OpenCV 4 Computer Vision Application Programming Cookbook

Fourth Edition

Build complex computer vision applications with OpenCV and C++

David Millán Escrivá and Robert Laganiere

Packt
www.packt.com

Copyright 2019 Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/2/2021 1:41 AM
AN: 2117444 ; David Millán Escrivá, Robert Laganière.; OpenCV 4 Computer Vision Application Programming Cookbook : Build Complex Computer Vision Applications with OpenCV and C++
Account: ns335141

OpenCV 4 Computer Vision Application Programming Cookbook

Fourth Edition

Build complex computer vision applications with OpenCV and C++

David Millán Escrivá
Robert Laganiere

Packt

BIRMINGHAM - MUMBAI

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

OpenCV 4 Computer Vision Application Programming Cookbook
Fourth Edition

Copyright © 2019 Packt Publishing

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

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

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

Commissioning Editor: Pavan Ramchandani
Acquisition Editor: Sandeep Mishra
Content Development Editor: Zeeyan Pinheiro
Technical Editor: Gaurav Gala
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Alishon Mendonsa
Production Coordinator: Nilesh Mohite

First published: May 2011
Second edition: August 2014
Third edition: February 2017
Fourth edition: May 2019

Production reference: 1020519

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

ISBN 978-1-78934-072-3

www.packtpub.com

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

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

Packt.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.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

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

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Contributors

About the authors

David Millán Escrivá was 8 years old when he wrote his first program on an 8086 PC in Basic, which enabled the 2D plotting of basic equations. In 2005, he finished his studies in IT with honors, through the Universitat Politécnica de Valencia, in human-computer interaction supported by computer vision with OpenCV (v0.96). He has worked with Blender, an open source, 3D software project, and on its first commercial movie, *Plumiferos*, as a computer graphics software developer. David has more than 10 years' experience in IT, with experience in computer vision, computer graphics, pattern recognition, and machine learning, working on different projects, and at different start-ups, and companies. He currently works as a researcher in computer vision.

Robert Laganiere is a professor at the University of Ottawa, Canada. He is also a faculty member of the VIVA research lab and is the coauthor of several scientific publications and patents in content-based video analysis, visual surveillance, driver-assistance, object detection, and tracking. He cofounded Visual Cortek, a video analytics start-up, which was later acquired by iWatchLife. He is also a consultant in computer vision and has assumed the role of chief scientist in a number of start-ups companies, including Cognivue Corp, iWatchLife, and Tempo Analytics. Robert has a Bachelor of Electrical Engineering degree from Ecole Polytechnique in Montreal (1987), and M.Sc. and Ph.D. degrees from INRS-Telecommunications, Montreal (1996).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

About the reviewers

Nibedit Dey is a software engineer turned entrepreneur with over 8 years' experience of building complex software-based products. Before starting his entrepreneurial journey, he worked for L&T and Tektronix in different R&D roles. He has reviewed books including *The Modern C++ Challenge*, *Hands-On GUI Programming with C++ and Qt5*, *Getting Started with Qt5* and *Hands-On High Performance Programming with Qt 5* for Packt Publishing.

I would like to thank the online programming communities, bloggers, and my peers from earlier organizations, from whom I have learned a lot over the years.

Christian Stehno studied computer science and got his diploma from Oldenburg University in 2000. Since then, he has worked on different topics in computer science, first as a researcher in theoretical computer science at university. Later on, he switched to embedded system design at a research institute. In 2010, he started his own company, CoSynth, which develops embedded systems and intelligent cameras for industrial automation. In addition to this, he has been a long-time member of the Irrlicht 3D engine developer team.

Packt is searching for authors like you

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

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Table of Contents

Preface 1

Chapter 1: Playing with Images 8
- **Installing the OpenCV library** 8
  - Getting ready 9
  - How to do it... 9
  - How it works... 12
  - There's more... 13
    - Using Qt for OpenCV developments 13
    - The OpenCV developer site 15
  - See also 15
- **Loading, displaying, and saving images** 15
  - Getting ready 15
  - How to do it... 16
  - How it works... 18
  - There's more... 20
    - Clicking on images 20
    - Drawing on images 21
    - Running the example with Qt 22
  - See also 23
- **Exploring the cv::Mat data structure** 23
  - How to do it... 23
  - How it works... 26
  - There's more... 29
    - The input and output arrays 30
  - See also 30
- **Defining regions of interest** 30
  - Getting ready 31
  - How to do it... 31
  - How it works... 32
  - There's more... 33
    - Using image masks 33
  - See also 34

Chapter 2: Manipulating the Pixels 35
- **Accessing pixel values** 36
  - Getting ready 37
  - How to do it... 37
  - How it works... 39
  - There's more... 40
    - The cv::Mat_ template class 41

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also 41
Scanning an image with pointers 41
Getting ready 42
How to do it... 42
How it works... 44
There's more... 45
Other color reduction formulas 45
Having input and output arguments 46
Efficient scanning of continuous images 48
Low-level pointer arithmetics 49
See also 50
Scanning an image with iterators 50
Getting ready 50
How to do it... 51
How it works... 52
There's more... 53
See also 53
Writing efficient image-scanning loops 53
How to do it... 54
How it works... 54
There's more... 56
See also 56
Scanning an image with neighbor access 57
Getting ready 57
How to do it... 57
How it works... 59
There's more... 60
See also 61
Performing simple image arithmetic 62
Getting ready 62
How to do it... 62
How it works... 63
There's more... 64
Overloaded image operators 64
Splitting the image channels 65
Remapping an image 66
How to do it... 66
How it works... 68
See also 69
Chapter 3: Processing Color Images with Classes 70
Comparing colors using the strategy design pattern 71
How to do it... 71
How it works... 73
There's more... 77
Computing the distance between two color vectors 77

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Using OpenCV functions 78
The functor or function object 80
The OpenCV base class for algorithms 81
See also 81
Segmenting an image with the GrabCut algorithm 82
How to do it... 82
How it works... 85
See also 86
Converting color representations 86
Getting ready 87
How to do it... 87
How it works... 89
See also 90
Representing colors with hue, saturation, and brightness 90
How to do it... 91
How it works... 93
There's more... 95
Using colors for detection – skin tone detection 95
Chapter 4: Counting the Pixels with Histograms 99
Computing the image histogram 99
Getting started 100
How to do it... 101
How it works... 105
There's more... 106
Computing histograms of color images 106
See also 108
Applying lookup tables to modify the image's appearance 109
How to do it... 109
How it works... 110
There's more... 110
Stretching a histogram to improve the image contrast 111
Applying a lookup table on color images 113
Equalizing the image histogram 114
How to do it... 114
How it works... 116
Backprojecting a histogram to detect specific image content 116
How to do it... 117
How it works... 119
There's more... 119
Backprojecting color histograms 120
Using the mean shift algorithm to find an object 123
How to do it... 124
How it works... 127
See also 128
Retrieving similar images using histogram comparison 128

