

# Hands-On Algorithms for Computer Vision

Learn how to use the best and most practical computer vision algorithms using OpenCV



By Amin Ahmadi Tazehkandi

Packt

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

# **Hands-On Algorithms for Computer Vision**

Learn how to use the best and most practical computer vision algorithms using OpenCV

Amin Ahmadi Tazehkandi

Packt

**BIRMINGHAM - MUMBAI**

# Hands-On Algorithms for Computer Vision

Copyright © 2018 Packt Publishing

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

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

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

**Commissioning Editor:** Richa Tripathi

**Acquisition Editor:** Sandeep Mishra

**Content Development Editor:** Tiksha Sarang

**Technical Editor:** Adhithya Haridas

**Copy Editor:** Safis Editing

**Project Coordinator:** Prajakta Naik

**Proofreader:** Safis Editing

**Indexer:** Mariammal Chettiar

**Graphics:** Jisha Chirayil

**Production Coordinator:** Deepika Naik

First published: July 2018

Production reference: 1250718

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78913-094-2

[www.packtpub.com](http://www.packtpub.com)

*This book is dedicated to Maman and Baba...*

*– Amin Ahmadi Tazehkandi, Summer of 2018, Vienna*



[mapt.io](https://mapt.io)

Mapt is an online digital library that gives you full access to over 5,000 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
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

# PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

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

# **Contributors**

# About the author

**Amin Ahmadi Tazehkandi** is an Iranian author, developer, and a computer vision expert. He completed his computer software engineering studies in Iran and has worked for numerous software and industrial companies around the world.

*I would like to thank my wife, Senem, who is a symbol of passion, love, and strength for me. I would also like to thank my family, a big pack of brilliant engineers born to an Iranian inventor father and a loving mother, for all of their unconditional love and support.*

# About the reviewer

**Zhuo Qingliang** (KDr2 online) is presently working at *paodingai*, a start-up fintech company that dedicated to improving the financial industry by using artificial intelligence technologies. He has over 10 years of experience in Linux, C, C++, Java, Python, and Perl development. He is interested in programming, doing consulting work, and contributing to the open source community.

He maintains a personal website, *KDr2*, where you can find out more about him.

# Packt is searching for authors like you

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

# Preface

We live in quite an exciting time. Every single day you can find a completely new application or digital device that can perform a certain task for you, entertain you, connect you to your family and friends, record your memories in the form of pictures or videos, and more. The list could probably go on forever. Most of these applications or devices—whether it is a small smartphone in your pocket, a tiny smartwatch on your wrist, or your connected smart car—exist thanks to the huge developments in producing cheaper and faster processors that have fantastic frameworks and libraries, which in turn house algorithms and technologies that efficiently perform tasks that were once only expected from humans with their natural sensors, such as their eyes and ears.

Computer vision is a field in computer science that has been revolutionized by the latest advancements in hardware and software technologies, and in response, it has influenced those hardware and software technologies just as much, if not more. Nowadays, computer vision is used for performing tasks as simple as taking photos using your digital camera and as complex as understanding the environment of a self-driving car so that you avoid accidents. It is used for turning your face into a rabbit's in a video recording and also for detecting cancerous cells and tissues in microscopic images that can save lives. The sky's the limit when it comes to what's possible for an application or digital device based on computer vision, especially considering the huge advancements in libraries and frameworks that businesses and developers can use to bring their ideas to life.

Over the past years, OpenCV, or the Open Source Computer Vision library, has turned into a complete set of tools for developers involved with computer vision. It contains almost everything that is needed for implementing most computer vision problems that you can think of, from the most basic operations that are expected from a proper computer vision library, such as resizing and filtering images, up to training models with machine learning that can be used for accurate and fast object detection purposes. The OpenCV library contains almost everything you need to develop computer vision applications, and it also supports some of the most popular programming languages, such as C++ and

Python. You can even find OpenCV bindings for the .NET Framework among others. OpenCV can run on almost any major operating system that you can think of, whether on mobile or desktop platforms.

The goal of this book is to teach the development of computer vision applications with the OpenCV library to developers by using a collection of hands-on examples and sample projects. Each chapter of this book includes a number of computer vision algorithms that correspond to the topics covered in that specific chapter, and by going through all the chapters in order, you will learn about a wide range of computer vision algorithms that can be used in your applications. Most of the computer vision algorithms covered in this book are independent of each other; however, it is highly recommended that you start from the beginning and try to build up your computer vision knowledge according to the order of the chapters. There's a reason why it's called a Hands-On book, so make sure you try each example presented in this book—all of them are fun and exciting enough to keep you going and build up your confidence as you proceed.

This book is the result of months of hard work, and it would have not been possible without the invaluable help of Tiksha Sarang, for her patience and amazing editing; Adhithya Haridas, for his precise and insightful technical reviews and comments; Sandeep Mishra, for this wonderful opportunity; the very helpful Technical Reviewer, Mr. Zhuo Qingliang; and everyone else at Packt Publishing who helped me create and deliver this book as it is and serve the open source and computer vision community.

# **Who this book is for**

Any developer with a solid understanding of the C++ programming language and knowledge about the usage of third-party libraries in the operating system of their choice will find the topics of this book very easy to follow and work with. On the other hand, developers familiar with the Python programming language can also use this book to learn about the usage of OpenCV library; however, they'll need to adapt the examples from C++ to Python by themselves, as this book's main focus will be on the C++ versions of the presented algorithms.

# What this book covers

[Chapter 1](#), *Introduction to Computer Vision*, lays out the basics of the computer vision science—what it is; where it is used; the definition of an image and its basic properties, such as pixels, depth, and channels; and so on. This short chapter is a totally introductory chapter meant for people who are completely new to the computer vision world.

[Chapter 2](#), *Getting Started with OpenCV*, introduces the OpenCV library and detail its core by going through the most important building blocks of OpenCV development. You'll also be presented with information about where to get it and how to use it. This chapter will even briefly go through the usage of CMake and how to create and build OpenCV projects, after which you'll learn about the Mat class and its variants, reading and writing images and videos, and accessing cameras (among other input source types).

[Chapter 3](#), *Array and Matrix Operations*, covers the fundamental algorithms that are used to create or alter matrices. In this chapter, you'll learn how to perform matrix operations, such as the cross product, the dot product, and inversion. This chapter will introduce you to many of the so-called per-element matrix operations, along with mathematical operations such as mean, sum, and Fourier transformation.

[Chapter 4](#), *Drawing, Filtering, and Transformation*, covers the wide category of image-processing algorithms as much as possible within the scope of this book. This chapter will teach you how to draw shapes and text on images. You'll learn how to draw lines, arrows, rectangles, and more. This chapter will also present you with a wide range of algorithms that are used for image-filtering operations, such as smoothing filters, dilation, erosion, and morphological operations on images. By the end of this chapter, you'll be familiar with the powerful remapping algorithm and the usage of color maps in computer vision.

[Chapter 5](#), *Back-Projection and Histograms*, introduces the concept of histograms and teach you how they are calculated from single- and multi-channel images.

You'll learn about the visualization of histograms for grayscale and color images, or, in other words, histograms calculated from the hue values of pixels. In this chapter, you'll also learn about back-projection images; that is, the reverse operation of histogram extraction. Histogram comparison and equalization are also among the topics covered in this chapter.

[Chapter 6](#), *Video Analysis – Motion Detection and Tracking*, explains how to process videos, especially for operations such as real-time object detection and tracking, using some of the most popular tracking algorithms in computer vision. After a brief introduction to how to process videos in general, you'll learn about the Mean Shift and CAM Shift algorithms, along with Kalman filtering, using real-world examples and object-tracking scenarios. By the end of this chapter, you'll also have learned about background- and foreground-extraction algorithms and how they are used in practice.

[Chapter 7](#), *Object Detection – Features and Descriptors*, starts with a brief introduction to object detection using template matching, and then moves on to teach you about a wide range of algorithms, that can be used for shape analysis. The topics covered in this chapter also include chain of keypoint detection, descriptor extraction, and descriptor matching, which are used for object detection based on features instead of simple pixel colors or intensity values.

[Chapter 8](#), *Machine Learning in Computer Vision*, covers the Machine Learning (ML) and Deep Neural Network (DNN) modules of OpenCV and some its most important algorithms, classes, and functions. Starting with the SVM algorithm, you'll learn how to train a model based on groups of similar training and then use that model to classify input data. You'll learn how to use HOG descriptors with SVM to classify images. This chapter also covers the implementation of artificial neural networks in OpenCV, and then moves on to teaches you about cascade classification. The last section of this chapter will teach you how to use pretrained models from third-party libraries, such as TensorFlow, to detect multiple objects of different type in real-time.

# To get the most out of this book

Although every required tool and piece of software for each chapter is mentioned in the initial section of each chapter, the following is a list that can be used as a simple and quick reference:

- A regular computer with a recent version of a Windows, macOS, or Linux (such as Ubuntu) operating system installed on it
- Microsoft Visual Studio (on Windows)
- Xcode (on macOS)
- CMake
- OpenCV

Firstly, to get an idea of what a regular computer is these days, you can search online or ask a local shop; however, the one you already have is most probably enough to get you started.

Also, your choice of Integrated Development Environment (IDE) or the build system you use (in this case, CMake) is pretty much irrelevant to the examples provided in this book. For instance, you can use the exact same examples in this book with any code editor and or build system, as long as you are familiar with the configuration of the OpenCV library for that IDE and build system.

# Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at [www.packtpub.com](http://www.packtpub.com).
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision>. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

# Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://www.packtpub.com/sites/default/files/downloads/HandsOnAlgorithmsforComputerVision\\_ColorImages.pdf](https://www.packtpub.com/sites/default/files/downloads/HandsOnAlgorithmsforComputerVision_ColorImages.pdf).

# 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. Here is an example: "On the other hand, to rotate an image, you can use the `rotate` function."

A block of code is set as follows:

```
| HistCompMethods method = HISTCMP_CORREL;  
| double result = compareHist(histogram1, histogram2, method);
```

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

```
| pip install opencv-python
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Before we start with the shape-analysis and feature-analysis algorithms, we are going to learn about an easy-to-use, extremely powerful method of object detection called **template matching**."



*Warnings or important notes appear like this.*



*Tips and tricks appear like this.*

# Get in touch

Feedback from our readers is always welcome.

**General feedback:** Email [feedback@packtpub.com](mailto:feedback@packtpub.com) and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at [questions@packtpub.com](mailto:questions@packtpub.com).

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

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

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

# Reviews

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

For more information about Packt, please visit [packtpub.com](http://packtpub.com).

# **Table of Contents**

[Title Page](#)

[Copyright and Credits](#)

[Hands-On Algorithms for Computer Vision](#)

[Dedication](#)

[Packt Upsell](#)

[Why subscribe?](#)

[PacktPub.com](#)

[Contributors](#)

[About the author](#)

[About the reviewer](#)

[Packt is searching for authors like you](#)

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Download the example code files](#)

[Download the color images](#)

[Conventions used](#)

[Get in touch](#)

[Reviews](#)

## [1. Introduction to Computer Vision](#)

[Technical requirements](#)

[Understanding computer vision](#)

[Learning all about images](#)

[Color spaces](#)

[Input, process, and output](#)

[Computer vision frameworks and libraries](#)

[Summary](#)

[Questions](#)

## 2. Getting Started with OpenCV

Technical requirements

Introduction to OpenCV

The Main modules in OpenCV

Downloading and building/installing OpenCV

Using OpenCV with C++ or Python

Understanding the Mat class

Constructing a Mat object

Deleting a Mat object

Accessing pixels

Reading and writing images

Reading and writing videos

Accessing cameras

Accessing RTSP and network feeds

Mat-like classes

Summary

Questions

Further reading

### 3. Array and Matrix Operations

Technical requirements

Operations contained in the Mat class

Cloning a matrix

Calculating the cross-product

Extracting a diagonal

Calculating the dot product

Learning about the identity matrix

Matrix inversion

Element-wise matrix multiplication

The ones and zeroes matrix

Transposing a matrix

Reshaping a Mat object

Element-wise matrix operations

Basic operations

The addition operation

Weighted addition

The subtraction operation

The multiplication and division operations

Bitwise logical operations

The comparison operations

The mathematical operations

Matrix and array-wise operations

Making borders for extrapolation

Flipping (mirroring) and rotating images

Working with channels

[Mathematical functions](#)

[Matrix inversion](#)

[Mean and sum of elements](#)

[Discrete Fourier transformation](#)

[Generating random numbers](#)

[The search and locate functions](#)

[Locating non-zero elements](#)

[Locating minimum and maximum elements](#)

[Lookup table transformation](#)

[Summary](#)

[Questions](#)

## 4. Drawing, Filtering, and Transformation

Technical requirements

Drawing on images

Printing text on images

Drawing shapes

Filtering images

Blurring/smoothening filters

Morphological filters

Derivative-based filters

Arbitrary filtering

Transforming images

Thresholding algorithms

Color space and type conversion

Geometric transformation

Applying colormaps

Summary

Questions

## 5. Back-Projection and Histograms

Technical requirements

Understanding histograms

Displaying histograms

Back-projection of histograms

Learning more about back-projections

Comparing histograms

Equalizing histograms

Summary

Questions

Further reading

## 6. Video Analysis & Motion Detection and Tracking

Technical requirements

Processing videos

Understanding the Mean Shift algorithm

Using the Continuously Adaptive Mean (CAM) Shift

Using the Kalman filter for tracking and noise reduction

How to extract the background/foreground

An example of background segmentation

Summary

Questions

## 7. Object Detection &#x2013; Features and Descriptors

Technical requirements

Template matching for object detection

Detecting corners and edges

Learning the Harris corner-detection algorithm

Edge-detection algorithms

Contour calculation and analysis

Detecting, descripting, and matching features

Summary

Questions

## 8. Machine Learning in Computer Vision

Technical requirements

Support vector machines

Classifying images using SVM and HOG

Training models with artificial neural networks

The cascading classification algorithm

Object detection using cascade classifiers

Training cascade classifiers

Creating samples

Creating the classifier

Using deep learning models

Summary

Questions

### Assessments

Chapter 1, Introduction to Computer Vision

Chapter 2, Getting Started with OpenCV

Chapter 3, Array and Matrix Operations

Chapter 4, Drawing, Filtering, and Transformation

Chapter 5, Back-Projection and Histograms

Chapter 6, Video Analysis & Motion Detection and Tracking

Chapter 7, Object Detection&#xA0;&#x2013; Features and Descriptors

Chapter 8, Machine Learning in Computer Vision

### Other Books You May Enjoy

Leave a review - let other readers know what you think

# Introduction to Computer Vision

Without a doubt, computer science, and especially the approach to implementing algorithms, has developed rapidly over the years. This is due to the fact that your personal computer and even the smartphone in your pocket are much faster and a lot cheaper than their predecessors. One of the most important fields in computer science that has been impacted by this change is the field of computer vision.

The way computer vision algorithms are implemented and used has seen a dramatic change in recent years. This book, starting with this introductory chapter, is an effort to teach computer vision algorithms using the most up-to-date and modern technologies that are used to implement them.

This is intended as a brief introductory chapter that lays out the foundation of concepts that will be used in many, if not all, of the computer vision algorithms available. Even if you are already familiar with computer vision and the basics, such as images, pixels, channels, and so on, it is still a good idea to go through this chapter briefly to make sure that you understand the fundamental concepts of computer vision and to refresh your memory.

In this chapter, we'll be starting with a brief introduction to the field of computer vision. We'll go through some of the most important industries where computer vision is used, with examples. After that, we'll directly dive into some basic computer vision concepts, starting with images. We'll learn what images are in terms of computer vision and what their building blocks are, too. During this process, we'll cover concepts such as pixels, depth, and channels, all of which are crucial to understanding and successfully working hands-on with computer vision algorithms.

By the end of this chapter, you will have learned about the following:

- What computer vision is and where it is used?
- What an image is in terms of computer vision?
- Pixels, depth, and channels and their relationships

# **Technical requirements**

As this is an introductory chapter, we are focusing solely on theory. There are, therefore, no technical requirements.

# Understanding computer vision

Defining computer vision is not an easy task, and computer vision experts tend to disagree with each other when it comes to providing a textbook definition for it. Doing so is completely out of the scope and interest of this book, so we'll focus on a simple and practical definition that suits our purpose instead.

Historically, computer vision has been synonymous with image processing, which essentially refers to the methods and technologies that take an image as input and produce an output image or a set of output values (or measurements) based on that input image, which is done after performing a set of processes.

Fast forward to now and you'll notice that, when computer vision engineers talk about computer vision, what they mean, in most cases, is a concept relating to an algorithm that is able to mimic human vision, such as seeing (detecting) an object or a person in an image.

So, which definition are we to accept? The answer is quite simple—both. To put it in just a few words, computer vision refers to the algorithms, methods, and technologies that deal with digital visual data (or any data that can be visualized) in any way imaginable. Note that visual data in this sense does not mean just images taken using conventional cameras, but they might be, for instance, a graphical representation or elevation on a map, a heat intensity map, or any data that can be visualized regardless of its real-world meaning.

With this definition, all of the following questions—as well as many more—can be solved with computer vision:

- How do we soften or sharpen an image?
- How do we reduce the size of an image?
- How do we increase or decrease the brightness of an image?
- How do we detect the brightest region in an image?
- How do we detect and track a face in a video (or a series of consecutive images)?
- How do we recognize faces in a video feed from a security camera?
- How do we detect motion in a video?

 In modern computer vision science, image processing is usually a subcategory of computer



*vision methods and algorithms dealing with image filtering, transformation, and so on. Still, many use the terms computer vision and image processing interchangeably.*

In this day and age, computer vision is one of the hottest topics in the computer science and software industry. The reason for this lies in the fact that it is used in a variety of ways, whether it's bringing to life the ideas for applications, digital devices, or industrial machines that handle or simplify a wide range of tasks that are usually expected from the human eye. There are a lot of working examples for what we just mentioned, which vary across a wide spectrum of industries, including the automotive, motion picture, biomedical devices, defense, photo editing and sharing tools, and video game industries. We are going to talk about just a couple of these examples and leave the rest of them for you to research.

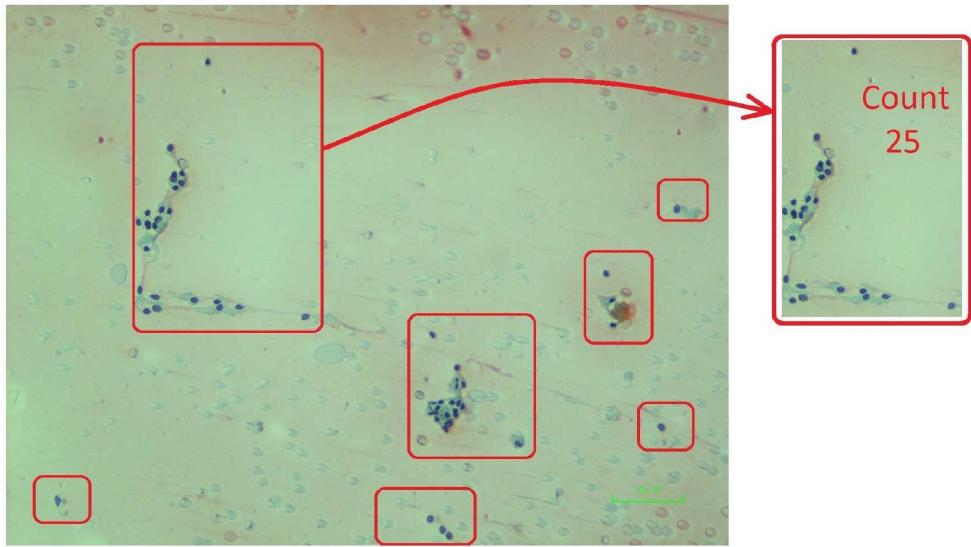
Computer vision is used persistently in the automotive industry to improve the safety and functionality of modern vehicles. Vehicles are able to detect traffic signs and warn the driver about a speed limit breach or even detect lanes and obstacles on the road and notify drivers about possible hazards. There is no end to the number of practical examples we can present about how computer vision can be used to modernize the automotive industry—and that's without touching on self-driving cars. Major tech companies are investing a huge amount of resources and are even sharing some of their achievements with the open source community. As you'll see in the final chapters of this book, we'll make use of some of them, especially for the real-time detection of multiple objects of multiple types.

The following image depicts some of the objects, symbols, and areas of interest for the automotive industry, as images that are commonly seen through the cameras mounted on vehicles:



Another great example of an industry on the verge of a technological revolution is the biomedical industry. Not only have the imaging methods of human organs and body parts undergone a great deal of enhancement, but the way these images are interpreted and visualized has also been improved by computer vision algorithms. Computers are used to detect cancerous tissues in images taken by microscopes with an extremely high level of precision. There are also promising and emerging results from robots that are able to perform surgery, for example.

The following image is an example of using computer vision to count a specific type of biological object of interest (cells, in this case) in various areas of tissue, scanned by a digital microscope:



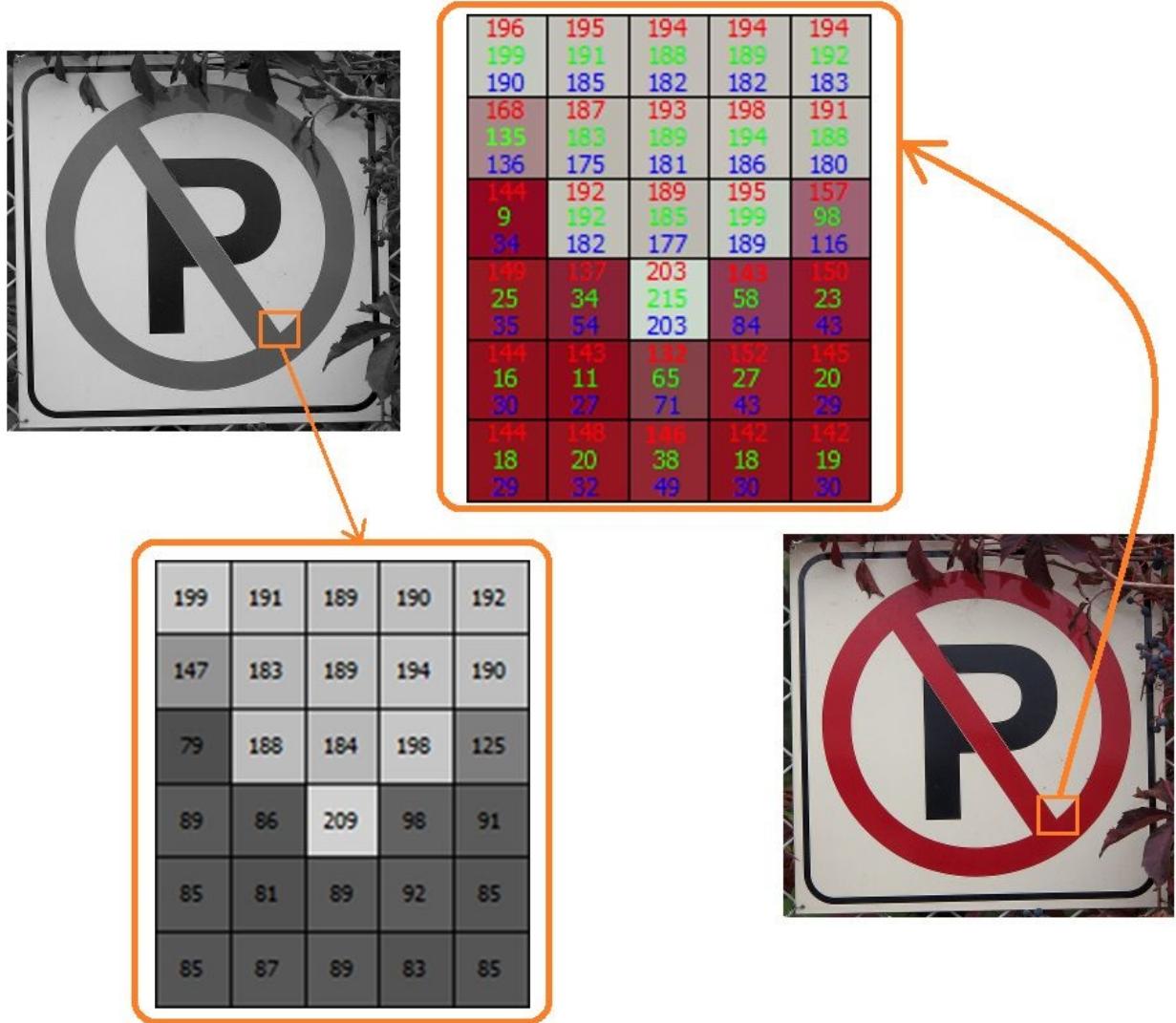
Besides the automotive and biomedical industries, computer vision is also used in thousands of mobile and desktop applications to perform many different tasks. It's a good idea to browse through online application stores on your smartphone to view some computer vision-related application examples. Do so, and you'll immediately realize that there is literally nothing but your imagination standing between you and your potential computer vision application ideas.

# Learning all about images

Now, it's time to cover the basics of computer vision, starting with images. So, what exactly is an image? In terms of computer vision, an image is simply a matrix, or in other words a 2D vector, with a valid number of rows, columns, and so on. This way of looking at an image simplifies not just the description of an image itself, but also all of its components, which are as follows:

- The width of an image corresponds to the number of columns in the matrix.
- The height of the image is the number of rows in the matrix.
- Each element of the matrix represents a pixel, which is the most basic component of an image. An **image** is a collection of pixels.
- Each pixel, or each element in the matrix, can contain one or more numeric values that correspond to the visual representation (color, brightness, and so on) of it. We'll learn more about this later when we talk about color spaces in computer vision. However, it's important to note that each numeric value associated with a pixel represents a channel. For instance, pixels in a grayscale image are commonly represented using a single unsigned 8-bit integer value that varies between 0 and 255; thus, a grayscale image is a single-channel image. In this form of representation, 0 represents black and 255 represents white, while all other numbers correspond to a grayscale value. Another example is standard RGB image representation, in which each pixel is represented by three unsigned 8-bit integer values that vary between 0 and 255. The three channels representing each pixel in RGB images correspond to the intensity values of red, blue, and green, which in combination can form any possible color. Such an image is known as a **three-channel image**.

The following image depicts two zoomed-in versions of the same area from the same image in both a grayscale and colored (RGB) format. Notice how higher values in the grayscale image (on the left-hand side) correspond to brighter values and vice versa. Similarly, in the color image (on the right-hand side), you can see that the value of the red channel is quite high, which is consistent with the reddish color of that area, as well as the white channels:



In addition to what we mentioned previously, an image has a few more specifications, which are as follows:

- Each pixel, or an element of the matrix, can be an integer or a floating-point number. It can be an 8-bit number, 16-bit, and so on. The type of the numeric value representing each pixel in conjunction with the number of channels resembles the depth of an image. For instance, a four-channel image which uses a 16-bit integer value to represent each channel would have a depth of 16 multiplied by 4-bits, or 64-bits (or 4 bytes).
- The resolution of an image refers to the number of pixels in it. For instance, an image which has a width of 1920 and height of 1080 (as is the case in full-HD images) has a resolution of 1920 multiplied by 1080, which is a bit more than 2 million pixels, or about 2 megapixels.

It is all because of this form of representation of an image that it can be easily perceived as a mathematical entity, meaning many different types of algorithms can be designed to act or work on images. If we go back to the simplest representation of an image (a grayscale image), with a few simple examples we can see that most picture editing software (and computer vision algorithms) use this representation along with fairly simple algorithms and matrix operations to modify the image with ease. In the following image, a constant number (80, in our example) is simply added to each pixel in the input image (the middle image), which has made the resulting image brighter (the right-hand image). A number can also be subtracted from each pixel to make the resulting image darker (the left-hand image):



*For now, we'll only focus on the basic concepts of computer vision and not go into the implementation details of the preceding image modification example. We'll learn about this and many more image processing techniques and algorithms in the upcoming chapters.*

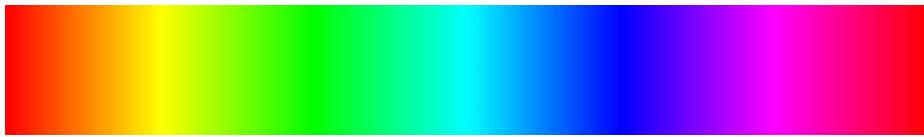
The image properties mentioned in this section (width, height, resolution, depth, and channels) are extensively used in computer vision. For instance, in several cases, if an image processing algorithm is too complex and time-consuming, then the image can be resized to make it smaller so that less time is used to process it. Once processed, the results can then be mapped back to the original image size and displayed to the user. The same process also applies to depth and channels. If an algorithm only needs a specific channel of an image, you can either extract and process it separately or use the grayscale-converted version of an image. Note that, once the object detection algorithm has completed its job, you'll want to display the results over the original colored image. Having a correct understanding of these kinds of image properties will help you a lot when confronting various computer vision problems and when working with computer vision algorithms. Without further ado, let's move on to color spaces.

# Color spaces

Although its definition can vary, in general, a color space (sometimes referred to as a color model) is a method that is used for interpreting, storing, and reproducing a set of colors. Let's break this down with an example—a grayscale color space. In a grayscale color space, each pixel is represented with a single 8-bit unsigned integer value that corresponds to brightness or gray-intensity of that pixel. This makes it possible to store 256 different levels of grayscale, in which zero corresponds to absolute black and 255 corresponds to the absolute white. In other words, the higher the value of a pixel, the brighter it is, and vice versa. The following image displays all possible colors that exist within the grayscale color space:



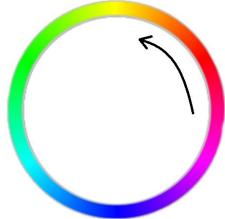
Another commonly-used color space is RGB, in which each pixel is represented by three different 8-bit integer values that correspond to the red, green, and blue color intensity of that pixel. This color space is particularly known for being used in TVs, LCDs, and similar displays. You can check this out for yourself by looking at the surface of your monitor using a magnifier. It relies on the simple fact that all colors can be represented by combining various amounts of red, green, and blue. The following image depicts how all other colors (such as yellow or pink) between the three main colors are formed:



*An RGB image that has the same R, G, and B values in each of its individual pixels would result in a grayscale image. In other words, the same intensity of red, green, and blue would result in a shade of gray.*

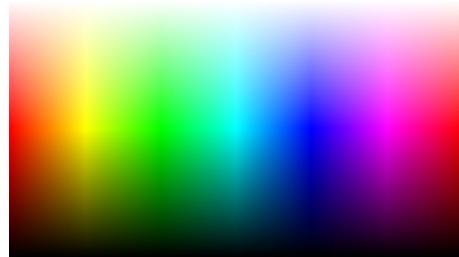
Another color space that is widely used in computer vision is the **HSV (Hue, Saturation, and Value)** color space. In this color space, each pixel is represented by three values for hue (the color), saturation (the color strength), and value (how bright or dark it is). Hue, as seen in the following image, can be a value

between 0 and 360 (degrees), which represents the color of that pixel. For instance, zero and nearby degrees correspond to red and other similar colors:



This color space is especially popular in computer vision detection and tracking algorithms that are based on the color of an object, as you'll see later on in this book. The reason for this is that the HSV color space allows us to work with colors regardless of how dark or bright they are. This is not easy to achieve with RGB and similar color spaces, as looking at an individual pixel channel value cannot tell us its color.

The following image is another representation of the HSV color space, which shows the variation of hue (from left to right), saturation, and value in one image, thus producing all possible colors:



Besides the color spaces mentioned in this section, there are many other color spaces, each with their own use cases. For instance, the four channel **CMYK** color space (**Cyan**, **Maroon**, **Yellow**, and **Key/Black**) has proven to be most effective in printing systems.

Make sure to learn about other popular color spaces from the internet and how they might be useful for any particular computer vision problem.

# Input, process, and output

So, now that we know images are basically matrix-like entities that have essential properties such as width, height, element type, channels, depth, and so on, the only big question that remains is where do they come from, what happens to them, and where do they go?

Let's break this down further with a simple photo gallery application as an example. Odds are, you have a smartphone that contains such an application by default. A photo gallery application usually allows you to take new pictures or videos using the built-in camera on your smartphone, use the previously-recorded files, apply filters on images, and even share them on social media, via email, or with your friends and family. This example, though it might look like a simple application when you are using it, contains all the crucial parts of a proper computer vision application.

By taking this example as a pretext, we can say that images are provided by a wide range of different input devices based on the use case. Some of the most common image input devices are as follows:

- Image files stored on a disk, memory, a network, or any other accessible location. Note that stored image files can be raw (containing the exact image data) or encoded (such as JPG); nevertheless, they're still considered image files.
- Images captured by cameras. Note that cameras in this sense mean webcams on a personal computer, cameras on a smartphone, or any other professional photography device, digital microscope, telescope, and so on.
- Consecutive or non-consecutive frames from a video file that are stored on a disk, memory, a network, and so on. Similar to image files, video files can be encoded, in which case a special type of software (called a **codec**) is needed to encode them before they can be used.
- Consecutive frames from a live video camera feed.

After an image is read using an input device, the actual processing of it starts. This is probably the part of the computer vision process cycle that you are looking for in this book —and for good reason. This is where actual computer

vision algorithms are used to extract values from an image, modify it in one way or another, or perform any type of computer vision task in general. This part is usually done by software on a given device.

Now, the output of this whole process needs to be created. This part completely depends on the computer vision algorithm and the type of device that the computer vision process is running on, but in general, the following types of outputs are expected from computer vision algorithms:

- Numbers, shapes, graphs, or any other non-image type of output that is derived from the processed image. For instance, an algorithm that counts the number of people in an image needs to only output a single integer number or a graph representing the number of people that are found in consecutive video frames from a security camera.
- Images or video files stored on a disk, memory, and similar devices. A typical example of this is the photo editing software on your phone or personal computer that allows you to record the modified image as a JPG or PNG file.
- Images and video frames drawn and rendered on a display screen. Displays are usually controlled with a firmware (which might be on an operating system) that controls whatever is being shown on them.

Similar to input devices, a slightly different interpretation of an image output device will yield more results and entries (such as printers, plotters, video projectors, and so on). However, the preceding list is still sufficient as it covers the most basic and crucial types of output we'll be dealing with when working with computer vision algorithms.

# Computer vision frameworks and libraries

In order to build computer vision applications, we need a set of tools, a framework, or a library that supports the input, output, and the processing of images. The choice of a computer vision library is a very important one because you might end up in a position where you'll need to *reinvent the wheel* all by yourself. You may also end up writing functions and codes that will take up a lot of your resources and time, such as reading or writing an image in the format that you require.

In general, there are two main types of computer vision libraries that you can choose from when developing computer vision applications; they are as follows:

- **Proprietary:** Proprietary computer vision libraries are usually well-documented and supported by the companies providing it, but they come at a price and are often aimed at a specific set of computer vision problems
- **Open source:** Open source libraries, on the other hand, usually cover a much wider range of computer vision related issues and they are free to use and explore

You can search for many good examples of both proprietary and open source computer vision libraries online to compare them for yourself.

The library that we'll be using throughout this book is the **Open Source Computer Vision library (OpenCV)**. OpenCV is a computer vision library with the following features:

- It is open source and free for use on academic or commercial projects
- It supports C++, Python, and Java languages
- It is cross-platform, which means it can be used to develop applications for Windows, macOS, Linux, Android, and iOS
- It is built in a modular fashion and it is fast, well-documented, and well-supported

It is worth noting that OpenCV also uses a few third-party libraries to take care of various computer vision tasks. For instance, the FFmpeg library is used within OpenCV to deal with reading certain video file formats.

# Summary

In this chapter, we introduced the most basic concepts of computer vision science. We started the chapter by learning about computer vision as a term and its use cases, before taking a look at some of the industries that make extensive use of it. Then, we moved on to learn about images and their most crucial properties, namely pixels, resolution, channels, depth, and so on. We then discussed some of the most widely-used color spaces and learned how they affect the number of channels and other properties of an image. After that, we were presented with the common input and output devices used in computer vision and how computer vision algorithms and processes fit between the two. We ended the chapter with a very brief discussion on computer vision libraries and introduced our computer vision library of choice, which is OpenCV.

In the next chapter, we'll introduce the OpenCV framework and start with some hands-on computer vision lessons. We'll learn how OpenCV can be used to access input devices, perform computer vision algorithms, and access output devices to display or record results. The next chapter will be the first real hands-on chapter in the book and will set us up for later, more practical chapters.

# Questions

1. Name two industries, besides the ones mentioned in this chapter, that can significantly benefit from computer vision.
2. What would be an example of a computer vision application used for security purposes? (Think about an idea for an application you haven't come across.)
3. What would be an example of a computer vision application used for productivity reasons? (Again, think about an idea for an application that you haven't come across, even though you might suspect that it exists.)
4. How many megabytes would be needed to store a 1920 x 1080 image with four channels and a depth of 32-bits?
5. Ultra-HD images, also known as 4K or 8K images, are quite common nowadays, but how many megapixels does an ultra-HD image contain?
6. Name two commonly-used color spaces besides the ones mentioned in this chapter.
7. Compare OpenCV libraries with computer vision tools in MATLAB. What are the pros and cons of each?

# Getting Started with OpenCV

In the previous chapter, we were presented with an introduction to computer vision, with some examples of the industries that use it extensively to improve their services and products. Then, we learned about the most basic concepts used in this field, such as images and pixels. We learned about color spaces and finished the chapter with a brief discussion of computer vision libraries and frameworks. We're going to continue from where we left off, which is with introducing you to one of the most powerful and widely used computer vision libraries, called **OpenCV**.

OpenCV is a huge collection of classes, functions, modules, and other related resources used to build cross-platform computer vision applications. In this chapter, we will learn about the structure of OpenCV, the modules that it contains and their purposes, and the programming languages that it supports. We will learn how and where to get OpenCV and briefly go through the possible tools you can use to build applications with it. Then, we'll learn how to use the power of CMake to easily create projects that use OpenCV. Even though this means that our main focus will be on C++ classes and functions, we'll also cover Python equivalents of them wherever it makes sense so that developers familiar with both languages can follow the topics that are presented here in this chapter.

After learning about the initial stages of using the OpenCV library, we'll move on to learn about the `Mat` class. We'll see how all of the concepts that we covered in the previous chapter about images are embedded into the structure of the `Mat` class in OpenCV. We'll also talk about the various other classes that are compatible with (or closely related to) the `Mat` class. OpenCV's method of handling input and output parameters in its functions is a very important subject that we'll be covering in the later sections of this chapter. Finally, we'll learn how OpenCV is used to apply the three steps of input, process, and output in computer vision applications. This will require learning about accessing (and writing into) images and video files using OpenCV.

This chapter, as a direct connection to the previous introductory chapter, will lay out the foundations of learning computer vision algorithms using hands-on and

practical examples.

In this chapter, we'll look at the following:

- What OpenCV is, where to get it, and how to use it?
- How to use CMake to create OpenCV projects?
- Understanding the `Mat` class and how it is used to access pixels
- How to use the `Mat_`, `Matx`, and `UMat` classes?
- How to use the `imread` and `imwrite` functions to read and write images?
- How to use the `VideoCapture` and `Videowriter` classes to read and write videos?
- How to access cameras and video feeds from a network (using **Real Time Streaming Protocol (RTSP)**)?

# Technical requirements

- Microsoft Visual Studio, Xcode, or any IDE that can be used to develop C++ programs
- Visual Studio Code or any other code editor that can be used to edit CMake files, Python source files, and so on
- Python 3.X
- CMake 3.X
- OpenCV 3.X



*It is always best to try to use a more recent version of the technologies and software that you are trying to learn. The topics covered in this book, and computer vision in general, are no exception, so make sure to download and install the latest versions of the mentioned software.*

Wherever necessary, a brief set of instructions are provided for the installation and configuration. You can use the following URL to download the source codes and examples for this chapter:

<https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter02>

# Introduction to OpenCV

**OpenCV**, or **Open Source Computer Vision**, is a set of libraries, tools, and modules that contain classes and functions required for building computer vision applications. Downloaded millions of times by computer vision developers around the world, the OpenCV library is fast and optimized to be used in real-life projects (including commercial). As of the time of writing this book, the most recent version of OpenCV is 3.4.1, which is also the version we'll be using in all of the examples in this book. OpenCV supports the C/C++, Python, and Java languages, and it can be used to build computer vision applications for desktop and mobile operating systems alike, including Windows, Linux, macOS, Android, and iOS.



*It's important to note that the OpenCV library and OpenCV framework are both used to refer to OpenCV, and in the computer vision community the terms are used interchangeably most of the time. For the same reason, we will also use the terms interchangeably throughout this book. However, strictly speaking, framework is usually the term that is used to refer to sets of related libraries and tools working toward achieving a common goal, such as OpenCV.*

OpenCV consists of the following two types of modules:

- **Main modules:** These modules are included in OpenCV release versions by default, and they contain all of the core OpenCV functionality along with modules that are used for image-processing tasks, filtering, transformation, and many more capabilities that we'll talk about in this section.
- **Extra modules:** These modules include all of the OpenCV functionalities that are not included in the OpenCV library by default and they mostly include additional computer vision-related functionalities. For instance, the Extra modules include libraries used for text recognition and non-free feature detectors. Note that our focus will be on the Main module and covering the functionalities included in it but, wherever it might be helpful, we'll try to also refer to possible options in the extra modules for you to research on your own.



# The Main modules in OpenCV

As mentioned, OpenCV contains a number of Main modules that contain all of its core and default functionalities. Here is a list of those modules:

- `core`: This contains all of the core OpenCV functionalities. For instance, all basic structures, including the `Mat` class (which we'll learn about in detail later) and matrix operations are some of the functionalities embedded into this module.
- `imgproc`: This module contains all image-processing functionalities, such as filtering, transformations, and histograms.
- `imgcodecs`: This module includes functions that are used for reading and writing images.
- `videoio`: This is similar to the `imgcodecs` module, but this one is used to work with videos, as the name implies.
- `highgui`: This module, which we'll use extensively throughout the book, contains all of the functionalities used for displaying results and GUI creation in general. Note that even though the `highgui` module is enough for the purpose of this book and learning about computer vision algorithms while visualizing the results as we move forward, it's still not meant for full-scale applications. Refer to the *Further reading* section at the end of this chapter for more references about proper GUI creation tools for full-scale computer vision applications.
- `video`: Contains video analysis functionalities of OpenCV, such as motion detection and tracking, the Kalman filter, and the infamous CAM Shift algorithm (used for object tracking).
- `calib3d`: This module includes calibration and 3D reconstruction functionalities. A well-known example of the capabilities of this module is the estimation of transformation between two images.
- `features2d`: Supported keypoint-detection and descriptor-extraction algorithms are included in this module. As we'll learn in the upcoming chapters, this module contains some of the most widely used object detection and categorization algorithms.
- `objdetect`: As the name implies, this module is used for object detection

using OpenCV. We'll learn about the functionalities contained within this module in the final chapters of this book.

- `dnn`: Similar to the `objdetect` module, this module is also used for object detection and classification purposes, among others. The `dnn` module is relatively new in the list of the Main modules of OpenCV, and it contains all of the capabilities related to deep learning.
- `ml`: This machine learning module contains the classes and functions used to handle classification and regression. Simply put, all strictly machine learning-related capabilities are included in this module.
- `flann`: This is OpenCV's interface to **Fast Library for Approximate Nearest Neighbors (FLANN)**. FLANN contains a wide set of optimized algorithms that are used to deal with the nearest neighbor search of high-dimensional features in large datasets. The algorithms mentioned here are mostly used in conjunction with algorithms in other modules, such as `features2d`.
- `photo`: An interesting module for photography-related computer vision, it contains classes and functions that are used to deal with tasks such as denoising, HDR imaging, and restoring a region in a photo using its neighborhood.
- `stitching`: This module contains classes and functions used for image stitching. Note that stitching in itself is a very complex task and it requires functions for rotation estimation and image warping, all of which are also part of this very interesting OpenCV module.
- `shape`: This module is used to deal with shape transformation, matching, and distance-related topics.
- `superres`: Algorithms that fall into the category of resolution enhancement are included in the super-resolution module.
- `videostab`: This module contains algorithms used for video stabilization.
- `viz`: Otherwise known as the 3D Visualizer module, it contains classes and functions that are used to deal with displaying widgets on the 3D visualization window. This module will not be part of the topics discussed in this book, but we'll just mention it.

Apart from the modules that we just covered, OpenCV also contains a number of Main modules that are based on CUDA (an API created by Nvidia). These modules are easily distinguished by their names, which start with the word `cuda`. Since the availability of these modules totally depends on a specific type of hardware, and almost all of the functionalities inside those modules are covered, one way or another, by other modules, we're going to skip them for now. But it's

worth noting that using the OpenCV `cuda` modules can significantly improve the performance of your applications, provided that the algorithms you need are implemented in them, and that your hardware meets the minimum requirements for them.

# Downloading and building/installing OpenCV

OpenCV, for the most part, does not have a prebuilt and ready-to-use version (there are some exceptions we'll cover in this section) and similar to most of the open source libraries, it needs to be configured and built from sources. In this section, we'll quickly describe how OpenCV is built (and installed) on a computer. But first, you need to get the OpenCV source codes in your computer. You can use the following link for this:

<https://opencv.org/releases.html>

On this page, you can find release versions of OpenCV. The latest version, as of the time of writing this book, is 3.4.1 so you should download it, or if there is a higher version then just go with that.

As seen in the following screenshot, for each version of the OpenCV release, there are various downloadable entries, such as the Win, iOS, and Android packs, but you should download Sources and build OpenCV yourself based on the platform you want to work with:



ABOUT NEWS EVENTS RELEASES PLATFORMS BOOKS LINKS LICENSE

## Releases

<b>3.4.1</b>	Documentation
2018-02-27	Sources
3.4.1	Win pack
	iOS pack
	Android pack



<b>2.4.13.6</b>	Documentation
2018-02-26	Sources
2.4.13.6	Win pack
	iOS pack



*OpenCV 3.4.1, by default, provides prebuilt versions of the Android, iOS, and 64-bit MSVC14 and MSVC15 (the same as Microsoft Visual C++ 2015 and Microsoft Visual C++ 2017) libraries. So, if you want to build applications for any of these platforms, you can download the relevant pack and skip the OpenCV build process altogether.*

To build OpenCV from sources, you need the following tools on your computer:

- **C/C++ compiler with C++11 support:** On Windows, this means any of the more recent Microsoft Visual C++ compilers, such as MSVC15 (2017) or MSVC14 (2015). On Linux operating systems, you can use an up-to-date GCC and, on macOS, you can use the command-line tools for Xcode that contain all the required tools.
- **CMake:** Make sure you use the latest version of CMake, such as 3.10, to be on the safe side with more recent versions of OpenCV, although you can use CMake 3.1 and later.
- **Python:** This is especially important if you are aiming to use the Python programming language.



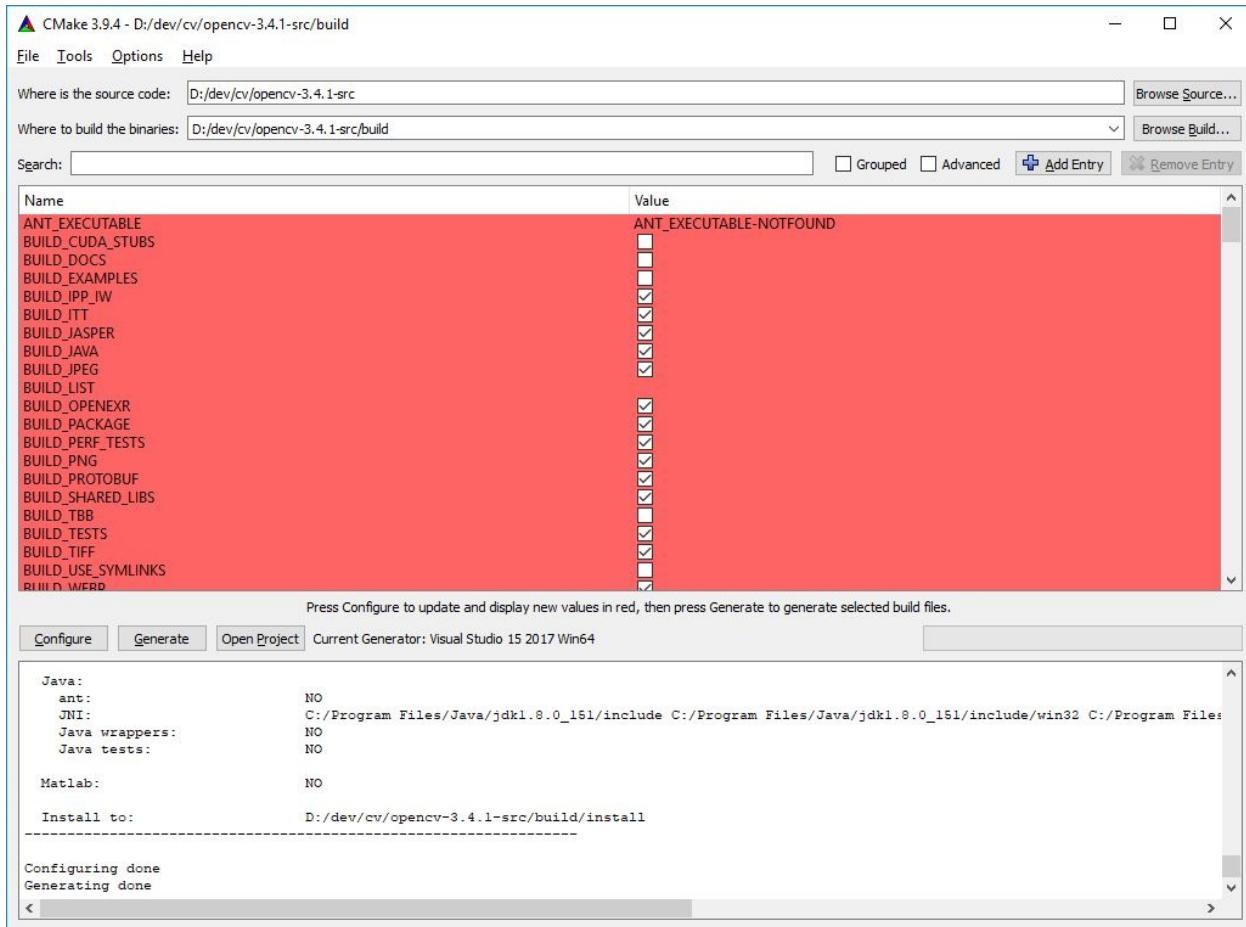
*OpenCV contains a large number of tools and libraries and it's possible to customize your build in many different ways. For instance, you can use the Qt Framework, Intel **Threading Building Blocks (TBB)**, Intel **Integrated Performance Primitives (IPP)**, and other third-*



party libraries to further enhance and customize your OpenCV build, but since we'll be using OpenCV with the default settings and set of tools, we're ignoring the aforementioned third-party tools in the list of requirements.

After getting all of the prerequisites we just mentioned, you can configure and build OpenCV by using CMake and the corresponding compilers, depending on your operating system and desired platforms.

The following screenshot depicts the CMake tool with a default set of configurations visible. Usually, you don't need to make any change to the configurations unless you want to apply your own set of customizations to the OpenCV build:



Note that when CMake is first opened, you need to set the source and build folders, which are visible in the preceding screenshot, as Where is the source code: and Where to build the binaries:, respectively. After hitting the Configure button, you need to set a generator and apply the settings, then press Generate.

After generation, you can simply switch to the CMake output folder using a Terminal or Command Prompt instance and execute the following commands:

```
| make  
make install
```

 Be aware that running each one of these commands can take some time depending on your computer's speed and configurations. Also note that the `make` commands can be different depending on the set of tools you are aiming to use. For instance, if you are using Microsoft Visual Studio, then you need to replace `make` with `nmake`, or if you are using MinGW then you have to replace `make` with `mingw32-make`.

After the build process is completed, you can start using OpenCV. The only thing you need to take care of is configuring your C++ projects so that they can use your set of OpenCV libraries and installation.

 On Windows operating systems, you need to make sure OpenCV DLL files are accessible by the applications you are building. This can be done either by copying all of the required DLLs to the same folder where your applications are being built or simply by adding the path of the OpenCV DLL files to the PATH environment variable. Make sure to take care of this before proceeding further, otherwise your applications will crash when they are executed, even though they might build successfully and not report any issues at compile time.

If you are going to use Python to build computer vision applications, then things are going to be extremely simple for you since you can use `pip` (package manager) to install OpenCV for Python, using the following command:

```
| pip install opencv-python
```

This will automatically get you the latest OpenCV version and all of its dependencies (such as `numpy`) or, if you have already installed OpenCV, you can use the following command to make sure it's upgraded to the latest version:

```
| pip install --upgrade opencv-python
```

It goes without saying that you need a working internet connection for these commands to work.

# Using OpenCV with C++ or Python

In this section, we'll demonstrate how you can use OpenCV in your C++ or Python projects with a very simple example that we'll call `HelloOpenCV`. You might already know that the purpose of such a project is either one of the following:

- To get started with a new library, such as OpenCV, that you've never used before
- To make sure your OpenCV installation is functional and works fine

So, even if you are not an OpenCV beginner, it's still worth going through the following instructions and running the simple example in this section to test your OpenCV build or installation.

We'll start with the required steps for using OpenCV in a C++ project:

1. Create a new folder named `HelloOpenCV`
2. Create two new text files inside this folder and name them `CMakeLists.txt` and `main.cpp`
3. Make sure the `CMakeLists.txt` file contains the following:

```
cmake_minimum_required(VERSION 3.1)

project(HelloOpenCV)

set(OpenCV_DIR "path_to_opencv")

find_package(OpenCV REQUIRED)

include_directories(${OpenCV_INCLUDE_DIRS})
```

```
add_executable(${PROJECT_NAME} "main.cpp")
```

```
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

 *In the preceding code, you need to replace "path\_to\_opencv" with the path to the folder containing the `opencvconfig.cmake` and `opencvConfig-version.cmake` files, which is the same folder where you have installed your OpenCV libraries. If you are using the Linux operating system and the prebuilt OpenCV libraries, you might not need an exact path to the `opencv` folder.*

4. As for the `main.cpp` file, make sure it contains the following, which is the actual C++ code we'll be running:

```
#include <iostream>

#include <opencv2/opencv.hpp>

using namespace std;

using namespace cv;

int main()

{

    Mat image = imread("MyImage.png");

    if(!image.empty())

    {
```

```
    imshow("image", image);

    waitKey();

}

else

{

    cout << "Empty image!" << endl;

}

return 0;

}
```

We'll be covering the functions used in the preceding code individually later on, in this and the upcoming chapters, however, for now it's worth noting that this program is trying to open and display an image saved on disk. If it succeeds, the image will be displayed until any key is pressed, otherwise the `Empty image!` message will be displayed. Note that this program, under normal circumstances, should not crash at all and it should build successfully. So, if the opposite happens to you, then you'll need to go through the topics discussed previously in this chapter.

