extremities of the circular image plane. The image's diagonal **field of view** (**FOV**: the angular extent of the 3D space being imaged) bears a trigonometric relationship to the focal length, the image sensor's width, and the image sensor's height. We shall explore this relationship soon.

Here is a diagram to illustrate the preceding definitions:

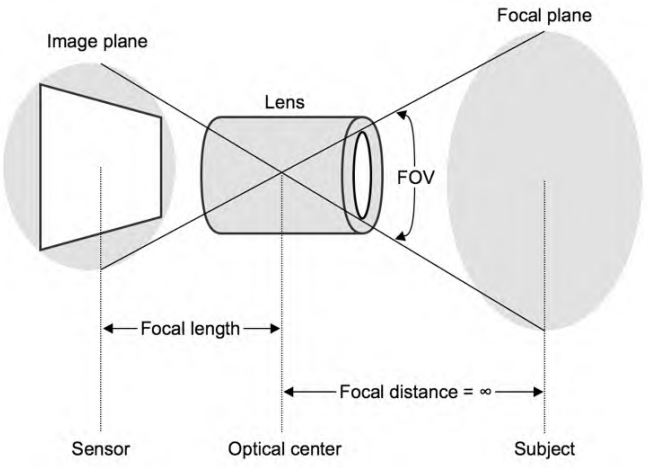


Figure 9.3: Geometric definitions of the basic parameters of a camera and lens

Note

For computer vision, we typically use a lens with a fixed focal length that is optimal for a given application. However, a lens can have a variable focal length; such a lens is called a **zoom lens**. **Zooming in** means increasing the focal length, while **zooming out** means decreasing the focal length. Mechanically, a zoom lens achieves this by moving the optical elements inside the lens.

Let's use the variable *f* to represent the focal length, and the variables (*cx*, *cy*) to represent the image sensor's center point within the image plane. OpenCV uses the following matrix, which it calls a **camera matrix**, to represent the basic parameters of a camera and lens:

f	0	c_x
0	f	c_y
0	0	1

Assuming that the image sensor is centered in the image plane (as it normally should be), we can calculate *cx* and *cy* based on the image sensor's width, *w*, and height, *h*, as follows:

$$c_x = \frac{w}{2}$$

$$c_y = \frac{h}{2}$$

If we know the diagonal FOV, θ, we can calculate the focal length using the following trigonometric formula:

$$f = \frac{\sqrt{w^2 + h^2}}{2\left(\tan\frac{\theta}{2}\right)}$$

Alternatively, if we do not know the diagonal FOV, but we know the horizontal FOV, ϕ, and the vertical FOV, ψ, we can calculate the focal length as follows:

$$f = \frac{\sqrt{w^2 + h^2}}{2\sqrt{\left(\tan\frac{\phi}{2}\right)^2 + \left(\tan\frac{\psi}{2}\right)^2}}$$

You might be wondering how we obtain values for any of these variables as starting points. Sometimes, the manufacturer of a camera or lens provides data on the sensor size, focal length, or FOV in the product's specification sheet. For example, the specification sheet might list the sensor size and focal length in millimeters and the FOV in degrees. However, if the specification sheet is not so informative, we have other ways of obtaining the necessary data. Importantly, the sensor size and focal length do not need to be expressed in real-world units such as millimeters. We can express them in arbitrary units, such as **pixel-equivalent units**.

You may well ask, what is a pixel-equivalent unit? Well, when we capture a frame from a camera, each pixel in the image corresponds to some region of the image sensor, and this region has a real-world width (and a real-world height, which is normally the same as the width, as pixels normally represent square regions). Therefore, if we are capturing frames with a resolution of 1280x720, we can say that the image sensor's width, *w*, is 1280 pixel-equivalent units and its height, *h*, is 720 pixel-equivalent units. These units are not comparable across different real-world sensor sizes or different resolutions; however, for a given camera and resolution, they allow us to make internally consistent measurements without needing to know the real-world scale of these measurements.

This trick gets us as far as being able to define *w* and *h* for any image sensor (since we can always check the pixel dimensions of a captured frame). Now, to be able to calculate the focal length, we just need one more type of data: the FOV. We can measure this using a simple experiment. Take a piece of paper and tape it to a wall (or another vertical surface). Position the camera and lens so that they are directly facing the piece of paper and the edges of the paper lie along the edges of the frame. (If the paper's aspect ratio does not match the frame's aspect ratio, cut the paper to match.) Measure the diagonal size, *s*, from one corner of the paper

to the diagonally opposite corner. Additionally, measure the distance, *d*, from the paper to a point halfway down the barrel of the lens. Then, calculate the diagonal FOV, θ, with trigonometry:

$$\theta = 2\left(\tan^{-1}\frac{s}{2d}\right)$$

Let's suppose that with this experiment, we determine that a given camera and lens have a diagonal FOV of 70 degrees. If we know that we are capturing frames at a resolution of 1280x720, then we can calculate the focal length in pixel-equivalent units as follows:

$$f = \frac{\sqrt{w^2 + h^2}}{2\left(\tan\frac{\theta}{2}\right)} = \frac{\sqrt{1280^2 + 720^2}}{2\left(\tan\frac{70 * \frac{\pi}{180}}{2}\right)} = 1048.7$$

In addition to this, we can calculate the image sensor's center coordinates:

$$c_x = \frac{w}{2} = \frac{1280}{2} = 640$$

$$c_y = \frac{h}{2} = \frac{720}{2} = 360$$

Therefore, we have the following camera matrix:

1048.7	0	640
0	1048.7	360
0	0	1

The preceding parameters are necessary for 3D tracking, and they correctly represent an ideal camera and lens. However, real equipment may deviate noticeably from this ideal, and the camera matrix alone cannot represent all the possible types of deviations. **Distortion coefficients** are a set of additional parameters that can represent the following kinds of deviations from the ideal model:

- **Radial distortion**: This means that the lens does not magnify all parts of the image equally; thus, it makes straight edges appear curvy or wavy. For radial distortion coefficients, variable names such as *kn* (for example, *k1*, *k2*, *k3*, and so forth) are typically used. If *k1<0*, this usually implies that the lens suffers from **barrel distortion**, meaning that straight edges appear to bend outward toward the borders of the image. Conversely, *k1>0* usually implies that the lens suffers from **pincushion distortion**,

meaning that straight edges appear to bend inward toward the center of the image. If the sign (positive or negative) alternates across the series (for example, *k1>0, k2<0*, and *k3>0*), this might imply that the lens suffers from **mustache distortion**, meaning that straight edges appear wavy.

- **Tangential distortion**: This means that the lens's main (lengthwise) axis is not perpendicular to the image sensor; thus, the perspective is skewed, and the angles between the straight edges appear to be different than in a normal perspective projection. In other words, the lens or the sensor is sitting askew in the camera. This is normally considered a defect; however, specialized equipment may be designed so that a photographer can twist it out of shape to produce tangential distortions or, in other words, a false perspective. (Two such types of equipment are **bellows cameras** and **tilt-shift lenses**.) For tangential distortion coefficients, variable names such as *pn* (for example, *p1*, *p2*, and so forth) are typically used. The sign of the coefficient depends on the direction of the lens's tilt relative to the image sensor.

The following diagram illustrates some types of radial distortion:

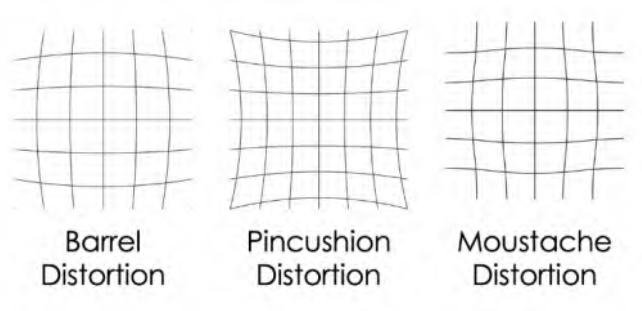


Figure 9.4: Types of radial distortion

OpenCV provides functions to work with as many as five distortion coefficients: *k1*, *k2*, *p1*, *p2*, and *k3*. (OpenCV expects them in this order, as elements of an array.) Rarely, you might be able to obtain official data about distortion coefficients from the vendor of a camera or lens. Alternatively, you can estimate the distortion coefficients, along with the camera matrix, using OpenCV's chessboard calibration process. This involves capturing a series of images of a printed chessboard pattern, viewed from various positions and angles. For further details, you can refer to OpenCV's official tutorial at `https://docs.opencv.org/5.0/py_tutorials/py_calib3d/py_calibration/py_calibration.html`.

For our demo purposes, we will simply assume that all the distortion coefficients are 0, meaning there is no distortion. Of course, we do not really believe that our webcam lenses are distortion-less masterpieces of optical engineering; we just believe that the distortion is not

bad enough to noticeably affect our demo of 3D tracking and AR. If we were trying to build a precise measurement device instead of a visual demo, we would be more concerned about the effects of distortion.

> **Note**
>
> Compared to the chessboard calibration process, the formulas and assumptions that we have outlined in this section produce a more constrained or idealistic model. However, our approach has the advantages of being simpler and more easily reproducible. The chessboard calibration process is more laborious, and each user might execute it differently, producing different (and, sometimes, erroneous) results.

Having absorbed this background information on camera and lens parameters, let's now examine an OpenCV function that uses these parameters as part of a solution to the 6DOF tracking problem.

Understanding cv2.solvePnPRansac

Suppose that we know where certain keypoints are located on a 3D model of an object, and we know where the same keypoints are located in a 2D photo of this object. The `cv2.solvePnPRansac` function implements a solution for the so-called **Perspective-n-Point (PnP)** problem. Given a set of *n* unique matches between 3D and 2D points, along with the parameters of the camera and lens that generated this 2D projection of the 3D points, the solver attempts to estimate the 6DOF pose of the 3D object relative to the camera. This problem is somewhat similar to finding the homography for a set of 2D-to-2D keypoint matches, as we did in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. However, in the PnP problem, we have enough additional information to estimate a more specific spatial relationship – the 6DOF pose – as opposed to the homography, which just tells us a projective relationship.

So, how does `cv2.solvePnPRansac` work? As the function's name suggests, it implements a Ransac algorithm, which is a general-purpose iterative approach designed to deal with a set of inputs that may contain outliers – in our case, bad matches. Each Ransac iteration finds a potential solution that minimizes a measurement of mean error for the inputs. Then, before the next iteration, any inputs with an unacceptably large error are marked as outliers and discarded. This process continues until the solution converges, meaning that no new outliers are found and the mean error is acceptably low.

For the PnP problem, the error is measured in terms of **reprojection error**, meaning the distance between the observed position of a 2D point and the predicted position according to the camera and lens parameters, and the 6DOF pose that we are currently considering as the

potential solution. At the end of the process, we hope to obtain a 6DOF pose that is consistent with most of the 3D-to-2D keypoint matches. Additionally, we want to know which of the matches are inliers for this solution.

Let's consider the function signature of `cv2.solvePnPRansac`:

```
retval, rvec, tvec, inliers = cv2.solvePnPRansac(
    objectPoints,
    imagePoints,
    cameraMatrix,
    distCoeffs,
    rvec=None,
    tvec=None,
    useExtrinsicGuess=False
    iterationsCount=100,
    reprojectionError=8.0,
    confidence=0.98,
    inliers=None,
    flags=cv2.SOLVEPNP_ITERATIVE
)
```

As we can see, the function has four return values:

- `retval`: This is `True` if the solver converged on a solution; otherwise, it is `False`.
- `rvec`: This array contains *rx*, *ry*, and *rz* – the three rotational degrees of freedom in the 6DOF pose.
- `tvec`: This array contains *tx*, *ty*, and *tz* – the three translational (positional) degrees of freedom in the 6DOF pose.
- `inliers`: If the solver converged on a solution, this vector contains the indices of the input points (in `objectPoints` and `imagePoints`) that are congruous with the solution.

The function also has 12 arguments:

- `objectPoints`: This is an array of 3D points that represent the object's keypoints when there is no translation and no rotation – in other words, when the 6DOF pose variables are all 0.
- `imagePoints`: This is an array of 2D points that represent the object's keypoint matches in the image. Specifically, `imagePoints[i]` is believed to be a match for `objectPoints[i]`.

- `cameraMatrix`: This 2D array is the camera matrix, which we can derive in the manner described in the previous section, *Understanding camera and lens parameters*.
- `distCoeffs`: This is the array of distortion coefficients. If we do not know them, we can assume (for simplicity) that they are all 0, as mentioned in the previous section.
- `rvec`: If the solver converges on a solution, it will put the solution's *rx*, *ry*, and *rz* values in this array (or, if this argument is `None`, the solver will return a new array instead).
- `tvec`: If the solver converges on a solution, it will put the solution's *tx*, *ty*, and *tz* values in this array (or, if this argument is `None`, the solver will return a new array instead).
- `useExtrinsicGuess`: If this is `True`, the solver treats the values in the `rvec` and `tvec` arguments as an initial guess, and then it tries to find a solution that is close to these. Otherwise, the solver takes an unbiased approach in its search for a solution.
- `iterationsCount`: This is the maximum number of iterations that the solver should attempt. If it does not converge on a solution after this number of iterations, it gives up.
- `reprojectionError`: This is the maximum reprojection error that the solver will accept; if a point has a greater reprojection error than this, the solver treats it as an outlier.
- `confidence`: The solver attempts to converge on a solution that has a confidence score greater than or equal to this value.
- `inliers`: If the solver converges on a solution, it will put the indices of the solution's inlier points in this array (or, if this argument is `None`, the solver will return a new array instead).
- `flags`: The flags specify the solver's algorithm. The default, `cv2.SOLVEPNP_ITERATIVE`, is an approach that minimizes reprojection error and has no special restrictions, so it is generally the best choice. Specifically, `cv2.SOLVEPNP_ITERATIVE` implements an iterative algorithm called **Levenberg–Marquardt** (**LM**) or **damped least squares** (**DLS**). A useful alternative is `cv2.SOLVEPNP_IPPE` – **IPPE**, short for **Infinitesimal Plane-Based Pose Estimation** – but it is restricted to planar objects such as a book cover. Compared to LM, IPPE is a bit faster but a bit less accurate. For a comparison of the two algorithms' results, see the paper *Insights into the robustness of control point configurations for homography and planar pose estimation*, by Raul Acuna and Volker Willert. An electronic version is available at `https://arxiv.org/pdf/1803.03025.pdf`.

Although this function involves a lot of variables, we will see that its usage is a natural extension of the keypoint matching problems we covered in *Chapter 6, Retrieving Images and*

Searching Using Image Descriptors, and the 3D and projective problems we are introducing in this chapter. With this in mind, let's begin to explore this chapter's sample code.

Implementing the demo application

We are going to implement our demo in a single script, `ImageTrackingDemo.py`, which will contain the following components:

1. Import statements
2. A helper function for a custom grayscale conversion
3. Helper functions to convert keypoints from 2D to 3D space
4. An application class, `ImageTrackingDemo`, which will encapsulate a model of the camera and lens, a model of the reference image, a Kalman filter, 6DOF tracking results (including the translation and both the Rodrigues and Euler representations of the rotation), and an application loop that will track the image and draw a simple AR visualization
5. A `main` function to launch the application

The script will depend on one other file, `reference_image.png`, which will represent the image that we want to track.

Tip

By preparing a reference image in advance, and by loading it from the file at runtime, we can ensure that its technical qualities are good: it has a high resolution (important for close-up tracking), it is properly focused, it is free of distortions, it is evenly lit and well exposed, the subject is upright and flat and not covered by anyone's thumb…

On the other hand, in some applications, the user might need to gather reference images in real time instead. Then, for example, you could implement a user interface to let the user show a reference image to the camera. For such applications, you may want to read about OpenCV's `selectROI` function, which lets the user click and drag the mouse to select a rectangular region of an image (such as the current camera frame). The selected region could be treated as a new reference image.

Without further ado, let's dive into the script's implementation.

Importing modules

From the Python standard library, we will use the `math` module for trigonometric calculations, and the `timeit` module for accurate time measurement (which will enable us to use a Kalman

filter more effectively). As usual, we will also use NumPy and OpenCV. Thus, our implementation of `ImageTrackingDemo.py` begins with the following `import` statements:

```
import math
import timeit

import cv2
import numpy
```

Now, let's proceed to the implementation of the helper functions.

Performing grayscale conversion

Throughout this book, we have performed grayscale conversions using code such as the following:

```
gray_img = cv2.cvtColor(bgr_img, cv2.COLOR_BGR2GRAY)
```

Perhaps a question is long overdue: how exactly does this function map BGR values to grayscale values? The answer is that each output pixel's grayscale value is a weighted average of the corresponding input pixel's B, G, and R values, as follows:

```
gray = (0.114 * blue) + (0.587 * green) + (0.299 * red)
```

These weights are widely used. They come from a telecommunications industry standard called CCIR 601, which was issued in 1982. They are loosely consistent with a characteristic of human vision; when we see a brightly lit scene, our eyes are most sensitive to yellowish-green light. Moreover, these weights should produce high contrast in scenes with yellowish light and bluish shadows, such as an outdoor scene on a sunny day. Are these good reasons for us to use the CCIR 601 weights? No, they are not; there is no scientific evidence that the CCIR 601 conversion weights yield optimal grayscale input for any particular purpose in computer vision.

Indeed, for the purpose of image tracking, there is evidence in favor of other grayscale conversion algorithms. Samuel Macêdo, Givânio Melo, and Judith Kelner address this topic in their paper, *A comparative study of grayscale conversion techniques applied to SIFT descriptors* (SBC Journal on Interactive Systems, vol. 6, no. 2, 2015). They test a variety of conversion algorithms, including the following types:

- A weighted-average conversion, `gray = (0.07 * blue) + (0.71 * green) + (0.21 * red)`, which is somewhat similar to CCIR 601

- An unweighted-average conversion, `gray = (blue + green + red) / 3`
- Conversions based on only a single color channel, such as `gray = green`
- Gamma-corrected conversions, such as `gray = 255 * (green / 255) ^ (1/2.2)`, in which the grayscale value varies exponentially (not linearly) with the inputs

According to the paper, the weighted-average conversion produces results that are relatively unstable – good for finding matches and a homography with some images, but bad with others. The unweighted-average conversion and the single-channel conversions yield more consistent results. For some images, the gamma-corrected conversions yield the best results, but these conversions are computationally more expensive.

For our demo purposes, we will perform grayscale conversion by taking the simple (unweighted) average of each pixel's B, G, and R values. This approach is computationally cheap (which is desirable in real-time tracking), and we expect that it leads to more consistent tracking results than the default weighted-average conversion in OpenCV. Here is our implementation of a helper function to perform the custom conversion:

```
def convert_bgr_to_gray(src, dst=None):
    weight = 1.0 / 3.0
    m = numpy.array([[weight, weight, weight]], numpy.float32)
    return cv2.transform(src, m, dst)
```

Note the use of the `cv2.transform` function. This is a well-optimized, general-purpose matrix transformation function, provided by OpenCV. We can use it to perform operations where the values of a pixel's output channels are a linear combination of the values of the input channels. In the case of our BGR-to-grayscale conversion, we have one output channel and three input channels, so our transformation matrix, `m`, has one row and three columns.

Having written our helper function for grayscale conversions, let's go on to consider helper functions for conversions from 2D to 3D space.

Performing 2D-to-3D spatial conversions

Remember that we have a reference image, `reference_image.png`, and we want our AR application to track a print copy of this image. For the purpose of 3D tracking, we can represent this printed image as a plane in 3D space. Let's define the local coordinate system by saying that, normally (when the elements of the 6DOF pose are all 0), this planar object stands upright like a picture hanging on a wall; its front is the side with the image on it, and its origin is the center of the image.

Now, let's suppose that we want to map a given pixel from the reference image onto this 3D plane. Given the 2D pixel coordinates, the image's pixel dimensions, and a scaling factor to

convert from pixels to some unit of measurement we want to use in 3D space, we can use the following helper function to map a pixel onto the plane:

```
def map_2D_point_onto_3D_plane(point_2D, image_size, image_scale):
    x, y = point_2D
    w, h = image_size
    return (image_scale * (x - 0.5 * w),
            image_scale * (y - 0.5 * h),
            0.0)
```

The scaling factor depends on the real-world size of the printed image and our choice of unit. For example, we might know that our printed image is 20 cm tall – or we might not care about the absolute scale, in which case we could define an arbitrary unit such that the printed image is one unit tall. Anyway, given a list of 2D pixel coordinates, the reference image's size in pixels, and the reference image's real-world height in any unit (absolute or relative), we can use the following helper function to obtain a list of the corresponding 3D coordinates on the plane:

```
def map_2D_points_onto_3D_plane(points_2D, image_size,
                                image_real_height):

    w, h = image_size
    image_scale = image_real_height / h

    points_3D = [
        map_2D_point_onto_3D_plane(
            point_2D, image_size, image_scale)
        for point_2D in points_2D
    ]
    return numpy.array(points_3D, numpy.float32)
```

Note that we have a helper function for multiple points, `map_2D_points_onto_3D_plane`, and it calls a helper function for every single point, `map_point_to_plane`.

Later, in the *Initializing the tracker* section, we will generate ORB keypoint descriptors for the reference image, and we will use our `map_2D_points_onto_3D_plane` helper function in order to convert the keypoint coordinates from 2D to 3D. We will also convert the image's four 2D vertices (that is, its top-left, top-right, bottom-right, and bottom-left corners) to obtain the four 3D vertices of the plane. We will use these vertices when we perform our AR drawing – specifically, in the *Drawing the tracking results and masking the tracked object* section. Drawing-related functions (in OpenCV and many other frameworks) expect the vertices to be specified

in clockwise order (from a frontal perspective) for each face of the 3D shape. To deal with this requirement, let's implement another helper function that is specific to mapping vertices; here it is:

```
def map_vertices_to_plane(image_size, image_real_height):

    w, h = image_size

    vertices_2D = [(0, 0), (w, 0), (w, h), (0, h)]
    vertex_indices_by_face = [[0, 1, 2, 3]]

    vertices_3D = map_2D_points_onto_3D_plane(
        vertices_2D, image_size, image_real_height)
    return vertices_3D, vertex_indices_by_face
```

Note that our vertex-mapping helper function, `map_vertices_to_plane`, calls our `map_2D_points_onto_3D_plane` helper function, which, in turn, calls `map_point_to_plane`. Therefore, all our mapping functionality shares a common core.

Of course, 2D-to-3D keypoint mapping and vertex mapping can be applied to other kinds of 3D shapes besides planes. To learn how our approach can extend to 3D cuboids and 3D cylinders, please refer to the *Visualizing the Invisible* demo project, by Joseph Howse, which is available at `https://github.com/JoeHowse/VisualizingTheInvisible/`.

We have finished implementing the helper functions. Now, let's proceed to the object-oriented part of the code.

Implementing the application class

We will implement our application in a class named `ImageTrackingDemo`, which will have the following methods:

- `__init__(self, capture, diagonal_fov_degrees, target_fps, reference_image_path, reference_image_real_height)`: The initializer will set up a capture device, a camera matrix, a Kalman filter, and 2D and 3D keypoints for the reference image.
- `run(self)`: This method will run the application's `main` loop, which captures, processes, and displays frames until the user quits by hitting the *Esc* key. The processing of each frame is performed with the help of other methods, which are mentioned next in this list.

- `_track_object(self)`: This method will perform 6DOF tracking and draw an AR visualization of the tracking result.
- `_init_kalman_transition_matrix(self, fps)`: This method will configure the Kalman filter to ensure that acceleration and velocity are simulated properly for the specified frame rate.
- `_apply_kalman(self)`: This method will stabilize the 6DOF tracking result by applying the Kalman filter.

Let's walk through the methods' implementations one by one, starting with `__init__`.

Initializing the tracker

The `__init__` method involves a lot of steps to initialize the camera matrix, the ORB descriptor extractor, the Kalman filter, the reference image's 2D and 3D keypoints, and other variables related to our tracking algorithm:

1. To begin, let's look at the several arguments that `__init__` accepts. These include a `cv2.VideoCapture` object, called `capture` (the camera); the camera's diagonal FOV, in degrees; the expected frame rate, in **frames per second** (**FPS**); a path to a file containing the reference image; and a measurement of the reference image's real-world height (in any unit):

```
class ImageTrackingDemo():

    def __init__(self, capture, diagonal_fov_degrees=70.0,
                 target_fps=25.0,
                 reference_image_path='reference_image.png',
                 reference_image_real_height=1.0):
```

2. We attempt to capture a frame from the camera in order to determine its pixel dimensions; failing that, we get the dimensions from the camera's properties:

```
        self._capture = capture
        success, trial_image = capture.read()
        if success:
            # Use the actual image dimensions.
            h, w = trial_image.shape[:2]
        else:
            # Use the nominal image dimensions.
            w = capture.get(cv2.CAP_PROP_FRAME_WIDTH)
```

```
            h = capture.get(cv2.CAP_PROP_FRAME_HEIGHT)
        self._image_size = (w, h)
```

3. Now, given the frame's dimensions in pixels, and the FOV of the camera and lens, we can use trigonometry to calculate the focal length in pixel-equivalent units. (The formula is the one we derived earlier in this chapter, in the *Understanding camera and lens parameters* section.) Moreover, using the focal length and the frame's center point, we can construct the camera matrix. Here is the relevant code:

```
        diagonal_image_size = (w ** 2.0 + h ** 2.0) ** 0.5
        diagonal_fov_radians = \
            diagonal_fov_degrees * math.pi / 180.0
        focal_length = 0.5 * diagonal_image_size / math.tan(
            0.5 * diagonal_fov_radians)
        self._camera_matrix = numpy.array(
            [[focal_length, 0.0, 0.5 * w],
             [0.0, focal_length, 0.5 * h],
             [0.0, 0.0, 1.0]], numpy.float32)
```

4. For the sake of simplicity, we assume that the lens does not suffer from any distortion whatsoever:

```
        self._distortion_coefficients = None
```

5. Initially, we are not tracking the object, so we have no estimate of its rotation and position; we just define the relevant variables as if the rotation and position are 0:

```
        self._rodrigues_rotation_vector = numpy.array(
            [[0.0], [0.0], [1.0]], numpy.float32)
        self._euler_rotation_vector = numpy.zeros(
            (3, 1), numpy.float32)
        self._rotation_matrix = numpy.zeros((3, 3), numpy.float64)

        self._translation_vector = numpy.zeros(
            (3, 1), numpy.float32)
```

6. Now, let's set up a Kalman filter:

```
        self._kalman = cv2.KalmanFilter(18, 6)
```

```
        self._kalman.processNoiseCov = numpy.identity(
            18, numpy.float32) * 1e-5
        self._kalman.measurementNoiseCov = numpy.identity(
            6, numpy.float32) * 1e-2
        self._kalman.errorCovPost = numpy.identity(
            18, numpy.float32)

        self._kalman.measurementMatrix = numpy.array(
            [[1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]],
            numpy.float32)

        self._init_kalman_transition_matrix(target_fps)
```

As indicated by the preceding code, `cv2.KalmanFilter(18, 6)`, this Kalman filter will track 18 output variables (or predictions), based on 6 input variables (or measurements). Specifically, the input variables are the elements of the 6DOF tracking result: *tx*, *ty*, *tz*, *rx*, *ry*, and *rz*. The output variables are the elements of the stabilized 6DOF tracking result, plus their first-order derivatives (velocity) and their second-order derivatives (acceleration), in the following order: *tx*, *ty*, *tz*, *tx'*, *ty'*, *tz'*, *tx''*, *ty''*, *tz''*, *rx*, *ry*, *rz*, *rx'*, *ry'*, *rz'*, *rx''*, *ry''*, and *rz''*. The Kalman filter's measurement matrix has 18 columns (representing the output variables) and 6 rows (representing the input variables). Within each row, we put `1.0` in the index that corresponds to the matching output variable; elsewhere, we put `0.0`. We also initialize a transition matrix, which defines the relationships among the output variables over time. This part of the initialization is handled by a helper method, `_init_kalman_transition_matrix(target_fps)`, which we will examine later, in the *Initializing and applying the Kalman filter* section.

> **Note**
>
> Not all of the Kalman filter's matrices are initialized by our `__init__` method. The transition matrix is updated every frame during tracking because the actual frame rate (and, thus, the time step) may change. The state matrices are initialized every time we start tracking an object. We will cover these aspects of the Kalman filter's usage in due course, in the *Initializing and applying the Kalman filter* section.

7. We need a Boolean variable (initially, `False`) to indicate whether we successfully tracked the object in the previous frame:

```
        self._was_tracking = False
```

8. We need to define the vertices of some 3D graphics that we will draw every frame as part of our AR visualization. Specifically, the graphics will be a set of arrows representing the object's x, y, and z axes. The scale of these graphics will relate to the scale of the real object – that is, the printed image that we intend to track. Remember that, as one of its arguments, the `__init__` method takes the image's scale – specifically, its height – and that this measurement may be in any unit. Let's define the length of the 3D axis arrows to be half the height of the printed image:

```
        self._reference_image_real_height = \
            reference_image_real_height
        reference_axis_length = 0.5 * reference_image_real_height
```

9. Using the length that we have just defined, let's define the vertices of the axis arrows relative to the printed image's center, `[0.0, 0.0, 0.0]`:

```
        self._reference_axis_points_3D = numpy.array(
            [[0.0, 0.0, 0.0],
             [-reference_axis_length, 0.0, 0.0],
             [0.0, -reference_axis_length, 0.0],
             [0.0, 0.0, -reference_axis_length]], numpy.float32)
```

   Note that OpenCV's coordinate system has nonstandard axis directions, as follows:

   - $+x$ (the positive X direction) is the object's left-hand direction, or the viewer's right-hand direction in a frontal view of the object

- $+y$ is down
- $+z$ is the object's backward direction, or the viewer's frontward direction in a frontal view of the object

We must negate all of the preceding directions in order to obtain the following standard right-handed coordinate system, like the one used in many 3D graphics frameworks such as OpenGL:

- $+x$ is the object's right-hand direction, or the viewer's left-hand direction in a frontal view of the object
- $+y$ is up
- $+z$ is the object's forward direction, or the viewer's backward direction in a frontal view of the object

Note

For the purposes of this book, we use OpenCV to draw 3D graphics, so we could simply adhere to OpenCV's nonstandard axis directions, even when we draw visualizations. However, if you do further AR work in the future, you will likely need to integrate your computer vision code with OpenGL and other 3D graphics frameworks using a right-handed coordinate system. To better prepare you for this eventuality, we will convert the axis directions in our OpenCV-centric demo.

10. We will use three arrays to hold three kinds of images: the BGR video frame (where we will do our AR drawing), the grayscale version of the frame (which we will use for keypoint matching), and the mask (where we will draw a silhouette of the tracked object). Initially, these arrays are all `None`:

```
        self._bgr_image = None
        self._gray_image = None
        self._mask = None
```

11. We will use a `cv2.ORB` object to detect keypoints and compute descriptors for the reference image and, later, for camera frames. We initialize the `cv2.ORB` object as follows:

```
        # Create and configure the feature detector.
        patchSize = 31
```

```
        self._feature_detector = cv2.ORB_create(
            nfeatures=250, scaleFactor=1.2, nlevels=16,
            edgeThreshold=patchSize, patchSize=patchSize)
```

> **Note**
>
> For a refresher on the ORB algorithm and its usage in OpenCV, refer back to *Chapter 6, Retrieving Images and Searching Using Image Descriptors*, specifically to the *Using ORB with FAST features and BRIEF descriptors* section.

Here, we have specified several optional parameters for the constructor of `cv2.ORB`. The diameter covered by a descriptor is 31 pixels, our image pyramid has 16 levels with a scale factor of 1.2 between consecutive levels, and we want, at most, 250 keypoints and descriptors per detection attempt. You may want to experiment with these values. For example, if you increase `nfeatures`, you may get more accurate results but the frame rate may drop because there are more descriptors to match.

12. Now, we load the reference image from a file, resize it, convert it to grayscale, and create an empty mask for it:

```
        bgr_reference_image = cv2.imread(
            reference_image_path, cv2.IMREAD_COLOR)
        reference_image_h, reference_image_w = \
            bgr_reference_image.shape[:2]
        reference_image_resize_factor = \
            (2.0 * h) / reference_image_h
        bgr_reference_image = cv2.resize(
            bgr_reference_image, (0, 0), None,
            reference_image_resize_factor,
            reference_image_resize_factor, cv2.INTER_CUBIC)
        gray_reference_image = convert_bgr_to_gray(
            bgr_reference_image)
        reference_mask = numpy.empty_like(gray_reference_image)
```

When resizing the reference image, we have chosen to make it twice as high as the camera frame. The exact number is arbitrary; however, the idea is that we want to perform keypoint detection and description with an image pyramid that covers a useful range of magnifications. The base of the pyramid (that is, the resized reference image) should be larger than the camera frame so that we can match keypoints at an appropriate scale even when the target object is so close to the camera that it cannot all

fit into the frame. Conversely, the top level of the pyramid should be smaller than the camera frame so that we can match keypoints at an appropriate scale even when the target object is too far away to fill the whole frame.

Let's consider an example. Suppose that our original reference image is 4000x3000 pixels and that our camera frame is 1280x720 pixels. We resize the reference image to 1920x1440 pixels (twice the height of the frame, and the same aspect ratio as the original reference image). Thus, the base of our image pyramid is also 1920x1440 pixels. Since our `cv2.ORB` object is configured to use 16 pyramid levels and a scale factor of 1.2, the top of the image pyramid has a width of $1920/(1.2^{16-1})=124$ pixels and a height of $1440/(1.2^{16-1})=93$ pixels; in other words, it is 124x93 pixels. Therefore, we can potentially match keypoints and track the object even if it is so far away that it fills just 10% of the frame's width or height. Realistically, to perform useful keypoint matching at this scale, we would need a good lens, the object would need to be in focus, and the lighting would need to be good as well.