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it... 129
How it works... 131
See also 131
Counting pixels with integral images 132
How to do it... 132
How it works... 134
There's more... 135
Adaptive thresholding 135
Visual tracking using histograms 139
See also 143

Chapter 5: Transforming Images with Morphological Operations 144
Eroding and dilating images using morphological filters 145
Getting ready 145
How to do it... 146
How it works... 147
There's more... 149
See also 149
Opening and closing images using morphological filters 150
How to do it... 150
How it works... 152
See also 153
Detecting edges and corners using morphological filters 153
Getting ready 153
How to do it... 154
How it works... 156
See also 158
Segmenting images using watersheds 158
How to do it... 158
How it works... 163
There's more... 164
See also 166
Extracting distinctive regions using MSER 166
How to do it... 166
How it works... 169
See also 172
Extracting foreground objects with the GrabCut algorithm 173
How to do it... 173
How it works... 176
See also 176

Chapter 6: Filtering the Images 177
Filtering images using low-pass filters 178
How to do it... 178
How it works... 180
See also 183

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Downsampling an image 183
How to do it... 183
How it works... 185
There's more... 187
Interpolating pixel values 187
See also 189
Filtering images using a median filter 189
How to do it... 190
How it works... 190
Applying directional filters to detect edges 191
How to do it... 192
How it works... 195
There's more... 198
Gradient operators 199
Gaussian derivatives 200
See also 201
Computing the Laplacian of an image 201
How to do it... 201
How it works... 203
There's more... 207
Enhancing the contrast of an image using the Laplacian 207
Difference of Gaussians 207
See also 208
Chapter 7: Extracting Lines, Contours, and Components 209
Detecting image contours with the Canny operator 209
How to do it... 210
How it works... 211
See also 213
Detecting lines in images with the Hough transform 213
Getting ready 213
How to do it... 214
How it works... 219
There's more... 222
Detecting circles 222
See also 225
Fitting a line to a set of points 225
How to do it... 225
How it works... 228
There's more... 228
Extracting the components' contours 229
How to do it... 229
How it works... 231
There's more... 232
Computing components' shape descriptors 234
How to do it... 234

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works... 236
There's more... 237
Quadrilateral detection 238

Chapter 8: Detecting Interest Points 240
Detecting corners in an image 241
How to do it... 241
How it works... 247
There's more... 249
Good features to track 250
The feature detector's common interface 251
See also 252
Detecting features quickly 253
How to do it... 253
How it works... 254
There's more... 256
Adapted feature detection 256
See also 259
Detecting scale-invariant features 259
How to do it... 259
How it works... 261
There's more... 263
The SIFT feature-detection algorithm 263
See also 265
Detecting FAST features at multiple scales 265
How to do it... 266
How it works... 266
There's more... 268
The ORB feature-detection algorithm 268
See also 269

Chapter 9: Describing and Matching Interest Points 270
Matching local templates 271
How to do it... 271
How it works... 274
There's more... 276
Template matching 276
See also 277
Describing local intensity patterns 278
How to do it... 278
How it works... 281
There's more... 283
Cross-checking matches 283
The ratio test 284
Distance thresholding 285
See also 286
Describing keypoints with binary features 287

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it... 287
How it works... 289
There's more... 290
FREAK 290
See also 292

Chapter 10: Estimating Projective Relations in Images 293
Computing the fundamental matrix of an image pair 296
Getting ready 296
How to do it... 298
How it works... 301
See also 302
Matching images using a random sample consensus 302
How to do it... 303
How it works... 306
There's more... 308
Refining the fundamental matrix 308
Refining the matches 309
Computing a homography between two images 309
Getting ready 310
How to do it... 311
How it works... 313
There's more... 314
Detecting planar targets in an image 314
How to do it... 314
See also 317

Chapter 11: Reconstructing 3D Scenes 318
Digital image formation 319
Calibrating a camera 320
Getting ready 321
How to do it... 322
How it works... 326
There's more... 328
Calibration with known intrinsic parameters 329
Using a grid of circles for calibration 329
See also 329
Recovering the camera pose 330
How to do it... 330
How it works... 333
There's more... 334
cv::Viz – a 3D visualizer module 334
See also 336
Reconstructing a 3D scene from calibrated cameras 336
How to do it... 337
How it works... 343

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

There's more... 345
Decomposing a homography 345
Bundle adjustment 346
See also 346
Computing depth from a stereo image 346
Getting ready 347
How to do it... 348
How it works... 350
See also 351
Chapter 12: Processing Video Sequences 352
Reading video sequences 352
How to do it... 353
How it works... 355
There's more... 356
See also 357
Processing video frames 357
How to do it... 357
How it works... 359
There's more... 363
Processing a sequence of images 363
Using a frame processor class 364
See also 366
Writing video sequences 366
How to do it... 366
How it works... 367
There's more... 370
The codec four-character code 371
See also 372
Extracting the foreground objects in a video 372
How to do it... 374
How it works... 376
There's more... 377
The mixture of Gaussian method 377
See also 379
Chapter 13: Tracking Visual Motion 380
Tracing feature points in a video 381
How to do it... 381
How it works... 386
See also 387
Estimating the optical flow 388
Getting ready 389
How to do it... 390
How it works... 392
See also 394

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Tracking an object in a video 394
How to do it... 395
How it works... 398
See also 402

Chapter 14: Learning from Examples 403
Recognizing faces using the nearest neighbors of local binary patterns 404
How to do it... 404
How it works... 407
See also 410
Finding objects and faces with a cascade of Haar features 411
Getting ready 411
How to do it... 413
How it works... 417
There's more... 420
Face detection with a Haar cascade 420
See also 421
Detecting objects and people using SVMs and histograms of oriented gradients 422
Getting ready 422
How to do it... 423
How it works... 427
There's more... 429
HOG visualization 430
People detection 432
Deep learning and convolutional neural networks (CNNs) 434
See also 435

Chapter 15: OpenCV Advanced Features 436
Face detection using deep learning 436
How to do it... 437
How it works... 441
See also 443
Object detection with YOLOv3 443
How to do it... 443
How it works... 447
See also 448
Enabling Halide to improve efficiency 449
How to do it... 449
How it works... 451
See also 451
OpenCV.js introduction 451
How to do it... 452
How it works... 454

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Other Books You May Enjoy 455

Index 458

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Preface

Augmented reality, driving assistance, video monitoring; more and more applications are now using computer vision and image analysis technologies, and yet we are still in the infancy of the development of new computerized systems capable of understanding our world through the sense of vision. And with the advent of powerful and affordable computing devices and visual sensors, it has never been easier to create sophisticated imaging applications.