5. Our C++ project is ready. Now, we can use CMake to generate Visual Studio or any other type of project that we want (depending on the platform, compiler, and IDE we're going to use) and then build and run it. Note that CMake is simply used to make sure a cross-platform and IDE-independent C++ project is created.

By running this example project, your input image (in this case `MyImage.png`)

will be read and displayed until any key on the keyboard is pressed. If any problems occur during the reading of the image, then the `Empty image!` message will be displayed.

We can create and run the same project in Python by using the following code:

```
import cv2

image = cv2.imread("MyImage.png")

if image is not None :

    cv2.imshow("image", image)

    cv2.waitKey()

else:

    print("Empty image!")
```

The resemblance is quite unmistakable here. The exact same `imshow` and `waitKey` functions are also used in the Python version of the same code. As was mentioned before, for now don't bother with the exact way of using any of the functions and just focus on making sure that you are able to run these programs, either in C++ or Python, or both, and that you are able to see the image displayed.

If you were able to successfully run the `HelloOpenCV` example project in this section, then you are ready to take on the next sections of this chapter and the next chapters of this book without any problems. If you still face problems with the topics discussed so far, or you feel that you need a stronger understanding of those topics, you can go through them once again from the start of the chapter, or

even better, you can refer to the additional books mentioned in the *Further reading* section at the end of this chapter.

# Understanding the Mat class

Refer back to the description that was provided for an image in computer vision from the previous chapter, and that any image is in fact a matrix with a given width, height, number of channels, and depth. With this description in mind, we can say that the OpenCV `Mat` class can be used to handle image data and it supports all properties required by an image, such as width and height. In fact, the `Mat` class is an n-dimensional array that can be used to store single or multiple channels of data with any given data type, and it contains a number of members and methods to create, modify, or manipulate it in many ways.

In this section, we're going to learn about some of the most important members and methods of the `Mat` class with example use cases and code samples.



*The equivalent of the OpenCV C++ `Mat` class in Python is not originally an OpenCV class, and it is represented by the `numpy.ndarray` type. NumPy is a Python library that contains a wide set of numerical algorithms and mathematical operations, and it supports working with large multi-dimensional arrays and matrices. The reason why the `numpy.ndarray` type in Python is used as `Mat` is that it offers the best (if not the same) set of members and methods required by the OpenCV `Mat` class in C++. For a complete list of members and methods supported by `numpy.ndarray`, you can refer to NumPy documentation.*

# Constructing a Mat object

`Mat` contains about 20 different constructors that can be used to create instances of it, based on what kind of initialization is needed. Let's see some of the most commonly used constructors with some examples.

Creating a `Mat` object (or class instance) that has a width of `1920` and height of `1080`, with three channels that contain 32-bit floating-point values looks as follows:

```
| Mat image(1080, 1920, CV_32FC3);
```

Note that the `type` parameter in the `Mat` constructors accepts a special type of parameter, that is, a constant value that contains the depth, type, and number of channels. The pattern is seen here:

```
|     cv_<depth><type>c<channels>
```

`<depth>` can be replaced by 8, 16, 32, or 64, which represent the number of bits used to store each element in the pixels. The actual number of bits needed for each pixel can be calculated by multiplying this number with the number of channels, or in other words, `<channels>`. Finally, `<type>` needs to be replaced with `u`, `s`, or `f` for the unsigned integer, signed integer, and floating-point values, respectively. For example, you can use the following to create standard grayscale and colored images with a width of `800` and height of `600` pixels. Notice that only the number of channels is different, and that the depth and type parameters represent 8-bit unsigned integers:

```
|     Mat grayscaleImage(600, 800, CV_8UC1);
|     Mat colorImage(600, 800, CV_8UC3);
```

You can use the following constructor to create a three-channel RGB image with a width of `w`, height of `h`, and 8-bit unsigned integer elements, and then initialize all elements with the `R`, `G`, and `B` color value:

```
|     int w = 800, h = 600, R = 50, G = 150, B = 200;
|     Mat image(h, w, CV_8UC3, Scalar(R, G, B));
```

 *It's important to note that the default order of colors in OpenCV is BGR (instead of RGB), which means a swapped `B` and `R` value. This is especially important if we're aiming to display our processed images at some point during the application run time.*

So, the correct way of the scalar initializer in the preceding code would be as follows:

```
|   Scalar(B, G, R)
```

If we need a `Mat` object with higher dimensions, we can use the following. Note that in the following example, a `Mat` object of seven dimensions is created. The size of each dimension is provided in the `sizes` array and each element in the high dimensional `Mat`, which is called `hdm`, contains two channels of 32-bit floating-point values:

```
| const int dimensions = 7;
| const int sizes[dimensions] = {800, 600, 3, 2, 1, 1, 1};
| Mat hdm(dimensions, sizes, CV_32FC2);
```

Another way of achieving the same thing would be using C++ vectors, as seen in the following example:

```
| vector<int> sizes = {800, 600, 3, 2, 1, 1, 1};
| Mat hdm(sizes, CV_32FC2);
```

Similarly, you can provide an additional `scalar` parameter to initialize all of the values in `Mat`. Note that the number of values in `scalar` must match the number of channels. For instance, to initialize all of the elements in the previous seven-dimensional `Mat`, we can use the following constructor:

```
| Mat hdm(sizes, CV_32FC2, Scalar(1.25, 3.5));
```

The `Mat` class allows us to use already-stored image data to initialize it. Using this constructor, you can make your `Mat` class contain the same data that the `data` pointer is pointing to. Note that this constructor does not create a completely new copy of the original data, and it only makes this newly created `Mat` object point to it. This allows very efficient initialization and construction of the `Mat` classes, but has the obvious downside of not taking care of the memory cleanup when it is not needed, so you need to be extra careful when using this constructor:

```
| Mat image(1080, 1920, CV_8UC3, data);
```

Note that, unlike the previous constructors and their initializers, `data` here is not a `scalar` but a pointer to a chunk of memory that contains a `1920` by `1080` pixel, three-channel image data. This method of initializing a `Mat` object with a pointer to a

memory space can also be used with higher dimensionalities of the `Mat` class.

One last constructor type, which is also one of the most important constructors of the `Mat` class, is the **region of interest (ROI)** constructor. This constructor is used to initialize a `Mat` object with a region inside another `Mat` object. Let's break this down with an example. Imagine you have an image and you want to apply some modifications to a specific region inside that image, or in other words to the ROI. You can use the following constructor to create a `Mat` class that has access to the ROI, and any changes applied to it will affect the same region of the original image. Here's how you can do that:

```
| Mat roi(image, Rect(240, 140, 300, 300));
```

If the preceding constructor is used while `image` (which itself is a `Mat` object) contains the picture on the left side of the following image, then `roi` will have access to the region highlighted in that image, and it will contain the image seen on the right side:



The `Rect` class in OpenCV is used to represent a rectangle that has a top-left point, width, and height. For instance, the `Rect` class that was used in the preceding code example has a top-left point of `240` and `140`, width of `300`, and height of `300` pixels, as seen here:

```
| Rect(240, 140, 300, 300)
```

As was mentioned earlier, modifying the ROI in any way will result in the original image being modified. For instance, we can apply something similar to the following image-processing algorithm to `roi` (for now don't bother with the nature of the following algorithm, as we'll learn more about it in the upcoming chapters, and just focus on the concept of ROIs):

```
| dilate(roi, roi, Mat(), Point(-1,-1), 5);
```

If we attempt to display the image, the result would be similar to the following. Notice the area that was highlighted in the previous image is modified (dilated) in the following image, even though we applied the change to `roi`, and not the image itself:



Similar to constructing an `ROI Mat` object using a `Rect` class that corresponds to a rectangular region of an image, you can also create an ROI that corresponds to a range of columns and rows in the original `Mat` object. For instance, the same region in the preceding example can also be accessed with the ranges seen in the following constructor:

```
|     Mat roi(image, Range(140, 440), Range(240, 540));
```

The `Range` class in OpenCV represents a range that has `start` and `end`. Depending on the values of `start` and `end`, `Range` class can be checked to see whether it's empty or not. In the preceding constructor, the first `Range` class corresponds to the rows of the original image, from row `140` to row `440`. The second `Range` corresponds to the columns of the original image, from column `240` to column `540`. The shared space of the provided two ranges is considered to be the resulting ROI.

# Deleting a Mat object

A `Mat` object can be cleaned up by using its `release` function, however, since `release` is called in the destructor of the `Mat` class, there is usually no need to call this function at all. It's important to note that the `Mat` class shares the same data between multiple objects that point to it. This has the advantage of less data copying and less memory usage, and since all of the reference counting is done automatically, you don't usually need to take care of anything.

The only scenario in which you need to take extra care of how and when your objects and data are cleaned up is when you use a data pointer to construct a `Mat` object, as mentioned in the previous section. In such cases, calling the `release` function of the `Mat` class, or its destructor, will have nothing to do with the external data you have used to construct it, and the cleanup will be completely on your shoulders.

# Accessing pixels

Apart from using an ROI to access the pixels in a rectangular region of an image, as we did in the previous sections, there are a few other methods for achieving the same goal or even for accessing individual pixels of an image. To be able to access any single pixel in an image (in other words, a `Mat` object), you can use the `at` function, as seen in the following example:

```
| image.at<TYPE>(R, C)
```

In the preceding example, in the usage of the `at` function, `TYPE` must be replaced with a valid type name that is in accordance with the number of channels and depth of the image. `R` must be replaced with the row number, and `C` with the column number of the pixels we want to have access to. Notice that this is slightly different from the usual pixel-access methods in many libraries, in which the first parameter is `X` (or left) and the second parameter is `Y` (or top). So basically, the parameters appear reversed here. Here are some examples of accessing individual pixels in different types of `Mat` objects.

Accessing a pixel in a single-channel `Mat` object with 8-bit integer elements (grayscale images) is done as follows:

```
| image.at<uchar>(R, C)
```

Accessing a pixel in a single-channel `Mat` object with floating-point elements is done as follows:

```
| image.at<float>(R, C)
```

Access a pixel in a three-channel `Mat` object with 8-bit integer elements as follows:

```
| image.at<Vec3b>(R, C)
```

In the preceding code, the `Vec3b` (vector of 3 bytes) type is used. This and various other similar vector types are defined in OpenCV for convenience. Here is the pattern of the OpenCV `Vec` types that you can use with the `at` function, or for any other purposes:

```
|     Vec<N><Type>
```

<N> can be replaced with 2, 3, 4, 6, or 8 (or omitted in the case of 1) and it corresponds to the number of channels in a `Mat` object. <Type>, on the other hand, can be one of the following, which represent the type of the data stored in each channel of each pixel:

- `b` for `uchar` (`unsigned char`)
- `s` for `short` (`signed word`)
- `w` for `ushort` (`unsigned word`)
- `i` for `int`
- `f` for `float`
- `d` for `double`

For instance, `vec4b` can be used to access the pixels of a four-channel `Mat` object with the `uchar` elements, and `vec6f` can be used to access the pixels of a six-channel `Mat` object with the `float` elements. It's important to note that the `vec` type can be treated like an array to access individual channels too. Here's an example of how to access the second channel of a three-channel `Mat` object with the `uchar` elements:

```
|     image.at<Vec3b>(R, C)[1]
```

It's important to note that by access we mean both reading and writing to a pixel and its individual channels. For instance, the following example is one way to apply a sepia filter to an image:

```
for(int i=0; i<image.rows; i++)
{
    for(int j=0; j<image.cols; j++)
    {
        int inputBlue = image.at<Vec3b>(i,j)[0];
        int inputGreen = image.at<Vec3b>(i,j)[1];
        int inputRed = image.at<Vec3b>(i,j)[2];

        int red =
            inputRed * 0.393 +
            inputGreen * 0.769 +
            inputBlue * 0.189;

        if(red > 255 ) red = 255;

        int green =
            inputRed * 0.349 +
            inputGreen * 0.686 +
            inputBlue * 0.168;

        if(green > 255) green = 255;
    }
}
```

```

        int blue =
            inputRed * 0.272 +
            inputGreen * 0.534 +
            inputBlue * 0.131;

        if(blue > 255) blue = 255;

        image.at<Vec3b>(i,j)[0] = blue;
        image.at<Vec3b>(i,j)[1] = green;
        image.at<Vec3b>(i,j)[2] = red;
    }
}

```

First, a few things to note here are the `rows` and `cols` members of the `image`, which basically represent the number of rows (or height) and the number of columns (or width) in it. Also notice how the `at` function is used both to extract the values of channels and to write updated values into them. Don't worry about the values used in this example to multiply and get the correct sepia tone, as they are specific to the tone itself, and essentially any type of operation can be applied to the individual pixels to change them.

The following image depicts the result of applying the preceding example code on a three-channel color image (left—original image, right—filtered image):



Another method of accessing the pixels in an image is by using the `forEach` function of the `Mat` class. `forEach` can be used to apply an operation on all pixels in parallel, instead of looping through them one by one. Here's a simple example that shows how `forEach` is used to divide the value of all pixels by 5, which would result in a darker image if it is executed on a grayscale image:

```

image.forEach<uchar>([](uchar &p, const int *)
{
    p /= 5;
});

```

In the preceding code, the second parameter, or the position parameter (which is not needed and therefore omitted here) is the pointer to the position of the pixel.

Using the previous `for` loop, we would need to write the following:

```
|     for(int i=0; i<image.rows; i++)
|         for(int j=0; j<image.cols; j++)
|             image.at<uchar>(i, j) /= 5;
```

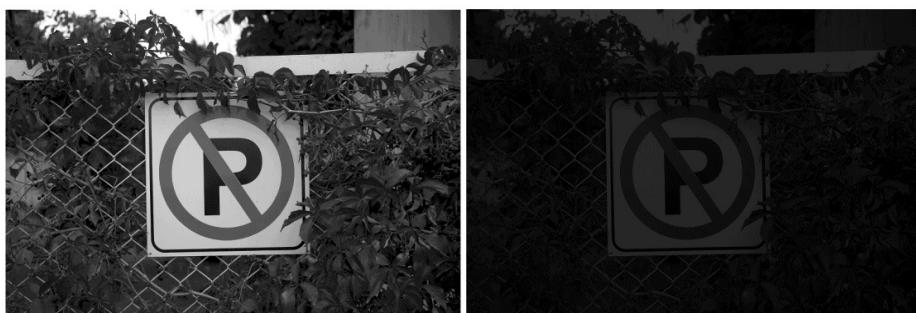
OpenCV also allows the use of STL-like iterators to access or modify individual pixels in an image. Here's the same example but written using STL-like iterators:

```
|     MatIterator_<uchar> it_begin = image.begin<uchar>();
|     MatIterator_<uchar> it_end = image.end<uchar>();
|     for( ; it_begin != it_end; it_begin++)
|     {
|         *it_begin /= 5;
|     }
```

It's interesting to note that the same operation in all of the preceding three examples can also be done by using the following simple statement:

```
|     image /= 5;
```

This is because the `Mat` object in OpenCV treats this statement as an element-wise divide operation, which we'll learn more about it in the upcoming chapters. The following image depicts the result of applying the preceding examples to a grayscale image (left—original image, right—modified image):



Obviously, `forEach`, the C++ `for` loop, and STL-like iterators can all be used to access and modify the pixels within a `Mat` object. We'll suffice to the functions and members discussed in this section about the `Mat` class, but make sure to explore the huge set of functionalities it provides to work with images and their underlying properties in an efficient way.

# Reading and writing images

OpenCV allows reading an image from a disk into a `Mat` object using the `imread` function, which we briefly used in this chapter in a previous example. The `imread` function accepts an input image file name and a `flag` parameter, and returns a `Mat` object filled with the input image. The input image file must have one of the image formats supported by OpenCV. Here are some of the most popular supported formats:

- **Windows bitmaps:** \*.bmp, \*.dib
- **JPEG files:** \*.jpeg, \*.jpg, \*.jpe
- **Portable Network Graphics:** \*.png
- **Portable Image Format:** \*.pbm, \*.pgm, \*.ppm, \*.pxm, \*.pnm
- **TIFF files:** \*.tiff, \*.tif

Make sure to always check the OpenCV documentation for a complete and updated list, and especially for exception cases and notes that might apply to some formats on some operating systems. As for the

`flag` parameter, it can be one or a combination of the values from the `IMREADModes` enum, which is defined in OpenCV. Here are a few of the most widely used and self-explanatory entries:

- `IMREAD_UNCHANGED`
- `IMREAD_GRAYSCALE`
- `IMREAD_COLOR`
- `IMREAD_IGNORE_ORIENTATION`

For example, the following code can be used to read an image from a disk, without taking the orientation value stored in the EXIF data of the image and also converted to grayscale:

```
Mat image = imread("MyImage.png",
    IMREAD_GRAYSCALE | IMREAD_IGNORE_ORIENTATION);
```

**Exchangeable Image File Format (EXIF)** is a standard for adding tags and additional data (or metadata) to images taken by digital cameras. These tags might include the manufacturer and camera model and the orientation of the camera while the photo was taken. OpenCV is capable of reading certain tags (such as orientation) and interpreting them, or in the case of



*the preceding sample code, ignoring them.*

After the image is read, you can call `empty` to see whether it was read successfully or not. You can also use `channels` to get the number of channels, `depth` to get the depth, `type` to get the type of the image, and so on. Alternatively, you can call the `imshow` function to display it, as we saw previously in this chapter.

Similarly, the `imreadmulti` function can be used to read a multipage image into a vector of `Mat` objects. The obvious difference here is that `imreadmulti` returns a `bool` value that can be checked for successful reading of the pages and fills the `vector<Mat>` object passed to it by reference.

To be able to write an image to a file on disk, you can use the `imwrite` function. `imwrite` takes the file name that will be written to, a `Mat` object, and a `vector` of `int` values containing the write parameters, which can be ignored in the case of default parameters. See the following enums in OpenCV for a complete list of parameters that can be used with the `imwrite` function to alter the behavior of the write process:

- `ImwriteFlags`
- `ImwriteEXRTypeFlags`
- `ImwritePNGFlags`
- `ImwritePAMFlags`

The following is an example code that depicts how to write a `Mat` object into an image file on disk using the `imwrite` function. Note that the format of the image is derived from the extension provided, in this case `png`:

```
bool success = imwrite("c:/my_images/image1.png", image);
cout << (success ?
           "Image was saved successfully!"
       :
           "Image could not be saved!")
<< endl;
```

Besides the `imread` and `imwrite` functions, which are used to read and write images from and into image files on disk, you can also use the `imdecode` and `imencode` functions to read images stored in a buffer in memory or write into them. We'll leave those two functions for you to discover, and move on to the next topic, which is accessing videos using OpenCV.

# Reading and writing videos

OpenCV, using its `videoio` module or, to be precise, using the `VideoCapture` and `VideoWriter` classes, allows us to read and write video files. The obvious difference in the case of videos, is that they contain a consecutive set of images (or better yet, frames) as opposed to just a single image. So, they are usually read and processed or written in a loop that covers the whole or any desired number of the frames in a video. Let's start with example code that shows how to read and play a video using the OpenCV `VideoCapture` class:

```
VideoCapture vid("MyVideo.mov");
// check if video file was opened correctly
if(!vid.isOpened())
{
    cout << "Can't read the video file";
    return -1;
}
// get frame rate per second of the video file
double fps = vid.get(CAP_PROP_FPS);
if(fps == 0)
{
    cout << "Can't get video FPS";
    return -1;
}
// required delay between frames in milliseconds
int delay_ms = 1000.0 / fps;
// infinite loop
while(true)
{
    Mat frame;
    vid >> frame;
    if(frame.empty())
        break;
    // process the frame if necessary ...
    // display the frame
    imshow("Video", frame);
    // stop playing if space is pressed
    if(waitKey(delay_ms) == ' ')
        break;
}
// release the video file
vid.release();
```

As seen in the preceding code, the video file name is passed to the `VideoCapture` class when it is constructed. This automatically leads to the opening of the video file, if it exists and if the format is supported by your computer (and OpenCV). Consequently, you can check whether the video file was successfully opened or not using the `isOpened` function. Right after that, the `get` function of the `VideoCapture`

class is used to retrieve the **framerate per second (FPS)** of the video file that has been opened. `get` is an extremely important function of `VideoCapture` and it allows us to retrieve a wide range of properties of an opened video file. Here are some example parameters that can be provided to the `get` function to get the desired result:

- `CAP_PROP_POS_FRAMES`: 0-based index of the frame to be decoded or captured next
- `CAP_PROP_FRAME_WIDTH`: Width of the frames in the video stream
- `CAP_PROP_FRAME_HEIGHT`: Height of the frames in the video stream
- `CAP_PROP_FPS`: Frame rate of the video
- `CAP_PROP_FRAME_COUNT`: Number of frames in the video file

For a complete list, you can refer to the `VideoCaptureProperties` enum documentation in OpenCV. Back to the preceding example code, after the frame rate is retrieved using the `get` function, it is used to calculate the delay needed in between two frames, so that it is not too fast or slow when played back. Then, inside an infinite loop, the frames are read using the `>>` operator and displayed. Note that this operator is essentially the simplified and convenient way of using the `VideoCapture` functions, such as `read`, `grab`, and `retrieve`. We are already familiar with the `imshow` function and how it's used. `waitKey`, on the other hand, which is used slightly differently from what we saw before, can be used to insert delays and wait for key presses at the same time. In this case, the desired delay (in milliseconds), which was previously calculated, is inserted between displayed frames and if the space key is pressed, the loop will break. The `release` function in the end is pretty much self-explanatory.

Apart from the way we used the `VideoCapture` class and its methods, we can also call its `open` function to open the video file, if we do not want to pass the file name to the constructor, or if the video file is not present at the `VideoCapture` construction time. Another important function of `VideoCapture` is the `set` function. Think of `set` as the exact opposite of the `get` function, in the sense that it allows setting the parameters of `VideoCapture` and the opened video file. Try experimenting with it for yourself with different parameters, mentioned before in the `VideoCaptureProperties` enum.

To be able to write into a video file, you can use the `VideoWriter` class in a very similar way. Here's an example that shows how a `VideoWriter` object is created:

```
| VideoWriter wrt("C:/output.avi",
|   VideoWriter::fourcc('M','J','P','G'),
|   30, Size(1920, 1080));
```

This will create a video file at "C:/output.avi" with a width of 1920 and height of 1080 pixels at 30 frames per second, ready to be filled with frames. But what is `fourcc`? **Four Character Code (FourCC)** is simply a four-byte code of the format (or the codec, to be precise) that will be used to record the video file. In this example, we have used one of the most common FourCC values, but you can check online for a more comprehensive list of FourCC values and their specifications.

After a `videowriter` object is created, you can use the `<<` operator or the `write` function to write an image (of the exact same size as the video) into the video file:

```
| wrt << image;
```

Or you can also use the following code:

```
| vid.write(frame);
```

Finally, you can call the `release` function to make sure the video file is released, and all of your changes are written into it.



*Apart from the aforementioned methods of using the `videocapture` and `videowriter` classes, you can also set the preferred backend used by them. For more information about this, refer to the `VideoCaptureAPIs` enum in the OpenCV documentation. When omitted, which was the case in our examples, the default backend supported by your computer is used.*

# Accessing cameras

OpenCV supports accessing cameras that are available on a system by using the same `VideoCapture` class we used for accessing the video files. The only difference is that instead of passing a file name to the constructor of the `VideoCapture` class or its `open` function, you must provide a *0-based index number* that corresponds to each available camera. For instance, the default webcam on a computer can be accessed and displayed by using the following example code:

```
VideoCapture cam(0);

// check if camera was opened correctly

if(!cam.isOpened())

    return -1;

// infinite loop

while(true)

{

    Mat frame;

    cam >> frame;

    if(frame.empty())

        break;
```

```
// process the frame if necessary ...  
  
// display the frame  
  
imshow("Camera", frame);  
  
// stop camera if space is pressed  
  
if(waitKey(10) == ' ')  
  
    break;  
  
}  
  
cam.release();
```

As you can see, the only difference is in the constructor. This implementation of the `Videocapture` class allows users to treat any type of video source the same way, thus writing almost the exact same code to deal with cameras instead of video files. This is also the case with network feeds, as described in the next section.

# Accessing RTSP and network feeds

OpenCV allows users to read video frames from a network feed or, to be precise, from an RTSP stream located on a network, such as the local network or even the internet. To be able to do this, you need to pass the URL of the RTSP stream to the `VideoCapture` constructor or its `open` function, exactly the same way as if it were a file on a local hard disk. Here's the most common pattern and an example URL that can be used:

```
| rtsp://user:password@website.com/somevideo
```

In this URL, `user` is replaced by the actual username, `password` with the password of that user, and so on. In the event that the network feed does not require a username and password, they can be omitted.

# Mat-like classes

Besides the `Mat` class, OpenCV provides a number of other classes that are quite similar to `Mat`, but different in how and when they are used. Here are the most important `Mat`-like classes that can be used instead of, or together with, the `Mat` class:

- `Mat_`: This is a subclass of the `Mat` class, but it provides a better access method than the `at` function, that is, using `()`.`Mat_` is a template class and obviously needs to be provided with the type of the elements at compile time, something that can be avoided with the `Mat` class itself.
- `Matx`: This is best used with matrices that have a small size, or to be precise, a known and small size at compile time.
- `UMat`: This is a more recent implementation of the `Mat` class, which allows us to use OpenCL for faster matrix operations.



*Using `UMat` can significantly increase the performance of your computer vision applications, but since it is used in exactly the same way as the `Mat` class, we'll ignore it throughout the chapters of this book; however, in practice and especially in real-time computer vision applications, you must always make sure to use classes and functions that are better optimized and more performant, such as `UMat`.*

# Summary

We're through all the crucial topics that are needed to easily take on computer vision algorithms using hands-on examples and real-life scenarios. We started this chapter by learning about OpenCV and its overall structure, including its modules and main building blocks. This helped us gain a perspective of the computer vision library that we'll be working on, but, more importantly, this gave us an overview of what's possible and what to expect when dealing with computer vision algorithms in general. We moved on to learn where and how to get OpenCV and how to install or build it on our own. We also learned how to create, build, and run C++ and Python projects that use the OpenCV library. Then, by learning all about the `Mat` class and dealing with pixels in an image, we learned how to alter and display an image. The final sections of this chapter included everything we need to know about reading and writing images from files saved on disk, whether they are single (or multi-page) images or videos files, and also cameras and network feeds. We finished the chapter by learning about a few other types from the OpenCV Mat family that can help improve our applications.

Now that we are aware of the real nature of an image (that it is, in essence, a matrix), we can start with possible operations on matrices and matrix-like entities. In the next chapter, we're going to learn all about the matrix and array operations that fall into the realm of computer vision. By the end of the next chapter, we'll be able to perform tons of pixel-wise and image-wise operations and transformations using OpenCV.

# Questions

1. Name three Extra OpenCV modules along with their usages.
2. What is the effect of building OpenCV 3 with the `BUILD_opencv_world` flag turned on?
3. Using the ROI pixel-access method described in this chapter, how can we construct a `Mat` class that can access the middle pixel, plus all of its neighboring pixels (the middle nine pixels) in another image?
4. Name another pixel-access method of the `Mat` class besides the ones mentioned in this chapter.
5. Write a program only using the `at` method and a `for` loop, which creates three separate color images, each one containing only one channel of an RGB image read from disk.
6. Using STL-like iterators, calculate the average pixel value of a grayscale image.
7. Write a program using `VideoCapture`, `waitKey`, and `imwrite`, that displays your webcam and saves the visible image when the `S` key is pressed. This program will stop the webcam and exit if the spacebar key is pressed.

# Further reading

- *Computer Vision with OpenCV 3 and Qt5*: <https://www.packtpub.com/application-development/computer-vision-opencv-3-and-qt5>
- *Learn Qt5*: <https://www.packtpub.com/web-development/learn-qt-5>
- *Qt5 Projects*: <https://www.packtpub.com/application-development/qt-5-projects>

# Array and Matrix Operations

Now that we're past the first part of the book and the introductions and fundamental concepts in computer vision, we can start with the computer vision algorithms and functions provided by the OpenCV library, which more or less cover any computer vision topic that comes to mind, with optimized implementations. As we learned in the previous chapters, OpenCV uses a modular structure to categorize the computer vision functionalities contained in it. We'll move forward with the topics in this book with a similar structure in mind, so that the skills learned are related to each other in each chapter, not just from a theoretical point of view, but also from a hands-on perspective.

In the previous chapter, we learned about the relationship between images and matrices, and covered the most crucial functionalities of the `Mat` class, such as constructing it using a given width, height, and type. We also learned how to read images from disk, a network feed, video files, or cameras. During the process, we learned how to access pixels in an image using various methods. We can now start with actual image- and pixel-modification and manipulation functionalities.

In this chapter, we're going to learn about a large number of functions and algorithms that are used to deal with images, either for calculating a value that might be useful in another process or for directly modifying the values of the pixels in an image. Almost all of the algorithms presented in this chapter are based on the fact that images are essentially matrices, and also the fact that matrices are implemented using arrays of data, hence the name of this chapter!

We'll start this chapter by covering the functionalities within the `Mat` class itself, which are few, but quite important when it comes to the creation of initial matrices and so on. Then, we'll move on to learn about a large number of per-element (or element-wise) algorithms. These algorithms, as we'll learn by going through many hands-on examples, apply a specific operation on each individual element of a matrix and they are not concerned with any other element (or pixel). Finally, we'll learn about matrix and array operations that are not element-wise and the result may depend on the whole image or groups of elements. This

will all clear up as we move forward with the algorithms in this chapter. It's important to note that all of the algorithms and functions in this chapter are contained within the core module of OpenCV library.

By the end of this chapter, you'll have a better understanding of the following:

- Operations contained in the `Mat` class
- Element-wise matrix operations
- Matrix and array-wise operations

# Technical requirements

- An IDE to develop C++ or Python applications
- The OpenCV library

Refer to [chapter 2, Getting Started with OpenCV](#), for more information about how to set up a personal computer and make it ready for developing computer vision applications using OpenCV library.

You can use the following URL to download the source codes and examples for this chapter: <https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter03>

# Operations contained in the Mat class

In this section, we're going to go through the set of mathematical and other operations that are contained inside the `Mat` class itself. Even though there is no general pattern of usage for the functions in the `Mat` class, most of them are related to creating a new matrix, whether it is using an existing one or from scratch. So, let's start.



*During the course of the book, the words `image`, `matrix`, the `Mat` class, and so on will be used interchangeably, and they all mean the same thing, unless explicitly stated. Take this chance to get used to the idea of thinking about images in terms of matrices, just like a computer vision expert would do.*

# Cloning a matrix

You can use `Mat::clone` to create a completely independent clone of a `Mat` object. Note that this function creates a full copy of the image with its own allocated space in memory. Here's how it's used:

```
| Mat clone = image.clone();
```

You can also use the `copyTo` function to do the same, as seen here:

```
| Mat clone;  
| image.copyTo(clone);
```

In both of the preceding code examples, `image` is the original matrix (or image) that is read from an image, camera, or produced in any possible way prior to performing the cloning operation. From now on, and in all of the examples in this and upcoming chapters, `image` is simply a `Mat` object that is the source of our operations, unless otherwise stated.

# Calculating the cross-product

You can use `Mat::cross` to calculate the cross-product of two `Mat` objects that have three floating-point elements, as seen in the following example: Mat A(1, 1, CV\_32FC3), B(1, 1, CV\_32FC3);

A.at<Vec3f>(0, 0)[0] = 0; A.at<Vec3f>(0, 0)[1] = 1; A.at<Vec3f>(0, 0)[2] = 2;

B.at<Vec3f>(0, 0)[0] = 3; B.at<Vec3f>(0, 0)[1] = 4; B.at<Vec3f>(0, 0)[2] = 5;

Mat AxB = A.cross(B);

Mat BxA = B.cross(A);

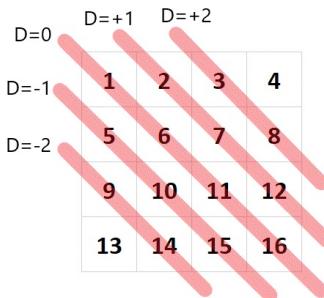
Obviously, `AxB` is not the same as `BxA` in the cross-product of two vectors.

# Extracting a diagonal

`Mat::diag` can be used to extract a diagonal from a `Mat` object, as seen in the following example:

```
| int D = 0; // or +1, +2, -1, -2 and so on  
| Mat dg = image.diag(D);
```

This function accepts an index parameter that can be used to extract additional diagonals beside the main diagonal, as depicted in the following diagram:



If **D=0**, then the extracted diagonal will contain **1, 6, 11, and 16**, which is the main diagonal. But depending on the value of **D**, the extracted diagonal will be above or below the main diagonal, as seen in the preceding diagram.

# Calculating the dot product

To calculate the dot product, or scalar product, or inner product of two matrices, you can use the `Mat::dot` function, as seen here:

```
|     double result = A.dot(B);
```

In this, `A` and `B` are both OpenCV `Mat` objects.

# Learning about the identity matrix

Identity matrices are created using the `Mat::eye` function in OpenCV. Here's an example:

```
| Mat id = Mat::eye(10, 10, CV_32F);
```

In case you need a value other than the ones in the identity matrix diagonal, you can use a `scale` parameter:

```
| double scale = 0.25;
| Mat id = Mat::eye(10, 10, CV_32F) * scale;
```

Another way of creating an identity matrix is by using the `setIdentity` function. Make sure to check the OpenCV documentation for more information about this function.

# Matrix inversion

You can use the `Mat::inv` function to invert a matrix:

```
| Mat inverted = m.inv();
```

Note that you can supply a matrix decomposition type to the `inv` function, which can be an entry in the `cv::DecompTypes` enum.

# Element-wise matrix multiplication

`Mat::mul` can be used to perform an element-wise multiplication of two `Mat` objects. Needless to say, this same function can also be used for element-wise division. Here's an example:

```
|   Mat result = A.mul(B);
```

You can also supply an additional `scale` parameter that will be used to scale the result. Here's another example:

```
double scale = 0.75;  
  
|   Mat result = A.mul(B, scale);
```

# The ones and zeroes matrix

`Mat::ones` and `Mat::zeroes` can be used to create a matrix of a given size with all of its elements set to one or zero, respectively. These matrices are usually used to create initializer matrices. Here are some examples:

```
| Mat m1 = Mat::zeroes(240, 320, CV_8UC1);  
|  
| Mat m2 = Mat::ones(240, 320, CV_8UC1);
```

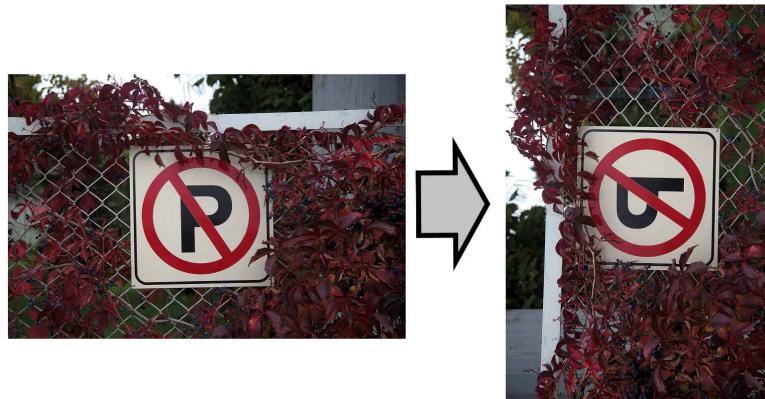
In case you need to create a matrix that is filled with a value other than ones, you can use something like the following:

```
| Mat white = Mat::ones(240, 320, CV_8UC1) * 255;
```

# Transposing a matrix

You can use `Mat::t` to transpose a matrix. Here's an example: `Mat transpose = image.t();`

Here's an example that demonstrates the transposition of an image:



The image on the left is the original image and the image on the right is the transpose of the original image. As you can see, the result is a rotated (clockwise) and flipped (vertically) version of the original image. As you'll learn later in this chapter, `transpose`, or the `Mat::t` function, can be used in conjunction with the `flip` function to rotate or flip/mirror an image in all possible directions.



*Transposing a transpose matrix is the same as the original matrix, and running the following code will result in the original image itself:*

```
Mat org = image.t().t();
```

Another way of calculating the transpose of a matrix is by using the `transpose` function. Here's an example of how this function is used: `transpose(mat, trp);`

In which `mat` and `trp` are both `Mat` objects.

# Reshaping a Mat object

A `Mat` object can be reshaped using the `Mat::reshape` function. Note that reshaping, in this sense, means changing the number of channels and rows of an image. Here's an example: `int ch = 1; int rows = 200; Mat rshpd = image.reshape(ch, rows);`

Note that passing a value of zero as the number of channels means that the number of channels will stay the same as the source. Similarly, passing a value of zero as the number of rows means the number of rows in the image will stay the same.

Note that `Mat::resize` is another useful function for reshaping a matrix, but it only allows changing the number of rows in an image. Another function that can come in handy when reshaping matrices or dealing with the number of elements in a matrix, is the `Mat::total` function, which returns the total number of elements in an image.

That's about it for the functionalities embedded into the `Mat` class itself. Make sure to go through the `Mat` class documentation and familiarize yourself with possible variants of the methods you just learned in this section.

# **Element-wise matrix operations**

Per-element or element-wise matrix operations are mathematical functions and algorithms in computer vision that work on individual elements of a matrix or, in other words, pixels of an image. It's important to note that element-wise operations can be parallelized, which fundamentally means that the order in which the elements of a matrix are processed is not important. This specification is the most important feature that separates the functions and algorithms in this section and the upcoming sections of this chapter.

# **Basic operations**

OpenCV provides all the necessary functions and overloaded operators that you need to perform all four basic operations of addition, subtraction, multiplication, and division between two matrices or a matrix and a scalar.

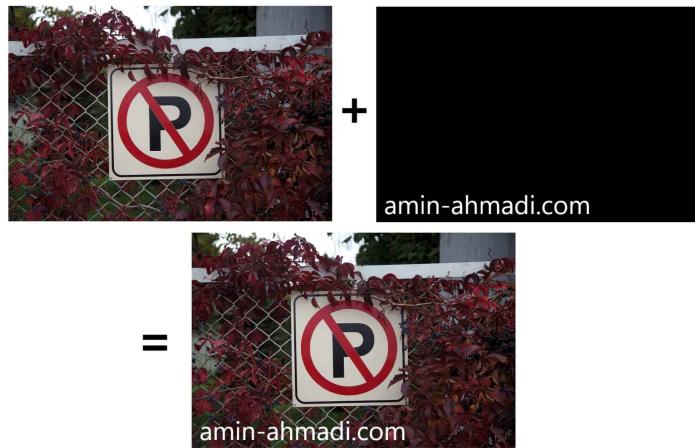
# The addition operation

The `add` function and the `+` operator can be used to add the elements of two matrices, or a matrix and a scalar, as seen in the following examples: `Mat image = imread("Test.png"); Mat overlay = imread("Overlay.png"); Mat result; add(image, overlay, result);`

You can replace the last line in the preceding code with the following:

```
|     result = image + overlay;
```

The following image demonstrates the resulting image of an add operation of two images:



In case, you want to add a single scalar value to all elements of a `Mat` object, you can simply use something similar to the following: `result = image + 80;`

If the preceding code is executed on a grayscale image, the result will be brighter than the source image. Note that if the image has three channels, you must use a three-item vector instead of a single value. For instance, to be able to make an RGB image brighter, you can use the following: `result = image + Vec3b(80, 80, 80);`

Here's an image depicting the brighter result image, when the preceding code is executed on it:



In the preceding example codes, simply increase the added value to get an even brighter image.

# Weighted addition

Besides the simple addition of two images, you can also use the weighted addition function to consider a weight for each of the two images that are being added. Think of it as setting an opacity level for each of the participants in an `add` operation. To perform a weighted addition, you can use the `addWeighted` function:

```
double alpha = 1.0; // First image weight double beta = 0.30; // Second image weight double gamma = 0.0; // Added to the sum addWeighted(image, alpha, overlay, beta, gamma, result);
```

If executed on the sample pictures from the previous section with the `add` example, the result would be similar to the following:



Notice the transparent text that is similar to the watermark usually applied by photo-editing applications. Notice the comments in the code regarding the `alpha`, `beta`, and `gamma` values? Obviously, providing a `beta` value of `1.0` would have made this example exactly the same as a regular `add` function with no transparency for the overlay text.

# The subtraction operation

Similar to adding two `Mat` objects to each other, you can also subtract all elements of one image from another using the `subtract` function or the `-` operator. Here's an example:

```
| Mat image = imread("Test.png");
| Mat overlay = imread("Overlay.png");
| Mat result;
| subtract(image, overlay, result);
```

The last line in the preceding code can also be replaced with this:

```
| result = image - overlay;
```

Here's the result of the subtraction operation if we used the same two images from the previous examples:



Notice how the subtraction of higher pixel values (brighter pixels) from the source image results in the dark color of the overlay text. Also note that the subtraction operation depends on the order of its operands, unlike addition. Try swapping the operands and see what happens for yourself.

Just like addition, it's also possible to multiply a constant number with all of the pixels of an image. You can guess that subtraction of a constant value from all pixels will result in a darker image (depending on the subtracted value) which is the opposite of the addition operation. Here's an example of making an image darker with a simple subtraction operation:

```
|     result = image - 80;
```

In case the source image is a three-channel RGB image, you need to use a vector as the second operand:

```
|     result = image - Vec3b(80, 80, 80);
```

# The multiplication and division operations

Similar to addition and subtraction, you can also multiply all elements of a `Mat` object with all elements of another `Mat` object. The same can be done with the division operation. Again, both operations can be performed with a matrix and a scalar. Multiplication can be done using OpenCV's `multiply` function (similar to the `Mat::mul` function), while division can be performed using the `divide` function.

Here are some examples: `double scale = 1.25; multiply(imageA, imageB, result1, scale); divide(imageA, imageB, result2, scale);`

`scale` in the preceding code is an additional parameter that can be supplied to the `multiply` and `divide` functions to scale all of the elements in the result `Mat` object. You can also perform multiplication or division with a scalar, as seen in the following examples: `resultBrighter = image * 5; resultDarker = image / 5;`

Obviously, the preceding code will produce two images, one that is five times brighter and one that is five times darker than the original image. The important thing to note here is that, unlike addition and subtraction, the resulting image will not be homogeneously brighter or darker, and you'll notice that brighter areas become much brighter and vice versa. The reason for this is obviously the effect of multiplication and division operations, in which the value of brighter pixels grows or drops much faster than smaller values after the operation. It's interesting to note that this same technique is used in most photo-editing applications to brighten or darken the bright areas of an image.