13. At this stage, we have an appropriately sized reference image in BGR color and in grayscale, and we have an empty mask for this image. We are going to partition the image into 36 equally sized regions of interest (in a 6x6 grid), and for each region, we will attempt to generate as many as 250 keypoints and descriptors (since our `cv2.ORB` object is configured to use this maximum number of keypoints and descriptors). This partitioning scheme helps to ensure that we have some keypoints and descriptors in every region, so we can potentially match keypoints and track the object even if most parts of the object are not visible in a given frame. The following code block shows how we iterate over the regions of interest and, for each region, create a mask, perform keypoint detection and descriptor extraction, and append the keypoints and descriptors to master lists:

```
        # Find keypoints and descriptors for multiple segments of
        # the reference image.
        reference_keypoints = []
        self._reference_descriptors = numpy.empty(
            (0, 32), numpy.uint8)
        num_segments_y = 6
        num_segments_x = 6
        for segment_y, segment_x in numpy.ndindex(
                (num_segments_y, num_segments_x)):
            y0 = reference_image_h * \
                segment_y // num_segments_y - patchSize
            x0 = reference_image_w * \
```

```
                segment_x // num_segments_x - patchSize
            y1 = reference_image_h * \
                (segment_y + 1) // num_segments_y + patchSize
            x1 = reference_image_w * \
                (segment_x + 1) // num_segments_x + patchSize
            reference_mask.fill(0)
            cv2.rectangle(
                reference_mask, (x0, y0), (x1, y1), 255, cv2.FILLED)
            more_reference_keypoints, more_reference_descriptors = \
                self._feature_detector.detectAndCompute(
                    gray_reference_image, reference_mask)
            if more_reference_descriptors is None:
                # No keypoints were found for this segment.
                continue
            reference_keypoints += more_reference_keypoints
            self._reference_descriptors = numpy.vstack(
                (self._reference_descriptors,
                 more_reference_descriptors))
```

14. Now, we draw a visualization of the keypoints atop the grayscale reference image:

```
        cv2.drawKeypoints(
            gray_reference_image, reference_keypoints,
            bgr_reference_image,
            flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

15. Next, we save the visualization to a file with `_keypoints` appended to the name. For example, if the filename of the reference image were `reference_image.png`, we'd save the visualization as `reference_image_keypoints.png`. Here is the relevant code:

```
        ext_i = reference_image_path.rfind('.')
        reference_image_keypoints_path = \
            reference_image_path[:ext_i] + '_keypoints' + \
            reference_image_path[ext_i:]
        cv2.imwrite(
            reference_image_keypoints_path, bgr_reference_image)
```

16. We proceed to initialize the FLANN-based matcher with custom parameters:

```
FLANN_INDEX_LSH = 6
index_params = dict(algorithm=FLANN_INDEX_LSH,
                    table_number=6, key_size=12,
                    multi_probe_level=1)
search_params = dict()
self._descriptor_matcher = cv2.FlannBasedMatcher(
    index_params, search_params)
```

These parameters specify that we are using a multi-probe **LSH (locality-sensitive hashing)** indexing algorithm with 6 hash tables, a hash key size of 12 bits, and 1 multi-probe level.

> **Note**
>
> For a description of the multi-probe LSH algorithm, refer to the paper *Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search* (VLDB, 2007), by Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. An electronic version is available at `https://www.cs.princeton.edu/cass/papers/mplsh_vldb07.pdf`.

17. We train the matcher by feeding the reference descriptors to it:

```
self._descriptor_matcher.add([self._reference_descriptors])
```

18. We take the 2D coordinates of the keypoints, and we feed these to our `map_2D_points_onto_3D_plane` helper function in order to obtain equivalent 3D coordinates on the surface of the object's plane:

```
reference_points_2D = [
    keypoint.pt
    for keypoint in reference_keypoints
]
self._reference_points_3D = map_2D_points_onto_3D_plane(
    reference_points_2D,
    gray_reference_image.shape[::-1],
```

```
            reference_image_real_height
        )
```

19. Similarly, we call our `map_vertices_to_plane` function in order to obtain the 3D vertices and 3D face of the plane:

```
        (self._reference_vertices_3D,
         self._reference_vertex_indices_by_face) = \
            map_vertices_to_plane(
                gray_reference_image.shape[::-1],
                reference_image_real_height)
```

This concludes the implementation of the `__init__` method. Next, let's take a look at the `run` method, which represents the application's `main` loop.

Implementing the main loop

As usual, our `main` loop's primary role is to capture and process frames, until the user hits the `Esc` key. The processing of each frame – including 3D tracking and AR drawing – is delegated to a helper method called `_track_object`, which we will examine later, in the *Tracking the image in 3D* section. The `main` loop also has a secondary role: that is, to perform timekeeping by measuring the frame rate and updating the Kalman filter's transition matrix accordingly. This update is delegated to another helper method, `_init_kalman_transition_matrix`, which we will examine in the *Initializing and applying the Kalman filter* section. With these roles in mind, we can implement the `main` loop in the run method as follows:

```
    def run(self):

        num_images_captured = 0
        start_time = timeit.default_timer()

        while cv2.waitKey(1) != 27:  # Escape
            success, self._bgr_image = self._capture.read(
                self._bgr_image)
            if success:
                num_images_captured += 1
                self._track_object()
                cv2.imshow('Image Tracking', self._bgr_image)
            delta_time = timeit.default_timer() - start_time
            if delta_time > 0.0:
```

```
                fps = num_images_captured / delta_time
                self._init_kalman_transition_matrix(fps)
```

Note the use of the `timeit.default_timer` function from Python's standard library. This function provides a precise measurement of the current system time in seconds (as a floating-point number, so fractions of seconds can be expressed). As the name `timeit` suggests, this module contains useful functionality for situations where you have time-sensitive code and you want to *time it*.

Let's move on to the implementation of `_track_object`, since this helper performs the largest part of the application's work on behalf of `run`.

Tracking the image in 3D

The `_track_object` method is directly responsible for keypoint matching, keypoint visualizations, and solving the PnP problem. Additionally, it calls other methods to deal with Kalman filtering, AR drawing, and masking the tracked object:

1. To begin `_track_object`'s implementation, we call our `convert_bgr_to_gray` helper function to convert the frame to grayscale:

```
    def _track_object(self):

        self._gray_image = convert_bgr_to_gray(
            self._bgr_image, self._gray_image)
```

2. Now, we use our `cv2.ORB` object to detect keypoints and compute descriptors in a masked region of the grayscale image:

```
        if self._mask is None:
            self._mask = numpy.full_like(self._gray_image, 255)

        keypoints, descriptors = \
            self._feature_detector.detectAndCompute(
                self._gray_image, self._mask)
```

   If we were already tracking the object in the previous frame, the mask covers the region where we previously found the object. Otherwise, the mask covers the whole frame because we have no idea where the object might be. We will see how the mask is created later, in the *Drawing the tracking results and masking the tracked object* section.

3. Next, we use our FLANN matcher to find matches between the reference image's keypoints and the frame's keypoints, and we filter these matches according to the ratio test:

```
        # Find the 2 best matches for each descriptor.
        matches = self._descriptor_matcher.knnMatch(descriptors, 2)

        # Filter the matches based on the distance ratio test.
        good_matches = [
            match[0] for match in matches
            if len(match) > 1 and \
            match[0].distance < 0.8 * match[1].distance
        ]
```

> **Note**
>
> For details about FLANN matching and the ratio test, refer back to *Chapter 6, Retrieving Images and Searching Using Image Descriptors*.

4. At this stage, we have a list of good matches that passed the ratio test. Let's select the subset of the frame's keypoints that correspond to these good matches, and let's draw red circles on the frame to visualize these keypoints:

```
        # Select the good keypoints and draw them in red.
        good_keypoints = [keypoints[match.queryIdx]
                          for match in good_matches]
        cv2.drawKeypoints(self._gray_image, good_keypoints,
                          self._bgr_image, (0, 0, 255))
```

5. Having found the good matches, we obviously know how many of them there are. If the count is small, then, overall, the set of matches can be considered doubtful and inadequate for tracking. We define two different thresholds for the minimum number of good matches: a higher threshold if we are just starting tracking (that is, we were not tracking the object in the previous frame), and a lower threshold if we are continuing tracking (after already tracking the object in the previous frame):

```
        min_good_matches_to_start_tracking = 8
        min_good_matches_to_continue_tracking = 6
        num_good_matches = len(good_matches)
```

6. If we fail to meet even the lower threshold, then we note that we did not track the object in this frame, and we reset the mask so that it covers the whole frame:

```
        if num_good_matches < min_good_matches_to_continue_tracking:
            self._was_tracking = False
            self._mask.fill(255)
```

7. On the other hand, if we have enough matches to satisfy the applicable threshold, we proceed to try to track the object. The first step in this is to select the good matches' 2D coordinates in the frame and their 3D coordinates in the model of the `reference` object:

```
        elif num_good_matches >= \
                min_good_matches_to_start_tracking or \
                    self._was_tracking:

            # Select the 2D coordinates of the good matches.
            # They must be in an array of shape (N, 1, 2).
            good_points_2D = numpy.array(
                [[keypoint.pt] for keypoint in good_keypoints],
                numpy.float32)

            # Select the 3D coordinates of the good matches.
            # They must be in an array of shape (N, 1, 3).
            good_points_3D = numpy.array(
                [[self._reference_points_3D[match.trainIdx]]
                 for match in good_matches],
                numpy.float32)
```

8. Now, we are ready to call `cv2.solvePnPRansac` using the kinds of arguments we described near the start of this chapter, in the *Understanding cv2.solvePnPRansac* section. Notably, we use the 3D reference keypoints and the 2D scene keypoints from only the good matches:

```
            # Solve for the pose and find the inlier indices.
            (success, rodrigues_rotation_vector_temp,
             translation_vector_temp, inlier_indices) = \
                cv2.solvePnPRansac(good_points_3D, good_points_2D,
                                   self._camera_matrix,
                                   self._distortion_coefficients,
```

```
                                        None, None,
                                        useExtrinsicGuess=False,
                                        iterationsCount=100,
                                        reprojectionError=8.0,
                                        confidence=0.99,
                                        flags=cv2.SOLVEPNP_ITERATIVE)
```

Tip

Note that at this stage, we have put the new rotation and translation results in local variables, `rodrigues_rotation_vector_temp` and `translation_vector_temp`. We are not yet sure whether the new rotation and translation results are good, so we do not yet want to overwrite the old results in `self._rogdrigues_rotation_vector` and `self._translation_vector`.

9. The solver may or may not have converged on a solution to the PnP problem. If it did not converge, we do nothing further in this method. If it did converge, we overwrite the old rotation and translation results. As part of this step, we call a helper method, `_convert_rodrigues_to_euler`, to convert the new rotation result to Euler angles:

```
            if success:

                self._translation_vector[:] = translation_vector_temp
                self._rodrigues_rotation_vector[:] = \
                    rodrigues_rotation_vector_temp
                self._convert_rodrigues_to_euler()
```

10. The next thing we do is check whether we were already tracking the object in the previous frame. If we were not already tracking it – in other words, if we are starting to track the object afresh in this frame – then we reinitialize the Kalman filter by calling a helper method, `_init_kalman_state_matrices`:

```
                if not self._was_tracking:
                    self._init_kalman_state_matrices()
                    self._was_tracking = True
```

11. Now, in any case, we are tracking the object in this frame, so we can apply the Kalman filter by calling another helper method, `_apply_kalman`:

```
                self._apply_kalman()
```

12. At this stage, we have a Kalman-filtered 6DOF pose. We also have a list of the inlier keypoints from `cv2.solvePnPRansac`. To help the user visualize the result, let's draw the inlier keypoints in green:

```
                # Select the inlier keypoints.
                inlier_keypoints = [good_keypoints[i]
                                    for i in inlier_indices.flat]

                # Draw the inlier keypoints in green.
                cv2.drawKeypoints(self._bgr_image, inlier_keypoints,
                                  self._bgr_image, (0, 255, 0))
```

    Remember that earlier in this method, we drew all the keypoints in red. Now that we have drawn over the inlier keypoints in green, only the outlier keypoints are still red.

13. Finally, we call two more helper methods: the `self._ draw_object_axes` to draw the tracked object's 3D axes, and `self._make_and_draw_object_mask` to make and draw a mask of the region that contains the object:

```
                # Draw the axes of the tracked object.
                self._draw_object_axes()

                # Make and draw a mask around the tracked object.
                self._make_and_draw_object_mask()
```

There ends the implementation of our `_track_object` method. By now, we have a mostly complete picture of our tracking algorithm's implementation, but we still need to implement helper methods relating to the Kalman filter (in the next section, *Initializing and applying the Kalman filter*), and masking and AR drawing (in the section after that, *Drawing the tracking results and masking the tracked object*).

Initializing and applying the Kalman filter

We looked at some aspects of the Kalman filter's initialization earlier, in the *Initializing the tracker* section. However, in that section, we noted that some of the Kalman filter's matrices need to be initialized or reinitialized multiple times, as the application runs through various

frames and various states of tracking or not tracking. Specifically, the following matrices will change:

- **The transition matrix**: This matrix expresses the temporal relationships among all the output variables. For example, this matrix can model the effects of acceleration on velocity, and of velocity on position. We will reinitialize the transition matrix every frame because the frame rate (and, therefore, the time step between frames) is variable. Effectively, this is a way of scaling the previous predictions of acceleration and velocity to match the new time step.
- **The pre-correction and post-correction state matrices**: These matrices contain the predictions of the output variables. The predictions in the pre-correction matrix only take account of the previous state and the transition matrix. The predictions in the post-correction matrix also take account of new inputs and the Kalman filter's other matrices. We will reinitialize the state matrices whenever we go from a non-tracking state to a tracking state – in other words, when we failed to track the object in the previous frame but now we succeed in tracking it in the current frame. Effectively, this is a way of clearing outdated predictions and starting afresh with new measurements.

Let's take a look at the transition matrix first. Its initialization method will take one argument, `fps`, that is, the frame rate in frames per second. We can implement the method in three steps:

1. We begin by validating the `fps` argument. If it is not positive, we return immediately without updating the transition matrix:

```
    def _init_kalman_transition_matrix(self, fps):

        if fps <= 0.0:
            return
```

2. Having determined that `fps` is positive, we proceed to calculate transition rates for velocity and acceleration. We want the velocity transition rate to be proportional to the time step (that is, the time per frame). Because `fps` (frames per second) is the inverse of the time step (that is, seconds per frame), the velocity transition rate is inversely proportional to `fps`. The acceleration transition rate is proportional to the square of the velocity transition rate (and thus, indirectly, the acceleration transition rate is inversely proportional to the square of `fps`). Choosing 1.0 as a base scale for the velocity

transition rate and 0.5 as a base scale for the acceleration transition rate, we can calculate them in code as follows:

```
        # Velocity transition rate
        vel = 1.0 / fps

        # Acceleration transition rate
        acc = 0.5 * (vel ** 2.0)
```

3. Next, we populate the transition matrix. Since we have 18 output variables, the transition matrix has 18 rows and 18 columns. First, let's take a look at the content of the matrix, and, afterward, we will consider how to interpret it:

```
        self._kalman.transitionMatrix = numpy.array(
            [[1.0, 0.0, 0.0, vel, 0.0, 0.0, acc, 0.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 1.0, 0.0, 0.0, vel, 0.0, 0.0, acc, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 1.0, 0.0, 0.0, vel, 0.0, 0.0, acc,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, vel, 0.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, vel, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, vel,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              1.0, 0.0, 0.0, vel, 0.0, 0.0, acc, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 1.0, 0.0, 0.0, vel, 0.0, 0.0, acc, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 0.0, 1.0, 0.0, 0.0, vel, 0.0, 0.0, acc],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
              0.0, 0.0, 0.0, 1.0, 0.0, 0.0, vel, 0.0, 0.0],
             [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
                0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, vel, 0.0],
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, vel],
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]],
              numpy.float32)
```

Each row expresses a formula for calculating a new output value based on the previous frame's output values. Let's consider the first row as an example. We can interpret it as follows:

$$t_x \leftarrow 1(t_x) + 0(t_y) + 0(t_z) + v(t'_x) + 0(t'_y) + 0(t'_z) + a(t''_x) + 0(t''_y) + 0(t''_z)$$
$$= t_x + v(t'_x) + a(t''_x)$$

The new t_x (x position) value depends on the old t_x, t'_x (x velocity), and t''_x (x acceleration) values, along with the velocity transition rate, v, and the acceleration transition rate, a. As we saw earlier in this function, these transition rates may vary because the time step may vary.

That concludes the implementation of the helper method to initialize or update the transition matrix. Remember that we call this function every frame because the frame rate (and thus the time step) may have changed.

We also need a helper function to initialize the state matrices. Remember that we call this method whenever we transition from a non-tracking state to a tracking state. This transition is an appropriate time to clear out any previous predictions; instead, we are starting afresh with the belief that the object's 6DOF pose is exactly what the PnP solver says it is. Moreover, we assume that the object is stationary, with zero velocity and zero acceleration. Here is the helper method's implementation:

```
    def _init_kalman_state_matrices(self):

        t_x, t_y, t_z = self._translation_vector.flat
        pitch, yaw, roll = self._euler_rotation_vector.flat

        self._kalman.statePre = numpy.array(
            [[t_x],   [t_y], [t_z],
             [0.0],   [0.0], [0.0],
```

```
             [0.0],   [0.0], [0.0],
             [pitch], [yaw], [roll],
             [0.0],   [0.0], [0.0],
             [0.0],   [0.0], [0.0]], numpy.float32)
        self._kalman.statePost = numpy.array(
            [[t_x],   [t_y], [t_z],
             [0.0],   [0.0], [0.0],
             [0.0],   [0.0], [0.0],
             [pitch], [yaw], [roll],
             [0.0],   [0.0], [0.0],
             [0.0],   [0.0], [0.0]], numpy.float32)
```

Note that the state matrices have 1 row and 18 columns since we have 18 output variables. Also, note that we are performing Kalman filtering on the Euler representation of the rotation, not the Rodrigues representation.

Now that we have covered the process of initializing and reinitializing the Kalman filter's matrices, let's take a look at how we apply the filter. As we have previously seen in *Chapter 8, Tracking Objects*, we can ask the Kalman filter to estimate the object's new pose (the pre-correction state of the output variables), then we can tell it to take into account the latest unstabilized tracking result (the input variables) in order to adjust its estimate (thereby producing the post-correction state), and, finally, we can extract variables from the adjusted estimate to use as our stabilized tracking result. Compared to our previous work, the only difference this time is that we have more input and output variables. The following code shows how we implement a method to apply the Kalman filter in the context of our 6DOF tracker:

```
    def _apply_kalman(self):

        self._kalman.predict()

        t_x, t_y, t_z = self._translation_vector.flat
        pitch, yaw, roll = self._euler_rotation_vector.flat

        estimate = self._kalman.correct(numpy.array(
            [[t_x],   [t_y], [t_z],
             [pitch], [yaw], [roll]], numpy.float32))

        translation_estimate = estimate[0:3]
        euler_rotation_estimate = estimate[9:12]

        self._translation_vector[:] = translation_estimate
```

```
        angular_delta = cv2.norm(self._euler_rotation_vector,
                                 euler_rotation_estimate, cv2.NORM_L2)

        MAX_ANGULAR_DELTA = 30.0 * math.pi / 180.0
        if angular_delta > MAX_ANGULAR_DELTA:
            # The rotational motion stabilization seems to be drifting
            # too far, probably due to an Euler angle singularity.

            # Reset the rotational motion stabilization.
            # Let the translational motion stabilization continue as-is.

            self._kalman.statePre[9] = pitch
            self._kalman.statePre[10] = yaw
            self._kalman.statePre[11] = roll
            self._kalman.statePre[12:18] = 0.0

            self._kalman.statePost[9] = pitch
            self._kalman.statePost[10] = yaw
            self._kalman.statePost[11] = roll
            self._kalman.statePost[12:18] = 0.0
        else:
            self._euler_rotation_vector[:] = euler_rotation_estimate
            self._convert_euler_to_rodrigues()
```

Here, note that `estimate[0:3]` corresponds to *tx*, *ty*, and *tz*, while `estimate[9:12]` corresponds to *rx*, *ry*, and *rz* (or, in other words, pitch, yaw, and roll). The rest of the `estimate` array corresponds to the first-order derivatives (velocity) and second-order derivatives (acceleration).

Also, note that if we detect an abrupt change of direction in the Euler angle vector, we assume that we passed through a singularity and we reset the rotational motion stabilization by treating the latest rotational tracking result as correct. (Otherwise, if we attempted to apply smoothing to Euler values near a singularity, we would see the object spin wildly.)

Having seen the way we apply a Kalman filter to Euler angles, now let's look at implementations of helper methods to convert a Rodrigues rotation vector to an Euler rotation vector and vice versa.

Converting between Rodrigues vectors and Euler vectors

EuclideanSpace, a mathematics and geometry website, provides an excellent pair of articles on the relationship between the 3x3 rotation matrix and Euler angles, specifically including the yaw-pitch-roll Tait-Bryan variant:

- A conceptual overview is available at `https://www.euclideanspace.com/maths/geometry/rotations/euler/index.htm`
- Conversion formulas, along with sample code in C++, are available at `http://euclideanspace.com/maths/geometry/rotations/conversions/matrixToEuler/index.htm`

Tip

You may want to open the two preceding links now so that you can compare the explanation and formulas from EuclideanSpace to our OpenCV code, below.

We will adapt the sample code from EuclideanSpace and also make use of an OpenCV function, `cv2.Rodrigues`, which converts between the Rodrigues vector and the 3x3 rotation matrix.

Here is our Python implementation of a helper method to convert the Rodrigues vector to an Euler vector:

```
    def _convert_rodrigues_to_euler(self):

        self._rotation_matrix, jacobian = cv2.Rodrigues(
            self._rodrigues_rotation_vector, self._rotation_matrix)

        m00 = self._rotation_matrix[0, 0]
        m02 = self._rotation_matrix[0, 2]
        m10 = self._rotation_matrix[1, 0]
        m11 = self._rotation_matrix[1, 1]
        m12 = self._rotation_matrix[1, 2]
        m20 = self._rotation_matrix[2, 0]
        m22 = self._rotation_matrix[2, 2]

        # Convert to Euler angles using the yaw-pitch-roll
        # Tait-Bryan convention.
        if m10 > 0.998:
            # The rotation is near the "vertical climb" singularity.
            pitch = 0.5 * math.pi
```

```
            yaw = math.atan2(m02, m22)
            roll = 0.0
        elif m10 < -0.998:
            # The rotation is near the "nose dive" singularity.
            pitch = -0.5 * math.pi
            yaw = math.atan2(m02, m22)
            roll = 0.0
        else:
            pitch = math.asin(m10)
            yaw = math.atan2(-m20, m00)
            roll = math.atan2(-m12, m11)

        self._euler_rotation_vector[0] = pitch
        self._euler_rotation_vector[1] = yaw
        self._euler_rotation_vector[2] = roll
```

Conversely, the following method implements the conversion from the Euler vector back to the Rodrigues vector:

```
    def _convert_euler_to_rodrigues(self):

        pitch = self._euler_rotation_vector[0].item()
        yaw = self._euler_rotation_vector[1].item()
        roll = self._euler_rotation_vector[2].item()

        cyaw = math.cos(yaw)
        syaw = math.sin(yaw)
        cpitch = math.cos(pitch)
        spitch = math.sin(pitch)
        croll = math.cos(roll)
        sroll = math.sin(roll)

        # Convert from Euler angles using the yaw-pitch-roll
        # Tait-Bryan convention.
        m00 = cyaw * cpitch
        m01 = syaw * sroll - cyaw * spitch * croll
        m02 = cyaw * spitch * sroll + syaw * croll
        m10 = spitch
        m11 = cpitch * croll
        m12 = -cpitch * sroll
        m20 = -syaw * cpitch
```

```
        m21 = syaw * spitch * croll + cyaw * sroll
        m22 = -syaw * spitch * sroll + cyaw * croll

        self._rotation_matrix[0, 0] = m00
        self._rotation_matrix[0, 1] = m01
        self._rotation_matrix[0, 2] = m02
        self._rotation_matrix[1, 0] = m10
        self._rotation_matrix[1, 1] = m11
        self._rotation_matrix[1, 2] = m12
        self._rotation_matrix[2, 0] = m20
        self._rotation_matrix[2, 1] = m21
        self._rotation_matrix[2, 2] = m22

        self._rodrigues_rotation_vector, jacobian = cv2.Rodrigues(
            self._rotation_matrix, self._rodrigues_rotation_vector)
```

Note

Note that `cv2.Rodrigues` is a two-way function, which can convert from a Rodrigues rotation vector to a rotation matrix, or vice versa. If we pass it a rotation matrix, it returns a Rodrigues vector (and Jacobian matrix); if we pass it a Rodrigues vector, it returns a rotation matrix (and Jacobian matrix).

By this point, we have almost fully explored the implementation of our 3D tracking algorithm, including the use of the Kalman filter to stabilize the 6DOF pose, as well as the velocity and acceleration. Along the way, we have dealt with conversions between different representations of the rotation. Now, let's turn our attention to two final implementation details of our `ImageTrackingDemo` class: the AR drawing methods and the creation of a mask based on the tracking results.

Drawing the tracking results and masking the tracked object

We will implement one helper method, `_draw_object_axes`, to draw a visualization of the tracked object's x, y, and z axes. We will also implement another helper method, `_make_and_draw_object_mask`, to project the object's vertices from 3D to 2D, create a mask based on the object's silhouette, and tint this masked region yellow as a visualization.

Let's start with the implementation of `_draw_object_axes`. We can consider it in three stages:

1. First, we want to take a set of 3D points located along the axes and project these points to the 2D image space. Remember that we defined the 3D axis points in our `__init__`

method, in the *Initializing the tracker* section. They will simply serve as endpoints of the axis arrows that we will draw. Using the `cv2.projectPoints` function, our 6DOF tracking result, and our camera matrix, we can find the 2D projected points as follows:

```
    def _draw_object_axes(self):

        points_2D, jacobian = cv2.projectPoints(
            self._reference_axis_points_3D,
            self._rodrigues_rotation_vector,
            self._translation_vector, self._camera_matrix,
            self._distortion_coefficients)
```

Note

Besides returning the projected 2D points, `cv2.projectPoints` also returns the **Jacobian matrix**, which represents the partial derivatives (with respect to the input parameters) of the function used to calculate the 2D points. This information is potentially useful for camera calibration, but we do not use it in our example.

2. The projected points are in floating-point format, but we will need integers to pass to OpenCV's drawing functions. Thus, we perform the following conversions to integer format:

```
        origin = (int(points_2D[0, 0, 0]), int(points_2D[0, 0, 1]))
        right = (int(points_2D[1, 0, 0]), int(points_2D[1, 0, 1]))
        up = (int(points_2D[2, 0, 0]), int(points_2D[2, 0, 1]))
        forward = (int(points_2D[3, 0, 0]), int(points_2D[3, 0, 1]))
```

3. Having calculated the endpoints, we can now draw three arrowed lines to represent the x, y, and z axes:

```
        # Draw the X axis in red.
        cv2.arrowedLine(
            self._bgr_image, origin, right, (0, 0, 255), 2)

        # Draw the Y axis in green.
        cv2.arrowedLine(
            self._bgr_image, origin, up, (0, 255, 0), 2)
```

```
        # Draw the Z axis in blue.
        cv2.arrowedLine(
            self._bgr_image, origin, forward, (255, 0, 0), 2)
```

We have finished implementing `_draw_object_axes`. Now, let's turn our attention to `_make_and_draw_object_mask`, which we can also consider in terms of three steps:

1. Like the previous function, this one begins by projecting points from 3D to 2D. This time, we are projecting the reference object's vertices, which we defined in our `__init__` method, in the *Initializing the tracker* section. Here is the projection code:

```
    def _make_and_draw_object_mask(self):

        # Project the object's vertices into the scene.
        vertices_2D, jacobian = cv2.projectPoints(
            self._reference_vertices_3D,
            self._rodrigues_rotation_vector,
            self._translation_vector, self._camera_matrix,
            self._distortion_coefficients)
```

2. Again, we convert the projected points from a floating-point format to an integer format (as OpenCV's drawing functions expect integers):

```
        vertices_2D = vertices_2D.astype(numpy.int32)
```

3. The projected vertices form a convex polygon. We can paint the mask black (as a background), and then draw this convex polygon in white:

```
        # Make a mask based on the projected vertices.
        self._mask.fill(0)
        for vertex_indices in \
                self._reference_vertex_indices_by_face:
            cv2.fillConvexPoly(
                self._mask, vertices_2D[vertex_indices], 255)
```

   Remember that our `_track_object` method will use this mask when it processes the next frame. Specifically, `_track_object` will only look for keypoints in the masked region. Therefore, it will attempt to find the object in the region where we recently found it.

> **Tip**
>
> Potentially, we could improve this technique by applying a morphological dilation operation to expand the masked region. In this way, we would search for the object not only in the region where we recently found it but also in the surrounding region.

4. Now, in the BGR frame, let's highlight the masked region in yellow in order to visualize the shape of the tracked object. To make a region more yellow, we can subtract a value from the blue channel. The `cv2.subtract` function suits our purpose because it accepts an optional mask argument. Here is how we use it:

```
        # Draw the mask in semi-transparent yellow.
        cv2.subtract(
            self._bgr_image, 48, self._bgr_image, self._mask)
```

> **Tip**
>
> When we tell `cv2.subtract` to subtract a single scalar value such as 48 from an image, it subtracts the value only from the image's first channel – in this case (and most cases), the blue channel of a BGR image. This is arguably a bug, but it is convenient for tinting things yellow!

That was the last method in the `ImageTrackingDemo` class. Now, let's bring the demo to life by instantiating this class and calling its `run` method!

Running and testing the application

To complete the implementation of `ImageTrackingDemo.py`, let's write a `main` function that launches the application with a specified capture device, FOV, and target frame rate:

```
def main():

    capture = cv2.VideoCapture(0)
    capture.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
    capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)
    diagonal_fov_degrees = 70.0
    target_fps = 25.0
```

```
    demo = ImageTrackingDemo(
        capture, diagonal_fov_degrees, target_fps)
    demo.run()

if __name__ == '__main__':
    main()
```

Here, we are using a capture resolution of 1280x720, a diagonal FOV of 70 degrees, and a target frame rate of 25 FPS. You should choose parameters that are appropriate for your camera and lens, and for the speed of your system.

Let's suppose we run the application and it loads the following image from `reference_image.png`:

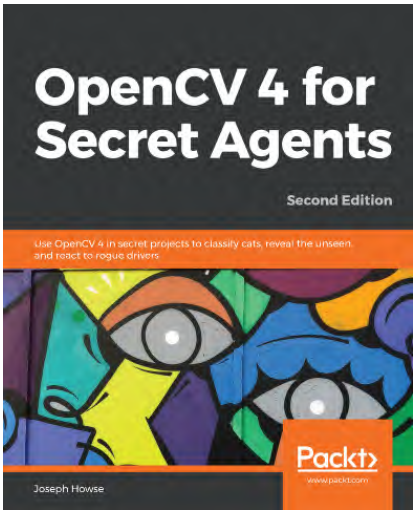


Figure 9.5: A book cover, which we will use as an image to track

Note

This is, of course, the cover of *OpenCV 4 for Secret Agents* (Packt Publishing, 2019), a book by Joseph Howse. Not only is it a vault of secret knowledge, but it is also a good target for image tracking. You should buy a print copy!

During initialization, the application saves the following visualization of the reference keypoints to a new file called `reference_image_keypoints.png`:

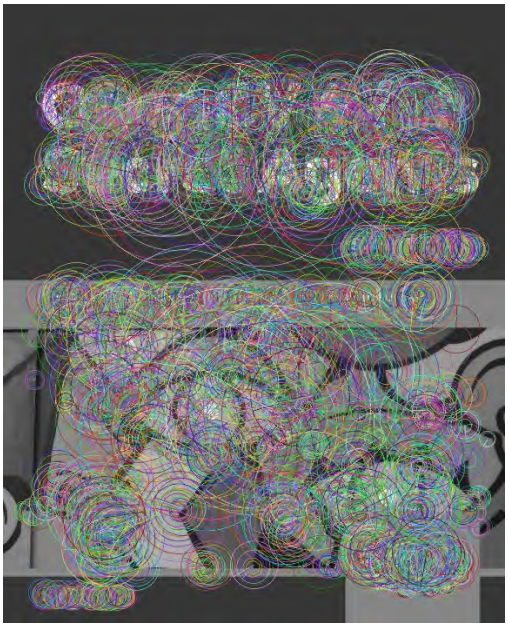


Figure 9.6: Visualization of ORB descriptors for a book cover

We have seen this type of visualization before in *Chapter 6, Retrieving Images and Searching Using Image Descriptors*. The large circles represent keypoints that can be matched at a small scale (for example, when we view the printed image from a large distance or with a low-resolution camera). The small circles represent keypoints that can be matched at a large scale (for example, when we view the printed image at a close distance or with a high-resolution camera). The best keypoints are the ones marked by many concentric circles, since these can be matched at various scales. Within each circle, the radial line represents the normal orientation of the keypoint.

Studying this visualization, we can infer that this image's best keypoints are concentrated in the high-contrast text (white against dark gray) in the top part of the image. Other useful keypoints are found in many regions, including the high-contrast lines (black against saturated colors) in the bottom part of the image.

Next, we see a camera feed. When we put a print of the reference image in front of the camera, we see an AR visualization of the tracking results:



Figure 9.7: Results of image tracking and AR visualization with a book cover

Of course, the preceding screenshot shows a nearly frontal view of the book cover. The axis directions are drawn as expected. The *x* axis (in red) points to the book cover's right (the viewer's left). The *y* axis (in green) points up. The *z* axis (in blue) points forward from the book cover (toward the viewer). As an AR effect, a semitransparent yellow highlight is superimposed atop the tracked book cover (including the part covered up by Joseph Howse's index finger and middle finger). The positions of the small green and red dots show that in this frame, the good keypoint matches are concentrated in the region of the book's title, and most of these good matches are inliers for `cv2.solvePnPRansac`.

Note

If you are reading the print edition of this book, the screenshots are reproduced in grayscale. To make the *x*, *y*, and *z* axes easier to distinguish in a grayscale print, text labels have been added to the screenshots manually; these text labels are not part of the program's output.

Because we took care to find good keypoints in several regions throughout the image, the tracking can succeed even when a large part of the tracked image is in shadow, covered up, or outside the frame. For example, in the following screenshot, the axis directions and highlighted region are correct, even though most of the book cover (including nearly all of the book's title, with the best keypoints) is outside the frame:

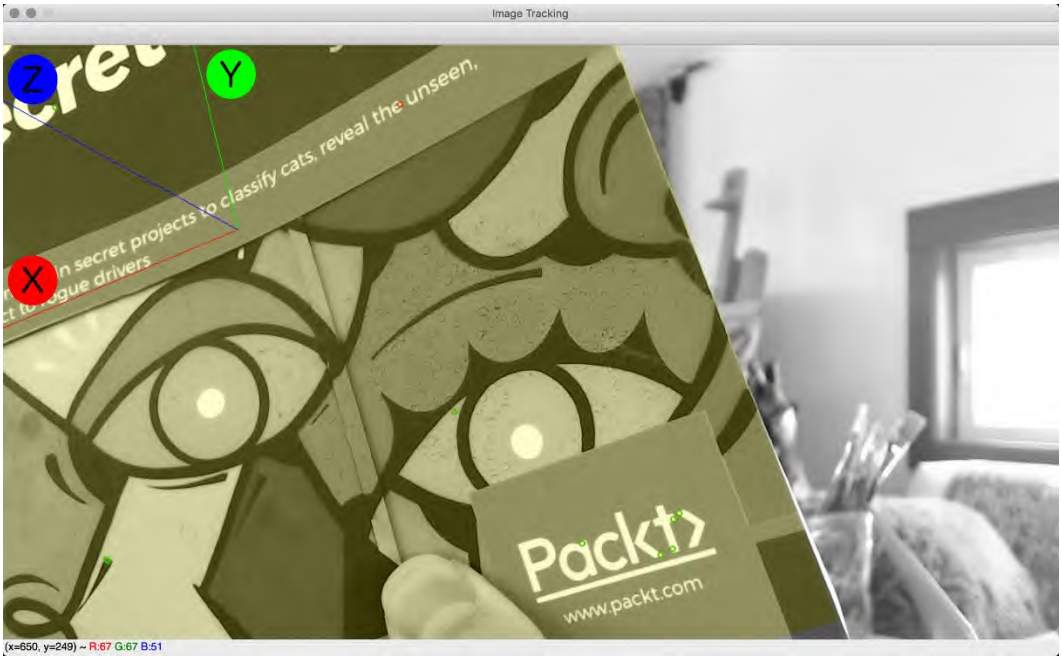


Figure 9.8: Results of image tracking and AR visualization when only part of the book cover is visible

Go ahead and conduct your own experiments with various reference images, cameras, and viewing conditions. Try various resolutions for the reference image and camera. Remember to measure your camera's FOV and adjust the FOV argument accordingly. Study the keypoint visualizations and the tracking results. What kinds of input yield good (or bad) tracking results in our demo?

Note

If you find it inconvenient to use printed images for tracking, you can instead point your camera at a screen (such as a smartphone screen) where you are displaying the image you want to track. Because a screen is backlit (and it might be glossy, too), it might not give a faithful representation of how a printed image would look in any given scene, but it typically works well for a tracker's purposes.

Once you have experimented to your heart's content, let's consider some of the ways our 3D tracker could be improved.

Improving the 3D tracking algorithm

Essentially, our 3D tracking algorithm combines three approaches:

1. Find a 6DOF pose with a PnP solver, whose inputs depend on FLANN-based matches of ORB descriptors
2. Use a Kalman filter to stabilize the 6DOF tracking result
3. If an object was tracked in the previous frame, use a mask to limit the search to the region where the object is now most likely to be found

Moreover, it is worth noting that our tracker just uses input from a conventional monocular camera (not a depth camera). Often, commercial solutions for 3D tracking involve additional approaches and additional inputs.

We have relied on successfully using a pretrained descriptor matcher and a PnP solver for every frame; however, a more complex algorithm may provide some alternatives as fallbacks or as cross-checking mechanisms. This is in case the descriptor matcher and PnP solver miss the object in some frames, or in case they are too computationally expensive to use for every frame. The following alternatives are widely used:

- Update the previous keypoint matches based on optical flow, and update the previous 6DOF pose based on the homography between the keypoints' old and new positions (according to the optical flow).
- Update the rotational component of the 6DOF pose based on an accelerometer, gyroscope, and magnetometer (compass). Typically, even in consumer devices, these sensors together can successfully measure small or large changes in rotation. Often, this set of sensor data is used in combination with a Kalman filter, and this filtering can be done in a way that avoids the singularity problems associated with conventional Euler angles. For example, see the paper *Euler Angle Based Attitude Estimation Avoiding the Singularity Problem*, by Chul Woo Kang and Chan Gook Park; it is available at `www.sciencedirect.com/science/article/pii/S1474667016439212`.
- Update the positional component of the 6DOF pose based on a barometer and GPS. Typically, in consumer devices, a barometer can measure changes in altitude with an accuracy of approximately 10 cm, while GPS can measure changes in longitude and latitude with an accuracy of approximately 10 m. Depending on the use case, these may or may not be usable levels of accuracy. If we are attempting to perform AR on large and distant features of a landscape – for example, if we want to draw a virtual dragon perched atop a real mountaintop – then 10 m accuracy may be fine. For detailed work –

for example, if we want to draw a virtual ring on a real finger – 10 m or even 10 cm accuracy is unusable.

- Update the positional acceleration component of the Kalman filter based on an **accelerometer** (a sensor that measures acceleration). Typically, in consumer devices, accelerometers suffer from drift (a tendency for errors to exhibit runaway growth in one direction or another), so this option should be approached with caution.
- Dynamically retrain the matcher, potentially with the help of a depth camera, to gather new data about an object's geometry in real time. All of this can be approached as a machine learning problem. Notably, Magic Leap's SuperGlue tracking algorithm achieves real-time retraining on the basis of GPU-accelerated neural networks and depth camera data. For details, see the SuperGlue researchers' webpage at `https://psarlin.com/superglue/`.

These alternative techniques are beyond the scope of this book – and, indeed, some of them are not computer vision techniques – so we leave them to you for independent study.

A final word: Sometimes, significant improvements in tracking results are achievable by altering a preprocessing algorithm rather than the tracking algorithm. Earlier in this chapter, in the *Performing grayscale conversion* section, we mentioned Macêdo, Melo, and Kelner's paper on grayscale conversion algorithms and SIFT descriptors. You may wish to read that paper and conduct your own experiments to determine how the choice of grayscale conversion algorithm affects the number of tracking inliers when using ORB descriptors or other types of descriptors.

Summary

This chapter introduced AR, along with a robust set of approaches to the problem of tracking an image in 3D space.

We began by learning the concept of 6DOF tracking and different ways of representing rotations. We recognized that familiar tools such as ORB descriptors, FLANN-based matching, and Kalman filtering are useful in this kind of tracking, but that we also needed to work with camera and lens parameters in order to solve the PnP problem.

Next, we addressed practical considerations of how best to represent a reference object (such as a book cover or a photo print) in the form of a grayscale image, a set of 2D keypoints, and a set of 3D keypoints.

We proceeded to implement a class that encapsulated a demo of image tracking in 3D space, with a 3D highlighting effect as a basic form of AR. Our implementation dealt with real-time considerations, such as the need to update the Kalman filter's transition matrix based on fluctuations in the frame rate.

Finally, we considered ways to potentially improve the 3D tracking algorithm using additional computer vision techniques, or other sensor-based techniques.

Next, in one of the most advanced chapters of this book, we will tackle even bigger challenges in the real, three-dimensional world; we will scan and navigate a whole 3D environment in real time!

10

Introduction to 3D Navigation, Odometry, and Point Clouds

Previously, in *Chapter 9*, *Camera Models and Augmented Reality*, we tracked a planar surface (specifically, a 2D printed image) with **6 degrees of freedom** (**6DOF**) in a 3D coordinate space centered on our camera. Now, suppose that instead, the object we want to track is the camera itself and we have no point of reference except the camera's previous pose. We must match visual features in the current camera frame to visual features in the previous frame and thereby estimate the changes in the camera's position and rotation. This problem is called **visual odometry**.

More generally, **odometry** is the process of measuring one's steps, and in this case, the process is visual. Odometry tells us how far – and in which direction – we have traveled, so it is an essential component of a 3D navigation system. For example, self-driving cars and self-piloting drones must solve odometry problems (in conjunction with other navigational problems) in order to maintain precise tracking and control of the vehicle at all times.

Typically, to produce accurate results, visual odometry algorithms need at least two types of input: color frames (such as RGB frames) and depth frames. Thus, we will put visual odometry into practice using an OAK-D stereo camera, which can capture both RGB and depth frames, as we saw in *Chapter 4*, *Depth Estimation and Segmentation*.

While this chapter focuses on visual odometry, we will also take note of some other common components of 3D navigation systems. Specifically, we will examine the manner in which a 3D scene can be represented as a **point cloud** (a collection of 3D points), and we will look at a couple of 3D segmentation techniques.

As we progress through the chapter, we will tackle the following tasks:

- Understand the concept of visual odometry, its usefulness, its limitations, and ways of overcoming these limitations.
- Build a visual odometry application using an OAK-D camera, the `depthai` Python module, and OpenCV's `Odometry` class. As part of this application, we will solve the following problems:
    - Define the camera matrix for the OAK-D camera.
    - Align each depth frame to match the perspective of the corresponding RGB frame.
    - Estimate the change in the camera's position and rotation between consecutive pairs of RGB and depth frames.
    - Over multiple frames, keep track of the cumulative change in the camera's position and rotation so that we know how far it has moved since the start of the application.
    - Visualize each RGB frame, depth frame, and cumulative odometry result, all in one GUI window.
- Understand point clouds and how they may be generated.
- Write a set of scripts to load, segment, and visualize point cloud data using OpenCV, scikit-learn, and Matplotlib. As part of this work, we will experiment with two segmentation algorithms called **Balanced Iterative Reducing and Clustering using Hierarchies** (**BIRCH**) and **Density-Based Spatial Clustering of Applications with Noise** (**DBSCAN**).

By the time we reach this chapter's end, you will have a clear, practical understanding of visual odometry in the context of OpenCV, `depthai`, and the OAK-D hardware. You will also know how to perform segmentation of a point cloud, as a step toward advanced analysis of objects in 3D scenes.

Technical requirements

Like the rest of this book, this chapter uses Python, OpenCV, and NumPy. Please refer to *Chapter 1*, *Setting Up OpenCV*, for installation instructions.

Like *Chapter 4*, *Depth Estimation and Segmentation*, this chapter uses an OAK-D stereo camera and the `depthai` Python module to capture depth images. Please refer to the *Technical requirements* section of *Chapter 4* for instructions about setting up the OAK-D device and `depthai`.

Additionally, this chapter uses Matplotlib (to plot 3D points) and scikit-learn (to segment 3D points into clusters, which may be objects). If you have not already installed these Python modules, install them now by running `$ pip install matplotlib scikit-learn` (or `$ pip3 install matplotlib scikit-learn`, depending on your environment).

The complete code for this chapter can be found in this book's GitHub repository in the `chapter10` folder. The sample 3D models can be found in the `3d_models` folder.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Note

For readers who do not have access to an OAK-D camera, the repository contains a recording of OAK-D color and depth frames, located at `videos/oak_bgr_and_depth.npz`. These pre-recorded frames are used by an odometry script located at `chapter10/odometry_captured_oak.py`. The latter script is a substitute for `chapter10/odometry_oak.py`, which we will examine in detail in this chapter, in the *Building a visual odometry application for OAK-D* section.

Understanding visual odometry

Visual odometry is the process of estimating a change in the 6DOF camera pose based on the analysis of camera frames and camera parameters. Like the 6DOF image tracking problem that we examined in *Chapter 9*, *Camera Models and Augmented Reality*, visual odometry may involve the use of a feature matcher and a **Perspective-n-Point** (**PnP**) solver, with iterative refinements to arrive at a result that minimizes the reprojection error. This result – that is, the camera's new pose relative to its old pose – can be expressed as the 3x3 rotation matrix R and 3-element translation vector t, a pair of formats that we saw in *Chapter 9*. Alternatively, R and t can be expressed together in a 4x4 **rigid transformation matrix**, as follows:

R_{00}	R_{01}	R_{02}	t_0
R_{10}	R_{11}	R_{12}	t_1
R_{20}	R_{21}	R_{22}	t_2
0	0	0	1

Note

OpenCV puts odometry results in the format of a 4x4 rigid transformation matrix, often called `Rt` in code samples.

If the frames provide depth data with a known scale, then, likewise, the visual odometry solution can be computed with a known scale. Otherwise, in the general case, the true scale is unknown and the solution is said to be *only accurate up to a scale factor*. For real-world navigation, odometry becomes pretty much useless if the scale is wrong, so we will focus on cases where depth frames are available and thus the true scale can be computed. Notably, the OAK-D stereo camera (via the `depthai` module) can capture depth frames with a known scale.

For visual odometry with depth data, OpenCV implements an algorithm called **Iterative Closest Point** (**ICP**). ICP seeks to minimize the reprojection error in terms of a metric called **Point-to-Plane Distance** (**PPD**), also known as **Point-to-Plane Error**. This objective penalizes a prospective odometry solution if the solution fails to place a pair of matched points on (or near) the same plane.

Note

For a detailed description of ICP and its relevance to odometry, see OpenCV's official documentation at `https://docs.opencv.org/doc/doxygen/html/d7/dbe/kinfu_icp.html`. Then, for background on an influential implementation of this technique, see the Microsoft Research paper, *KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera*, which is available at `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/kinectfusion-uist-comp.pdf`.

Even with the use of depth data, visual odometry has two significant limitations:

1. Subject motion and camera motion are, generally, visually indistinguishable from each other. If the moving camera is looking at other moving objects, visual odometry cannot reliably estimate the camera motion.
2. Navigation by visual odometry is a form of **dead reckoning**, meaning that it relies on a series of uncorrected estimates. Like all forms of dead reckoning, it suffers from drift or, in other words, it accumulates errors over time. Starting from a known pose, we may continually update our estimate of the current pose based on visual odometry

results for new frames, yet still, the only known pose was the starting pose, and we are unsure of the direction or magnitude of the errors we have subsequently accumulated.

For these reasons, state-of-the-art navigation systems may combine visual odometry with several other approaches in order to correct mistakes. We will consider such combinations later, in the *Extending the odometry application* section. First, though, we are going to explore visual odometry in practice.

Building a visual odometry application for OAK-D

Let's implement our visual odometry application in a new script called `odometry_oak.py`. We will use OpenCV for odometry and `depthai` to control the OAK-D camera. We will also use NumPy and Python's standard `math` library. At the start of our script, let's import all these modules as follows:

```
import math

import cv2
import depthai as dai
import numpy as np
```

Throughout the following subsections, as we code our application, we are going to explore the nuances of several tasks:

1. Configuring the odometry.
2. Configuring the OAK-D camera.
3. Computing an odometry result for a pair of frames.
4. Combining multiple odometry results from a series of frames.
5. Visualizing the results. Here, our end goal is to show the user a real-time estimate of how far the camera has moved and rotated since the start of the application. We will also show a stream of frames, with depth data superimposed onto color data.

To begin, let's see how our camera parameters affect the odometry configuration.

Defining the camera matrix and other odometry parameters

As part of its configuration, OpenCV's `Odometry` class uses a camera matrix. This is the exact same kind of camera matrix as the one we used in *Chapter 9*, *Camera Models and Augmented Reality*. You may recall that we can compute an idealized camera matrix based on just three parameters: image width (in pixels), image height (in pixels), and diagonal **field of view** (**FOV**). For details, refer back to *Chapter 9*, particularly the *Understanding camera and lens*

parameters section. To encapsulate the code that builds the camera matrix, let's add the following helper function to our `odometry_oak.py` script:

```
def createCameraMatrix(w, h, diagonal_fov_degrees):
    diagonal_image_size = (w ** 2.0 + h ** 2.0) ** 0.5
    diagonal_fov_radians = \
        diagonal_fov_degrees * math.pi / 180.0
    focal_length = 0.5 * diagonal_image_size / math.tan(
        0.5 * diagonal_fov_radians)
    return np.array(
        [[focal_length, 0.0, 0.5 * w],
         [0.0, focal_length, 0.5 * h],
         [0.0, 0.0, 1.0]], np.float32)
```

Note that this helper function's body is similar to the code we used in *Chapter 9*, in *step 4* of *Initializing the tracker*. We are using trigonometry to compute the focal length in pixel-equivalent units because OpenCV's `Odometry` class expects the camera matrix to use such units. Then, we are constructing the camera matrix based on the focal length and image dimensions.

Later in this chapter, we will add more helper functions to our `odometry_oak.py` script but, for now, let's proceed with the script's main application code.

We are going to configure our OAK-D device to use an RGB image resolution of 1,920 x 1,080. For most OAK-D hardware, the diagonal FOV of the RGB camera is 81 degrees. Let's define variables for these camera parameters as follows:

```
w = 1920
h = 1080
diagonal_fov_degrees = 81.0
```

Tip

Check the FOV specifications for your OAK-D model at `https://docs.luxonis.com/projects/hardware/en/latest/`. Remember, you want to use the RGB camera's diagonal FOV, listed as *DFOV* in the specifications. At the time of writing, the most common value is 81 degrees, though for some models (designated as *wide* models) the value is 120 degrees.

Now, let's proceed to define our odometry parameters, starting with the type of odometry solution we want to find. OpenCV offers the following three options:

- `cv2.ODOMETRY_TYPE_DEPTH`: The odometry solution will minimize the reprojection error for the depth image in terms of PPD.
- `cv2.ODOMETRY_TYPE_RGB`: The odometry solution will minimize the reprojection error for the RGB image.
- `cv2.ODOMETRY_TYPE_RGB_DEPTH`: The odometry solution will minimize some weighted combination of depth-based and RGB-based reprojection errors.

Note

Note that the odometry types differ in terms of how they define an optimal solution, not in terms of their required inputs. All types require both depth input and RGB input.

We will aim for a balanced solution using `ODOMETRY_TYPE_RGB_DEPTH`. Let's define our choice as follows:

```
odometryType = cv2.ODOMETRY_TYPE_RGB_DEPTH
```

Next, we will create an instance of `cv2.OdometrySettings`, a class that encapsulates a large number of odometry parameters, including the camera matrix. Also, we will initialize the camera matrix using our helper function, and we will assign the camera matrix to the odometry settings, as follows:

```
odometrySettings = cv2.OdometrySettings()
odometrySettings.setCameraMatrix(createCameraMatrix(
    w, h, diagonal_fov_degrees))
```

If we did not assign our own camera matrix, then by default, the odometry settings would use a matrix based on an image resolution of 640x480 and a diagonal FOV of 74.6 degrees. This default is badly mismatched with our OAK-D camera parameters, so it would give us inaccurate odometry results.

The `OdometrySettings` class also provides getter and setter methods for many other parameters. Although we will use the defaults in our case, here are a few of the setter methods that you may want to use in the future:

- `setMinDepth(val)`: Here, `val` is a depth measurement in meters. Pixels with a depth less than `val` will not be used. The default (if we do not call this method to override it) is `0.0`.
- `setMaxDepth(val)`: Here, `val` is a depth measurement in meters. Pixels with a depth greater than `val` will not be used. The default is `4.0`. Thus, by default, we assume that depths beyond 4 m are not useful for estimating frame-to-frame motion. If our frames had a higher resolution or a narrower FOV, we might consider raising this threshold to take advantage of the greater level of detail in such frames.

`setMaxDepthDiff(val)`: Here, `val` is a depth measurement in meters. Matches between the pairs of frames will be filtered out if the difference in depth values is greater than `val`. The default is `0.07`. Thus, by default, we assume that the camera is moving no faster than 7 cm per frame along the depth axis. We might consider raising this value if we expect a low frame rate or fast movement.

Note

For more options, see the official documentation on `OdometrySettings` at `https://docs.opencv.org/5.0/main_modules/classcv_1_1OdometrySettings.html`.

Finally, we must choose between two further subtypes of odometry algorithms: `cv2.ODOMETRY_ALGO_TYPE_COMMON` (which favors accuracy) and `cv2.ODOMETRY_ALGO_TYPE_FAST` (which favors speed). For real-time applications, the choice is rather complicated because the goals of accuracy and timeliness are, to some extent, inseparable. A solution that is accurate at one point may well become inaccurate if it arrives late. Moreover, if the odometry updates are less frequent, the odometry problem becomes harder because the camera may have moved farther. We will opt for `ODOMETRY_ALGO_TYPE_FAST` so that we can update the odometry more frequently:

```
odometryAlgoType = cv2.ODOMETRY_ALGO_TYPE_FAST
```

Now, with the odometry parameters we have just defined, we can initialize the `Odometry` object as follows:

```
odometry = cv2.Odometry(
    odometryType, odometrySettings, odometryAlgoType)
```

Perhaps you have been pondering a question about the camera matrix: why is there only one? After all, the OAK-D device has three cameras: a left monochrome camera, a right monochrome camera, and an RGB camera in between. As users of the `Odometry` class, it is our responsibility to align the depth image and the RGB image so that, effectively, the camera matrix becomes the same for everything. Let's take a look now at how to capture aligned images from the OAK-D device.

Aligning depth frames to match RGB frames

The `depthai` module gives us a convenient option to align depth frames with RGB frames so that the same camera matrix is applicable to both. Before we arrive at this configuration option, let's first review the process of initializing an imaging pipeline with `depthai`. We did this previously in *Chapter 4*, *Depth Estimation and Segmentation*. Specifically, in the *Stereo imaging with an OAK-D camera* section, we captured the disparity frames and RGB frames from an OAK-D camera. For our current application, the pipeline code will be similar, but we will `highlight` a few important differences in the code as we go along.

First, it is important to note that in `depthai`'s terminology, a depth frame and a disparity frame are different (but related) types of output. For our demo application in *Chapter 4*, we used *disparity* frames with an 8-bit grayscale depth. The disparity pixel values were low (dark) in distant regions and high (bright) in near regions. For our odometry application, we are going to use *depth* frames with a 16-bit grayscale depth. The depth pixel values will be distances measured in millimeters, so they will be high (bright) in distant regions and low (dark) in near regions.

Given the 16-bit range, the depth image is technically capable of representing distances from 0 mm up to 65,535 mm, though the OAK-D is not actually accurate across this entire range. Currently, most OAK-D hardware is designed so that, theoretically, it can measure depth in the

75 mm to 38,250 mm range, though the practical range may be narrower, depending partly on the lighting conditions. For details, see the official documentation:

- Documentation on minimum stereo depth distance: `https://docs.luxonis.com/software/depthaicomponents/nodes/stereo_depth#StereoDepth-Min%20stereo%20depth%20distance`
- Documentation on maximum stereo depth distance: `https://docs.luxonis.com/software/depthaicomponents/nodes/stereo_depth#StereoDepth-Max%20stereo%20depth%20distance`

Moving on with the code, let's create a `depthai` pipeline, along with nodes for the left and right stereo cameras and the processed depth image, as follows:

```
# Create the pipeline.
pipeline = dai.Pipeline()

# Define the nodes in the mono -> stereo -> output chain.
monoLeft = pipeline.create(dai.node.MonoCamera)
monoRight = pipeline.create(dai.node.MonoCamera)
depth = pipeline.create(dai.node.StereoDepth)
depthOut = pipeline.create(dai.node.XLinkOut)
depthOut.setStreamName('depth')
```

This bit of code is nothing new; we created a pipeline and nodes in exactly the same manner in *Chapter 4*. Similarly, we will proceed to configure the nodes with many of the same options as in *Chapter 4*:

```
# Configure the mono cameras.
monoResolution = dai.MonoCameraProperties.SensorResolution.THE_400_P
monoLeft.setResolution(monoResolution)
monoLeft.setBoardSocket(dai.CameraBoardSocket.CAM_B)
monoRight.setResolution(monoResolution)
monoRight.setBoardSocket(dai.CameraBoardSocket.CAM_C)

# Configure the depth node for depth output.

depth.setDefaultProfilePreset(
    dai.node.StereoDepth.PresetMode.DEFAULT)

# Median filter options:
#     MEDIAN_OFF
```

```
#     KERNEL_3x3
#     KERNEL_5x5
#     KERNEL_7x7 (default)
depth.initialConfig.setMedianFilter(dai.MedianFilter.KERNEL_7x7)

# True -> better occlusion handling
depth.setLeftRightCheck(True)
```

Next, we are coming to a new configuration option. The following line of code configures our depth node so that it will produce depth frames that match the resolution and perspective of the RGB frames:

```
# Align the depth output to the RGB output.
depth.setDepthAlign(dai.CameraBoardSocket.CAM_A)
```

Now, as we did in *Chapter 4*, we proceed to link the inputs and outputs in our stereo imaging chain. However, note that this time, we are linking a `depth.depth` output (which will give us 16-bit depth data measured in millimeters), not a `depth.disparity` output (which gave us 8-bit disparity data when we used it in *Chapter 4*):

```
# Link the nodes in the mono -> stereo -> output chain.
monoLeft.out.link(depth.left)
monoRight.out.link(depth.right)
depth.depth.link(depthOut.input)
```

Next, to set up the RGB imaging chain, we use exactly the same code as in *Chapter 4*. For convenience, it is reproduced here:

```
# Define the nodes in the RGB chain.
rgb = pipeline.create(dai.node.ColorCamera)
rgbOut = pipeline.create(dai.node.XLinkOut)
rgbOut.setStreamName('rgb')

# Configure the RGB camera.
rgb.setBoardSocket(dai.CameraBoardSocket.CAM_A)
rgb.setResolution(
    dai.ColorCameraProperties.SensorResolution.THE_1080_P)
rgb.setVideoSize(w, h)
```

```
# Link the nodes in the RGB -> output chain.
rgb.video.link(rgbOut.input)
```

So far, we have configured the `Odometry` object and the OAK-D camera based on the kinds of depth and RGB images we plan to capture. Next, let's write our application's main loop, where we will capture and process frames.

Preparing odometry frames and computing the odometry

With each iteration of our application's main loop, we will try to capture frames from our pipeline's depth and RGB queues. Then, if frames are available, we will use them to update our odometry results.

Let's open a connection to the OAK-D device and get the depth and RGB queues as follows:

```
# Connect to the device and start the pipeline.
with dai.Device(pipeline) as device:

    # Get the output queues.
    depthQ = device.getOutputQueue(
        name='depth', maxSize=4, blocking=False)
    rgbQ = device.getOutputQueue(
        name='rgb', maxSize=4, blocking=False)
```

Let's also define variables for the previous BGR frame (which is an RGB frame converted into OpenCV's format) and the previous odometry frame (which is a bundle of frame data in OpenCV's `cv2.OdometryFrame` format). Initially, these variables will be `None` because we start with no previous frames:

```
    lastBGRFrame = None
    lastOdometryFrame = None
```

Now, let's define a variable called `Rt` (short for rigid transformation), which will be a 4x4 transformation matrix representing the cumulative odometry result. We define the camera's initial position and rotation as the origin of our coordinate system, so `Rt` is initially an identity matrix:

```
    Rt = np.array([[1.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0],
                   [0.0, 0.0, 1.0, 0.0],
                   [0.0, 0.0, 0.0, 1.0]], dtype=np.float64)
```

Having initialized our variables, let's start the main loop. As in many of our demos, the loop will continue until the user presses any key:

```
    while cv2.waitKey(1) == -1:
```

For each iteration, we will check whether a new depth frame and a new RGB frame are available:

```
        # Grab the next depth frame, if any is ready.
        depthGrab = depthQ.tryGet()

        # Grab the next RGB frame, if any is ready.
        rgbGrab = rgbQ.tryGet()
```

If a new RGB frame is available, we will convert it into OpenCV's BGR format and assign it to our `lastBGRFrame` variable:

```
        if rgbGrab is not None:
            # Get the frame and convert it from its native NV12
            # encoding to OpenCV's BGR format.
            lastBGRFrame = rgbGrab.getCvFrame()
```

If a new depth frame is available, and either a new or old BGR frame is also available, we can attempt to update the odometry. First, let's get the depth frame:

```
        if depthGrab is not None and lastBGRFrame is not None:
            depthFrame = depthGrab.getFrame()
```

For odometry frames, OpenCV requires us to provide a mask to indicate which depth pixels are valid. (For a discussion of why depth images may contain some invalid pixels, refer back to *Chapter 4*, particularly the *Capturing frames from an OpenNI-compatible depth camera* section.) The mask should use the value `0` for an invalid pixel and `255` for a valid pixel. Although `depthai` does not provide a mask as a separate type of image, it does have a similar way of representing invalid pixels; these have the value `0` in the depth frame. Thus, we can extract a mask from the depth frame as follows:

```
            # Invalid pixels have the value 0 in the depth map.
            mask = np.where(
                depthFrame == 0, 0, 255).astype(np.uint8)
```

We can bundle together the depth frame, BGR frame, and mask by creating a `cv2.OdometryFrame` object, as follows:

```
            odometryFrame = cv2.OdometryFrame(
                depthFrame, lastBGRFrame, mask)
```

Now, if we have a previous odometry frame, we can try to solve for the transformation between the previous and current frames. To do so, we pass the pair of odometry frames to two methods of the `Odometry` class: first, the `prepareFrames` method and then the `compute` method, as follows:

```
            if lastOdometryFrame is not None:
                odometry.prepareFrames(
                    lastOdometryFrame, odometryFrame)
                success, RtStep = odometry.compute(
                    lastOdometryFrame, odometryFrame)
```

The `compute` method returns two values: first, a Boolean, which is `True` if the odometry algorithm successfully converged on a solution; and second, a 4x4 transformation matrix, which represents the solution (if a solution was found, of course).

Note

For depth frames in 16-bit integer format, OpenCV's odometry implementation assumes that the unit is 1 mm. (This is correct in our case.) Then, for the floating-point result in `RtStep`, OpenCV converts the translational (positional) values to a unit of 1 m (not 1 mm).

Now, if the odometry algorithm did converge on a solution for the previous and current frames, we can proceed to update the cumulative odometry result. We will do so by calling a helper function, `accumulateResult`, which we will implement in the next section of this chapter. Here is the call to `accumulateResult`:

```
                if success:
                    accumulateResult(Rt, RtStep)
```

Regardless of whether the cumulative odometry result is old or is updated, we can visualize it by calling another helper function, `showResult`:

```
            showResult(Rt, lastBGRFrame, depthFrame)
```

We will implement `showResult` a little later in this chapter, in the *Visualizing the frames and odometry results* section.

Finally, we assign the current odometry frame to our `lastOdometryFrame` variable because we will use it again in a subsequent iteration's odometry update:

```
            lastOdometryFrame = odometryFrame
```

That is the end of our odometry application's main loop. We have seen how to integrate `depthai` and OpenCV's `Odometry` API, but we still need to process the odometry results a little bit to make them intelligible. We will proceed to write a couple of helper functions for this purpose.

Accumulating the odometry result over multiple frames

Going back to earlier lines in our `odometry_oak.py` script, let's add the `accumulateResult` helper function after the `createCameraMatrix` helper function. The function signature of `accumuluateResult` is as follows:

```
def accumulateResult(Rt, RtStep):
```

Here, `Rt` is the 4x4 rigid transformation matrix containing the cumulative odometry result so far. `RtStep` is another 4x4 rigid transformation matrix, containing the pairwise odometry result for the previous and current frames. We want to update `Rt` based on the change represented by `RtStep`.

Tip

You may be tempted to think that we could perform the update just by multiplying the two 4x4 transformation matrices: in pseudocode, `Rt := Rt.transpose() @ RtStep`, where the `@` symbol represents matrix multiplication. However, this naïve formulation is incorrect. For `Rt`, the Cartesian coordinate axes are based on the camera's initial rotation; for `RtStep`, this coordinate system has already been rotated. Thus, we need to decompose `Rt` and `RtStep` and deal with the difference in rotation between the two coordinate systems. We shall do so in our implementation of `accumulateResult`.

First, let's decompose each of the 4x4 transformation matrices into a 3x3 rotation matrix and a 1x3 translation (or position) vector, as follows:

```
    # Get the 3x3 rotation submatrices
    # and 1x3 translation submatrices.
    rotation = Rt[:3,:3]
    translation = Rt[:,3][:3]
    rotationStep = RtStep[:3,:3]
    translationStep = RtStep[:,3][:3]
```

We can update the translation by adding the two 1x3 translation vectors, though first, we must convert them into a common coordinate system, accounting for the camera's cumulative rotation before now: in pseudocode, `translation := translation + rotation.transpose() @ translationStep`. Here is the corresponding Python code, which uses the `cv2.gemm` function for **Generalized Matrix Multiplication** (**GeMM**):

```
    # Update the translation.
    translation += cv2.gemm(
        rotation, translationStep,
        1.0, None, 0.0).squeeze()
```

Note

For `gemm`, the function signature is `cv2.gemm(src1, src2, alpha, src3, beta, dst=None)`. Here, `src1`, `src2`, `src3`, and `dst` are matrices, while `alpha` and `beta` are scalars. If `dst` is `None`, a new matrix is created as `dst`. The function always returns `dst`.

The function's result, in pseudocode, is `dst := alpha * src1.transpose() @ src2 + beta * src3`.

If `src3` is `None`, the result is simplified as `dst := alpha * src1.transpose() @ src2`.

Now, we can update the rotation by multiplying the two 3x3 rotation matrices: in pseudocode, `rotation := rotation.transpose() @ rotationStep`. Here is the corresponding Python code:

```
    # Update the rotation.
    cv2.gemm(rotation, rotationStep,
        1.0, None, 0.0, rotation)
```

Note that our `translation` and `rotation` variables refer to subarrays of `Rt`, so by this point, the relevant values in `Rt` are already updated. We do not need to assign or return anything.

Next, in the `showResult` helper function, let's consider how to visualize the data in `Rt`, as well as the image data in the BGR and depth frames.

Visualizing the frames and odometry results

So far, we have implemented all the necessary code to capture color frames and depth frames and to compute a cumulative odometry result over a series of frames. This result is a 4x4 transformation matrix, `Rt`, which represents the camera's current 6DOF pose relative to its starting pose. Now, we just need a good way to visualize the latest frames and the latest `Rt`.

Our visualization technique will be simple. We will superimpose the depth frame onto the BGR frame, and we will draw text atop the blended frame to show the camera's rotation and position. We will implement all this in a helper function with the following signature:

```
def showResult(Rt, bgrFrame, depthFrame):
```

Let's begin this function's implementation by converting the 4x4 transformation matrix, `Rt`, into a more human-readable representation of the 6DOF pose. We will represent the rotation in terms of Euler angles (pitch, yaw, and roll) and the position in terms of XYZ coordinates. You may recall that we previously converted a 3x3 rotation matrix into Euler angles in *Chapter 9*, *Camera Models and Augmented Reality*, specifically in the *Converting between Rodrigues vectors and Euler vectors* section. As the 4x4 transformation matrix contains a 3x3 rotation matrix, we can reuse essentially the same conversion code. Here it is, with `highlighting` to show the trivial changes:

```
    m00 = Rt[0, 0]
    m02 = Rt[0, 2]
    m10 = Rt[1, 0]
    m11 = Rt[1, 1]
    m12 = Rt[1, 2]
    m20 = Rt[2, 0]
    m22 = Rt[2, 2]

    # Convert to Euler angles using the yaw-pitch-roll
    # Tait-Bryan convention.
    if m10 > 0.998:
        # The rotation is near the "vertical climb" singularity.
        pitch = 0.5 * math.pi
        yaw = math.atan2(m02, m22)
```

```
            roll = 0.0
        elif m10 < -0.998:
            # The rotation is near the "nose dive" singularity.
            pitch = -0.5 * math.pi
            yaw = math.atan2(m02, m22)
            roll = 0.0
        else:
            pitch = math.asin(m10)
            yaw = math.atan2(-m20, m00)
            roll = math.atan2(-m12, m11)
```

So far, the Euler angles are in radians. For readability, we will convert them to degrees using NumPy's `rad2deg` function, as follows:

```
        eulerDegrees = np.rad2deg([pitch, yaw, roll])
```

Also, we will extract the translation (position) vector from the transformation matrix, as follows:

```
        translation = Rt[:,3][:3]
```

Now, let's turn our attention to the data in the depth frame. As discussed earlier in the *Aligning depth frames to RGB frames* section, the depth frame contains 16-bit integer values; the unit is 1 mm, and the upper part of this 16-bit numeric range is beyond the practical measurement range for all current OAK-D hardware. For the purposes of visualization, we will convert the integer range to an 8-bit range where the unit is 40 mm. Thus, we will truncate the depth measurements at an arbitrary ceiling of `255 * 40 mm = 10,200 mm = 10.2 m`. The following code performs the conversion and truncation:

```
        # Scale the depth map for better visualization.
        depthFrame = np.minimum(
            255.0, (depthFrame / 40.0)).astype(np.uint8)
```

Next, let's apply a false colorization scheme to the depth frame. We will use OpenCV's `cv2.applyColorMap` function with the `cv2.COLORMAP_COOL` option to turn the dark (near)

regions cyan and the bright (far) regions magenta, with various shades of blue and purple in between. Here is the relevant code:

```
    # Apply false colorization to the depth map.
    depthFrame = cv2.applyColorMap(
        depthFrame, cv2.COLORMAP_COOL)
```

We will superimpose the colorized depth frame onto the BGR frame by taking a weighted average of the two frames, as follows:

```
    # Blend the BGR frame and the colorized depth map.
    blendedFrame = cv2.addWeighted(
        bgrFrame, 0.7, depthFrame, 0.3, 0.0)
```

On some displays, the frame's resolution of 1,920x1,080 may be inconveniently large, so we will scale it down to 960x540, as follows:

```
    # Resize the blended frame for display.
    blendedFrame = cv2.resize(blendedFrame, (960, 540))
```

Now, we will draw text on the frame to show the Euler angles (in degrees) and the XYZ coordinates (in meters):

```
    # Print the odometry result on the blended frame.
    textColor = (64, 192, 192)
    cv2.putText(blendedFrame,
                'yaw (degrees): %f' % eulerDegrees[0],
                (5, 25), cv2.FONT_HERSHEY_SIMPLEX, 1,
                textColor, 2)
    cv2.putText(blendedFrame,
                'pitch (degrees): %f' % eulerDegrees[1],
                (5, 55), cv2.FONT_HERSHEY_SIMPLEX, 1,
                textColor, 2)
    cv2.putText(blendedFrame,
                'roll (degrees): %f' % eulerDegrees[2],
                (5, 85), cv2.FONT_HERSHEY_SIMPLEX, 1,
                textColor, 2)
    cv2.putText(blendedFrame,
                'x (meters): %f' % translation[0],
                (5, 130), cv2.FONT_HERSHEY_SIMPLEX, 1,
                textColor, 2)
```

```
        cv2.putText(blendedFrame,
                    'y (meters): %f' % translation[1],
                    (5, 160), cv2.FONT_HERSHEY_SIMPLEX, 1,
                    textColor, 2)
        cv2.putText(blendedFrame,
                    'z (meters): %f' % translation[2],
                    (5, 190), cv2.FONT_HERSHEY_SIMPLEX, 1,
                    textColor, 2)
```

Finally, we will show the visualization in a window:

```
        # Show the blended frame.
        cv2.imshow('Result', blendedFrame)
```

This completes the code for our `odometry_oak.py` script. You can plug in an OAK-D camera, run the script, and see a window similar to the following:



Figure 10.1a: A screenshot of our visual odometry application

This frame was captured shortly after the start of the application, so the camera has not moved from its initial pose. Accordingly, the angles and coordinates of the cumulative odometry result are nearly zero. The foreground contains a table, which is tinted cyan; the middle ground contains a speaker and a sofa, which are tinted a deeper shade of blue; and in the background, a staircase and wall are tinted magenta.

Initially, keep the camera still and note that the displayed Euler angles and Cartesian coordinates stay close to 0. Then, slowly rotate or move the camera to see how the variables respond. Try to get a sense of direction in terms of one of the 6DOF at a time. For the following screenshot, I have tilted the camera upward (rotating it around its lengthwise axis, that is, the axis of the stereo baseline), and I have found that the roll angle decreases, becoming negative:

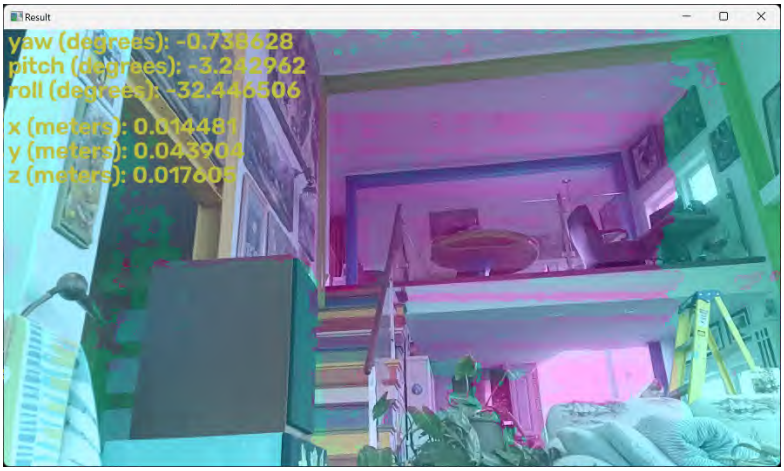


Figure 10.1b: Another screenshot of our visual odometry application

Relative to the previous screenshot, now the camera is angled upward and we see a change in the cumulative odometry result, including a roll of -32 degrees. We also see that the frame now contains a larger background region, including a loft area, which is tinted magenta.

Now, slowly rotate or move the camera back toward its initial pose. You should see the displayed variables move back toward 0. However, you will surely see that the cumulative estimate somewhat undershoots or overshoots the mark, never quite hitting 0 when the camera arrives back at its starting pose. The farther we roam, the harder it is to correctly zero in on the original pose. What can we do about this?

Extending the odometry application

As discussed earlier in the *Understanding visual odometry* section, if an odometry solution relies purely on frame-to-frame visual comparisons, it is a form of dead reckoning. Over time, the cumulative result will suffer from drift or, in other words, accumulation of errors. To improve upon our odometry application, you could try some combination of the following techniques:

- **Apply a Kalman filter**: By using a Kalman filter, smooth out the errors in the cumulative odometry result. For a detailed example of how to apply a Kalman filter to a 3D pose, refer back to the sample application in *Chapter 9*, *Camera Models and Augmented Reality*.

- **Keep a history of multiple frames**: Keep a history of several previous frames instead of just one. Then, every time you capture a new frame, try to find an odometry solution for each possible pair of a previous frame and the current frame. Discard the solutions that are outliers and take an average of the remaining solutions. This approach will work best if the frame rate is high or, in other words, the time between frames is small.
- **Recognize visual landmarks**: Set up an environment that contains several recognizable visual landmarks, such as posters on the wall. Treat one of these landmarks as the origin of your coordinate system. For every other landmark, measure the exact pose relative to the origin. Train your application to recognize and track these various landmarks so that it can correct its estimate of the camera's position and rotation whenever a landmark is in view. For a detailed example of an image-tracking technique, refer back to the sample application in *Chapter 9*, *Camera Models and Augmented Reality*.
- **Combine visual odometry with other inputs**: Use GPS and an altimeter (or barometer) to correct the estimate of position. Use a compass and gyroscope to correct the estimate of rotation. Use an accelerometer, in combination with a Kalman filter, to correct the estimate of acceleration.

Odometry is an impressive capability in its own right, but it becomes even more powerful when combined with the ability to create and process 3D scans of an environment. We will devote the remainder of this chapter to an introduction to 3D point clouds, which are a common way to represent a 3D scan.

Understanding point clouds

Until now, we have dealt with various types of image data (including depth data) in the format of pixels in a dense 2D grid. These 2D image formats are convenient when we need to represent a scene from a given camera perspective. On the other hand, for the purposes of 3D mapmaking or 3D scanning, we would use input from multiple camera perspectives and it would be convenient to remap all those pixel coordinates to a common Cartesian coordinate system, with the help of odometry results. We would replace dense 2D grids of pixels with a sparse 3D cloud of points – in short, a **point cloud** – and each point in this cloud would correspond to one or more of the pixels in the original set of 2D images. This book does not delve into the details of creating a point cloud, but we will load a point cloud from a file and process its data.

> **Note**
>
> Within the context of point clouds, the OpenCV documentation uses the terms **point** and **vertex** interchangeably. We will do the same.

Like a 2D image, a point cloud is a versatile format that may contain various types of data. At minimum, each point has a position, expressed in XYZ coordinates. Also, it may have a color. Another optional type of data is the point's **normal vector**. The normal is a 3D vector that points outward from the surface where the vertex lies. When we say *outward*, we mean perpendicular to the surface and toward a viewpoint whence the vertex is visible. The normal vector is a useful type of data because it helps us distinguish between the outside and inside of a closed 3D shape. That said, for our sample scripts, we will simply work with points' positional coordinates, not colors or normals.

Note

The `depthai` repository contains a set of samples relating to point clouds with color data. See `https://github.com/luxonis/oak-examples/tree/master/gen2-pointcloud`. These samples generate a point cloud in real time using only the latest depth frame and the latest color frame. Starting from this frame-by-frame approach, you could make use of odometry to convert a series of point clouds into a common coordinate system and thus build up a larger point cloud over time. When merging new and old point clouds, you could deem that very similar points (close to each other in position and color) are actually matches, and fuse them into one point.

Point clouds can be saved in various standard file formats, including the **PLY** format (also called the **Polygon File Format** or the **Stanford File Format**) and the **Wavefront OBJ** format. Besides storing point clouds, these two formats can also store complete 3D scans or 3D models, including data about polygons, colors, and textures that characterize the surfaces of 3D objects. Here is an excerpt from a `PLY` file:

```
ply
format ascii 1.0
element vertex 4440
property float32 x
property float32 y
property float32 z
element face 8828
property list uint8 int32 vertex_indices
end_header
0 8.40703 0
0.053054 8.38004 -0.198
0 8.38004 -0.204985
0.102492 8.38004 -0.177522
```

```
0.144946 8.38004 -0.144946
[...]
3 4130 4152 4153
3 4133 4132 4154
3 4439 4007 4436
3 4007 4439 4438
3 4007 4438 4006
```

This `PLY` file starts with a header that describes its format, along with the number of points (or vertices) and the number of polygons (or faces) that it contains. Then, it continues with a list of points, each described in terms of its XYZ coordinates. Finally, it defines a set of polygons in terms of the number of vertices and the vertex indices for each polygon. The whole file is 13,278 lines long and you can find it in this book's GitHub repository, at the subpath `3d_models/jburkardt/street_lamp.ply`. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Of course, if you looked at only the raw data and not the filename, you would be hard-pressed to guess that this is a 3D model of a street lamp. Much like an image file, a point cloud file (or a 3D model file) becomes readily intelligible to human beings only when we have an application to visualize it. Let's write such an application now.

Loading and showing point clouds

Fortunately for us, OpenCV provides functionality for loading point clouds, and Matplotlib provides functionality for plotting any kind of 3D data, including point clouds. Let's create a new script called `show_points_with_matplotlib.py` and import these two familiar libraries:

```
import cv2
import matplotlib.pyplot as plt
```

Now, let's define a path to a `PLY` file – in this case, a model of a street lamp, which we mentioned in the previous section – and let's load its data using OpenCV's `cv2.loadPointCloud` function, as follows:

```
model_path = '../3d_models/jburkardt/street_lamp.ply'

vertices, normals, rgb = cv2.loadPointCloud(model_path)
```

Note

`cv2.loadPointCloud` supports `PLY` and `OBJ` files. The vertex coordinate data, vertex normal data, and vertex color data are loaded. The given file may contain additional data types such as polygon data, but OpenCV ignores these.

For this particular `PLY` file, `normals` is `None` (because the file does not contain any data on vertex normals) and `vertices` is a NumPy array of shape `(4440, 1, 3)`. The second dimension of length `1` is a bit inconvenient, so we squeeze it out to get a shape of `(4440, 3)` instead:

```
vertices = vertices.squeeze(axis=1)
```

Now, for the purpose of plotting the points with Matplotlib, we want to slice the X, Y, and Z coordinates into separate arrays. We also want to find the minimum and maximum values in each dimension so that we can crop the coordinate axes appropriately. The following code obtains the slices, the mins, and the maxes:

```
xs = vertices[:,0]
ys = vertices[:,2]
zs = vertices[:,1]

min_x, max_x = xs.min(), xs.max()
min_y, max_y = ys.min(), ys.max()
min_z, max_z = zs.min(), zs.max()
```

The remainder of our script will deal with Matplotlib's API for 3D scatterplots. First, we can define the whole figure and its size as follows:

```
fig = plt.figure(figsize=(6, 6))
```

Note

The `figsize` argument is used to specify the figure's width and height, nominally in terms of inches. Depending on the user's system, the plot's real size on-screen may or may not match the specified number of inches. By default, `matplotlib` assumes that the screen has a density of 100 **Pixels per Inch** (**ppi**), so `figsize=(6, 6)` implies a size of 600x600 pixels.

Our figure will contain one plot, which will be a 3D projection. We define this plot and get its axes as follows:

```
ax = fig.add_subplot(111, projection='3d')
```

Note

Here, the argument `111` effectively means that the figure contains one plot. Potentially, a figure may contain multiple plots in a grid. The three-digit argument expresses the shape of this grid and the position of the plot we are currently adding. The first digit is the number of rows; the second digit is the number of columns; and the third digit is the plot's index within the grid, starting at `1` in the upper left and increasing to the right.

Now, using the minimum and maximum coordinate values, we define the range of each axis, and we set the aspect ratio so that all axes have the same scale. Here is the relevant code:

```
ax.set_xlim3d(min_x, max_x)
ax.set_ylim3d(min_y, max_y)
ax.set_zlim3d(min_z, max_z)

ax.set_box_aspect([max_x - min_x,
                   max_y - min_y,
                   max_z - min_z])
```

Finally, we plot the XYZ data and show the result on-screen:

```
ax.scatter(xs, ys, zs)

plt.show()
```

Note

Alternatively, it is possible to show point clouds using OpenCV's `viz3d` module. However, this module is only compatible with Windows and Linux, and it requires that OpenCV is built using the optional `WITH_OPENGL` CMake flag. For details about building OpenCV, refer back to *Chapter 1*, *Setting Up OpenCV*. For an example of how to show a point cloud using `viz3d`, see the official tutorial at `https://docs.opencv.org/doc/doxygen/html/d0/d27/tutorial_point_cloud.html`.

Run the script, and you should see a 3D scatterplot of the lamppost's vertices. Click and drag the plot to view it from different angles. The following screenshots show the results from two different perspectives:

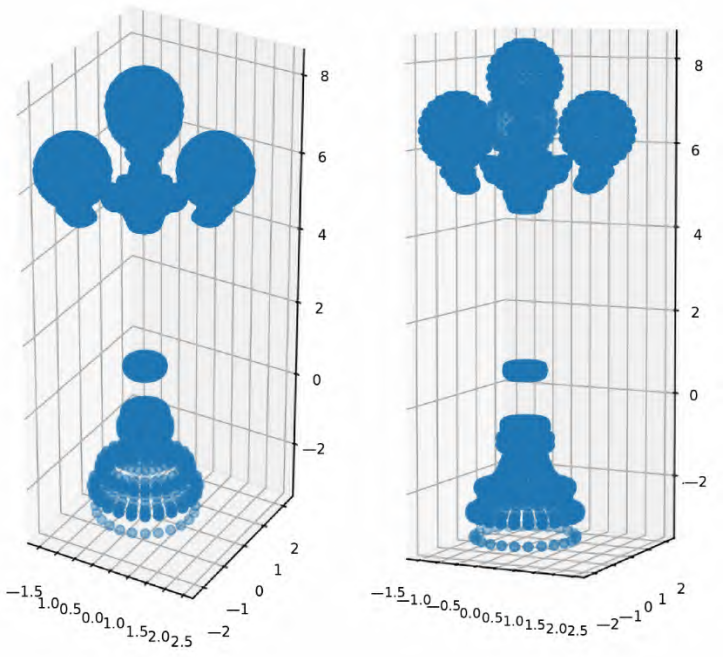


Figure 10.2: The vertices of a lamppost, loaded as a point cloud using OpenCV and visualized with Matplotlib

As human beings, we can tell from this visualization that the point cloud represents several pieces of a lamppost, including four spherical lights mounted at the top. However, as practitioners of computer vision, it behooves us to automate this type of visual inference. We should ask ourselves: How can a computer distinguish between these pieces of the lamppost? Next, let's consider this question of segmentation.

Segmenting point clouds into multiple objects

We looked at segmentation problems previously in *Chapter 4*, *Depth Estimation and Segmentation*. There, we segmented 2D images using algorithms that ultimately rely on the idea of locating transitions in depth values or color values; these transitions are presumed to be boundaries between objects (or parts of an object). Segmenting 3D point clouds is a comparatively simple problem. We may presume that objects (or parts of an object) are separated by empty 3D space, containing no points or only a few noisy points. Thus, objects appear as dense clusters of points within the cloud.

Note

Here, we will examine the simplest case where the point cloud data consists of XYZ vertex coordinates only. If the cloud also has other data types, such as vertex colors or vertex normals, these may be useful inputs for more complex segmentation algorithms, perhaps providing a way to refine the results after an initial stage of clustering based on XYZ coordinates.

Clustering is potentially a useful operation for any kind of vector data, not just spatial coordinates. Indeed, in *Chapter 7*, *Building Custom Object Detectors*, we applied k-means clustering to SIFT descriptors in order to produce a **Visual Bag of Words** (**VBoW**). While k-means is popular as a general-purpose clustering algorithm, it is not a great choice for spatial data because its centroid model assumes that each cluster is convex. On the contrary, many 3D objects are concave. Think of the many concavities in an ornate lamppost or in a wine glass with a fluted stem.

For spatial data, two of the better clustering algorithms are **BIRCH** and **DBSCAN**. Both of these were developed in 1996, and they were the first clustering algorithms designed to deal with noisy data (a common problem in point clouds). Conceptually, they can be compared as follows:

- BIRCH uses a hierarchical connectivity model, whereby geometric descriptors are computed for various nodes (small neighborhoods of points), and adjacent nodes are connected to form clusters, iteratively, as long as the connections satisfy certain metrics in terms of the nodes' descriptors. At the end of the clustering process, all clusters must be composed of well-connected nodes. There may be some remaining disconnected nodes, which are deemed outliers or noise. For full details of the algorithm, see the original paper, *BIRCH: An Efficient Data Clustering Method for Very Large Databases*, by Zhang, Ramakrishnan, and Livny. A copy can be downloaded from `https://www2.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf`.

DBSCAN uses a density model, whereby each cluster seeks to expand outward from its core points, iteratively absorbing more and more fringe points, as long as this expansion continues to satisfy certain metrics of density. At the end of the clustering process, all clusters must be dense except at their fringes. Beyond the fringes, there may be some remaining points that do not belong to any cluster; these are deemed outliers or noise. For a complete description of the algorithm, see the original paper, *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*, by Ester, Kriegel, Sander, and Xu. A copy is available online at `https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf`.

A Python library called scikit-learn provides convenient implementations of BIRCH, DBSCAN, and many other clustering algorithms. Overall, scikit-learn is designed as an accessible toolkit for machine learning and data science, and it is interoperable with NumPy, SciPy, and Matplotlib. The clustering algorithms reside in the `sklearn.cluster` module of scikit-learn, and a wealth of examples and explanations can be found in this module's documentation at

Let's proceed to use this module in our samples to perform the segmentation of an OpenCV point cloud.

Using BIRCH for segmentation

As a starting point for our BIRCH segmentation demo, let's make a copy of our `show_points_with_matplotlib.py` script. Name the copy `find_birch_clusters_with_sklearn.py`.

We will use a class called `Birch` from the `sklearn.cluster` module of scikit-learn. As you might expect, `Birch` implements the BIRCH algorithm and, like everything in scikit-learn, it uses NumPy arrays for its inputs and outputs. Let's import `Birch` by adding the `highlighted` code to the following block:

```
import cv2
import matplotlib.pyplot as plt

from sklearn.cluster import Birch
```

We will proceed to load the point cloud, with no changes to the following block of code:

```
model_path = '../3d_models/jburkardt/street_lamp.ply'

vertices, normals, rgb = cv2.loadPointCloud(model_path)
vertices = vertices.squeeze(axis=1)
```

Now, we will add some code to perform BIRCH clustering on the 3D vertex data. Specifically, we will create an instance of the `Birch` class, we will pass the `vertices` to its `fit` method, and we will get the resulting `labels` from its `predict` method.

```
clusterer = Birch(threshold=0.3, n_clusters=6)
clusterer.fit(vertices)
labels = clusterer.predict(vertices)
```

Here, `labels` is a NumPy `int64` array of shape `(4440,)`, since we have 4,440 vertices. The order of the `labels` array corresponds to the order of the `vertices` array. Each value in `labels` represents the cluster to which the given vertex belongs. For example, if `labels` contains the values `[ 4, 4, 4, ... ]`, this means that the first three vertices belong to cluster 4.

We will proceed to slice the coordinate data, find the minimum and maximum in each dimension, and set up the bounds of the plot and its axes, all without any changes to the following code:

```
xs = vertices[:,0]
ys = vertices[:,2]
zs = vertices[:,1]

min_x, max_x = xs.min(), xs.max()
min_y, max_y = ys.min(), ys.max()
min_z, max_z = zs.min(), zs.max()

fig = plt.figure(figsize=(6, 6))

ax = fig.add_subplot(111, projection='3d')

ax.set_xlim3d(min_x, max_x)
ax.set_ylim3d(min_y, max_y)
ax.set_zlim3d(min_z, max_z)

ax.set_box_aspect([max_x - min_x,
                   max_y - min_y,
                   max_z - min_z])
```

Finally, when we plot the 3D data, we will provide an additional parameter, `c=labels`, to indicate that the points in the scatterplot should be color-coded according to the values in `labels`. Here is the relevant code:

```
ax.scatter(xs, ys, zs, c=labels)

plt.show()
```

When you run the script, you should get a multicolored 3D scatterplot, where each color represents a different cluster. Click and drag the plot to view it from different angles. You should see results similar to the following screenshots:

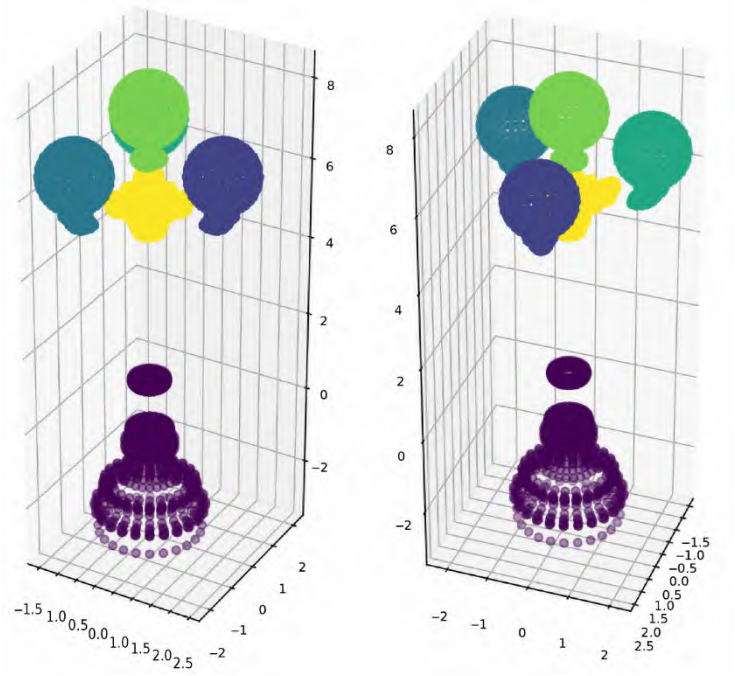


Figure 10.3: The results of running BIRCH segmentation on the vertices of a lamppost, as visualized with Matplotlib

Here, BIRCH has produced a segmentation that makes sense. There is a different segment for each of the four lights (including an orb and the region around its socket). The region where the four branches join is another distinct segment. Finally, the lower base is a segment. If you asked a human being to describe the lamppost in terms of six segments, you would probably get a similar answer.

Next, let's see how the code and the segmentation result will change when we replace BIRCH with another algorithm, DBSCAN.

Using DBSCAN for segmentation

For our DBSCAN sample, we will copy and modify `find_birch_clusters_with_sklearn.py`. Let's call the copy `find_dbscan_clusters_with_sklearn.py`.

Instead of importing scikit-learn's `Birch` class, we will import its `DBSCAN` class, as `highlighted` in the following code:

```
import cv2
import matplotlib.pyplot as plt
```

```
from sklearn.cluster import DBSCAN
```

We will proceed to load the point cloud as in our previous scripts:

```
model_path = '../3d_models/jburkardt/street_lamp.ply'

vertices, normals, rgb = cv2.loadPointCloud(model_path)
vertices = vertices.squeeze(axis=1)
```

Next, we will create the `clusterer`, feed the `vertices` to it, and get the resulting `labels`. Compared to the `Birch` class, the `DBSCAN` class has a slightly different API for these steps but, in the end, the format of the `labels` is the same. The differences are `highlighted` in the following code:

```
clusterer = DBSCAN(eps=0.3)
clusterer.fit(vertices)
labels = clusterer.labels_
```

Note

When we initialize a `DBSCAN` object, the `eps` argument allows us to specify the maximum distance between inliers in a cluster. This is the most important parameter of the DBSCAN algorithm, and it needs to be tailored based on the scale and density of the point cloud.

The remainder of `find_dbscan_clusters_with_sklearn.py` is identical to `find_birch_clusters_with_sklearn.py`. Indeed, the two scripts produce the same kind of plot but with somewhat different segmentation results. When you run the DBSCAN script and drag the plot to view it from different angles, you should see an output similar to the following screenshots:

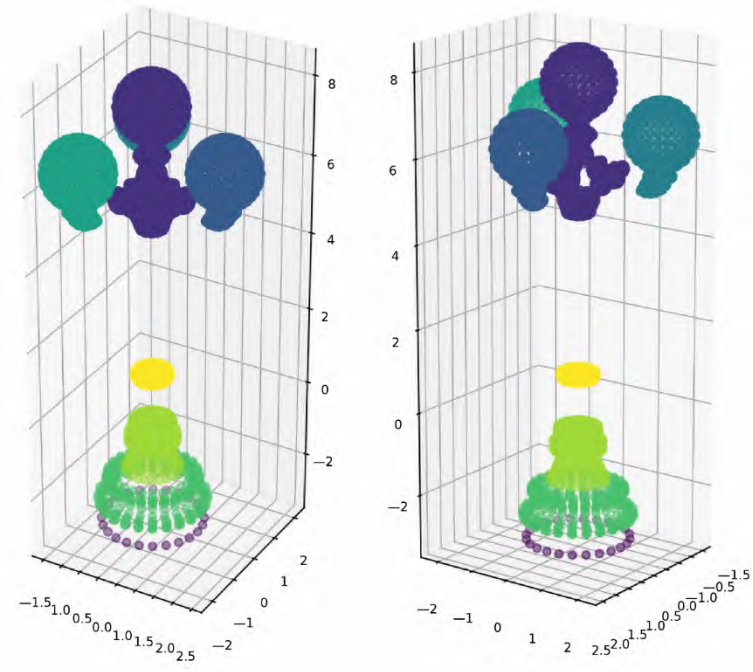


Figure 10.4: The results of running DBSCAN segmentation on the vertices of a lamppost, as visualized with Matplotlib

Here, we see that DBSCAN has produced eight clusters in total. The lower base is divided into four clusters. Another cluster consists of the central light and the region where the branches join. Then, each of the other three lights (an orb along with the region around its socket) is also a distinct cluster. Overall, this segmentation result seems a bit less intuitive than the one produced by BIRCH, but in both cases, the algorithm distinguishes the four lights from each other.

You may want to study the official documentation for `sklearn.cluster` at `https://scikit-learn.org/stable/modules/clustering.html` and try to adapt our clustering scripts to use some other algorithms. Also, you may want to adjust the parameters of the BIRCH clusterer, the DBSCAN clusterer, and any others you try. How do the results differ? For additional tests, you can find more `PLY` files in the `3d_models/jburkardt` subfolder of this book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

When you are finished with your experiments, let's review what we have accomplished in this chapter.

Summary

We built a visual odometry application, which analyzes frames from an OAK-D camera in order to track changes in the camera's position and rotation over time. As part of this work, we learned how to use the `depthai` Python module to capture depth frames, which are aligned with RGB frames. We also learned about several classes in OpenCV's 3D module: `Odometry`, `OdometrySettings`, and `OdometryFrame`. We explored ways to combine visual odometry with other techniques to achieve more robust estimates of the camera's pose relative to its starting point and other landmarks.

We also learned how to load point clouds using OpenCV and visualize them using Matplotlib. Finally, we used scikit-learn to segment a point cloud into several clusters, which may be separate objects or appendages of one object. With this new knowledge of odometry, point clouds, and 3D segmentation, you have a strong foundation for future work in 3D navigation.

Note

Open3D is another modern library that you may encounter in your future work in 3D computer vision. Depth data, visual odometry, and point clouds are all within Open3D's scope, and it offers both C++ and Python interfaces. You can read more about it at `http://www.open3d.org/`.

Next, in *Chapter 11*, *Introduction to Neural Networks with OpenCV*, we will return to problems of 2D image classification, viewed from a fresh perspective. Particularly, we will learn how to use sophisticated neural network classifiers via convenient interfaces in OpenCV and in Google's MediaPipe library.

11

Introduction to Neural Networks with OpenCV

This chapter introduces a family of machine learning models **called artificial neural networks (ANNs)**, or sometimes just **neural networks**, and we will utilize this knowledge to perform various tasks such as handwritten digit recognition, object detection and recognition, and finally, gesture recognition from a video input.

A key characteristic of neural networks is that they attempt to learn relationships among variables in a multi-layered fashion; they learn multiple functions to predict intermediate results before combining these into a single function to predict something meaningful (such as the class of an object). Recent versions of OpenCV contain an increasing amount of functionality related to ANNs, and, in particular, ANNs with many layers, called **deep neural networks (DNNs)**. We will experiment with both shallower ANNs and DNNs in this chapter.

We have already gained some exposure to machine learning in other chapters—especially in *Chapter 7, Building Custom Object Detectors*, where we developed a car/non-car classifier using SURF descriptors, a BoW, and an SVM. With this basis for comparison, you might be wondering, what is so special about ANNs? Why are we devoting this book's final chapter to them?

ANNs aim to provide superior accuracy in the following circumstances:

- There are many input variables, which may have complex, nonlinear relationships to each other.
- There are many output variables, which may have complex, nonlinear relationships to the input variables. (Typically, the output variables in a classification problem are the

confidence scores for the classes, so if there are many classes, there are many output variables.)

- There are many hidden (unspecified) variables that may have complex, nonlinear relationships to the input and output variables. DNNs even aim to model *multiple* layers of hidden variables, which are interrelated primarily to each other rather than being related primarily to input or output variables.

These circumstances exist in many—perhaps most—real-world problems. Thus, the promised advantages of ANNs and DNNs are enticing. On the other hand, ANNs and especially DNNs are notoriously opaque models, insofar as they work by predicting the existence of an arbitrary number of nameless, hidden variables that may relate to everything else.

Over the course of this chapter, we will cover the following topics:

- Understanding ANNs as a statistical model and as a tool for supervised machine learning.
- Understanding ANN topology or, in other words, the organization of an ANN into layers of interconnected neurons. Particularly, we will consider the topology that enables an ANN to act as a type of classifier known as a **multi-layer perceptron** (**MLP**).
- Training and using ANNs as classifiers in OpenCV.
- Building an application that detects and recognizes handwritten digits (0 to 9). For this, we will train an ANN based on a widely used dataset called MNIST, which contains samples of handwritten digits.
- Loading and using pre-trained DNNs in OpenCV. We will cover examples of object classification, face detection, and gender classification with DNNs.

By the end of this chapter, you will be in a good position to train and use ANNs in OpenCV, to use pre-trained DNNs from a variety of sources, and to start exploring other libraries that allow you to train your own DNNs.

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Please refer to *Chapter 1, Setting Up OpenCV*, for installation instructions.

The completed code and sample videos for this chapter can be found in the `chapter11` folder of this book's GitHub repository.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Understanding ANNs

Let's define ANNs in terms of their basic role and components. Although much of the literature on ANNs emphasizes the idea that they are *biologically inspired by* the way neurons connect in a brain, we don't need to be biologists or neuroscientists to understand the fundamental concepts of an ANN.

First of all, an ANN is a **statistical model**. What is a statistical model? A statistical model is a pair of elements, namely the space, `S`, (a set of observations) and the probability, `P`, where `P` is a distribution that approximates `S` (in other words, a function that would generate a set of observations that is very similar to `S`).

Here are two different ways to think of `P`:

- `P` is a simplification of a complex scenario.
- `P` is the function that generated `S` in the first place, or at the very least a set of observations very similar to `S`.

Thus, ANNs are models that take a complex reality, simplify it, and deduce a function to (approximately) represent the statistical observations we would expect from that reality, in a mathematical form.

ANNs, like other types of machine learning models, can learn from observations in one of the following ways:

- **Supervised learning**: Under this approach, we want the model's training process to produce a function that maps a known set of input variables to a known set of output variables. We know, *a priori*, the nature of the prediction problem, and we delegate the process of finding a function that solves this problem to the ANN. To train the model, we must provide input samples along with the correct, corresponding outputs. For a classification problem, the output variables may be confidence scores for one or more classes.
- **Unsupervised learning**: Under this approach, the set of output variables is not *known a priori*. The model's training process must yield a set of output variables, as well as a function to map the input variables to these output variables. For a classification problem, unsupervised learning can lead to the discovery of previously unknown classes, such as previously unknown diseases in the context of medical data. Unsupervised learning may use techniques including (but not limited to) clustering,

which we explored in the context of BoW models in *Chapter 7, Building Custom Object Detectors*.

- **Reinforcement learning**: This approach turns the typical prediction problem upside down. Before training the model, we already have a system that yields values for a known set of output variables when we feed it values for a known set of input variables. We know, *a priori*, a way of scoring a sequence of outputs based on their goodness (desirability) or lack thereof. However, we might not know the real function that maps inputs to outputs—or, even if we do know it, it is so complex that we cannot solve it for optimal inputs. Thus, we want the model's training process to produce a function that predicts the next-in-sequence optimal inputs, based on the last outputs. During training, the model learns from the score that eventually arises from its actions (its chosen inputs). Essentially, the model must learn to become a good decision-maker within the context of a particular system of rewards and punishments.

Throughout the remainder of this chapter, we will confine our discussions to supervised learning, as this is the most common approach to machine learning in the context of computer vision.

The next step in our journey toward comprehending ANNs is to understand how an ANN improves on the concept of a simple statistical model, and on other types of machine learning.

What if the function that generated the dataset is likely to take a large number of (unknown) inputs?

The strategy that ANNs adopt is to delegate work to a number of **neurons**, **nodes**, or **units**, each of which is capable of approximating the function that created the inputs. In mathematics, approximation is the process of defining a simpler function whose output is similar to that of a more complex function, at least for some range of inputs.

The difference between the approximate function's output and the original function's output is called the **error**. A defining characteristic of a neural network is that the neurons must be capable of approximating a nonlinear function.

Let's take a closer look at neurons.

Understanding neurons and perceptrons

Often, to solve a classification problem, an ANN is designed as an MLP in which each neuron acts as a kind of binary classifier called a **perceptron**. The perceptron is a concept that dates back to the 1950s. To put it simply, a perceptron is a function that takes a number of inputs and produces a single value. Each of the inputs has an associated weight that signifies its importance in an **activation function**. The activation function should have a nonlinear response; for example, a sigmoid function (sometimes called an *S*-curve) is a common choice. A threshold function, called a **discriminant**, is applied to the activation function's output to convert it into a binary classification of 0 or 1. Here is a visualization of this sequence, with inputs on the left, the activation function in the middle, and the discriminant on the right:

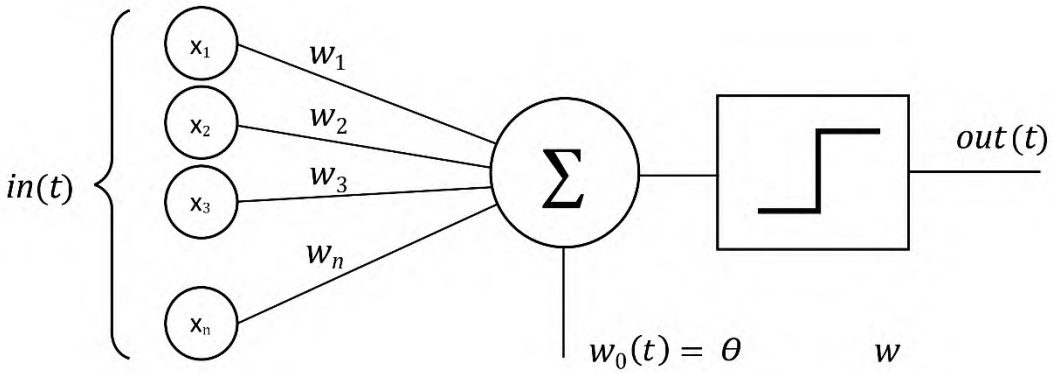


Figure 11.1: A representation of an MLP

What do the input weights represent, and how are they determined?

Neurons are interconnected, insofar as one neuron's output can be an input for many other neurons. Each input weight defines the strength of the connection between two neurons. These weights are adaptive, meaning that they change in time according to a learning algorithm.

Due to the neurons' interconnectedness, the network has layers. Now, let's examine how these layers are typically organized.

Understanding the layers of a neural network

Here is a visual representation of a neural network:

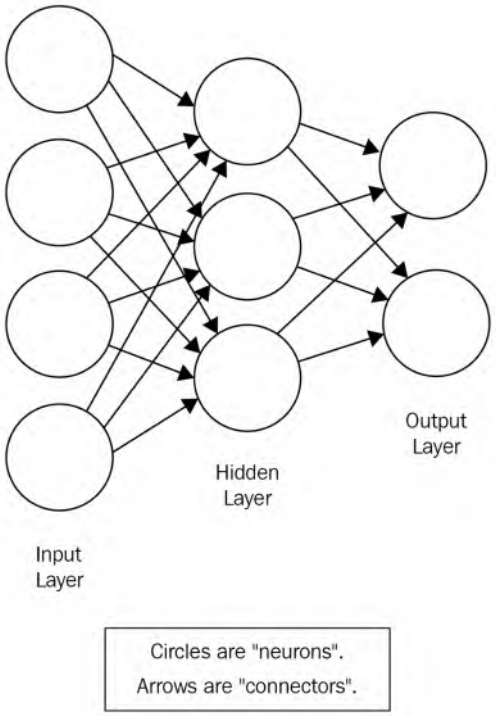


Figure 11.2: Diagram of a neural network

As the preceding diagram shows, there are at least three distinct layers in a neural network: the **input layer**, the **hidden layer**, and the **output layer**. There can be more than one hidden layer; however, one hidden layer is enough to resolve many real-life problems, such as basic image classification or anomaly detection in normally distributed datasets. A neural network with multiple hidden layers is sometimes called a DNN.

Note

If we are using an ANN as a classifier, then each output node's output value is a confidence score for a class. For a given sample (that is, a given set of input values), we want to know which output node produces the highest output value. This highest-scoring output node corresponds to the predicted class.

How do we determine the network's topology, and how many neurons do we need to create for each layer? Let's make this determination layer by layer.

Choosing the size of the input layer

The number of nodes in the input layer is, by definition, the number of inputs in the network. For example, let's say you want to create an ANN to help you determine an animal's species based on measurements of its physical attributes. In principle, we can choose any measurable attributes. If we choose to classify animals based on weight, length, and number of teeth, that is a set of three attributes, and thus our network needs to contain three input nodes.

Are these three input nodes an adequate basis for species classification? Well, in a real-life problem, surely not—but in a toy problem, it depends on the output we are trying to achieve, and this is our next consideration.

Choosing the size of the output layer

For a classifier, the number of nodes in the output layer is, by definition, the number of classes the network can distinguish. Continuing with the preceding example of an animal classification network, we can use an output layer of four nodes if we know we are going to deal with the following animals: dog, condor, dolphin, and dragon (!). If we try to classify data for an animal that is not in one of these categories, the network will predict the class that is most likely to resemble this unrepresented animal.

Now, we come to a difficult problem—the size of the hidden layer.

Choosing the size of the hidden layer

There are no agreed-upon rules of thumb for choosing the size of the hidden layer; it must be chosen based on experimentation. For every real-world problem where you want to apply ANNs, you will need to train, test, and retrain your ANN until you find a number of hidden nodes that yield acceptable accuracy.

Of course, even when choosing a parameter's value by experimentation, you might wish for the experts to suggest a starting value, or a range of values, for your tests. Unfortunately, there is no expert consensus on these points either. Some experts offer rules of thumb based on the following broad suggestions (these should be taken with a grain of salt):

- If the input layer is large, the number of hidden neurons should be between the size of the input layer and the size of the output layer—and, typically, closer to the size of the output layer.

- On the other hand, if the input and output layers are both small, the hidden layer should be the largest layer.
- If the input layer is small but the output layer is large, the hidden layer should be closer to the size of the input layer.

Other experts suggest that the number of training samples also needs to be taken into account; a greater number of training samples implies that a greater number of hidden nodes might be useful.

One critical factor to keep in mind is **overfitting**. Overfitting occurs when there is such an inordinate amount of pseudo-information contained in the hidden layer, compared to the information actually provided by the training data, that the classification is not very meaningful. The larger the hidden layer, the more training data it requires in order for it to learn properly. Of course, as the size of the training dataset grows, so does the training time.

For some of the ANN sample projects in this chapter, we will use a hidden layer size of 60 as a starting point. Given a large training set, 60 hidden nodes can yield decent accuracy for a variety of classification problems.

Now that we have a general idea of what ANNs are, let's see how OpenCV implements them, and how to put them to good use. We will start with a minimal code example. Then, we will flesh out the animal-themed classifier that we discussed in the previous two sections. Finally, we will work our way up to a more realistic application, in which we will classify handwritten digits based on image data.

Training a basic ANN in OpenCV

OpenCV provides a class, `cv2.ml.ANN_MLP`, that implements an ANN as an MLP. This is exactly the kind of model we described earlier, in *the Understanding neurons and perceptrons* section.

To create an instance of `cv2.ml.ANN_MLP`, and to format data for this ANN's training and use, we rely on functionality in OpenCV's machine learning module, `cv2.ml`. As you may recall, this is the same module that we used for SVM-related functionality in *Chapter 7, Building Custom Object Detectors*. Moreover, `cv2.ml.ANN_MLP` and `cv2.ml.SVM` share a common base class called `cv2.ml.StatModel`. Therefore, you will find that OpenCV provides similar APIs for ANNs and SVMs.

Let's examine a dummy example as a gentle introduction to ANNs, which you can find in the book's code repository. To access the repository link, follow the steps in the *Download the*

example code files section in the *Preface*. This example will use completely meaningless data, but it will show us the basic API for training and using an ANN in OpenCV:

1. To begin, we import OpenCV and NumPy as usual:

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

2. Now, we create an untrained ANN:

```
ann = cv2.ml.ANN_MLP_create()
```

3. After creating the ANN, we need to configure its number of layers and nodes:

```
ann.setLayerSizes(np.array([9, 15, 9], np.uint8))
```

   The layer sizes are defined by the NumPy array that we pass to the `setLayerSizes` method. The first element is the size of the input layer, the last element is the size of the output layer, and all the in-between elements define the sizes of the hidden layers. For example, `[9, 15, 9]` specifies 9 input nodes, 9 output nodes, and a single hidden layer with 15 nodes. If we changed this to `[9, 15, 13, 9]`, it would specify two hidden layers with 15 and 13 nodes, respectively.

   We can also configure the activation function, the training method, and the training termination criteria, as follows:

```
ann.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 0.6, 1.0)
ann.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP, 0.1, 0.1)
ann.setTermCriteria(
    (cv2.TERM_CRITERIA_MAX_ITER | cv2.TERM_CRITERIA_EPS, 100, 1.0))
```

   Here, we are using a symmetrical sigmoid activation function (`cv2.ml.ANN_MLP_SIGMOID_SYM`) and a backpropagation training method (`cv2.ml.ANN_MLP_BACKPROP`). Backpropagation is an algorithm that calculates errors of predictions at the output layer, traces the sources of the errors backward through previous layers, and updates the weights in order to reduce errors.

4. Let's train the ANN. We need to specify training inputs (or `samples`, in OpenCV's terminology), the corresponding correct outputs (or `responses`), and whether the

data's format (or `layout`) is one row per sample or one column per sample. Here is an example of how we train the model with a single sample:

```
training_samples = np.array(
    [[1.2, 1.3, 1.9, 2.2, 2.3, 2.9, 3.0, 3.2, 3.3]], np.float32)
layout = cv2.ml.ROW_SAMPLE
training_responses = np.array(
    [[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]], np.float32)
data = cv2.ml.TrainData_create(
    training_samples, layout, training_responses)
ann.train(data)
```

Note

Realistically, we would want to train any ANN with a larger dataset that contains far more than one sample. We could do this by extending `training_samples` and `training_responses` so that they contain multiple rows, representing multiple samples and their corresponding responses. Alternatively, we could call the ANN's `train` method multiple times, with new data each time. The latter approach requires some additional arguments for the `train` method, and it is demonstrated in the next section, *Training an ANN classifier in multiple epochs*.

Note that in this case, we are training the ANN as a classifier. Each response is a confidence score for a class, and in this case, there are nine classes. We will refer to them by their 0-based indices, as classes 0 to 8. Our training sample in this case has a response of `[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]`, meaning that it is an instance of class 5 (with confidence 1.0), and it is definitely not an instance of any other class (as the confidence is 0.0 for every other class).

5. To complete our minimal tour of the ANN's API, let's make another sample, classify it, and print the result:

```
test_samples = np.array(
    [[1.4, 1.5, 1.2, 2.0, 2.5, 2.8, 3.0, 3.1, 3.8]],
    np.float32
)
prediction = ann.predict(test_samples)
print(prediction)
```

This will print the following result:

```
(5.0, array([[-0.08763029, -0.01616517, 0.13196233, 0.0402631 , 0.05711843,
    1.1642447 , 0.18130444, 0.1857026 , -0.07486832]],
    dtype=float32))
```

This means that the provided input was classified as belonging to class 5. Again, this is only a dummy example, and the classification is pretty meaningless; however, the network behaved correctly. In the preceding code, we only provided one training record, which was a sample of class 5, so the network classified a new input as belonging to class 5. (As far as our woefully limited training dataset suggests, other classes besides 5 might never occur.)

As you may have guessed, the output of a prediction is a tuple, with the first value being the class and the second being an array containing the probabilities for each class. The predicted class will have the highest value.

Let's move on to a slightly more believable example—animal classification.

Training an ANN classifier in multiple epochs

Let's create an ANN that attempts to classify animals based on three measurements: weight, length, and number of teeth. This is, of course, a mock scenario. Realistically, no one would describe an animal with just these three statistics. However, our intent is to improve our understanding of ANNs before we start applying them to image data.

Compared to the minimal example in the previous section, our animal classification mock-up will be more sophisticated in the following ways:

- We will increase the number of neurons in the hidden layer.
- We will use a larger training dataset. For convenience, we will generate this dataset pseudorandomly.
- We will train the ANN in multiple epochs, meaning that we will train and retrain it multiple times with the same dataset each time.

Note

The number of neurons in the hidden layer is an important parameter that needs to be tested in order to optimize the accuracy of any ANN. You will find that a larger hidden layer may improve accuracy up to a point, and then it will overfit, unless you start compensating with an enormous training dataset. Likewise, up to a point, a greater number of epochs may improve accuracy, but too many will result in overfitting.

Let's go through the implementation step by step (you can find the code in this book's GitHub repository `ch11` folder. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*):

1. First, we import OpenCV and NumPy as usual. Then, from the Python standard library, we import the `randint` function to generate pseudorandom integers and the `uniform` function to generate pseudorandom floating-point numbers:

```
import cv2
import numpy as np
from random import randint, uniform
```

2. Next, we create and configure the ANN. This time, we use a three-neuron input layer, a 50-neuron hidden layer, and a four-neuron output layer, as `highlighted` in the following code:

```
animals_net = cv2.ml.ANN_MLP_create()
animals_net.setLayerSizes(np.array([3, 50, 4]))
animals_net.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 0.6, 1.0)
animals_net.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP, 0.1, 0.1)
animals_net.setTermCriteria(
    (cv2.TERM_CRITERIA_MAX_ITER | cv2.TERM_CRITERIA_EPS, 100, 1.0))
```

3. Now, we need some data. We aren't really interested in representing animals accurately; we just require a bunch of records to be used as training data. Thus, we define four functions in order to generate random samples of different classes, along with another four functions to generate the correct classification results for training purposes:

```
"""Input arrays
weight, length, teeth
"""

"""Output arrays
dog, condor, dolphin, dragon
"""

def dog_sample():
    return [uniform(10.0, 20.0), uniform(1.0, 1.5),
        randint(38, 42)]
```

```
def dog_class():
    return [1, 0, 0, 0]

def condor_sample():
    return [uniform(3.0, 10.0), randint(3.0, 5.0), 0]

def condor_class():
     return [0, 1, 0, 0]

def dolphin_sample():
    return [uniform(30.0, 190.0), uniform(5.0, 15.0),
        randint(80, 100)]

def dolphin_class():
    return [0, 0, 1, 0]

def dragon_sample():
    return [uniform(1200.0, 1800.0), uniform(30.0, 40.0),
        randint(160, 180)]

def dragon_class():
    return [0, 0, 0, 1]
```

4. We also define the following helper function in order to convert a sample and classification into a pair of NumPy arrays:

```
def normalize_sample(sample):
    return (np.array([sample], np.float32) /
            np.array([1800.0, 40.0, 180.0], np.float32))

def record(sample, classification):
    return (normalize_sample(sample),
            np.array([classification], np.float32))
```

5. Let's proceed with the creation of our fake animal data. We will create 20,000 samples per class:

```
RECORDS = 20000
records = []
for x in range(0, RECORDS):
```

```
    records.append(record(dog_sample(), dog_class()))
    records.append(record(condor_sample(), condor_class()))
    records.append(record(dolphin_sample(), dolphin_class()))
    records.append(record(dragon_sample(), dragon_class()))
```

6. Now, let's train the ANN. As we discussed at the start of this section, we will use multiple training epochs. Each epoch is an iteration of a loop, as shown in the following code:

```
EPOCHS = 10
for e in range(0, EPOCHS):
    print("epoch: %d" % e)
    for t, c in records:
        data = cv2.ml.TrainData_create(t, cv2.ml.ROW_SAMPLE, c)
        if animals_net.isTrained():
            animals_net.train(
                data,
                cv2.ml.ANN_MLP_UPDATE_WEIGHTS |
                cv2.ml.ANN_MLP_NO_INPUT_SCALE |
                cv2.ml.ANN_MLP_NO_OUTPUT_SCALE
            )
        else:
            animals_net.train(data, cv2.ml.ANN_MLP_NO_INPUT_SCALE |
cv2.ml.ANN_MLP_NO_OUTPUT_SCALE)
```

Note

For real-world problems with large and diverse training datasets, an ANN can potentially benefit from hundreds of training epochs. For the best results, you may wish to keep training and testing an ANN until you reach convergence, which means that further epochs no longer produce a noticeable improvement in the accuracy of the results. However, libraries such as **scikit-learn** provide utilities for "early stopping" and "checkpoints" that allow training to stop if the accuracy of the model drops with the passing of epochs and create periodic checkpoints that can be reused for further training.

Note that we must pass the `cv2.ml.ANN_MLP_UPDATE_WEIGHTS` flag to the ANN's `train` function to update the previously trained model, rather than training a

new model from scratch. This is a critical point to remember whenever you are training a model incrementally, as we are doing here.

7. Having trained our ANN, we should test it. For each class, let's generate 100 new random samples, classify them using the ANN, and keep track of the number of correct classifications:

```
TESTS = 100

dog_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict(
        normalize_sample(dog_sample()))[0])
    print("class: %d" % clas)
    if clas == 0:
        dog_results += 1

condor_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict(
        normalize_sample(condor_sample()))[0])
    print("class: %d" % clas)
    if clas == 1:
        condor_results += 1

dolphin_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict(
        normalize_sample(dolphin_sample()))[0])
    print("class: %d" % clas)
    if clas == 2:
        dolphin_results += 1

dragon_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict(
        normalize_sample(dragon_sample()))[0])
    print("class: %d" % clas)
```

```
    if clas == 3:
        dragon_results += 1
```

8. Finally, let's print the accuracy statistics:

```
print("dog accuracy: %.2f%%" % (100.0 * dog_results / TESTS))
print("condor accuracy: %.2f%%" % (100.0 * condor_results / TESTS))
print("dolphin accuracy: %.2f%%" % \
    (100.0 * dolphin_results / TESTS))
print("dragon accuracy: %.2f%%" % (100.0 * dragon_results / TESTS))
```

When we run the script, the preceding code block should produce the following output:

```
dog accuracy: 100.00%
condor accuracy: 100.00%
dolphin accuracy: 100.00%
dragon accuracy: 100.00%
```

Since we are dealing with random data, the results may vary each time you run the script. Typically, the accuracy should be high or even perfect because we have set up a simple classification problem with non-overlapping ranges of input data. (The range of random weight values for a dog does not overlap with the range for a dragon, and so forth.)

You may wish to take some time to experiment with the following modifications (one at a time) so that you can see how the ANN's accuracy is affected:

- Change the number of training samples by modifying the value of the `RECORDS` variable.
- Change the number of training epochs by modifying the value of the `EPOCHS` variable.
- Make the ranges of input data partially overlap by editing the parameters of the `uniform` and `randint` function calls in our `dog_sample`, `condor_sample`, `dolphin_sample`, and `dragon_sample` functions.

When you are ready, we will proceed with an example containing real-life image data. With this, we will train an ANN to recognize handwritten digits.

Recognizing handwritten digits with an ANN

A handwritten digit is any of the 10 Arabic numerals (0 to 9), written manually with a pen or pencil, as opposed to being printed by a machine. The appearance of handwritten digits can vary significantly. Different people have different handwriting, and—with the possible

exception of a skilled calligrapher—a person does not produce identical digits every time he or she writes. This variability means that the visual recognition of handwritten digits is a non-trivial problem for machine learning. Indeed, students and researchers in machine learning often test their skills and new algorithms by attempting to train an accurate recognizer for handwritten digits. We will approach this challenge in the following manner:

1. Load data from a Python-friendly version of the MNIST database. This is a widely used database containing images of handwritten digits.
2. Using the MNIST data, train an ANN in multiple epochs.
3. Load an image of a sheet of paper with many handwritten digits on it.
4. Based on contour analysis, detect the individual digits on the paper.
5. Use our ANN to classify the detected digits.
6. Review the results in order to determine the accuracy of our detector and our ANN-based classifier.

Before we delve into the implementation, let's review some information about the MNIST database.

Understanding the MNIST database of handwritten digits

The **MNIST** database (or **Modified National Institute of Standards and Technology** database) is publicly available at `https://github.com/sunsided/mnist`. The database includes a training set of 60,000 images of handwritten digits. Half of these were written by employees of the United States Census Bureau, while the other half were written by high school students in the United States.

The database also includes a test set of 10,000 images, gathered from the same writers. All the training and test images are in grayscale format, with dimensions of 28 x 28 pixels. The digits are white (or shades of gray) on a black background. For example, here are three of the MNIST training samples:

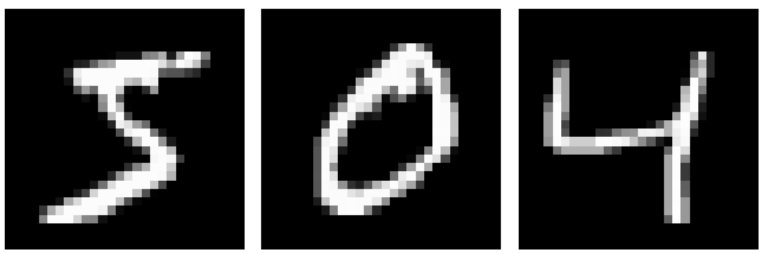


Figure 11.3: Sample of MNIST images

Note

As an alternative to using MNIST, you could, of course, build a similar database yourself. This would involve collecting a large number of images of handwritten digits, converting the images into grayscale, cropping them so that each image contains a single digit in a standardized position, and scaling the images so that they are all the same size. You would also need to label the images so that a program could read the correct classification, for the purpose of training and testing a classifier.

Many authors provide examples of how to use the MNIST database with various machine learning libraries and algorithms—not just OpenCV and not just ANNs. Michael Nielsen, the author of the free online book *Neural Networks and Deep Learning*, devotes a chapter to MNIST and ANNs here: `http://neuralnetworksanddeeplearning.com/chap1.html`. He shows how to implement an ANN almost from scratch, using only NumPy, and this is excellent reading if you want to deepen your understanding beyond the kind of high-level functionality that OpenCV exposes. His code is freely available on GitHub at `https://github.com/mnielsen/neural-networks-and-deep-learning`.

Nielsen provides a version of MNIST as a `PKL.GZ` (gzip-compressed Pickle) file, which can be easily loaded into Python. For the purposes of our book's OpenCV sample, we (the authors) have taken Nielsen's `PKL.GZ` version of MNIST, reorganized it for our purposes, and placed it inside this book's GitHub repository at `chapter11/digits_data/mnist.pkl.gz`. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Now that we know something about the MNIST database, let's consider what ANN parameters are appropriate for this training set.

Choosing training parameters for the MNIST database

Each MNIST sample is an image containing 784 pixels (that is, 28 x 28 pixels). Thus, our ANN's input layer will have 784 nodes. The output layer will have 10 nodes because there are 10 classes of digits (0 to 9).

We are free to choose the values of other parameters, such as the number of nodes in the hidden layer, the number of training samples to use, and the number of training epochs. As usual, experimentation can help us find values that offer acceptable training time and accuracy, without overfitting the model to the training data. Based on some experimentation that the authors of this book have done, we will use 60 hidden nodes, 50,000 training samples, and 10 epochs. These parameters will be good enough for a preliminary test, keeping the training time down to a few minutes (depending on the processing power of your machine).

Implementing a module to train the ANN

Training an ANN based on MNIST is something you might want to do in future projects as well. To make our code more reusable, we can write a Python module that is solely dedicated to this training process. Then (in the next section, *Implementing the main module*), we will import this training module into a main module, where we will implement our demonstration of digit detection and classification.

Let's implement the training module in a file called `digits_ann.py`:

1. To begin, we will import the `gzip` and `pickle` modules from the Python standard library. As usual, we will also import OpenCV and NumPy:

```
import gzip
import pickle

import cv2
import numpy as np
```

   We will use the `gzip` and `pickle` modules to decompress and load the MNIST data from the `mnist.pkl.gz` file. We briefly mentioned this file earlier, in the *Understanding the MNIST database of handwritten digits* section. It contains the MNIST data in nested tuples, in the following format:

```
((training_images, training_ids),
 (test_images, test_ids))
```

   In turn, the elements of these tuples are in the following format:

   - `training_images` is a NumPy array of 60,000 images, where each image is a vector of 784-pixel values (flattened from an original shape of 28 x 28 pixels). The pixel values are floating-point numbers in the range 0.0 (black) to 1.0 (white), inclusive.
   - `training_ids` is a NumPy array of 60,000 digit IDs, where each ID is a number in the range 0 to 9, inclusive. `training_ids[i]` corresponds to `training_images[i]`.
   - `test_images` is a NumPy array of 10,000 images, where each image is a vector of 784-pixel values (flattened from an original shape of 28 x 28 pixels). The pixel

values are floating-point numbers in the range 0.0 (black) to 1.0 (white), inclusive.

- `test_ids` is a NumPy array of 10,000 digit IDs, where each ID is a number in the range 0 to 9, inclusive. `test_ids[i]` corresponds to `test_images[i]`.

2. Let's write the following helper function to decompress and load the contents of `mnist.pkl.gz`:

```
def load_data():
    mnist = gzip.open('./digits_data/mnist.pkl.gz', 'rb')
    training_data, test_data = pickle.load(mnist)
    mnist.close()
    return (training_data, test_data)
```

Note that in the preceding code, `training_data` is a tuple, equivalent to `(training_images, training_ids)`, and `test_data` is also a tuple, equivalent to `(test_images, test_ids)`.

3. We must reformat the raw data in order to match the format that OpenCV expects. Specifically, when we provide sample output to train the ANN, it must be a vector with 10 elements (for 10 classes of digits), rather than a single digit ID. For convenience, we will also apply Python's built-in `zip` function to reorganize the data in such a way that we can iterate over matching pairs of input and output vectors as tuples. Let's write the following helper function to reformat the data:

```
def wrap_data():
    tr_d, te_d = load_data()
    training_inputs = tr_d[0]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    test_data = zip(te_d[0], te_d[1])
    return (training_data, test_data)
```

Note that the preceding code calls `load_data` and another helper function, `vectorized_result`. The latter function converts an ID into a classification vector, as follows:

```
def vectorized_result(j):
    e = np.zeros((10,), np.float32)
```

```
        e[j] = 1.0
        return e
```

For example, the ID `1` is converted into a NumPy array containing the values `[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]`. This 10-element array, as you may have guessed, corresponds to the ANN's output layer, and we can use it as a sample of correct output when we train the ANN.

> **Note**
>
> The preceding functions – `load_data`, `wrap_data`, and `vectorized_result` – have been adapted from Nielsen's code for loading his version of `mnist.pkl.gz`. For more information about Nielsen's work, refer to the *Understanding the MNIST database of handwritten digits* section of this chapter.

4. So far, we have written functions that load and reformat MNIST data. Now, let's write a function that will create an untrained ANN:

```
def create_ann(hidden_nodes=60):
    ann = cv2.ml.ANN_MLP_create()
    ann.setLayerSizes(np.array([784, hidden_nodes, 10]))
    ann.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 0.6, 1.0)
    ann.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP, 0.1, 0.1)
    ann.setTermCriteria(
        (cv2.TERM_CRITERIA_MAX_ITER | cv2.TERM_CRITERIA_EPS,
        100, 1.0))
    return ann
```

Note that we have hardcoded the sizes of the input and output layers, based on the nature of the MNIST data. However, we have allowed the caller of this function to specify the number of nodes in the hidden layer.

For further discussion of parameters, refer to the *Choosing training parameters for the MNIST database* section of this chapter.

5. Now, we need a training function that allows the caller to specify the number of MNIST training samples and the number of epochs. Much of the training functionality should

be familiar from our previous ANN samples, so let's look at the implementation in its entirety and then discuss some details afterward:

```
def train(ann, samples=50000, epochs=10):

    tr, test = wrap_data()

    # Convert iterator to list so that we can iterate multiple
    # times in multiple epochs.
    tr = list(tr)

    for epoch in range(epochs):
        print("Completed %d/%d epochs" % (epoch, epochs))
        counter = 0
        for img in tr:
            if (counter > samples):
                break
            if (counter % 1000 == 0):
                print("Epoch %d: Trained on %d/%d samples" % \
                      (epoch, counter, samples))
            counter += 1
            sample, response = img
            data = cv2.ml.TrainData_create(
                np.array([sample], dtype=np.float32),
                cv2.ml.ROW_SAMPLE,
                np.array([response], dtype=np.float32))
            if ann.isTrained():
                ann.train(
                    data,
                    cv2.ml.ANN_MLP_UPDATE_WEIGHTS |
                    cv2.ml.ANN_MLP_NO_INPUT_SCALE |
                    cv2.ml.ANN_MLP_NO_OUTPUT_SCALE
                )
            else:
                ann.train(
                    data,
                    cv2.ml.ANN_MLP_NO_INPUT_SCALE |
                    cv2.ml.ANN_MLP_NO_OUTPUT_SCALE
                )
    print("Completed all epochs!")
```

```
    return ann, test
```

Note that we load the data and then train the ANN incrementally by iterating over a specified number of training epochs, with a specified number of samples in each epoch. For every 1,000 training samples that we process, we print a message about the progress of the training. Finally, we return both the trained ANN and the MNIST test data. We could have just returned the ANN, but having the test data on hand is useful in case we want to check the ANN's accuracy.

6. Of course, the purpose of a trained ANN is to make predictions, so we will provide the following `predict` function in order to wrap the ANN's own `predict` method:

```
def predict(ann, sample):
    if sample.shape != (784,):
        if sample.shape != (28, 28):
            sample = cv2.resize(sample, (28, 28),
                                interpolation=cv2.INTER_LINEAR)
        sample = sample.reshape(784,)
    return ann.predict(np.array([sample], dtype=np.float32))
```

This function takes a trained ANN and a sample image; it performs a minimal amount of data sanitization by making sure the sample image is 28 x 28 and by resizing it if it isn't. Then, it flattens the image data into a vector before giving it to the ANN for classification.

7. That's all the ANN-related functionality we will need to support our demo application. However, let's also implement a `test` function that measures a trained ANN's accuracy by classifying a given set of test data, such as the MNIST test data. Here is the relevant code:

```
def test(ann, test_data):
    num_tests = 0
    num_correct = 0
    for img in test_data:
        num_tests += 1
        sample, correct_digit_class = img
        digit_class = predict(ann, sample)[0]
        if digit_class == correct_digit_class:
            num_correct += 1
    print('Accuracy: %.2f%%' % (100.0 * num_correct / num_tests))
```

Now, let's take a short detour and write a minimal test that leverages all the preceding code and the MNIST dataset. After that, we will proceed to implement the main module of our demo application.

Implementing a minimal test module

Let's make another script, `test_digits_ann.py`, in order to test the functions from our `digits_ann` module. The test script is quite trivial; here it is:

```
from digits_ann import create_ann, train, test

ann, test_data = train(create_ann())
test(ann, test_data)
```

Note that we haven't specified the number of hidden nodes, so `create_ann` will use its default parameter value: 60 hidden nodes. Similarly, `train` will use its default parameter values: 50,000 samples and 10 epochs.

When we run this script, it should print training and test information similar to the following:

```
Completed 0/10 epochs
Epoch 0: Trained on 0/50000 samples
Epoch 0: Trained on 1000/50000 samples
... [more reports on progress of training] ...
Completed all epochs!
Accuracy: 95.39%
```

Here, we can see that the ANN achieved 95.39% accuracy when classifying the 10,000 test samples in the MNIST dataset. This is an encouraging result, but let's see how well the ANN can generalize. Can it accurately classify data from an entirely different source, unrelated to MNIST? Our main application, which detects digits from our own image of a sheet of paper, will provide this kind of challenge to the classifier.

Implementing the main module

Our demo's main script takes everything we have learned in this chapter about ANNs and MNIST and combines it with some of the object detection techniques that we studied in previous chapters. Thus, in many ways, this is a capstone project for us.

Let's implement the main script in a new file called `detect_and_classify_digits.py`:

1. To begin, we will import OpenCV, NumPy, and our `digits_ann` module:

```
import cv2
import numpy as np

import digits_ann
```

2. Now, let's write a couple of helper functions to analyze and adjust the bounding rectangles of digits and other contours. As we have seen in previous chapters, overlapping detections are a common problem. The following function, called `inside`, will help us determine whether one bounding rectangle is entirely contained inside another:

```
def inside(r1, r2):
    x1, y1, w1, h1 = r1
    x2, y2, w2, h2 = r2
    return (x1 > x2) and (y1 > y2) and \
           (x1+w1 < x2+w2) and (y1+h1 < y2+h2)
```

   With the help of the `inside` function, we will be able to easily choose only the outermost bounding rectangle for each digit. This is important because we do not want our detector to miss any extremities of a digit; such a mistake in detection could make the classifier's job impossible. For example, if we detected only the bottom half of a digit, 8, the classifier might reasonably see this region as a 0.

   To further ensure that the bounding rectangles meet the classifier's needs, we will use another helper function, called `wrap_digit`, to convert a tightly-fitting bounding rectangle into a square with padding around the digit. Remember that the MNIST data contains 28 x 28-pixel square images of digits, so we must rescale any region of interest to this size before we attempt to classify it with our MNIST-trained ANN. By using a padded bounding square instead of a tightly-fitting bounding rectangle, we ensure that skinny digits (such as a 1) and fat digits (such as a 0) are not stretched differently.

3. Let's look at the implementation of `wrap_digit` in multiple stages. First, we modify the rectangle's smaller dimension (be it width or height) so that it equals the larger

dimension, and we modify the rectangle's x or y position so that the center remains unchanged:

```
def wrap_digit(rect, img_w, img_h):

    x, y, w, h = rect

    x_center = x + w//2
    y_center = y + h//2
    if (h > w):
        w = h
        x = x_center - (w//2)
    else:
        h = w
        y = y_center - (h//2)
```

4. Next, we add 5-pixel padding on all sides:

```
    padding = 5
    x -= padding
    y -= padding
    w += 2 * padding
    h += 2 * padding
```

At this point, our modified rectangle could possibly extend outside the image.

5. To avoid out-of-bounds problems, we crop the rectangle so that it lies entirely within the image. This could leave us with non-square rectangles in these edge cases, but this is an acceptable compromise; we would prefer to use a non-square region of interest rather than having to entirely throw out a detected digit just because it is at the edge of the image. Here is the code for bounds checking and cropping the rectangle:

```
    if x < 0:
        x = 0
    elif x > img_w:
        x = img_w

    if y < 0:
        y = 0
    elif y > img_h:
        y = img_h
```

```
    if x+w > img_w:
        w = img_w - x

    if y+h > img_h:
        h = img_h - y
```

6. Finally, we return the modified rectangle's coordinates:

```
    return x, y, w, h
```

This concludes the implementation of the `wrap_digit` helper function.

7. Now, let's proceed to the main part of the program. Here, we start by creating an ANN and training it on MNIST data:

```
ann, test_data = digits_ann.train(
    digits_ann.create_ann(60), 50000, 10)
```

Note

Note that we are using the `create_ann` and `train` functions from our `digits_ann` module. As we mentioned earlier (in the *Choosing training parameters for the MNIST database* section), we are using 60 hidden nodes, 50,000 training samples, and 10 epochs. Although these are the default parameter values for our functions, we specify them here anyway so that they are easier to see and modify later, in case we want to experiment with other values.

8. Now, let's load a test image that contains many handwritten digits on a white sheet of paper:

```
img_path = "./digit_images/digits_0.jpg"
img = cv2.imread(img_path, cv2.IMREAD_COLOR)
```

We are using the following image of Joe Minichino's handwriting (but, of course, you could substitute another image if you prefer):

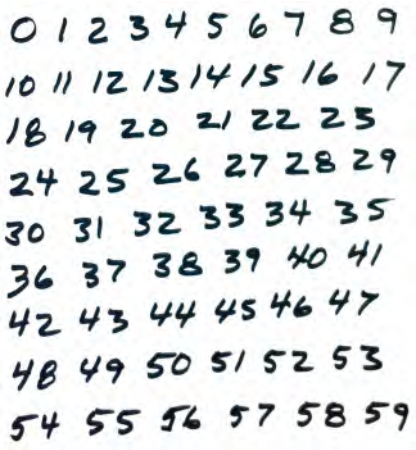


Figure 11.4: Sample of handwritten digits

9. Let's convert the image into grayscale and blur it in order to remove noise and make the darkness of the ink more uniform:

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.GaussianBlur(gray, (7, 7), 0, gray)
```

10. Now that we have a smoothed grayscale image, we can apply a threshold and some morphology operations to ensure that the numbers stand out from the background and that the contours are relatively free of irregularities, which might throw off the prediction. Here is the relevant code:

```
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)
erode_kernel = np.ones((2, 2), np.uint8)
thresh = cv2.erode(thresh, erode_kernel, thresh, iterations=2)
```

Note

Note the threshold flag, `cv2.THRESH_BINARY_INV`, which is for an inverse binary threshold. Since the samples in the MNIST database are white on black (and not black on white), we turn the image into a black background with white numbers. We use the thresholded image for both detection and classification.

11. After the morphology operation, we need to separately detect each digit in the picture. As a step toward this, first, we need to find the contours:

```
contours, hier = cv2.findContours(
    thresh, cv2.RETR_TREE,
    cv2.CHAIN_APPROX_SIMPLE
)
```

12. Then, we iterate through the contours and find their bounding rectangles. We discard any rectangles that we deem too large or too small to be digits. We also discard any rectangles that are entirely contained in other rectangles. The remaining rectangles are appended to a list of good rectangles, which (we believe) contain individual digits. Let's look at the following code snippet:

```
rectangles = []

img_h, img_w = img.shape[:2]
img_area = img_w * img_h
for c in contours:

    a = cv2.contourArea(c)
    if a >= 0.98 * img_area or a <= 0.0001 * img_area:
        continue

    r = cv2.boundingRect(c)
    is_inside = False
    for q in rectangles:
        if inside(r, q):
            is_inside = True
            break
    if not is_inside:
        rectangles.append(r)
```

13. Now that we have a list of good rectangles, we can iterate through them, sanitize them using our `wrap_digit` function, and classify the image data inside them:

```
for r in rectangles:
    x, y, w, h = wrap_digit(r, img_w, img_h)
    roi = thresh[y:y+h, x:x+w]
    digit_class = int(digits_ann.predict(ann, roi)[0])
```

Moreover, after classifying each digit, we draw the sanitized bounding rectangle and the classification result:

```
    cv2.rectangle(img, (x,y), (x+w, y+h), (255, 0, 0), 2)
    cv2.putText(img, "%d" % digit_class, (x, y-5),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
```

14. After processing all the regions of interest, we save the thresholded image and the fully annotated image and display them until the user hits any key to end the program:

```
cv2.imwrite("detected_and_classified_digits_thresh.png", thresh)
cv2.imwrite("detected_and_classified_digits.png", img)
cv2.imshow("thresh", thresh)
cv2.imshow("detected and classified digits", img)
cv2.waitKey()
```

That is the end of the script. When running it, we should see the thresholded image as well as a visualization of detection and classification results. (The two windows may overlap initially, so you might need to move one to see the other.) Here is the thresholded image:



Figure 11.5: The processed image of a handwritten digits sample

Here is the visualization of the results:



Figure 11.6: Example of detection

This image contains 110 sample digits: 10 digits in the single-digit numbers from 0 to 9, plus 100 digits in the double-digit numbers from 10 to 59. Out of these 110 samples, the bounds are correctly detected for 108 samples, meaning that the detector's accuracy is 98.18%. Then, out of these 108 correctly detected samples, the classification result is correct for 80 samples, meaning that the ANN classifier's accuracy is 74.07%. This is a lot better than a random classifier, which would correctly classify a digit only 10% of the time. But we can do better than that, and a classifier of this kind should be near 100% accuracy.

Thus, the ANN is evidently capable of learning to classify handwritten digits in general, not just the ones in the MNIST training and test datasets. Let's consider some ways to improve its learning.

Trying to improve the ANN's training

We could apply a number of potential improvements to the problem of training our ANN. We have already mentioned some of these potential improvements, but let's review them here:

- You could experiment with the size of your training dataset, the number of hidden nodes, and the number of epochs until you find a peak level of accuracy.
- You could modify our `digits_ann.create_ann` function so that it supports more than one hidden layer.
- You could also try different activation functions. We have used `cv2.ml.ANN_MLP_SIGMOID_SYM`, but it isn't the only option; the others

include `cv2.ml.ANN_MLP_IDENTITY`, `cv2.ml.ANN_MLP_GAUSSIAN`, `cv2.ml.ANN_MLP_RELU`, and `cv2.ml.ANN_MLP_LEAKYRELU`.

- Similarly, you could try different training methods. We have used `cv2.ml.ANN_MLP_BACKPROP`. The other options include `cv2.ml.ANN_MLP_RPROP` and `cv2.ml.ANN_MLP_ANNEAL`.

Note

For more information about ANN-related parameters in OpenCV, refer to the official documentation at `https://docs.opencv.org/5.x/d0/dce/classcv_1_1ml_1_1ANN__MLP.html`.

Aside from experimenting with parameters, think carefully about your application requirements. For example, where and by whom will your classifier be used? Not everyone draws digits the same way. Indeed, people in different countries tend to draw numbers in slightly different ways.

The MNIST database was compiled in the United States, where the digit 7 is handwritten like the typewritten character 7. However, in Europe, the digit 7 is often handwritten with a small horizontal line halfway through the diagonal portion of the number. This stroke was introduced to help distinguish the handwritten digit 7 from the handwritten digit 1.

Note

For a more detailed overview of regional handwriting variations, check the Wikipedia article on the subject, which is a good introduction, available at `https://en.wikipedia.org/wiki/Regional_handwriting_variation`.

This variation means that an ANN trained on the MNIST database may be less accurate when applied to the classification of European handwritten digits. To avoid such an outcome, you may choose to create your own training dataset instead. In almost all circumstances, it is preferable to utilize training data that belongs to the current application domain.

Finally, remember that once you are happy with the accuracy of your classifier, you can always save it and reload it later so that it can be utilized in applications without having to train the ANN every time.

The interface for this is just like the interface we saw in the *Saving and loading a trained SVM* section, near the end of *Chapter 7, Building Custom Object Detectors*. Specifically, you can use code such as the following to save a trained ANN to an XML file:

```
ann = cv2.ml.ANN_MLP_create()
data = cv2.ml.TrainData_create(
    training_samples, layout, training_responses)
ann.train(data)
ann.save('my_ann.xml')
```

Subsequently, you can reload the trained ANN using code such as the following:

```
ann = cv2.ml.ANN_MLP_create()
ann.load('my_ann.xml')
```

Now that we have learned how to create a reusable ANN for handwritten digit classification, let's think about the use cases for such a classifier.

Finding other potential applications

The preceding demonstration is only the foundation of a handwriting recognition application. You could readily extend the approach to videos and detect handwritten digits in real time, or you could train your ANN to recognize the entire alphabet for a full-blown **optical character recognition** (**OCR**) system.

Detection and recognition of car registration plates would be another useful extension of the lessons we have learned up to this point. The characters on registration plates have a consistent appearance (at least within a given country), and this should be a simplifying factor in the OCR part of the problem.

You could also try applying ANNs to problems where we have previously used SVMs, or vice versa. This way, you could see how their accuracy compares for different kinds of data. Recall that in *Chapter 7, Building Custom Object Detectors*, we used SIFT descriptors as inputs for SVMs. Likewise, ANNs are capable of handling high-level descriptors and not just plain old pixel data.

As we have seen, the `cv2.ml.ANN_MLP` class is quite versatile, but in truth, it covers only a small subset of the ways an ANN can be designed. Next, we will learn about OpenCV's support for more complex DNNs that can be trained with a variety of other frameworks.

Using DNNs from other frameworks in OpenCV

OpenCV can load and use DNNs that have been trained in any of the following frameworks:

- TensorFlow (`https://www.tensorflow.org/`)
- ONNX (`https://onnx.ai/`)
- DLDT (`https://github.com/openvinotoolkit/openvino/`)

The **Deep Learning Deployment Toolkit** (**DLDT**) is part of Intel's OpenVINO toolkit (`https://software.intel.com/openvino-toolkit/`) for computer vision. DLDT provides tools for optimizing DNNs from other frameworks and for converting them into a common format. A collection of DLDT-compatible models is freely available in a repository called the Open Model Zoo (`https://github.com/openvinotoolkit/open_model_zoo/`). DLDT, the Open Model Zoo, and OpenCV have some of the same people on their development teams; all three of these projects are or have been sponsored by Intel.

These frameworks use various file formats to store trained DNNs. Several of these frameworks use a combination of a pair of file formats: a text file to describe the model's parameters, plus a binary file to store the model itself. The following code snippet shows the file types and OpenCV functions that are relevant to loading a model from each framework:

```
tensor_flow_model = cv2.dnn.readNetFromTensorflow(
    'my_model.pb', 'my_model_description.pbtxt')

onnx_model = cv2.dnn.readNetFromONNX('my_model.onnx')

dldt_model = cv2.dnn.readNetFromModelOptimizer(
    'my_model_description.xml', 'my_model.bin')
```

After we load a model, we need to preprocess the data we will use with the model. The necessary preprocessing is specific to the way the given DNN was designed and trained, so any time we use a third-party DNN, we must read about how that DNN was designed and trained. OpenCV provides a function, `cv2.dnn.blobFromImage`, that can perform some common preprocessing steps, depending on the parameters we pass to it. We can perform other preprocessing steps manually before passing data to this function.

Note

A neural network's input vector is sometimes called a **tensor** or **blob**—hence the function's name, `cv2.dnn.blobFromImage`.

Let's proceed to a practical example where we'll see a third-party DNN in action.

Detecting and classifying objects with third-party DNNs

For this demo, we are going to capture frames from a webcam in real time and use a DNN to detect and classify 80 kinds of objects that may be in any given frame. Yes, a single DNN can do all this in real time on a typical laptop that a programmer might use!

Before delving into the code, let's introduce the DNN that we will use. It is an ONNX version of a model called YOLO11n, from Ultralytics. This model is documented at `https://docs.ultralytics.com/models/yolo11/`. A copy of the model's ONNX file can be found in this book's repository at `chapter11/objects_data/yolo11n.onnx`.

Note

Our `yolo11n.onnx` file has been generated using a Python toolchain from Ultralytics. If you want to generate it for yourself, first install the necessary `pip` packages as follows:

```
pip install ultralytics onnx
```

Then, run the following command to export the given YOLO model to ONNX format, for an expected input image size of 640x640 and an ONNX version known as "opset 12":

```
yolo export model=yolo11n.pt imgsz=640 format=onnx opset=12
```

When using YOLO models from Ultralytics, be aware of the licensing considerations, described at https://docs.ultralytics.com. Generally (unless you have purchased an appropriate commercial license from Ultralytics), these models are licensed under AGPL-3.0, intended strictly for non-commercial research.

This model's capabilities and proper usage will soon become clear as we progress through our sample code:

1. As usual, we begin by importing OpenCV and NumPy:

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

2. We proceed to load the ONNX model with OpenCV in the same manner that we described in the previous section:

```
model = cv2.dnn.readNetFromONNX('objects_data/yolo11n.onnx')
```

3. We need to define some preprocessing parameters that are specific to this model. It expects the input image to be 640 pixels high and 640 pixels wide. Also, it expects the pixel values in the image to be on a scale from 0.0 to 1.0. This means that, relative to the usual scale from 0 to 255, it is necessary to divide by 255.0. We define the parameters as follows:

```
blob_length = 640
color_scale = 1.0/255.0
```

4. We also define a confidence threshold, representing the minimum confidence score that we require in order to accept a detection as a real object:

```
confidence_threshold = 0.5
```

5. The model supports 80 classes of objects, with IDs from 0 to 79. The labels for these classes can be defined as follows:

```
labels = ['person', 'bicycle', 'car', 'motorcycle', 'airplane',
          'bus', 'train', 'truck', 'boat', 'traffic light',
          'fire hydrant', 'stop sign', 'parking meter', 'bench',
          'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant',
          'bear', 'zebra', 'giraffe', 'backpack', 'umbrella',
          'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
          'snowboard', 'sports ball', 'kite', 'baseball bat',
          'baseball glove', 'skateboard', 'surfboard',
          'tennis racket', 'bottle', 'wine glass', 'cup', 'fork',
          'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich',
          'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
          'donut', 'cake', 'chair', 'sofa', 'potted plant', 'bed',
          'dining table', 'toilet', 'TV or monitor', 'laptop',
          'mouse', 'remote', 'keyboard', 'cell phone', 'microwave',
          'oven', 'toaster', 'sink', 'refrigerator', 'book',
          'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
          'toothbrush']
```

With the model and parameters at hand, we are ready to start capturing frames.

6. For each frame, we begin by calculating the aspect ratio. Remember that this DNN expects the input to be based on an image that is 640 pixels high and 640 pixels wide. If the image is not already square, we can pad it out with black pixels. The following code snippet shows how we capture a frame, pad it, and calculate its scale relative to the 640x640 blob:

```
cap = cv2.VideoCapture(0)

success, frame = cap.read()
while success:

    h, w = frame.shape[:2]

    # Pad the frame to make it square.
    padded_length = max(h, w)
    padded = np.zeros((padded_length, padded_length, 3), np.uint8)
    padded[0:h, 0:w] = frame

    # Calculate the scale factor of the frame relative to the blob.
    position_scale = padded_length / blob_length
```

7. At this point, we can simply use the `cv2.dnn.blobFromImage` function, with several of its optional arguments, to perform the necessary preprocessing, including resizing the frame, converting it from BGR to RGB, and scaling its pixel data to a range of 0.0 to 1.0:

```
    # Detect objects in the frame.
    blob = cv2.dnn.blobFromImage(
        padded, scalefactor=color_scale,
        size=(blob_length, blob_length), swapRB=True, crop=False)
```

8. We feed the resulting blob to the DNN and get the model's output:

```
    model.setInput(blob)
    results = model.forward()
```

The results are an array, in a format that is specific to the model we are using.

9. For this object detection DNN – and for other DNNs trained with the YOLO11 framework – the results include a subarray of detected objects, each with its own

confidence score, rectangle coordinates, and class ID. The following code shows how to access these, as well as how to use an ID to look up a label in the list we defined earlier:

```
# Iterate over the detected objects.

    for x0, y0, x1, y1, confidence, id in results[0]:

        if confidence > confidence_threshold:

            # Scale the coordinates back to the frame size.
            x0 = int(x0 * position_scale)
            y0 = int(y0 * position_scale)
            x1 = int(x1 * position_scale)
            y1 = int(y1 * position_scale)

            # Get the class label.
            id = int(id)
            label = labels[id]
```

10. As we iterate over the detected objects, we draw the detection rectangles, along with the classification labels and confidence scores:

```
            # Draw a blue rectangle around the object.
            cv2.rectangle(frame, (x0, y0), (x1, y1),
                          (255, 0, 0), 2)

            # Draw the classification result and confidence.
            text = '%s (%.1f%%)' % (label, confidence * 100.0)
            cv2.putText(frame, text, (x0, y0 - 20),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
```

11. The last thing we do with the frame is show it. Then, if the user has hit the `Esc` key, we exit; otherwise, we capture another frame and continue to the next iteration of the loop:

```
    cv2.imshow('Objects', frame)

    k = cv2.waitKey(1)
    if k == 27: # Escape
        break
```

```
    success, frame = cap.read()
```

If you plug in a webcam and run the script, you should see a visualization of detection and classification results, updated in real time. Here is a screenshot showing Joseph Howse and Sanibel Delphinium Andromeda (a mighty, great, and righteous cat) in their living room in a Canadian fishing village:

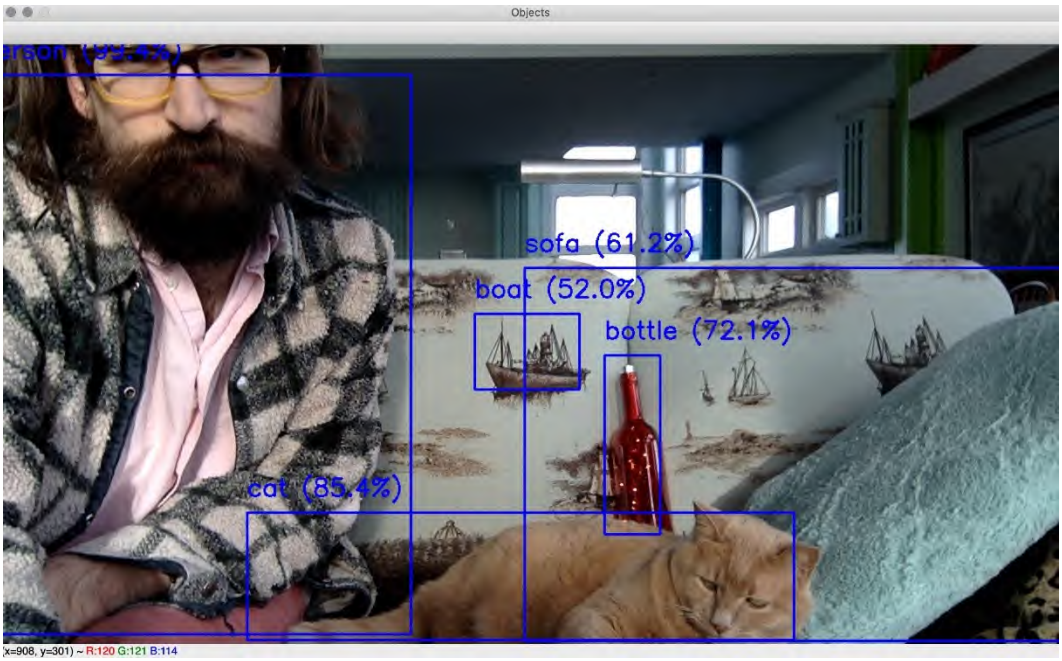


Figure 11.7: DNN detection and classification example

The DNN has correctly detected and classified a human *person* (with 99.4% confidence), a *cat* (85.4%), a decorative *bottle* (72.1%), part of a *sofa* (61.2%), and a woven picture of a *boat* (52.0%). Evidently, this DNN is well equipped to classify the contents of living rooms in nautical settings!