A multitude of software tools and libraries manipulating images and videos are available, but for anyone who wishes to develop smart vision-based applications, the OpenCV library is the tool to use. OpenCV is an open source library containing more than 500 optimized algorithms for image and video analysis. Since its introduction in 1999, it has been largely adopted as the primary development tool by the community of researchers and developers in computer vision.

OpenCV was originally developed at Intel by a team led by Gary Bradski as an initiative to advance research in vision and promote the development of rich vision-based, CPU-intensive applications. After a series of beta releases, version 1.0 was launched in 2006. A second major release occurred in 2009 with the launch of OpenCV 2, which proposed important changes, especially the new C++ interface, which we use in this book. In 2012, OpenCV reshaped itself as a non-profit foundation (`https://opencv.org/`) relying on crowdfunding for its future development.

OpenCV 3 was introduced in 2013; changes were made mainly to improve the usability of the library. Its structure has been revised to remove the unnecessary dependencies, large modules have been split into smaller ones, and the API has been refined. This book is the fourth edition of *OpenCV Computer Vision Application Programming Cookbook*, and the first one that covers OpenCV 4. All the programming recipes of the previous editions have been reviewed and updated. We have also added new content and new chapters to provide readers with even better coverage of the essential functionalities of the library.

This book covers many of the library's features and explains how to use them to accomplish specific tasks. Our objective is not to provide detailed coverage of every option offered by the OpenCV functions and classes, but rather to give you the elements you need to build your applications from the ground up. We also explore fundamental concepts in image analysis, and describe some of the important algorithms in computer vision.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This book is an opportunity for you to get introduced to the world of image and video analysis—but this is just the beginning. The good news is that OpenCV continues to evolve and expand. Just consult the OpenCV online documentation at `https://opencv.org/` to stay updated about what the library can do for you. You can also visit the author's website at `http://www.laganiere.name/` for updated information about this cookbook.

Who this book is for

This cookbook is appropriate for novice C++ programmers who want to learn how to use the OpenCV library to build computer vision applications. It is also suitable for professional software developers who want to be introduced to the concepts of computer vision programming. It can be used as a companion book for university-level computer vision courses. It constitutes an excellent reference for graduate students and researchers in image processing and computer vision.

What this book covers

`Chapter 1`, *Playing with Images*, introduces the OpenCV library and shows you how to build simple applications that can read and display images. It also introduces basic OpenCV data structures.

`Chapter 2`, *Manipulating the Pixels*, explains how an image can be read. It describes different methods for scanning an image in order to perform an operation on each of its pixels.

`Chapter 3`, *Processing Color Images with Classes*, consists of recipes presenting various object-oriented design patterns that can help you to build better computer vision applications. It also discusses the concept of colors in images.

`Chapter 4`, *Counting the Pixels with Histograms*, shows you how to compute image histograms and how they can be used to modify an image. Different applications based on histograms are presented that achieve image segmentation, object detection, and image retrieval.

`Chapter 5`, *Transforming Images with Morphological Operations*, explores the concept of mathematical morphology. It presents different operators and how they can be used to detect edges, corners, and segments in images.

`Chapter 6`, *Filtering the Images*, teaches you the principles of frequency analysis and image filtering. It shows how low-pass and high-pass filters can be applied to images, and presents the concept of derivative operators.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

`Chapter 7`, *Extracting Lines, Contours, and Components*, focuses on the detection of geometric image features. It explains how to extract contours, lines, and connected components in an image.

`Chapter 8`, *Detecting Interest Points*, describes various feature point detectors in images.

`Chapter 9`, *Describing and Matching Interest Points*, explains how descriptors of interest points can be computed and used to match points between images.

`Chapter 10`, *Estimating Projective Relations in Images*, explores the projective relations that exist between two images in the same scene. It also describes how to detect specific targets in an image.

`Chapter 11`, *Reconstructing 3D Scenes*, allows you to reconstruct the 3D elements of a scene from multiple images and recover the camera pose. It also includes a description of the camera calibration process.

`Chapter 12`, *Processing Video Sequences*, provides a framework to read and write a video sequence and to process its frames. It also shows you how it is possible to extract foreground objects moving in front of a camera.

`Chapter 13`, *Tracking Visual Motion*, addresses the visual tracking problem. It also shows you how to compute apparent motion in videos, and explains how to track moving objects in an image sequence.

`Chapter 14`, *Learning from Examples*, introduces basic concepts in machine learning. It shows how object classifiers can be built from image samples.

`Chapter 15`, *OpenCV Advanced Features*, covers the most advanced and newest features of OpenCV. This chapter introduces the reader to state-of-the-art deep learning models in artificial intelligence and machine learning. Deep learning is applied to object detection, autonomous cars, and facial recognition. This chapter will introduce you to OpenCV.js, a new binding that ports web technology directly from OpenCV.

To get the most out of this book

This cookbook is based on the C++ API of the OpenCV library. It is therefore assumed that you have some experience with the C++ language. In order to run the examples presented in the recipes and experiment with them, you need a good C++ development environment. Microsoft Visual Studio and Qt are two popular choices.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.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/OpenCV-4-Computer-Vision-Application-Programming-Cookbook-Fourth-Edition`. 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/9781789340723_ColorImages.pdf`.

Conventions used

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

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

`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: "The `cv::Mat` variable's `image` refers to the input image, while `result` refers to the binary output image."

A block of code is set as follows:

```
// compute distance from target color
if (getDistanceToTargetColor(*it)<=maxDist) {
        *itout= 255;
```

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

```
Mat scores = outs[i].row(j).colRange(5, outs[i].cols);
Point classIdPoint;
double confidence;  // Get the value and location of the maximum score
```

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

```
cd llvm_root
```

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: "You then click on **Build Solution** in Visual Studio."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `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`.

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`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

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` with a link to the material.

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

Reviews

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

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

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

1
Playing with Images

This chapter will teach you the basic elements of OpenCV and will show you how to accomplish the most fundamental image-processing tasks—reading, displaying, and saving images. However, before you can start with OpenCV, you need to install the library. This is a simple process that is explained in the first recipe of this chapter.