# Bitwise logical operations

Just like basic operations, you can also perform bitwise logical operations on all of the elements of two matrices or a matrix and a scalar. For this reason, you can use the following functions:

- `bitwise_not`
- `bitwise_and`
- `bitwise_or`
- `bitwise_xor`

It's immediately recognizable from their names that these functions can perform `Not`, `And`, `Or`, and `Exclusive OR` operations, but let's see how they're used in detail with some hands-on examples:

First things first, the `bitwise_not` function is used to invert all the bits of all pixels in an image. This function has the same effect as the inversion operation that can be found in most photo editing applications. Here's how it's used:

```
|     bitwise_not(image, result);
```

The preceding code can be replaced with the following too, which uses an overloaded bitwise `not` operator (`-`) in C++:

```
|     result = ~image;
```

If the image is a monochrome black and white image, the result will contain an image with all white pixels replaced with black and vice versa. In case the image is an RGB color image, the result will be inverted (in the sense of its binary pixel values), which is depicted in the following example image:



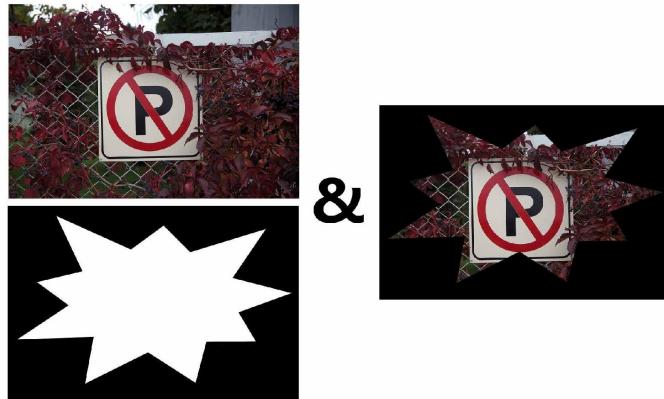
The `bitwise_and` function, or the `&` operator, is used to perform a bitwise `And` operation on pixels from two images or pixels from an image and a scalar. Here is an example:

```
|     bitwise_and(image, mask, result);
```

You can simply use the `&` operator and write the following instead:

```
|     result = image & mask;
```

The `bitwise_and` function can be easily used to mask and extract certain areas in images. For instance, the following image is a demonstration of how `bitwise_and` results in an image that passes the white pixels and removes the black pixels:



Besides masking certain areas of an image, the `bitwise_and` operation can be used to filter out a channel altogether. To be able to do this, you need to use the second form of the `&` operator, which takes a matrix and a scalar and performs the `And` operation between all pixels and that value. Here is an example code that can be used to mask (zero out) the green color channel in an RGB color image:

```
|     result = image & Vec3b(0xFF, 0x00, 0xFF);
```

Now, let's move on to the next bitwise operation, the `or` operation. The `bitwise_or` and `|` operators can both be used to perform a bitwise `or` operation on two images, or an image and a scalar. Here is an example:

```
|     bitwise_or(image, mask, result);
```

Similar to the `bitwise_and` operation, you can use the `|` operator in the `or` operation and simply write the following instead of the preceding code:

```
|     result = image | mask;
```

If the `and` operation was used to pass through the non-zero pixels (or non-black pixels), then it can be said that the `or` operation is used to pass through the pixel with the higher value (or brighter) in any of its input images. Here's the result of performing the bitwise `or` operation on the previous example images:



Similar to `bitwise_and` operation, you can also use `bitwise_or` operation to update an individual channel or all the pixels of an image. Here is an example code that shows how you can update only the green channel in an RGB image to have the maximum possible value (which is 255, or hexadecimal `FF`) in all of its pixels and leave the other channels as they are:

```
|     result = image | Vec3b(0x00, 0xFF, 0x00);
```

Finally, you can use `bitwise_xor`, or the `^` operator to perform an `Exclusive Or` between the pixels of two images, or an image and a scalar. Here is an example:

```
|     bitwise_xor(image, mask, result);
```

Or simply use the `^` operator and write the following instead:

```
|     result = image ^ mask;
```

Here is the resulting image, if the `Exclusive or` operation is performed on the example image from the preceding section:



Notice how this operation leads to the inversion of the pixels in the masked area? Think about the reason behind this by writing down the pixel values on a paper and trying to calculate the result by yourself. `Exclusive or`, and all bitwise operations, can be used for many other computer vision tasks if their behavior is clearly understood.

# The comparison operations

Comparing two images with each other (or given values) can be quite useful, especially for producing masks that can be used in various other algorithms, whether it is for tracking some object of interest in an image or performing an operation on an isolated (masked) region of an image. OpenCV provides a handful of functions to perform element-wise comparisons. For instance, the `compare` function can be used to compare two images with each other. Here's how:

```
|     compare(image1, image2, result, CMP_EQ);
```

The first two parameters are the first and second image that will participate in the comparison. The `result` will be saved into a third `Mat` object, and the last parameter, which must be an entry from the `CmpTypes` enum, is used to select the type of comparison, which can be any of the following:

- `CMP_EQ`: Means that the first image equals the second image
- `CMP_GT`: Means that the first image is greater than the second image
- `CMP_GE`: Means that the first image is greater than or equal to the second image
- `CMP_LT`: Means that the first image is less than the second image
- `CMP_LE`: Means that the first image is less than or equal to the second image
- `CMP_NE`: Means that the first image is not equal to the second image



*Note that we are still talking about element-wise operations, so when we say first image is less than or equal to the second image, what we actually mean is each individual pixel in the first image contains a value less than or equal to its exact corresponding pixel in the second image, and so on.*

Note that you can also use the overloaded C++ operators to achieve the same goal as the `compare` function. Here is how, for each individual comparison type:

```
result = image1 == image2; // CMP_EQ
result = image1 > image2; // CMP_GT
result = image1 >= image2; // CMP_GE
result = image1 < image2; // CMP_LT
result = image1 <= image2; // CMP_LE
result = image1 != image2; // CMP_NE
```

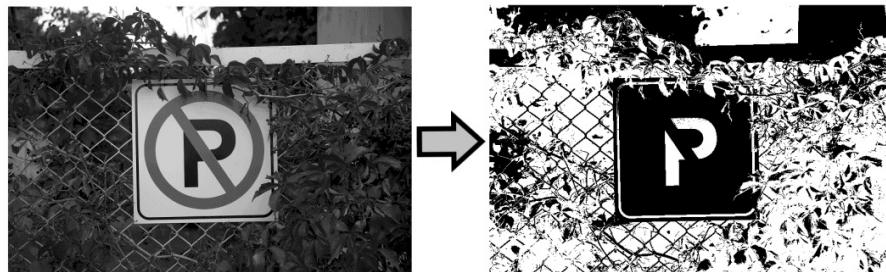
The `inRange` function, which is another useful comparison function in OpenCV,

can be used to find the pixels that have a value between a certain lower bound, and higher bound values. You can use any existing image as the boundary value matrices, or you can create them by yourself. Here's an example code that can be used to find the pixel values between 0 and 50 in a grayscale image:

```
Mat lb = Mat::zeros(image.rows,
                     image.cols,
                     image.type());
Mat hb = Mat::ones(image.rows,
                    image.cols,
                    image.type()) * 50;
inRange(image, lb, hb, result);
```

Note that `lb` and `hb` are both `Mat` objects of the same size and type of source image, except `lb` is filled with zeroes and `hb` is filled with the value of 50. This way, when `inRange` is called, it checks each individual pixel in the source image with their corresponding pixels in `lb` and `hb` and sets the corresponding pixel in the result to white if the value fits between the provided boundaries.

The following image depicts the result of the `inRange` function when it is executed on our example image:



The `min` and `max` functions, as it can be easily guessed from their names, are two other comparison functions that can be used to compare two images (element-wise) and find out the minimum or maximum pixel values. Here's an example:

```
| min(image1, image2, result);
```

Or you can use `max` to find the maximum:

```
| max(image1, image2, result);
```

Simply put, these two functions compare the pixels of two images of the same size and type and set the corresponding pixel in the result matrix to the minimum or maximum pixel value from the input images.

# The mathematical operations

Besides the functions we've learned about so far, OpenCV also provides a number of functions to deal with element-wise mathematical operations. In this section, we're going to go through them briefly, but you can, and you should experiment with them by yourself to make sure that you're familiar with them, and you can comfortably use them in your projects.

The element-wise mathematical functions in OpenCV are as follows:

- The `absdiff` function can be used to calculate the absolute difference between the pixels of two images of the same size and type or an image and a scalar. Here's an example:

```
|     absdiff(image1, image2, result);
```

In the preceding code, `image1`, `image2`, and `result` are all `Mat` objects, and each element in the `result` represents the absolute difference between corresponding pixels in `image1` and `image2`.

- The `exp` function can be used to calculate the exponential of all elements in a matrix:

```
|     exp(mat, result);
```

- The `log` function can be used to calculate the natural logarithm of every element in a matrix:

```
|     log(mat, result);
```

- `pow` can be used to raise all of the elements in a matrix to a given power. This function requires a matrix and a `double` value that will be the power value. Here's an example:

```
|     pow(mat, 3.0, result);
```

- The `sqr` function is used to calculate the square root of all elements in a

matrix, and it's used as follows:

|     sqrt(mat, result);

 *Functions such as `log` and `pow` should not be confused with the functions of the same name in standard C++ libraries. For better readability of your code, consider using the `cv` namespace before the function name in your C++ code. For instance, you can call the `pow` function as follows:*

*cv::pow(image1, 3.0, result);*

# Matrix and array-wise operations

Unlike the functions and algorithms that we've seen so far in this chapter, the algorithms in this section perform an atomic and complete operation on the image (or matrix) itself and are not considered element-wise operations in the sense that has been described so far. If you recall, the rule of thumb for element-wise operations was that they can be easily parallelized since the resulting matrix is dependent on corresponding pixels of two images, whereas the functions and algorithms we'll learn about in this chapter are not easily parallelizable, or the resulting pixels and values might have little or nothing at all to do with their corresponding source pixels, or quite the opposite, the resulting pixels might depend on some or all of the input pixels at the same time.

# Making borders for extrapolation

As you'll see in this and upcoming chapters, one of the most important issues to handle when dealing with many computer vision algorithms is extrapolation, or to put it simply, the assumption of the non-existing pixels outside of an image. You might be wondering, why would I need to think about the non-existing pixels, and the simplest answer is that there are many computer vision algorithms that work with not just a single pixel, but also with their surrounding pixels. In such cases, when the pixel is in the middle of an image, there are no issues. But for the pixels at the borders of the image (for instance, in the topmost row), some of the surrounding pixels will fall outside of the image. That is exactly where you need to think about extrapolation and the assumption of non-existing pixels. Are you going to simply assume those pixels have zero values? Maybe it's better to assume they have the same value as the border pixels? These questions are all taken care of in an OpenCV function called `copyMakeBorder`.

`copyMakeBorder` allows us to form borders outside an image and provides enough customizations to deal with all possible scenarios. Let's see how `copyMakeBorder` is used with a couple of simple examples:

```
int top = 50;
int bottom = 50;
int left = 50;
int right = 50;
BorderTypes border = BORDER_REPLICATE;
copyMakeBorder(image,
               result,
               top,
               bottom,
               left,
               right,
               border);
```

As seen in the preceding example, `copyMakeBorder` accepts an input image and produces a `result` image, just like most of the OpenCV functions we've learned about so far. In addition, this function must be provided with four integer values that represent the number of pixels that are added to the `top`, `bottom`, `left`, and `right` side of the image. However, the most important parameter that must be provided here is the `border` type parameter, which must be an entry of the `BorderTypes` enum. Here are a few of the most commonly used `BorderType` values:

- BORDER\_CONSTANT
- BORDER\_REPLICATE
- BORDER\_REFLECT
- BORDER\_WRAP

Note that when `BORDER_CONSTANT` is used as the border type parameter, an additional scalar parameter must be provided to the `copyMakeBorder` function, which represents the constant color value of the created border. If this value is omitted, zero (or black) is assumed. The following image demonstrates the output of the `copyMakeBorder` function when it is executed on our example image:



`copyMakeBorder`, and many other OpenCV functions, use the `borderInterpolate` function, internally, to calculate the position of the donor pixel used for the extrapolation and creating the non-existing pixels. You won't need to call this function directly yourself, so we'll leave it for you to explore and discover on your own.

# Flipping (mirroring) and rotating images

You can use the `flip` function to flip or mirror an image. This function can be used to flip an image around the `x` or `y` axes, or both, depending on the provided `flip code`. Here's how this function is used: `int code = +1; flip(image, result, code);`

If `code` is zero, the input image will be flipped/mirrored vertically (around the `x` axis), if `code` is a positive value, the input image will be flipped/mirrored horizontally (around the `y` axis), and if `code` is a negative value, the input image will be flipped/mirrored around both the `x` and `y` axes at the same time.

On the other hand, to rotate an image, you can use the `rotate` function. What you need to take care of, when calling the `rotate` function, is providing a correct rotation flag, as seen in the following example: `RotateFlags rt = ROTATE_90_CLOCKWISE; rotate(image, result, rt);`

The `RotateFlag` enum can be one of the following self-explanatory constant values:

- `ROTATE_90_CLOCKWISE`
- `ROTATE_180`
- `ROTATE_90_COUNTERCLOCKWISE`

Here's an image that depicts all possible results of the `flip` and `rotate` functions. Notice that the result of flip around both axes is the same as a 180-degree rotation in the following resulting images:



**i** As mentioned previously in this chapter, `Mat::t`, or the transpose of a matrix, can also be used to rotate an image when used in conjunction with the `flip` function.

# Working with channels

OpenCV provides a handful of functions to deal with channels, whether we need to merge, split, or perform various actions on them. In this section, we're going to learn how to split an image to its consisting channels or make a multi-channel image using multiple single-channel images. So, let's start.

You can use the `merge` function to merge a number of single-channel `Mat` objects and create a new multi-channel `Mat` object, as seen in the following sample code:

```
| Mat channels[3] = {ch1, ch2, ch3};  
| merge(channels, 3, result);
```

In the preceding code, `ch1`, `ch2`, and `ch3` are all single-channel images of the same size. The result will be a three-channel `Mat` object.

You can also use the `insertChannel` function to insert a new channel into an image. Here's how:

```
| int idx = 2;  
| insertChannel(ch, image, idx);
```

In the preceding code, `ch` is a single channel `Mat` object, `image` is the matrix we want to add an additional channel into it, and `idx` refers to the zero-based index number of the position where the channel will be inserted.

The `split` function can be used to perform exactly the opposite of the `merge` function, which is splitting a multi-channel image to create a number of single-channel images. Here's an example:

```
| Mat channels[3];  
| split(image, channels);
```

The `channels` array in the preceding code will contain three single-channel images of the same size that correspond to each individual channel in the image.

To be able to extract a single channel from an image, you can use the `extractChannel` function:

```
| int idx = 2;
```

```
| Mat ch;  
| extractChannel(image, ch, idx);
```

In the preceding code, quite obviously, the channel at position `idx` will be extracted from the image and saved in `ch`.

Even though `merge`, `split`, `insertChannel`, and `extractChannel` are enough for most use cases, you still might need a more complex shuffling, extraction, or manipulation of the channels in an image. For this reason, OpenCV provides a function called `mixChannels`, which allows a much more advanced way of handling channels. Let's see how `mixChannels` is used with an example case. Let's say we want to shift all of the channels of an image to the right. To be able to perform such a task, we can use the following sample code:

```
| Mat image = imread("Test.png");  
| Mat result(image.rows, image.cols, image.type());  
  
| vector<int> fromTo = {0,1,  
|                         1,2,  
|                         2,0};  
| mixChannels(image, result, fromTo);
```

The only important piece of code in the preceding example is the `fromTo` vector, which must contain pairs of values that correspond to the channel number in the source and destination. The channels are, as always, 0-index based, so 0, 1 means the first channel in the source image will be copied to the second channel in the result and so on.



*It's worth noting that all previous functions in this section (`merge`, `split`, and so on), are a partial case of the `mixChannels` function.*

# Mathematical functions

The functions and algorithms that we'll learn about in this section are solely used for mathematical calculations of a non-element-wise nature, as opposed to what we saw earlier on in this chapter. They include simple functions, such as `mean` or `sum` or more complex operations, such as the discrete Fourier transform. Let's browse through some of the most important ones with hands-on examples.

# Matrix inversion

The `invert` function can be used to calculate the inverse of a matrix. This should not be confused with the `bitwise_not` function, which inverts each and every bit in the pixels of an image. The `invert` function is not an element-wise function, and the inversion method needs to be provided as a parameter to this function. Here's an example:

```
DecompTypes dt = DECOMP_LU;  
  
invert(image, result, dt);
```

The `DecompTypes` enum contains all possible entries that can be used in the `invert` function as the decomposition type. Here they are:

- `DECOMP_LU`
- `DECOMP_SVD`
- `DECOMP_EIG`
- `DECOMP_CHOLESKY`
- `DECOMP_QR`

Refer to the OpenCV documentation for the `DecompTypes` enum if you're interested in a detailed description of each and every decomposition method.

# Mean and sum of elements

You can calculate the mean, or the average value of the elements in a matrix, by using the `mean` function. Here is an example that shows how to read an image and calculate and display the mean value of all of its individual channels: Mat image = imread("Test.png"); Mat result; Scalar m = mean(image); cout << m[0] << endl; cout << m[1] << endl; cout << m[2] << endl;

You can use the `sum` function exactly the same way, to calculate the sum of elements in a matrix:

```
|   Scalar s = sum(image);
```

OpenCV also includes a `meanStdDev` function that can be used to calculate the mean and standard deviation of all elements in the matrix at the same time. Here is an example:

```
|   Scalar m;  
|   Scalar stdDev;  
|   meanStdDev(image, m, stdDev);
```

Similar to the `mean` function, the `meanStdDev` function also calculates the results for each individual channel separately.

# Discrete Fourier transformation

A discrete Fourier transformation of a 1D or 2D array, or in other words—an image, is one of the many ways to analyze an image in computer vision. The interpretation of the result depends completely on the field it is being used for and that is not what we are concerned with in this book, however, how to perform a discrete Fourier transformation is what we're going to learn in this section.

Simply put, you can use the `dft` function to calculate the Fourier transformation of an image. However, there are some preparations needed before the `dft` function can be safely called. The same applies to the result of the Fourier transformation. Let's break this down with an example code, and by calculating and displaying the Fourier transformation of our example image used in the previous sections.

The `dft` function can process matrices of certain sizes (powers of 2, such as 2, 4, and 8) a lot more efficiently, that is why it is always best to increase the size of our matrix to the nearest optimum size and pad it with zeroes before calling the `dft` function. This can be done by using the `getOptimalDFTSize` function. Assuming that `image` is the input image that we want to calculate its discrete Fourier transformation, we can write the following code to calculate and resize it to the optimum size for the `dft` function:

```
int optRows = getOptimalDFTSize( image.rows );
int optCols = getOptimalDFTSize( image.cols );

Mat resizedImg;
copyMakeBorder(image,
               resizedImg,
               0,
               optRows - image.rows,
               0,
               optCols - image.cols,
               BORDER_CONSTANT,
               Scalar::all(0));
```

As you can see, the `getOptimalDFTSize` function must be called twice, for rows and columns separately. You are already familiar with the `copyMakeBorder` function. Resizing an image and padding the new pixels with zeroes (or any other desired

value) is one of countless example use cases for the `copyMakeBorder` function.

The rest is quite easy, we need to form a two-channel image and pass it to the `dft` function and get the complex (real and imaginary) result in the same matrix. This will later simplify the process of displaying the result. Here's how it's done:

```
vector<Mat> channels = {Mat<float>(resizedImg),  
                         Mat::zeros(resizedImg.size(), CV_32F)};  
  
Mat complexImage;  
merge(channels, complexImage);  
  
dft(complexImage, complexImage);
```

We have already learned how to use the `merge` function. The only important thing to note in the preceding code is the fact that the result is saved into the same image as the input. `complexImage` now contains two channels, one for the real and the other for the imaginary part of the discrete Fourier transformation. That's it! We now have our result, however, to be able to display it, we must calculate the magnitude of the result. Here's how it's done:

```
split(complexImage, channels);  
  
Mat mag;  
magnitude(channels[0], channels[1], mag);
```

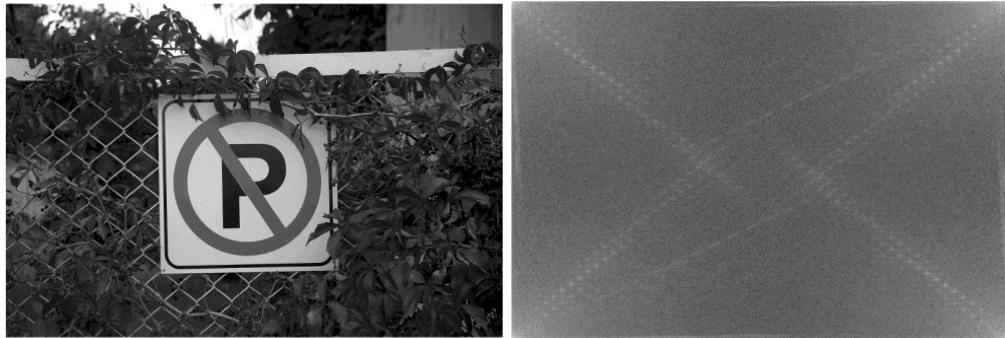
In the preceding code, we have split the complex result into its consisting channels and then calculated the magnitude using the `magnitude` function. Theoretically, `mag` is a displayable result, but in reality, it contains values much higher than what is displayable using OpenCV, so we need to perform a couple of conversions before being able to display it. First, we need to make sure the result is in the logarithmic scale, by performing the following conversion:

```
mag += Scalar::all(1);  
log(mag, mag);
```

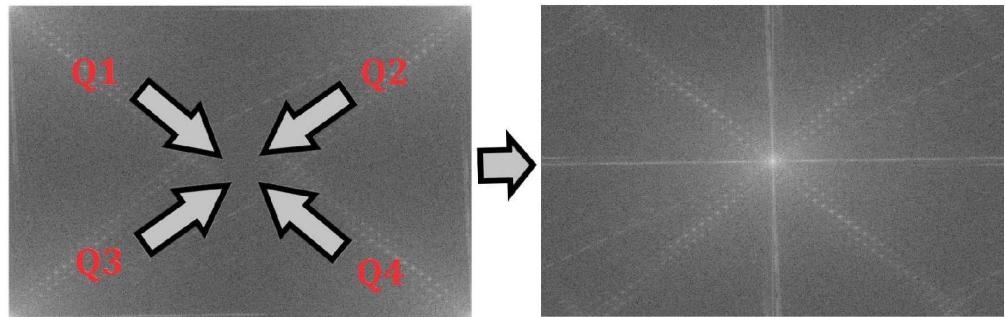
Next, we must make sure the result values are scaled and normalized to fit between `0.0` and `1.0`, to be displayable by the `imshow` function in OpenCV. You need to use the `normalize` function for this reason:

```
normalize(mag, mag, 0.0, 1.0, CV_MINMAX);
```

You can now try to display the result using the `imshow` function. Here's an example that displays the result of the discrete Fourier transformation:



The problem with this result is that the quadrants need to be swapped before the origin of the result is at the center of the image. The following image depicts how the four quadrants of the result must be swapped before the result has an origin point at the center:



The following is the code needed to swap the four quadrants of the Fourier transform result. Notice how we first find the center of the result, then create four **region of interest (ROI)** matrices, and then swap them:

```

int cx = mag.cols/2;
int cy = mag.rows/2;

Mat Q1(mag, Rect(0, 0, cx, cy));
Mat Q2(mag, Rect(cx, 0, cx, cy));
Mat Q3(mag, Rect(0, cy, cx, cy));
Mat Q4(mag, Rect(cx, cy, cx, cy));

Mat tmp;
Q1.copyTo(tmp);
Q4.copyTo(Q1);
tmp.copyTo(Q4);

Q2.copyTo(tmp);
Q3.copyTo(Q2);
tmp.copyTo(Q3);

```

The `dft` function accepts an additional parameter that can be used to further customize its behavior. This parameter can be a combination of values from the

`DftFlags` enum. For instance, to perform an inverse Fourier transformation, you need to call the `dft` function with the `DFT_INVERSE` parameter:

```
| dft(input, output, DFT_INVERSE);
```

This can also be done by using the `idft` function:

```
| idft(input, output);
```

Make sure to check out the `DftFlags` enum and the `dft` function documentation for more information about how the discrete Fourier transform is implemented in OpenCV.

# Generating random numbers

Random number generation is one of the most widely used algorithms in computer vision, especially when it comes to testing an existing algorithm with random values between given ranges. When using the OpenCV library, you can use the following functions to produce values or matrices containing random values:

- The `randn` function can be used to fill a matrix or an array with random numbers with a given mean value and standard deviation. Here's how this function is used:

```
|     randn(rmat, mean, stddev);
```

- The `randu` function, similar to the `randn` function, is used to fill an array with random values, however, instead of mean and standard deviation, this function uses a lower band and higher band (both inclusive) for the produced random values. Here's an example:

```
|     randu(rmat, lowBand, highBand);
```

- The `randShuffle` function, as it can be guessed from its title, is used to randomly shuffle the contents of an array or matrix. It is used as simply as it is depicted in the following example:

```
|     randShuffle(array);
```

# The search and locate functions

When working on your computer vision projects, you'll be faced with countless scenarios and cases in which you'll need to look for certain pixels, or the maximum values (brightest point), and so on. The OpenCV library contains a number of functions that can be used for this same purpose, and they are the subject of this section.

# Locating non-zero elements

Locating, or counting, non-zero elements can come in quite handy, especially when looking for specific regions in an image after a threshold operation, or when looking for the area covered by a specific color. OpenCV contains the `findNonZero` and `countNonZero` functions, which simply allow you to find or count the pixels with non-zero (or bright) values in an image.

Here is an example that depicts how you can use the `findNonZero` function to find the first non-black pixel in a grayscale image and print its position: Mat image = imread("Test.png", IMREAD\_GRAYSCALE); Mat result;

```
vector<Point> idx;  
  
findNonZero(image, idx);  
  
if(idx.size() > 0)  
  
cout << idx[0].x << "," << idx[0].y << endl;
```

Here is another example code that shows how you can find the percentage of black pixels in a grayscale image: Mat image = imread("Test.png", IMREAD\_GRAYSCALE); Mat result;

```
int nonZero = countNonZero(image); float white = float(nonZero) /  
float(image.total()); float black = 1.0 - white; cout << black << endl;
```

# Locating minimum and maximum elements

Locating the brightest (maximum) and darkest (minimum) points in an image or matrix are two of the most important types of searches within images in computer vision, especially after performing certain types of thresholding algorithms, or template-matching functions (which we'll learn about in the upcoming chapters). OpenCV provides the following two functions to locate global minimum and maximum values, along with their positions in a matrix:

- `minMaxIdx`
- `minMaxLoc`

The `minMaxLoc` function searches the whole image (only single-channel images) for the brightest and darkest points and returns the value of the brightest and darkest pixel, along with their positions, whereas the `minMaxIdx` function returns a pointer to the found minimum and maximum position, instead of a position (the `Point` object with `x` and `y`). Here's how the `minMaxLoc` function is used: double `minVal`, `maxVal`; `Point minLoc, maxLoc`;

```
minMaxLoc(image, &minVal, &maxVal, &minLoc, &maxLoc);
```

This is an example of using the `minMaxIdx` function: double `minVal`, `maxVal`; int `minIdx`, `maxIdx`;

```
minMaxIdx(image, &minVal, &maxVal, &minIdx, &maxIdx);
```

# Lookup table transformation

In computer vision, replacing pixels with new values based on their value from a given table that contains the required replacement is called **lookup table transformation**. This might sound a bit confusing at first but it's a very powerful and simple method for modifying images using lookup tables. Let's see how it's done with a hands-on example.

Let's assume we have an example image and we need to replace the bright pixels (with a value greater than 175) with absolute white, dark pixels (with a value less than 125) with absolute black, and leave the rest of the pixels as they are. To perform such a task, we can simply use a lookup table `Mat` object along with the `LUT` function in OpenCV:

```
Mat lut(1, 256, CV_8UC1);
for(int i=0; i<256; i++)
{
    if(i < 125)
        lut.at<uchar>(0, i) = 0;
    else if(i > 175)
        lut.at<uchar>(0, i) = 255;
    else
        lut.at<uchar>(0, i) = i;
}

Mat result;
LUT(image, lut, result);
```

The following image depicts the result of this lookup table transformation when it is executed on our example image. As you can see, the lookup table transformation can be executed on both color (on the right) and grayscale (on the left) images:



Using the `LUT` function wisely can lead to many creative solutions for computer vision problems where replacing pixels with a given value is desired.

# Summary

What we saw in this chapter was a taste of what can be done using matrix and array operations in computer vision with the OpenCV library. We started the chapter with nothing but a firm background in the fundamental concepts of computer vision from the previous chapters, and we ended up learning about lots of algorithms and functions using hands-on examples. We learned about ones, zeroes, identity matrix, transpose, and other functions embedded into the heart of the `Mat` class. Then, we continued on to learn about many element-wise algorithms. We are now totally capable of performing element-wise matrix operations, such as basic operations, comparisons, and bitwise operations. Finally, we learned about the non-element-wise operations in the OpenCV core module. Making borders, modifying channels, and discrete Fourier transformation were among the many algorithms we learned to use in this chapter.

In the next chapter, we're going to learn about computer vision algorithms used for filtering, drawing, and other functions. Topics covered in the upcoming chapter are essential before you step into the world of advanced computer vision development and algorithms used for highly complex tasks, such as face or object detection and tracking.

# Questions

1. Which of the element-wise mathematical operations and bitwise operations would produce the exact same results?
2. What is the purpose of the `gemm` function in OpenCV? What is the equivalent of `A*B` with the `gemm` function?
3. Use the `borderInterpolate` function to calculate the value of a non-existing pixel at point `(-10, 50)`, with a border type of `BORDER_REPLICATE`. What is the function call required for such a calculation?
4. Create the same identity matrix as in the *Learning about the identity matrix* section of this chapter, but use the `setIdentity` function instead of the `Mat::eye` function.
5. Write a program using the `LUT` function (lookup table transformation) that performs the same task as `bitwise_not` (invert colors) when executed on grayscale and color (RGB) images.
6. Besides normalizing the values of a matrix, the `normalize` function can be used to brighten or darken images. Write the required function call to darken and brighten a grayscale image using the `normalize` function.
7. Remove the blue channel (first channel) from an image (BGR image created using the `imread` function) using the `merge` and `split` functions.

# Drawing, Filtering, and Transformation

We started the previous chapter with nothing but the basics and fundamental concepts used in computer vision, and we ended up learning about many algorithms and functions used to perform a wide range of operations on matrices and images. First, we learned all about the functions that are embedded into the `Mat` class for convenience, such as cloning (or getting a full and independent copy) a matrix, calculating the cross and the dot product of two matrices, getting the transpose or inverse of a matrix, and producing an identity matrix. Then we moved on to learn about various element-wise operations in OpenCV. Element-wise operations, as we already know, are parallelizable algorithms that perform the same process on all individual pixels (or elements) of an image. During the process, we also experimented with the effect of such operations on actual image files. We completed the previous chapter by learning about operations and functions that treat images as a whole, unlike element-wise operations, but they are still considered matrix operations in terms of computer vision.

Now, we're ready to dig even deeper and learn about numerous powerful algorithms that are used in computer vision applications for tasks such as drawing, filtering, and the transformation of images. As mentioned in the previous chapters, these categories of algorithms as a whole are considered image-processing algorithms. In this chapter, we're going to start by learning about drawing shapes and text on empty images (similar to canvases) or existing images and video frames. The examples in the first section of this chapter will also include a tutorial about adding trackbars to OpenCV windows, for easily adjusting required parameters. After that, we'll move on to learn about image filtering techniques, such as blurring an image, dilation, and erosion. The filtering algorithms and functions that we'll learn about in this chapter include many popular and widely used algorithms, especially by professional photo-editing applications. This chapter will include a comprehensive section about image-transformation algorithms, including algorithms such as the simple

resizing of photos, or complex algorithms such as remapping pixels. We'll end this chapter by learning how to apply colormaps to images to transform their colors.

In this chapter, we'll learn about the following:

- Drawing shapes and text on images
- Applying smoothening filters to images
- Applying dilation, erosion, and various other filters to images
- Remapping pixels and performing geometric transformation algorithms on images
- Applying colormaps to images

# Technical requirements

- An IDE to develop C++ or Python applications
- The OpenCV library

Refer to [chapter 2, \*Getting Started with OpenCV\*](#), for more information about how to set up a personal computer and make it ready for developing computer vision applications using the OpenCV library.

You can use the following URL to download the source codes and examples for this chapter:

<https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter04>

# Drawing on images

Without a doubt, one of the most important tasks when developing computer vision applications is drawing on images. Imagine you want to print the timestamp on pictures or draw a rectangle or ellipse around some areas in an image, and many similar examples that would require you to draw text and digits on images or shapes (rectangles and so on). As you can see, the examples that can be pointed out are quite obvious and countless, so without further ado, let's start with the functions and algorithms in OpenCV that can be used for drawing.

# Printing text on images

OpenCV contains a very easy-to-use function named `putText` to draw, or print, text on images. This function requires an image as the input/output parameter, which means the source image itself will be updated. So, be sure to make a copy of your original image in memory before calling this function. You also need to provide this function with an origin point, which is simply the point where the text will be printed. The font of the text must be one of the entries in the `HersheyFonts` enum, which can take one (or a combination) of the following values:

- `FONT_HERSHEY_SIMPLEX`
- `FONT_HERSHEY_PLAIN`
- `FONT_HERSHEY_DUPLEX`
- `FONT_HERSHEY_COMPLEX`
- `FONT_HERSHEY_TRIPLEX`
- `FONT_HERSHEY_COMPLEX_SMALL`
- `FONT_HERSHEY_SCRIPT_SIMPLEX`
- `FONT_HERSHEY_SCRIPT_COMPLEX`
- `FONT_ITALIC`

For details of how each entry looks when printed, you can check out OpenCV or simply search online for more information about Hershey fonts.

Apart from the parameters we just mentioned, you also need a few additional parameters, such as the scale, color, thickness, and line type of the text. Let's break them all down with a simple example.

Here's an example code that demonstrates the usage of the `putText` function:

```
string text = "www.amin-ahmadi.com";
int offset = 25;
Point origin(offset, image.rows - offset);
HersheyFonts fontFace = FONT_HERSHEY_COMPLEX;
double fontScale = 1.5;
Scalar color(0, 242, 255);
int thickness = 2;
LineTypes lineType = LINE_AA;
bool bottomLeftOrigin = false;

putText(image,
        text,
```

```
origin,  
fontFace,  
fontScale,  
color,  
thickness,  
lineType,  
bottomLeftOrigin);
```

When executed on our example picture from the previous chapters, the following result will be created:



Obviously, increasing or decreasing `scale` will result in an increased or decreased text size. The `thickness` parameter corresponds to the thickness of the printed text and so on. The only parameter that is worth discussing more is `lineType`, which was `LINE_AA` in our example, but it can take any value from the `LineTypes` enum. Here are the most important line types along with their differences, demonstrated with a printed `w` character on a white background:



`LINE_4` stands for four-connected line types, and `LINE_8` stands for eight-connected line types. `LINE_AA` though, which is the anti-aliased line type, is slower to draw than the other two, but, as can be seen in the preceding diagram, it also provides a much better quality.

 *The `LineTypes` enum also includes a `FILLED` entry, which is used for filling the shapes drawn on*



*an image with the given color. It's important to note that almost all drawing functions in OpenCV (not just `putText`) require a line type parameter.*

OpenCV provides two more functions that are related to handling texts, but not exactly for drawing them. The first one is called `getFontSizeFromHeight`, which is used to get the required scale value based on a font type, height (in pixels), and thickness. Here's an example:

```
double fontScale = getFontSizeFromHeight(fontFace,  
                                         50, // pixels for height  
                                         thickness);
```

We could have used the preceding code instead of providing a constant value for `scale` in our previous example usage of the `putText` function. Obviously, we need to replace `50` with any desired pixel height value for our text.

Besides `getFontSizeFromHeight`, OpenCV also includes a function named `getTextSize` that can be used to retrieve the width and height that is required for printing a specific text on an image. Here's an example code that shows how we can find out the width and height in pixels required for printing the "Example" word with the `FONT_HERSHEY_PLAIN` font type, a scale of `3.2`, and a thickness of `2` on an image using the `getTextSize` function:

```
int baseLine;  
Size size = getTextSize("Example",  
                      FONT_HERSHEY_PLAIN,  
                      3.2,  
                      2,  
                      &baseLine);  
  
cout << "Size = " << size.width << " , " << size.height << endl;  
cout << "Baseline = " << baseLine << endl;
```

The results should look like the following:

```
Size = 216 , 30  
Baseline = 17
```

This means the text will need `216` by `30` pixels of space to be printed and the baseline will be `17` pixels farther from the bottom of the text.

# Drawing shapes

You can use a set of very simple OpenCV functions to draw various types of shapes on images. These functions are all included in the `imgproc` module, just like the `putText` function, and they can be used to draw markers, lines, arrowed lines, rectangles, ellipses and circles, polylines, and so on in given points.

Let's start with the `drawMarker` function, which is used to draw a marker with a given type on an image. Here's how this function is used to print a marker at the center of a given image:

```
Point position(image.cols/2,
               image.rows/2);
Scalar color = Scalar::all(0);
MarkerTypes markerType = MARKER_CROSS;
int markerSize = 25;
int thickness = 2;
int lineType = LINE_AA;
drawMarker(image,
           position,
           color,
           markerType,
           markerSize,
           thickness,
           lineType);
```

`position` is the central point of the marker, and the rest of the parameters are almost exactly what we saw in the `putText` function previously. This is the same pattern of parameters for most (if not all) of the drawing functions in OpenCV. The only parameters that are specific to the `drawMarker` function are `markerSize`, which is simply the size of the marker, and `markerType`, which can take one of the following values from the `MarkerTypes` enum:

- `MARKER_CROSS`
- `MARKER_TILTED_CROSS`
- `MARKER_STAR`
- `MARKER_DIAMOND`
- `MARKER_SQUARE`
- `MARKER_TRIANGLE_UP`
- `MARKER_TRIANGLE_DOWN`

The following diagram depicts all possible marker types mentioned in the

previous list when they are printed on a white background, from left to right:



Drawing lines in OpenCV is possible using the `line` function. This function requires two points and it will draw a line connecting the given points. Here's an example:

```
Point pt1(25, image.rows/2);
Point pt2(image.cols/2 - 25, image.rows/2);
Scalar color = Scalar(0,255,0);
int thickness = 5;
int lineType = LINE_AA;
int shift = 0;

line(image,
      pt1,
      pt2,
      color,
      thickness,
      lineType,
      shift);
```

The `shift` parameter corresponds to the number of fractional bits in the given points. You can omit or simply pass zero to make sure it has no effect on your results.

Similar to the `line` function, `arrowedLine` can be used to draw an arrowed line. Obviously, the order of the given points determines the direction of the arrow. The only parameter that this function needs is the `tipLength` parameter, which corresponds to the percentage of the line length that will be used to create the tip of the arrow. Here is an example:

```
double tipLength = 0.2;

arrowedLine(image,
            pt1,
            pt2,
            color,
            thickness,
            lineType,
            shift,
            tipLength);
```

To be able to draw a circle on an image, we can use the `circle` function. Here's how this function is used to draw a circle right at the center of an image:

```
Point center(image.cols/2,
             image.rows/2);
```

```
int radius = 200;
circle(image,
       center,
       radius,
       color,
       thickness,
       lineType,
       shift);
```

Apart from `center` and `radius`, which are obviously the center point and radius of the circle, the rest of the parameters are the same as the functions and examples we learned about in this section.

Drawing a rectangle or square on an image is possible using the `rectangle` function. This function is very similar to the `line` function in that it requires only two points. The difference is that the points given to the `rectangle` function correspond to the top-left and bottom-right corner points of a rectangle or square. Here's an example:

```
rectangle(image,
          pt1,
          pt2,
          color,
          thickness,
          lineType,
          shift);
```

Instead of two separate `Point` objects, this function can also be provided with a single `Rect` object. Here's how:

```
Rect rect(pt1,pt2);
rectangle(image,
          color,
          thickness,
          lineType,
          shift);
```

Similarly, an ellipse can be drawn by using the `ellipse` function. This function requires the size of the axes along with the angle of the ellipse to be provided. Additionally, you can use start and ending angles to draw all or part of an ellipse, or, in others, an arc instead of an ellipse. You can guess that passing `0` and `360` as the start and ending angles will result in a full ellipse being drawn. Here's an example:

```
Size axes(200, 100);
double angle = 20.0;
double startAngle = 0.0;
double endAngle = 360.0;
ellipse(image,
```

```
center,  
axes,  
angle,  
startAngle,  
endAngle,  
color,  
thickness,  
lineType,  
shift);
```

Another way of calling the `ellipse` function is by using a `RotatedRect` object. In this version of the same function, you must first create a `RotatedRect` with a given width and height (or, in other words, size) and an `angle`, and then call the `ellipse` function as seen here:

```
Size size(150, 300);  
double angle = 45.0;  
RotatedRect rotRect(center,  
                     axes,  
                     angle);  
ellipse(image,  
        rotRect,  
        color,  
        thickness,  
        lineType);
```

Note that with this method, you can't draw an arc, and this is only used for drawing full ellipses.

We're down to the last type of shapes that can be drawn using OpenCV drawing functions, and that is polyline shapes. You can draw polyline shapes by using the `polylines` function. You must make sure to create a vector of points that corresponds to the vertices required for drawing a polyline. Here's an example:

```
vector<Point> pts;  
pts.push_back(Point(100, 100));  
pts.push_back(Point(50, 150));  
pts.push_back(Point(50, 200));  
pts.push_back(Point(150, 200));  
pts.push_back(Point(150, 150));  
bool isClosed = true;  
polylines(image,  
          pts,  
          isClosed,  
          color,  
          thickness,  
          lineType,  
          shift);
```

The `isClosed` parameter is used to determine whether the polyline must be closed, by connecting the last vertex to the first one or not.

The following image depicts the result of the `arrowedLine`, `circle`, `rectangle`, and `polylines` functions that we used in the preceding code snippets, when they are used to draw on our example image:



Before proceeding with the next section and learning about algorithms used for image filtering, we're going to learn about adjusting parameters at runtime by using trackbars added to OpenCV display windows. This is a very useful method in order to try out a wide range of different parameters at runtime when you are experimenting with different values to see their effect, and it allows the value of a variable to be changed by simply readjusting a trackbar (or slider) position.

Let's first see an example and then further break down the code and learn how trackbars are handled using OpenCV functions. The following complete example shows how we can use a trackbar to adjust the radius of a circle drawn on an image at runtime:

```
string window = "Image"; // Title of the image output window
string trackbar = "Radius"; // Label of the trackbar
Mat image = imread("Test.png");
Point center(image.cols/2, image.rows/2); // A Point object that points to the center
of the image
int radius = 25;
Scalar color = Scalar(0, 255, 0); // Green color in BGR (OpenCV default) color space
int thickness = 2; LineTypes lineType = LINE_AA; int shift = 0;

// Actual callback function where drawing and displaying happens
void drawCircle(int, void*)
{
    Mat temp = image.clone();

    circle(temp,
           center,
           radius,
           color,
```

```

        thickness,
        lineType,
        shift);

    imshow(window, temp);
}

int main()
{
    namedWindow(window); // create a window titled "Image" (see above)

    createTrackbar(trackbar, // label of the trackbar
                   window, // label of the window of the trackbar
                   &radius, // the value that'll be changed by the trackbar
                   min(image.rows, image.cols) / 2, // maximum accepted value
                   drawCircle);

    setTrackbarMin(trackbar, window, 25); // set min accepted value by trackbar
    setTrackbarMax(trackbar, window, min(image.rows, image.cols) / 2); // set max again

    drawCircle(0,0); // call the callback function and wait
    waitKey();

    return 0;
}

```

In the preceding code, `window` and `trackbar` are `string` objects that are used to identify and access a specific trackbar on a specific window. `image` is a `Mat` object containing the source image. `center`, `radius`, `color`, `thickness`, `lineType`, and `shift` are parameters required for drawing a circle, as we learned previously in this chapter. `drawCircle` is the name of the function (the callback function, to be precise) that will be called back when the trackbar is used to update the `radius` value for the circle we want to draw. This function must have the signature that is used in this example, which has an `int` and a `void` pointer as its parameters. This function is quite simple; it just clones the original image, draws a circle on it, and then displays it.

The `main` function is where we actually create the window and the trackbar. First, the `namedWindow` function must be called to create a window with the window name that we want. The `createTrackbar` function can then be called, as seen in the example, to create a trackbar on that window. Note that the trackbar itself has a name that is used to access it. This name will also be printed next to the trackbar to display its purpose to the user when the application is running. `setTrackbarMin` and `setTrackbarMax` are called to make sure our trackbar doesn't allow `radius` values less than 25 or greater than the width or height of the image (whichever is smaller), divided by 2 (since we're talking radius, not diameter).

The following is a screenshot that demonstrates the output of our `window` along

with trackbar on it that can be used to adjust the radius of the circle:



Try adjusting to see for yourself how the radius of the circle changes according to the position of the trackbar. Make sure to use this method when you want to experiment with the parameters of a function or algorithms that you learn in this book. Note that you can add as many trackbars as you need. However, adding more trackbars will use more space on your window, which might lead to a poor user interface and experience and, consequently, a hard-to-use program instead of a simplified one, so try to make use of trackbars wisely.

# Filtering images

It really doesn't matter whether you are trying to build a computer vision application to perform highly complex tasks, such as object detection in real-time, or simply modifying an input image in one way or another. Almost inevitably, you'll have to apply a certain type of filter to your input or output images. The reason for this is quite simple—not all pictures are ready to be processed out of the box, and most of the time, applying filters to make images smoother is one of the ways to make sure they can be fed into our algorithms.

The variety of filters that you can apply to your images in computer vision is huge, but in this section, we're going to be learning about some of the most important filters and especially the ones with an implementation in the OpenCV library that we can use.

# Blurring/smoothening filters

Blurring an image is one of the most important image filtering tasks, and there are many algorithms that can perform it, each one with their own pros and cons, which we'll be talking about in this section.

Let's start with the simplest filter used for smoothening images, which is called the median filter, and it can be done by using the `medianBlur` function, as seen in the following example:

```
| int ksize = 5; // must be odd  
| medianBlur(image, result, ksize);
```

This filter simply finds the median value of the neighboring pixels of each pixel in an image. `ksize`, or the kernel size parameter, decides how big the kernel used for blurring is, or, in other words, how far the neighboring pixels will be considered in the blurring algorithm. The following image depicts the result of increasing the kernel size from 1 to 7:



Note that the kernel size must be an odd value, and a kernel size of 1 will produce the exact same image as the input. So, in the previous images, you can see the increase in blurring level from the original image (the leftmost) until the kernel size of 7.

Note that you can use extremely high kernel sizes if you need, but it is rarely necessary and would usually be needed in case you need to de-noise an extremely noisy image. Here's an example image depicting the result of a kernel size of 21:



Another method of blurring an image is by using the `boxFilter` function. Let's see how it's done with an example code and then break it down further to better understand its behavior:

```
int ddepth = -1;
Size ksize(7,7);
Point anchor(-1, -1);
bool normalize = true;
BorderTypes borderType = BORDER_DEFAULT;

boxFilter(image,
          result,
          ddepth,
          ksize,
          anchor,
          normalize,
          borderType);
```

Box filtering is referred to as a blurring method in which a matrix of ones with the given `ksize` and `anchor` point is used for blurring an image. The major difference here, when compared with the `medianBlur` function, is that you can actually define the `anchor` point to be anything other than the center point in any neighborhood. You can also define the border type used in this function, whereas in the `medianBlur` function, `BORDER_REPLICATE` is used internally and cannot be changed. For more information about border types, you might want to refer to [Chapter 3, Array and Matrix Operations](#). Finally, the `normalize` parameter allows us to normalize the result to the displayable result.



*The `ddepth` parameter can be used to change the depth of the result. However, you can use -1 to make sure the result has the same depth as the source. Similarly, `anchor` can be supplied with -1 values to make sure the default anchor point is used.*

The following image depicts the result of the previous example code. The image on the right is the box-filtered result of the image on the left-hand side:



We can perform the exact same task, in other words, the normalized box filter, by using the `blur` function, as seen in the following example code:

```
Size ksize(7,7);
Point anchor(-1, -1);
BorderTypes borderType = BORDER_DEFAULT;

blur(image,
      result,
      ksize,
      anchor,
      borderType);
```

The result of this sample code is exactly the same as calling `boxFilter`, as we saw previously. The obvious difference here is that this function does not allow us to change the depth of the result and applies normalization by default.

In addition to the standard box filter, you can also apply a square box filter with the `sqrBoxFilter` function. In this method, instead of calculating the sum of the neighboring pixels, the sum of their square values is calculated. Here is an example, which is incredibly similar to calling the `boxFilter` function:

```
int ddepth = -1;
Size ksize(7,7);
Point anchor(-1, -1);
bool normalize = true;
BorderTypes borderType = BORDER_DEFAULT;

sqrBoxFilter(image,
             result,
             ddepth,
             ksize,
             anchor,
             normalize,
             borderType);
```



*The unnormalized versions of the `boxFilter` and `sqrBoxFilter` functions can also be used to find statistical information about the neighboring areas of all pixels in an image, and their use*



case is not limited to just blurring an image.

One of the most popular blurring methods in computer vision is the **Gaussian blur** algorithm, and it can be performed in OpenCV by using the `GaussianBlur` function. This function, similar to the previous blur functions that we learned about, requires a kernel size, along with standard deviation values in the X and Y direction, called `sigmax` and `sigmay`, respectively. Here's an example of how this function is used:

```
Size ksize(7,7);
double sigmax = 1.25;
double sigmay = 0.0;
BorderTypes borderType = BORDER_DEFAULT;

GaussianBlur(image,
              result,
              ksize,
              sigmax,
              sigmay,
              borderType);
```

Note that a value of zero for `sigmay` means the value of `sigmax` will also be used in the Y direction. For more information about Gaussian blur, you can read about the Gaussian function in general and the `GaussianBlur` function in the OpenCV documentation pages.

The last smoothening filter that we'll learn about in this section is called the **bilateral filter**, and it can be achieved by using the `bilateralFilter` function. Bilateral filtering is a powerful method of de-noising and smoothening an image, while preserving edges. This function is also much slower and more CPU-intensive in comparison with the blur algorithms that we saw previously. Let's see how `bilateralFilter` is used with an example and then break down the required parameters:

```
int d = 9;
double sigmaColor = 250.0;
double sigmaSpace = 200.0;
BorderTypes borderType = BORDER_DEFAULT;

bilateralFilter(image,
                 result,
                 d,
                 sigmaColor,
                 sigmaSpace,
                 borderType);
```

`d`, or the filter size, is the diameter of the pixel neighborhood that will participate

in the filter. The `sigmaColor` and `sigmaSpace` values are both used to define the effect of the color and coordinate the pixels nearer or farther from the pixel whose filtered value is being calculated. Here's a screenshot that demonstrates the effect of the `bilateralFilter` function when executed on our example image:



# Morphological filters

Similar to smoothening filters, morphology filters are the algorithms that change the value of each pixel based on the value of the neighboring pixels, although the obvious difference is that they do not have a blurring effect and are mostly used to produce some form of erosion or dilation effect on images. This will be clarified with a few hands-on examples further in this section, but for now, let's see how morphological operations (also called **transformations**) are performed using OpenCV functions.