This is only a first taste of the things that DNNs can do—and in real time! Next, let's see what we can achieve by combining three DNNs in one application.

Detecting and classifying faces with third-party DNNs

For this demonstration, we are going to use a single DNN to detect faces and classify the age of each detected face. Specifically, we will use an ONNX version of a pre-trained model called yolo11n-face-age. A copy of it can be found at `chapter11/faces_data/`

`yolo11n-face-age.onnx`. Alternatively, the model and its documentation can be found at `https://huggingface.co/AdamCodd/yolo11n-face-age/`.

> **Note**
>
> The yolo11n-face-age model is released under a CC-BY-NC-4.0 license and is intended strictly for non-commercial research.

Let's see how we can use this model in our code:

1. To begin the sample program, we load the face detection and age classification DNN, define its parameters, and define a confidence threshold. We do this in much the same way as we did for the object detection DNN in the previous section's sample:

```
import cv2
import numpy as np

model = cv2.dnn.readNetFromONNX('faces_data/yolo11n-face-age.onnx')
blob_length = 640
color_scale = 1.0/255.0
confidence_threshold = 0.5
```

2. Now, let's define the class labels:

```
labels = ['age: 0-14', 'age: 15-22', 'age: 22+']
```

   Note that in this model, each label corresponds to a broad age range. Essentially, the model is designed to distinguish accurately between children (age 0-14) and adults (age 22+). These two classes are clearly **separable** with respect to the inputs; that is to say, the face of a 14-year-old looks very different from the face of a 22-year-old. As described in the documentation at `https://huggingface.co/AdamCodd/yolo11n-face-age/`, the model is a bit less accurate in classifying adolescents or young adults (15-22), as this class is harder to separate from the neighboring age ranges. (A 21-year-old might look much like a 22-year-old). A DNN could be trained to predict age as a continuous variable (such as a floating-point number of years), but this would be altogether different than a classifier, which predicts confidence scores for various classes.

3. Having set up our model and its parameters, let's proceed to capture and process frames from a camera. With each frame, we begin by creating a square blob and we feed this blob to the face detection DNN:

```
cap = cv2.VideoCapture(0)

success, frame = cap.read()
while success:
    h, w = frame.shape[:2]

    # Pad the frame to make it square.
    padded_length = max(h, w)
    padded = np.zeros((padded_length, padded_length, 3), np.uint8)
    padded[0:h, 0:w] = frame

    # Calculate the scale factor of the frame relative to the blob.
    position_scale = padded_length / blob_length

    # Detect objects in the frame.

    blob = cv2.dnn.blobFromImage(
        padded, scalefactor=color_scale,
        size=(blob_length, blob_length), swapRB=True, crop=False)

    model.setInput(blob)
    results = model.forward()
```

4. Like the object detector that we used in the previous section's sample, the face detector provides confidence scores and rectangle coordinates as part of its results, though the format differs somewhat. For each detected face, we need to check whether the best confidence score (across the three classes) is acceptably high, and, if it is, we'll compute the coordinates of the face rectangle in the original image:

```
    # Iterate over the detected objects.
    for cx, cy, w, h, *confidences in results[0].transpose():

        confidence = max(confidences)
        if confidence > confidence_threshold:

            # Scale the coordinates back to the frame size.
```

```
            x0 = int((cx-w/2) * position_scale)
            y0 = int((cy-h/2) * position_scale)
            x1 = int((cx+w/2) * position_scale)
            y1 = int((cy+h/2) * position_scale)
```

5. Also, for whichever class yields the highest confidence, we get the class label:

```
            # Get the class label.
            id = confidences.index(confidence)
            label = labels[id]
```

6. We draw a visualization of the detected face rectangle, the expanded square ROI, and the classification results:

```
            # Draw a blue rectangle around the face.
            cv2.rectangle(frame, (x0, y0), (x1, y1),
                          (255, 0, 0), 2)
            # Draw the classification result and confidence.
            text = '%s (%.1f%%)' % (label, confidence * 100.0)
            cv2.putText(frame, text, (x0_roi, y0 - 20),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
```

7. To conclude, we show the annotated frame, and we keep capturing more frames until the user hits the `Esc` key:

```
cv2.imshow('Faces and age', frame)

k = cv2.waitKey(1)
if k == 27: # Escape
    break

success, frame = cap.read()
```

What does this program report about Joseph Howse? Let's take a look:

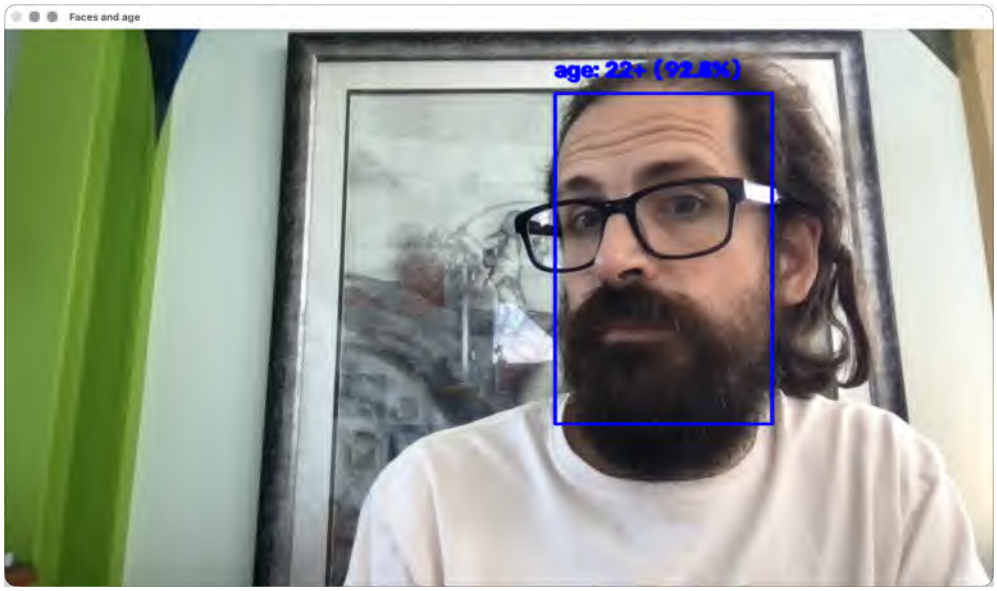


Figure 11.8: Joseph Howse as an example

Without vanity, Joseph Howse is going to write a couple of paragraphs about this result.

First, let's consider the detection of the face. Even though the face is slightly rotated, the detection rectangle accurately covers the region from cheek to cheek and from hairline to chin. Much of the beard is included, and this could be an important subregion for the purposes of classifying age.

Second, let's consider the classification. The truth is that Joseph Howse is approximately 42.4 years old at the time of this picture. Other human beings who see Joseph Howse's face are able to judge with perfect confidence that he is an adult, certainly more than 22 years old. Similarly, the age classification DNN says with 92.5% confidence that Joseph Howse is in the 22+ age range. Thus, we have a correct classification result.

This demonstration concludes our introductory tour of ANNs and DNNs. Let's review what we have learned and done before proceeding with other tools that are complementary to OpenCV.

Using OpenCV with MediaPipe and TensorFlow to classify gestures and actions

In the previous sections of the chapter, you have familiarized yourself with the idea of Artificial Neural Networks, and how they can be used in a computer vision context, coupled with OpenCV.

This wonderful match of Computer Vision and ML/AI, coupled with the exponential increase in computing power, has enabled engineers to resolve really difficult challenges, such as general object detection and recognition, tracking, etc.

Like every other area of software engineering, at first some tasks are difficult to accomplish, then they become run of the mill and then a plethora of libraries and frameworks becomes available in many programming languages, often hiding the underlying logic.

In many respects, it's like driving a car: you don't have to know how an engine works to drive a car, and if all you want to do is to drive the car from point A to point B respecting limits and rules of the road, then knowing how an engine works will add nothing to your experience.

OpenCV is your car for computer vision tasks: you don't necessarily need to know how OpenCV detects shapes to be able to do so; you just need to know OpenCV's API.

When it comes to detecting poses, faces, hands, and gestures, we have another car – it is called MediaPipe! I could not be more relieved that a library so efficient and so easy to use exists. I don't know about you, but I really do not like reinventing the wheel; I'd rather take a library and use it to achieve goals that the library alone cannot achieve for me.

Let's take a look at how we can use MediaPipe, OpenCV, and TensorFlow to build an ML classifier that will recognize a number of hand gestures such as an open hand, an "OK" sign, a "peace" sign, and a "thumb up" sign.

What is MediaPipe?

MediaPipe is a library developed by Google that makes the detection of body poses, faces, hands, the iris, and lots more (to do with a human body) a trivial task. It is highly optimized and performs really well even on commodity hardware like normal development machines.

MediaPipe's website is available at this address:

`https://developers.google.com/mediapipe/`.

MediaPipe does a lot more than simply detect body parts, so I encourage you to check it out and learn how to integrate it into your programs.

Installing MediaPipe is as easy as

```
pip install mediapipe
```

TensorFlow

TensorFlow should be familiar to you as one of the available backends for OpenCV's DNN module. If you're not familiar with it, and specifically with Keras, feel free to visit `https://keras.io/about/` where you can learn all about it. Keras is a user-friendly API

running on top of TensorFlow, which makes the building of deep learning models simple and easy.

Installing TensorFlow is also as easy as `pip install tensorflow`.

Hand recognition

Hand recognition is a task made easy by MediaPipe; let's take a look at how the documentation shows that you can perform it using webcam input:

```
import cv2
import mediapipe as mp
from mediapipe.tasks import python
from mediapipe.tasks.python import vision

# Download hand_landmarker.task from the MediaPipe model gallery:
# https://storage.googleapis.com/mediapipe-models/hand_lanmarker/hand_landmarker/
float16/1/hand_landmarker.task

HAND_CONNECTIONS = [
    (0, 1), (1, 2), (2, 3), (3, 4),
    (0, 5), (5, 6), (6, 7), (7, 8),
    (5, 9), (9, 10), (10, 11), (11, 12),
    (9, 13), (13, 14), (14, 15), (15, 16),
    (13, 17), (17, 18), (18, 19), (19, 20),
    (0, 17)
]

cap = cv2.VideoCapture(0)

base_options = python.BaseOptions(model_asset_path='hand_landmarker.task')
options = vision.HandLandmarkerOptions(
    base_options=base_options,
    num_hands=2,
    min_hand_detection_confidence=0.5,
    min_hand_presence_confidence=0.5,
    min_tracking_confidence=0.5)

with vision.HandLandmarker.create_from_options(options) as landmarker:
    while cap.isOpened():
        success, image = cap.read()
        if not success:
```

```
            print("Ignoring empty camera frame.")
            # If loading a video, use 'break' instead of 'continue'.
            continue

        rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        mp_image = mp.Image(image_format=mp.ImageFormat.SRGB,
                            data=rgb_image)
        results = landmarker.detect(mp_image)

        h, w = image.shape[:2]
        for hand_landmarks in results.hand_landmarks:
            points = [(int(lm.x * w), int(lm.y * h))
                      for lm in hand_landmarks]
            for x, y in points:
                cv2.circle(image, (x, y), 5, (0, 0, 255), -1)
            for start, end in HAND_CONNECTIONS:
                cv2.line(image, points[start],
                         points[end], (0, 255, 0), 2)

        # Flip the image horizontally for a selfie-view display.
        cv2.imshow('MediaPipe Hands', cv2.flip(image, 1))
        if cv2.waitKey(5) & 0xFF == 27:
            break

cap.release()
```

That's it for the entire code, but if we strip some of the boilerplate and the loop to continuously display the processed input, there are very few lines of real importance (aside from the necessary imports at the beginning):

```
cap = cv2.VideoCapture(0)
base_options = python.BaseOptions(model_asset_path='hand_landmarker.task')
options = vision.HandLandmarkerOptions(
    base_options=base_options, num_hands=2, ...)

with vision.HandLandmarker.create_from_options(options) as landmarker:
    while cap.isOpened():
        success, image = cap.read()
```

These lines read the input and create the hand detector, then

```
results = landmarker.detect(mp_image)
```

We perform hand detection and lastly

```
for hand_landmarks in results.hand_landmarks:
    # draw landmarks manually using cv2.circle and cv2.line
```

We draw the landmarks.

If you run the full code above, you should get an output similar to this:

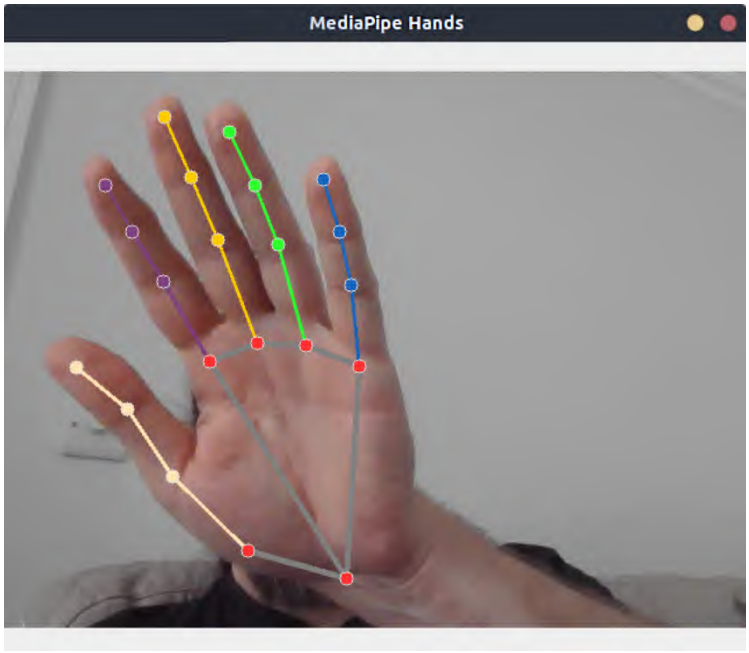


Figure 11.9: Hand keypoints detection

Hand gesture recognition

Now let's take the previous information that MediaPipe generates for us and use it to build a gesture classifier.

Here is the high-level overview of the process, which should follow a very common pattern:

1. Generate labeled training data
2. Build a model

3. Train the model
4. Test it

The two biggest challenges are the generation of data (it's a custom gesture recognition classifier; we presume such data does not exist at all) and creating a deep learning model that's accurate enough (upward of 98%).

Generating labeled training data

Let's tackle the first challenge by creating data. I am going to offer my approach to this, but clearly, you can modify it to suit your needs.

The approach I offer is as follows: start capturing video input, and for as long as you keep pressing one number key (0 to 3), the landmark coordinates (normalized) will be stored in a `csv` file with a label corresponding to the number key you are pressing.

To increase the model accuracy, make sure to move the hand around, tilt it, and turn it, all while keeping the gesture "intact," so to speak. Even more importantly, don't break the gesture or you will be polluting the training dataset and introducing noise. Lastly, make sure to be consistent with the gestures and keys; don't mix number keys and gestures up.

The algorithm looks like this:

```
WHILE key is pressed:
    IF hand detected:
        STORE key, hand coordinates
```

So let's take a look at the training data generation program:

```
import cv2
import mediapipe as mp
from mediapipe.tasks import python
from mediapipe.tasks.python import vision
import landmark_utils as u

# Download hand_landmarker.task from the MediaPipe model gallery:
# https://storage.googleapis.com/mediapipe-models/hand_landmarker/hand_landmarker/
float16/1/hand_landmarker.task

HAND_CONNECTIONS = [
    (0, 1), (1, 2), (2, 3), (3, 4),
    (0, 5), (5, 6), (6, 7), (7, 8),
    (5, 9), (9, 10), (10, 11), (11, 12),
```

```
    (9, 13), (13, 14), (14, 15), (15, 16),
    (13, 17), (17, 18), (18, 19), (19, 20),
    (0, 17)
]


def main():
    # For webcam input:
    cap = cv2.VideoCapture(0)
    number = 0

    base_options = python.BaseOptions(
        model_asset_path='hand_landmarker.task')
    options = vision.HandLandmarkerOptions(
        base_options=base_options,
        num_hands=1,
        min_hand_detection_confidence=0.5,
        min_hand_presence_confidence=0.5,
        min_tracking_confidence=0.5)

    with vision.HandLandmarker.create_from_options(options) as landmarker:
        while cap.isOpened():
            success, image = cap.read()
            if not success:
                print("Ignoring empty camera frame.")
                # If loading a video, use 'break' instead of 'continue'.
                continue
            receivedKey = cv2.waitKey(20)
            number = (receivedKey - 48) if receivedKey != -1 else -1

            rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            mp_image = mp.Image(image_format=mp.ImageFormat.SRGB,
                                data=rgb_image)
            results = landmarker.detect(mp_image)

            if results.hand_landmarks and number in [0, 1, 2, 3]:
                for hand_landmarks in results.hand_landmarks:
                    landmark_list = u.calc_landmark_list(
                        image, hand_landmarks)
                    pre_processed_landmark_list = u.pre_process_landmark(
                        landmark_list)
```

```
                    u.log_csv(number, pre_processed_landmark_list)

                    h, w = image.shape[:2]
                    points = [(int(lm.x * w), int(lm.y * h))
                              for lm in hand_landmarks]
                    for x, y in points:
                        cv2.circle(image, (x, y), 5, (0, 0, 255), -1)
                    for start, end in HAND_CONNECTIONS:
                        cv2.line(image, points[start],
                                 points[end], (0, 255, 0), 2)

            final = cv2.flip(image, 1)
            text = ""
            if number == -1:
                text = "Press key for gesture number"
            else:
                text = "Gesture: {}".format(number)
            cv2.putText(final, text, (10, 30),
                        cv2.FONT_HERSHEY_DUPLEX, 1, 255)
            # Flip the image horizontally for a selfie-view display.
            cv2.imshow('MediaPipe Hands', final)
            if cv2.waitKey(5) & 0xFF == 27:
                break
    cap.release()

if __name__ == '__main__':
    main()
```

Now let's take a closer look at the more important parts.

The first crucial part is the introduction of a "number" variable that allows us to store the class (gesture) we are capturing, so we use the familiar `waitKey` function to do that, and since the keys 0 to 9 have code 48 to 57, we subtract "48" to obtain the actual number key that we pressed:

```
receivedKey = cv2.waitKey(20)
number = (receivedKey - 48) if receivedKey != -1 else -1
```

I tailored the script to train only four classes, so I spare the program from performing any hand detection unless we are generating training data, which means that the number variable needs to be a value between 0 and 3, like so:

```
if results.hand_landmarks and number in [0, 1, 2, 3]:
```

You can edit the value in the array to your use case, but I would still use that condition to ensure you only store data when you have the correct hand position being captured.

Now for the beefiest parts of our code. We create a module called `landmark_utils` where we store two functions that will help us correct, normalize, and store the landmark's coordinates:

```
def calc_landmark_list(image, landmarks):
    image_width, image_height = image.shape[1], image.shape[0]
    landmark_point = []

    # Keypoint
    for landmark in landmarks:
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)
        # landmark_z = landmark.z

        landmark_point.append([landmark_x, landmark_y])

    return landmark_point
```

This function creates an array of landmark points and populates it by taking the values of the landmark and ensuring that the landmark coordinates are within the boundaries of the image.

This is one way of doing it; for even more accurate data, you could potentially discard any hand detection whose landmark points lie outside the frame.

The second function takes the landmark points (which are stored as two-element arrays symbolizing x and y), recalculates the x and y values as coordinates relative to the first landmark coordinates, and then normalizes the values.

```
def pre_process_landmark(landmark_list):
    temp_landmark_list = copy.deepcopy(landmark_list)

    # Convert to relative coordinates
    base_x, base_y = 0, 0
    for index, landmark_point in enumerate(temp_landmark_list):
```

```
            if index == 0:
                base_x, base_y = landmark_point[0], landmark_point[1]

            temp_landmark_list[index][0] = temp_landmark_list[
                index][0] - base_x
            temp_landmark_list[index][1] = temp_landmark_list[
                index][1] - base_y

        # Convert to a one-dimensional list
        temp_landmark_list = list(
            itertools.chain.from_iterable(temp_landmark_list))

        # Normalization
        max_value = max(list(map(abs, temp_landmark_list)))

        def normalize_(n):
            return n / max_value

        temp_landmark_list = list(map(normalize_, temp_landmark_list))

        return temp_landmark_list
```

You should be very familiar with the notion of data normalization at this point, but if you are wondering why we are doing it, it's because a hand closer to the camera appears bigger than one that is further away, and normalizing values means storing values that take proportion into consideration but ignore size.

We also squash the list into a one-dimensional array, so it can be stored in a CSV file.

The next function is the CSV logging:

```
def log_csv(number, landmark_list):
    if number > 9 or number == -1:
        pass
    csv_path = 'model/keypoint_classifier/keypoint.csv'
    with open(csv_path, 'a', newline="") as f:
        writer = csv.writer(f)
        writer.writerow([number, *landmark_list])
    return
```

Nothing too special here. If the number is not in the desired range, we don't store anything; otherwise, we simply store the flat array that we created in the previous utility function, putting the label (the number) as the first element in the row.

When we go to train the model, we will separate this first column from the rest of the data.

Great, we are ready to give it a go! Run `python train.py`, put your hand in front of the camera in the gesture you want to generate data for, press the key you are going to associate with that gesture, and move the hand around. Each frame in which a key is pressed and a hand is detected will generate a row in the `csv` training file.

If everything worked as planned, you should have a file in the folder `model/keypoint_classifier` called `keypoint.csv` with content similar to this:

```
3,0.0,0.0,-0.12693498452012383,-0.13622291021671826,-0.15789473684210525,-0.349845
2012383901,-0.09597523219814241,-0.5015479876160991,0.006191950464396285,-0.588235
2941176471,-0.034055727554179564,-0.56656346749226,-0.04643962848297214,-0.7554179
566563467,-0.043343653250773995,-0.8699690402476781,-0.030959752321981424,-0.96594
42724458205,0.09287925696594428,-0.5386996904024768,0.17647058823529413,-0.7585139
318885449,0.21671826625386997,-0.891640866873065,0.25077399380804954,-1.0,0.195046
43962848298,-0.4582043343653251,0.18885448916408668,-0.6130030959752322,0.07739938
080495357,-0.5015479876160991,0.018575851393188854,-0.3993808049535604,0.266253869
96904027,-0.3560371517027864,0.20743034055727555,-0.4582043343653251,0.09907120743
034056,-0.3746130030959752,0.04024767801857585,-0.29102167182662536
```

Granted, it doesn't look very informative, but you will notice that there are 43 fields (42 commas). Why 43? It's close to the "42" of *The Hitchhiker's Guide to the Galaxy* fame (and for that reason often used as a value for a random seed in machine learning models).

No, unfortunately, Adams's novel series has nothing to do with it. The reason why there are 43 entries is actually quite simple:

1. The first field is the label.
2. MediaPipe detects 21 landmarks (4 per finger/thumb + 1 for the base of the palm), each of which has an x value and a y value, making it 42 values.

If your generated `csv` looks like this, you are on a roll and you can move on to training the model and then testing it.

Building and training the classifier with Jupyter Notebooks

Now let's build our classifier. As we've seen for the handwritten digits, a deep learning model will do perfectly well in this case.

We know the following:

1. Each hand is represented by 42 features.
2. We are classifying 4 gestures.

So the input layer will have 42 nodes, and the output is 4. We will put another 2 layers in between, one with 20 and one with 10 nodes, but let's not rush ahead; let's go through the entire notebook step by step.

First, we import the necessary libraries and functions:

```
import csv
import tensorflow as tf
import numpy as np
from sklearn.model_selection import train_test_split
```

Following this, we establish three paths:

```
dataset = 'model/keypoint_classifier/keypoint.csv'
model_save_path = 'model/keypoint_classifier/keypoint_classifier.keras'
tflite_save_path = 'model/keypoint_classifier/keypoint_classifier.tflite'
```

The first one is the path to the training data, the second is the path to where the Keras model will be saved, and the third is where the `TFLite` model will be saved.

What is `TFLite`? **TensorFlow Lite** is a framework that allows you to run deep learning models on devices and hardware that may be constrained in terms of performance, like mobile or Raspberry Pi. The idea here is to train a TensorFlow model, save it, and then convert it to TFLite for a smaller yet fast deep learning model that can be deployed anywhere.

Next, we set the constant `NUM_CLASSES`:

```
NUM_CLASSES = 4
```

This specifies the number of gestures we are recognizing.

Now we load the training data. Note that in the `usecols` option of `loadtxt` we skip the first column (which would have index 0), because the first column is the labels:

```
X_dataset = np.loadtxt(
    dataset, delimiter=',', dtype='float32',
    usecols=list(range(1, (21 * 2) + 1))
)
```

Then we load the labels:

```
y_dataset = np.loadtxt(dataset, delimiter=',', dtype='int32', usecols=(0))
```

As expected, we only load the column at index 0, which we skipped for the training data.

```
RANDOM_SEED = 42
X_train, X_test, y_train, y_test = train_test_split(
    X_dataset, y_dataset, train_size=0.75,
    random_state=RANDOM_SEED
)
```

Then we operate the classic `x`, `y` / `train`, test split so the model can test accuracy on part of the data that has not been used for training.

The Deep Learning model

Here's our Keras model:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Input((21 * 2, )),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(20, activation='relu'),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
])
model.summary()
```

Nothing too exotic here: we have a `Dense` input layer with 42 nodes, and we gradually reduce the hidden layers (20 and 10) interspersed with `Dropout` layers to help reduce overfitting.

Next, we set up checkpoints and early stopping:

```
# Model checkpoint callback
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    model_save_path, verbose=1, save_weights_only=False)
# Callback for early stopping
es_callback = tf.keras.callbacks.EarlyStopping(patience=20, verbose=1)
```

And then we compile the model:

```
# Model compilation
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Now we "fit" the data (we train the model):

```
model.fit(
    X_train,
    y_train,
    epochs=1000,
    batch_size=128,
    validation_data=(X_test, y_test),
    callbacks=[cp_callback, es_callback]
)
```

Once the model has finished training (which, depending on the size of the training data you accumulated, might take some time), we evaluate its accuracy:

```
val_loss, val_acc = model.evaluate(X_test, y_test, batch_size=128)
```

Then we save the model:

```
model.save(model_save_path, include_optimizer=False)
```

The notebook code also contains a cell for the display of a confusion matrix if you are interested in it, but that's not particularly relevant here.

Finally, we convert the model to TFLite:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quantized_model = converter.convert()

open(tflite_save_path, 'wb').write(tflite_quantized_model)
```

And that's it; your model is ready for testing.

Testing the model on camera input

All that's left to do now is to capture some video input, create a classifier that uses the model we just saved, and do real-time predictions on any hand detected by MediaPipe.

For this we need, first of all, to create a classifier Python object. We do this in the `model/keypoint_classifier` folder, in a file called `keypoint_classifier.py`, and here is the code for it:

```
import numpy as np
import tensorflow as tf

class KeyPointClassifier(object):
    def __init__(
        self,
        model_path='model/keypoint_classifier/keypoint_classifier.tflite',
        num_threads=1,
    ):
        self.interpreter = tf.lite.Interpreter(model_path=model_path,
                                               num_threads=num_threads)

        self.interpreter.allocate_tensors()
        self.input_details = self.interpreter.get_input_details()
        self.output_details = self.interpreter.get_output_details()

    def __call__(
        self,
        landmark_list,
    ):
        input_details_tensor_index = self.input_details[0]['index']
        self.interpreter.set_tensor(
            input_details_tensor_index,
            np.array([landmark_list], dtype=np.float32))
        self.interpreter.invoke()

        output_details_tensor_index = self.output_details[0]['index']
        result = self.interpreter.get_tensor(output_details_tensor_index)
        result_index = np.argmax(np.squeeze(result))
        return result_index
```

Although some of the lines in this code may look a bit obscure, it really does not do much else other than loading the model file and specifying how to invoke the prediction.

If you are unsure, please consult the documentation for the `TFLite` interpreter at `https://www.tensorflow.org/api_docs/python/tf/lite/Interpreter`.

Of particular note are the constructor (which specifies which file to use to load the model), the `allocate_tensors()` call (which is required for execution), the `set_tensor` call (which loads the features for prediction), the `invoke` call (which performs the prediction), and the `get_tensor` call (which retrieves the result of the last prediction).

Now that we have a classifier, we only need to include it in our normal video capture loop, which I have done like so:

```
import cv2
import mediapipe as mp
from mediapipe.tasks import python
from mediapipe.tasks.python import vision
from model import KeyPointClassifier
import landmark_utils as u

# Download hand_landmarker.task from the MediaPipe model gallery:
# https://storage.googleapis.com/mediapipe-models/hand_landmarker/hand_landmarker/
float16/1/hand_landmarker.task

HAND_CONNECTIONS = [
    (0, 1), (1, 2), (2, 3), (3, 4),
    (0, 5), (5, 6), (6, 7), (7, 8),
    (5, 9), (9, 10), (10, 11), (11, 12),
    (9, 13), (13, 14), (14, 15), (15, 16),
    (13, 17), (17, 18), (18, 19), (19, 20),
    (0, 17)
]

kpclf = KeyPointClassifier()

gestures = {
    0: "Open Hand",
    1: "Thumb up",
    2: "OK",
    3: "Peace",
    4: "No Hand Detected"
```

```
}

# For webcam input:
cap = cv2.VideoCapture(0)

base_options = python.BaseOptions(model_asset_path='hand_landmarker.task')
options = vision.HandLandmarkerOptions(
    base_options=base_options,
    num_hands=2,
    min_hand_detection_confidence=0.5,
    min_hand_presence_confidence=0.5,
    min_tracking_confidence=0.5)

with vision.HandLandmarker.create_from_options(options) as landmarker:
    while cap.isOpened():
        success, image = cap.read()
        if not success:
            print("Ignoring empty camera frame.")
            # If loading a video, use 'break' instead of 'continue'.
            continue

        rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        mp_image = mp.Image(image_format=mp.ImageFormat.SRGB,
                            data=rgb_image)
        results = landmarker.detect(mp_image)

        gesture_index = 4
        h, w = image.shape[:2]
        for hand_landmarks in results.hand_landmarks:
            landmark_list = u.calc_landmark_list(image, hand_landmarks)
            keypoints = u.pre_process_landmark(landmark_list)
            gesture_index = kpclf(keypoints)

            points = [(int(lm.x * w), int(lm.y * h))
                      for lm in hand_landmarks]
            for x, y in points:
                cv2.circle(image, (x, y), 5, (0, 0, 255), -1)
            for start, end in HAND_CONNECTIONS:
                cv2.line(image, points[start],
                         points[end], (0, 255, 0), 2)
```

```
        # Flip the image horizontally for a selfie-view display.
        final = cv2.flip(image, 1)
        cv2.putText(final, gestures[gesture_index], (10, 30),
                    cv2.FONT_HERSHEY_DUPLEX, 1, (255, 255, 255))
        cv2.imshow('MediaPipe Hands', final)
        if cv2.waitKey(5) & 0xFF == 27:
            break

cap.release()
```

Note these lines:

```
kpclf = KeyPointClassifier()

gestures = {
    0: "Open Hand",
    1: "Thumb up",
    2: "OK",
    3: "Peace",
    4: "No Hand Detected"
}
```

In the first line, we create a classifier; then we create a dictionary of gestures for display purposes.

After that, all we do is detect hands, flatten the landmarks into a one-dimensional 42-size array and pass it into our classifier, and use the detected class index to print the gesture's name in the top left corner of the screen.

If all goes well, you will have very satisfactory results like so:

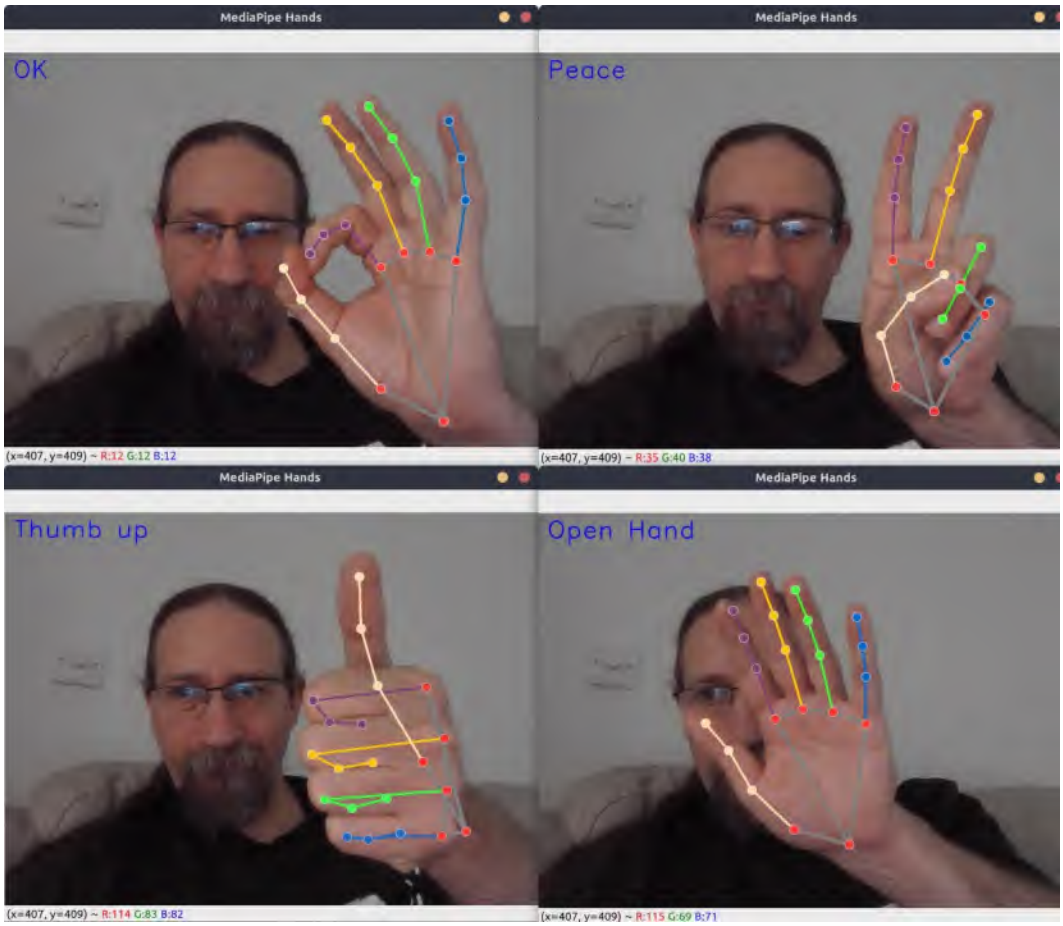


Figure 11.10: Hand gestures correctly recognized

As you can see, MediaPipe correctly detects the keypoints, which are then utilized for classification, yielding very satisfactory results!

Summary

This chapter scratched the surface of the vast and fascinating world of ANNs. We learned about the structure of ANNs and how to design a network topology based on application requirements. Then, we focused on OpenCV's implementation of MLP ANNs, as well as on OpenCV's support for diverse DNNs that have been trained in other frameworks.

We applied neural networks to real-world problems: notably, handwritten digit recognition; object detection and classification; and a combination of face detection and age classification in real time. We saw that even in these introductory demos, neural networks show a lot of promise in terms of versatility, accuracy, and speed. Hopefully, this encourages you to try out pre-trained models from various authors and learn to train advanced models of your own in various frameworks. Finally, we have learned how to use MediaPipe for hand detection, trained a classifier with TensorFlow, and utilized OpenCV for video capture and image manipulation.

We hope you enjoyed it and that you have learned useful techniques that you can apply in real-life scenarios or your personal projects!

12

OpenCV Applications at Scale

In this chapter, we will take OpenCV to the cloud by exploring technologies and methodologies that will allow you to package your application and deploy it on AWS so it can be made available to the public even on a large scale, without having to learn lengthy and complex technologies to maintain such applications.

We will provide a brief introduction to containers and containerization tools like **Docker**, an overview of the **AWS cloud provider** and **AWS Lambda**, and finally, we will demonstrate how to develop a simple application and deploy it.

We will cover the following topics in this chapter:

- What are containers?
- Docker basics
- AWS Serverless (Fargate, Lambda), and AWS SAM

Technical requirements

If you don't want to use AWS, the knowledge you will acquire about containerization should seamlessly transfer to other cloud providers or cloud-agnostic solutions such as Kubernetes (which can be deployed on any cloud).

To follow the examples in this chapter, you will need an AWS account. While you have to provide credit card credentials, if you follow the examples in this chapter, you will only incur charges when you exceed free-tier quotas, something you would normally only do if you were working with revenue-generating levels of loads.

You can download the example project and code for this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Understanding containers and Docker basics

We mentioned containers and Docker. Let's see what they are and why they are so important in our day and age.

First of all, containers are a form of virtualization; that is, making a single physical machine run on one or more "machines," which are called virtual.

But while **Virtual Machines** (**VMs**) are an abstraction of the hardware layer and simulate an entire operating system, in fact, VM images are huge, in the order of GBs.

Containers, however, are an abstraction of the application layer, and container images can be as small as tens of MB, and they package code and dependencies into a single image that can be deployed in several ways.

The most popular technology for container deployment and orchestration is Kubernetes, and in AWS you can do this with a Kubernetes equivalent (**Elastic Kubernetes Service** or **EKS**), or three separate tools called **Elastic Container Service** (**ECS**), **Fargate**, and **AWS Lambda**.

We will limit ourselves to the simplest and easiest (from a management point of view) of these tools, AWS Lambda, but all in good time.

Traditional VMs can be very slow to boot and, because they are so cumbersome, not very agile to work with. It is no wonder they are only used in specific cases (and mainly to do with security), but other than niche use cases, they have been supplanted by containers.

The main advantage of containers from a software development cycle point of view is portability.

This means that you can develop an application, package it into a container for local testing and use in your local development environment, and then deploy it as it is, and it will work because the abstraction is operated by Docker Engine, not the underlying OS.

It is quite literally that simple. Some say containers resolved the age-old rebuttal:

> *"It works on my machine."*

Developers often blurt this out when things don't seem to work as expected. If a container works on your machine and not in a production environment, there are a few very precise reasons for it, and it will not be due to complex and annoying dependency issues. This is where the "12-factor app" deserves a brief mention.

The 12-Factor app

There is a website called `https://12factor.net/` that illustrates a simple methodology that is perfectly suited to the delivery of an application as a web app or as **Software as a Service** (**SaaS**).

If you are reading this chapter (and not happily skipping), then you either want to deploy your OpenCV app in the cloud, or you intend to acquire the knowledge to do so; therefore, you might as well learn a tried-and-tested methodology that will greatly simplify your life, especially when it comes to preventing problems from happening at all and then resolving them in the easiest and quickest way possible.

Let's take a quick look at the factors as they are illustrated on the website and see how they fit into our container-based work:

1. **Codebase**: One codebase tracked in revision control, many deploys
2. **Dependencies**: Explicitly declare and isolate dependencies
3. **Config**: Store config in the environment
4. **Backing services**: Treat backing services as attached resources
5. **Build, release, and run**: Strictly separate build and run stages
6. **Processes**: Execute the app as one or more stateless processes
7. **Port binding**: Export services via port binding
8. **Concurrency**: Scale out via the process model
9. **Disposability**: Maximize robustness with fast startup and graceful shutdown
10. **Dev/Prod parity**: Keep development, staging, and production as similar as possible
11. **Logs**: Treat logs as event streams
12. **Admin processes**: Run admin/management tasks as one-off processes

Let's focus on a few that stand out as particularly relevant, or that containers can help to make easier:

1. **Dependencies**: You package the dependencies of your code in your container, and you have to declare such dependencies in your container build file (called a Dockerfile) so they are automatically downloaded and installed before your code can be built in the container itself. This achieves total isolation and explicit declaration.
2. **Config**: You declare configuration parameters in container-based development in environment variables. Things like database connection strings, stage, names of resources etc. are all declared as environment variables.

3. **Disposability**: Containers are so light they are destroyed and recreated rather than moved.
4. **Dev/Prod parity**: The fact that your container image deployed locally for testing only differs from the production image by the environment variables injected into it; it means you have complete dev/prod parity, which is incredibly important.

Docker basics

We have talked enough about containers and Docker; let's move on to an example so the whole idea becomes clearer for those unfamiliar with containerization technologies.

> **Note**
>
> Our overview of Docker will be minimal because it is far from the scope of this tutorial, which is not to teach you Docker but to teach how to deploy OpenCV applications to scale. We will explore basic concepts and refer you to the Docker documentation available at `https://docs.docker.com/` for anything else you need to know.

First, let's see how to install Docker Engine. The official page on the Docker website where you can obtain downloads is at `https://docs.docker.com/engine/install/`

Docker Engine used to be a CLI-only tool that lacked a bit of user-friendliness. Nowadays, what you download from Docker is a tool called Docker Desktop, which also includes the CLI client that Docker old-timers and CLI aficionados are used to (and probably prefer to use).

Throughout the chapter, we will use the CLI, not just because I'm used to it, but because the overwhelming majority of support material, including documentation and Stack Overflow questions, will be using the CLI, so it's simply the right thing to do. When all is said and done, the CLI is the more difficult tool to learn and the GUI the intuitive, easy one, so if we learn the difficult parts, I'm sure you will be flying with the easy ones.

Docker version

So, go ahead and install `docker-desktop`; you should then be able to verify that your Docker client is installed from a terminal window:

```
$ docker -v
```

Docker run

`docker run` allows you to launch a container. If you have never done so, try the obligatory "`hello-world`":

```
$ docker run hello-world
```

This will test that the installation is working correctly and will produce the following output:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:13e367d31ae85359f42d637adf6da428f76d75dc9afeb3c21faea0d976f5c651
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID: `https://hub.docker.com/`

For more examples and ideas, visit: `https://docs.docker.com/get-started/`

The output itself describes step by step what the command did, which in short meant contacting the local daemon, failing to find the "image" called `hello-world` and therefore downloading it from Docker Hub (a publicly available repository of container images), then creating the container and running it, and printing the output to your terminal.

Clearly the container itself does not do much, but it was enough to showcase the workflow adopted by the Docker client.

So, the first subcommand we learned is `run`, which is what we use to launch a container. If you `run` a container, it will immediately execute a command that you specified in the build file for the container (Dockerfile), which we will explore soon enough.

Docker pull

Next up, we want to actually do something with containers, so we want to download an image that will allow you to write and execute a Python program, which is a prerequisite of any Python OpenCV application.

To do so, try the following:

```
$ docker pull python:3.13
```

This will produce an output similar to the following:

```
1339eeaac5b67: Downloading [========>                                  ] 25.83MB/
55.01MB
4cd023948de34: Download complete
3298fed8291cd: Pulling fs layer

joe@hostname:~$ docker pull python:3.13 3.13: Pulling from library/python Status:
Downloaded newer image for python:3.13
1339eeaac5b67: Downloading [========>                                  ] 25.83MB/
55.01MB
4c78fa1b9799: Download complete
14f0d2bd5243: Download complete
76e5964a957d: Downloading [========>                                 ] 31.78MB/
54.58MB
cc4bb1a04a94: Downloading [========>                                 ] 35.51MB/
196.8MB
3dee34a94cb0: Waiting
14451d1a3feb: Pulling fs layer
B39eb9ce1023: Pulling fs layer
E90c569049a2: Waiting
```

This is deceptively important to understand: containers leverage "filesystem layers," which allows caching of parts of the image so that when you apply changes to it (and go to upload the image to a remote repository), you don't have to re-upload the entire image but only the parts

that changed. If the standard image has something like nine layers, a simple code change (and no dependency changes) may involve only one or two layer updates, making your data transfer lighter and faster, and future downloads quicker too.

This `python:3.13` image that you downloaded does not do anything, but you can launch it as a service and then log into it, much like "SSH'ing" into a remote server.

Dockerfile

Dockerfile is the Docker equivalent of a Makefile, and it is probably not a coincidence that they sound so similar.

It is a build file that specifies:

1. `FROM` what base image will your container be derived
2. What files to `COPY` or volumes to mount on your image
3. What preliminary operations to `RUN`, such as downloading dependencies and building an artifact
4. What network ports to `EXPOSE`
5. What command (`CMD`) to run once everything is ready

Notice where I put the words in capital letters. That's not a coincidence: I used Dockerfiles' directives to explain the steps involved in creating a container image, and they follow a precise order. There are a lot more directives in a Dockerfile, but these will get us started.

Now, let's create an extremely simple `hello-world`-type Python application and then containerize it by writing a Dockerfile and then using `docker build`.

Let's write a `main.py` file with the following code:

```
import cv2
print("hello docker and OpenCV, we're using OpenCV {}".format(cv2.__version__))
```

If you run it on your machine, it should output something like the following:

```
$ python3 main.py
hello docker and OpenCV, we're using OpenCV 5.0.0
```

(OpenCV version may vary, of course, depending on your installation.)

Now, if we run the base image `python:3.13` in Docker, but with the option to make the container interactive and allocate a terminal, we will notice that we are sent straight into the Python shell:

```
$ docker run -it python:3.13
Python 3.13.15 (main, Aug 25 2026, 03:39:59)
[GCC 14.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

That `-it` is a double flag, where `i` stands for `--interactive` and `t` for `AllocateTTY`; in other words, it allows you to interact with the container through a terminal.

If you then run, in another terminal window, the following command:

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS                         PORTS     NAMES
cf06fe9e6516   python:3.13   "python3"   57 minutes ago   Up 57 minutes
admiring_swanson
```

You will notice that the container's `COMMAND` is `python3`, which adds up to the behavior we just witnessed where launching the container and interacting with it brought us into the Python shell.

So what we want to do now is replace the "python3" command with the execution of our little script.

To do so, we need to perform a few operations:

1. Create a `requirements.txt` file for our script.
2. Write a Dockerfile that tells Docker to build a container image that contains a copy of `main.py` and `requirements.txt`, and installs the dependencies in `requirements.txt`.
3. Instruct the Dockerfile to run `python3 main.py` rather than just `python3`.
4. Build the image and run it to test that it works.

Note

You need to create the `requirements.txt` file. I normally use a handy tool called `pipreqs`; you can use `pip freeze` or any other tool that allows you to create a `requirements.txt` file, which is necessary to install dependencies in the container.

Let's jump straight into the Dockerfile to check how it matches the above description of the Dockerfile directives and the order of steps to be taken during the image build process:

```
FROM python:3.13

COPY main.py requirements.txt ./

RUN pip install -r requirements.txt

CMD ["python3", "main.py"]
```

The above Dockerfile essentially says, "create a container image that contains the `main.py` and `requirements.txt` files from the current directory, run `pip install`, and then run the program `main.py`."

To actually build the container, we use `docker build`. The minimal version of `docker build` is extremely simple, and you only need to specify two elements:

1. The tag of the container (basically its name, for future reference)
2. The path to the Dockerfile (normally `.` to represent the current directory)

Let's go ahead and do that:

```
$ docker build -t opencv5-python3.13:latest .
[+] Building 62.3s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
 => [internal] load metadata for docker.io/library/python:3.13
 => [1/3] FROM docker.io/library/python:3.13
 => [2/3] COPY main.py requirements.txt ./
 => [3/3] RUN pip install -r requirements.txt
 => => Collecting opencv-python==5.0.0.93
 => => Downloading opencv_python-5.0.0.93-cp37-abi3-manylinux_2_28_x86_64.whl
(73.8 MB)
 => => Collecting numpy>=2
 => => Downloading numpy-2.5.2-cp313-cp313-
manylinux_2_27_x86_64.manylinux2014_x86_64.whl (16.7 MB)
 => => Installing collected packages: numpy, opencv-python
 => => Successfully installed numpy-2.5.2 opencv-python-5.0.0.93
 => exporting to image
Successfully tagged opencv5-python3.13:latest
```

As you can see, we called the container `opencv5-python3.13:latest`.

Now we can run it with docker run:

```
$ docker run opencv5-python3.13:latest
Traceback (most recent call last):
  File "//main.py", line 1, in <module>
    import cv2
  File "/usr/local/lib/python3.13/site-packages/cv2/__init__.py", line 181, in
<module>
    bootstrap()
  File "/usr/local/lib/python3.13/site-packages/cv2/__init__.py", line 153, in
bootstrap
    native_module = importlib.import_module("cv2")
  File "/usr/local/lib/python3.13/importlib/__init__.py", line 127, in
import_module
    return _bootstrap._gcd_import(name[level:], package, level)
ImportError: libGL.so.1: cannot open shared object file: No such file or directory
```

That's terrible – we have a missing dependency. Not to worry; this is all very much calculated to give you a complete perspective of the Docker workflow. When you install opencv-python on your local machine, your OS already provides the GUI and display libraries it needs (GTK, Qt, OpenGL). In a minimal container, those libraries are absent, which is why the import fails with a missing libGL.so.1 error.

The fix is straightforward: use opencv-contrib-python-headless instead of opencv-python. This headless contrib build avoids GUI dependencies and also provides cv2.CascadeClassifier, which the face-detection example later in this chapter requires under OpenCV 5.

Update your requirements.txt to use the headless contrib package:

```
# In requirements.txt, change opencv-python to opencv-contrib-python-headless:
opencv-contrib-python-headless==5.0.0.93
```

A headless OpenCV package is the right choice for a server or cloud environment because it omits GUI and display-related code (GTK, Qt, and OpenGL). This example specifically uses opencv-contrib-python-headless because OpenCV 5 supplies CascadeClassifier through the contrib modules. On a desktop, use the corresponding non-headless contrib package if GUI features are required.

```
FROM python:3.13
```

```
COPY main.py requirements.txt ./

RUN pip install -r requirements.txt

CMD ["python3", "main.py"]
```

Before we run it again, we need to rebuild the image so it actually picks up the headless contrib package; otherwise, docker run would simply start the previous, still-broken image:

```
[-- omitted output --]
Step 1/4 : FROM python:3.13
 ---> 6faba2c56370
Step 2/4 : COPY main.py requirements.txt ./
 ---> bf13c9db77d2
Step 3/4 : RUN pip install -r requirements.txt
 ---> Running in e8e061148ac1
Collecting opencv-contrib-python-headless==5.0.0.93
Collecting numpy>=2
Downloading opencv_contrib_python_headless-5.0.0.93-cp37-abi3-
manylinux_2_28_x86_64.whl (69.5 MB)
Downloading numpy-2.5.3-cp313-cp313-
manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl (16.7 MB)
Installing collected packages: numpy, opencv-contrib-python-headless
Successfully installed numpy-2.5.3 opencv-contrib-python-headless-5.0.0.93
```

Now, when we go to run our previously failing command, we get a successful output:

```
$ docker run opencv5-python3.13:latest
hello docker and OpenCV, we're using OpenCV 5.0.0
```

A note on Linux distributions in Docker images

You might have noticed that we used Ubuntu or Ubuntu-based OSs for our Docker images, and that installation of software happens through `apt`, which is a package manager available in Debian-based distributions. This is not mandatory; you can use whichever distribution you want. In fact, in recent times Windows images have become available, so you could theoretically even use Windows for your containers. Illustrating all the possible OSs and ways to install software on those is unrealistic, so we will show you the most common way to do this, but you are free to use your Docker images on whatever Linux distribution or OS you prefer. You will have to adjust the instructions, such as using Windows-compatible paths, etc.

Also note that some of the feedback/output produced by the tools we are going to use may slightly differ from OS to OS, but not in its usage.

Now that we possess the basics for developing with OpenCV, we can go on to create an application that will leverage container technology and deploy it with some simple steps.

AWS Serverless (Fargate, Lambda) and AWS SAM

As mentioned, the reason why we are focusing on Lambda functions is their serverless nature.

Serverless clearly does not mean "without servers," but rather "without the need to provision and manage servers.", which, if you ask me, is a great selling point for all computer vision developers out there who want to put their code into production in a web context, without having to learn more than the basics of web development (which is really just how to handle a web request).

This idea of developing applications without having to provision the underlying resources isn't new, but the flexibility and stability provided by containers push this aspect of development and deployment to another level.

Moreover, even in existing products that encapsulate your application in an environment that makes deployment easy, scalability remains an issue.

AWS Fargate and AWS Lambda are resolving exactly this problem. They will take care of scalability for you, and if you reach a global scale, you can easily configure your Lambda deployment to be geographically distributed for a faster response time for your users.

Let's take a look at these technologies individually and understand why they are a big deal.

AWS Lambda

Think about a small OpenCV application, such as one that detects the coordinates of faces in a picture. This is a few lines of code in Python, along these lines:

```
import cv2

cascader = cv2.CascadeClassifier('data/haarcascades/
haarcascade_frontalface_default.xml')  # path relative to the repository root

image = cv2.imread('napoli.jpg')  # downloaded beforehand from https://
img.konami.com/kde_cms/eu_publish/uploads/pes2021_sscnapoli_players-600x338.jpg

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

faces = cascader.detectMultiScale(gray, 1.1, 4)
```

```
print(faces)
```

This is so simple that it doesn't even draw the rectangles around the faces like we've been doing in the whole book; it simply gives the x and y coordinates of the top-left corner of the face, its width, and its height.

If you were to deploy this as a web application, you would need to provision servers, configure them, and provision and configure ancillary services such as web servers, load balancers, etc.; that's before we talk about the boilerplate code necessary to finally be able to wrap your six lines of code into an endpoint that can return such simple information.

This is precisely what Lambda resolves. Lambda takes its name from Lambda functions, inline functions that take an input and produce an output, because AWS wants you to think about your code exactly like that: a self-contained, isolated piece of code that can be deployed as is and executed on demand.

You, as a computer vision engineer, need only focus on the part you love (the computer vision coding), and AWS will take care of the rest.

To learn how to deploy a simple piece of code in a Lambda, let's take a look at the **Serverless Application Model** (**SAM**).

AWS SAM

If Lambda resolves the problem of being free of provisioning and configuring servers to deploy your applications, SAM makes it a piece of cake.

Although a complex and complete tool capable of provisioning the most complex of architectures and related infrastructure, SAM is particularly useful when you want to create applications from templates, and AWS generously provides you with ready-made configurations that cover the most common use cases, from a single, standalone Lambda, to a REST API, to a machine learning inference application.

To install AWS SAM, we need to install two tools: the `aws-cli` (your Swiss Army knife for all things AWS) and `sam-cli`. Both are very straightforward to install and only require minimal configuration.

The documentation for the installation of these tools on the various OSs is available at the following addresses:

- AWS CLI:

`https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html`

- AWS SAM: `https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install.html`

Both links contain specific installation instructions for Windows, Linux, and Mac OS.

The last step before you are good to use either tool is to configure it with your account credentials. There are two different files that can be used to configure your command-line tools: the `.config` file and the `.credentials` file.

In the `.config` file, we store configuration preferences, such as the region where you are going to operate and the output format of the commands. In the .credentials file, we store the actual credentials needed to connect to your AWS account and perform operations on it.

The full guide on configuration is here: `https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started-set-up-credentials.html`

Bear in mind that a region, an access key, and a secret key are all that are really needed to start working; the rest is entirely optional and down to preference.

If everything works as expected, you will have AWS SAM installed on your machine.

So what can SAM do for us? First of all, SAM interactively guides you through the creation of a project by asking questions that have multiple-choice answers. AWS was feeling really helpful when they created SAM because if you accept all default options, it all works.

However, in our case we are deploying OpenCV Python applications, so this blind acceptance of default options won't work, but don't worry, it is still pretty simple.

The second thing SAM takes care of is the deployment of our application. Again, if you just accepted all default choices, it would work like a charm (I know; I have done it multiple times!). All you need to do is answer interactive questions about your deployment preferences.

So let's go ahead and create a simple Python project, and later evolve it into a face detection application.

First, initialize the application:

```
$ sam init
Which template source would you like to use?
    1 - AWS Quick Start Templates
```

```
    2 - Custom Template Location
Choice: 1
```

Here we told `sam-cli` to choose one of the ready-made templates. We are new to SAM, so we most definitely do not have custom templates of our own.

SAM proceeds to download these templates' definitions:

```
Cloning from https://github.com/aws/aws-sam-cli-app-templates

Choose an AWS Quick Start application template
    1 - Hello World Example
    2 - Multi-step workflow
    3 - Serverless API
    4 - Scheduled task
    5 - Standalone function
    6 - Data processing
    7 - Infrastructure event management
    8 - Machine Learning
Template: 1
```

Once again, we choose the first option because we want to start from scratch and are completely unfamiliar with serverless APIs. The `Hello World` example creates a SAM application for a single Lambda function that will perform some kind of work for us.

Use the most popular runtime and package type? `(Nodejs and zip) [y/N]: N`

```
Which runtime would you like to use?
1 - dotnet8
2 - go1.x
3 - go (provided.al2)
4 - graalvm.java11 (provided.al2)
5 - graalvm.java17 (provided.al2)
6 - java11
7 - java17
8 - java21
9 - nodejs18.x
10 - nodejs20.x
11 - nodejs22.x
12 - python3.10
13 - python3.11
```

```
14 - python3.12
15 - python3.13
16 - ruby3.2
17 - ruby3.3
Runtime: 15
```

Now we are asked about the runtime to use: we reject using the defaults (because it's Node.js packaged as a .zip) and instead choose python3.13 for our runtime…

```
What package type would you like to use?
    1 - Zip
    2 - Image
Package type: 2
```

And `Image` as the package type. Image means we are going to use a container image, so yes, we are going to put our newfound Docker knowledge to good use:

```
Based on your selections, the only dependency manager available is pip.

We will proceed with copying the template using pip.
```

```
Project name [sam-app]: sam-opencv

    ----------------------
    Generating application:
    ----------------------
    Name: sam-opencv
    Base Image: amazon/python3.13-base
    Architectures: x86_64
    Dependency Manager: pip
    Output Directory: .
    Next steps can be found in the README file at ./sam-opencv/README.md


    Commands you can use next
    =========================
    [*] Create pipeline: cd sam-opencv && sam pipeline init --bootstrap
    [*] Test Function in the Cloud: sam sync --stack-name {stack-name} --watch
```

SAM subsequently creates the skeleton of the application, which we are going to use as a basis for our further OpenCV work.

Let's move over to the newly created `sam-opencv` folder and start working from there.

The structure of the project is as follows:

```
.
├── events
│   └── event.json
├── hello_world
│   ├── app.py
│   ├── Dockerfile
│   ├── __init__.py
│   └── requirements.txt
├── __init__.py
├── README.md
├── template.yaml
└── tests
    ├── __init__.py
    └── unit
        ├── __init__.py
        └── test_handler.py

4 directories, 11 files
```

Let's go through it. The main file that defines the resources in the project is `template.yaml`, which is a CloudFormation template.

For those unfamiliar, CloudFormation is an infrastructure management tool that allows you to declare resources in code (Infrastructure as Code) and to deploy, update, and destroy infrastructure from the command line.

The sam-cli will take care of all of that.

The second main element of interest is the `hello-world` folder, which contains the code resources for the Lambda function. For simplicity, we will leave the function name as `HelloWorld`, but clearly in production scenarios you may want to rename it to a more meaningful identifier such as `FaceDetectionFunction`, for example.

Within the hello-world folder, we have four files:

1. `app.py` containing the actual code.
2. Dockerfile, which will be needed for the container image for the Lambda.

3. The __init__.py is only there to make sure the current folder is treated as a module.
4. The `requirements.txt` file is there because the Dockerfile will run pip install during the build phase.

So let's take a look at the `app.py` script:

```
import json


def lambda_handler(event, context):
    """Sample pure Lambda function

    Parameters
    ----------
    event: dict, required
        API Gateway Lambda Proxy Input Format

        Event doc: https://docs.aws.amazon.com/apigateway/latest/developerguide/
set-up-lambda-proxy-integrations.html#api-gateway-simple-proxy-for-lambda-input-
format

    context: object, required
        Lambda Context runtime methods and attributes

        Context doc: https://docs.aws.amazon.com/lambda/latest/dg/python-context-
object.html

    Returns
    ------
    API Gateway Lambda Proxy Output Format: dict

        Return doc: https://docs.aws.amazon.com/apigateway/latest/developerguide/
set-up-lambda-proxy-integrations.html
    """

    return {
        "statusCode": 200,
        "body": json.dumps(
            {
                "message": "hello world",
            }
```

```
        ),
    }
```

It does not do much, but it reveals important information about how to structure our code. Global imports at the top of the file, a main function called `lambda_handler`, which will perform the actual work, and since the hello-world function is a simple web request, we see that we can just return a dictionary.

With this information in mind, we move to rewrite our code to do something extremely simple, but that proves we have OpenCV correctly installed and running in the cloud in a Lambda function:

```
import json
import cv2

def lambda_handler(event, context):
    return {
        "statusCode": 200,
        "body": json.dumps(
            {
                "message": "hello OpenCV {} in a Lambda".format(cv2.__version__),
            }
        ),
    }
```

Great, at this point we only want to focus on learning how to take this simple code and deploy it and nothing more.

Once this part works, adding complexity to the code will not affect the deployment process in any way.

Before moving on to building the Dockerfile, we need to create the `requirements.txt` file. As I mentioned, there are a variety of ways (not least typing one manually, but I would not advise doing so). I use `pipreqs`; you can use pip freeze or whatever tool you prefer for the task.

With the requirements.txt file created and stored in the same folder as the Dockerfile, we can now edit the Dockerfile. The official AWS Lambda Python base image already includes the Lambda **Runtime Interface Client** (**RIC**) and **Runtime Interface Emulator** (**RIE**), so all we need is to install our dependencies and copy our code:

```
FROM public.ecr.aws/lambda/python:3.13

# Install pinned dependencies from requirements.txt (opencv-contrib-python-
headless
# is required under OpenCV 5 for cv2.CascadeClassifier)
COPY requirements.txt ${LAMBDA_TASK_ROOT}/
RUN pip install --no-cache-dir -r ${LAMBDA_TASK_ROOT}/requirements.txt
# Copy function code and Haar cascade classifier
COPY app.py cascade.xml ${LAMBDA_TASK_ROOT}/

CMD ["app.lambda_handler"]
```

So, to build the Lambda (which is a required step before we can deploy it), we run:

```
$ sam build --use-container
```

Let's examine some of the output (I purposely omitted some as it was unnecessarily verbose):

```
Starting Build inside a container
Building image for HelloWorldFunction function
Setting DockerBuildArgs for HelloWorldFunction function
Step 1/5 : FROM public.ecr.aws/lambda/python:3.13
Step 2/5 : COPY requirements.txt ${LAMBDA_TASK_ROOT}/
Step 3/5 : RUN pip install --no-cache-dir -r ${LAMBDA_TASK_ROOT}/requirements.txt
Collecting opencv-contrib-python-headless==5.0.0.93 (from -r /var/task/
requirements.txt (line 1))
Collecting requests==2.32.3 (from -r /var/task/requirements.txt (line 2))
[-- omitted dependency and download output --]
Successfully installed certifi-2026.7.22 charset-normalizer-3.5.2 idna-3.20
numpy-2.5.3 opencv-contrib-python-headless-5.0.0.93 requests-2.32.3
[-- omitted intermediate container output --]
Step 4/5 : COPY app.py cascade.xml ${LAMBDA_TASK_ROOT}/
Step 5/5 : CMD ["app.lambda_handler"]
Successfully built ce1cdf240d56
Successfully tagged helloworldfunction:python3.13-v1

Build Succeeded

Built Artifacts  : .aws-sam/build
Built Template   : .aws-sam/build/template.yaml
```

Great, the container image is built; now we have to deploy it, and SAM is so kind as to actually show us the command for the purpose:

```
Build Succeeded

Built Artifacts  : .aws-sam/build
Built Template   : .aws-sam/build/template.yaml

Commands you can use next
=========================
[*] Invoke Function: sam local invoke
[*] Test Function in the Cloud: sam sync --stack-name {stack-name} --watch
[*] Deploy: sam deploy --guided
```

Let's stop for a second, though. We don't want to go through the hassle of deployment, which takes a little time, without verifying that things are working properly.

You may have noted that SAM suggests you use `sam local invoke`, which is a command that allows us to test the Lambda locally. Naturally, if our Lambda were connecting to resources, it would be a little trickier to test, but our code does not rely on external resources, so it should work.

Let's run the `sam local invoke`:

```
$ sam local invoke
Invoking Container created from helloworldfunction:python3.13-v1
Building image.................
Skip pulling image and use local one: helloworldfunction:rapid-1.38.0-x86_64.

START RequestId: 5d87f6db-3151-481f-97f4-92852478000f Version: $LATEST
END RequestId: 5d87f6db-3151-481f-97f4-92852478000f
REPORT RequestId: 5d87f6db-3151-481f-97f4-92852478000f    Init Duration: 0.12 ms
Duration: 158.74 ms    Billed Duration: 159 ms    Memory Size: 128 MB    Max
Memory Used: 128 MB
{"statusCode": 200, "body": "{\"message\": \"hello OpenCV 5.0.0 in a Lambda\"}"}
```

Perfect! Our Lambda returned a message that correctly detects the version of OpenCV installed on the container.

We are now ready to deploy the function, and as mentioned earlier, we will make SAM do all the hard work by selecting the `--guided` option:

```
$ sam deploy --guided

Configuring SAM deploy
======================

    Looking for config file [samconfig.toml] :  Not found

    Setting default arguments for 'sam deploy'
    =========================================
    Stack Name [sam-app]: sam-opencv-lambda
    AWS Region [eu-west-1]:
    #Shows you resource changes to be deployed and requires a 'Y' to initiate
deploy
    Confirm changes before deploy [y/N]:
    #SAM needs permission to be able to create roles to connect to the resources
in your template
    Allow SAM CLI IAM role creation [Y/n]:
    #Preserves the state of previously provisioned resources when an operation
fails
    Disable rollback [y/N]:
    HelloWorldFunction may not have authorization defined. Is this okay? [y/N]: y
    Save arguments to configuration file [Y/n]: Y
    SAM configuration file [samconfig.toml]:
    SAM configuration environment [default]:

    Looking for resources needed for deployment:
     Managed S3 bucket: aws-sam-cli-managed-default-
samclisourcebucket-1nq23i7aviy5t
     A different default S3 bucket can be set in samconfig.toml
     Image repositories: Not found.
     #Managed repositories will be deleted when their functions are removed from
the template and deployed
     Create managed ECR repositories for all functions? [Y/n]: Y

    Saved arguments to config file
    Running 'sam deploy' for future deployments will use the parameters saved
above.
    The above parameters can be changed by modifying samconfig.toml
```

```
    Learn more about samconfig.toml syntax at
    https://docs.aws.amazon.com/serverless-application-model/latest/
developerguide/serverless-sam-cli-config.html
```

As you can see, it's just a series of questions regarding deployment preferences and options. You do have to answer y to the question about the lack of authorization. Without it (meaning "no, you are not ok with the Lambda not having an authorization defined"), the deployment process will be halted.

I highly recommend you let SAM create all IAM roles to execute the Lambda and the ECR repository needed to store your container images, unless you know what you are doing.

Once the resources are created, SAM will start "pushing" your container image into the ECR repository (it's just AWS's equivalent of Docker Hub), and when that's completed, it will deploy the container into a Lambda function. At that point, we are able to test our mini-application in production.

When SAM is done deploying, you will have a list of the changes that were applied to your AWS account:



Figure 12.1: AWS SAM output of the deployment operation

Now we want to test that the Lambda is still working correctly, even in production. You can do this in two ways: you can either hit the endpoint listed in the output of the deployment operation (that long URL in the first of the three output groups in the image), or you can go to your AWS web console, navigate to `Lambda`, select your newly deployed function, and click on

`Test`. You need to create a test event (you can just hit save on the default "hello-world" event), and then click **test**. The output will be exactly like the one you had locally:

```
$ curl -X GET "https://slrnph6l6j.execute-api.eu-west-1.amazonaws.com/Prod/hello/"
{"message": "hello OpenCV 5.0.0 in a Lambda"}
```

Note that the `slrnph6l6j` part of the URL will be different for you, as this is uniquely generated at deployment time.

Great, you now know how to create a container image that has all the necessary dependencies to run an OpenCV application on it. We will soon move to a slightly more complex example.

Is this really necessary?

You may be wondering if this trip down the container lane is really necessary. Are we complicating our lives unnecessarily, and adding moving parts to an already complex landscape?

The answer is yes, it is necessary. Or, to better quantify, it is necessary, but it's also good.

It's necessary because the `.zip` deployment method (thus working without container images) only prevents *you* (the developer) from dealing with containers, but the underlying Lambda function will simply be a container image with your code baked into it. If you do that, the base image used by Lambda will be a base Python image. This will not contain OpenCV, and as you have seen, you cannot simply install the OpenCV Python library for it to work; you need the actual underlying libraries to be present. Therefore, your only choice is to use custom-built containers.

The second part of the answer is that I wholeheartedly encourage you to develop using containers; they are fun, you can learn a lot about computers, and above all, they make your development life cycle extremely reliable. Containers make deploying OpenCV applications very feasible, and now you can leverage very powerful computing resources (those made available by AWS) to run your compute-intensive CV applications. The things you can do at an absolutely unfathomable scale with containers running OpenCV applications are simply limitless. Think real-time object detection while streaming, face recognition, or cataloging image metadata on GBs of incoming images.

So yes, it's necessary, and it's good for you.

In fact, let's see how we can use Lambda to analyze images and return coordinates for detected faces.

Example application: Face detection

Let's now extend our fairly useless mini-application into a function that does something. Let's take a run-of-the-mill task such as detecting faces in an image.

A short premise before proceeding: it's important that you understand that this is a web endpoint, so while you would normally draw rectangles on an image, in this case you want to just return the coordinates of the faces, so whatever client application calls your endpoint will use that information (and potentially draw rectangles on an image in the client).

As for the input image, we are not going to pass this as an actual image, but as a URL. So, the Lambda function will download the image at the provided URL, perform face detection, and return the coordinates.

So let's take a look at the code:

```
import json
import os

import cv2
import requests
from urllib.parse import urlparse

ALLOWED_SCHEMES = {"http", "https"}
MAX_IMAGE_BYTES = 10 * 1024 * 1024  # 10 MB
DOWNLOAD_TIMEOUT = 5  # seconds

# Resolve the cascade path relative to this file, not the process's
# working directory, so it loads correctly regardless of how/where
# the Lambda runtime invokes the handler.
CASCADE_PATH = os.path.join(os.path.dirname(__file__), "cascade.xml")
cascade = cv2.CascadeClassifier(CASCADE_PATH)
if cascade.empty():
    raise RuntimeError(f"Failed to load cascade classifier from {CASCADE_PATH}")


def _error(status_code, message):
    return {
        "statusCode": status_code,
        "body": json.dumps({"error": message}),
    }
```

```
def lambda_handler(event, context):
    image_url = (event.get('queryStringParameters') or {}).get('url')
    if not image_url:
        return _error(400, "missing required query string parameter 'url'")

    parsed = urlparse(image_url)
    if parsed.scheme not in ALLOWED_SCHEMES or not parsed.netloc:
        return _error(400, "url must be an http(s) URL")

    try:
        response = requests.get(image_url, timeout=DOWNLOAD_TIMEOUT, stream=True)
        response.raise_for_status()
    except requests.RequestException as exc:
        return _error(502, f"could not fetch image: {exc}")

    content_type = response.headers.get("Content-Type", "")
    if not content_type.startswith("image/"):
        return _error(400, "url did not return an image")

    content_length = int(response.headers.get("Content-Length", 0) or 0)
    if content_length > MAX_IMAGE_BYTES:
        return _error(413, "image exceeds maximum allowed size")

    tmp_image = "/tmp/image.jpg"
    bytes_written = 0
    with open(tmp_image, 'wb') as handler:
        for chunk in response.iter_content(chunk_size=65536):
            bytes_written += len(chunk)
            if bytes_written > MAX_IMAGE_BYTES:
                return _error(413, "image exceeds maximum allowed size")
            handler.write(chunk)

    img = cv2.imread(tmp_image)
    if img is None:
        return _error(400, "downloaded file is not a readable image")

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = cascade.detectMultiScale(gray, 1.1, 4)
    coords = [{"x": int(x), "y": int(y), "w": int(w), "h": int(h)}
              for (x, y, w, h) in faces]
```

```
    return {
        "statusCode": 200,
        "body": json.dumps({"coords": coords}),
    }
```

A few points of note:

1. We need the `requests` package to perform the URL download.
2. We need to include the `cascade` XML file and copy it into the container.
3. The URL is accessed through the `url` property of the `queryStringParameters` property of the `event` object. In practical terms, the URL will be passed into the function in the format `?url=<url of the image>`.
4. Lambda functions allow you to write data to the `/tmp` folder — 512 MB by default, configurable up to 10,240 MB (10 GB), and that's why we are using that particular location as the download location of the input image.

So let's start from the beginning and include the `cascade` XML file, which we have used multiple times in this book; you should simply place a copy of it in the hello-world folder.

After that, let's refresh the `requirements.txt` file by re-running `pipreqs` or pip freeze so that the `requests` package is included.

Because the Dockerfile already installs from `requirements.txt`, no Dockerfile edit is needed this time — it already lists the pinned dependencies:

```
opencv-contrib-python-headless==5.0.0.93
requests==2.32.3
```

And now we can re-run sam build `--use-container` to rebuild the image.

Now we can test the application. Again, we can either deploy and test, or, more advisable, first run a local test.

This time, however, we can't just run `sam local invoke` and hope it works, because the function expects a URL for the input image. So we need to create an event that conforms to the expected input. You can read how to create events at this address: `https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-cli-command-reference-sam-local-generate-event.html`.

However, to simplify your life, I created an event for you and saved it in the project's repository at `events/awsproxy.json`.

The notable part of the event's JSON is this block:

```
"queryStringParameters": {
    "url": "https://img.konami.com/kde_cms/eu_publish/uploads/
pes2021_sscnapoli_players-600x338.jpg"
  },
```

You can replace the actual `url` with whatever image containing faces from the internet you prefer.

Now we can run `sam local invoke -e events/awsproxy.json`, which specifies the event to use as an input.

Let's take a look at the output:

```
$ sam local invoke -e events/awsproxy.json
Invoking Container created from helloworldfunction:python3.13-v1
Building image.................
Using local image: helloworldfunction:rapid-x86_64.
START RequestId: 78fe1ede-c753-4ae2-8c83-c09dcc716147 Version: $LATEST
END RequestId: 78fe1ede-c753-4ae2-8c83-c09dcc716147
REPORT RequestId: 78fe1ede-c753-4ae2-8c83-c09dcc716147 Init Duration: 0.04 ms
Duration: 294.35 ms Billed Duration: 295 ms Memory Size: 128 MB Max Memory Used:
128 MB
{"statusCode": 200, "body": "{\"coords\": [{\"x\": 480, \"y\": 20, \"w\": 49,
\"h\": 49}, {\"x\": 267, \"y\": 28, \"w\": 53, \"h\": 53}, {\"x\": 46, \"y\": 15,
\"w\": 49, \"h\": 49}, {\"x\": 386, \"y\": 16, \"w\": 52, \"h\": 52}, {\"x\": 157,
\"y\": 21, \"w\": 53, \"h\": 53}]}"}
```

A successful run! Now we just need to deploy the function. If you remember, the first time you ran the guided deployment, SAM saved a deployment configuration in the file `samconfig.toml`.

Now we can run a deployment with the same parameters by simply specifying a configuration file, and avoid re-answering the interactive questions:

```
$ sam deploy --config-file samconfig.toml
```

This will update the Lambda, which you can reach at the same URL we tested above, but this time we need to remember to add the URL query string parameter to obtain the identical output as the local test.

A note on AWS Fargate

What you have learned so far in this chapter is entirely transferable to AWS Fargate; in fact, the process is simpler in that you don't need to install the Lambda RIC to turn your standard container image into a runnable workload on the cloud.

The reason why we concentrated on Lambda is that Lambdas are by far the most suitable tool to perform small and specific jobs, since they have a maximum execution time of 15 minutes. Fargate can launch long-running jobs and is more suited to large and complex workloads such as batch jobs. So if you are processing one image at a time (or a small number), Lambda is the way to go. If you are tackling processing of GBs of image or video data, then Fargate is more suited.

The tool previously recommended for deploying jobs on AWS Fargate was AWS Copilot, which reached end-of-support in February 2025. The current recommended alternatives are Amazon ECS Express Mode (for quick deployments from a single command) or AWS CDK with ECS L3 constructs (for infrastructure-as-code). You can read more at `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/getting-started.html`

Summary

In this chapter, we familiarized ourselves with the idea of containers and created OpenCV containers that can be deployed on the cloud; then we learned about AWS Lambda, which enables the deployment of OpenCV applications that can be automatically scaled, and finally, we showed how to develop a face detection web service that returns face location information about an input image.

You are now ready to take your OpenCV knowledge and build massively scalable applications with it!

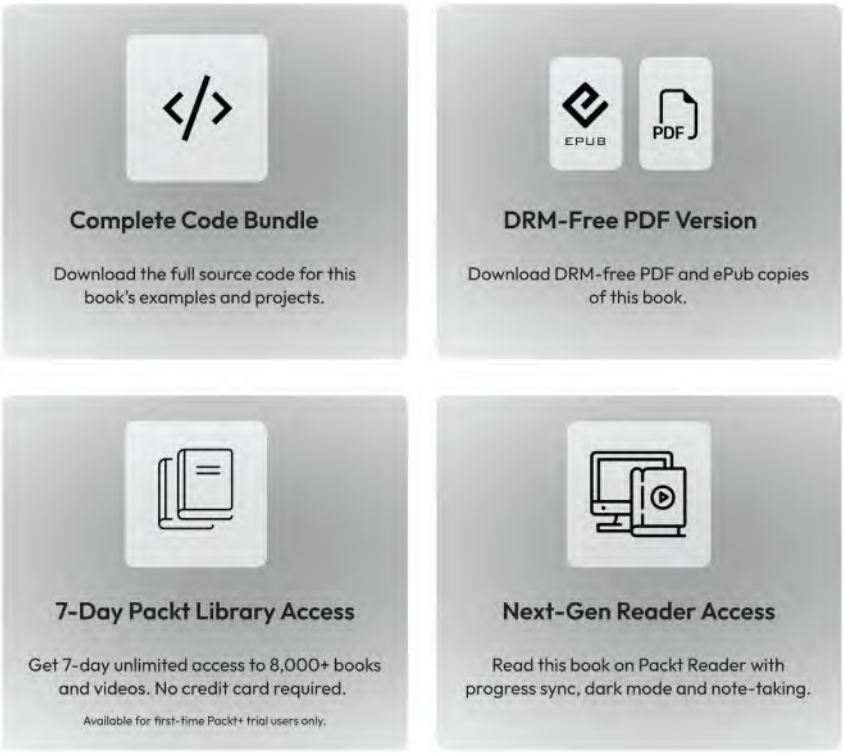
With that thought, and with good wishes, we shall part for now.

This book's authors hope that you have enjoyed our journey together through the Python bindings of OpenCV 5. Although covering all of OpenCV 5's functionality and all its bindings would take a series of books, we have explored a broad selection of fascinating and futuristic concepts, and we encourage you to get in touch with us and the OpenCV community to let us know about your next groundbreaking project in the field of computer vision!

13

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

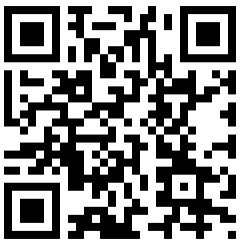
Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.

> **Note**
>
> **Note**: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.



On the page that opens (similar to *Figure 13.1* on desktop), search for this book by name and select the correct edition.

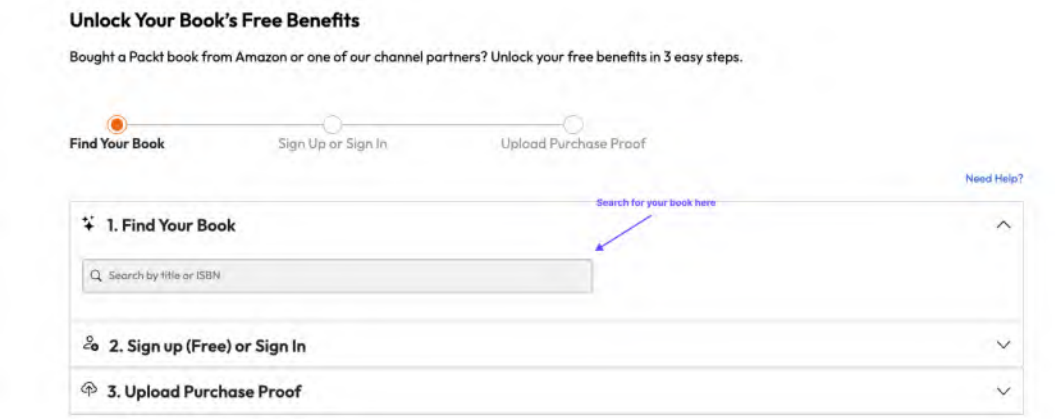


Figure 13.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.

Appendix A

Bending Color Space with the Curves Filter

Starting in *Chapter 3, Processing Images with OpenCV*, our Cameo demo application incorporates an image processing effect called curves, which it uses to emulate the color bias of certain photo films. This appendix describes the concept of curves and their implementation using SciPy.

Curves are a technique for remapping colors. With curves, a channel's value at a destination pixel is a function of (only) the same channel's value at the source pixel. Moreover, we do not define the functions directly; instead, for each function, we define a set of control points from which the function is interpolated. In pseudocode, for a BGR image:

```
dst.b = funcB(src.b) where funcB interpolates pointsB
dst.g = funcG(src.g) where funcG interpolates pointsG
dst.r = funcR(src.r) where funcR interpolates pointsR
```

The type of interpolation may vary between implementations, though it should avoid discontinuous slopes at control points and, instead, produce curves. We will use cubic spline interpolation whenever the number of control points is sufficient.

Let's start by taking a look at how interpolation can be implemented.

Formulating a curve

Our first step toward curve-based filters is to convert control points to a function. Most of this work is done for us by a SciPy function called `scipy.interp1d`, which takes two arrays (x and y coordinates) and returns a function that interpolates the points. As an optional argument to `scipy.interp1d`, we may specify the `kind` of interpolation; supported options include

`'linear'`, `'nearest'`, `'zero'`, `'slinear'` (spherical linear), `'quadratic'`, and `'cubic'`. Another optional argument, `bounds_error`, may be set to `False` to permit extrapolation as well as interpolation.

Let's edit the `utils.py` script that we use with our Cameo demo and add a function that wraps `interp1d` with a slightly simpler interface:

```
def createCurveFunc(points):
    """Return a function derived from control points."""
    if points is None:
        return None
    numPoints = len(points)
    if numPoints < 2:
        return None
    xs, ys = zip(*points)
    if numPoints < 3:
        kind = 'linear'
    elif numPoints < 4:
        kind = 'quadratic'
    else:
        kind = 'cubic'
    return scipy.interpolate.interp1d(xs, ys, kind,
                                      bounds_error = False)
```

Rather than two separate arrays of coordinates, our function takes an array of (x, y) pairs, which is probably a more readable way of specifying control points. The array must be ordered such that x increases from one index to the next. Typically, for natural-looking effects, the y values should always increase too, and the first and last control points should be (0, 0) and (255, 255) in order to preserve black and white. Note that we will treat x as a channel's input value and y as the corresponding output value. For example, (128, 160) would brighten a channel's midtones.

Note that cubic interpolation requires at least four control points. If there are only three control points, we fall back to quadratic interpolation, and if there are only two, we fall back to linear interpolation. For natural-looking effects, these fallback cases should be avoided.

Throughout the remainder of this appendix, we seek to use the curves generated by our `createCurveFunc` function in an efficient and well-organized way.

Caching and applying a curve

By now, we can get the function of a curve that interpolates arbitrary control points. However, this function might be expensive. We do not want to run it once per channel, per pixel (for example, 921,600 times per frame if applied to three channels of 640x480 video). Fortunately, we are typically dealing with just 256 possible input values (in 8 bits per channel), and we can cheaply precompute and store that many output values. Then, our per-channel, per-pixel cost is just a lookup of the cached output value.

Let's edit `utils.py` and add functions to create a lookup array for a given function:

```
def createLookupArray(func, length = 256):
    """Return a lookup for whole-number inputs to a function.

    The lookup values are clamped to [0, length - 1].

    """
    if func is None:
        return None
    lookupArray = numpy.empty(length)
    i = 0
    while i < length:
        func_i = func(i)
        lookupArray[i] = min(max(0, func_i), length - 1)
        i += 1
    return lookupArray
```

Let's also add a function that will apply a lookup array (such as the result of the preceding function) to another array (such as an image):

```
def applyLookupArray(lookupArray, src, dst):
    """Map a source to a destination using a lookup."""
    if lookupArray is None:
        return
    dst[:] = lookupArray[src]
```

Note that the approach in `createLookupArray` is limited to input values that are whole numbers (non-negative integers) since the input value is used as an index into an array. The `applyLookupArray` function works by using a source array's values as indices in the lookup array. Python's slice notation (`[:]`) is used to copy looked-up values into a destination array.

Let's consider another optimization. What if we want to apply two or more curves in succession? Performing multiple lookups is inefficient and may cause a loss in precision. We can avoid these problems by combining two curve functions into one function before creating a lookup array. Let's edit `utils.py` again and add the following function that returns a composite of two given functions:

```
def createCompositeFunc(func0, func1):
    """Return a composite of two functions."""
    if func0 is None:
        return func1
    if func1 is None:
        return func0
    return lambda x: func0(func1(x))
```

The approach in `createCompositeFunc` is limited to input functions that each take a single argument. The arguments must be of compatible types. Note the use of Python's `lambda` keyword to create an anonymous function.

The following is one final optimization issue. What if we want to apply the same curve to all channels of an image? In this case, splitting and remerging channels is wasteful because we do not need to distinguish between channels. We just need one-dimensional indexing, as used by `applyLookupArray`. For this, we can use the `numpy.ravel` function, which returns a one-dimensional interface to a preexisting, given array that may be multidimensional. Normally, `numpy.ravel` does not copy any data; it just returns a flattened view that refers to the same underlying data as the original array.

> **Note**
>
> NumPy arrays have a `flatten` method, but this returns a copy.

`numpy.ravel` works for images with any number of channels. Thus, it allows us to abstract the difference between grayscale and color images in cases when we wish to treat all channels the same.

Now that we have addressed several important optimization issues concerning the use of curves, let's consider how to organize our code to provide a simple and reusable interface to applications such as Cameo.

Designing object-oriented curve filters

We took a first look at object-oriented programming in *Chapter 2, Handling Files, Cameras, and GUIs*, when we began to implement the Cameo application. Now, as we design a set of curve filters, we will continue to follow object-oriented principles such as encapsulation of data and inheritance of common behaviors.

> **Note**
>
> Microsoft Learn offers tutorials on object-oriented programming in *More Python for Beginners* at `https://learn.microsoft.com/en-us/shows/more-python-for-beginners/`. If you are still relatively new to the topic or need a refresher, spend an hour studying this excellent tutorial.

Since we cache a lookup array for each curve, our curve-based filters have data associated with them. Thus, we will implement them as classes, not just functions. Let's make a pair of curve filter classes, along with corresponding higher-level classes that can apply any function, not just a curve function:

- `VFuncFilter`: This is a class that is instantiated with a function, which it can later apply to an image. For this purpose, it has an `apply` method. The function is applied to the V (value) channel of a grayscale image or to all channels of a color image.
- `VCurveFilter`: This is a subclass of `VFuncFilter`. Instead of being instantiated with a function, it is instantiated with a set of control points, which it uses internally to create a curve function.
- `BGRFuncFilter`: This is a class that is instantiated with up to four functions, which it can later apply to a BGR image. For this purpose, it has an `apply` method. One of the functions is applied to all the channels, while the other three functions are each applied to a single channel. The overall function is applied first, and then the per-channel functions.
- `BGRCurveFilter`: This is a subclass of `BGRFuncFilter`. Instead of being instantiated with four functions, it is instantiated with four sets of control points, which it uses internally to create curve functions.

Additionally, all these classes accept a constructor argument that is a numeric type, such as `numpy.uint8` for 8 bits per channel. This type is used to determine how many entries should be in the lookup array. The numeric type should be an integer type, and the lookup array will cover the range from 0 to the type's maximum value (inclusive).

First, let's look at the implementations of `VFuncFilter` and `VCurveFilter`, which can both be added to `filters.py`:

```
class VFuncFilter(object):
    """A filter that applies a function to V (or all of BGR)."""

    def __init__(self, vFunc=None, dtype=numpy.uint8):
        length = numpy.iinfo(dtype).max + 1
        self._vLookupArray = utils.createLookupArray(vFunc, length)

    def apply(self, src, dst):
        """Apply the filter with a BGR or gray source/destination."""
        srcFlatView = numpy.ravel(src)
        dstFlatView = numpy.ravel(dst)
        utils.applyLookupArray(self._vLookupArray, srcFlatView,
                               dstFlatView)

class VCurveFilter(VFuncFilter):
    """A filter that applies a curve to V (or all of BGR)."""

    def __init__(self, vPoints, dtype=numpy.uint8):
        VFuncFilter.__init__(self, utils.createCurveFunc(vPoints), dtype)
```

Here, we are internalizing the use of several of our previous functions: `utils.createCurveFunc`, `utils.createLookupArray()`, and `utils.applyLookupArray()`. We are also using `numpy.iinfo` to determine the relevant range of lookup values, based on the given numeric type.

Now, let's look at the implementations of `BGRFuncFilter` and `BGRCurveFilter`, which can both be added to `filters.py` as well:

```
class BGRFuncFilter(object):
    """A filter that applies different functions to each of BGR."""

    def __init__(self, vFunc=None, bFunc=None, gFunc=None,
                 rFunc=None, dtype=numpy.uint8):
        length = numpy.iinfo(dtype).max + 1
        self._bLookupArray = utils.createLookupArray(
            utils.createCompositeFunc(bFunc, vFunc), length)
        self._gLookupArray = utils.createLookupArray(
            utils.createCompositeFunc(gFunc, vFunc), length)
```

```
        self._rLookupArray = utils.createLookupArray(
            utils.createCompositeFunc(rFunc, vFunc), length)

    def apply(self, src, dst):
        """Apply the filter with a BGR source/destination."""
        b, g, r = cv2.split(src)
        utils.applyLookupArray(self._bLookupArray, b, b)
        utils.applyLookupArray(self._gLookupArray, g, g)
        utils.applyLookupArray(self._rLookupArray, r, r)
        cv2.merge([b, g, r], dst)

class BGRCurveFilter(BGRFuncFilter):
    """A filter that applies different curves to each of BGR."""

    def __init__(self, vPoints=None, bPoints=None,
                 gPoints=None, rPoints=None, dtype=numpy.uint8):
        BGRFuncFilter.__init__(self,
                               utils.createCurveFunc(vPoints),
                               utils.createCurveFunc(bPoints),
                               utils.createCurveFunc(gPoints),
                               utils.createCurveFunc(rPoints), dtype)
```

Again, we are internalizing the use of several of our previous functions: `utils.createCurveFunc`, `utils.createCompositeFunc`, `utils.createLookupArray`, and `utils.applyLookupArray`. We are also using `numpy.iinfo`, `cv2.split` and `cv2.merge`.

These four classes can be used as-is, with custom functions or control points being passed as arguments at instantiation. Alternatively, we can make further subclasses that hard-code certain functions or control points. Such subclasses could be instantiated without any arguments.

Now, let's look at some examples of subclasses.

Emulating photo films

A common use of curves is to emulate the palettes that were characteristic of pre-digital photography. Every type of photo film has its own unique rendition of color (or gray), but we can generalize about some of the differences from digital sensors. Film tends to suffer a loss of detail and saturation in shadows, whereas digital tends to suffer these failings in highlights. Also, film tends to have uneven saturation across different parts of the spectrum, so each film has certain colors that pop or jump out.

Thus, when we think of good-looking film photos, we might think of scenes (or renditions) that are bright and that have certain dominant colors. At the other extreme, maybe we remember the murky look of underexposed film that could not be improved much by the efforts of the lab technician.

We are going to create four different film-like filters using curves. They are inspired by three kinds of film and a processing technique:

- Kodak Portra, a family of films that are optimized for portraits and weddings.
- Fuji Provia, a family of general-purpose films.
- Fuji Velvia, a family of films that are optimized for landscapes.
- Cross-processing, a nonstandard film processing technique, sometimes used to produce a grungy look in fashion and band photography.

Each film emulation effect is a very simple subclass of `BGRCurveFilter`. We just need to override the constructor to specify a set of control points for each channel. The choice of control points is based on recommendations by photographer Petteri Sulonen, who (in the past) maintained a blog on such topics.

The Portra, Provia, and Velvia effects should produce normal-looking images. These effects should not be obvious except in before-and-after comparisons.

Let's examine the implementation of each of the four film emulation filters, starting with the Portra filter.

Emulating Kodak Portra

Portra has a broad highlight range that tends toward warm (amber) colors, while shadows are cooler (more blue). As portrait film, it tends to make people's complexions fairer. Also, it exaggerates certain common clothing colors, such as milky white (for example, a wedding dress) and dark blue (for example, a suit or jeans). Let's add this implementation of a Portra filter to `filters.py`:

```
class BGRPortraCurveFilter(BGRCurveFilter):
    """A filter that applies Portra-like curves to BGR."""

    def __init__(self, dtype=numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            vPoints=[(0,0),(23,20),(157,173),(255,255)],
            bPoints=[(0,0),(41,46),(231,228),(255,255)],
            gPoints=[(0,0),(52,47),(189,196),(255,255)],
            rPoints=[(0,0),(69,69),(213,218),(255,255)],
```

```
            dtype=dtype
        )
```

Moving from Kodak to Fuji, we will emulate Provia next.

Emulating Fuji Provia

Provia has strong contrast and is slightly cool (blue) throughout most tones. Sky, water, and shade are enhanced more than sunlight. Let's add this implementation of a Provia filter to `filters.py`:

```
class BGRProviaCurveFilter(BGRCurveFilter):
    """A filter that applies Provia-like curves to BGR."""

    def __init__(self, dtype=numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            bPoints=[(0,0),(35,25),(205,227),(255,255)],
            gPoints=[(0,0),(27,21),(196,207),(255,255)],
            rPoints=[(0,0),(59,54),(202,210),(255,255)],
            dtype=dtype
        )
```

Next up is our Fuji Velvia filter.

Emulating Fuji Velvia

Velvia has deep shadows and vivid colors. It can often produce azure skies in the daytime and crimson clouds at sunset. The effect is difficult to emulate but here is a good attempt that we can add to `filters.py`:

```
class BGRVelviaCurveFilter(BGRCurveFilter):
    """A filter that applies Velvia-like curves to BGR."""
    def __init__(self, dtype=numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            vPoints=[(0,0),(128,118),(221,215),(255,255)],
            bPoints=[(0,0),(25,21),(122,153),(165,206),(255,255)],
            gPoints=[(0,0),(25,21),(95,102),(181,208),(255,255)],
            rPoints=[(0,0),(41,28),(183,209),(255,255)],
            dtype=dtype
        )
```

Now, let's go for the cross-processed look!

Emulating cross-processing

Cross-processing produces a strong blue or greenish-blue tint in shadows and a strong yellow or greenish-yellow tint in highlights. Black and white are not necessarily preserved. Also, the contrast is very high. Cross-processed photos take on a sickly appearance. People look jaundiced, while inanimate objects look stained. Let's edit `filters.py` to add the following implementation of a cross-processing filter:

```
class BGRCrossProcessCurveFilter(BGRCurveFilter):
    """A filter that applies cross-process-like curves to BGR."""

    def __init__(self, dtype=numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            bPoints=[(0,20),(255,235)],
            gPoints=[(0,0),(56,39),(208,226),(255,255)],
            rPoints=[(0,0),(56,22),(211,255),(255,255)],
            dtype=dtype
        )
```

Now that we have looked at a few examples of how to implement film emulation filters, we'll wrap up this appendix so that you can return to the main implementation of the Cameo application in *Chapter 3, Processing Images with OpenCV*.

Summary

Building on the `scipy.interp1d` function, we have implemented a collection of curve filters that are efficient (due to the use of lookup arrays) and easily extensible (due to an object-oriented design). Our work has included special-purpose curve filters that can make digital images look more like film shots. These filters can be readily integrated into an application such as Cameo, as demonstrated by the use of our Portra film emulation filter in *Chapter 3, Processing Images with OpenCV*.



packtpub.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

At www.packtpub.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.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

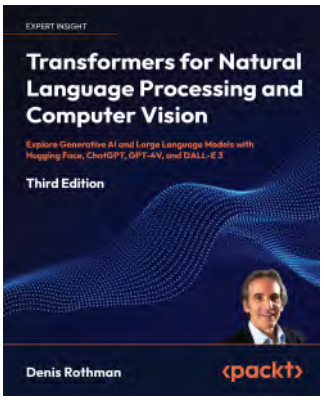


Hands-On Image Processing and Computer Vision with Python, Second Edition

Sandipan Dey

ISBN: 9781837636235

- Build image processing and computer vision pipelines
- Apply image enhancement, restoration, and segmentation
- Implement image classification and object detection models
- Explore CNNs, Vision Transformers, and attention models
- Generate and edit images using GANs and diffusion models
- Develop multimodal vision-language AI applications
- Apply visual AI across diverse real-world domains
- Implement super-resolution, style transfer, and image-to-image translation



Transformers for Natural Language Processing and Computer Vision, Third Edition

Denis Rothman

ISBN: 9781805128724

- Breakdown and understand the architectures of the Transformer, BERT, GPT, T5, PaLM, ViT, CLIP, and DALL-E
- Fine-tune BERT, GPT, and PaLM models
- Learn about different tokenizers and the best practices for preprocessing language data
- Pretrain a RoBERTa model from scratch
- Implement retrieval augmented generation and rules bases to mitigate hallucinations
- Visualize transformer model activity for deeper insights using BertViz, LIME, and SHAP
- Go in-depth into vision transformers with CLIP, DALL-E, and GPT

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.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.

Share your thoughts

Now you've finished *Learning OpenCV 5 Computer Vision with Python, Fourth Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



`https://packt.link/r/1803230223`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

10-bit images
converting, to 8-bit images 107 – 109
12-factor app **455**
reference link 455
3D image tracking **304 – 307**
3D tracking **304**
4x4 rigid transformation matrix **359**
6 degrees of freedom (6DOF) **304, 357**

A

AWS Fargate **481**
AWS Lambda **453, 454, 464**
AWS SAM **464 – 475**
example 476
AWS Serverless **464**
AWS Lambda 464
AWS SAM 465 – 476
AWS cloud provider **453**
Arch Linux
OpenCV, installing on 18
activation function **395**
additive models **69**
application's loop
modifying 164 – 166
array slicing **36**
artificial neural networks (ANNs) **391 – 394**
classifier, training in multiple epochs 401 – 406
handwritten digits, recognizing 406
main module, implementing 414 – 421
minimal test module, implementing 414
module, implementing 409 – 413
potential applications, finding 423
training improvement 421, 422
training, in OpenCV 398 – 401
augmented reality (AR) **308 – 310**

B

BIRCH **384**
BIRCH segmentation **385 – 387**
BRIEF algorithm
ORB, using with 184
Binary Robust Independent Elementary Features (BRIEF) **172**
background subtraction **252**
Godbehere-Matsukawa-Goldberg (GMG), using 264 – 266
implementing 254 – 257
KNN background subtractor, using 262, 263
MOG background subtractor, using 257 – 261
moving objects, detecting 252 – 254
bag of visual words (BoVW) **224**
bag-of-words (BoW) **212, 224, 225**
applying, to computer vision 225, 226
barrel distortion **314**
blob **173, 424**
blue-green-red (BGR) **29, 68, 269**
bounding box **83 – 85**
brute-force matcher
ORB, using with 184, 185

C

Canny
for edge detection 80, 81
Clojure **286**

Continuously Adaptive MeanShift (CamShift) **277**
colorful objects, tracking 267, 268
using 277, 278
Continuously AdaptiveMeanShift (CamShift) **251**
camera parameter **311 – 315**
car detector
custom HOG detector, training 243 – 249
training 227 – 235
cascade **145**
channels **68**
circles
detecting 94 – 99
clustering **226**
codebook **224**
color histogram **267**
color models
images, converting between 68
colorful objects
tracking, with CamShift 267, 268
tracking, with MeanShift 267, 268
containers **454**
contour detection **82, 83**
example 84 – 87
convex contours **87 – 89**
convolution matrix **76**
convolutions
obtaining 76 – 78
copy operation
masking 167 – 169
corners **173**
cross-processing
emulating 496
curve-based filters
applying 489, 490
caching 489, 490
formulating 487, 488
custom HOG detector
training 243 – 249
custom kernels **76 – 78**
cv2.KeyPoint class
anatomy 179, 180
cv2.calcBackProject parameters **273**
cv2.calcHist parameters **272, 273**
cv2.solvePnPRansac **316 – 318**

D

DBSCAN **384**
DBSCAN segmentation **387 – 389**
Deep Learning Deployment Toolkit (DLDT) **424**
Difference of Gaussian (DoG) **177, 230**
DoG features
detecting 176 – 179
keypoint, anatomy 179, 180
Docker **453, 454**
reference link 456
Docker Engine
reference link 456
Docker basics **456**
Docker pull 458
Docker run 457, 458
Docker version 456
Dockerfile 459 – 463
Docker images
Linux distributions 463
Docker pull **458**
Docker run **457, 458**
Docker version **456**
Dockerfile **459 – 463**
Douglas-Peucker algorithm **88**
damped least squares (DLS) **318**
dead reckoning **360**
deep learning model **445, 446**
deep neural networks (DNNs) **391**
using, from frameworks in OpenCV 424
demo application
2D-to-3D spatial conversions, performing 321 – 323
3D tracking algorithm, improving 354, 355

grayscale conversion, performing 320, 321
implementing 319
modules, importing 319
running 349 – 353
testing 349 – 353

demo application class
image, tracking in 3D 334 – 338
implementing 323
Kalman filter, applying 338 – 343
Kalman filter, initializing 338 – 343
main loop, implementing 333
Rodrigues and Euler vectors, converting between 344 – 346
track results, drawing 346 – 349
tracked object, masking 346 – 349
tracker, initializing 324 – 333

depth estimation
with normal camera 115 – 122

depth map **104**

discrete Fourier transform (DFT) **70**

discriminant **395**

disparity map **104**
mask, creating 109, 110

E

Edge Drawing (ED) **90**

Eigenfaces
used, for performing face recognition 158 – 160

Elastic Container Service (ECS) **454**

Elastic Kubernetes Service (EKS) **454**

edge detection **74, 75**
with Canny 80, 81

edges **173**

epipolar geometry **115**

error **394**

F

FAST algorithm
ORB, using with 183

FLANN-based matches
used, for finding homography 199 – 203

FaceTracker **165**

Fargate **454**

Fast Fourier Transform (FFT) **7, 70**

Fast Hessian algorithm
features, detecting 180 – 182

Fast Library for Approximate Nearest Neighbors (FLANN) **229**
matching with 194 – 199

Features from Accelerated Segment Test (FAST) **172**

Fisherfaces
used, for performing face recognition 160

Fourier transform **69, 70**

Fuji Provia
emulating 495

Fuji Velvia
emulating 495

face detection **477 – 480**
performing, with OpenCV 147

face detection, with OpenCV
performing, on still image 147 – 149
performing, on video 149 – 153

face recognition
algorithm, selecting 155, 156
data, gathering for 153 – 155
data, loading and training for 156, 157
performing 153
performing, with Eigenfaces 158 – 160
performing, with Fisherfaces 160
performing, with LBPH 161
results based, discarding on confidence score 161, 162

face tracking **53**

faces
classifying, with third-party DNNs 429 – 433
detecting, with third-party DNNs 430 – 433

faces, in infrared
application's loop, modifying 164 – 166
copy operation, masking 167 – 169
swapping 162, 163
features 145, 173
detection and matching, types 172
field of view (FOV) 311, 361
focal distance 311
foreground detection
with GrabCut algorithm 128 – 133
frames
capturing, from OpenNI-compatible depth camera 104 – 107
frames per second (FPS) 324
functional programming (FP) 252

G

Gaussian Mixture Model (GMM) 129
General Matrix Multiply (GEMM) 7, 372
Godbehere-Matsukawa-Goldberg (GMG) 251
using 264 – 266
GrabCut 128
GrabCut algorithm
foreground detection 128 – 133
gimbal lock 307
grayscale 68

H

Haar cascades
conceptualizing 144 – 146
data, obtaining 146
Harris 172
Harris corner detection algorithm 173 – 176
Haskell 286
HfsSegment class and parameters
reference link 138
Hierarchical Feature Selection (HFS) 102, 136
image segmentation 136 – 138
Homebrew 12
using, with custom packages 14, 15
using, with ready-made packages 13, 14
hand gesture recognition 437
classifier, building with Jupyter Notebooks 443 – 445
classifier, training with Jupyter Notebooks 443 – 445
labeled training data, generating 438 – 443
model, testing on camera input 447 – 451
hand recognition 435, 436
handwritten digits
MNIST database 407, 408
recognizing, in artificial neural networks (ANNs) 406
hard negative mining 248
headless system 45
hidden layer 396
size 397, 398
high-pass filters (HPFs) 70 – 73
histogram back-projection 269
calculating 269 – 272
cv2.calcBackProject parameters 273
cv2.calcHist parameters 272, 273
histogram of oriented gradients (HOG) 211, 212
people detecting 218 – 223
using, to describe image regions 215, 216
visualizing 213 – 215
homography
finding, with FLANN-based matches 199 – 203
hue-saturation-value (HSV) 29, 68, 269

I

I/O scripts 28
camera frames, capturing 42 – 44
camera frames, displaying in window 51 – 53

image and raw bytes, converting between 33 – 35
image data, accessing with numpy.ndarray 35 – 38
image file, reading 28 – 32
image file, writing 28 – 32
image, displaying in window 50
network video stream, sending 45 – 50
numpy.ndarray and cv2.UMat, converting between 38, 39
video file, reading 39, 40
video file, writing 39, 40

IEEE subscribers
reference link 268

IP camera **45**

Infinitesimal Plane-Based Pose Estimation (IPPE) **318**

Iterative Closest Point (ICP) **360**

image descriptors **171**
saving, to file 203 – 205
scanning, for matches 205 – 208

image manipulation **53**

image pyramid **145**

image segmentation
with Hierarchical Feature Selection (HFS) 136 – 138
with Watershed algorithm 133 – 136

image sensor **311**

images
converting, between color models 68

infrared (IR) **105**

input layer **396**
size 397

K

K-nearest neighbors
used, for filtering matches 189 – 193

KNN background subtractor
using 262, 263

Kalman filter **377**
motion trends, finding 279, 280
mouse cursor, tracking 281 – 284
predict and update phases 280

Keras
reference link 434

Kernelized Correlation Filter (KCF) algorithm **299**

Kodak Portra
emulating 494, 495

k-means clustering **226**

k-nearest neighbors (KNN) **251**

kernel **70**

kernel density tree (kd-tree) **197**

L

Levenberg-Marquardt (LM) **318**

Linux distributions
in Docker images 463

lens parameter **311 – 315**

lines
detecting 91 – 94

local binary pattern histogram (LBPH) **213**

locality-sensitive hashing (LSH) **332**

low-pass filters (LPFs) **72, 73**

M

MNIST database
training parameters, selecting 408

MOG background subtractor
using 258 – 261

MeanShift **251**
colorful objects, tracking 267, 268
example, implementing 274 – 277
planning 268, 269

MediaPipe **434**
OpenCV, using to classify gestures and actions 433, 434
reference link 434

Meta Language (ML) **286**

Microsoft Media Foundation (MSMF) **44**

Mixture of Gaussians (MOG) **251**
Modified National Institute of Standards and Technology (MNIST) **407**
Motion JPEG (MJPEG) **46**
Multiple Instance Learning Tracker (MILTrack) **252**
macOS
OpenCV, installing on 12
magnitude spectrum **70**
mask
creating, from disparity map 109, 110
minimum area rectangle **83**
minimum enclosing circle **83 – 85**
minimum enclosing rectangle **85**
modules
creating 74, 103
motion trends
finding, with Kalman filter 279, 280
moving objects
detecting, with background subtraction 252 – 254
multi-layer perceptron (MLP) **392**
mustache distortion **315**

N

name mangling **57**
near-infrared (NIR) **105, 162**
neural networks **391**
layers 396
neurons **395**
node **123**
non-maximum suppression (NMS) **81, 211, 216, 217**
normal vector **379**

O

OAK-D camera
stereo imaging 122 – 128
Open Model Zoo **424**
Open Source Computer Vision (OpenCV) **3**
building, from source 16 – 18
documentation 23
features 5
help 24
optimizing, for specific hardware 6, 7
reference link 24
updates 24
Open Source ComputerVision (OpenCV)
using, to perform face detection 147
using, with MediaPipe to classify gestures and actions 433, 434
using, with TensorFlow to classify gestures and actions 433, 434
OpenCV 5
highlights 6
OpenCV Universal Matrix **38**
OpenCV installation
building, from source 19 – 21
on Arch Linux 18, 19
on Debian 15
on Linux Mint 15
on Manjaro 18, 19
on Ubuntu 15
on Unix-like systems 22
ready-made OpenCV package, using 16, 19
OpenCV installation, on Windows **8**
building, from source 9 – 12
ready-made OpenCV package, using 8
OpenCV installation, on macOS **12**
Homebrew, using with custom packages 14
Homebrew, using with ready-made packages 13, 14
OpenNI-compatible cameras
application, modifying 110 – 114
OpenNI-compatible depth camera
frames, capturing 104 – 107
Orbbec

reference link 122
Oriented FAST and Rotated BRIEF (ORB) 172, 212
Oriented FAST andRotated BRIEF (ORB)
used, for matching logo in two images 185 – 189
using, with BRIEF algorithm 184
using, with brute-force matcher 184, 185
using, with FAST algorithm 183
object detector
creating 223
training 223
object-oriented curve filters
designing 491 – 493
object-oriented design 54
applying 62 – 64
video stream, abstracting with managers.CaptureManager 55 – 60
window and keyboard, abstracting with managers.WindowManager 61, 62
object-oriented programming (OOP) 252
objects
classifying, with third-party DNNs 425 – 429
detecting, with third-party DNNs 425 – 429
odometry 357
official sample code
running 22, 23
optical character recognition (OCR) 423
output layer 396
size 397
overfitting 398

P

Pattern Histograms (LBPHs) algorithm
used, for performing face recognition 161
Perspective-n-Point (PnP) 316, 359
Point-to-Plane Distance (PPD) 360
Point-to-Plane Error 360
Polygon File Format 379
Python
reference link 239
Python documentation
reference link 7
pedestrians
application flow, planning 285
class, implementing 287 – 290
improvements 295
main function, implementing 291 – 294
object-oriented and functional paradigms, comparing 285 – 287
tracker, modifying to use MIL 295 – 299
tracking 284
perceptron 395
photo films
cross-processing, emulating 496
emulating 493, 494
Fuji Provia, emulating 495
Fuji Velvia, emulating 495
Kodak Portra, emulating 494, 495
pincushion distortion 314
pixel-equivalent units 313
point cloud 357, 378 – 380
loading 380 – 383
segmentation, with BIRCH 385 – 387
segmentation, with DBSCAN 387 – 389
segmenting, into multiple objects 383, 384
showing 380 – 383
point cloud map 104

Q

quaternion 307

R

RealSense
reference link 122

Rodrigues rotation vector **305**
Runtime Interface Client (RIC) **471**
Runtime Interface Emulator (RIE) **471**
radial distortion **314**
ratio test **189 – 193**
red-green-blue (RGB) **30, 68**
region of interest (ROI) **37, 82, 167, 240, 275**
reinforcement learning **394**
reprojection error **316**
ridge **173**

S

SIFT descriptors
extracting 176 – 179
keypoint, anatomy 179, 180
Scale-Invariant Feature Transform (SIFT) **172, 212**
Serverless Application Model (SAM) **465**
Software as a Service (SaaS) **455**
Speeded Up Robust Features (SURF) **172, 212**
used, for descriptor extraction 180 – 182
Stanford File Format **379**
scale-invariant **145**
scikit-learn **404**
semiglobal block matching (SGBM) **101**
setup tools
selecting 7
shapes
detecting 99
singularities **307**
statistical model **393**
stereo imaging **101**
with OAK-D camera 122 – 128
stereo vision **115**
structure from motion (SfM) **101, 115**
subtractive color model **69**
supervised learning **393**
support vector machines (SVMs) **211, 217, 218, 295**
car detection 236 – 242
combining, with sliding window 235, 236
trained model, loading 243
trained model, saving 243

T

TensorFlow **434**
OpenCV, using to classify gestures and actions 433, 434
TensorFlow Lite **444**
Threading Building Blocks (TBB) **6**
tangential distortion **315**
tattoo forensics **203**
image descriptors, saving to file 203 – 205
scanning, for matches 205 – 208
tensor **424**
third-party DNNs
faces, classifying 429 – 433
faces, detecting 429 – 433
objects, classifying 425 – 429
objects, detecting 425 – 429

U

unsupervised learning **393**

V

Variante Ascari **177**
Virtual Machines (VMs) **454**
Vision Processing Unit (VPU) **122**
Visual Bag of Words (VBoW) **384**
valid depth mask **104**
vector **145**
video file
hardware acceleration, enabling 40, 41
visual odometry **357 – 360**
limitations 360
visual odometry application, for OAK-D

building 361
camera matrix, defining 361 – 365
depth frames, aligning to match RGB frames 365 – 368
extending 377, 378
frames, visualizing 373 – 377
odometry frames, preparing 368 – 371
odometry parameters, defining 361 – 365
odometry result, accumulating 371, 372
odometry results, visualizing 373 – 377
odometry, computing 368 – 371

W

Watershed algorithm
image segmentation 133 – 136
Wavefront OBJ format **379**
Windows
OpenCV, installing on 8
waveforms **69**
window size **145**

Y

yaw-pitch-roll (YPR) **306**

Z

zoom lens **312**