All your computer vision applications will involve the processing of images. This is why the most fundamental tool that OpenCV offers you is a data structure to handle images and matrices. It is a powerful data structure, with many useful attributes and methods. It also incorporates an advanced memory management model that greatly facilitates the development of applications. The last two recipes of this chapter will teach you how to use this important OpenCV data structure.

In this chapter, we will get you started with the OpenCV library. You will learn how to perform the following tasks:

- Installing the OpenCV library
- Loading, displaying, and saving images
- Exploring the `cv::Mat` data structure
- Defining regions of interest

Installing the OpenCV library

OpenCV is an open source library for developing computer vision applications that run on Windows, Linux, Android, and macOS. It can be used in both academic and commercial applications under a BSD license that allows you to use, distribute, and adapt it freely. This recipe will show you how to install the library on your machine.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

When you visit the OpenCV official website at `https://opencv.org/`, you will find the latest release of the library, the online documentation, and many other useful resources concerning OpenCV.

How to do it...

The following steps will help take us through the installation, as follows:

1. From the OpenCV website, go to the downloads page that corresponds to the platform of your choice (Unix/Windows or Android). From there, you will be able to download the OpenCV package.
2. You will then uncompress it, normally under a directory with a name that corresponds to the library version (for example, in Windows, you can save the uncompressed directory under `C:\OpenCV4.0.0`).

   Once this is done, you will find a collection of files and directories that constitute the library at the chosen location. Notably, you will find the `modules` directory here, which contains all the source files. (Yes, it is open source!)

3. However, in order to complete the installation of the library and have it ready for use, you need to undertake an additional step—generating the binary files of the library for the environment of your choice. This is indeed the point where you have to make a decision on the target platform that you will use to create your OpenCV applications. Which operating system should you use? Windows or Linux? Which compiler should you use? Microsoft Visual Studio 2013 or MinGW? 32-bit or 64-bit? The **integrated development environment** (**IDE**) that you will use in your project development will also guide you to make these choices.

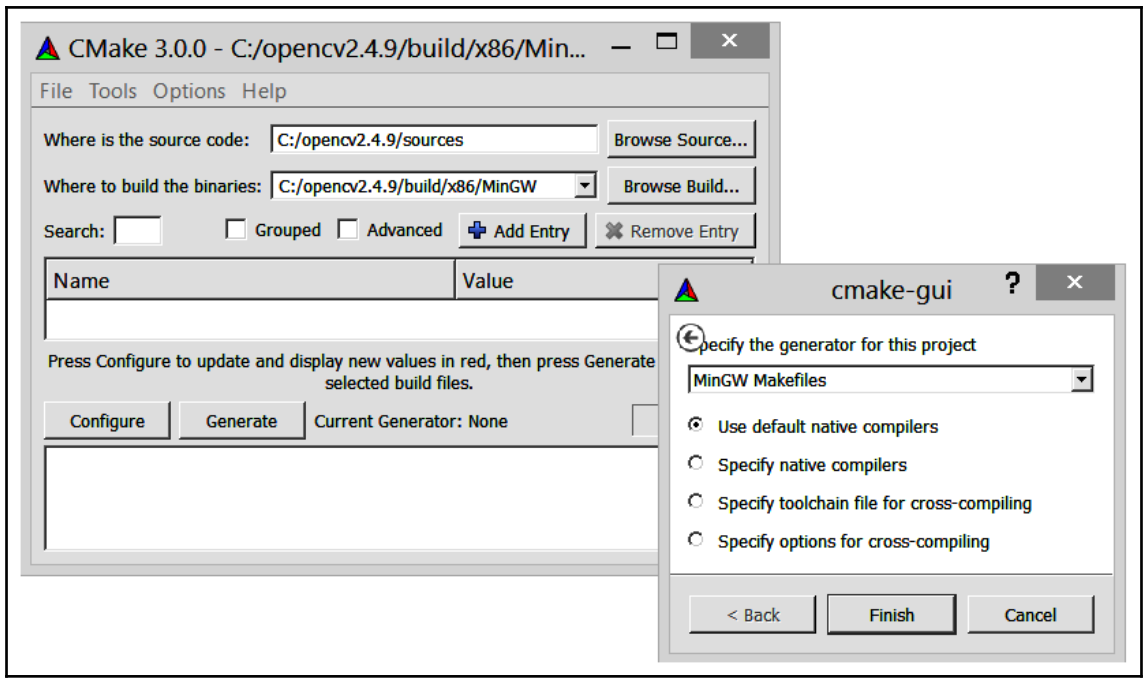
Note that if you are working under Windows with Visual Studio, the executable installation package will, most probably, not only install the library sources, but also install all of the precompiled binaries needed to build your applications. Check for the `build` directory; it should contain the `x64` and `x86` subdirectories (corresponding to the 64-bit and 32-bit versions). Within these subdirectories, you should find directories such as `vc14` and `vc15`; these contain the binaries for the different versions of Microsoft Visual Studio. In that case, you are ready to start using OpenCV. Therefore, you can skip the compilation step described in this recipe, unless you want a customized build with specific options.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. To complete the installation process and build the OpenCV binaries, you need to use the **CMake** tool, available at `https://cmake.org/`.

   CMake is another open source software tool designed to control the compilation process of a software system using platform-independent configuration files. It generates the required **makefiles** or **workspaces** needed for compiling a software library in your environment. Therefore, you need to download and install CMake.