You can use the `morphologyEx` function to perform morphological operations on images. This function can be provided with an entry from the `MorphTypes` enum to specify the morphological operation. Here are the values that can be used:

- `MORPH_ERODE`: For erosion operation
- `MORPH_DILATE`: For dilation operation
- `MORPH_OPEN`: For opening operation, or the dilation of eroded images
- `MORPH_CLOSE`: For closing operation, or the erosion of dilated images
- `MORPH_GRADIENT`: For morphological gradient operation, or subtraction of the eroded image from the dilated image
- `MORPH_TOPHAT`: For Top-hat operation, or subtraction of the opening operation result from the source image
- `MORPH_BLACKHAT`: For Black-hat operation, or subtraction of the source image from the result of the closing operation

To understand all of the possible morphological operations mentioned in the previous list, it's important to first understand the effect of erosion and dilation (the first two entries in the list) since the rest are simply different combinations of these two morphological operations. Let's first try erosion with an example and, at the same time, learn how the `morphologyEx` function is used:

```
MorphTypes op = MORPH_ERODE;
MorphShapes shape = MORPH_RECT;
Size ksize(3,3);
Point anchor(-1, -1);
Mat kernel = getStructuringElement(shape,
                                    ksize,
                                    anchor);

int iterations = 3;
BorderTypes borderType = BORDER_CONSTANT;
```

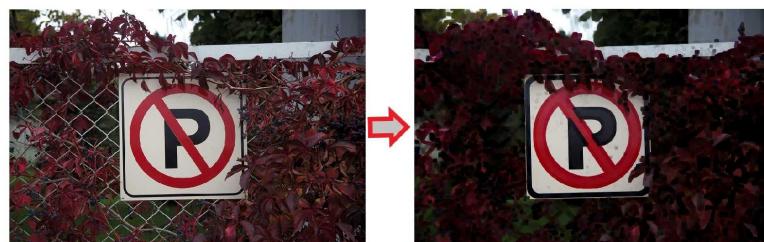
```

Scalar borderValue = morphologyDefaultBorderValue();
morphologyEx(image,
    result,
    op,
    kernel,
    anchor,
    iterations,
    borderType,
    borderValue);

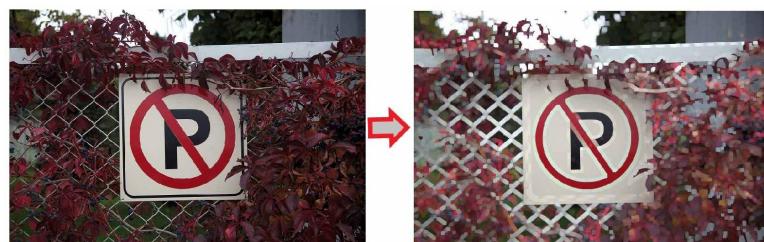
```

`op`, or the operation, is an entry from the `MorphTypes` enum, as mentioned before. `kernel`, or the structuring element, is the kernel matrix used for the morphological operation, which itself is either created manually or by using the `getStructuringElement` function. You must provide the `getStructuringElement` with a shape `morph`, kernel size (`ksize`), and `anchor`. `shape` can be a rectangle, cross, or ellipse, and is simply an entry from the `MorphShapes` enum. The `iterations` variable refers to the number of times the morphological operation is performed on an image. `borderType` is interpreted exactly the same way as with all the functions we've seen so far for the extrapolation of pixels. In case a constant border type value is used, the `morphologyEx` function must also be provided with a border value, which can be retrieved by using the `morphologyDefaultBorderValue` function, or specified manually.

Here is the result of our preceding code (for erosion), when executed on our sample image:



Dilation, on the other hand, is performed by simply replacing the `op` value with `MORPH_DILATE` in the previous example. Here's the result of the dilation operation:



A highly simplified description of what erosion and dilation operations do is that they cause the neighboring pixels of darker pixels to become darker (in case of erosion) or the neighboring pixels of brighter pixels to become brighter (in case of dilations), which, after more iterations, will cause a stronger and easily visible effect.

As was mentioned before, all of the other morphological operations are simply a combination of erosion and dilation. Here are the results of opening and closing operations when they are executed on our example images:



Make sure you try the rest of the morphological operations by yourself to see their effects. Create user interfaces that have trackbars on them and try changing the value of `iteration` and other parameters to see how they affect the result of morphological operations. If used carefully and wisely, morphological operations can have very interesting results and further simplify an operation that you want to perform on an image.

Before proceeding to the next section, it is worth noting that you can also use the `erode` and `dilate` functions to perform exactly the same operation. Although these functions do not require an operation parameter (since the operation is in their name already), the rest of the parameters are exactly the same as with the `morphologyEx` function.

# Derivative-based filters

In this section, we're going to learn about filtering algorithms that are based on calculating and using the derivatives of an image. To understand the concept of derivatives in an image, you can recall the fact that images are matrices, so you can calculate the derivatives (of any order) in  $X$  or  $Y$  directions, for instance, which, in the simplest case, would be the same as finding the change across the pixels in a direction.

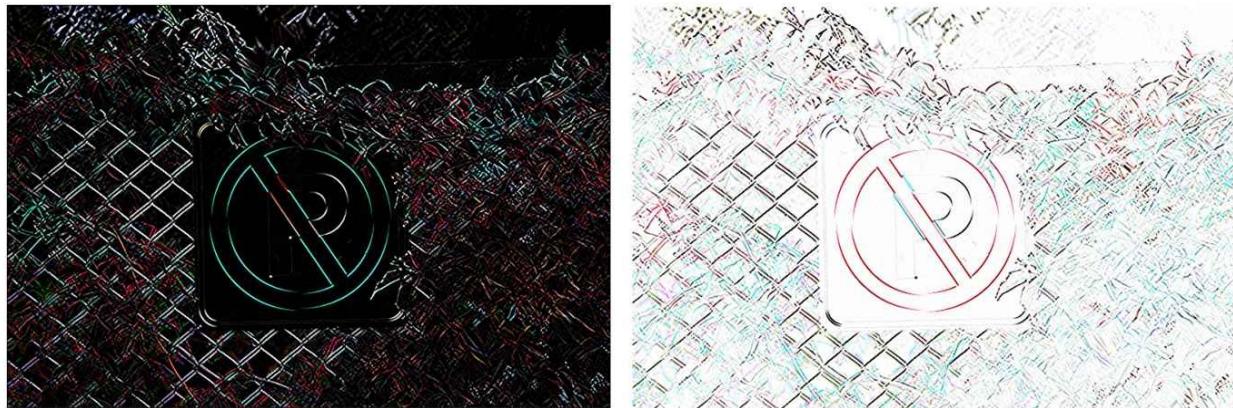
Let's start with the `Sobel` function, which is used to calculate the derivative of an image using a `Sobel` operator. Here's how this function is used in practice:

```
int ddepth = -1;
int dx = 1;
int dy = 1;
int ksize = 5;
double scale = 0.3;
double delta = 0.0;

BorderTypes borderType = BORDER_DEFAULT;
Sobel(image,
      result,
      ddepth,
      dx,
      dy,
      ksize,
      scale,
      delta,
      borderType);
```

`ddepth`, similar to what we saw in previous examples throughout this chapter, is used to define the depth of the output, and using `-1` makes sure the result has the same depth as the input. `dx` and `dy` are used to set the order of the derivative in both the  $X$  and  $Y$  directions. `ksize` is the size of the `Sobel` operator, which can be 1, 3, 5, or 7. `scale` is used as the `scale` factor for the result, and `delta` is added to the `result`.

The following images depict the result of the `Sobel` function when called with the parameter values from the preceding example code, but for a `delta` value of zero (on the left) and 255 (on the right):



Try setting different `delta` and `scale` values and experiment with the results. Also try different derivative orders and see the effects for yourself. As you can see from the preceding output images, calculating the derivative of an image is a method for calculating the edges in an image.

You can also use the `spatialGradient` function to calculate the first-order derivative of an image in both the *X* and *Y* directions using the `sobel` operator, at the same time. In other words, calling `spatialGradient` once is like calling the `sobel` function twice, for the first-order derivative in both directions. Here's an example:

```
Mat resultDX, resultDY;
int ksize = 3;
BorderTypes borderType = BORDER_DEFAULT;
spatialGradient(image,
                resultDX,
                resultDY,
                ksize,
                borderType);
```

Note that the `ksize` parameter must be 3 and the input image type must be grayscale, otherwise this function will fail to execute, although this might change in an upcoming OpenCV version.

Similar to the way we used the `sobel` function, you can use the `Laplacian` function to calculate the Laplacian of an image. It's important to note that this function essentially sums up the second derivatives in the *X* and *Y* directions calculated using the `sobel` operator. Here's an example that demonstrates the usage of the `Laplacian` function:

```
int ddepth = -1;
int ksize = 3;
double scale = 1.0;
double delta = 0.0;
```

```
BorderTypes borderType = BORDER_DEFAULT;
Laplacian(image,
           result,
           ddepth,
           ksize,
           scale,
           delta,
           borderType);
```

All of the parameters used in the `Laplacian` function are already described in previous examples, and especially the `sobel` function.

# Arbitrary filtering

OpenCV supports the application of arbitrary filters on images by using the `filter2D` function. This function is capable of creating the result of many algorithms we already learned, but it requires a `kernel` matrix to be provided. This function simply convolves the whole image with the given `kernel` matrix. Here's an example of applying an arbitrary filter on an image:

```
int ddepth = -1;

Mat kernel{+1, -1, +1,
           -1, +2, -1,
           +1, -1, +1};

Point anchor(-1, -1);

double delta = 0.0;

BorderTypes borderType = BORDER_DEFAULT;

filter2D(image,
          result,
          ddepth,
          kernel,
          anchor,
```

```
    delta,  
  
    borderType);
```

Here is the result of this arbitrary filter. You can see the original image on the left, and the result of the filtering operation on the right-hand side:



There is absolutely no limit to the number of possible filters you can create and use using the `filter2D` function. Make sure you try different kernel matrices and experiment with the `filter2D` function. You can also search online for popular filter kernel matrices and apply them using the `filter2D` function.

# Transforming images

In this section, we'll learn about computer vision algorithms that are used to transform images in one way or another. The algorithms that we'll be learning in this section cover algorithms that change the content of an image, or the way its contents is interpreted.

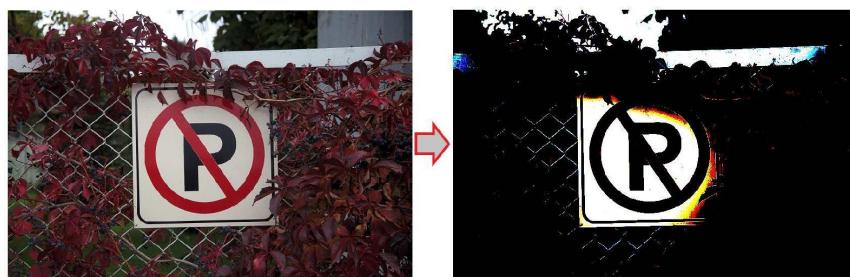
# Thresholding algorithms

Thresholding algorithms are used to apply a threshold value to the pixels of an image. These algorithms can be used to effectively create masks from images that have possible regions or pixels of interest in them that pass a certain threshold value.

You can use the `threshold` function to apply a threshold value on all pixels of an image. The `threshold` function must be provided with the type of threshold that is desired, and it can be an entry from the `ThresholdTypes` enum. The following is an example that can be used to find the brightest areas in an image using the `threshold` function:

```
double thresh = 175.0;
double maxval = 255.0;
ThresholdTypes type = THRESH_BINARY;
threshold(image,
          result,
          thresh,
          maxval,
          type);
```

`thresh` is the minimum threshold value, and `maxval` is the maximum allowed value. Simply put, all pixel values between `thresh` and `maxval` are allowed to pass and so the `result` is created. The following is the result of the preceding `threshold` operation demonstrated with an example image:



Increase the `thresh` parameter (or the threshold value) and you'll notice that fewer pixels are allowed to pass. Setting a correct threshold value requires experience and knowledge about the scene. In some cases, however, you can develop programs that set the threshold automatically for you, or adaptively. Note that the threshold type completely affects the result of the `threshold` function. For

instance, `THRESH_BINARY_INV` will produce the inverted result of `THRESH_BINARY` and so on. Make sure to try different threshold types and experiment for yourself with this interesting and powerful function.

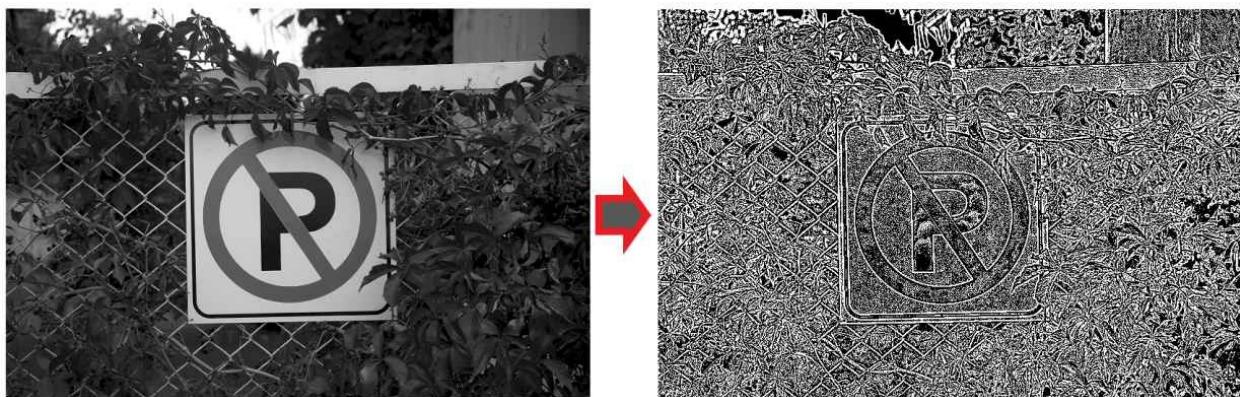
Another, more sophisticated, way of applying thresholds to images is by using the `adaptiveThreshold` function, which works with grayscale images. This function assigns the given `maxValue` parameter to the pixels that pass the threshold criteria. Besides that, you must provide a threshold type, an adaptive `threshold` method, a block size defining the diameter in the pixel neighborhood, and a constant value that is subtracted from the mean (depending on the adaptive `threshold` method). Here is an example:

```
double maxValue = 255.0;
AdaptiveThresholdTypes adaptiveMethod =
    ADAPTIVE_THRESH_GAUSSIAN_C;
ThresholdTypes thresholdType = THRESH_BINARY;
int blockSize = 5;
double c = 0.0;
adaptiveThreshold(image,
    result,
    maxValue,
    adaptiveMethod,
    thresholdType,
    blockSize,
    c);
```

Note that `adaptiveMethod` can take any of the following examples:

- `ADAPTIVE_THRESH_MEAN_C`
- `ADAPTIVE_THRESH_GAUSSIAN_C`

The higher the `blockSize` parameter value, the more pixels are used in the adaptive `threshold` method. The following is an example image that depicts the result of the `adaptiveThreshold` method called with the values in the preceding example code:



# Color space and type conversion

Converting various color spaces and types to each other is extremely important, especially when it comes to dealing with images from different device types or intending to display them on different devices and formats. Let's see what this means with a very simple example. You'll need grayscale images for various OpenCV functions and computer vision algorithms, while some will require RGB color images. In such cases, you can use the `cvtColor` function to convert between various color spaces and formats.

The following is an example that converts a color image to grayscale:

```
| ColorConversionCodes code = COLOR_RGB2GRAY;  
| cvtColor(image,  
|           result,  
|           code);
```

`code` can take a conversion code, which must be an entry from the `ColorConversionCodes` enum. The following are some examples of the most popular color-conversion codes, that can be used with the `cvtColor` function:

- `COLOR_BGR2RGB`
- `COLOR_RGB2GRAY`
- `COLOR_BGR2HSV`

The list goes on and on. Make sure to check the `ColorConversionCodes` enum for all possible color-conversion codes.

# Geometric transformation

This section is dedicated to geometric transformation algorithms and OpenCV functions. It's important to note that the name, Geometric transformation, is based on the fact that the algorithms falling in this category do not change the content of an image, they simply deform the existing pixels, while using an extrapolation and interpolation method to calculate the pixels that fall outside the area of the existing pixels, or over each other, respectively.

Let's start with the simplest geometric transformation algorithm, which is used for resizing an image. You can use the `resize` function to resize an image. Here is how this function is used:

```
Size dsize(0, 0);
double fx = 1.8;
double fy = 0.3;
InterpolationFlags interpolation = INTER_CUBIC;
resize(image,
       result,
       dsize,
       fx,
       fy,
       interpolation);
```

If the `dsize` parameter is set to a non-zero size, the `fx` and  parameters are used to scale the input image. Otherwise, if `fx` and  are both zero, the input image is resized to the given `dsize`. The `interpolation` parameter, on the other hand, is used to set the `interpolation` method used for the `resize` algorithm, and it must be one of the entries in the `InterpolationFlags` enum. Here are some of the possible values for the `interpolation` parameter:

- `INTER_NEAREST`
- `INTER_LINEAR`
- `INTER_CUBIC`
- `INTER_AREA`
- `INTER_LANCZOS4`

Make sure to check out the OpenCV documentation pages for `InterpolationFlags` to find out about the details of each possible method.

The following image depicts the result of the previous example code that was used for resizing an image:



Probably the most important geometric transformation algorithm, which is able to perform most of the other geometric transformations, is the remapping algorithm, and it can be achieved by calling the `remap` function.

The `remap` function must be provided with two mapping matrices, one for  $X$  and another one for the  $Y$  direction. Besides that, an interpolation and extrapolation (border type) method, along with a border value in case of constant border type, must be provided to the `remap` function. Let's first see how this function is called and then try a couple of different remaps. Here's an example that shows how the `remap` function is called:

```
Mat mapX(image.size(), CV_32FC1);
Mat mapY(image.size(), CV_32FC1);
// Create maps here...
InterpolationFlags interpolation = INTER_CUBIC;
BorderTypes borderMode = BORDER_CONSTANT;
Scalar borderValue = Scalar(0, 0, 0);
remap(image,
      result,
      mapX,
      mapY,
      interpolation,
      borderMode,
      borderValue);
```

You can create an infinite number of different mappings and use them with the `remap` function to resize, flip, warp, and do many other transformations on images. For instance, the following code can be used to create a remapping that will cause a vertical flip of the resulting image:

```
for(int i=0; i<image.rows; i++)
    for(int j=0; j<image.cols; j++)
    {
        mapX.at<float>(i,j) = j;
        mapY.at<float>(i,j) = image.rows-i;
    }
```

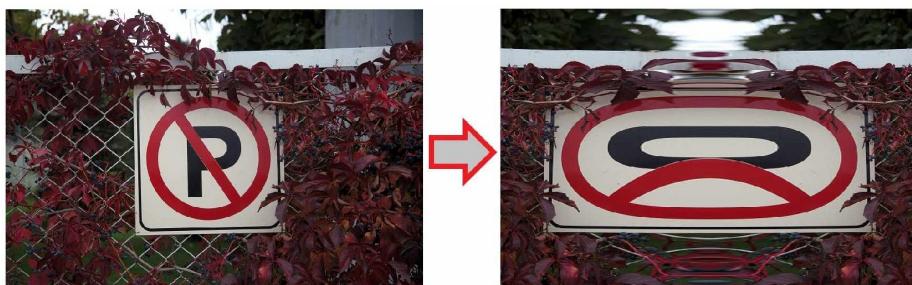
Replace the code in the preceding `for` loop with the following and the result of the `remap` function call will be a horizontally flipped image:

```
| mapX.at<float>(i,j) = image.cols - j;  
| mapY.at<float>(i,j) = i;
```

Besides simple flipping, you can use the `remap` function to perform many interesting pixel deformations. Here's an example:

```
Point2f center(image.cols/2,image.rows/2);  
for(int i=0; i<image.rows; i++)  
    for(int j=0; j<image.cols; j++)  
    {  
        // find i,j in the standard coordinates  
        double x = j - center.x;  
        double y = i - center.y;  
  
        // Perform any mapping for X and Y  
        x = x*x/750;  
        y = y;  
  
        // convert back to image coordinates  
        mapX.at<float>(i,j) = x + center.x;  
        mapY.at<float>(i,j) = y + center.y;  
    }
```

As can be seen from the inline comments of the preceding example code, it is common to convert the OpenCV `i` and `j` values (row and column number) to a standard coordinate system and use `X` and `Y` with known mathematical and geometrical functions, and then convert them back to the OpenCV image coordinates. The following image demonstrates the result of the preceding example code:



The `remap` function is incredibly powerful and efficient as long as the calculation of `mapX` and `mapY` are handled efficiently. Make sure to experiment with this function to learn about more remapping possibilities.

There are a huge number of geometric transformation algorithms in computer

vision and in the OpenCV library, and covering all of them would need a book of its own. So, we'll leave the rest of the geometric transformation algorithms for you to explore and try on your own. Refer to the *Geometric Image Transformations* section in the OpenCV `imgproc` (Image Processing) module documentation for more geometric transformation algorithms and functions. In particular, make sure to learn about functions including `getPerspectiveTransform` and `getAffineTransform`, which are used to find the perspective and affine transformation between two sets of points. Such functions return transformation matrices that can be used to apply perspective and affine transformation to images by using the `warpPerspective` and `warpAffine` functions.

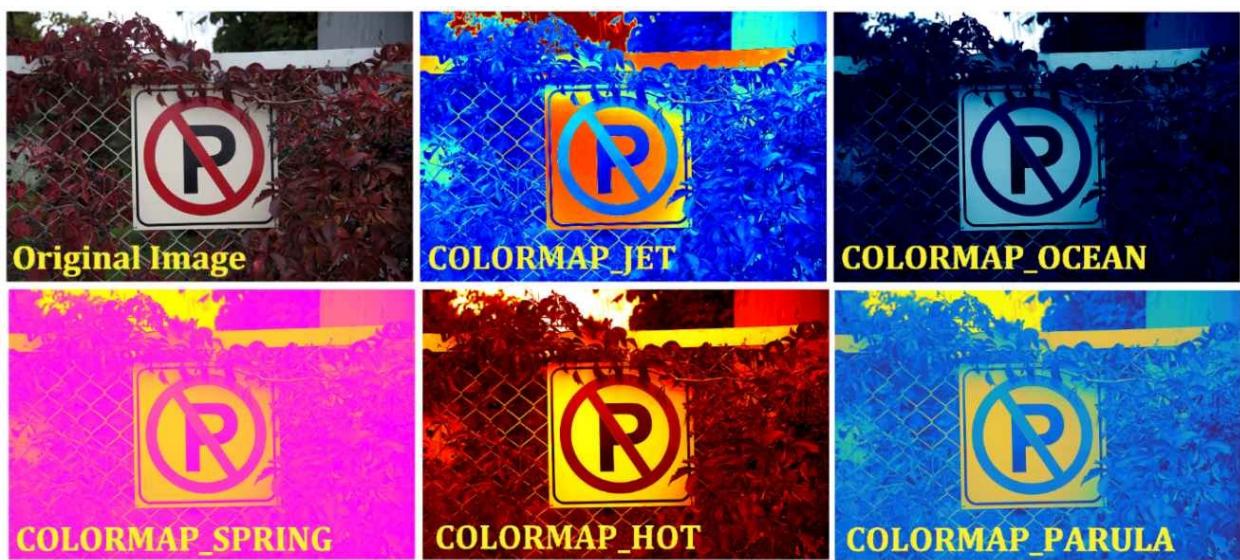
# Applying colormaps

We'll end this chapter by learning about applying colormaps to images. This is a fairly simple but powerful method that can be used to modify the colors of an image or its tone in general. This algorithm simply replaces the colors of an input image using a colormap and creates a result. A **colormap** is a 256-element array of color values, in which each element represents the color that must be used for the corresponding pixel values in the source image. We will break this down further with a couple of examples, but before that, let's see how colormaps are applied to images.

OpenCV contains a function named `applyColorMap` that can be used to either apply predefined colormaps or custom ones created by the user. In case predefined colormaps are used, `applyColorMap` must be provided with a colormap type, which must be an entry from the `ColormapTypes` enum. Here's an example:

```
ColormapTypes colormap = COLORMAP_JET;  
applyColorMap(image,  
              result,  
              colormap);
```

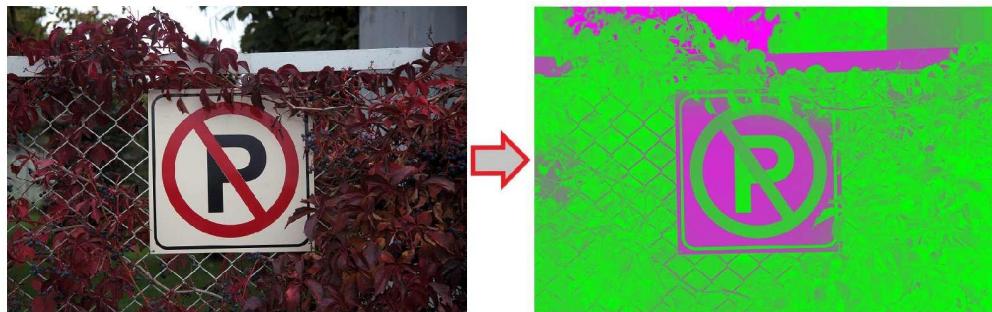
The following image depicts the result of various predefined colormaps that can be applied using the `applyColorMap` function:



As mentioned earlier, you can also create your own custom colormaps. You just need to make sure you follow the instructions for creating a colormap. Your colormap must have a size of 256 elements (a `Mat` object with 256 rows and 1 column) and it must contain color or grayscale values, depending on the type of image you are planning to apply the colormap to. The following is an example that shows how to create a custom colormap by simply inverting the green channel color:

```
Mat userColor(256, 1, CV_8UC3);
for(int i=0; i<=255; i++)
    userColor.at<Vec3b>(i,0) = Vec3b(i, 255-i, i);
applyColorMap(image,
              result,
              userColor);
```

Here is the result of the preceding example code:



# Summary

Even though we tried to cover the most important topics of image-processing algorithms in computer vision, we still have a long way to go and many more algorithms to learn. The reason is quite simple, and that is the variety of applications that these algorithms can be used for. We learned a great deal of widely used computer vision algorithms in this chapter, but it should also be noted that whatever we learned in this chapter is also meant to ease your way when you start exploring the rest of the image-processing algorithms by yourself.

We started this chapter by learning about drawing functions that can be used to draw shapes and text on images. Then we moved on to learn about one of the most important computer vision topics: image filtering. We learned how to use smoothening algorithms, we experimented with morphological filters, and learned about erosion, dilation, opening, and closing. We also got to try some simple edge detection algorithms, or, in other words, derivative-based filters. We learned about thresholding images and changing their color spaces and types. The final sections of this chapter introduced us to geometric transformations and applying colormaps on images. We even created a custom colormap of our own. You can easily find the trace of the algorithms that we learned in this chapter, in many professional photo-editing or social-networking and photo-sharing applications.

In the next chapter, we'll learn all about histograms, how they are calculated, and how they are used in computer vision. We'll cover the algorithms that are crucial when working on certain object detection and tracking algorithms that we'll be exploring in the upcoming chapters.

# Questions

1. Write a program that draws a cross mark over the whole image, with a thickness of 3 pixels and in the color red.
2. Create a window with a trackbar to change the `ksize` of a `medianBlur` function. The possible range for the `kszise` value should be between 3 and 99.
3. Perform a gradient morphological operation on an image, considering a kernel size of 7 and a rectangular morphological shape for the structuring element.
4. Using `cvtColor`, convert a color image to grayscale and make sure only the darkest 100 shades of gray are filtered out using the `threshold` function. Make sure that filtered pixels are set to white in the result image and the rest of the pixels are set to black.
5. Use the `remap` function to resize an image to half of its original width and height, thus preserving the aspect ratio of the original image. Use a default border type for the extrapolation.
6. a) Use colormaps to convert an image to grayscale. b) Convert an image to grayscale and invert its pixels at the same time.
7. Did you read about perspective transformation functions? Which OpenCV function covers all similar transformations in one single function?

# Back-Projection and Histograms

In the previous chapter, we learned about many computer vision algorithms and OpenCV functions that can be used to prepare images for further processing or modify them in one way or another. We learned how to draw text and shapes on images, filter them using smoothening algorithms, perform morphological transformations on them, and calculate their derivatives. We also learned about geometric and miscellaneous transformations of images and applied colormaps to alter the tone of our images.

In this chapter, we'll be learning about a few more algorithms and functions that are used mostly to prepare images for further processing, inference, and modification. This will be further clarified later on in this chapter, after we learn about histograms in computer vision. We'll be introduced to the concept of histograms, and then we'll learn how they are calculated and utilized with hands-on example codes. The other extremely important concept we'll be learning about in this chapter is called **back-projection**. We'll learn how back-projection of a histogram can be used to create an altered version of the original image.

Besides their standard usage, the concepts and algorithms that we'll learn in this chapter are also essential when it comes to dealing with some of the most widely used algorithms for object detection and tracking, which we'll be learning in the upcoming chapters.

In this chapter, we'll cover the following:

- Understanding histograms
- Back projection of histograms
- Histogram comparison
- Equalizing histograms

# Technical requirements

- An IDE to develop C++ or Python applications
- OpenCV library

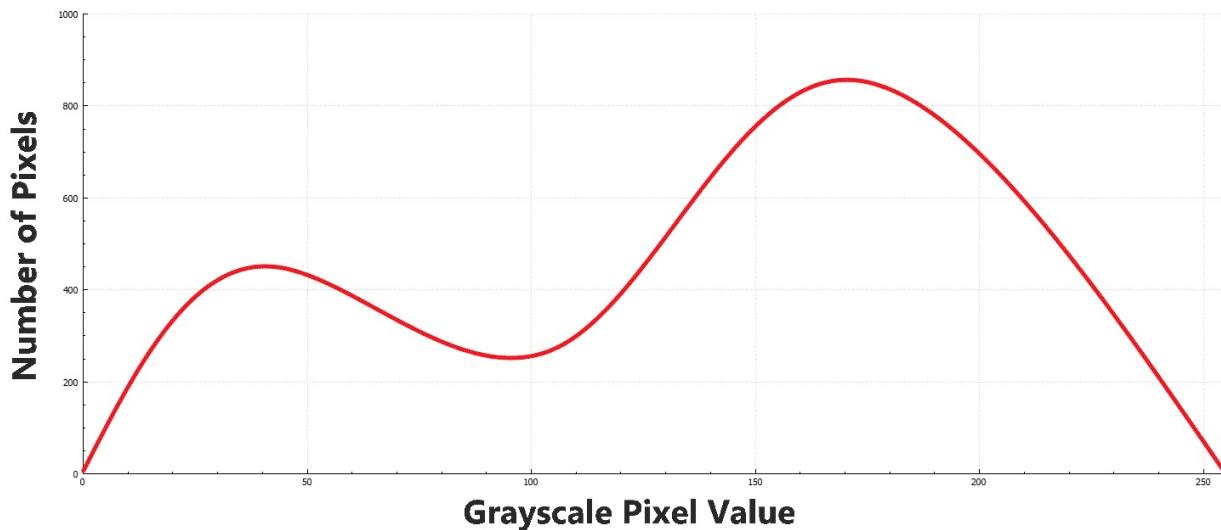
Refer to [chapter 2, \*Getting Started with OpenCV\*](#), for more information about how to set up a personal computer and make it ready to develop computer vision applications using the OpenCV library.

You can use the following URL to download the source codes and examples for this chapter:

<https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter05>

# Understanding histograms

In computer vision, histograms are simply graphs that represent the distribution of pixel values over the possible range of the accepted values for those pixels, or, in other words, the probability distribution of pixels. Well, this might not be as crystal clear as you would expect, so let's take single-channel grayscale images as a simple example to describe what histograms are, and then expand it to multi-channel colored images, and so on. We already know that the pixels in a standard grayscale image can contain values between 0 and 255. Considering this fact, a graph similar to the following, which depicts the ratio of the number of pixels containing each and every possible grayscale pixel value of an arbitrary image, is simply the histogram of that given image:



Keeping in mind what we just learned, it can be easily guessed that the histogram of a three-channel image, for example, would be three graphs representing the distribution of values for each channel, similar to what we just saw with the histogram of a single-channel grayscale image.

You can use the `calcHist` function in the OpenCV library to calculate the histogram of one or multiple images that can be single-channel or multi-channel themselves. This function requires a number of parameters that must be provided carefully for it to produce the desired results. Let's see how this function is used

with a few examples.

The following example code (followed by description of all the parameters) demonstrates how you can calculate the histogram of a single grayscale image:

```
Mat image = imread("Test.png");
if(image.empty())
    return -1;
Mat grayImg;
cvtColor(image, grayImg, COLOR_BGR2GRAY);

int bins = 256;
int nimages = 1;
int channels[] = {0};
Mat mask;
int dims = 1;
int histSize[] = { bins };
float rangeGS[] = {0, 256};
const float* ranges[] = { rangeGS };
bool uniform = true;
bool accumulate = false;
Mat histogram;
calcHist(&grayImg,
         nimages,
         channels,
         mask,
         histogram,
         dims,
         histSize,
         ranges,
         uniform,
         accumulate);
```

We infer the following from the preceding code:

- `grayImg` is the input grayscale image that wants to calculate its histogram, and `histogram` will contain the result.
- `nimages` must contain the number of images for which we want histograms calculated, which, in this case, is just one image.
- `channels` is an array that is supposed to contain the zero-based index number of the channels in each image for which we want their histogram calculated. For instance, if we want to calculate the histogram of the first, second, and fourth channels in a multi-channel image, the `channels` array must contain the values of 0, 1, and 3. In our example, `channels` only contained 0, since we're calculating the histogram of the only channel in a grayscale image.
- `mask`, which is common to many other OpenCV functions, is a parameter that is used to mask (or ignore) certain pixels, or, in other words, prevent them from participating in the calculated result. In our case, and as long as we are not working on a certain portion of an image, `mask` must contain an empty matrix.

- `dims`, or the dimensionality parameters, corresponds to the dimensionality of the result histogram that we are calculating. It must not be greater than `cv_MAX_DIM`, which is 32 in current OpenCV versions. We'll be using `1` in most cases, since we expect our histogram to be a simple array-shaped matrix. Consequently, the index number of each element in the resulting histogram will correspond to the bin number.
- `histsize` is an array that must contain the size of the histogram in each dimension. In our example, since the dimensionality was `1`, `histsize` must contain a single value. The size of the histogram, in this case, is the same as the number of bins in a histogram. In the preceding example code, `bins` is used to define the number of bins in the histogram, and it is also used as the single `histsize` value. Think of `bins` as the number of groups of pixels in a histogram. This will be further clarified with examples later on, but for now, it is important to note that a value of `256` for `bins` will result in a histogram containing the count of all individual possible pixel values.
- `ranges` must contain pairs of values corresponding to the lower and higher bounds of each range of possible values when calculating the histogram of an image. In our example, this means a value in the single range of `(0, 256)`, which is what we have provided to this parameter.
- The `uniform` parameter is used to define the uniformity of the histogram. Note that if the histogram is non-uniform, as opposed to what is demonstrated in our example, the `ranges` parameter must contain the lower and higher bounds of all dimensions, respectively.
- The `accumulate` parameter is used to decide whether the histogram should be cleared before it is calculated, or the calculated values should be added to an existing histogram. This can be quite useful when you need to calculate a single histogram using multiple images.



*We'll cover the parameters mentioned here as much as possible in the examples provided in this chapter. However, you can also refer to the online documentation of the `calcHist` function for more information.*

# Displaying histograms

Quite obviously, trying to display the resulting histogram using a function, such as `imshow`, is futile since the raw format of the stored histogram is similar to a single column matrix that has `bins` number of rows in it. Each row, or, in other words, each element, of the histogram corresponds to the number of pixels that fall into that specific bin. Considering this, we can draw the calculated histogram by using drawing functions from [Chapter 4, Drawing, Filtering, and Transformation](#).

Here's an example that shows how we can display the histogram that we calculated in the previous code sample as a graph with custom size and properties:

```
int gRows = 200; // height
int gCol = 500; // width
Scalar backGroundColor = Scalar(0, 255, 255); // yellow
Scalar graphColor = Scalar(0, 0, 0); // black
int thickness = 2;
LineTypes lineType = LINE_AA;

Mat theGraph(gRows, gCol, CV_8UC3, backGroundColor);

Point p1(0,0), p2(0,0);
for(int i=0; i<bins; i++)
{
    float value = histogram.at<float>(i,0);
    value = maxVal - value; // invert
    value = value / maxVal * theGraph.rows; // scale
    line(theGraph,
        p1,
        Point(p1.x,value),
        graphColor,
        thickness,
        lineType);
    p1.y = p2.y = value;
    p2.x = float(i+1) * float(theGraph.cols) / float(bins);
    line(theGraph,
        p1, p2,
        Scalar(0,0,0));
    p1.x = p2.x;
}
```

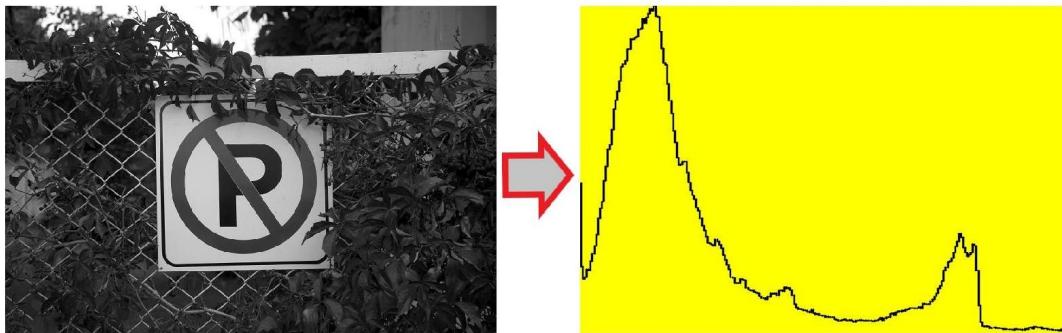
In the preceding code, `gRow` and `gCol` refer to the height and width of the resulting graph, respectively. The rest of the parameters are either self-explanatory (`backGroundColor` and so on), or you have already learned about them in the previous chapters. Notice how each value in `histogram` is used to calculate the

position of the line that needs to be drawn. In the preceding code, `maxVal` is simply used to scale the results to the visible range. Here's how `maxVal` itself is calculated:

```
double maxVal = 0;
minMaxLoc(histogram,
            0,
            &maxVal,
            0,
            0);
```

Refer to [chapter 3, Array and Matrix Operations](#), if you need to refresh your memory about how the `minMaxLoc` function is used. In our example, we only need the value of the biggest element in the histogram, so we ignore the rest of the parameters by passing zero to them.

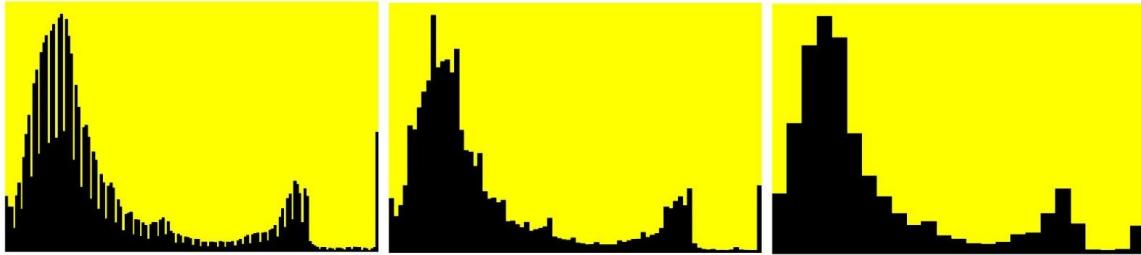
Here is the result of the preceding example codes:



You can easily change the background or the graph color using the provided `backGroundColor` or `graphColor` parameters, or make the graph thinner or thicker by changing the `thickness` parameter, and so on.

Interpretation of a histogram is extremely important, especially in photography and photo-editing applications, so being able to visualize them is essential for easier interpretation of the results. For instance, in the preceding example, it can be easily noticed from the resulting histogram that the source image contains more tones of darker colors than brighter ones. We'll see more examples of darker and brighter images later on, but before that, let's see how changing the number of bins would affect the result.

The following are the resulting histograms of the same image as the previous example, with 150, 80, and 25 bins, from left to right, drawn using a bar chart visualization method:



You can easily notice that the lower the bins value is, the more grouped together the pixels are. Even though this might seem more like a lower resolution of the same data (from left to right), it is actually better to use a lower number of bins to group similar pixels together. Note that the bar-chart visualization in the preceding example is produced by replacing the `for` loop from the previous example code with the following:

```
Point p1(0,0), p2(0, theGraph.rows-1);
for(int i=0; i<bins; i++)
{
    float value = histogram.at<float>(i,0);
    value *= 0.95f; // 5% empty at top
    value = maxVal - value; // invert
    value = value / (maxVal) * theGraph.rows; // scale
    p1.y = value;
    p2.x = float(i+1) * float(theGraph.cols) / float(bins);
    rectangle(theGraph,
              p1,
              p2,
              graphColor,
              CV_FILLED,
              lineType);
    p1.x = p2.x;
}
```

Both of these visualizations (graph or bar-chart) have their own pros and cons, which will be more obvious as you try calculating histograms of different types of images. Let's try calculating the histogram of a color image. We'll need to calculate the histogram of individual channels, as was mentioned previously. Here's an example code that demonstrates how it's done:

```
Mat image = imread("Test.png");
if(image.empty())
{
    cout << "Empty input image!";
    return -1;
}

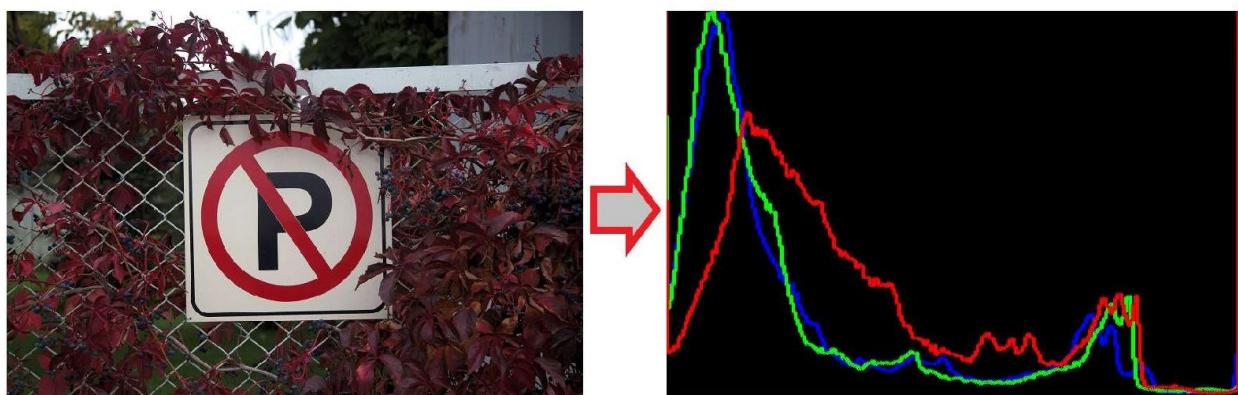
Mat imgChannels[3];
Mat histograms[3];
```

```

| split(image, imgChannels);
| // each imgChannels element is an individual 1-channel image

```

You can download the complete source code for the preceding example from the online source code repository for this chapter. The example project containing the full source code of the preceding code snippet is called `cvHistGraphColor`, and running it would produce a result similar to what is seen in the following diagram:



As you can see in the preceding example code, the `split` function is used to create three individual images, each containing a single channel, out of our source color image (BGR by default). The part of the code that is mentioned with the commented line in the preceding code is simply a `for` loop that iterates over the elements of `imgChannels` and draws each graph using the exact same code as you saw before, but with each graph having its own unique color that is calculated using the following code in the loop:

```

| Scalar graphColor = Scalar(i == 0 ? 255 : 0,
|                           i == 1 ? 255 : 0,
|                           i == 2 ? 255 : 0);

```

Depending on the value of `i`, `graphColor` is set to blue, green, or red, hence the resulting histography depicted in the previous picture.

Besides interpreting the content of an image, or to seeing how pixel values are distributed in an image, histograms have many use cases. In the following sections, we'll be learning about back-projection and other algorithms that are used for utilizing histograms in our applications.

# Back-projection of histograms

Considering the definition of histograms from the start of the previous section, it can be said that back-projection of a histogram on an image means replacing each of its pixels with their probability distribution value. This is, in a way (not exactly), the reverse operation of calculating the histogram of an image. When we back-project a histogram on an image, we actually use a histogram to modify an image. Let's first see how back-projection is performed using OpenCV and, afterwards, dive into how it is actually used.

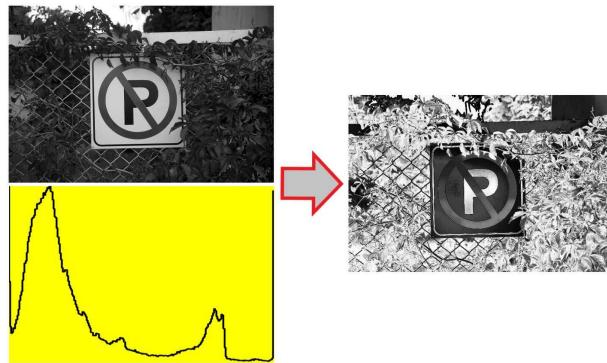
You can use the `calcBackProject` function to calculate the back-projection of a histogram on an image. This function needs a similar set of parameters to the `calcHist` function. Let's see how it is called and then further break down its parameters:

```
calcBackProject(&image,
                nimages,
                channels,
                histogram,
                backProj,
                ranges,
                scale,
                uniform);
```

The `nimages`, `channels`, `ranges`, and `uniform` parameters in the `calcBackProject` function are used exactly the way they were with the `calcHist` function. `image` must contain the input image and `histogram` needs to be calculated with a prior call to the `calcHist` function or with any other method (or even manually). The result will be scaled by using the `scale` parameter and finally, it will be saved in `backProj`. It's important to note that the values in `histogram` can be over the correctly displayable range, so after performing the back-projection, the resulting `backProj` object will not be displayable correctly. To fix this issue, we need to first make sure `histogram` is normalized to the displayable range by OpenCV. The following code must be executed before the preceding call to `calcBackProject` in order for the resulting `backProj` to be displayable:

```
normalize(histogram,
          histogram,
          0,
          255,
          NORM_MINMAX);
```

The following image depicts the result of the back-projection of the image with its original histogram (unaltered histogram). The image on the right-hand side is the result of the back-projection algorithm:



According to the definition of histograms and back-projection, it can be said that the darker areas in the preceding back-projection result image contain pixels that are less common to the original image, and vice versa. This algorithm can be used (or even abused) to alter an image using a modified, or manually-made histogram. This technique is commonly used, for example, to create masks that extract only portions of an image that contains a given color or intensity.

Here is an example that demonstrates how you can use the concept of histograms and back-projection to detect the pixels in an image that are in the range of the brightest 10% of possible pixel values:

```
int bins = 10; // we need 10 slices
float rangeGS[] = {0, 256};
const float* ranges[] = {rangeGS};
int channels[] = {0};
Mat histogram(bins, 1, CV_32FC1, Scalar(0.0));
histogram.at<float>(9, 0) = 255.0;
calcBackProject(&imageGray,
    1,
    channels,
    histogram,
    backProj,
    ranges);
```

Notice that the histogram is formed manually, with 10 bins, instead of being calculated from the original image. Then, the last bin, or, in other words, the last element in the histogram, is set to 255, which means absolute white. Obviously, if this wasn't done, we'd need to perform a normalization to make sure the result of back-projection is in the displayable range of colors.

The following image depicts the result of the preceding code snippet when it is executed on the same sample image from the previous examples:



The extracted mask image can be used to further modify an image, or, in the case of a uniquely-colored object, it can be used to detect and track the object. The detection and tracking algorithms will be covered thoroughly in the upcoming chapters, but how exactly we can use the color of an object is what we're going to learn next.

# Learning more about back-projections

First off, let's recall that the HSV color space is far better suited to dealing with the actual color value of pixels in an image than the standard RGB (or BGR and so on) color space. You might want to revisit [chapter 1, Introduction to Computer Vision](#), for more information about this phenomenon. We're going to use this simple fact to find regions in an image that have a special color, regardless of their color intensity, brightness, and so on. For this reason, we need to first convert an image to the HSV color space.