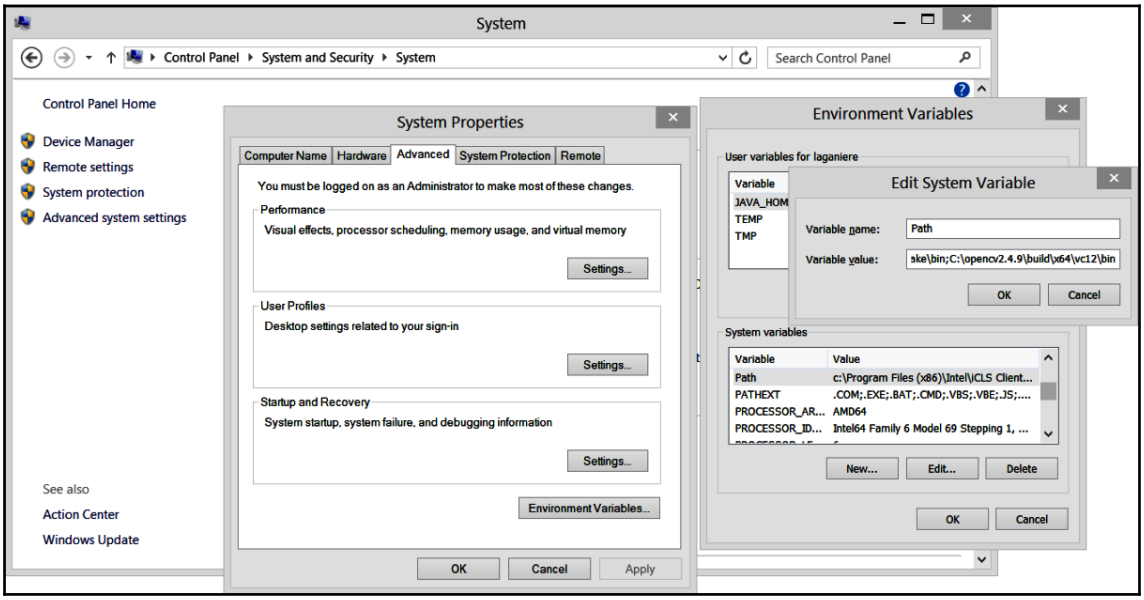
5. Then, run it using the command line. Thereafter, it is easier to use CMake with its GUI (`cmake-gui`).
6. Specify the folder containing the OpenCV library source and the one that will contain the binaries. You need to click on **Configure** in order to select the compiler of your choice, and then click on **Configure** again as shown in the following screenshot:



7. You are now ready to generate your project files by clicking on the **Generate** button. These files will allow you to compile the library.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. This is the last step of the installation process, which will make the library ready to be used under your development environment:
    1. If you have selected Visual Studio, then all you need to do is to open the top-level solution file that CMake has created for you (most probably, the `OpenCV.sln` file).
    2. You then click on **Build Solution** in Visual Studio.
    3. To get both a **Release** and a **Debug** build, you will have to repeat the compilation process twice, one for each configuration.
    The `bin` directory that is created contains the dynamic library files that your executable will call at runtime.
    4. Make sure to set your system `PATH` environment variable from the **Control Panel** such that your operating system can find the `dll` files when you run your applications:



9. In Linux environments, you will use the generated makefiles by running your `make` utility command. To complete the installation of all the directories, you also have to run a **Build INSTALL** or `sudo make INSTALL` command.

If you wish to use Qt as your IDE, the *There's more...* section of this recipe describes an alternative way to compile the OpenCV project.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Since Version 2.2, the OpenCV library has been divided into several modules. These modules are built-in library files located in the `lib` directory. Some of the commonly used modules are as follows:

- The `opencv_core` module that contains the core functionalities of the library, in particular, basic data structures and arithmetic functions
- The `opencv_imgproc` module that contains the main image-processing functions
- The `opencv_highgui` module that contains the image and video reading and writing functions along with some user interface functions
- The `opencv_features2d` module that contains the feature point detectors and descriptors and the feature point matching framework
- The `opencv_calib3d` module that contains the camera calibration, two-view geometry estimation, and stereo functions
- The `opencv_video` module that contains the motion estimation, feature tracking, and foreground extraction functions and classes
- The `opencv_objdetect` module that contains the object detection functions such as the face and people detectors

The library also includes other utility modules that contain machine learning functions (`opencv_ml`), computational geometry algorithms (`opencv_flann`), contributed code (`opencv_contrib`), and many more. You will also find other specialized libraries that implement higher level functions, such as `opencv_photo` for computational photography and `opencv_stitching` for image-stitching algorithms. There is also a new branch that contains other library modules, which include non-free algorithms, non-stable modules, or experimental modules. This branch is on the `opencv-contrib` GitHub branch. When you compile your application, you will have to link your program with the libraries that contain the OpenCV functions you are using, linking it with the `opencv-contrib` folder.

All these modules have a header file associated with them (located in the `include` directory). A typical OpenCV C++ code will, therefore, start by including the required modules. For example (and this is the suggested declaration style), it will look like the following code:

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

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

You might see an OpenCV code starting with the following command:

```
#include "cv.h"
```

This is because it used the old style before the library was restructured into modules and became compatible with older definitions.

There's more...

The OpenCV website at `https://opencv.org/` contains detailed instructions on how to install the library. It also contains complete online documentation that includes several tutorials on the different components of the library.

Using Qt for OpenCV developments

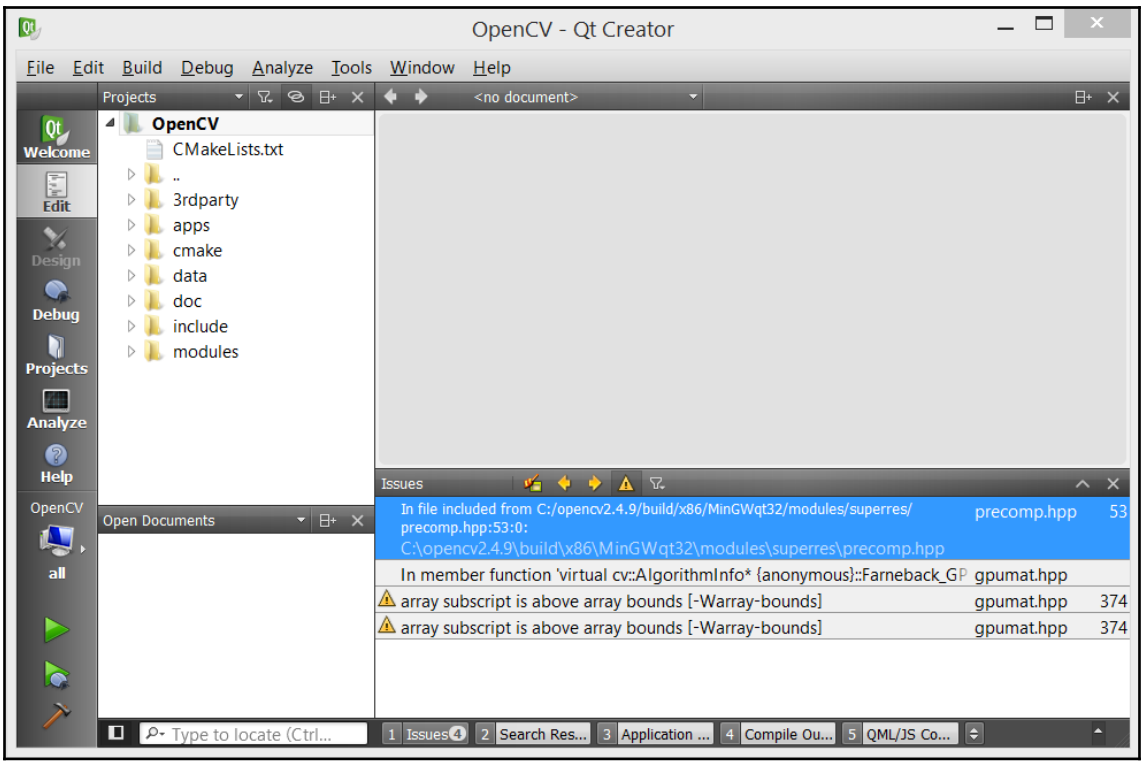
Qt is a cross-platform IDE for C++ applications developed as an open source project. It is offered under the **GNU Lesser General Public License** (**LGPL**) open source license as well as under a commercial (and paid) license for the development of proprietary projects. It is composed of two separate elements—a cross-platform IDE called Qt Creator, and a set of Qt class libraries and development tools. Using Qt to develop C++ applications has the following benefits:

- It is an open source initiative, developed by the Qt community, that gives you access to the source code of the different Qt components
- It is a cross-platform IDE, meaning that you can develop applications that can run on different operating systems, such as Windows, Linux, macOS, and so on
- It includes a complete and cross-platform GUI library that follows an effective object-oriented and event-driven model
- Qt also includes several cross-platform libraries that help you to develop multimedia, graphics, databases, multithreading, web applications, and many other interesting building blocks useful for designing advanced applications

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can download Qt from `https://www.qt.io/developers/`. When you install it, you will be offered the choice of different compilers. Under Windows, MinGW is an excellent alternative to the Visual Studio compilers.

Compiling the OpenCV library with Qt is particularly easy because it can read CMake files. Once OpenCV and CMake have been installed, simply select **Open File** or **Project...** from the Qt menu, and open the `CMakeLists.txt` file that you will find under the `sources` directory of OpenCV. This will create an OpenCV project that you will have built by clicking on **Build Project** in the Qt menu:



You might get a few warnings, but these can be overlooked without consequences.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The OpenCV developer site

OpenCV is an open source project that welcomes user contributions. You can access the developer site at `https://docs.opencv.org/`. Among other things, you can access the currently developed version of OpenCV. The community uses Git as its version control system. You then have to use it to check out the latest version of OpenCV. Git is also a free and open source software system; it is probably the best tool you can use to manage your own source code. You can download it from `https://git-scm.com/`.

See also

- The website of the author of this cookbook (`www.laganiere.name`) also presents step-by-step instructions on how to install the latest versions of the library.
- The *There's more...* section of the next recipe explains how to create an OpenCV project with Qt.

We've successfully learned how to install the OpenCV library. Now, let's move on to the next recipe!

Loading, displaying, and saving images

It is now time to run your first OpenCV application. Since OpenCV is about processing images, this task will show you how to perform the most fundamental operations needed in the development of imaging applications. These are loading an input image from a file, displaying an image on a window, applying a processing function, and storing an output image on a disk.

Getting ready

Using your favorite IDE (for example, MS Visual Studio or Qt), create a new console application with the `main` function that is ready to be filled.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Let's take a look at the following steps:

1. Include the header files, declaring the classes and functions you will use. Here, we simply want to display an image, so we need the `core` library that declares the image data structure and the `highgui` header file that contains all the graphical interface functions:

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```

2. Our `main` function starts by declaring a variable that will hold the image. Under `OpenCV2`, define an object of the `cv::Mat` class:

```
cv::Mat image; // create an empty image
```

3. This definition creates an image sized *0 x 0*. This can be confirmed by accessing the `cv::Mat` size attributes:

```
std::cout << "This image is " << image.rows << " x "  << image.cols
<< std::endl;
```

4. Next, a simple call to the reading function will read an image from the file, decode it, and allocate the memory:

```
image= cv::imread("puppy.bmp"); // read an input image
```

5. You are now ready to use this image. However, you should first check whether the image has been correctly read (an error will occur if the file is not found, if the file is corrupted, or if it is not in a recognizable format) using the `empty()` function. The `empty` method returns `true` if no image data has been allocated:

```
if (image.empty()) {  // error handling
  // no image has been created...
  // possibly display an error message
  // and quit the application
  ...
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. The first thing you might want to do with this image is to display it. You can do this by using the functions of the `highgui` module. Start by declaring the window on which you want to display the images, and then specify the image to be shown on this special window:

```
// define the window (optional)
cv::namedWindow("Original Image");
// show the image
cv::imshow("Original Image", image);
```

As you can see, the window is identified by a name. You can reuse this window to display another image later, or you can create multiple windows with different names. When you run this application, you will see an image window as follows:

7. Now, you would normally apply some processing to the image. OpenCV offers a wide selection of processing functions, and several of them are explored in this book. Let's start with a very simple one that flips an image horizontally. Several image transformations in OpenCV can be performed **in-place**, meaning that the transformation is applied directly on the input image (no new image is created). This is the case with the flipping method. However, we can always create another matrix to hold the output result, and that is what we will do:

```
cv::Mat result; // we create another empty image
cv::flip(image,result,1); // positive for horizontal
                          // 0 for vertical,
                          // negative for both
```

8. We are going to display the result on another window:

```
cv::namedWindow("Output Image"); // the output window
cv::imshow("Output Image", result);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Since it is a console window that will terminate when it reaches the end of the `main` function, we add an extra `highgui` function to wait for a user keypress before ending the program:

```
cv::waitKey(0); // 0 to indefinitely wait for a key pressed
                // specifying a positive value will wait for
                // the given amount of msec
```

You can then see that the output image is displayed on a distinct window, as shown in the following screenshot:

10. Finally, you will probably want to save the processed image on your disk. This is done using the following `highgui` function:

```
cv::imwrite("output.bmp", result); // save result
```

The file extension determines which codec will be used to save the image. Other popular supported image formats are JPG, TIFF, and PNG.

How it works...