Let's simplify this with an example case. Imagine we want to replace a specific color in an image, preserving the highlights, brightness, and so on. To be able to perform such a task, we need to be able to accurately detect a given color and then make sure we only change the color in the detected pixels and not their brightness and similar properties. The following example code demonstrates how we can use a manually-formed histogram of the hue channel and its back-projection to extract pixels that have a specific color, which, in this example, is assumed to be blue:

1. To perform such an operation, we need to start by reading an image, converting it to the HSV color space, and extracting the hue channel, or, in other words, the first channel, as seen here:

```
Mat image = imread("Test.png");
if(image.empty())
{
    cout << "Empty input image!";
    return -1;
}

Mat imgHsv, hue;
vector<Mat> hsvChannels;
cvtColor(image, imgHsv, COLOR_BGR2HSV);
split(imgHsv, hsvChannels);
hue = hsvChannels[0];
```

2. Now that we have the hue channel inside the `hue` object, we need to form a proper histogram of the hue channel, which contains only the pixels with the color blue. The hue value can be a value between 0 and 360 (in degrees)

and the hue value of blue is 240. So, we can create a histogram using the following code, which can be used to extract the blue-colored pixels, with an offset (or threshold) of 50 pixels:

```
int bins = 360;
int blueHue = 240;
int hueOffset = 50;
Mat histogram(bins, 1, CV_32FC1);
for(int i=0; i<bins; i++)
{
    histogram.at<float>(i, 0) =
        (i > blueHue - hueOffset)
        &&
        (i < blueHue + hueOffset)
        ?
        255.0 : 0.0;
}
```



The preceding code acts like a simple threshold, in which all elements in the histogram that have an index of 240 (plus/minus 50) are set to 255 and the rest are set to zero.

3. Visualizing the manually-created hue channel histogram will allow us to have a better idea of the exact colors that are going to be extracted using it. The following code can be used to easily visualize a hue histogram:

```
double maxVal = 255.0;

int gW = 800, gH = 100;
Mat theGraph(gH, gW, CV_8UC3, Scalar::all(0));

Mat colors(1, bins, CV_8UC3);
for(int i=0; i<bins; i++)
{
    colors.at<Vec3b>(i) =
        Vec3b(saturate_cast<uchar>(
            (i+1)*180.0/bins), 255, 255);
}
cvtColor(colors, colors, COLOR_HSV2BGR);
Point p1(0,0), p2(0,theGraph.rows-1);
for(int i=0; i<bins; i++)
{
    float value = histogram.at<float>(i,0);
    value = maxVal - value; // invert
    value = value / maxVal * theGraph.rows; // scale
    p1.y = value;
    p2.x = float(i+1) * float(theGraph.cols) / float(bins);
    rectangle(theGraph,
              p1,
              p2,
              Scalar(colors.at<Vec3b>(i)),
              CV_FILLED);
    p1.x = p2.x;
}
```

Before proceeding with the next steps, let's break down the preceding example code. It is almost exactly the same as visualizing a grayscale

histogram or a single red-, green-, or blue-channel histogram. However, the interesting fact to note about the preceding code is where we form the `colors` object. The `colors` object is going to be a simple vector that contains all possible colors across the hue spectrum, but according to the number of bins we have. Notice how we have used the `saturate_cast` function in OpenCV to make sure the hue values are saturated into the acceptable range. The S and V channels are simply set to their highest possible value, which is 255. After the `colors` object is correctly created, we have used the same visualization function as before. However, since OpenCV does not display images in the HSV color space by default (and you can expect the same behavior in most image display functions and libraries), we need to convert the HSV color space to BGR in order to display the colors correctly.



*Even though hue can take a value in the range of (0, 360), it is not possible to store it in single-byte C++ types (such as `uchar`), which are capable of storing values in the range of (0, 255). That is why hue values are considered to be in the range of (0, 180) in OpenCV, or, in other words, they are simply divided by two.*

The following image depicts the result of the preceding example code, if we try to display `theGraph` using the `imshow` function:



These are the colors we're going to extract in a mask, if we use its corresponding histogram to calculate the back-project of an image. This range of colors is created using the simple threshold (in a loop) that we did when we formed the histogram manually. Obviously, if you set all the values of the histogram to `255.0` instead of just the blue range, you'd have the whole spectrum of colors. Here's a simple example:

```
Mat histogram(bins, 1, CV_32FC1);
for(int i=0; i<bins; i++)
{
    histogram.at<float>(i, 0) = 255.0;
}
```

The visualization output would be the following:



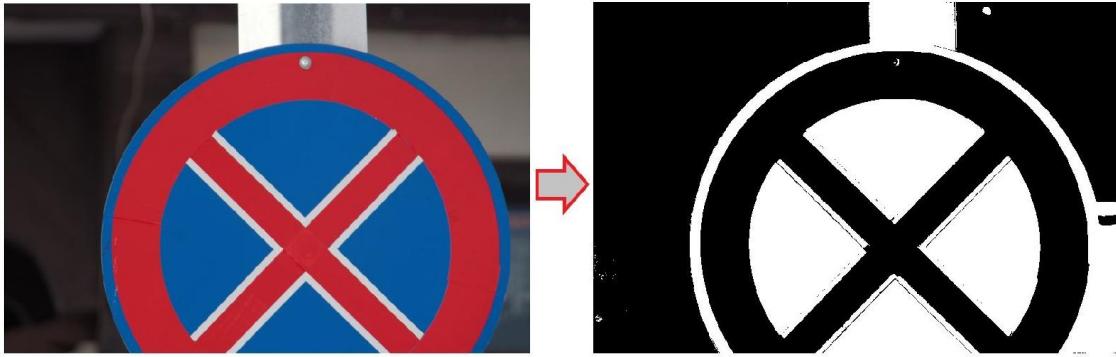
Now let's go back to our original histogram of only the blue colors and continue with the remaining steps.

4. We are ready to calculate the back-projection of our histogram on the hue channel that we had extracted in the initial step of our example. Here's how it's done:

```
int nimages = 1;
int channels[] = {0};
Mat backProject;
float rangeHue[] = {0, 180};
const float* ranges[] = {rangeHue};
double scale = 1.0;
bool uniform = true;
calcBackProject(&hue,
                nimages,
                channels,
                histogram,
                backProject,
                ranges,
                scale,
                uniform);
```

It's quite similar to how we created the back-projection of grayscale channels, but the range, in this case, is adjusted to correctly represent the possible values for the hue channel, which is 0 to 180.

The following image displays the result of such a back-projection, in which pixels with blue colors are extracted:



Note that pixels with grayscale colors (including white and black) might also have a value similar to the hue value that we want to extract, but since changing their hue value would not have any effect on their color, we can simply ignore them in our example case.

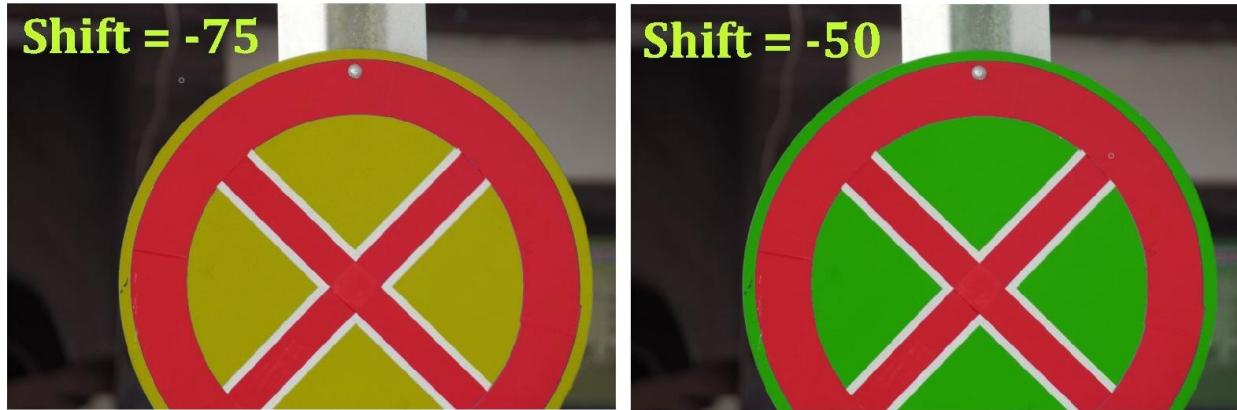
- Shift and change the hue in the pixels we extracted using the `calcBackProject` function. We simply need to loop through the pixels and shift their first channel with any desired value. The result obviously must be converted to BGR before it is suitable for being displayed. Here's how:

```

int shift = -50;
for(int i=0; i<imgHsv.rows; i++)
{
    for(int j=0; j<imgHsv.cols; j++)
    {
        if(backProject.at<uchar>(i, j))
        {
            imgHsv.at<Vec3b>(i, j)[0] += shift;
        }
    }
}
Mat imgHueShift;
cvtColor(imgHsv, imgHueShift, CV_HSV2BGR);

```

We used a `shift` value of `-50` in the preceding example, which will cause the blue pixels to turn to green, preserving their brightness, and so on. Using various `shift` values would result in different colors replacing the blue pixels. Here are two examples:



What we learned in the preceding example is the basis of many color-based detection and tracking algorithms, as we'll learn in the upcoming chapters. Being able to correctly extract pixels of a certain color, regardless of their brightness shift, is extremely handy. Brightness shift in a color is what happens when the lighting over an object of a certain color is changed, or during day and night, which is taken into account when hue is used instead of red, green, or blue channels in an RGB image.

Before proceeding to the final section of this chapter, it's worth noting that the exact same visualization method that we used for displaying the manually-made histogram of an imaginary hue channel can also be used to visualize the color histograms calculated from an image. Let's see how it's done with an example.

In the preceding example, right after the initial step, instead of forming the histogram manually, simply calculate it using the `calcHist` algorithm, as seen here:

```

int bins = 36;
int histSize[] = {bins};
int nimages = 1;
int dims = 1;
int channels[] = {0};
float rangeHue[] = {0, 180};
const float* ranges[] = {rangeHue};
bool uniform = true;
bool accumulate = false;
Mat histogram, mask;

calcHist(&hue,
        nimages,
        channels,
        mask,
        histogram,
        dims,
        histSize,
        ranges,
        uniform,
```

```
|     accumulate);
```

Changing the bin size effect is similar to what we saw in grayscale and single channel histograms, in the sense that it groups nearby values together. However, and in case of visualizing the hue channel, the nearby hue values will be grouped together, which results in a hue histogram that better represents similar colors in an image. The following example images depict the result of the preceding visualization, but with different `bins` values. From top to bottom, the `bins` value used to calculate each histogram is 360, 100, 36, and 7. Notice how the resolution of the histogram decreases as the bins value is decreased:



Choosing the right bins value completely depends on what type of objects you are dealing with and your definition of similar colors. What can be seen from the preceding image is that obviously choosing a very high bin value (such as 360) is not useful when we need at least some level of grouping of similar colors. On the other hand, choosing a very low bin size can result in an extreme grouping of colors in which calculating a back-projection would not produce an accurate result. Make sure to choose the bins value wisely, varying them for different subjects.

# Comparing histograms

Histograms can be compared with each other in order to get some insight into the content of an image. OpenCV allows histogram comparison using a method called `compareHist`, which requires the comparison method to be set first. The following example code depicts how this function can be used to calculate the result of comparison between two histograms calculated using previous calls to the `calcHist` function:

```
| HistCompMethods method = HISTCMP_CORREL;  
| double result = compareHist(histogram1, histogram2, method);
```

`histogram1` and `histogram2`, in the preceding example, are simply histograms of two different images, or different channels of an image. `method`, on the other hand, which must contain a valid entry from the `HistCompMethods` enum, defines the comparison algorithm used by the `compareHist` function and it can be any one of the following methods:

- `HISTCMP_CORREL`, for the Correlation method
- `HISTCMP_CHISQR`, for the Chi-square method
- `HISTCMP_INTERSECT`, for the Intersection method
- `HISTCMP_BHATTACHARYYA`, for the Bhattacharyya distance method
- `HISTCMP_HELLINGER`, same as `HISTCMP_BHATTACHARYYA`
- `HISTCMP_CHISQR_ALT`, for the Alternative Chi-square method
- `HISTCMP_KL_DIV`, for the Kullback-Leibler divergence method

You can refer to the latest OpenCV documentation to get more information about the mathematical details of each method, and how and what properties of histograms are used by them. The same can be said about the interpretation of the results of any method. Let's see what this means with an example. Using the following sample code, we can output the result of all histogram-comparison methods:

```
| cout << "HISTCMP_CORREL: " <<  
|   compareHist(histogram1, histogram2, HISTCMP_CORREL)  
|   << endl;  
| cout << "HISTCMP_CHISQR: " <<
```

```

    compareHist(histogram1, histogram2, HISTCMP_CHISQR)
        << endl;
    cout << "HISTCMP_INTERSECT: " <<
        compareHist(histogram1, histogram2, HISTCMP_INTERSECT)
            << endl;
    cout << "HISTCMP_BHATTACHARYYA: " <<
        compareHist(histogram1, histogram2, HISTCMP_BHATTACHARYYA)
            << endl;
    cout << "HISTCMP_HELLINGER: " <<
        compareHist(histogram1, histogram2, HISTCMP_HELLINGER)
            << endl;
    cout << "HISTCMP_CHISQR_ALT: " <<
        compareHist(histogram1, histogram2, HISTCMP_CHISQR_ALT)
            << endl;
    cout << "HISTCMP_KL_DIV: " <<
        compareHist(histogram1, histogram2, HISTCMP_KL_DIV)
            << endl;

```

We use the same example image we used throughout this chapter to calculate both `histogram1` and `histogram2`, or, in other words, if we compare one histogram with an equal histogram, here's what we would get:

```

HISTCMP_CORREL: 1
HISTCMP_CHISQR: 0
HISTCMP_INTERSECT: 426400
HISTCMP_BHATTACHARYYA: 0
HISTCMP_HELLINGER: 0
HISTCMP_CHISQR_ALT: 0
HISTCMP_KL_DIV: 0

```

Notice how distance- and divergence-based methods return a value of zero, while correlation returns a value of one, for exact correlation. All of the results in the preceding output mean equal histograms. Let's shed more light on this by calculating the histograms from the following two images:



The following results would be created if the image on the left is used to create `histogram1` and the image on the right is used to create `histogram2`, or, in other words, an arbitrary bright image is compared with an arbitrary dark image:

```
| HISTCMP_CORREL: -0.0449654
```

```
HISTCMP_CHISQR: 412918
HISTCMP_INTERSECT: 64149
HISTCMP_BHATTACHARYYA: 0.825928
HISTCMP_HELLINGER: 0.825928
HISTCMP_CHISQR_ALT: 1.32827e+06
HISTCMP_KL_DIV: 3.26815e+06
```

It's important to note that the order of the histograms being passed to the `compareHist` function matters in some cases, such as when `HISTCMP_CHISQR` is used as the method. Here are the results with `histogram1` and `histogram2` passed in reverse order to the `compareHist` function:

```
HISTCMP_CORREL: -0.0449654
HISTCMP_CHISQR: 3.26926e+06
HISTCMP_INTERSECT: 64149
HISTCMP_BHATTACHARYYA: 0.825928
HISTCMP_HELLINGER: 0.825928
HISTCMP_CHISQR_ALT: 1.32827e+06
HISTCMP_KL_DIV: 1.15856e+07
```

Comparing histograms is extremely useful, especially when we need to get a better and more meaningful impression of changes across various images. For instance, comparing histograms of consecutive frames from a camera can give us an idea of the intensity of change between those consecutive frames.

# Equalizing histograms

Using the functions and algorithms that we've learned so far, we can enhance the intensity distribution of images, or, in other words, adjust the brightness of too dark or overly bright images, among many other operations. In computer vision, the histogram-equalization algorithm is used for the exact same reason. This algorithm performs the following tasks:

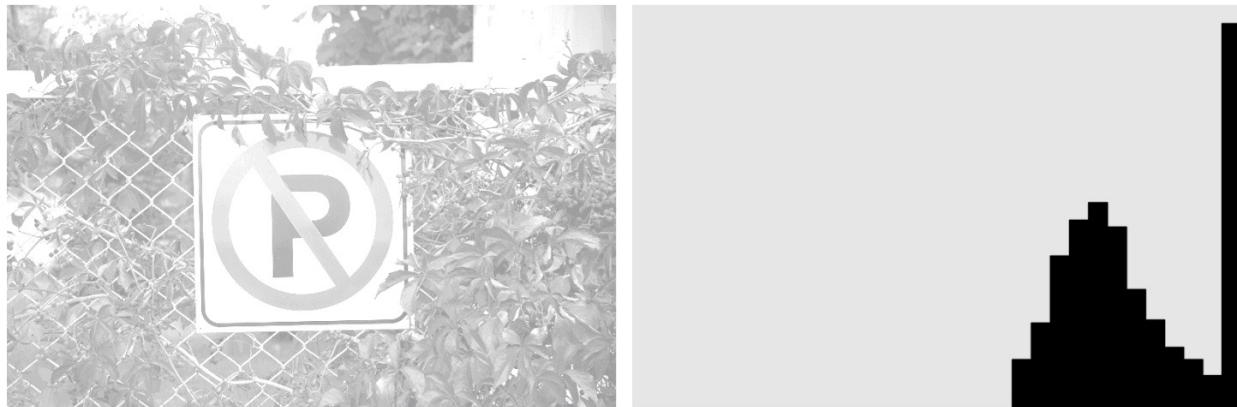
- Calculates the histogram of an image
- Normalizes the histogram
- Calculates the integral of the histogram
- Uses the updated histogram to modify the source image

Except the integral part, which is simply calculating the sum of the values in all bins, the rest is what we already performed in this chapter, in one way or another. OpenCV includes a function called `equalizeHist` that performs all of the mentioned operations and produces an image with an equalized histogram. Let's first see how this function is used and then try an example to see the effect for ourselves.

The following example codes depict how the `equalizeHist` function is used, which is extremely easy to use and requires no special parameters whatsoever:

```
| Mat equalized;  
| equalizeHist(gray, equalized);
```

Let's consider for instance that we have the following image, which is extremely overexposed (or bright), and its histogram, which is depicted on the right-hand side:



Using the `equalizeHist` function, we can get an image with better contrast and brightness. Here are the resulting image and histogram of the preceding example image when its histogram is equalized:



Histogram equalization is quite helpful when we have to deal with images that have the potential to be overexposed (too bright) or underexposed (too dark). For instance, x-ray scan images, where the details are only visible when the contrast and brightness is increased using a powerful backlight, or when we are working with video frames from an environment that can have intensive light changes, are examples of conditions in which histogram equalization can be used to make sure the rest of the algorithms always deal with the same, or just slightly different, brightness and contrast levels.

# Summary

We started this chapter by learning about histograms, what they are, and how they are calculated using the OpenCV library. We learned about the bin size of a histogram and how it can affect the accuracy or grouping of the values in a histogram. We continued to learn about visualizing histograms using the functions and algorithms we learned in [Chapter 4, Drawing, Filtering and Transformation](#). After going through various visualization types, we learned about back-projection and how we can update an image using a histogram. We learned about detecting pixels with a certain color and how to shift the hue value, and consequently the color of only those specific pixels. In the final sections of this chapter, we learned about comparing histograms and histogram-equalization algorithms. We performed hands-on examples for possible histogram comparison scenarios and enhanced the contrast and brightness of an overexposed image.

Histograms and how they are used to enhance and modify images using back-projection is one of the computer vision subjects that cannot be easily skipped over or missed, since it is the foundation of many image enhancement algorithms and techniques in photo-editing applications, or, as we'll see later on in the upcoming chapters, the basis of some of the most important real-time detection and tracking algorithms. What we learned in this chapter were a few of the most practical use cases of histograms and back-projection, but there is certainly much more to these algorithms if you start building real-life projects that make use of histograms.

In the next chapter, we'll use all of the concepts we learned in this and the previous chapters to work videos and video frames to detect objects with a certain color, track them in real-time, or detect motion in a video.

# Questions

1. Calculate the histogram of the second channel in a three-channel image. Use an optional bin size and a range of 0 to 100 for possible values of the second channel.
2. Create a histogram that can be used with the `calcBackProject` function to extract the darkest pixels from a grayscale image. Consider the darkest 25% possible pixel values as the grayscale intensities we are looking to extract.
3. In the previous question, what if we needed the darkest and brightest 25% to be excluded, instead of extracted, in a mask?
4. What is the hue value of the color red? How much should it be shifted to get the color blue?
5. Create a hue histogram that can be used to extract red-colored pixels from an image. Consider an offset of 50 for pixels that are considered reddish. Finally, visualize the hue histogram calculated.
6. Calculate the integral of a histogram.
7. Perform histogram equalization on a color image. Note that the `equalizeHist` function only supports histogram equalization of single-channel 8-bit grayscale images.

# Further reading

- *OpenCV 3.x with Python By Example – Second Edition* (<https://www.packtpub.com/application-development/opencv-3x-python-example-second-edition>)
- *Computer Vision with OpenCV 3 and Qt5* (<https://www.packtpub.com/application-development/computer-vision-opencv-3-and-qt5>)

# Video Analysis & Motion Detection and Tracking

As a computer vision developer, there is absolutely no way you can avoid dealing with video feeds from stored video files or cameras and other such sources. Treating video frames as individual images is one way to process videos, which surprisingly doesn't require much more effort or knowledge of the algorithms than what you have learned so far. For instance, you can apply a smoothening filter on a video, or in other words, a set of video frames, the same way as you would when you apply it on an individual image. The only trick here is that you must extract each frame from a video, as described in [Chapter 2, Getting Started with OpenCV](#). However, in computer vision, there are certain algorithms that are meant to work with consecutive video frames and the result on their operation depends not just on an individual image but also on the result of the same operation on the previous frames. Both of the algorithm types we just mentioned will be the main topics covered in this chapter.

After learning about histograms and back-projection images in the previous chapter, we are ready to take on computer vision algorithms that are used to detect and track objects in real-time. These algorithms highly rely on a firm understanding on all the topics we learned in [chapter 5, Back-Projection and Histograms](#). Based on this, we'll start this chapter with a couple of simple examples about how to use the computer vision algorithms we've learned so far, to process frames from a video file or camera, and then we'll move on to learn about two of the most famous object detection and tracking algorithms, the Mean Shift and CAM Shift algorithms. Then, we'll learn how to use the Kalman filter to correct the result of our object detection and tracking algorithms and how to remove noise from the results to get a better tracking result. We'll end this chapter by learning about motion analysis and background/foreground extraction.

In this chapter, we'll cover the following:

- How to apply filters and perform such operations on videos
- Using the Mean Shift algorithm to detect and track objects
- Using the CAM Shift algorithm to detect and track objects
- Using the Kalman filter to improve tracking results and remove noise
- Using the background and foreground extraction algorithms

# Technical requirements

- IDE to develop C++ or Python applications
- OpenCV library

Refer to [chapter 2, \*Getting Started with OpenCV\*](#), for more information about how to set up a personal computer and make it ready for developing computer vision applications using the OpenCV library.

You can use the following URL to download the source codes and examples for this chapter: <https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter06>.

# Processing videos

To be able to use any of the algorithms that we've learned so far on videos, we need to be able to read video frames and store them in `Mat` objects. We have already learned about how to deal with video files, cameras, and RTSP feeds in the initial chapters of this book. So, extending that, using what we learned in the previous chapters, we can use a code similar to the following, in order to apply colormaps to the video feed from the default camera on a computer:

```
VideoCapture cam(0);
// check if camera was opened correctly
if(!cam.isOpened())
    return -1;
// infinite loop
while(true)
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
        break;

    applyColorMap(frame, frame, COLORMAP_JET);

    // display the frame
    imshow("Camera", frame);

    // stop camera if space is pressed
    if(waitKey(10) == ' ')
        break;
}
cam.release();
```

Just as we learned in [Chapter 2, Getting Started with OpenCV](#), we simply need to create a `VideoCapture` object and read the video frames from the default camera (which has an index of zero). In the preceding example, we have added a single line of code that is responsible for applying a colormap on the extracted video frames. Try the preceding example code and you'll see the `COLORMAP_JET` colormap applied to every frame of the camera, much like what we learned in [Chapter 4, Drawing, Filtering, and Transformation](#), and finally the results are displayed in real-time. Pressing the spacebar should stop the video processing altogether.

Similarly, we can perform different video-processing algorithms in real-time, based on a specific key being pressed. Here's an example, replacing only the `for` loop in the preceding code, which results in the original video being displayed

unless either the *J* or *H* key is pressed:

```
int key = -1;
while(key != ' ')
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
        break;
    switch (key)
    {
    case 'j': applyColorMap(frame, frame, COLORMAP_JET);
        break;
    case 'h': applyColorMap(frame, frame, COLORMAP_HOT);
        break;
    }
    imshow("Camera", frame);
    int k = waitKey(10);
    if(k > 0)
        key = k;
}
```

The original output video of the camera will be displayed unless any of the mentioned keys are pressed. Pressing *J* will trigger `COLORMAP_JET`, while pressing *H* will trigger the `COLORMAP_HOT` colormap being applied to the camera frames. Similar to the previous example, pressing the spacebar key will stop the process. Also, pressing any keys other than space, *J*, or *H* will result in the original video being displayed.

 *The `applyColorMap` function in the preceding examples is just a random algorithm that is used to describe the technique used to process videos in real-time. You can use any of the algorithms you have learned in this book that perform on a single image. You can, for instance, write a program that performs a smoothening filter on the video, or Fourier transformation, or even a program that displays the Hue channel histogram in real-time. The use cases are infinite, however the method used is almost identical for all algorithms that perform a single complete operation on any individual image.*

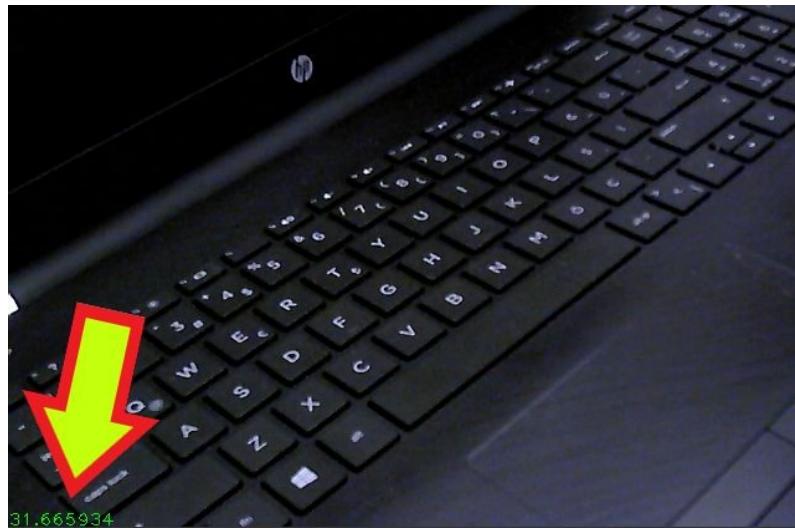
Besides performing an operation on individual video frames, one can also perform operations that depend on any number of consecutive frames. Let's see how this is done using a very simple, but extremely important, use case.

Let's assume we want to find the average brightness of the last 60 frames read from a camera at any given moment. Such a value is quite useful when we want to automatically adjust the brightness of the video when the content of the frames is extremely dark or extremely bright. In fact, a similar operation is usually performed by the internal processor of most digital cameras, and even the smartphone in your pocket. You can give it a try by turning on the camera on your smartphone and pointing it toward a light, or the sun, or by entering a very

dark environment. The following code demonstrates how the average of the brightness of the last 60 frames is calculated and displayed in the corner of the video:

```
VideoCapture cam(0);
if(!cam.isOpened())
    return -1;
vector<Scalar> avgs;
int key = -1;
while(key != ' ')
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
        break;
    if(avgs.size() > 60) // remove the first item if more than 60
        avgs.erase(avgs.begin());
    Mat frameGray;
    cvtColor(frame, frameGray, CV_BGR2GRAY);
    avgs.push_back( mean(frameGray) );
    Scalar allAvg = mean(avgs);
    putText(frame,
            to_string(allAvg[0]),
            Point(0,frame.rows-1),
            FONT_HERSHEY_PLAIN,
            1.0,
            Scalar(0,255,0));
    imshow("Camera", frame);
    int k = waitKey(10);
    if(k > 0)
        key = k;
}
cam.release();
```

For the most part, this example code is quite similar to the examples we saw earlier in this chapter. The main difference here is that we are storing the average of the last 60 frames that are calculated using the OpenCV mean function in a vector of scalar objects, and then we calculate the average of all averages. The calculated value is then drawn on the input frame using the `putText` function. The following image depicts a single frame that is displayed when the preceding example code is executed:



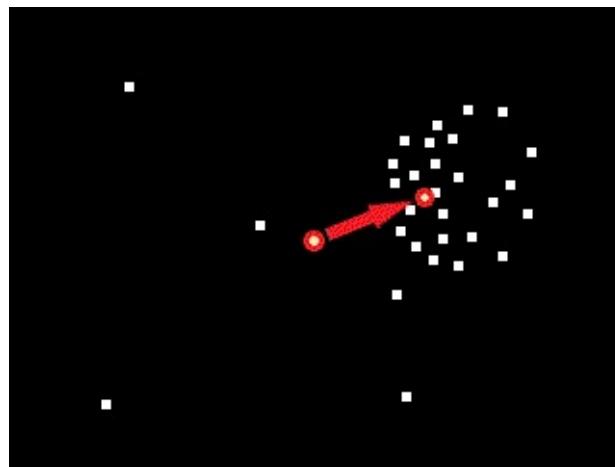
Notice the value displayed in the bottom-left corner of the image, which will start to decrease when the content of the video frames becomes darker and increase when they become brighter. Based on this result, you can, for instance, change the brightness value or warn the user of your application that the content is too dark or bright, and so on.

The examples in this initial section of the chapter were meant to teach you the idea of processing individual frames using the algorithms you've learned in the previous chapters, and a few simple programming techniques used to calculate a value based on consecutive frames. In the following sections of this chapter, we'll be learning about some of the most important video-processing algorithms, namely object detection and tracking algorithms, which depend on the concepts and techniques we learned in this section and the previous chapters of this book.

# Understanding the Mean Shift algorithm

The Mean Shift algorithm is an iterative algorithm that can be used to find the maxima of a density function. A very rough translation of the preceding sentence to computer vision terminology would be the following—the Mean Shift algorithm can be used to find an object in an image using a back-projection image. But how is it achieved in practice? Let's walk through this step by step. Here are the individual operations that are performed to find an object using the Mean Shift algorithm, in order:

1. The back-projection of an image is created using a modified histogram to find the pixels that are most likely to contain our object of interest. (It is also common to filter the back-projection image to get rid of unwanted noise, but this is an optional operation to improve the results.)
2. An initial search window is needed. This search window will contain our object of interest after a number of iterations, which we'll get to in the next step. After each iteration, the search window is updated by the algorithm. The updating of the search window happens by calculating the mass center of the search window in the back-projection image, and then shifting the current center point of the search window to the mass center of the window. The following picture demonstrates the concept of the mass center in a search window and how the shifting happens:



The two points at the two ends of the arrow in the preceding picture correspond to the search-window center and mass center.

3. Just like any iterative algorithm, some termination criteria are required by the Mean Shift algorithm to stop the algorithm when the results are as expected or when reaching an accepted result does not happen as fast as needed. So, the number of iterations and an epsilon value are used as termination criteria. Either by reaching the number of iterations in the algorithm or by finding a shift distance that is smaller than the given epsilon value (convergence), the algorithm will stop.

Now, let's see a hands-on example of how this algorithm is used in practice by using the OpenCV library. The `meanshift` function in OpenCV implements the Mean Shift algorithm almost exactly as it was described in the preceding steps. This function requires a back-projection image, a search window, and the termination criteria, and it is used as seen in the following example:

```
Rect srchWnd(0, 0, 100, 100);
TermCriteria criteria(TermCriteria::MAX_ITER
                      + TermCriteria::EPS,
                      20, // number of iterations
                      1.0 // epsilon value
                      );
// Calculate back-projection image
meanShift(backProject,
           srchWnd,
           criteria);
```

`srchWnd` is a `Rect` object, which is simply a rectangle that must contain an initial value that is used and then updated by the `meanshift` function. `backProjection` must contain a proper back-projection image that is calculated with any of the methods that we learned in [Chapter 5, Back-Projection and Histograms](#). The `TermCriteria` class is an OpenCV class that is used by iterative algorithms that require similar termination criteria. The first parameter defines the type of the termination criteria, which can be `MAX_ITER` (same as `COUNT`), `EPS`, or both. In the preceding example, we have used the termination criteria of `20` iterations and an epsilon value of `1.0`, which of course can be changed depending on the environment and application. The most important thing to note here is that a higher number of iterations and a lower epsilon can yield more accurate results, but it can also lead to slower performance, and vice versa.

The preceding example is just a demonstration of how the `meanshift` function is

called. Now, let's walk through a complete hands-on example to learn our first real-time object-tracking algorithm:

1. The structure of the tracking example we'll create is quite similar to the previous examples in this chapter. We need to open a video, or a camera, on the computer using the `VideoCapture` class and then start reading the frames, as seen here:

```
VideoCapture cam(0);
if(!cam.isOpened())
    return -1;

int key = -1;
while(key != ' ')
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
        break;

    int k = waitKey(10);
    if(k > 0)
        key = k;
}

cam.release();
```

Again, we have used the `waitKey` function to stop the loop if the spacebar key is pressed.

2. We're going to assume that our object of interest has a green color. So, we're going to form a hue histogram that contains only the green colors, as seen here:

```
int bins = 360;
int grnHue = 120; // green color hue value
int hueOffset = 50; // the accepted threshold
Mat histogram(bins, 1, CV_32FC1);
for(int i=0; i<bins; i++)
{
    histogram.at<float>(i, 0) =
        (i > grnHue - hueOffset)
        &&
        (i < grnHue + hueOffset)
        ?
            255.0 : 0.0;
}
```

This needs to happen before entering the process loop, since our histogram is going to stay constant throughout the whole process.

3. One last thing to take care of before entering the actual process loop and the tracking code is the termination criteria, which will stay constant throughout the whole process. Here's how we'll create the required termination criteria:

```
Rect srchWnd(0,0, 100, 100);
TermCriteria criteria(TermCriteria::MAX_ITER
    + TermCriteria::EPS,
    20,
    1.0);
```



*The initial value of the search window is quite important when using the Mean Shift algorithm to track objects, since this algorithm always makes an assumption about the initial position of the object to be tracked. This is an obvious downside of the Mean Shift algorithm, which we'll learn how to deal with later on in this chapter when we discuss the CAM Shift algorithm and its implementation in the OpenCV library.*

4. After each frame is read in the `while` loop we're using for the tracking code, we must calculate the back-projection image of the input frame using the green hue histogram that we created. Here's how it's done:

```
Mat frmHsv, hue;
vector<Mat> hsvChannels;
cvtColor(frame, frmHsv, COLOR_BGR2HSV);
split(frmHsv, hsvChannels);
hue = hsvChannels[0];

int nimages = 1;
int channels[] = {0};
Mat backProject;
float rangeHue[] = {0, 180};
const float* ranges[] = {rangeHue};
double scale = 1.0;
bool uniform = true;
calcBackProject(&hue,
    nimages,
    channels,
    histogram,
    backProject,
    ranges,
    scale,
    uniform);
```

You can refer to [Chapter 5, Back-Projection and Histograms](#), for more detailed instructions about calculating the back-projection image.

5. Call the `meanshift` function to update the search window using the back-projection image and provided termination criteria, as seen here:

```
meanShift(backProject,
    srchWnd,
    criteria);
```

6. To visualize the search window, or in other words the tracked object, we need to draw the search-window rectangle on the input frame. Here's how you can do this by using the rectangle function:

```
rectangle(frame,
          srchWnd, // search window rectangle
          Scalar(0,0,255), // red color
          2 // thickness
        );
```

We can do the same on back-projection image result, however, first we need to convert the back-projection image to BGR color space.

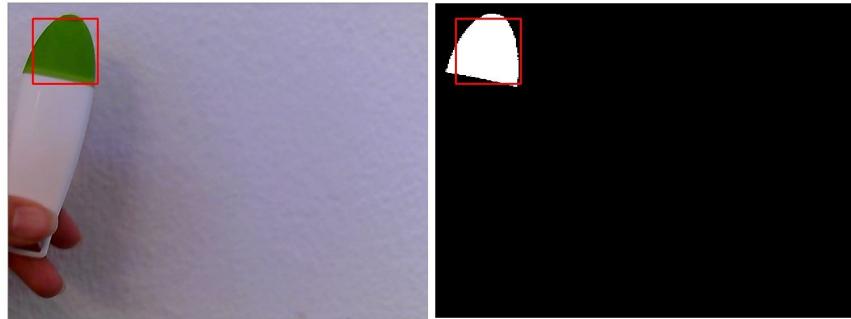
Remember that the result of the back-projection image contained a single channel image with the same depth as the input image. Here's how we can draw a red rectangle at the search-window position on the back-projection image:

```
cvtColor(backProject, backProject, COLOR_GRAY2BGR);
rectangle(backProject,
          srchWnd,
          Scalar(0,0,255),
          2);
```

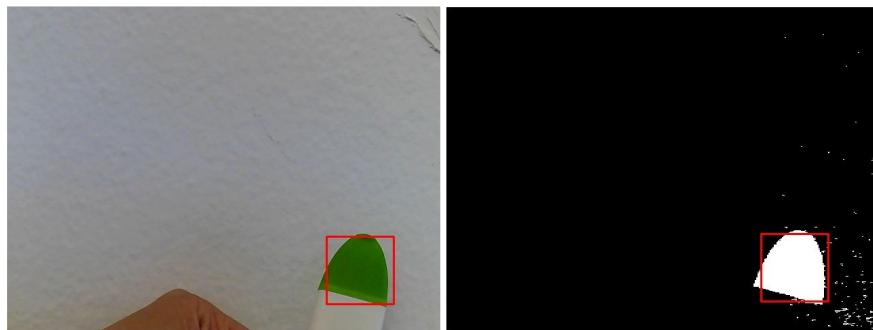
7. Add the means to switch between the back-projection and original video frame using the *B* and *V* keys. Here's how it's done:

```
switch(key)
{
  case 'b': imshow("Camera", backProject);
              break;
  case 'v': default: imshow("Camera", frame);
              break;
}
```

Let's give our program a try and see how it performs when executed in a slightly controlled environment. The following picture demonstrates the initial position of the search window and our green object of interest, both in the original frame view and the back-projection view:



Moving the object around will cause the `meanshift` function to update the search window and consequently track the object. Here's another result, depicting the object tracked to the bottom-right corner of the view:



Notice the small amount of noise that can be seen in the corner, which would be taken care of by the `meanshift` function since the mass center is not affected too much by it. However, as mentioned previously, it is a good idea to perform some sort of filtering on the back-projection image to get rid of noise. For instance, and in case of noise similar to what we have in the back-projection image, we can use the `GaussianBlur` function, or even better, the `erode` function, to get rid of unwanted pixels in the back-projection image. For more information on how to use filtering functions, you can refer to [Chapter 4, Drawing, Filtering, and Transformation](#).

In such tracking applications, we usually need to observe, record, or in any way process the route that the object of interest has taken before any given moment and for a desired period of time. This can be simply achieved by using the center point of the search window, as seen in the following example:

```

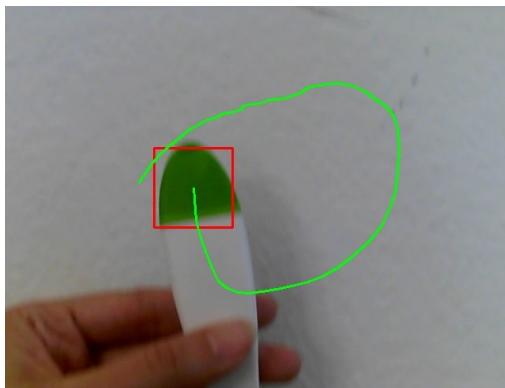
Point p(srchWnd.x + srchWnd.width/2,
        srchWnd.y + srchWnd.height/2);
route.push_back(p);
if(route.size() > 60) // last 60 frames
    route.erase(route.begin()); // remove first element

```

Obviously, the `route` is a vector of `Point` objects. `route` needs to be updated after the `meanshift` function call, and then we can use the following call to the `polyline` function in order to draw the `route` over the original video frame:

```
polyline(frame,  
        route, // the vector of Point objects  
        false, // not a closed polyline  
        Scalar(0,255,0), // green color  
        2 // thickness  
);
```

The following picture depicts the result of displaying the tracking route (for the last 60 frames) on the original video frames read from the camera:



Now, let's address some issues that we observed while working with the `meanshift` function. First of all, it is not convenient to create the hue histogram manually. A flexible program should allow the user to choose the object they want to track, or at least allow the user to choose the color of the object of interest conveniently. The same can be said about the search window size and its initial position. There are a number of ways to deal with such issues and we're going to address them with a hands-on example.

When using the OpenCV library, you can use the `setMouseCallback` function to customize the behavior of mouse clicks on an output window. This can be used in combination with a few simple methods, such as `bitwise_not` to mimic an easy-to-use object selection for the users. `setMouseCallback`, as it can be guessed from its name, sets a callback function to handle the mouse clicks on a given window.

The following callback function in conjunction with the variables defined here

can be used to create a convenient object selector:

```
bool selecting = false;
Rect selection;
Point spo; // selection point origin

void onMouse(int event, int x, int y, int flags, void*)
{
    switch(event)
    {
        case EVENT_LBUTTONDOWN:
        {
            spo.x = x;
            spo.y = y;
            selection.x = spo.x;
            selection.y = spo.y;
            selection.width = 0;
            selection.height = 0;
            selecting = true;

        } break;
        case EVENT_LBUTTONUP:
        {
            selecting = false;
        } break;
        default:
        {
            selection.x = min(x, spo.x);
            selection.y = min(y, spo.y);
            selection.width = abs(x - spo.x);
            selection.height = abs(y - spo.y);
        } break;
    }
}
```

The `event` contains an entry from the `MouseEventTypes` enum, which describes whether a mouse button was pressed or released. Based on such a simple event, we can decide when the user is actually selecting an object that's visible on the screen. This is depicted as follows:

```
if(selecting)
{
    Mat sel(frame, selection);
    bitwise_not(sel, sel); // invert the selected area

    srchWnd = selection; // set the search window

    // create the histogram using the hue of the selection
}
```

This allows a huge amount of flexibility for our applications, and the code is bound to work with objects of any color. Make sure to check out the example codes for this chapter from the online Git repository for a complete example project that uses all the topics we've learned so far in this chapter.

Another method of selecting an object or a region on an image is by using the `selectROI` and `selectROIs` functions in the OpenCV library. These functions allow the user to select a rectangle (or rectangles) on an image using simple mouse clicks and drags. Note that the `selectROI` and `selectROIs` functions are easier to use than handling mouse clicks using callback functions, however they do not offer the same amount of power, flexibility, and customization.

Before moving on to the next section, let's recall that `meanshift` does not handle an increase or decrease in the size of the object that is being tracked, nor does it take care of the orientation of the object. These are probably the main issues that have led to the development of a more sophisticated version of the Mean Shift algorithm, which is the next topic we're going to learn about in this chapter.

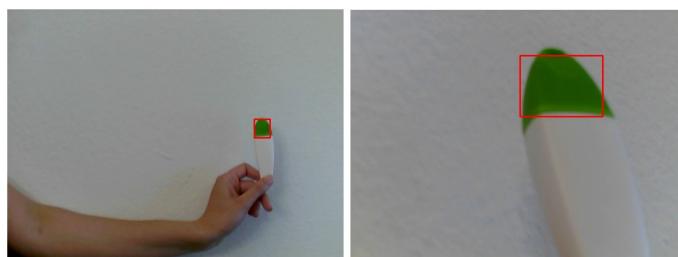
# Using the Continuously Adaptive Mean (CAM) Shift

To overcome the limitations of the Mean Shift algorithm, we can use an improved version of it, which is called the **Continuously Adaptive Mean** Shift, or simply the **CAM** Shift algorithm. OpenCV contains the implementation for the CAM Shift algorithm in a function named `camshift`, which is used almost in an identical manner to the `meanshift` function. The input parameters of the `camshift` function are the same as `meanshift`, since it also uses a back-projection image to update a search window using a given set of termination criteria. In addition, `camshift` also returns a `RotatedRect` object, which contains both the search window and its angle.

Without using the returned `RotatedRect` object, you can simply replace any call to the `meanshift` function with `camshift`, and the only difference would be that the results will be scale-invariant, meaning the search window will become bigger if the object is nearer (or bigger) and vice versa. For instance, we can replace the call to the `meanshift` function in the preceding example code for the Mean Shift algorithm with the following:

```
| CamShift(backProject,  
|     srchWnd,  
|     criteria);
```

The following images depict the result of replacing the `meanshift` function with `camshift` in the example from the previous section:



Notice that the results are now scale-invariant, even though we didn't change anything except replace the mentioned function. As the object moves farther

away from the camera, or becomes smaller, the same Mean Shift algorithm is used to calculate its position, however, this time the search window is resized to fit the exact size of the object, and the rotation is calculated, which we didn't use. To be able to use the rotation value of the object, we need to store the result of the `CamShift` function in a `RotatedRect` object first, as seen in the following example:

```
RotatedRect rotRect = CamShift(backProject,  
                               srchWnd,  
                               criteria);
```

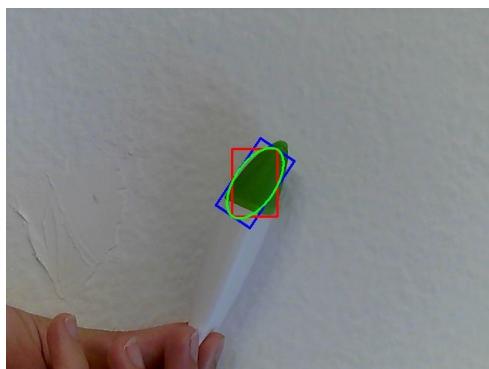
To draw a `RotatedRect` object, or in other words a rotated rectangle, you must use the `points` method of `RotatedRect` to extract the consisting 4 points of the rotated rectangle first, and then draw them all using the `line` function, as seen in the following example:

```
Point2f rps[4];  
rotRect.points(rps);  
for(int i=0; i<4; i++)  
    line(frame,  
         rps[i],  
         rps[(i+1)%4],  
         Scalar(255,0,0),// blue color  
         2);
```

You can also use a `RotatedRect` object to draw a rotated ellipse that is covered by the rotated rectangle. Here's how:

```
ellipse(frame,  
        rotRect,  
        Scalar(255,0,0),  
        2);
```

The following image displays the result of using the `RotatedRect` object to draw a rotated rectangle and ellipse at the same time, over the tracked object:



In the preceding image, the red rectangle is the search window, the blue

rectangle is the resulting rotated rectangle, and the green ellipse is drawn by using the resulting rotated rectangle.

To summarize, we can say that `camshift` is far better suited to dealing with objects of varying size and rotation than `meanshift`, however, there are still a couple of possible enhancements that can be done when using the `camshift` algorithm. First things first, the initial window size still needs to be set, but since `camshift` is taking care of the size changes, then we can simply set the initial window size to be the same as the whole image size. This would help us avoid having to deal with the initial position and size of the search window. If we can also create the histogram of the object of interest using a previously saved file on disk or any similar method, then we will have an object detector and tracker that works out of the box, at least for all the cases where our object of interest has a visibly different color than the environment.

Another huge improvement to such a color-based detection and tracking algorithm can be achieved by using the `inRange` function to enforce a threshold on the S and V channels of the HSV image that we are using to calculate the histogram. The reason is that in our example, we simply used the **hue** (or the **H**, or the first) channel, and we didn't take into account the high possibility of having extremely dark or bright pixels that might have the same hue as our object of interest. This can be done by using the following code when calculating the histogram of the object to be tracked:

```
int lbHue = 00 , hbHue = 180;
int lbSat = 30 , hbSat = 256;
int lbVal = 30 , hbVal = 230;

Mat mask;
inRange(objImgHsv,
        Scalar(lbHue, lbSat, lbVal),
        Scalar(hbHue, hbSat, hbVal),
        mask);

calcHist(&objImgHue,
         nimages,
         channels,
         mask,
         histogram,
         dims,
         histSize,
         ranges,
         uniform);
```

In the preceding example code, the variables starting with `lb` and `hb` refer to the lower bound and higher bound of the values that are allowed to pass the `inRange`

function. `objImgHSV` is obviously a `Mat` object containing our object of interest, or a ROI that contains our object of interest. `objImgHue` is the first channel of `objImgHSV`, which is extracted using a previous call to the `split` function. The rest of the parameters are nothing new, and you've already used them in previous calls to the functions used in this example.

Combining all of the algorithms and techniques described in this section can help you create an object-detector, or even a face-detector and tracker that can work in realtime and with stunning speed. However, you might still need to account for the noise that will interfere, especially with the tracking, which is almost inevitable because of the nature of color-based or histogram-based trackers. One of the most widely used solutions to these issues is the subject of the next section in this chapter.

# Using the Kalman filter for tracking and noise reduction

The Kalman filter is a popular algorithm that is used for reducing the noise of a signal, such as the result of the tracking algorithm that we used in the preceding section. To be precise, the Kalman filter is an estimation algorithm that is used to predict the next state of a signal based on previous observations. Digging deep into the definition and details of the Kalman filter would require a chapter of its own, but we'll try to walk through this simple, yet extremely powerful, algorithm with a couple of hands-on examples to learn how it is used in practice.

For the first example, we're going to write a program that tracks the mouse cursor while it is moved on a canvas, or the OpenCV window. The Kalman filter is implemented using the `KalmanFilter` class in OpenCV and it includes all (and many more) of the Kalman filter implementation details, which we'll discuss in this section.