All classes and functions in the C++ API of OpenCV are defined within the `cv` namespace. You have two ways to access them. First, precede the `main` function's definition with the following declaration:

```
using namespace cv;
```

Alternatively, prefix all OpenCV class and function names by the namespace specification, that is, `cv::`, as we will do in this book. The use of the prefix makes the OpenCV classes and functions easier to identify.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `highgui` module contains a set of functions that allows you to visualize and interact with your images easily. When you load an image with the `imread` function, you also have the option to read it as a gray-level image. This is very advantageous since several computer vision algorithms require gray-level images. Converting an input color image on the fly as you read it will save your time and minimize your memory usage. This can be done as follows:

```
// read the input image as a gray-scale image
image= cv::imread("puppy.bmp", cv::IMREAD_GRAYSCALE);
```

This will produce an image made of `unsigned bytes` (`unsigned char` in C++) that OpenCV designates with the `CV_8U` defined constant. Alternatively, it is sometimes necessary to read an image as a three-channel color image even if it has been saved as a gray-level image. This can be achieved by calling the `imread` function with a positive second argument:

```
// read the input image as a 3-channel color image
image= cv::imread("puppy.bmp", cv::IMREAD_COLOR);
```

This time, an image made of three bytes per pixel will be created, designated as `CV_8UC3` in OpenCV. Of course, if your input image has been saved as a gray-level image, all three channels will contain the same value. Finally, if you wish to read the image in the format in which it has been saved, then simply input a negative value as the second argument. The number of channels in an image can be checked by using the `channels` method:

```
std::cout << "This image has " << image.channels() << " channel(s)";
```

Pay attention when you open an image with `imread` without specifying a full path (as we did here). In that case, the default directory will be used. When you run your application from the console, this directory is obviously one of your executable files. However, if you run the application directly from your IDE, the default directory will most often be the one that contains your project file. Consequently, make sure that your input image file is located in the right directory.

When you use `imshow` to display an image made up of integers (designated as `CV_16U` for 16-bit unsigned integers, or as `CV_32S` for 32-bit signed integers), the pixel values of this image will be divided by 256 first, in an attempt to make it displayable with 256 gray shades. Similarly, an image made of floating points will be displayed by assuming a range of possible values between `0.0` (displayed as black) and `1.0` (displayed as white). Values outside this defined range are displayed in white (for values above `1.0`) or black (for values below `1.0`).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `highgui` module is very useful for building quick prototypal applications. When you are ready to produce a finalized version of your application, you will probably want to use the GUI module offered by your IDE in order to build an application with a more professional look.

Here, our application uses both input and output images. As an exercise, you should rewrite this simple program such that it takes advantage of the function's in-place processing, that is, by not declaring the output image and writing it instead:

```
cv::flip(image,image,1); // in-place processing
```

There's more...

The `highgui` module contains a rich set of functions that help you to interact with your images. Using these, your applications can react to mouse or key events. You can also draw shapes and write texts on images.

Clicking on images

You can program your mouse to perform specific operations when it is over one of the image windows you created. This is done by defining an appropriate **callback** function. A callback function is a function that you do not explicitly call but which is called by your application in response to specific events (here, the events that concern the mouse interacting with an image window). To be recognized by applications, callback functions need to have a specific signature and must be registered. In the case of the mouse event handler, the callback function must have the following signature:

```
void onMouse( int event, int x, int y, int flags, void* param);
```

The first parameter is an integer that is used to specify which type of mouse event has triggered the call to the callback function. The other two parameters are simply the pixel coordinates of the mouse location when the event occurred. The flags are used to determine which button was pressed when the mouse event was triggered. Finally, the last parameter is used to send an extra parameter to the function in the form of a pointer to an object. This callback function can be registered in the application through the following call:

```
cv::setMouseCallback("Original Image", onMouse,
reinterpret_cast<void*>(&image));
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this example, the `onMouse` function is associated with the image window called `Original Image`, and the address of the displayed image is passed as an extra parameter to the function. Now, if we define the `onMouse` callback function as shown in the following code, then each time the mouse is clicked, the value of the corresponding pixel will be displayed on the console (here, we assume that it is a gray-level image):

```
void onMouse( int event, int x, int y, int flags, void* param)  {

  cv::Mat *im= reinterpret_cast<cv::Mat*>(param);
  switch (event) {  // dispatch the event
  case cv::EVENT_LBUTTONDOWN: // left mouse button down event
        // display pixel value at (x,y)
        std::cout << "at (" << x << "," << y << ") value is: " <<
        static_cast<int>(im->at<uchar>(cv::Point(x,y))) << std::endl;
        break;
    }
}
```

Note that in order to obtain the pixel value at `(x,y)`, we used the `at` method of the `cv::Mat` object here; this is discussed in `Chapter 2`, *Manipulating the Pixels*. Other possible events that can be received by the mouse event callback function include `cv::EVENT_MOUSE_MOVE`, `cv::EVENT_LBUTTONUP`, `cv::EVENT_RBUTTONDOWN`, and `cv::EVENT_RBUTTONUP`.

Drawing on images

OpenCV also offers a few functions to draw shapes and write texts on images. The examples of basic shape-drawing functions are `circle`, `ellipse`, `line`, and `rectangle`. The following is an example of how to use the `circle` function:

```
cv::circle(image,  // destination image
           cv::Point(155,110), // center coordinate
           65,                 // radius
           0,                  // color (here black)
           3);                 // thickness
```

The `cv::Point` structure is often used in OpenCV methods and functions to specify a pixel coordinate. Note that here we assume that the drawing is done on a gray-level image; this is why the color is specified with a single integer. In the next recipe, you will learn how to specify a color value in the case of color images that use the `cv::Scalar` structure. It is also possible to write text on an image. This can be done as follows:

```
cv::putText(image,  // destination image
           "This is a dog.",// text
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
cv::Point(40,200), // text position
cv::FONT_HERSHEY_PLAIN,  // font type
2.0, // font scale
255, // text color (here white)
2);  // text thickness
```

Calling these two functions on our test image will then result in the following screenshot:

Let's see what happens when you run the example using Qt.

Running the example with Qt

If you wish to use Qt to run your OpenCV applications, you will need to create project files. For the example of this recipe, here is how the project file (`loadDisplaySave.pro`) will look:

```
QT += core
QT -= gui

TARGET = loadDisplaySave
CONFIG  += console
CONFIG  -= app_bundle

TEMPLATE = app

SOURCES += loadDisplaySave.cpp
INCLUDEPATH += C:\OpenCV4.0.0\build\include
LIBS += -LC:\OpenCV4.0.0\build\x86\MinGWqt32\lib \
-lopencv_core400 \
-lopencv_imgproc400 \
-lopencv_highgui400
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This file shows you where to find the `include` and library files. It also lists the library modules that are used by the example. Make sure to use the library binaries compatible with the compiler that Qt is using. Note that if you download the source code of the examples for this book, you will find the `CMakeLists` files that you can open with Qt (or CMake) in order to create the associated projects.

See also

- The `cv::Mat` class is the data structure that is used to hold your images (and obviously, other matrix data). This data structure is at the core of all OpenCV classes and functions; the next recipe offers a detailed explanation of this data structure.
- You can download the source code of the examples for this book from `https://github.com/PacktPublishing/OpenCV-4-Computer-Vision-Application-Programming-Cookbook-Fourth-Edition`.