First of all, `KalmanFilter` must be initialized with a number of dynamic parameters, measurement parameters, and control parameters, in addition to the type of the underlying data used in the Kalman filter itself. We're going to ignore control parameters, since they are outside the scope of our examples, so we'll set them simply to zero. As for the data type, we'll go for the default 32-bit float, or `cv_32F` in terms of OpenCV types. Dynamic parameters in a 2D movement, which is the case with our example, correspond to the following:

- $X$ , or position in  $x$  direction
- $Y$ , or position in  $y$  direction
- $X'$ , or velocity in  $x$  direction
- $Y'$ , or velocity in  $y$  direction



*A higher dimensionality of the parameters can also be used, which would then cause the preceding list to be followed by  $X''$  (acceleration in  $x$  direction) and so on.*

As for the measurement parameters, we'll simply have  $X$  and  $Y$ , which correspond to the mouse position in our first example. Keeping in mind what

was said about dynamic and measurement parameters, here's how we can initialize a `KalmanFilter` class instance (object) that fits for tracking a point on a 2D space:

```
| KalmanFilter kalman(4, // dynamic parameters: X,Y,X',Y'  
|                     2 // measurement parameters: X,Y  
| );
```

Note that in this example, control parameters and type parameters are simply ignored and set to their default values, otherwise we could have written the same code as seen here:

```
| KalmanFilter kalman(4, 2, 0, CV_32F);
```

The `KalmanFilter` class requires a transition matrix to be set before it is correctly usable. This transition matrix is used for calculating (and updating) the estimated, or next, state of the parameters. We'll be using the following transition matrix in our example for tracking the mouse position:

```
| Mat<float> tm(4, 4); // transition matrix  
| tm << 1,0,1,0, // next x = 1X + 0Y + 1X' + 0Y'  
|           0,1,0,1, // next y = 0X + 1Y + 0X' + 1Y'  
|           0,0,1,0, // next x'= 0X + 0Y + 1X' + 0Y  
|           0,0,0,1; // next y'= 0X + 0Y + 0X' + 1Y'  
| kalman.transitionMatrix = tm;
```

After completing the steps required for this example, it would be wise to return here and update the values in the transition matrix and observe the behavior of the Kalman filter. For instance, try to update the matrix row that corresponds to the estimated Y (marked as `next_y` in the comments) and you'll notice that the tracked position Y value is affected by it. Try experimenting with all of the values in the transition matrix for a better understanding of its effect.

Besides the transition matrix, we also need to take care of the initialization of the dynamic parameters' state and measurements, which are the initial mouse positions in our example. Here's how we initialize the mentioned values:

```
| Mat<float> pos(2,1);  
| pos.at<float>(0) = 0;  
| pos.at<float>(1) = 0;  
  
| kalman.statePre.at<float>(0) = 0; // init x  
| kalman.statePre.at<float>(1) = 0; // init y  
| kalman.statePre.at<float>(2) = 0; // init x'  
| kalman.statePre.at<float>(3) = 0; // init y'
```

As you'll see later on, the `KalmanFilter` class requires a vector instead of a `Point` object, since it is designed to work with higher dimensionalities too. For this reason, we'll update the `pos` vector in the preceding code snippet with the last mouse positions before performing any calculations. Other than the initializations that we just mentioned, we also need to initialize the measurement matrix of the Kalman filter. This is done as seen here:

```
| setIdentity(kalman.measurementMatrix);
```

The `setIdentity` function in OpenCV is simply used to initialize matrices with a scaled identity matrix. If only a single matrix is provided as a parameter to the `setIdentity` function, it will set to the identity matrix, however if a second `Scalar` is provided in addition, then all elements of the identity matrix will be multiplied (or scaled) using the given `Scalar` value.

One last initialization is the process noise covariance. We'll use a very small value for this, which causes a tracking with a natural-movement feeling, although with a little bit of overshoot when tracking. Here's how we initialize the process-noise covariance matrix:

```
| setIdentity(kalman.processNoiseCov,  
|             Scalar::all(0.000001));
```

It is also common to initialize the following matrices before using the `KalmanFilter` class:

- `controlMatrix` (not used if the control parameter count is zero)
- `errorCovPost`
- `errorCovPre`
- `gain`
- `measurementNoiseCov`

Using all of the matrices mentioned in the preceding list will provide a huge amount of customization to the `KalmanFilter` class, but it also requires a great amount of knowledge about the type of noise-filtering and tracking that is needed and the environment that the filter will be implemented in. These matrices and their usage have deep roots in control theory, and control science in general, which is a topic for another book. Note that in our example, we'll simply use the default values of the mentioned matrices, thus we have ignored them altogether.

The next thing we need in our tracking example using the Kalman filter is to set up a window on which we can track mouse movements. We are assuming that the position of the mouse on the window is the position of a detected and tracked object and we'll use our `KalmanFilter` object to predict and de-noise these detections, or in Kalman-filter-algorithm terminology, we are going to correct these measurements. We can use the `namedWindow` function to create a window using OpenCV. Consequently, the `setMouseCallback` function can be used to assign a callback function for mouse interactions with that specific window. Here's how we can do it:

```
| string window = "Canvas";
| namedWindow(window);
| setMouseCallback(window, onMouse);
```

We've used the word `Canvas` for the window, but obviously you can use any other name you like. `onMouse` is the callback function that will be assigned to react on mouse interactions with this window. It is defined like this:

```
| void onMouse(int, int x, int y, int, void*)
| {
|     objectPos.x = x;
|     objectPos.y = y;
| }
```

`objectPos` in the preceding code snippet is the `Point` object that is used to store the last position of the mouse on the window. It needs to be defined globally in order to be accessible by both `onMouse` and the main function in which we'll use the `KalmanFilter` class. Here's the definition:

```
| Point objectPos;
```

Now, for the actual tracking, or to use the more correct terminology, correcting the measurements that contain noise, we need to use the following code, which is followed by the required explanations:

```
| vector<Point> trackRoute;
| while(waitKey(10) < 0)
| {
|     // empty canvas
|     Mat canvas(500, 1000, CV_8UC3, Scalar(255, 255, 255));
|
|     pos(0) = objectPos.x;
|     pos(1) = objectPos.y;
|
|     Mat estimation = kalman.correct(pos);
|
|     Point estPt(estimation.at<float>(0),
```

```

        estimation.at<float>(1));

    trackRoute.push_back(estPt);
    if(trackRoute.size() > 100)
        trackRoute.erase(trackRoute.begin());

    polylines(canvas,
              trackRoute,
              false,
              Scalar(0,0,255),
              5);

    imshow(window, canvas);

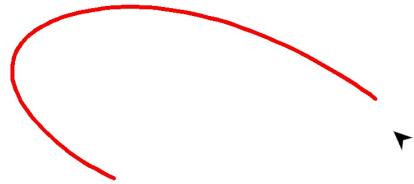
    kalman.predict();
}

```

In the preceding code, we are using the `trackRoute` vector to record the estimations over the last `100` frames. Pressing any key will cause the `while` loop, and consequently the program, to return. Inside the loop, and where we actually use the `KalmanFilter` class, we simply perform the following operations in order:

1. Create an empty `Mat` object to be used as a canvas to draw, and for the content of the window on which the tracking will happen
2. Read the `objectPos`, which contains the last position of the mouse on the window and store it in the `pos` vector, which is usable with the `KalmanFilter` class
3. Read an estimation using the correct method of the `KalmanFilter` class
4. Convert the result of the estimation back to a `Point` object that can be used for drawing
5. Store the estimated point (or the tracked point) in the `trackRoute` vector, and make sure the number of items in the `trackRoute` vector doesn't exceed `100`, since that is the number of frames for which we want to keep a record of estimated points
6. Use the `polylines` function to draw the route, stored as `Point` objects in `trackRoute`
7. Display the results using the `imshow` function
8. Update the internal matrices of the `KalmanFilter` class using the `predict` function

Try executing the tracking program and move your mouse cursor around the window that is shown. You'll notice a smooth tracking result, which is drawn using the thick red line that's visible in the following screenshot:



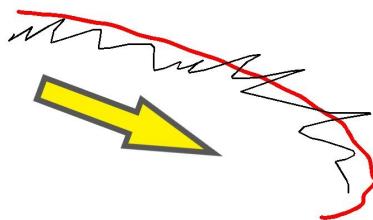
Note that the position of the mouse cursor is ahead of the tracking, and the noise in mouse movement is almost completely removed. It is a good idea to try to visualize the mouse movement for a better comparison between the `KalmanFilter` results and actual measurements. Simply add the following code to the loop after the point where `trackRoute` was drawn in the preceding code:

```
| mouseRoute.push_back(objectPos);  
| if(mouseRoute.size() > 100)  
|     mouseRoute.erase(mouseRoute.begin());  
| polylines(canvas,  
|           mouseRoute,  
|           false,  
|           Scalar(0,0,0),  
|           2);
```

And obviously, you need to define the `mouseRoute` vector before entering the `while` loop, as seen here:

```
| vector<Point> mouseRoute;
```

Let's try the same application with this minor update and see how the results compare to each other. Here's another screenshot depicting the results of actual mouse movements and corrected movements (or tracked, or filtered, depending on the terminology) drawn in the same window:

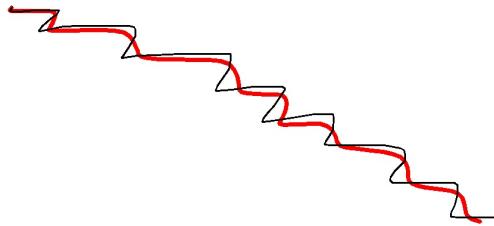


In the preceding result, the arrow is simply used to depict the overall direction of an extremely noisy measurement (mouse movement or detected object position, depending on the application), which is drawn using a thin black color, and the

corrected results using the Kalman filter algorithm, which are drawn with a thick red line in the image. Try moving you mouse around and comparing the results visually. Remember what we mentioned about the `KalmanFilter` internal matrices and how you need to set their values according to the use case and application? For instance, a bigger process-noise covariance would have resulted in less de-noising and consequently less filtering. Let's try setting the process noise covariance value to `0.001`, instead of the previous `0.000001` value, try the same program once again, and compare the results. Here's how you can set the process-noise covariance:

```
| setIdentity(kalman.processNoiseCov,  
|             Scalar::all(0.001));
```

Now, try running the program once again and you can easily notice that less de-noising is happening as you move the mouse cursor around the window:



By now, you can probably guess that the result of setting an extremely high value for process-noise covariance would be almost the same as using no filter at all. This is the reason why setting correct values for the Kalman filter algorithm is extremely important and also the reason why it depends so much on the application. However, there are methods for setting most of those parameters programmatically, and even dynamically while the tracking is being done to achieve the best results. For instance, using a function that is able to determine the possible amount of noise at any given moment, we can dynamically set the process noise covariance to high or low values for less and more de-noising of the measurements.

Now, let's use `KalmanFilter` to perform a real-life tracking correction on objects using the `CamShift` function, instead of mouse movements. It's important to note that the applied logic is exactly the same. We need to initialize a `KalmanFilter` object and set its parameters according to the amount of noise and so on. For

simplicity, you can start off with the exact same set of parameters that we set for tracking the mouse cursor in the previous example, and then try to adjust them. We need to start by creating the same tracking program that we wrote in the previous sections. However, right after calling `camShift` (or `meanshift` function) to update the search window, instead of displaying the results, we'll perform a correction using the `KalmanFilter` class to de-noise the results. Here's how it is done with a similar example code:

```
CamShift(backProject,
         srchWnd,
         criteria);

Point objectPos(srchWnd.x + srchWnd.width/2,
                 srchWnd.y + srchWnd.height/2);

pos(0) = objectPos.x;
pos(1) = objectPos.y;

Mat estimation = kalman.correct(pos);

Point estPt(estimation.at<float>(0),
            estimation.at<float>(1));

drawMarker(frame,
           estPt,
           Scalar(0, 255, 0),
           MARKER_CROSS,
           30,
           2);

kalman.predict();
```

You can refer to the online source code repository of this chapter for the complete example project containing the preceding code, which is almost identical to what you saw in the previous sections, except for the simple fact that `KalmanFilter` is used to correct the detected and tracked object position. As you can see, `objectPos`, which was previously read from the mouse movement position, is now set to the central point of the search window. After that, the `correct` function is called to perform an estimation and the results are displayed by drawing a green cross mark for the corrected tracking result. Besides the main advantage of using the Kalman filter algorithm, which is useful for getting rid of noise in detection and tracking results, it can also help with cases where detection is momentarily lost or impossible. Although losing detection is, technically speaking, an extreme case of noise, we're trying to point out the difference in terms of computer vision.

By going through a few examples and also experimenting with your own

projects where the Kalman filter can be of help and trying different set of parameters for it, you'll instantly understand its long-standing popularity for whenever a practical algorithm for correcting a measurement (that contains noise) is required. What we learned in this section was a fairly simple case of how the Kalman filter is used (which was enough for our use case), but it is important to note that the same algorithm can be used to de-noise measurements of higher dimensionalities and with much more complexity.

# How to extract the background/foreground

The segmentation of background and foreground content in images is one of the most important video and motion analysis topics, and there has been a huge amount of research done in this area to provide some very practical and easy-to-use algorithms, which we're going to learn in the final section of this chapter. Current versions of OpenCV include the implementation of two background segmentation algorithms.



*To use a terminology that is shorter, clearer, and more compatible with OpenCV functions and classes, we'll refer to background/foreground extraction and background/foreground segmentation simply as background segmentation.*

The following two algorithms are available by default to be used for background segmentation using OpenCV:

- `BackgroundSubtractorKNN`
- `BackgroundSubtractorMOG2`

Both of these classes are subclasses of `BackgroundSubtractor`, which contains all of the required interfaces that one can expect from a proper background segmentation algorithm, which we'll get to later on. This simply allows us to use polymorphism to switch between algorithms that produce the same results and can be used in a very similar fashion for the exact same reason. The `BackgroundSubtractorKNN` class implements the K-nearest neighbors background segmentation algorithm, which is used in the case of a low foreground pixel count. `BackgroundSubtractorMOG2`, on the other hand, implements the Gaussian mixture-based background-segmentation algorithm. You can refer to the OpenCV online documentation for detailed information about the internal behavior and implementation of these algorithms. It's also a good idea to go through the referred articles for both of these algorithms, especially if you are looking for a custom background segmentation algorithm of your own.



*Besides the algorithms we already mentioned, there are many more algorithms that can be used for background segmentation using OpenCV, which are included in the extra module,*



*bgsegm*. We'll omit those algorithms, since their usage is quite similar to the algorithms we'll be talking about in this section, and also because they do not exist in OpenCV by default.

# An example of background segmentation

Let's start with the `BackgroundSubtractorKNN` class and a hands-on example to see how background segmentation algorithms are used. You can use the `createBackgroundSubtractorKNN` function to create an object of the `BackgroundSubtractorKNN` type. Here's how:

```
int history = 500;
double dist2Threshold = 400.0;
bool detectShadows = true;
Ptr<BackgroundSubtractorKNN> bgs =
    createBackgroundSubtractorKNN(history,
                                  dist2Threshold,
                                  detectShadows);
```

To understand the parameters used in the `BackgroundSubtractorKNN` class, it is important to first note that this algorithm uses a sampling technique over the history of pixels to create a sampled background image. With that being said, the `history` parameter is used to define the number of previous frames that are used for sampling the background image, and the `dist2Threshold` parameter is the threshold of squared distance between a pixel's current value and its corresponding pixel value in the sampled background image. `detectShadows` is a self-explanatory parameter that is used to determine whether the shadows are going to be detected during background segmentation or not.

Now, we can simply use `bgs` to extract foreground masks from a video and use it to detect movements or an object entering the scene. Here's how:

```
VideoCapture cam(0);
if(!cam.isOpened())
    return -1;

while(true)
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
        break;

    Mat fgMask; // foreground mask
    bgs->apply(frame,
                fgMask);
```

```

    Mat fg; // foreground image
    bitwise_and(frame, frame, fg, fgMask);

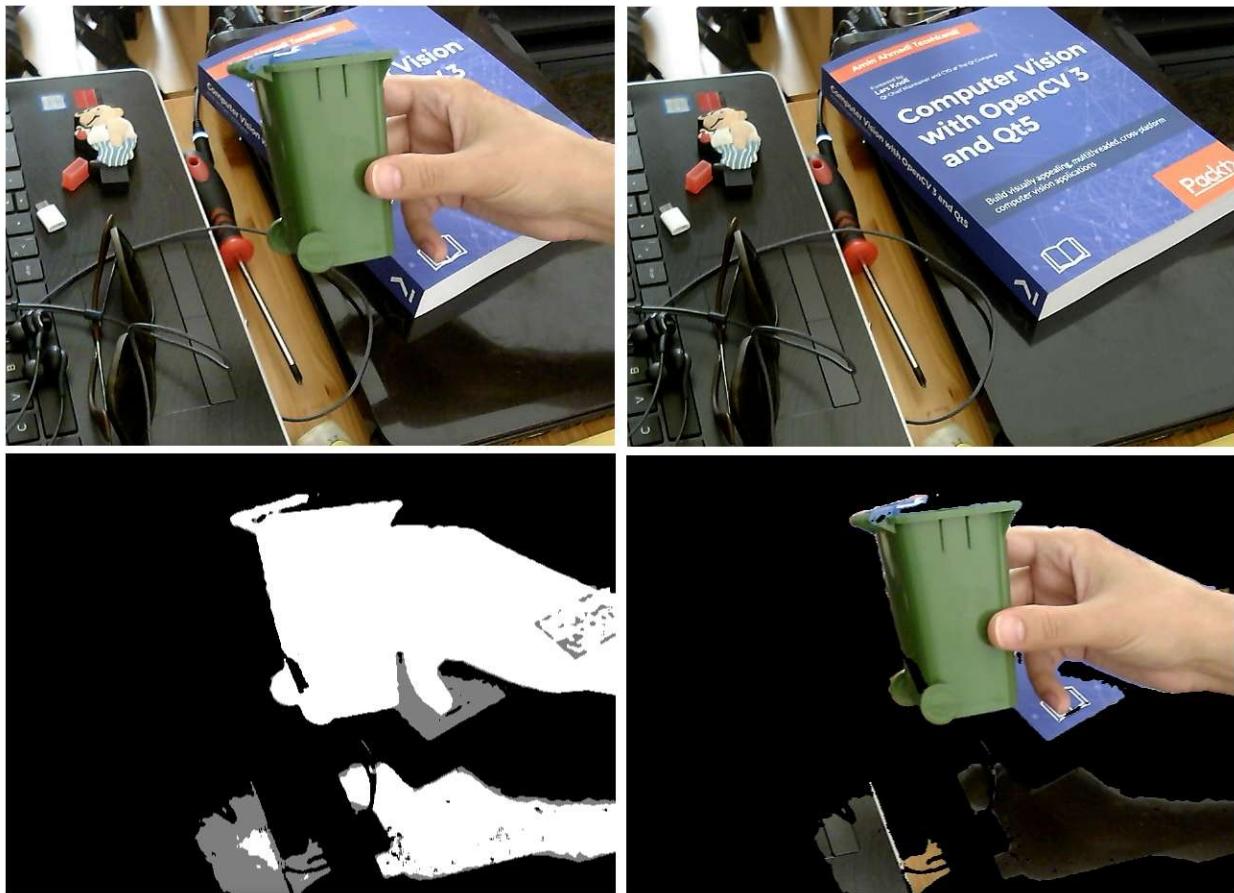
    Mat bg; // background image
    bgs->getBackgroundImage(bg);

    imshow("Input Image", frame);
    imshow("Background Image", bg);
    imshow("Foreground Mask", fgMask);
    imshow("Foreground Image", fg);

    int key = waitKey(10);
    if(key == 27) // escape key
        break;
}
cam.release();

```

Let's quickly review the parts of the previous code that are new and maybe not that obvious. First things first, we use the `apply` function of the `BackgroundSubtractorKNN` class to perform a background/foreground segmentation operation. This function also updates the internal sampled background image for us. After that, we use the `bitwise_and` function with the foreground mask to extract the foreground image's content. To retrieve the sampled background image itself, we simply use the `getBackgroundImage` function. Finally, we display all of the results. Here are some example results that depict a scene (top-left), the extracted background image (top-right), the foreground mask (bottom-left), and the foreground image (bottom-right):



Notice that the shadow of the hand that moved into the scene is also captured by the background segmentation algorithm. In our example, we omitted the `learningRate` parameter when using the `apply` function. This parameter can be used to set the rate at which the learned background model is updated. A value of 0 means the model will not be updated at all, which can be quite useful if you are sure that the background will stay the same for any known period. A value of 1.0 means an extremely quick update of the model. As in the case of our example, we skipped this parameter, which causes it to use -1.0, and it means that the algorithm itself will decide on the learning rate. Another important thing to note is that the result of the `apply` function can yield an extremely noisy mask, which can be smoothed out by using a simple blur function, such as `medianBlur`, as seen here:

```
| medianBlur(fgMask, fgMask, 3);
```

Using the `BackgroundSubtractorMOG2` class is quite similar to `BackgroundSubtractorKNN`. Here's an example:

```
int history = 500;
double varThreshold = 16.0;
bool detectShadows = true;
Ptr<BackgroundSubtractorMOG2> bgs =
    createBackgroundSubtractorMOG2(history,
                                    varThreshold,
                                    detectShadows);
```

Note that the `createBackgroundSubtractorMOG2` function is used quite similarly to what we saw before to create an instance of the `BackgroundSubtractorMOG2` class. The only parameter that differs here is `varThreshold`, which corresponds to the variance threshold used for matching the pixels value and the background model. Using the `apply` and `getBackgroundImage` functions is identical in both background segmentation classes. Try modifying the threshold values in both algorithms to learn more about the visual effects of the parameters.

Background-segmentation algorithms have great potential for video editing software or even detecting and tracking objects in an environment with backgrounds that do not change too much. Try to use them in conjunction with the algorithms that you learned previously in this chapter to build tracking algorithms that make use of multiple algorithms to improve the results.

# Summary

The video analysis module in OpenCV is a collection of extremely powerful algorithms, functions, and classes that we have learned about in this chapter. Starting from the whole idea of video processing and simple calculations based on the content of consecutive video frames, we moved on to learn about the Mean Shift algorithm and how it is used to track objects with known colors and specifications using a back-projection image. We also learned about the more sophisticated version of the Mean Shift algorithm, which is called the Continuously Adaptive Mean Shift, or simply CAM Shift. We learned that this algorithm is also capable of handling objects of different sizes and determining their orientation. Moving on with the tracking algorithms, we learned about the powerful Kalman filter and how it is used for de-noising and correcting the tracking results. We used the Kalman filter to track mouse movements and to correct the tracking results of the Mean Shift and CAM Shift algorithms. Finally, we learned about OpenCV classes that implement background-segmentation algorithms. We wrote a simple program to use background-segmentation algorithms and output the calculated background and foreground images. By now, we are familiar with some of the most popular and widely used computer vision algorithms that allow real-time detection and tracking of objects.

In the next chapter, we'll be learning about many feature extraction algorithms, functions, and classes, and how to use features to detect objects or extract useful information from images based on their key points and descriptors.

# Questions

1. All the examples in this chapter that deal with cameras return when there is a single failed or corrupted frame that leads to the detection of an empty frame. What type of modification is needed to allow a predefined number of retries before stopping the process?
2. How can we call the `meanshift` function to perform the Mean Shift algorithm with 10 iterations and an epsilon value of 0.5?
3. How can we visualize the hue histogram of the tracked object? Assume `camshift` is used for tracking.
4. Set the process-noise covariance in the `KalmanFilter` class so that the filtered and measured values overlap. Assume only the process-noise covariance is set, of all the available matrices for `KalmanFilter` class-behavior control.
  
5. Let's assume that the Y position of the mouse on a window is used to describe the height of a filled rectangle that starts from the top-left corner of the window and has a width that equals the window width. Write a Kalman filter that can be used to correct the height of the rectangle (single value) and remove noise in the mouse movement that will cause a visually smooth resizing of the filled rectangle.
6. Create a `BackgroundSubtractorMOG2` object to extract the foreground image contents while avoiding shadow changes.
7. Write a program to display the *current* (as opposed to sampled) background image using a background-segmentation algorithm.

# Object Detection – Features and Descriptors

In the previous chapter, we learned about processing videos and how to perform the operations and algorithms from all of the previous chapters on frames read from cameras or video files. We learned that each video frame can be treated as an individual image, so we can easily use algorithms, such as filtering, on videos in almost the same way as we did with images. After learning how to process videos using algorithms that work on single individual frames, we moved on to learn about video processing algorithms that require a set of consecutive video frames to perform object detection, tracking, and so on. We learned about how to use the magic of the Kalman filter to improve object-tracking results, and ended the chapter by learning about background and foreground extraction.

The object detection (and tracking) algorithms that we learned about in the previous chapter rely heavily on the color of an object, which has proven not to be too reliable, especially if the object and the environment we are working with are not controlled in terms of lighting. We all know that the brightness and color of an object can easily (and sometimes extremely) change under sunlight and moonlight, or if a light of a different color is near the object, such as a red traffic light. These difficulties are the reason why the detection of objects is more reliable when their physical shape and features are used as a basis for object detection algorithms. Obviously, the shape of an image is independent of its color. A circular object will remain circular during the day or night, so an algorithm that is capable of extracting the shape of such an object would be more reliable to be used for detecting that object.

In this chapter, we're going to learn about computer vision algorithms, functions, and classes that can be used to detect and recognize objects using their features. We'll learn about a number of algorithms that can be used for shape extraction and analysis, and then we'll proceed to learning about key-point detection and descriptor-extraction algorithms. We'll also learn how to match descriptors from two images to detect objects of known shapes in an image. In addition to the topics that we just mentioned, this chapter will also include the required

functions for proper visualization of key points and matching results.

In this chapter, you'll learn about the following:

- Template matching for object detection
- Detecting contours and using them for shape analysis
- Calculating and analyzing contours
- Extracting lines and circles using the Hough transformation
- Detecting, describing, and matching features

# Technical requirements

- An IDE to develop C++ or Python applications
- The OpenCV library

Refer to [chapter 2, \*Getting Started with OpenCV\*](#), for more information about how to set up a personal computer and make it ready for developing computer vision applications using the OpenCV library.

You can use the following URL to download the source code and examples for this chapter:

<https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter07>.

# Template matching for object detection

Before we start with the shape-analysis and feature-analysis algorithms, we are going to learn about an easy-to-use, extremely powerful method of object detection called **template matching**. Strictly speaking, this algorithm does not fall into the category of algorithms that use any knowledge about the shape of an object, but it uses a previously acquired template image of an object that can be used to extract a template-matching result and consequently objects of known look, size, and orientation. You can use the `matchTemplate` function in OpenCV to perform a templating-matching operation. Here's an example that demonstrates the complete usage of the `matchTemplate` function:

```
Mat object = imread("Object.png");
Mat objectGr;
cvtColor(object, objectGr, COLOR_BGR2GRAY);
Mat scene = imread("Scene.png");
Mat sceneGr;
cvtColor(scene, sceneGr, COLOR_BGR2GRAY);

TemplateMatchModes method = TM_CCOEFF_NORMED;

Mat result;
matchTemplate(sceneGr, objectGr, result, method);
```

`method` must be an entry from the `TemplateMatchModes` enum, which can be any of the following values:

- `TM_SQDIFF`
- `TM_SQDIFF_NORMED`
- `TM_CCORR`
- `TM_CCORR_NORMED`
- `TM_CCOEFF`
- `TM_CCOEFF_NORMED`

For detailed information about each template-matching method, you can refer to the OpenCV documentation. For our practical examples, and to learn how the

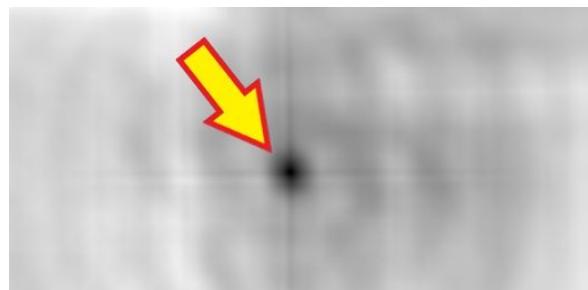
`matchTemplate` function is used in practice, it is important to note that each method will result in a different type of result, and consequently a different interpretation of the result is required, which we'll learn about in this section. In the preceding example, we are trying to detect an object in a scene by using an object image and a scene image. Let's assume the following images are the object (left-hand side) and the scene (right-hand side) that we'll be using:



The very simple idea in template matching is that we are searching for a point in the scene image on the right-hand side that has the highest possibility of containing the image on the left-hand side, or in other words, the template image. The `matchTemplate` function, depending on the method that is used, will provide a probability distribution. Let's visualize the result of the `matchTemplate` function to better understand this concept. Another important thing to note is that we can only properly visualize the result of the `matchTemplate` function if we use any of the methods ending with `_NORMED`, which means they contain a normalized result, otherwise we have to use the `normalize` method to create a result that contains values in the displayable range of the OpenCV `imshow` function. Here is how it can be done:

```
|normalize(result, result, 0.0, 1.0, NORM_MINMAX, -1);
```

This function call will translate all the values in `result` to the range of `0.0` and `1.0`, which can then be properly displayed. Here is how the resulting image will look if it is displayed using the `imshow` function:



As mentioned previously, the result of the `matchTemplate` function and how it should be interpreted depends completely on the template matching method that is used. In the case that we use the `TM_SQDIFF` or `TM_SQDIFF_NORMED` methods for template matching, we need to look for the global minimum point in the result (it is shown using an arrow in the preceding image), which has the highest possibility of containing the template image. Here's how we can find the global minimum point (along with global maximum, and so on) in the template matching result:

```
double minVal, maxVal;  
Point minLoc, maxLoc;  
minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc);
```

Since the template-matching algorithm works only with objects of a fixed size and orientation, we can assume that a rectangle that has an upper-left point that is equal to the `minLoc` point and has a size that equals the template image is the best possible bounding rectangle for our object. We can draw the result on the scene image, for better comparison, using the following sample code:

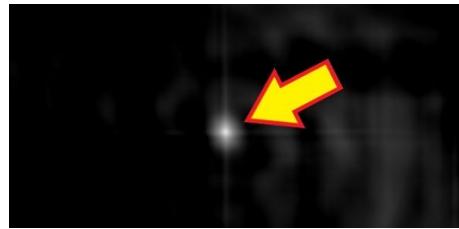
```
Rect rect(minLoc.x,  
         minLoc.y,  
         object.cols,  
         object.rows);  
  
Scalar color(0, 0, 255);  
int thickness = 2;  
rectangle(scene,  
          rect,  
          color,  
          thickness);
```

The following image depicts the result of the object detection operation that was performed using the `matchTemplate` function:



If we use `TM_CCORR`, `TM_CCOEFF`, or their normalized versions, we must use the global maximum point as the point with the highest possibility of containing our

template image. The following image depicts the result of the `TM_CCOEFF_NORMED` method used with the `matchTemplate` function:



As you can see, the brightest point in the resultant image corresponds to the upper-left point of the template image in the scene image.

Before ending our template matching lesson, let's also note that the width and height of the template matching resultant image is smaller than the scene image. This is because the template matching resultant image can only contain the upper-left point of the template image, so the template image width and height are subtracted from the scene image's width and height to determine the resultant image's width and height in the template-matching algorithm.

# Detecting corners and edges

It is not always possible to just compare images pixel-wise and decide whether an object is present in an image or not, or whether an object has the expected shape or not, and many more similar scenarios that we can't even begin to list here. That is why the smarter way of interpreting the contents of an image is to look for meaningful features in it, and then base our interpretation on the properties of those features. In computer vision, a feature is synonymous with a keypoint, so don't be surprised if we use them interchangeably in this book. In fact, the word keypoint is better suited to describe the concept, since the most commonly used features in an image are usually *key points* in that image where there is a sudden change in color intensity, which can happen in corners and edges of shapes and objects in an image.

In this section, we'll learn about some of the most important and widely used keypoint-detection algorithms, namely the corner- and edge-detection algorithms that are the basis of almost all of the feature-based object detection algorithms we'll be learning about in this chapter.

# Learning the Harris corner-detection algorithm

One of the most well-known corner- and edge-detection algorithms is the Harris corner-detection algorithm, which is implemented in the `cornerHarris` function in OpenCV. Here is how this function is used:

```
Mat image = imread("Test.png");
cvtColor(image, image, COLOR_BGR2GRAY);

Mat result;
int blockSize = 2;
int ksize = 3;
double k = 1.0;
cornerHarris(image,
             result,
             blockSize,
             ksize,
             k);
```

`blockSize` determines the width and height of the square block over which the Harris corner-detection algorithm will calculate a 2 x 2 gradient-covariance matrix. `ksize` is the kernel size of the Sobel operator internally used by the Harris algorithm. The preceding example demonstrates one of the most commonly used sets of Harris algorithm parameters, but for more detailed information about the Harris corner-detection algorithm and its internals mathematics, you can refer to the OpenCV documentation. It's important to note that the `result` object from the preceding example code is not displayable unless it is normalized using the following example code:

```
|normalize(result, result, 0.0, 1.0, NORM_MINMAX, -1);
```

Here is the result of the Harris corner-detection algorithm from the preceding example, when normalized and displayed using the OpenCV `imshow` function:



The OpenCV library includes another famous corner-detection algorithm, called **Good Features to Track (GFTT)**. You can use the `goodFeaturesToTrack` function in OpenCV to use the GFTT algorithm to detect corners, as seen in the following example:

```
Mat image = imread("Test.png");
Mat imgGray;
cvtColor(image, imgGray, COLOR_BGR2GRAY);

vector<Point2f> corners;
int maxCorners = 500;
double qualityLevel = 0.01;
double minDistance = 10;
Mat mask;
int blockSize = 3;
int gradientSize = 3;
bool useHarrisDetector = false;
double k = 0.04;
goodFeaturesToTrack(imgGray,
                    corners,
                    maxCorners,
                    qualityLevel,
                    minDistance,
                    mask,
                    blockSize,
                    gradientSize,
                    useHarrisDetector,
                    k);
```

As you can see, this function requires a single-channel image, so, before doing anything else, we have converted our BGR image to grayscale. Also, this function uses the `maxCorners` value to limit the number of detected corners based on how strong they are as candidates, and setting `maxCorners` to a negative value or to zero means all detected corners should be returned, which is not a good idea if you are looking for the best corners in an image, so make sure you set a reasonable value for this based on the environment in which you'll be using it. `qualityLevel` is the internal threshold value for accepting detected corners. `minDistance` is the minimum allowed distance between returned corners. This is

another parameter that is completely dependent on the environment this algorithm will be used in. You have already seen the remaining parameters in the previous algorithms from this chapter and the preceding one. It's important to note that this function also incorporates the Harris corner-detection algorithm, so, by setting `useHarrisDetector` to `true`, the resultant features will be calculated using the Harris corner-detection algorithm.

You might have already noticed that the `goodFeaturesToTrack` function returns a set of `Point` objects (`Point2f` to be precise) instead of a `Mat` object. The returned `corners` vector simply contains the best possible corners detected in the image using the GFTT algorithm, so we can use the `drawMarker` function to visualize the results properly, as seen in the following example:

```
Scalar color(0, 0, 255);
MarkerTypes markerType = MARKER_TILTED_CROSS;
int markerSize = 8;
int thickness = 2;
for(int i=0; i<corners.size(); i++)
{
    drawMarker(image,
               corners[i],
               color,
               markerType,
               markerSize,
               thickness);
}
```

Here is the result of the preceding example and detecting corners using the `goodFeaturesToTrack` function:



You can also use the `GFTTDetector` class to detect corners in a similar way as you did with the `goodFeaturesToTrack` function. The difference here is that the returned type is a vector of `KeyPoint` objects. Many OpenCV functions and classes use the

`KeyPoint` class to return various properties of detected keypoints, instead of just a `Point` object that corresponds to the location of the keypoint. Let's see what this means with the following:

```
| Ptr<GFTTDetector> detector =  
|   GFTTDetector::create(maxCorners,  
|                         qualityLevel,  
|                         minDistance,  
|                         blockSize,  
|                         gradientSize,  
|                         useHarrisDetector,  
|                         k);  
  
| vector<KeyPoint> keypoints;  
| detector->detect(image, keypoints);
```

The parameters passed to the `GFTTDetector::create` function are no different from the parameters we used with the `goodFeaturesToTrack` function. You can also omit all of the given parameters and simply write the following to use the default and optimal values for all parameters:

```
| Ptr<GFTTDetector> detector = GFTTDetector::create();
```

But let's get back to the `KeyPoint` class and the result of the `detect` function from the previous example. Recall that we used a loop to go through all of the detected points and draw them on the image. There is no need for this if we use the `GFTTDetector` class, since we can use an existing OpenCV function called `drawKeypoints` to properly visualize all of the detected keypoints. Here's how this function is used:

```
| Mat outImg;  
| drawKeypoints(image,  
|                 keypoints,  
|                 outImg);
```

The `drawKeypoints` function goes through all `KeyPoint` objects in the `keypoints` vector and draws them using random colors on `image` and saves the result in the `outImg` object, which we can then display by calling the `imshow` function. The following image is the result of the `drawKeypoints` function when it is called using the preceding example code:



The `drawKeypoints` function can be provided with an additional (optional) color parameter in case we want to use a specific color instead of random colors. In addition, we can also provide a flag parameter that can be used to further enhance the visualized result of the detected keypoints. For instance, if the flag is set to `DRAW_RICH_KEYPOINTS`, the `drawKeypoints` function will also use the size and orientation values in each detected keypoint to visualize more properties of keypoints.

*Each `keyPoint` object may contain the following properties, depending on the algorithm used for calculating it:*

- `pt`: A `Point2f` object containing the coordinates of the keypoint.
- `size`: The diameter of the meaningful keypoint neighborhood.
- `angle`: The orientation of the keypoint in degrees, or -1 if not applicable.
- `response`: The strength of the keypoint determined by the algorithm.
- `octave`: The octave or pyramid layer from which the keypoint was extracted. Using octaves allows us to deal with keypoints detected from the same image but in different scales. Algorithms that set this value usually require an input octave parameter, which is used to define the number of octaves (or scales) of an image that is used to extract keypoints.
- `class_id`: This integer parameter can be used to group keypoints, for instance, when keypoints belong to a single object, they can have the same optional `class_id` value.

In addition to Harris and GFTT algorithms, you can also use the FAST corner-detection algorithm using the `FastFeatureDetector` class, and the AGAST corner-detection algorithm (**Adaptive and Generic Corner Detection Based on the Accelerated Segment Test**) using the `AgastFeatureDetector` class, quite similar to how we used the `GFTTDetector` class. It's important to note that all of these classes belong to the `features2d` module in the OpenCV library and they are all subclasses of the `Feature2D` class, therefore all of them contain a static `create` function that

creates an instance of their corresponding classes and a `detect` function that can be used to extract the keypoints from an image.

Here is an example code demonstrating the usage of `FastFeatureDetector` using all of its default parameters:

```
int threshold = 10;
bool nonmaxSuppr = true;
int type = FastFeatureDetector::TYPE_9_16;
Ptr<FastFeatureDetector> fast =
    FastFeatureDetector::create(threshold,
                                nonmaxSuppr,
                                type);

vector<KeyPoint> keypoints;
fast->detect(image, keypoints);
```

Try increasing the `threshold` value if too many corners are detected. Also, make sure to check out the OpenCV documentation for more information about the `type` parameter used in the `FastFeatureDetector` class. As mentioned previously, you can simply omit all of the parameters in the preceding example code to use the default values for all parameters.

Using the `AgastFeatureDetector` class is extremely similar to using `FastFeatureDetector`. Here is an example:

```
int threshold = 10;
bool nonmaxSuppr = true;
int type = AgastFeatureDetector::OAST_9_16;
Ptr<AgastFeatureDetector> agast =
    AgastFeatureDetector::create(threshold,
                                nonmaxSuppr,
                                type);

vector<KeyPoint> keypoints;
agast->detect(image, keypoints);
```

Before moving on to edge-detection algorithms, it's worth noting that OpenCV also contains the `AGAST` and `FAST` functions, which can be employed to directly use their corresponding algorithms and avoid dealing with creating an instance to use them; however, using the class implementation of these algorithms has the huge advantage of switching between algorithms using polymorphism. Here's a simple example that demonstrates how we can use polymorphism to benefit

from the class implementations of corner-detection algorithms:

```
Ptr<Feature2D> detector;
switch (algorithm)
{
    case 1:
        detector = GFTTDetector::create();
        break;

    case 2:
        detector = FastFeatureDetector::create();
        break;

    case 3:
        detector = AgastFeatureDetector::create();
        break;

    default:
        cout << "Wrong algorithm!" << endl;
        return 0;
}

vector<KeyPoint> keypoints;
detector->detect(image, keypoints);
```

`algorithm`, in the preceding example, is an integer value that can be set at run-time and will change the type of the corner-detection algorithm assigned to the `detector` object, which has the `Feature2D` type, or in other words, the base class of all corner-detection algorithms.

# Edge-detection algorithms

Now that we've gone through corner-detection algorithms, let's take a look at edge-detection algorithms, which are crucial when it comes to shape analysis in computer vision. OpenCV contains a number of algorithms that can be used to extract edges from images. The first edge-detection algorithm that we're going to learn about is called the **line-segment-detection algorithm**, and it can be performed by using the `LineSegmentDetector` class, as seen in the following example:

```
Mat image = imread("Test.png");
Mat imgGray;
cvtColor(image, imgGray, COLOR_BGR2GRAY);

Ptr<LineSegmentDetector> detector = createLineSegmentDetector();

vector<Vec4f> lines;
detector->detect(imgGray,
                   lines);
```

As you can see, the `LineSegmentDetector` class requires a single-channel image as the input and produces a `vector` of lines. Each line in the result is `vec4f`, or four floating-point values that represent  $x_1$ ,  $y_1$ ,  $x_2$ , and  $y_2$  values, or in other words, the coordinates of the two points that form each line. You can use the `drawSegments` function to visualize the result of the `detect` function of the `LineSegmentDetector` class, as seen in the following example:

```
Mat result(image.size(),
           CV_8UC3,
           Scalar(0, 0, 0));

detector->drawSegments(result,
                        lines);
```

To have more control over how the resultant lines are visualized, you might want to manually draw the lines vector, as seen in the following example:

```
Mat result(image.size(),
           CV_8UC3,
           Scalar(0, 0, 0));

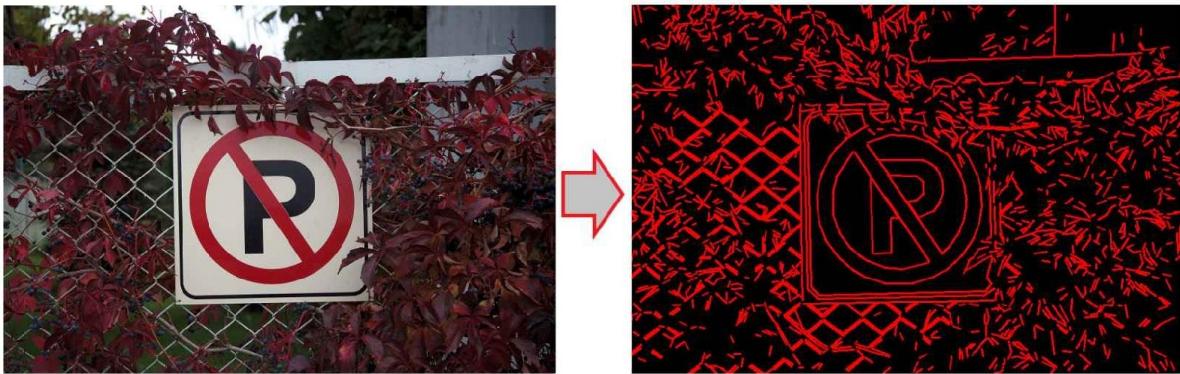
Scalar color(0, 0, 255);
int thickness = 2;
for(int i=0; i<lines.size(); i++)
{
    line(result,
          Point(lines.at(i)[0],
```

```

        lines.at(i)[1]),
    Point(lines.at(i)[2],
        lines.at(i)[3]),
    color,
    thickness);
}

```

The following image demonstrates the result of the line-segment-detection algorithm that was used in the preceding example codes:



For more details about how to customize the behavior of the `LineSegmentDetector` class, make sure to view the documentation of `createLineSegmentDetector` and its parameters. In our example, we simply omitted all of its input parameters and used the `LineSegmentDetector` class with the default values set for its parameters.

Another function of the `LineSegmentDetector` class is comparing two sets of lines to find the number of non-overlapping pixels, and at the same time drawing the result of the comparison on an output image for visual comparison. Here's an example:

```

vector<Vec4f> lines1, lines2;
detector->detect(imgGray1,
    lines1);

detector->detect(imgGray2,
    lines2);

Mat resultImg(imageSize, CV_8UC3, Scalar::all(0));
int result = detector->compareSegments(imageSize,
    lines1,
    lines2,
    resultImg);

```

In the preceding code, `imageSize` is a `size` object that contains the size of the input image where the lines were extracted from. The result is an integer value that contains the result of the comparison function, or the `compareSegments` function,

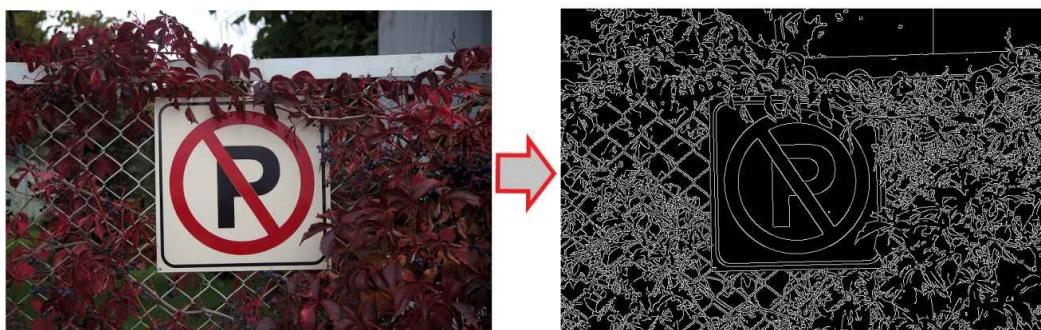
which will be zero in the case of the complete overlapping of pixels.

The next edge-detection algorithm is probably one of the most widely used and cited edge-detection algorithms in computer vision, called the **Canny algorithm**, which has a function of the same name in OpenCV. The biggest advantage of the `canny` function is the simplicity of its input parameters. Let's first see an example of how it's used, and then walk through its details:

```
Mat image = imread("Test.png");
double threshold1 = 100.0;
double threshold2 = 200.0;
int apertureSize = 3;
bool L2gradient = false;
Mat edges;
Canny(image,
       edges,
       threshold1,
       threshold2,
       apertureSize,
       L2gradient);
```

The threshold values (`threshold1` and `threshold2`) are the lower and higher bound values for thresholding the input image. `apertureSize` is the internal Sobel operator aperture size, and `L2gradient` is used to enable or disable the more accurate L2 norm when calculating the gradient image. The result of the `canny` function is a grayscale image that contains white pixels where edges are detected and black pixels for the rest of the pixels. This makes the result of the `canny` function a suitable mask wherever such a mask is needed, or, as you'll see later on, a suitable set of points to extract contours from.

The following image depicts the result of the `canny` function used in the previous example:



As we mentioned before, the result of the `canny` function is suitable to use as the

input to algorithms that require a binary image, or in other words, a grayscale image containing only absolute black and absolute white pixel values. The next algorithm that we'll learn about is one where the result of a previous `canny` function must be used as the input, and it is called the **Hough transformation**. The Hough transformation can be used to extract lines from an image, and it is implemented in a function called `HoughLines` in the OpenCV library.

Here is a complete example that demonstrates how the `HoughLines` function is used in practice:

1. Call the `Canny` function to detect edges in the input image, as seen here:

```
Mat image = imread("Test.png");

double threshold1 = 100.0;
double threshold2 = 200.0;
int apertureSize = 3;
bool L2gradient = false;
Mat edges;
Canny(image,
       edges,
       threshold1,
       threshold2,
       apertureSize,
       L2gradient);
```

2. Call the `HoughLines` function to extract lines from the detected edges:

```
vector<Vec2f> lines;
double rho = 1.0; // 1 pixel, r resolution
double theta = CV_PI / 180.0; // 1 degree, theta resolution
int threshold = 100; // minimum number of intersections to "detect" a line
HoughLines(edges,
           lines,
           rho,
           theta,
           threshold);
```

3. Use the following code to extract points in the standard coordinate system, and draw them over the input image:

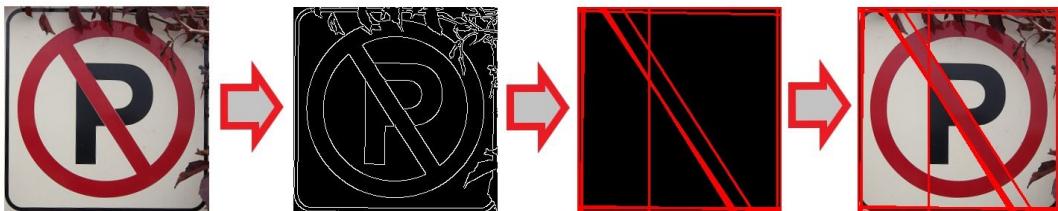
```
Scalar color(0,0,255);
int thickness = 2;
for(int i=0; i<lines.size(); i++)
{
    float rho = lines.at(i)[0];
    float theta = lines.at(i)[1];
    Point pt1, pt2;
    double a = cos(theta);
    double b = sin(theta);
    double x0 = a*rho;
    double y0 = b*rho;
    pt1.x = int(x0 + 1000*(-b));
```

```

        pt1.y = int(y0 + 1000*(a));
        pt2.x = int(x0 - 1000*(-b));
        pt2.y = int(y0 - 1000*(a));
        line( image, pt1, pt2, color, thickness);
    }
}

```

The following images depict the result of the preceding example from left to right, starting with the original image, edges detected using the `Canny` function, lines detected using the `HoughLines` function, and finally the output image:



To avoid having to deal with the coordinate-system change, you can use the `HoughLinesP` function to directly extract the points forming each detected line. Here's an example:

```

vector<Vec4f> lines;
double rho = 1.0; // 1 pixel, r resolution
double theta = CV_PI / 180.0; // 1 degree, theta resolution
int threshold = 100; // minimum number of intersections to "detect" a line
HoughLinesP(edges,
            lines,
            rho,
            theta,
            threshold);

Scalar color(0,0,255);
int thickness = 2;
for(int i=0; i<lines.size(); i++)
{
    line(image,
          Point(lines.at(i)[0],
                lines.at(i)[1]),
          Point(lines.at(i)[2],
                lines.at(i)[3]),
          color,
          thickness);
}

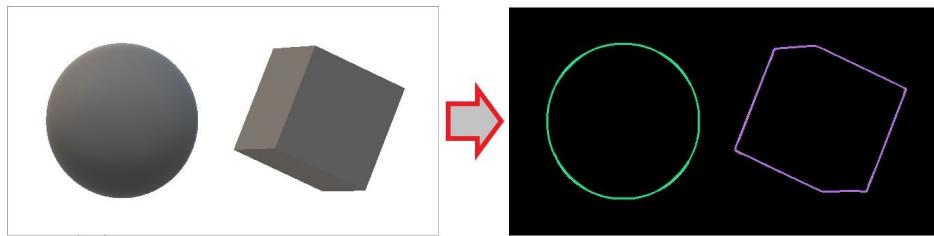
```

The Hough transformation is extremely powerful, and OpenCV contains more variations of the Hough transformation algorithm that we'll leave for you to discover using the OpenCV documentation and online resources. Note that using the Canny algorithm is a prerequisite of the Hough transformation, and, as you'll see in the next section, a prerequisite of many algorithms that deal with the shape of objects in an image.

# Contour calculation and analysis

Contours of shapes and objects in an image are an important visual property that can be used to describe and analyze them. Computer vision is no exception, so there are quite a few algorithms in computer vision that can be used to calculate the contours of objects in an image or calculate their area and so on.

The following image depicts two contours that are extracted from two 3D objects:



OpenCV includes a function, called `findContours`, that can be used to extract contours from an image. This function must be provided with a proper binary image that contains the best candidate pixels for contours; for instance, the result of the `Canny` function is a good choice. The following example demonstrates the steps required to calculate the contours of an image:

1. Find the edges using the `Canny` function, as seen here:

```
Mat image = imread("Test.png");
Mat imgGray;
cvtColor(image, imgGray, COLOR_BGR2GRAY);

double threshold1 = 100.0;
double threshold2 = 200.0;
int apertureSize = 3;
bool L2gradient = false;
Mat edges;
Canny(image,
       edges,
       threshold1,
       threshold2,
       apertureSize,
       L2gradient);
```

2. Use the `findContours` function to calculate the contours using the detected edges. It's worth noting that each contour is a `vector` of `Point` objects, making

all contours a vector of vector of Point objects, as seen here:

```
vector<vector<Point>> contours;
int mode = CV_RETR_TREE;
int method = CV_CHAIN_APPROX_TC89_KCOS;
findContours(edges,
             contours,
             mode,
             method);
```

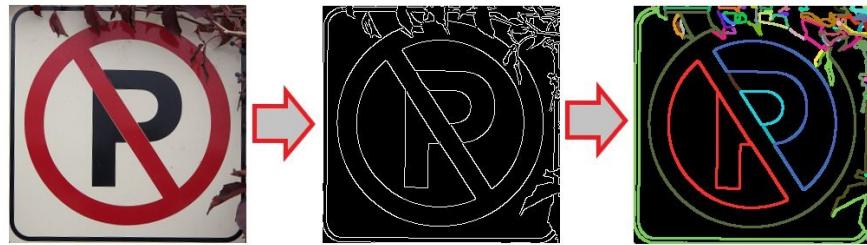
In the preceding example, the contour-retrieval mode is set to `CV_RETR_TREE` and the contour-approximation method is set to `CV_CHAIN_APPROX_TC89_KCOS`. Make sure to go through the list of possible modes and methods by yourself, and compare the results to find the best parameters for your use case.

3. A common method of visualizing detected contours is by using the `RNG` class, or the Random Number Generator class, to generate random colors for each detected contour. The following example demonstrates how you can use the `RNG` class in combination with the `drawContours` function to properly visualize the result of the `findContours` function:

```
RNG rng(12345); // any random number

Mat result(edges.size(), CV_8UC3, Scalar(0));
int thickness = 2;
for( int i = 0; i < contours.size(); i++ )
{
    Scalar color = Scalar(rng.uniform(0, 255),
                          rng.uniform(0, 255),
                          rng.uniform(0, 255) );
    drawContours(result,
                 contours,
                 i,
                 color,
                 thickness);
}
```

The following image demonstrates the result of the `Canny` and `findContours` functions:



Note the different colors in the image on the right-hand side, which correspond to one complete contour detected by using the `findContours` function.

After calculating the contours, we can use contour-analysis functions to further modify them or analyze the shape of the object in an image. Let's start with the `contourArea` function, which can be used to calculate the area of a given contour. Here is how this function is used:

```
| double area = contourArea(contour);
```

You can use `area` as a threshold for ignoring the detected contours that do not pass certain criteria. For example, in the preceding example code where we used the `drawContours` function, we could get rid of contours with smaller areas than some predefined threshold value. Here's an example:

```
| for( int i = 0; i< contours.size(); i++ )
{
    if(contourArea(contours[i]) > thresholdArea)
    {
        drawContours(result,
                    contours,
                    i,
                    color,
                    thickness);
    }
}
```

Setting the second parameter of the `contourArea` function (which is a Boolean parameter) to `true` would result in the orientation being considered in the contour area, which means you can get positive or negative values of the area depending on the orientation of the contour.

Another contour-analysis function that can be quite handy is the `pointPolygonTest` function. As you can guess from its name, this function is used to perform a point-in-polygon test, or in other words, a point-in-contour test. Here is how this

function is used:

```
| Point pt(x, y);  
| double result = pointPolygonTest(contours[i], Point(x,y), true);
```

If the result is zero, it means the test point is right on the edge of the contour. A negative result would mean the test point is outside, and a positive result would mean the test point is inside the contour. The value itself is the distance between the test point and the nearest contour edge.

To be able to check whether a contour is convex or not, you can use the `isContourConvex` function, as seen in the following example:

```
| bool isIt = isContourConvex(contour);
```

Being able to compare two contours with each other is probably the most essential algorithm that you'll need when dealing with contour and shape analysis. You can use the `matchShapes` function in OpenCV to compare and try to match two contours. Here is how this function is used:

```
| ShapeMatchModes method = CONTOURS_MATCH_I1;  
| double result = matchShapes(cntr1, cntr2, method, 0);
```

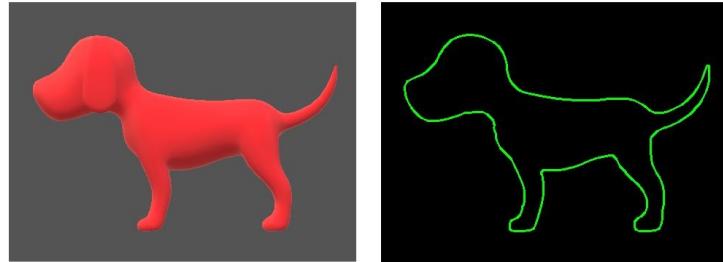
`method` can take any of the following values, while the last parameter must always be set to zero, unless specified by the method used:

- `CONTOURS_MATCH_I1`
- `CONTOURS_MATCH_I2`
- `CONTOURS_MATCH_I3`

For details about the mathematical difference between the preceding list of contour-matching methods, you can refer to the OpenCV documentation.

Being able to find the boundaries of a contour is the same as being able to

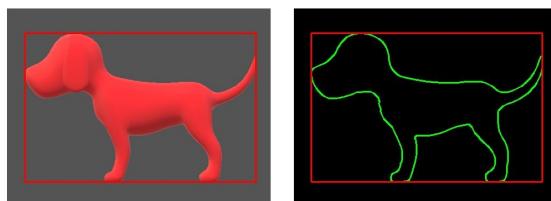
correctly localize it, for instance to find a region that can be used for tracking or performing any other computer vision algorithm. Let's assume we have the following image and its single contour detected using the `findContours` function:



Having this contour, we can perform any of the contour- and shape-analysis algorithms that we've learned about. In addition, we can use a number of OpenCV functions to localize the extracted contour. Let's start with the `boundingRect` function, which is used to find the minimal upright rectangle (`Rect` object) that contains a given point set or contour. Here's how this function is used:

```
| Rect br = boundingRect(contour);
```

The following is the result of drawing the upright rectangle acquired by using `boundingRect` in the preceding sample code:



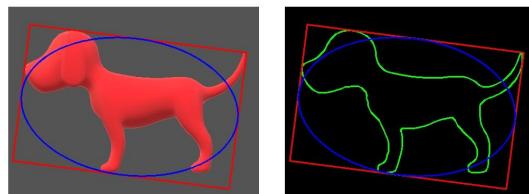
Similarly, you can use the `minAreaRect` function to find the minimal rotated rectangle that contains a given set of points or a contour. Here's an example:

```
| RotatedRect br = minAreaRect(contour);
```

You can use the following code to visualize the resultant rotated rectangle:

```
Point2f points[4];
br.points(points);
for (int i=0; i<4; i++)
    line(image,
        points[i],
        points[(i+1)%4],
        Scalar(0,0,255),
        2);
```

You can draw an ellipse instead, using the `ellipse` function, or you can do both, which would result in something similar to the following:



In addition to algorithms for finding the minimal upright and rotated bounding rectangles of contours, you can also use the `minEnclosingCircle` and `minEnclosingTriangle` functions to find the minimal bounding circle and rectangle of a given set of points or a contour. Here's an example of how these functions can be used:

```
// to detect the minimal bounding circle
Point2f center;
float radius;
minEnclosingCircle(contour, center, radius);

// to detect the minimal bounding triangle
vector<Point2f> triangle;
minEnclosingTriangle(contour, triangle);
```

There is no end to the list of possible use cases of contours, but we will name just a few of them before moving on to the next section. You can try using contour-detection and shape-analysis algorithms in conjunction with thresholding algorithms or back-projection images, for instance, to make sure your tracking algorithm uses the shape information in addition to the color and intensity values of pixels. You can also use contours to count and analyze shapes of objects on a production line, where the background and the visual environment is more controlled.

The final section of this chapter will teach you how to use feature detection, descriptor extraction, and descriptor-matching algorithms to detect known objects, but with rotation, scale, and even perspective invariance.

# Detecting, describing, and matching features

As we learned earlier in this chapter, features or keypoints can be extracted from images using various feature-extraction (detection) algorithms, most of which rely on detecting points with a significant change in intensity, such as corners. Detecting the right keypoints is the same as being able to correctly determine which parts of an image are helpful in identifying it. But just a keypoint, or in other words the location of a significant point in an image, by itself is not useful. One might argue that the collection of keypoint locations in an image is enough, but even then, another object with a totally different look can have keypoints in the exact same locations in an image, say, by chance.

This is where feature descriptors, or simply descriptors, come into play. A descriptor, as you can guess from the name, is an algorithm-dependent method of describing a feature, for instance, by using its neighboring pixel values, gradients, and so on. There are many different descriptor-extraction algorithms, each one with its own advantages and disadvantages, and going through all of them would not be a fruitful endeavor, especially for a hands-on book, but it's worth noting that most of them simply take a list of keypoints and produce a vector of descriptors. After a set of descriptors are extracted from sets of keypoints, we can use descriptor-matching algorithms to find the matching features from two different images, for instance, an image of an object and a scene where that object exists.

OpenCV contains a large number of feature detectors, descriptor extractors, and descriptor matchers. All feature-detector and descriptor-extractor algorithms in OpenCV are subclasses of the `Feature2D` class, and they are located in either the `features2d` module, which is included by default in OpenCV packages, or the `xfeatures2d` (extra module) module. You should use these algorithms with care and always refer to the OpenCV documentation, since some of them are actually patented and require permission from their owners to be used in commercial

projects. The following is a list of some of the main feature-detector and descriptor-extractor algorithms that are included in OpenCV by default:

- **BRISK (Binary Robust Invariant Scalable Keypoints)**
- **KAZE**
- **AKAZE (Accelerated KAZE)**
- **ORB, or Oriented BRIEF (Binary Robust Independent Elementary Features)**

All of these algorithms are implemented in classes of exactly the same title in OpenCV, and to repeat once more, they are all subclasses of the `Feature2D` class. They are extremely simple to use, especially when no parameters are modified. In all of them, you can simply use the static `create` method to create an instance of them, call the `detect` method to detect the keypoints, and finally call `compute` to extract descriptors of the detected keypoints.

As for descriptor-matcher algorithms, OpenCV contains the following matching algorithms by default:

- FLANNBASED
- BRUTEFORCE
- BRUTEFORCE\_L1
- BRUTEFORCE\_HAMMING
- BRUTEFORCE\_HAMMINGLUT
- BRUTEFORCE\_SL2

You can use the `DescriptorMatcher` class, or its subclasses, namely `BFMatcher` and `FlannBasedMatcher`, to perform various matching algorithms. You simply need to use the static `create` method of these classes to create an instance of them, and then use the `match` method to match two sets of descriptors.

Let's walk through all of what we've discussed in this section with a complete example, since breaking apart the feature detection, descriptor extraction, and matching is impossible, and they are all parts of a chain of processes that lead to the detection of an object in a scene using its features:

1. Read the image of the object, and the scene that will be searched for the object, using the following code:

```
Mat object = imread("object.png");
Mat scene = imread("Scene.png");
```

In the following picture, let's assume the image on the left is the object we are looking for, and the image on the right is the scene that contains the object:



2. Extract the keypoints from both of these images, which are now stored in `object` and `scene`. We can use any of the aforementioned algorithms for feature detection, but let's assume we're using KAZE for our example, as seen here:

```
Ptr<KAZE> detector = KAZE::create();
vector<KeyPoint> objKPs, scnKPs;
detector->detect(object, objKPs);
detector->detect(scene, scnKPs);
```

3. We have the keypoints of both the object image and the scene image. We can go ahead and view them using the `drawKeypoints` function, as we learned previously in this chapter. Try that on your own, and then use the same `KAZE` class to extract descriptors from the keypoints. Here's how it's done:

```
Mat objDesc, scnDesc;
detector->compute(object, objKPs, objDesc);
detector->compute(scene, scnKPs, scnDesc);
```

4. `objDesc` and `scnDesc` correspond to the descriptors of the keypoints extracted from the object and scene images. As mentioned previously, descriptors are algorithm-dependent, and interpreting the exact values in them requires in-detail knowledge about the specific algorithm that was used to extract them. Make sure to refer to the OpenCV documentation to gain more knowledge about them, however, in this step, we're going to simply use a brute-force

matcher algorithm to match the descriptors extracted from both images. Here's how:

```
Ptr<BFMatcher> matcher = BFMatcher::create();
vector<DMatch> matches;
matcher->match(objDesc, scnDesc, matches);
```

The `BFMatcher` class, which is a subclass of the `DescriptorMatcher` class, implements the brute-force matching algorithm. The result of descriptor matching is stored in a `vector` of `DMatch` objects. Each `DMatch` object contains all the necessary information for matched features, from the object descriptors to the scene descriptors.

5. You can now try to visualize the result of matching by using the `drawMatches` function, as seen here:

```
Mat result;
drawMatches(object,
            objKPs,
            scene,
            scnKPs,
            matches,
            result,
            Scalar(0, 255, 0), // green for matched
            Scalar::all(-1), // unmatched color (not used)
            vector<char>(), // empty mask
            DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
```

As you can see, some of the matched features are obviously incorrect, some at the top of the scene image and a few at the bottom:



6. The bad matches can be filtered out by using a threshold on the `distance` value of the `DMatch` objects. The threshold value depends on the algorithm and the type of image content, but in our example case, and with the KAZE

algorithm, a value of `0.1` seems to be enough for us. Here's how the thresholding is done to get good matches out of all the matches:

```
vector<DMatch> goodMatches;
double thresh = 0.1;
for(int i=0; i<objDesc.rows; i++)
{
    if(matches[i].distance < thresh)
        goodMatches.push_back(matches[i]);
}

if(goodMatches.size() > 0)
{
    cout << "Found " << goodMatches.size() << " good matches.";
}
else
{
    cout << "Didn't find a single good match. Quitting!";
    return -1;
}
```

The following image depicts the result of the `drawMatches` function on the `goodMatches` vector:



7. Obviously, the result of the filtered matches is much better now. We can use the `findHomography` function to find the transformation between good matched keypoints from the object image to the scene image. Here's how:

```
vector<Point2f> goodP1, goodP2;
for(int i=0; i<goodMatches.size(); i++)
{
    goodP1.push_back(objKPs[goodMatches[i].queryIdx].pt);
    goodP2.push_back(scnKPs[goodMatches[i].trainIdx].pt);
}
Mat homoChange = findHomography(goodP1, goodP2);
```

8. As we've already seen in the preceding chapters, the result of the

`findHomography` function can be used to transform a set of points. We can abuse this fact to create four points using the four corners of the object image, and then transform those points using the `perspectiveTransform` function to get the location of those points in the scene image. Here is an example:

```
vector<Point2f> corners1(4), corners2(4);
corners1[0] = Point2f(0,0);
corners1[1] = Point2f(object.cols-1, 0);
corners1[2] = Point2f(object.cols-1, object.rows-1);
corners1[3] = Point2f(0, object.rows-1);
perspectiveTransform(corners1, corners2, homoChange);
```

9. The transformed points can be used to draw four lines that localize the detected object in the scene image, as seen here:

```
line(result, corners2[0], corners2[1], Scalar::all(255), 2);
line(result, corners2[1], corners2[2], Scalar::all(255), 2);
line(result, corners2[2], corners2[3], Scalar::all(255), 2);
line(result, corners2[3], corners2[0], Scalar::all(255), 2);
```

It's important to also change the `x` values of the resultant points to account for the width of the object image, if you are going to draw the four lines that localize the object, over the `drawMatches` image result. Here's an example:

```
for(int i=0; i<4; i++)
    corners2[i].x += object.cols;
```

The following image depicts the final result of our detection operation:



Make sure to try the rest of the feature-detection, descriptor-extraction, and matching algorithms by yourself and compare their results. Also, try to measure the time of the calculations for each one. For instance, you might notice that AKAZE is much faster than KAZE, or that BRISK is better suited to some images, while KAZE or ORB is better with others. As mentioned before, the feature-based methods of object detection are much more reliable with scale, rotation, and even perspective change. Try different views of the same objects to figure out the best parameters and algorithms for your own project and use case. For instance, here's another example that demonstrates the rotation and scale invariance of the AKAZE algorithm and brute-force matching:



Note that the source code used for producing the preceding output is created by using exactly the same set of instructions we went through in this section.

# Summary

We started this chapter by learning about a template-matching algorithm and an object detection algorithm that, despite its popularity, lacks some of the most essential aspects of a proper object detection algorithm, such as scale and rotation invariance; moreover, it's a pure pixel-based object detection algorithm. Building upon that, we learned how to use global maximum- and minimum-detection algorithms to interpret the template-matching algorithm result. Then, we learned about corner- and edge-detection algorithms, or in other words, algorithms that detect points and areas of significance in images. We learned how to visualize them, and then moved on to learn about contour-detection and shape-analysis algorithms. The final section of this chapter included a complete tutorial on how to detect keypoints in an image, extract descriptors from those keypoints, and use matcher algorithms to detect an object in a scene. We're now familiar with a huge set of algorithms that can be used to analyze images based not only on their pixel colors and intensity values, but also their content and existing keypoints.

The final chapter of this book will take us through computer vision and machine learning algorithms in OpenCV and how they are employed to detect objects using a previously existing set of their images, among many other interesting artificial-intelligence-related topics.

# Questions

1. The template-matching algorithm is not scale- and rotation-invariant by itself. How can we make it so for a) double the scale of the template image, and b) a 90-degrees-rotated version of the template image?
2. Use the `GFTTDetector` class to detect keypoints with the Harris corner-detection algorithm. You can set any values for the corner-detection algorithm.
3. The Hough transformation can also be used to detect circles in an image, using the `HoughCircles` function. Search for it in the OpenCV documentation and write a program to detect circles in an image.
4. Detect and draw the convex contours in an image.
5. Use the `ORB` class to detect keypoints in two images, extract their descriptors, and match them.
6. Which feature-descriptor-matching algorithm is incompatible with the ORB algorithm, and why?
7. You can use the following OpenCV functions and the given sample to calculate the time required to run any number of lines of code. Use it to calculate the time it takes for the matching algorithms on your computer:

```
double freq = getTickFrequency(); double countBefore = getTickCount();  
  
// your code goes here ..  
  
double countAfter = getTickCount(); cout << "Duration: " << (countAfter -  
countBefore) / freq << " seconds";
```

# Machine Learning in Computer Vision

In the previous chapters, we learned about a number of algorithms for object detection and tracking. We learned how to use color-based algorithms, such as Mean Shift and CAM Shift, in conjunction with histograms and back-projection images to locate an object in an image with incredible speed. We also learned about template matching and how it can be used to find objects with a known template of pixels in an image. All of these algorithms rely in one way or another on image properties, such as brightness or color, that are easily affected by a change in lighting of the environment. Based on these facts, we moved on to learn about algorithms that are based on knowledge about significant areas in an image, called **keypoints** or **features**. We learned about many edge- and keypoint-detection algorithms and how to extract descriptors for those keypoints. We also learned about descriptor matchers and how to detect an object in an image using good matches of descriptors extracted from an image of the object of interest and the scene where we're looking for that object.

In this chapter, we're going to take one big step forward and learn about algorithms that can be used to extract a model from a large number of images of an object, and later use that model to detect an object in an image or simply classify an image. Such algorithms are the meeting point of machine learning algorithms and computer vision algorithms. Anyone familiar with artificial intelligence and machine learning algorithms in general will have an easy time proceeding with this chapter, even if they are not fluent in the exact algorithms and examples presented in this chapter. However, those who are totally new to such concepts will probably need to grab another book, preferably about machine learning, to familiarize themselves with algorithms, such as **support vector machines (SVM)**, **artificial neural networks (ANN)**, cascade classification, and deep learning, which we'll be learning about in this chapter.

In this chapter, we'll look at the following:

- How to train and use SVM for classification
- Using HOG and SVM for image classification
- How to train and use ANN for prediction
- How to train and use Haar or LBP cascade classifiers for real-time object detection
- How to use pre-trained models from third-party deep learning frameworks

# Technical requirements

- An IDE to develop C++ or Python applications
- The OpenCV library

Refer to [chapter 2, \*Getting Started with OpenCV\*](#) for more information about how to set up a personal computer and make it ready for developing computer vision applications using the OpenCV library.

You can use this URL to download the source codes and examples for this chapter: <https://github.com/PacktPublishing/Hands-On-Algorithms-for-Computer-Vision/tree/master/Chapter08>.

# Support vector machines

To put it as simply as possible, SVMs are used for creating a model from a labeled set of training samples that can be used to predict the label of new samples. For instance, assume we have a set of sample data belonging to two different groups. Each sample in our training dataset is a vector of floating-point numbers that can correspond to anything, such as a simple point in 2D or 3D space, and each sample is labeled with a number, such as 1, 2, or 3. Having such data, we can train an SVM model that can be used to predict the label of new 2D or 3D points. Let's think about another problem. Imagine we have the data of temperatures for 365 days in cities from all the continents in the world, and each vector of the 365 temperature values is labeled with 1 for Asia, 2 for Europe, 3 for Africa, and so on. We can use this data to train an SVM model that can be used to predict the continent of new vectors of temperature values (for 365 days) and associate them with a label. Even though these examples might not be useful in practice, they describe the concept of SVMs.

We can use the `svm` class in OpenCV to train and use SVM models. Let's go through the usage of the `svm` class in detail with a complete example:

1. Since machine learning algorithms in OpenCV are included under the `m1` namespace, we need to make sure we include those namespaces in our code, so that the classes within them are easily accessible, using the following code:

```
| using namespace cv;
| using namespace ml;
```

2. Create the training dataset. As we mentioned before, the training dataset is a set of vectors (samples) of floating-point numbers, and each vector is labeled with the class ID or category of that vector. Let's start with samples first:

```
| const int SAMPLE_COUNT = 8;
| float samplesA[SAMPLE_COUNT][2]
|   = { {250, 50},
|     {125, 100},
```

```

{50, 50},
{150, 150},
{100, 250},
{250, 250},
{150, 50},
{50, 250} };
Mat samples(SAMPLE_COUNT, 2, CV_32F, samplesA);

```

In this example, each sample in our dataset of eight samples contains two floating-point values that can be demonstrated using a point on an image with an *x* and *y* value.

3. We also need to create the label (or response) data, which obviously must be the same length as the samples. Here it is:

```

int responsesA[SAMPLE_COUNT]
    = {2, 2, 2, 2, 1, 2, 2, 1};
Mat responses(SAMPLE_COUNT, 1, CV_32S, responsesA);

```

As you can see, our samples are labeled with the `1` and `2` values, so we're expecting our model to be able to differentiate new samples between the given two groups of samples.

4. OpenCV uses the `TrainData` class to simplify the preparation and usage of the training dataset. Here's how it's used:

```

Ptr<TrainData> data;
SampleTypes layout = ROW_SAMPLE;
data = TrainData::create(samples,
                        layout,
                        responses);

```

`layout` in the preceding code is set to `ROW_SAMPLE` because each row in our dataset contains one sample. If the layout of the dataset was vertical, in other words, if each sample in the dataset was a column in the `samples` matrix, we'd need to set `layout` to `COL_SAMPLE`.

5. Create the actual `svm` class instance. This class in OpenCV implements various types of SVM classification algorithms, and they can be used by setting the correct parameters. In this example, we're going to use the most basic (and common) set of parameters for the `svm` class, but to be able to use all possible features of this algorithm, make sure to go through the OpenCV `svm` class documentation pages. Here's an example that shows how we can

use SVM to perform a linear  $n$ -class classification:

```
Ptr<SVM> svm = SVM::create();
svm->setType(SVM::C_SVC);
svm->setKernel(SVM::LINEAR);
svm->setTermCriteria(
    TermCriteria(TermCriteria::MAX_ITER +
    TermCriteria::EPS,
    100,
    1e-6));
```

6. Train the SVM model using the `train` (or `trainAuto`) method, as seen here:

```
if(!svm->train(data))
{
    cout << "training failed" << endl;
    return -1;
}
```

Based on the amount of data in our training samples dataset, the training process might take some time. In our case, it should be fast enough though, since we just used a handful of samples to train the model.

7. We're going to use the SVM model to actually predict the label of new samples. Remember that each sample in our training set was a 2D point in an image. We're going to find the label of each 2D point in an image with a width and height of 300 pixels, and then color each pixel as green or blue, based on whether its predicted label is 1 or 2. Here's how:

```
Mat image = Mat::zeros(300,
                      300,
                      CV_8UC3);
Vec3b blue(255,0,0), green(0,255,0);
for (int i=0; i<image.rows; ++i)
{
    for (int j=0; j<image.cols; ++j)
    {
        Mat<float> sampleMat(1,2);
        sampleMat << j, i;
        float response = svm->predict(sampleMat);

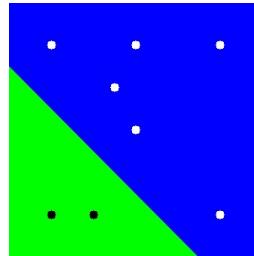
        if (response == 1)
            image.at<Vec3b>(i, j) = green;
        else if (response == 2)
            image.at<Vec3b>(i, j) = blue;
    }
}
```

8. Go ahead and display the result of predictions, but, to be able to perfectly visualize the classification result of the SVM algorithm, it's better to draw

the training samples we used to create the SVM model. Let's do it using the following code:

```
Vec3b black(0,0,0), white(255,255,255), color;
for(int i=0; i<SAMPLE_COUNT; i++)
{
    Point p(samplesA[i][0],
             samplesA[i][1]);
    if (responsesA[i] == 1)
        color = black;
    else if (responsesA[i] == 2)
        color = white;
    circle(image,
           p,
           5,
           color,
           CV_FILLED);
}
```

The two types of samples (<sub>1</sub> and <sub>2</sub>) are drawn as black and white circles over the resultant image. The following diagram depicts the result of the complete SVM classification we just performed:

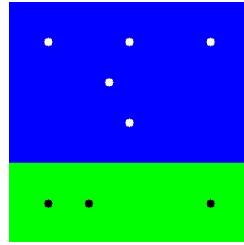


This demonstration is quite simple and, in reality, SVM can be used for much more complex classification problems, however, it literally shows the most essential aspect of SVM, which is the separation of various groups of data that are labeled the same. As you can see in the preceding image, the line separating the blue region from the green region is the best single line that can most efficiently separate the black dots and white dots on the image.

You can experiment with this phenomenon by updating the labels, or, in other words, the responses in the preceding example, as seen here:

```
| int responsesA[SAMPLE_COUNT]
|   = {2, 2, 2, 2, 1, 1, 2, 1};
```

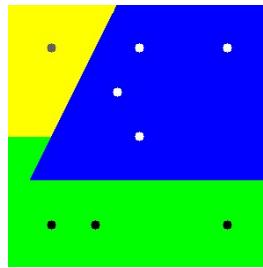
Trying to visualize the results now will produce something similar to the following, which again depicts the most efficient line for separating the two groups of dots:



You can very easily add more classes to your data, or, in other words, have more labels for your training sample set. Here's an example:

```
| int responsesA[SAMPLE_COUNT]  
|   = {2, 2, 3, 2, 1, 1, 2, 1};
```

We can try visualizing the results again by adding a yellow color, for instance, for the third class region, and a gray dot for training samples that belong to that class. Here's the result of the same SVM example when used with three classes instead of two:



If you recall the example of 365 days from before, it is quite obvious that we can also add more dimensionality to the SVM model and not just classes, but it wouldn't be visually possible to display the results with a simple image such as the one in the preceding example.

Before continuing with the usage of the SVM algorithm for actual object detection and image classification, it's worth noting that, just like any other machine learning algorithm, having more samples in your dataset will result in a much better classification and higher accuracy, but it will also take more time to train the model.

# Classifying images using SVM and HOG

**Histogram of Oriented Gradients (HOG)** is an algorithm that can be used to describe an image using a vector of floating-point descriptors that correspond to the oriented gradient values extracted from that image. The HOG algorithm is very popular and certainly worth reading about in detail to understand how it is implemented in OpenCV, but, for the purposes of this book and especially this section, we'll just mention that the number of the floating-point descriptors will always be the same when they are extracted from images that have exactly the same size with the same HOG parameters. To better understand this, recall that descriptors extracted from an image using the feature detection algorithms we learned about in the previous chapter can have different numbers of elements in them. The HOG algorithm, though, will always produce a vector of the same length if the parameters are unchanged across a set of images of the same size.

This makes the HOG algorithm ideal for being used in conjunction with SVM, to train a model that can be used to classify images. Let's see how it's done with an example. Imagine we have a set of images that contain images of a traffic sign in one folder, and anything but that specific traffic sign in another folder. The following pictures depicts the images in our samples dataset, separated by a black line in between:



Using images similar to the preceding samples, we're going to train the SVM model to detect whether an image is the traffic sign we're looking for or not. Let's start:

1. Create an `HOGDescriptor` object. `HOGDescriptor`, or the HOG algorithm, is a special type of descriptor algorithm that relies on a given window size, block size, and various other parameters; for the sake of simplicity, we'll avoid all but the window size. The HOG algorithm's window size in our example is 128 by 128 pixels, which is set as seen here:

```
| HOGDescriptor hog;
hog.winSize = Size(128, 128);
```

Sample images should have the same size as the window size, otherwise we need to use the `resize` function to make sure they are resized to the HOG window size later on. This guarantees the same descriptor size every time the HOG algorithm is used.

2. As we just mentioned, the vector length of the descriptor extracted using `HOGDescriptor` will be constant if the image size is constant, and, assuming that image has the same size as `winSize`, you can get the descriptor length using the following code:

```
| vector<float> tempDesc;
hog.compute(Mat(hog.winSize, CV_8UC3),
```

```

    tempDesc);
int descriptorSize = tempDesc.size();

```

We'll use `descriptorSize` later on when we read the sample images.

3. Assuming the images of the traffic sign are inside a folder called `pos` (for positive) and the rest inside a folder called `neg` (for negative), we can use the `glob` function to get the list of image files in those folders, as seen here:

```

vector<String> posFiles;
glob("/pos", posFiles);

vector<String> negFiles;
glob("/neg", negFiles);

```

4. Create buffers to store the HOG descriptors for negative and positive sample images (from `pos` and `neg` folders). We also need an additional buffer for the labels (or responses), as seen in the following example:

```

int scount = posFiles.size() + negFiles.size();

Mat samples(scount,
            descriptorSize,
            CV_32F);

Mat responses(scount,
               1,
               CV_32S);

```

5. We need to use the `HOGDescriptor` class to extract the HOG descriptors from positive images and store them in `samples`, as seen here:

```

for(int i=0; i<posFiles.size(); i++)
{
    Mat image = imread(posFiles.at(i));
    if(image.empty())
        continue;
    vector<float> descriptors;
    if((image.cols != hog.winSize.width)
        ||
        (image.rows != hog.winSize.height))
    {
        resize(image, image, hog.winSize);
    }
    hog.compute(image, descriptors);
    Mat(1, descriptorSize, CV_32F, descriptors.data())
        .copyTo(samples.row(i));
    responses.at<int>(i) = +1; // positive
}

```

It needs to be noted that we have added `+1` for the labels (responses) of the positive samples. We'll need to use a different number, such as `-1`,

when we label the negative samples.

6. After the positive samples, we add the negative samples and their responses to the designated buffers:

```
for(int i=0; i<negFiles.size(); i++)
{
    Mat image = imread(negFiles.at(i));
    if(image.empty())
        continue;
    vector<float> descriptors;
    if((image.cols != hog.winSize.width)
        ||
        (image.rows != hog.winSize.height))
    {
        resize(image, image, hog.winSize);
    }
    hog.compute(image, descriptors);
    Mat(1, descriptorSize, CV_32F, descriptors.data())
        .copyTo(samples.row(i + posFiles.size()));
    responses.at<int>(i + posFiles.size()) = -1;
}
```

7. Similar to the example from the previous section, we need to form a `TrainData` object using `samples` and `responses` to be used with the `train` function. Here's how it's done:

```
Ptr<TrainData> tdata = TrainData::create(samples,
                                            ROW_SAMPLE,
                                            responses);
```

8. Now, we need to train the SVM model as seen in the following example code:

```
Ptr<SVM> svm = SVM::create();
svm->setType(SVM::C_SVC);
svm->setKernel(SVM::LINEAR);
svm->setTermCriteria(
    TermCriteria(TermCriteria::MAX_ITER +
                TermCriteria::EPS,
                10000,
                1e-6));
svm->train(tdata);
```

After the training is completed, the SVM model is ready to be used for classifying images with the same size as the HOG window size (in this case, 128

by 128 pixels) using the `predict` method of the `svm` class. Here is how:

```
Mat image = imread("image.jpg");
if((image.cols != hog.winSize.width)
    ||
    (image.rows != hog.winSize.height))
{
    resize(image, image, hog.winSize);
}

vector<float> descs;
hog.compute(image, descs);
int result = svm->predict(descs);
if(result == +1)
{
    cout << "Image contains a traffic sign." << endl;
}
else if(result == -1)
{
    cout << "Image does not contain a traffic sign." << endl;
}
```

In the preceding code, we simply read an image and resize it to the HOG window size. Then we use the `compute` method of the `HOGDescriptor` class, just like when we were training the model. Except, this time, we use the `predict` method to find the label of this new image. If the `result` equals `+1`, which was the label we assigned for traffic sign images when we trained the SVM model, then we know that the image is the image of a traffic sign, otherwise it's not.

The accuracy of the result completely depends on the quantity and quality of the data you have used to train your SVM model. This, in fact, is the case for each and every machine learning algorithm. The more you train your model, the more accurate it becomes.



*This method of classification assumes that the input image is of the same characteristics as the trained images. Meaning, if the image contains a traffic sign, it is cropped similarly to the images we used to train the model. For instance, if you use an image that contains the traffic sign image we're looking for, but also contain much more, then the result will probably be incorrect.*

As the amount of data in your training set increases, it will take more time to train your model. So, it's important to avoid retraining your model every time you want to use it. The `svm` class allows you to save and load SVM models using the `save` and `load` methods. Here is how you can save a trained SVM model for later use and to avoid retraining it:

```
| svm->save("trained_svm_model.xml");
```

The file will be saved using the provided filename and extension (XML or any other file type supported by OpenCV). Later, using the static `load` function, you can create an SVM object that contains the exact parameters and trained model. Here's an example:

```
| Ptr<SVM> svm = SVM::load("trained_svm_model.xml ");
```

Try using the `svm` class along with `HOGDescriptor` to train models that can detect and classify more types using images of various objects stored in different folders.

# Training models with artificial neural networks

ANN can be used to train a model using a set of sample input and output vectors. ANN is a highly popular machine learning algorithm and the basis of many modern artificial intelligence algorithms that are used to train models for classification and correlation. Especially in computer vision, the ANN algorithm can be used along with a wide range of feature-description algorithms to learn about images of objects, or even faces of different people, and then used to detect them in images.

You can use the `ANN_MLP` class (which stands for **artificial neural networks —multi-layer perceptron**) in OpenCV to implement ANN in your applications. The usage of this class is quite similar to that of the `svm` class, so we're going to give a simple example to learn the differences and how it's used in practice, and we'll leave the rest for you to discover by yourself.

Creating the training samples dataset is exactly the same for all machine learning algorithms in OpenCV, or, to be precise, for all subclasses of the `StatsModel` class. The `ANN_MLP` class is no exception to this, so, just like with the `svm` class, first we need to create a `TrainData` object that contains all the sample and response data that we need to use when training our ANN model, as seen here:

```
| SampleTypes layout = ROW_SAMPLE;
| data = TrainData::create(samples,
|                         layout,
|                         responses);
```

`samples` and `responses`, in the preceding code, are both `Mat` objects that contain a number of rows that equals the number of all the training data we have in our dataset. As for the number of columns in them, let's recall that the ANN algorithm can be used to learn the relationship between vectors of input and output data. This means that the number of columns in the training input data, or `samples`, can be different from the number of columns in the training output data,

or responses. We'll refer to the number of columns in `samples` as the number of features, and to the number of columns in `responses` as the number of classes. Simply put, we're going to learn the relationship of features to classes using a training dataset.

After taking care of the training dataset, we need to create an `ANN_MLP` object using the following code:

```
| PPtr<ANN_MLP> ann = ANN_MLP::create();
```

We have skipped all the customizations and used the default set of parameters. In the case that you need to use a fully customized `ANN_MLP` object, you need to set the activation function, termination criteria, and various other parameters in the `ANN_MLP` class. To learn more about this, make sure to refer to the OpenCV documentation and online resources about artificial neural networks.

Setting the correct layer sizes in the ANN algorithm requires experience and depends on the use case, but it can also be set using a few trial-and-error sessions. Here's how you can set the number and size of each layer in the ANN algorithm, and the `ANN_MLP` class to be specific:

```
| Mat<int> layers(4,1);
| layers(0) = featureCount;      // input layer
| layers(1) = classCount * 4;    // hidden layer 1
| layers(2) = classCount * 2;    // hidden layer 2
| layers(3) = classCount;        // output layer
| ann->setLayerSizes(layers);
```

In the preceding code, the number of rows in the `layers` object refers to the number of layers we want to have in our ANN. The first element in the `layers` object should contain the number of features in our dataset, and the last element in the `layers` object should contain the number of classes. Recall that the number of features equals the column count of `samples`, and the number of classes equals the column count of `responses`. The rest of the elements in the `layers` object contain the sizes of hidden layers.

Training the ANN model is done by using the `train` method, as seen in the following example:

```
| if(!ann->train(data))
```

```
| {  
|     cout << "training failed" << endl;  
|     return -1;  
| }
```

After the training is completed, we can use the `save` and `load` methods in exactly the same way as we saw before, to save the model for later use, or reload it from a saved file.

Using the model with the `ANN_MLP` class is also quite similar to the `SVM` class. Here's an example:

```
| Mat<float> input(1, featureCount);  
| Mat<float> output(1, classCount);  
| // fill the input Mat  
| ann->predict(input, output);
```

Choosing the right machine learning algorithm for each problem requires experience and knowledge about where the project is going to be used. SVM is quite simple, and suitable when we need to work with the classification of data and in the segmentation of groups of similar data, whereas ANN can be easily used to approximate a function between sets of input and output vectors (regression). Make sure to try out different machine learning problems to better understand where and when to use a specific algorithm.

# The cascading classification algorithm

Cascading classification is another machine learning algorithm that can be used to train a model from many (hundreds, or even thousands) positive and negative image samples. As we explained earlier, a positive image refers to the image in an object of interest (such as a face, a car, or a traffic signal) that we want our model to learn and later classify or detect. On the other hand, a negative image corresponds to any arbitrary image that does not contain our object of interest. The model trained using this algorithm is referred to as a cascade classifier.

The most important aspect of a cascade classifier, as can be guessed from its name, is its cascading nature of learning and detecting an object using the extracted features. The most widely used features in cascade classifiers, and consequently cascade classifier types, are Haar and **local binary pattern (LBP)**. In this section, we're going to learn how to use existing OpenCV Haar and LBP cascade classifiers to detect faces, eyes, and more in real-time, and then learn how to train our own cascade classifiers to detect any other objects.

# Object detection using cascade classifiers

To be able to use previously trained cascade classifiers in OpenCV, you can use the `CascadeClassifier` class and the simple methods it provides for loading a classifier from file or performing scale-invariant detection in images. OpenCV contains a number of trained classifiers to detect faces, eyes, and so on in real-time. If we browse to the OpenCV installation (or build) folder, it usually contains a folder called `etc`, which contains the following subfolders:

- `haarcascades`
- `lbpcascades`

`haarcascades` contains pre-trained Haar cascade classifiers. `lbpcascades`, on the other hand, contains pre-trained LBP cascade classifiers. Haar cascade classifiers are usually slower than LBP cascade classifiers, but they also provide much better accuracy in most cases. To learn about the details of Haar and LBP cascade classifiers, make sure to refer to the OpenCV documentation as well as online resources about Haar wavelets, Haar-like features, and local binary patterns. As we'll learn in the next section, LBP cascade classifiers are also a lot faster to train than Haar classifiers; with enough training data samples, you can reach a similar accuracy for both of the classifier types.

Under each one of the classifier folders we just mentioned, you can find a number of pre-trained cascade classifiers. You can load these classifiers and

prepare them for object detection in real-time using the `load` method of the `CascadeClassifier` class, as seen in the following example:

```
CascadeClassifier detector;
if(!detector.load("classifier.xml"))
{
    cout << "Can't load the provided cascade classifier." << endl;
    return -1;
}
```

After a cascade classifier is successfully loaded, you can use the `detectMultiScale` method to detect objects in an image and return a vector containing the bounding rectangles of the detected objects, as seen in the following example:

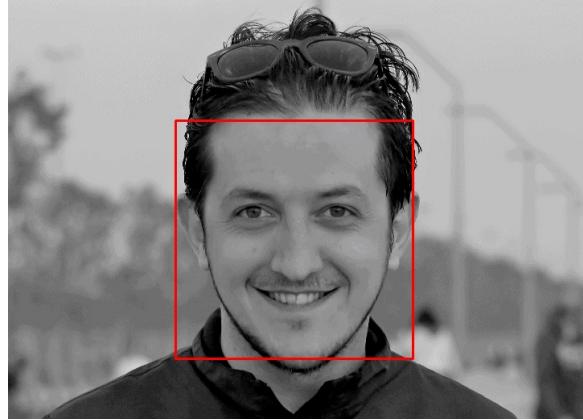
```
vector<Rect> objects;
detector.detectMultiScale(frame,
                         objects);

for(int i=0; i< objects.size(); i++)
{
    rectangle(frame,
              objects [i],
              color,
              thickness);
}
```

`color` and `thickness` are previously defined to affect the rectangle drawn for each detected object, as seen here:

```
Scalar color = Scalar(0,0,255);
int thickness = 2;
```

Try loading the `haarcascade_frontalface_default.xml` classifier in the `haarcascades` folder, which comes preinstalled with OpenCV, to test the preceding example. Trying to run the preceding code with an image that contains a face would result in something similar to this:



The accuracy of the cascade classifier, as with any other machine learning model, depends completely on the quality and quantity of the training samples dataset. As it was mentioned before, cascade classifiers are widely popular, especially for real-time object detection. To be able to view the performance of cascade classifiers on any computer, you can use the following code:

```
double t = (double)getTickCount();
detector.detectMultiScale(image,
                           objects);
t = ((double)getTickCount() - t)/getTickFrequency();
t *= 1000; // convert to ms
```

The last line in the preceding code is used to convert the unit of the time measurement from seconds to milliseconds. You can use the following code to print out the result over the output image, in the lower-left corner for example:

```
Scalar green = Scalar(0,255,0);
int thickness = 2;
double scale = 0.75;
putText(frame,
        "Took " + to_string(int(t)) + "ms to detect",
        Point(0, frame.rows-1),
        FONT_HERSHEY_SIMPLEX,
        scale,
        green,
        thickness);
```

This will produce an output image that contains text similar to what is seen in the following example:



Try different pre-trained cascade classifiers shipped with OpenCV and check their performance against each other. One very obvious observation will be the significantly faster detection speed of LBP cascade classifiers.

In the previous examples, we used only the default set of parameters needed for the `detectMultiScale` method of the `CascadeClassifier` class, however, to modify its behavior and, in some cases, to significantly improve its performance, you'll need to adjust a few more parameters, as seen in the following example:

```
double scaleFactor = 1.1;
int minNeighbors = 3;
int flags = 0; // not used
Size minSize(50,50);
Size maxSize(500, 500);
vector<Rect> objects;
detector.detectMultiScale(image,
                           objects,
                           scaleFactor,
                           minNeighbors,
                           flags,
                           minSize,
                           maxSize);
```

The `scaleFactor` parameter is used to specify the scaling of the image after each detection. This means resizing the image and performing the detection internally. This is in fact how multi-scale detection algorithms work. An image is searched for an object, its size is reduced by the given `scaleFactor`, and the search is performed again. Size reduction is performed repeatedly until the image size is smaller than the classifier size. The results from all detections in all scales are then returned. The `scaleFactor` parameter must always contain a value greater than 1.0 (not equal to and not lower than). For higher sensitivity in multi-scale detection, you can set a value such as 1.01 or 1.05, which will lead to much longer detection times, and vice versa. The `minNeighbors` parameter refers to the grouping of detections that are near or similar to each other to retain a detected object.

The `flags` parameter is simply ignored in recent versions of OpenCV. As for the `minSize` and `maxSize` parameters, they are used to specify the minimum and maximum possible sizes of an object in an image. This can significantly increase the accuracy and speed of the `detectMultiscale` function, since detected objects that do not fall into the given size range are simply ignored and rescaling is done only until `minsize` is reached.

`detectMultiscale` has two other variations that we skipped for the sake of simplifying the examples, but you should check them out for yourself to learn more about cascade classifiers and multi-scale detection in general. Make sure to also search online for pre-trained classifiers by fellow computer vision developers and try using them in your applications.

# Training cascade classifiers

As we mentioned previously, you can also create your own cascade classifiers to detect any other object if you have enough positive and negative sample images. Training a classifier using OpenCV involves taking a number of steps and using a number of special OpenCV applications, which we'll go through in this section.

# Creating samples

First things first, you need a tool called `opencv_createsamples` to prepare the positive image sample set. The negative image samples, on the other hand, are extracted automatically during the training from a provided folder containing arbitrary images that do NOT include the object of interest. The `opencv_createsamples` application can be found inside the `bin` folder of the OpenCV installation. It can be used to create the positive samples dataset, either by using a single image of the object of interest and applying distortions and transformations to it, or by using previously cropped or annotated images of the object of interest. Let's learn about the former case first.

Imagine you have the following image of a traffic sign (or any other object, for that matter) and you want to create a positive sample dataset using it:



You should also have a folder containing the source of the negative samples. As we mentioned previously, you need to have a folder containing arbitrary images that do not contain the object of interest. Let's assume we have some images similar to the following that we'll be using to create negative samples from:



Note that the size and aspect ratio of negative images, or to use the correct terminology, the background images, is not at all important. However, they must be at least as big as the minimum-detectable object (classifier size) and they must never contain images of the object of interest.

To train a proper cascade classifier, sometimes you need hundreds or even thousands of sample images that are distorted in different ways, which is not easy to create. In fact, gathering training data is one of the most time-consuming steps in creating a cascade classifier. The `opencv_createsamples` application can help with this problem by taking in the previous image of the object we're creating a classifier for and producing a positive samples dataset by applying distortions and using the background images. Here's an example of how it's used:

```
opencv_createsamples -vec samples.vec -img sign.png -bg bg.txt
    -num 250 -bgcolor 0 -bgthresh 10 -maxidev 50
    -maxxangle 0.7 -maxyangle 0.7 -maxzangle 0.5
    -w 32 -h 32
```

Here is a description of the parameters used in the preceding command:

- `vec` is used to specify the positive samples file that will be created. In this case, it is the `samples.vec` file.
- `img` is used to specify the input image that will be used to generate the

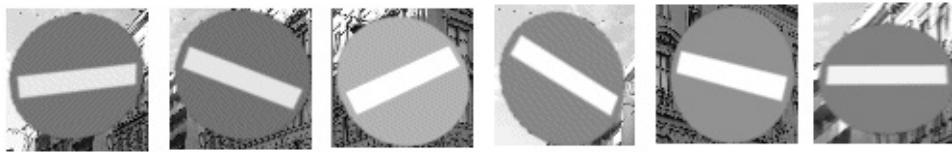
samples. In our case, it's `sign.png`.

- `bg` is used to specify the background's description file. A background's description file is a simple text file that contains the paths to all background images (each line in the background's description file contains the path to one background image). We have created a file named `bg.txt` and provided it to the `bg` parameter.
- The `num` parameter determines the number of positive samples you want to generate using the given input image and backgrounds; 250, in our case. You can, but of course, use a higher or lower number, depending on the accuracy and duration of training that you require.
- `bgcolor` can be used to define the background color in terms of its grayscale intensity. As you can see in our input image (the traffic sign image), the background color is black, thus the value of this parameter is zero.
- The `bgthresh` parameter specifies the threshold of the accepted `bgcolor` parameter. This is especially useful in the case of compression artifacts that are common to some image formats and might cause slightly different pixel values for the same color. We have used 10 for the value of this parameter to allow a slight level of tolerance for background pixels.
- `maxidev` can be used to set the maximum intensity deviation of the foreground pixel values while generating the samples. A value of 50 means the intensity of the foreground pixels can vary between their original values +/- 50.
- `maxxangle`, `maxyangle`, and `maxzangle` correspond to the maximum possible rotation allowed in the `x`, `y`, and `z` directions when creating new samples. These values are in radians, for which we have provided 0.7, 0.7, and 0.5.
- The `w` and `h` parameters define the width and height of the samples. We have used 32 for both of them since the object we're looking to train a classifier for fits in a square shape. These same values will be used later on, when training the classifier. Also note that this will be the minimum detectable size in your trained classifier later on.



*Besides the parameters in the preceding list, the `opencv_createsamples` application also accepts a `show` parameter that can be used to display the created samples, an `inv` parameter that can be used to invert the colors of samples, and a `randinv` parameter that can be used to set or unset the random inversion of pixels in samples.*

Running the preceding command will produce the given number of samples by performing rotations and intensity changes to the foreground pixels. Here are some of the resultant samples:



Now that we have a positive samples vector file, produced by `opencv_createsamples`, and a folder that contains the background images along with a background's description file (`bg.txt` from the previous example), we can start the training of our cascade classifier, but before that, let's also learn about the second method of creating our positive samples vector, which is by extracting them from various annotated images that contain our object of interest.

This second method involves using another official OpenCV tool which is used for annotating positive samples in images. This tool is called `opencv_annotation` and it can be used to conveniently mark the areas in a number of images that contain our positive samples, or in other words, the objects, we're going to train a cascade classifier for them. The `opencv_annotation` tool produces an annotation text file (after a manual annotation of the objects) that can be used with the `opencv_createsamples` tool to produce a positive samples vector suitable for use with the OpenCV cascade training tool that we'll learn about in the next section.

Let's assume we have a folder containing images similar to the following:



All of these images are located in a single folder and all of them contain one or

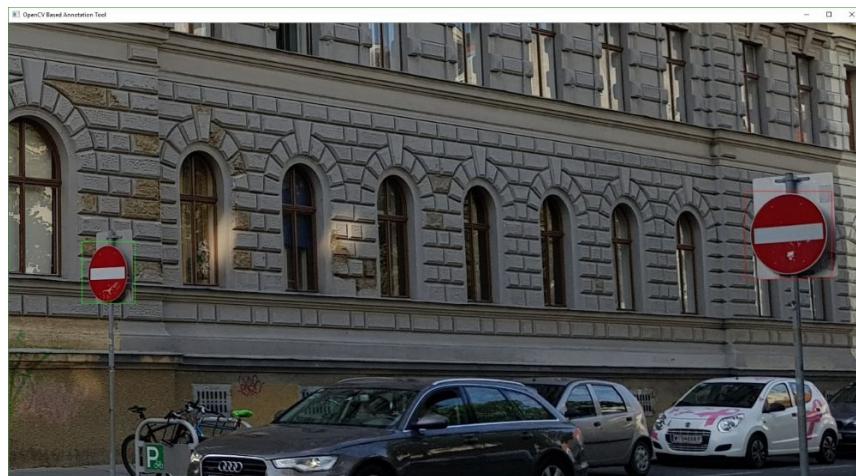
more samples of the traffic sign (the object of interest) that we're looking for. We can use the following command to start the `opencv_annotation` tool and manually annotate the samples:

```
| opencv_annotation --images=imgpath --annotations=anno.txt
```

In the preceding command, `imgpath` must be replaced with the path (preferably the absolute path and with forward slashes) to the folder containing the images. `anno.txt`, or any other file name provided instead, will be filled with the annotation results, which can be used with `opencv_createsamples` to create a positive samples vector. Executing the preceding command will start the `opencv_annotation` tool and output the following text, which describes how to use the tool and its shortcut keys:

```
* mark rectangles with the left mouse button,  
* press 'c' to accept a selection,  
* press 'd' to delete the latest selection,  
* press 'n' to proceed with next image,  
* press 'esc' to stop.
```

Immediately after the preceding output, a window similar to the following will be displayed:



You can highlight an object using your mouse's left button, which will cause a red rectangle to be drawn. Pressing the `C` key will finalize the annotation and it will become red. Continue this process for the rest of the samples (if any) in the same image and press `N` to go to the next image. After all images are annotated, you can exit the application by pressing the `Esc` key.

*In addition to the `-images` and `-annotations` parameters, the `opencv_annotation` tool also includes an*



*optional parameter, called `-maxWindowHeight`, that can be used to resize images that are bigger than a given size. The resize factor in this case can be specified with another optional parameter called `-resizeFactor`.*

The annotation file created by the `opencv_annotation` tool will look like the following:

```
| signs01.jpg 2 145 439 105 125 1469 335 185 180
| signs02.jpg 1 862 468 906 818
| signs03.jpg 1 1450 680 530 626
| signs04.jpg 1 426 326 302 298
| signs05.jpg 0
| signs06.jpg 1 1074 401 127 147
| signs07.jpg 1 1190 540 182 194
| signs08.jpg 1 794 460 470 488
```

Each line in the annotations file contains the path to an image, followed by the number of objects of interest in that image followed by the *x*, *y*, width, and height values of the bounding rectangles of those objects. You can use the following command to produce a samples vector using this annotation text file:

```
| opencv_createsamples -info anno.txt -vec samples.vec -w 32 -h 32
```

Note that this time we used the `opencv_createsamples` tool with the `-info` parameter, which wasn't present when we used this tool to generate samples from an image and arbitrary backgrounds. We are now ready to train a cascade classifier that is capable of detecting the traffic sign we created the samples for.

# Creating the classifier

The last tool we're going to learn about is called `opencv_traincascade`, which, as you can guess, is used to train cascade classifiers. If you have enough samples and background images, and if you have already taken care of the samples vector as it was described in the preceding section, then the only thing you need to do is to run the `opencv_traincascade` tool and wait for the training to be completed. Let's see an example training command and then go through the parameters in detail:

```
| opencv_traincascade -data classifier -vec samples.vec  
|   -bg bg.txt -numPos 200 -numNeg 200 -w 32 -h 32
```

This is the simplest way of starting the training process, and only uses the mandatory parameters. All parameters used in this command are self-explanatory, except the `-data` parameter, which must be an existing folder that will be used to create the files required during the training process and the final trained classifier (called `cascade.xml`) will be created in this folder.



*numPos cannot contain a number higher than the number of positive samples in your `samples.vec` file, however, `numNeg` can contain basically any number since the training process, will simply try to create random negative samples by extracting portions of the provided background images.*

The `opencv_traincascade` tool will create a number of XML files in the folder set as the `-data` parameter, which must not be modified until the training process is completed. Here is a short description for each one of them:

- The `params.xml` file will contain the parameters used for training the classifier.
- `stage#.xml` files are checkpoints that are created after each training stage is completed. They then can be used to resume the training later on if the training process was terminated for an unexpected reason.
- The `cascade.xml` file is the trained classifier and the last file that will be created by the training tool. You can copy this file, rename it to something convenient (such as `trsigh_classifier.xml` or something like that), and use it with the `CascadeClassifier` class, as we learned in the previous sections, to perform multi-scale object detection.

`opencv_traincascade` is an extremely customizable and flexible tool, and you can easily modify its many optional parameters to make sure the trained classifier fits your needs. Here is a description of some of its most used parameters:

- `numStages` can be used to set the number of stages used to train the cascade classifier. By default, `numStages` equals 20, but you can decrease this value to shorten the training time while sacrificing the accuracy or vice versa.
- The `precalcValBufSize` and `precalcIdxBufSize` parameters can be used to increase or decrease the amount of memory used for various calculations during the training of the cascade classifier. You can modify these parameters to make sure the training process is performed with more efficiency.
- `featureType` is one of the most important parameters of the training tool, and it can be used to set the type of the trained classifier to `HAAR` (default if ignored) or `LBP`. As mentioned before, LBP classifiers are trained much faster than Haar classifiers and their detection is also significantly faster, but they lack the accuracy of the Haar cascade classifiers. With a proper amount of training samples, you might be able to train an LBP classifier that can compete with a Haar classifier in terms of accuracy.

For a complete list of parameters and their descriptions, make sure to refer to the OpenCV online documentation.

# Using deep learning models

In recent years, there has been a huge improvement in the field of deep learning, or, to be precise, **deep neural networks (DNN)**, and more and more libraries and frameworks are being introduced that use deep learning algorithms and models, especially for computer vision purposes such as object detection in real-time. You can use the most recent versions of the OpenCV library to read pre-trained models for the most popular DNN frameworks, such as Caffe, Torch, and TensorFlow, and use them for object detection and prediction tasks.

DNN-related algorithms and classes in OpenCV are all located under the `dnn` namespace, so, to be able to use them, you need to make sure to include the following in your code:

```
| using namespace cv;
| using namespace dnn;
```

We're going to walk through the loading and use of a pre-trained model from the TensorFlow library in OpenCV for real-time object detection. This example demonstrates the basics of how to use deep neural network models trained by a third-party library (TensorFlow in this case). So, let's start:

1. Download a pre-trained TensorFlow model that can be used for object detection. For our example, make sure you download the latest version of `ssd_mobilenet_v1_coco` from the search, for official TensorFlow models online instead.

Note that this link can possibly change in the future (maybe not soon, but it's worth mentioning), so, in case this happens, you need to simply search online for the `TensorFlow` model zoo, which, in `TensorFlow` terms, is a zoo containing the pre-trained object detection models.

2. After downloading the `ssd_mobilenet_v1_coco` model package file, you need to extract it to a folder of your choice. You'll end up with the `frozen_inference_graph.pb` file in the folder where you extracted the model package, along with a few more files. You need to extract a text graph file from this model file before it can be used for real-time object detection in

OpenCV. This extraction can be performed by using a script called `tf_text_graph_ssd.py`, which is a Python script that is included in the OpenCV installation by default and can be found in the following path:

```
|   opencv-source-files/samples/dnntf_text_graph_ssd.py
```

You can execute this script using the following command:

```
|   tf_text_graph_ssd.py --input frozen_inference_graph.pb  
|                         --output frozen_inference_graph.pbtxt
```

 Note that the correct execution of this script totally depends on whether you have a correct TensorFlow installation on your computer or not.

3. You should have the `frozen_inference_graph.pb` and `frozen_inference_graph.pbtxt` files, so we can start using them in OpenCV to detect objects. For this reason, we need to create a DNN `Network` object and read the model files into it, as seen in the following example:

```
Net network = readNetFromTensorflow(  
    "frozen_inference_graph.pb",  
    "frozen_inference_graph.pbtxt");  
if(network.empty())  
{  
    cout << "Can't load TensorFlow model." << endl;  
    return -1;  
}
```

4. After making sure the model is correctly loaded, you can use the following code to perform a real-time object detection in a frame read from the camera, an image, or a video file:

```
const int inWidth = 300;  
const int inHeight = 300;  
const float meanVal = 127.5; // 255 divided by 2  
const float inScaleFactor = 1.0f / meanVal;  
bool swapRB = true;  
bool crop = false;  
Mat inputBlob = blobFromImage(frame,  
                               inScaleFactor,  
                               Size(inWidth, inHeight),  
                               Scalar(meanVal, meanVal, meanVal),  
                               swapRB,  
                               crop);  
  
network.setInput(inputBlob);  
  
Mat result = network.forward();
```

It's worth noting that the values passed to the `blobFromImage` function completely depend on the model, and you should use the exact same values if you're using the same model from this example. The `blobFromImage` function will create a BLOB that is suitable for use with the deep neural network's prediction function, or, to be precise, the `forward` function.

5. After the detection is complete, you can use the following code to extract the detected objects and their bounding rectangles, all into a single `Mat` object:

```
Mat detections(result.size[2],  
               result.size[3],  
               CV_32F,  
               result.ptr<float>());
```

6. The `detections` object can be looped through to extract the individual detections that have an acceptable detection confidence level and draw the results on the input image. Here's an example:

```
const float confidenceThreshold = 0.5f;  
for(int i=0; i<detections.rows; i++)  
{  
    float confidence = detections.at<float>(i, 2);  
    if(confidence > confidenceThreshold)  
    {  
        // passed the confidence threshold  
    }  
}
```

The confidence level, which is the third element in each row of the `detections` object, can be adjusted to get more accurate results, but `0.5` should be a reasonable value for most cases, or at least for a start.

7. After a detection passes the confidence criteria, we can extract the detected object ID and bounding rectangle and draw it on the input image, as seen here:

```
int objectClass = (int)(detections.at<float>(i, 1)) - 1;  
int left = static_cast<int>(  
    detections.at<float>(i, 3) * frame.cols);  
int top = static_cast<int>(  
    detections.at<float>(i, 4) * frame.rows);  
int right = static_cast<int>(  
    detections.at<float>(i, 5) * frame.cols);  
int bottom = static_cast<int>(  
    detections.at<float>(i, 6) * frame.rows);  
rectangle(frame, Point(left, top),  
         Point(right, bottom), Scalar(0, 255, 0));
```

```

String label = "ID = " + to_string(objectClass);
if(objectClass < labels.size())
    label = labels[objectClass];
int baseLine = 0;
Size labelSize = getTextSize(label, FONT_HERSHEY_SIMPLEX,
                             0.5, 2, &baseLine);
top = max(top, labelSize.height);

rectangle(frame,
          Point(left, top - labelSize.height),
          Point(left + labelSize.width, top + baseLine),
          white,
          CV_FILLED);

putText(frame, label, Point(left, top),
        FONT_HERSHEY_SIMPLEX, 0.5, red);

```

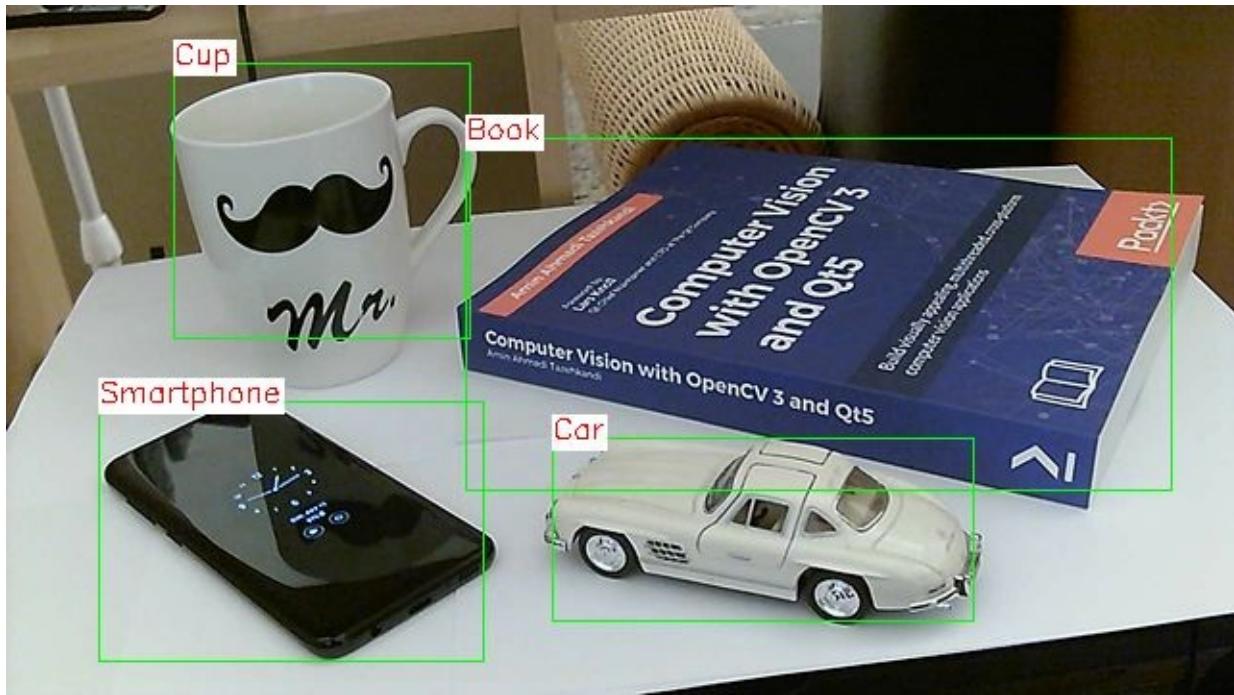
In the preceding example, `objectClass` refers to the ID of the detected object, which is the second element in each row of the detection's object. The third, fourth, fifth, and sixth elements, on the other hand, correspond to the left, top, right, and bottom values of the bounding rectangle of each detected object. The rest of the code is simply drawing the results, which leaves the `labels` object. `labels` is a vector of string values that can be used to retrieve the human-readable text of each object ID. These labels, similar to the rest of the parameters we used in this example, are model-dependent. For instance, in our example case, the labels can be found here:

[https://github.com/tensorflow/models/blob/master/research/object\\_detection/data/mscoco\\_label\\_map.pbtxt](https://github.com/tensorflow/models/blob/master/research/object_detection/data/mscoco_label_map.pbtxt)

We have converted this into the following labels vector used in the preceding example:

```
|const vector<string> labels = { "person", "bicycle" ...};
```

The following image demonstrates the result of the object detection using a pre-trained TensorFlow model in OpenCV:



Using deep learning has proven to be highly efficient, especially when we need to train and detect multiple objects in real-time. Make sure to refer to the TensorFlow and OpenCV documentation for more about how to use pre-trained models, or how to train and retrain DNN models for an object that doesn't have an already trained model.

# Summary

We started the final chapter of this book by learning about SVM models and how to train them to classify groups of similar data. We learned how SVM can be used in conjunction with the HOG descriptor to learn about one or more specific objects and then detect and classify them in new images. After learning about SVM models, we moved on to using ANN models, which offer much more power in cases where we have multiple columns in both the input and output of the training samples. This chapter also included a complete guide on how to train and use Haar and LBP cascade classifiers. We are now familiar with the usage of official OpenCV tools that can be used to prepare a training dataset from scratch and then train a cascade classifier using that dataset. Finally, we ended this chapter and this book by learning about the usage of pre-trained deep learning object detection models in OpenCV.

# Questions

1. What is the difference between the `train` and `trainAuto` methods in the `SVM` class?
2. Demonstrate the difference between the linear and histogram intersection.
3. How do you calculate the HOG descriptor size for a HOG window size of 128 x 96 pixels (the rest of the HOG parameters are untouched)?
4. How do you update an existing trained `ANN_MLP`, instead of training from scratch?
5. What is the required command (by using `opencv_createsamples`) to create a positive samples vector from a single image of a company logo? Assume we want to have 1,000 samples with a width of 24 and a height of 32, and by using default parameters for rotations and inversions.
6. What is the required command to train an LBP cascade classifier for the company logo from the previous question?
7. What is the default number of stages for training a cascade classifier in `opencv_traincascade`? How can we change it? What is the downside of increasing and decreasing the number of stages far beyond its default value?

# **Assessments**

# Chapter 1, Introduction to Computer Vision

1. Name two industries besides the ones mentioned in this chapter that can significantly benefit from computer vision.

The sports industry can use computer vision for better analysis of matches.  
The food industry can use computer vision for quality control of products.

2. What would be an example of a computer vision application used for security purposes? (Think about an idea for an application that you haven't come across.) A very random example would be an application that uses face recognition for ticket checks on trains, flights, and so on.
3. What would be an example of a computer vision application used for productivity reasons? (Again, think about an idea for an application that you haven't come across, even though you might suspect that it exists.) An application that uses its camera to help visually impaired people.
4. How many megabytes would be needed to store a 1920 x 1080 image with four channels and a depth of 32-bits?

Approximately 31.64 megabytes:

$$\begin{aligned} 32400 \div 1024 &= \\ \mathbf{31.640625} \\ \\ 33177600 \div 1024 &= \\ \mathbf{32,400} \\ \\ 265420800 \div 8 &= \\ \mathbf{33,177,600} \\ \\ 1920 \times 1080 \times 4 \times 32 &= \\ \mathbf{265,420,800} \end{aligned}$$

5. Ultra-HD images, also known as 4K, or 8K images are quite common

nowadays, but how many megapixels does an Ultra-HD image contain?

This mostly depends on the aspect ratio. For a common 16:9 aspect ratio, here are the answers:

- **4K:** 8.3 megapixels
- **8K:** 33.2 megapixels

Have a look at this link for more info:

[https://en.wikipedia.org/wiki/Ultra-high-definition\\_television](https://en.wikipedia.org/wiki/Ultra-high-definition_television)

6. Name two commonly used color spaces besides the ones mentioned in this chapter.

YUV and LUV color spaces

[https://en.wikipedia.org/wiki/List\\_of\\_color\\_spaces\\_and\\_their\\_uses](https://en.wikipedia.org/wiki/List_of_color_spaces_and_their_uses)

7. Compare OpenCV libraries with computer vision tools in MATLAB. What are the pros and cons of each one, when compared?

In general, MATLAB is best used when a computer vision application needs to be simulated and prototyped, while OpenCV is more straightforward when it comes to real-life scenarios and applications where speed and full control over the end product is needed.

# Chapter 2, Getting Started with OpenCV

1. Name three Extra OpenCV modules, along with their usage.

The `xfeatures2d` module can be used to access additional feature detection algorithms.

The `face` module can be used to include face analysis algorithms in OpenCV. The `text` module can be used to add OCR functionalities (Tesseract OCR) to OpenCV.

2. What is the effect of building OpenCV 3 with the `BUILD_opencv_world` flag turned on?

Building OpenCV 3 with the `BUILD_opencv_world` flag will combine all binary library files, such as `core`, `imcodecs`, and `highgui`, into a single world library.

3. Using the ROI pixel access method described in this chapter, how can we construct a `Mat` class that can access the middle pixel, plus all of its neighboring pixels (the middle nine pixels) in another image?

Here's an example code that can achieve this goal:

```
Mat image = imread("Test.png");
if(image.empty())
{
    cout << "image empty";
    return 0;
}
int centerRow = (image.rows / 2) - 1;
int centerCol = (image.cols / 2) - 1;
Mat roi(image, Rect(centerCol - 1, centerRow - 1, 3, 3));
roi = Scalar(0,0,255); // alter the pixels (make them red)
imshow("image", image);
waitKey();
```

4. Name another pixel access method of the `Mat` class, besides the ones mentioned in this chapter.

`Mat::row` can be used to access a single row  
`Mat::column` can be used to access a single column

5. Write a program, only using the `Mat` method and a `for` loop, that creates three separate color images, each one containing only one channel of an RGB image read from disk.

```
Mat image = imread("Test.png");
if(image.empty())
{
    cout << "image empty";
    return 0;
}
Mat red(image.rows, image.cols, CV_8UC3, Scalar::all(0));
Mat green(image.rows, image.cols, CV_8UC3, Scalar::all(0));
Mat blue(image.rows, image.cols, CV_8UC3, Scalar::all(0));
for(int i=0; i<image.rows; i++)
{
    for(int j=0; j<image.cols; j++)
    {
        blue.at<Vec3b>(i, j)[0] = image.at<Vec3b>(i, j)[0];
        green.at<Vec3b>(i, j)[1] = image.at<Vec3b>(i, j)[1];
        red.at<Vec3b>(i, j)[2] = image.at<Vec3b>(i, j)[2];
    }
}
imshow("Blue", blue);
imshow("Green", green);
imshow("Red", red);
waitKey();
```



6. Using STL-like iterators, calculate the average pixel value of a grayscale image.

```
Mat image = imread("Test.png", IMREAD_GRAYSCALE);
if(image.empty())
```

```

{
    cout << "image empty";
    return 0;
}
int sum = 0;
MatIterator<uchar> it_begin = image.begin<uchar>();
MatIterator<uchar> it_end = image.end<uchar>();
for( ; it_begin != it_end; it_begin++)
{
    sum += (*it_begin);
}
double average = sum / (image.cols * image.rows);
cout << "Pixel count is " << image.cols * image.rows << endl;
cout << "Average pixel value is " << average << endl;

```

7. Write a program using `VideoCapture`, `waitKey`, and `imwrite`, that displays your webcam and saves the visible image when the `S` key is pressed. This program will stop the webcam and exit if the spacebar is pressed.

```

VideoCapture cam(0);
if(!cam.isOpened())
    return -1;
while(true)
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
        break;
    imshow("Camera", frame);
    // stop camera if space is pressed
    char key = waitKey(10);
    if(key == ' ')
        break;
    if(key == 's')
        imwrite("d:/snapshot.png", frame);
    }
cam.release();

```

# Chapter 3, Array and Matrix Operations

1. Which of the element-wise mathematical operations and bitwise operations would produce the exact same results?

The `bitwise_xor` and `absdiff` functions would produce the same result.

2. What is the purpose of the `gemm` function in OpenCV? What is the equivalent of  $AxB$  with the `gemm` function?

The `gemm` function is the generalized multiplication function in OpenCV. Here's the `gemm` function call equivalent to the simple multiplication of two matrices:

```
|     gemm(image1, image2, 1.0, noArray(), 1.0, result);
```

3. Use the `borderInterpolate` function to calculate the value of a non-existing pixel at point (-10, 50), with a border type of `BORDER_REFLECT_101`. What is the function call required for such a calculation?

```
|     Vec3b val = image.at<Vec3b>(borderInterpolate(50,
|                                     image.rows,
|                                     cv::BORDER_REFLECT_101),
|                                     borderInterpolate(-10,
|                                     image.cols,
|                                     cv::BORDER_WRAP));
```

4. Create the same identity matrix in the *Identity matrix* section of this chapter, but use the `setIdentity` function instead of the `Mat::eye` function.

```
|     Mat m(10, 10, CV_32F);
|     setIdentity(m, Scalar(0.25));
```

5. Write a program using the `LUT` function (a look-up table transformation) that performs the same task as `bitwise_not` (invert colors) when executed on grayscale and color (RGB) images.

```
Mat image = imread("Test.png");
Mat lut(1, 256, CV_8UC1);
for(int i=0; i<256; i++)
{
    lut.at<uchar>(0, i) = 255 - i;
}
Mat result;
LUT(image, lut, result);
```

6. Besides normalizing the values of a matrix, the `normalize` function can be used to brighten or darken images. Write the required function call to darken and brighten a grayscale image using the `normalize` function.

```
normalize(image, result, 200, 255, CV_MINMAX); // brighten
normalize(image, result, 0, 50, CV_MINMAX); // darken
```

7. Remove the blue channel (the first channel) from an image (a BGR image created using the `imread` function) using the `merge` and `split` functions.

```
vector<Mat> channels;
split(image, channels);
channels[0] = Scalar::all(0);
merge(channels, result);
```

# Chapter 4, Drawing, Filtering, and Transformation

1. Write a program that draws a cross over the whole image, with a thickness of 3 pixels and a red color.

```
    line(image,
        Point(0,0),
        Point(image.cols-1,image.rows-1),
        Scalar(0,0,255),
        3);
    line(image,
        Point(0,image.rows-1),
        Point(image.cols-1,0),
        Scalar(0,0,255),
        3);
```

2. Create a window with a trackbar to change the `ksize` of a `medianBlur` function. The range for the `ksize` value should be between 3 and 99.

```
Mat image;
int ksize = 3;
string window = "Image";
string trackbar = "ksize";
void onChange(int ksize, void*)
{
if(ksize %2 == 1)
{
medianBlur(image,
image,
ksize);
imshow(window, image);
}
}
int main()
{
image = imread("Test.png");
namedWindow(window);
createTrackbar(trackbar, window, &ksize, 99, onChange);
setTrackbarMin(trackbar, window, 3);
setTrackbarMax(trackbar, window, 99);
onChange(3, NULL);
waitKey();
}
```

3. Perform a gradient morphological operation on an image, considering a kernel size of 7 and a rectangle morphological shape for the structuring element.

```
int ksize = 7;
morphologyEx(image,
```

```
| result,  
| MORPH_GRADIENT,  
| getStructuringElement(MORPH_RECT,  
| Size(ksize,ksize)));
```

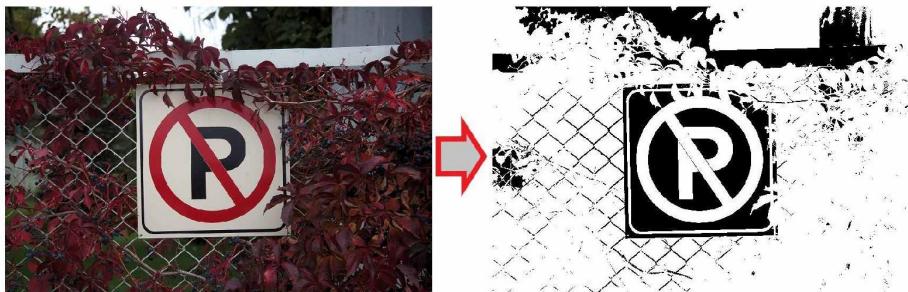
Here's an example:



4. Using `cvtColor`, convert a color image to grayscale and make sure only the darkest 100 shades of gray are filtered out using the `threshold` function. Make sure the filtered pixels are set to white in the resulting image, and the rest of the pixels are set to black.

```
| Mat imageGray;  
| cvtColor(image,  
| imageGray,  
| COLOR_BGR2GRAY);  
| threshold(imageGray,  
| result,  
| 100,  
| 255,  
| THRESH_BINARY_INV);
```

Here's an example:



5. Use the `remap` function to resize an image to half of its original width and height, thus preserving the aspect ratio of the original image. Use a default border type for the extrapolation.

```
| Mat mapX(image.size(), CV_32FC1);  
| Mat mapY(image.size(), CV_32FC1);
```

```

for(int i=0; i<image.rows; i++)
for(int j=0; j<image.cols; j++)
{
mapX.at<float>(i, j) = j*2.0;
mapY.at<float>(i, j) = i*2.0;
}
InterpolationFlags interpolation = INTER_LANCZOS4;
BorderTypes borderMode = BORDER_DEFAULT;
remap(image,
result,
mapX,
mapY,
interpolation,
borderMode);

```

Here's an example:



6. a) Use colormaps to convert an image to grayscale. b) How about converting an image to grayscale and inverting its pixels at the same time?

a)

```

Mat userColor(256, 1, CV_8UC3);
for(int i=0; i<=255; i++)
userColor.at<Vec3b>(i, 0) = Vec3b(i, i, i);
applyColorMap(image,
result,
userColor);

```

b)

```

Mat userColor(256, 1, CV_8UC3);
for(int i=0; i<=255; i++)
userColor.at<Vec3b>(i, 0) = Vec3b(255-i, 255-i, 255-i);
applyColorMap(image,
result,
userColor);

```

7. Did you read about perspective transformation functions? Which OpenCV function covers all similar transformations in one function?

The `findHomography` function.

# Chapter 5, Back-Projection and Histograms

1. Calculate the histogram of the second channel in a three-channel image. Use an optional bin size and a range of 0 to 100 for possible values of the second channel.

```
int bins = 25; // optional
int nimages = 1;
int channels[] = {1};
Mat mask;
int dims = 1;
int histSize[] = { bins };
float range[] = {0, 100};
const float* ranges[] = { range };
Mat histogram;
calcHist(&image,
nimages,
channels,
mask,
histogram,
dims,
histSize,
ranges);
```

2. Create a histogram that can be used with `calcBackProject` function to extract the darkest pixels from a grayscale image. Consider the darkest 25% possible pixel values as the grayscale intensities we are looking to extract.

```
int bins = 4;
float rangeGS[] = {0, 256};
const float* ranges[] = { rangeGS };
int channels[] = {0};
Mat histogram(bins, 1, CV_32FC1, Scalar(0.0));
histogram.at<float>(0, 0) = 255.0;
calcBackProject(&imageGray,
1,
channels,
histogram,
backProj,
ranges);
```

3. In the previous question, what if we needed the darkest and brightest 25% to be excluded, instead of extracted in a mask?

```
int bins = 4;
float rangeGS[] = {0, 256};
const float* ranges[] = { rangeGS };
```

```

int channels[] = {0};
Mat histogram(bins, 1, CV_32FC1, Scalar(0.0));
histogram.at<float>(1, 0) = 255.0;
histogram.at<float>(2, 0) = 255.0;
calcBackProject(&imageGray,
1,
channels,
histogram,
backProj,
ranges);

```

4. What is the hue value of the red color? How much should it be shifted to get the blue color?

0 and 360 are the hue values for the red color. Shifting it by 240 would result in the blue color.

5. Create a hue histogram that can be used to extract red colored pixels from an image. Consider an offset of 50 for pixels that are considered red. Finally, visualize the created hue histogram.

```

const int bins = 360;
int hueOffset = 35;
Mat histogram(bins, 1, CV_32FC1);
for(int i=0; i<bins; i++)
{
    histogram.at<float>(i, 0) =
    (i < hueOffset) || (i > bins - hueOffset) ? 255.0 : 0.0;
}
double maxVal = 255.0;
int gW = 800, gH = 100;
Mat theGraph(gH, gW, CV_8UC3, Scalar::all(0));
Mat colors(1, bins, CV_8UC3);
for(int i=0; i<bins; i++)
{
    colors.at<Vec3b>(i) =
    Vec3b(saturate_cast<uchar>(
    (i+1)*180.0/bins), 255, 255);
}
cvtColor(colors, colors, COLOR_HSV2BGR);
Point p1(0,0), p2(0,theGraph.rows-1);
for(int i=0; i<bins; i++)
{
    float value = histogram.at<float>(i,0);
    value = maxVal - value; // invert
    value = value / maxVal * theGraph.rows; // scale
    p1.y = value;
    p2.x = float(i+1) * float(theGraph.cols) / float(bins);
    rectangle(theGraph,
    p1,
    p2,
    Scalar(colors.at<Vec3b>(i)),
    CV_FILLED);
    p1.x = p2.x;
}

```



6. Calculate the integral of a histogram.

```
float integral = 0.0;
for(int i=0; i<bins; i++)
{
    integral += histogram.at<float>(i, 0);
```

7. Perform histogram equalization on a color image. Note that the `equalizeHist` function only supports histogram equalization of single-channel 8-bit grayscale images.

```
Mat channels[3], equalized[3];
split(image, channels);
equalizeHist(channels[0], equalized[0]);
equalizeHist(channels[1], equalized[1]);
equalizeHist(channels[2], equalized[2]);
Mat output;
cv::merge(equalized, 3, output);
```

# Chapter 6, Video Analysis – Motion Detection and Tracking

1. All the examples in this chapter that deal with cameras return when there is a single failed or corrupted frame that leads to the detection of an empty frame. What type of modification is needed to allow a predefined number of retries before stopping the process?

```
const int RETRY_COUNT = 10;
int retries = RETRY_COUNT;
while(true)
{
    Mat frame;
    cam >> frame;
    if(frame.empty())
    {
        if(--retries < 0)
            break;
        else
            continue;
    }
    else
    {
        retries = RETRY_COUNT;
    }
    // rest of the process
}
```

2. How can we call the `meanShift` function to perform the Mean Shift algorithm with 10 iterations and an epsilon value of 0.5?

```
TermCriteria criteria(TermCriteria::MAX_ITER
+ TermCriteria::EPS,
10,
0.5);
meanShift(backProject,
srchWnd,
criteria);
```

3. How do you visualize the hue histogram of the tracked object? Assume that `camShift` is used for tracking.

```
Having the following function:
void visualizeHue(Mat hue)
{
    int bins = 36;
    int histSize[] = {bins};
    int nimages = 1;
```

```

int dims = 1;
int channels[] = {0};
float rangeHue[] = {0, 180};
const float* ranges[] = {rangeHue};
bool uniform = true;
bool accumulate = false;
Mat histogram, mask;
calcHist(&hue,
        nimages,
        channels,
        mask,
        histogram,
        dims,
        histSize,
        ranges,
        uniform,
        accumulate);
double maxVal;
minMaxLoc(histogram,
0,
&maxVal,
0,
0);
int gW = 800, gH = 100;
Mat theGraph(gH, gW, CV_8UC3, Scalar::all(0));
Mat colors(1, bins, CV_8UC3);
for(int i=0; i<bins; i++)
{
colors.at<Vec3b>(i) =
Vec3b(saturate_cast<uchar>(
(i+1)*180.0/bins), 255, 255);
}
cvtColor(colors, colors, COLOR_HSV2BGR);
Point p1(0,0), p2(0,theGraph.rows-1);
for(int i=0; i<bins; i++)
{
float value = histogram.at<float>(i,0);
value = maxVal - value; // invert
value = value / maxVal * theGraph.rows; // scale
p1.y = value;
p2.x = float(i+1) * float(theGraph.cols) / float(bins);
rectangle(theGraph,
p1,
p2,
Scalar(colors.at<Vec3b>(i)),
CV_FILLED);
p1.x = p2.x;
}
imshow("Graph", theGraph);
}

```

We can call the following right after the `camShift` function call to visualize the hue of the detected object:

```

| CamShift(backProject,
| srchWnd,
| criteria);
| visualizeHue(Mat(hue, srchWnd));

```

4. Set the process noise covariance in the `KalmanFilter` class so that the filtered and

measured values overlap. Assume that only process noise covariance is set of all the available matrices for the `KalmanFilter` class's behavior control.

```
|     setIdentity(kalman.processNoiseCov,  
|                 Scalar::all(1.0));
```



5. Let's assume that the Y position of the mouse on a window is used to describe the height of a filled rectangle that starts from the top-left corner of the window and has a width that equals the window's width. Write a Kalman filter that can be used to correct the height of the rectangle (single value) and remove noise in the mouse's movement, resulting in a visually smooth resizing of the filled rectangle.

```
int fillHeight = 0;  
void onMouse(int, int, int y, int, void*)  
{  
    fillHeight = y;  
}  
int main()  
{  
    KalmanFilter kalman(2,1);  
    Mat<float> tm(2, 2); // transition matrix  
    tm << 1,0,  
        0,1;  
    kalman.transitionMatrix = tm;  
    Mat<float> h(1,1);  
    h.at<float>(0) = 0;  
    kalman.statePre.at<float>(0) = 0; // init x  
    kalman.statePre.at<float>(1) = 0; // init x'  
    setIdentity(kalman.measurementMatrix);  
    setIdentity(kalman.processNoiseCov,  
                Scalar::all(0.001));  
    string window = "Canvas";  
    namedWindow(window);  
    setMouseCallback(window, onMouse);  
    while(waitKey(10) < 0)  
    {  
        // empty canvas  
        Mat canvas(500, 500, CV_8UC3, Scalar(255, 255, 255));  
        h(0) = fillHeight;  
        Mat estimation = kalman.correct(h);  
        float estH = estimation.at<float>(0);  
        rectangle(canvas,  
                  Rect(0,0,canvas.cols, estH),  
                  Scalar(0),  
                  FILLED);  
        imshow(window, canvas);  
    }
```

```
    kalman.predict();
}
return 0;
}
```

6. Create a `BackgroundSubtractorMOG2` object to extract the foreground image's contents while avoiding the shadow changes.

```
Ptr<BackgroundSubtractorMOG2> bgs =
createBackgroundSubtractorMOG2(500, // hist
16, // thresh
false // no shadows
);
```

7. Write a program to display the current (as opposed to sampled) background image using a background segmentation algorithm.

```
VideoCapture cam(0);
if(!cam.isOpened())
return -1;
Ptr<BackgroundSubtractorKNN> bgs =
createBackgroundSubtractorKNN();
while(true)
{
Mat frame;
cam >> frame;
if(frame.empty())
break;
Mat mask;
bgs->apply(frame,
mask);
bitwise_not(mask, mask);
Mat bg;
bitwise_and(frame, frame, bg, mask);
imshow("bg", bg);
int key = waitKey(10);
if(key == 27) // escape key
break;
}
cam.release();
```

# Chapter 7, Object Detection – Features and Descriptors

1. The template matching algorithm is not scale- and rotation-invariant by itself. How can we make it so for a) double the scale of the template image, and b) a 90 degrees rotated version of the template image?

- a) Use the `resize` function to scale the template image, and then call the `matchTemplate` function:

```
| resize(temp1, templ, Size(), 2.0, 2.0);
| matchTemplate(image, templ, TM_CCOEFF_NORMED);
```

- b) Rotate the template by 90 degrees and then call the `matchTemplate` function:

```
| rotate(temp1, templ, ROTATE_90_CLOCKWISE);
| matchTemplate(image, templ, TM_CCOEFF_NORMED);
```

2. Use the `GFTTDetector` class to detect the keypoints with the Harris corner-detection algorithm. You can set any values for the corner-detection algorithm.

```
| Mat image = imread("Test.png");
| Ptr<GFTTDetector> detector =
| GFTTDetector::create(500,
| 0.01,
| 1,
| 3,
| true);
| vector<KeyPoint> keypoints;
| detector->detect(image, keypoints);
| drawKeypoints(image,
| keypoints,
| image);
```

3. The Hough transformation can also be used to detect circles in an image using the `Houghcircles` function. Search for it in the OpenCV documentation and write a program to detect circles in an image.

```
| Mat image = imread("Test.png");
| cvtColor(image, image, COLOR_BGR2GRAY);
| vector<Vec3f> circles;
| HoughCircles(image,
```

```

circles,
HOUGH_GRADIENT,
2,
image.rows/4);
for(int i=0; i<circles.size(); i++)
{
Point center(cvRound(circles[i][0]),
cvRound(circles[i][1]));
int radius = cvRound(circles[i][2]);
circle( image, center, radius, Scalar(0,0,255));
}

```

#### 4. Detect and draw the convex contours in an image.

```

Mat image = imread("Test.png");
Mat imgGray;
cvtColor(image, imgGray, COLOR_BGR2GRAY);
double threshold1 = 100.0;
double threshold2 = 200.0;
int apertureSize = 3;
bool L2gradient = false;
Mat edges;
Canny(image,
edges,
threshold1,
threshold2,
apertureSize,
L2gradient);
vector<vector<Point> > contours;
int mode = CV_RETR_TREE;
int method = CV_CHAIN_APPROX_TC89_KCOS;
findContours(edges,
contours,
mode,
method);
Mat result(image.size(), CV_8UC3, Scalar::all(0));
for( int i = 0; i< contours.size(); i++ )
{
if(isContourConvex(contours[i]))
{
drawContours(result,
contours,
i,
Scalar(0, 255, 0),
2);
}
}

```

#### 5. Use the ORB class to detect keypoints in two images, extract their descriptors, and match them.

```

Mat object = imread("Object.png");
Mat scene = imread("Scene.png");
Ptr<ORB> orb = ORB::create();
vector<KeyPoint> objKPs, scnKPs;
Mat objDesc, scnDesc;
orb->detectAndCompute(object,
Mat(),
objKPs,
objDesc);

```

```
orb->detectAndCompute(scene,
Mat(),
scnKPs,
scnDesc);
Ptr<BFMatcher> matcher = BFMatcher::create();
vector<DMatch> matches;
matcher->match(objDesc, scnDesc, matches);
Mat result;
drawMatches(object,
objKPs,
scene,
scnKPs,
matches,
result);
imshow("image", result);
```

6. Which feature descriptor matching algorithm is incompatible with the ORB algorithm, and why?

You cannot use the FLANN-based matching algorithm with descriptors that have a bit string type, such as ORB.

7. You can use the following OpenCV functions and the sample to calculate the time required to run any number of lines of code. Use it to calculate the time it takes for the matching algorithms on your computer.

```
double freq = getTickFrequency();
double countBefore = getTickCount();
// your code goes here ..
double countAfter = getTickCount();
cout << "Duration: " <<
(countAfter - countBefore) / freq << " seconds";
```

# Chapter 8, Machine Learning in Computer Vision

1. What is the difference between the `train` and `trainAuto` methods in the `svm` class?

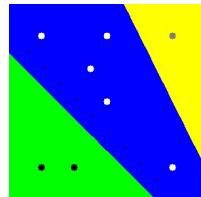
The `trainAuto` method chooses the optimal values for the SVM parameters, such as `c`, `gamma` and so on, and trains the model, while the `train` method simply uses any given parameters. (Read the `svm` class documentation for more details about the `trainAuto` function and how exactly the optimization happens.)

2. Demonstrate the difference between linear and histogram intersection.

We can set the kernel type to `LINEAR` using the following code:

```
| svm->setKernel(SVM::LINEAR);
```

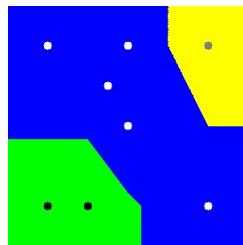
Here is the result of the classification (segmentation) if we have displayed groups of black, white, and gray dots:



Similarly, we can use the following code to set the kernel type to histogram intersection:

```
| svm->setKernel(SVM::INTER);
```

Here is the result of the same data segmented with the histogram intersection kernel:



3. How do you calculate the `HOGDescriptor` size for a HOG window size of 128 by 96 pixels? (The rest of the HOG parameters are untouched.)

```
HOGDescriptor hog;
hog.winSize = Size(128, 128);
vector<float> tempDesc;
hog.compute(Mat(hog.winSize, CV_8UC3),
            tempDesc);
int descriptorSize = tempDesc.size();
```

4. How do you update an existing trained `ANN_MLP`, instead of training from scratch?

You do this by setting the `UPDATE_WEIGHTS` flag during the training. Here's an example:

```
| ann->train(trainData, UPDATE_WEIGHTS);
```

5. What is the required command (by using `opencv_createsamples`) you need to use to create a positive samples vector from a single image of a company logo? Assume you want to have 1,000 samples with a width of 24 and a height of 32, and by using default parameters for rotations and inversions.

```
| opencv_createsamples -vec samples.vec -img sign.png -bg bg.txt
| -num 1000 -w 24 -h 32
```

6. What is the command required to train an LBP cascade classifier for the company logo from the previous question?

```
| opencv_traincascade -data classifier -vec samples.vec
| -bg bg.txt -numPos 1000 -numNeg 1000 -w 24 -h 32
| -featureType LBP
```

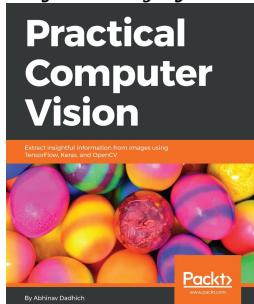
7. What is the default number of stages for training a cascade classifier in

`opencv_traincascade`? How can we change it? What is the downside of increasing and decreasing the number of stages far beyond its default value?

The default number of stages when training a classifier is 20, which is enough for most use cases. You can set it to any other value you want by using the `numStages` parameter. Increasing the number of stages too much can lead to overtraining the classifier and far more time is then required to train it, and vice versa.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

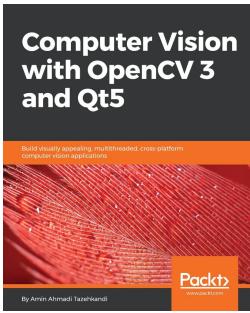


## Practical Computer Vision

Abhinav Dadhich

ISBN: 978-1-78829-768-4

- Learn the basics of image manipulation with OpenCV
- Implement and visualize image filters such as smoothing, dilation, histogram equalization, and more
- Set up various libraries and platforms, such as OpenCV, Keras, and Tensorflow, in order to start using computer vision, along with appropriate datasets for each chapter, such as MSCOCO, MOT, and Fashion-MNIST
- Understand image transformation and downsampling with practical implementations.
- Explore neural networks for computer vision and convolutional neural networks using Keras
- Understand working on deep-learning-based object detection such as Faster-R-CNN, SSD, and more
- Explore deep-learning-based object tracking in action
- Understand Visual SLAM techniques such as ORB-SLAM



## Computer Vision with OpenCV 3 and Qt5

Amin Ahmadi Tazehkandi

ISBN: 978-1-78847-239-5

- Get an introduction to Qt IDE and SDK
- Be introduced to OpenCV and see how to communicate between OpenCV and Qt
- Understand how to create UI using Qt Widgets
- Know to develop cross-platform applications using OpenCV 3 and Qt 5
- Explore the multithreaded application development features of Qt5
- Improve OpenCV 3 application development using Qt5
- Build, test, and deploy Qt and OpenCV apps, either dynamically or statically
- See Computer Vision technologies such as filtering and transformation of images, detecting and matching objects, template matching, object tracking, video and motion analysis, and much more
- Be introduced to QML and Qt Quick for iOS and Android application development

# **Leave a review - let other readers know what you think**

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!