We've successfully learned how to load, display, and save images. Now, let's move on to the next recipe!

Exploring the cv::Mat data structure

In the previous recipe, you were introduced to the `cv::Mat` data structure. As mentioned, this is a key element of the library. It is used to manipulate images and matrices (in fact, an image is a matrix from a computational and mathematical point of view). Since you will be using this data structure extensively in your application developments, it is imperative that you become familiar with it. Notably, you will learn in this recipe that this data structure incorporates an elegant memory management mechanism, allowing efficient usage.

How to do it...

Let's write the following test program that will allow us to test the different properties of the `cv::Mat` data structure, as follows:

1. Include the `opencv` headers and a `c++ i/o stream` utility:

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. We are going to create a function that generates a new gray image with a default value for all its pixels:

```
cv::Mat function() {
  // create image
  cv::Mat ima(500,500,CV_8U,50);
  // return it
  return ima;
}
```

3. In the `main` function, we are going to create six windows to show our results:

```
// define image windows
  cv::namedWindow("Image 1");
  cv::namedWindow("Image 2");
  cv::namedWindow("Image 3");
  cv::namedWindow("Image 4");
  cv::namedWindow("Image 5");
  cv::namedWindow("Image");
```

4. Now, we can start to create different mats (with different sizes, channels, and default values) and wait for the key to be pressed:

```
// create a new image made of 240 rows and 320 columns
  cv::Mat image1(240,320,CV_8U,100);

  cv::imshow("Image", image1); // show the image
  cv::waitKey(0); // wait for a key pressed

  // re-allocate a new image
  image1.create(200,200,CV_8U);
  image1= 200;

  cv::imshow("Image", image1); // show the image
  cv::waitKey(0); // wait for a key pressed

  // create a red color image
  // channel order is BGR
  cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));

  // or:
  // cv::Mat image2(cv::Size(320,240),CV_8UC3);
  // image2= cv::Scalar(0,0,255);

  cv::imshow("Image", image2); // show the image
  cv::waitKey(0); // wait for a key pressed
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. We are going to read an image with the `imread` function and copy it to another mat:

```
// read an image
  cv::Mat image3= cv::imread("puppy.bmp");

  // all these images point to the same data block
  cv::Mat image4(image3);
  image1= image3;

  // these images are new copies of the source image
  image3.copyTo(image2);
  cv::Mat image5= image3.clone();
```

6. Now, we are going to apply an image transformation (`flip`) to a copied image, show all images created, and wait for a keypress:

```
// transform the image for testing
  cv::flip(image3,image3,1);

  // check which images have been affected by the processing
  cv::imshow("Image 3", image3);
  cv::imshow("Image 1", image1);
  cv::imshow("Image 2", image2);
  cv::imshow("Image 4", image4);
  cv::imshow("Image 5", image5);
  cv::waitKey(0); // wait for a key pressed
```

7. Now, we are going to use the function created before to generate a new gray mat:

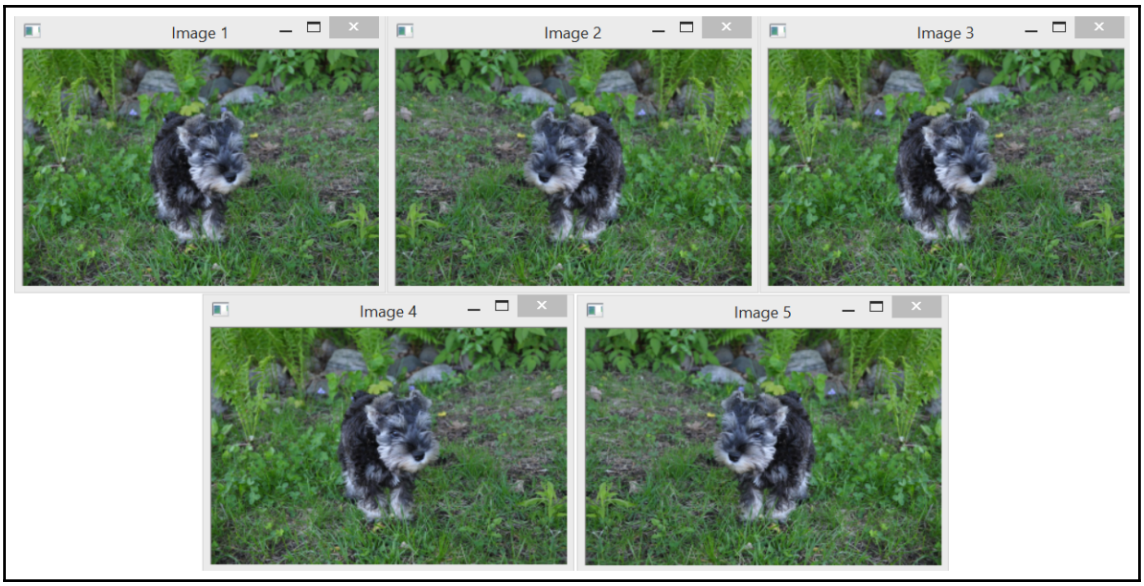
```
// get a gray-level image from a function
  cv::Mat gray= function();

  cv::imshow("Image", gray); // show the image
  cv::waitKey(0); // wait for a key pressed
```

8. Finally, we are going to load a color image but convert it to gray in the loading process. Then, we will convert its values to float mat:

```
  // read the image in gray scale
  image1= cv::imread("puppy.bmp", IMREAD_GRAYSCALE);
  image1.convertTo(image2,CV_32F,1/255.0,0.0);

  cv::imshow("Image", image2); // show the image
  cv::waitKey(0); // wait for a key pressed
```

Run this program and take a look at the following images produced:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Now, let's go behind the scenes to understand the code better.

How it works...

The `cv::Mat` data structure is essentially made up of two parts: a header and a data block. The header contains all the information associated with the matrix (size, number of channels, data type, and so on). The previous recipe showed you how to access some of the attributes of this structure contained in its header (for example, by using `cols`, `rows`, or `channels`). The data block holds all the pixel values of an image. The header contains a pointer variable that points to this data block; it is the `data` attribute. An important property of the `cv::Mat` data structure is the fact that the memory block is only copied when it is explicitly requested. Indeed, most operations will simply copy the `cv::Mat` header such that multiple objects will point to the same data block at the same time. This memory management model makes your applications more efficient while avoiding memory leaks, but its consequences have to be understood. The examples for this recipe illustrate this fact.

By default, the `cv::Mat` objects have a zero size when they are created, but you can also specify an initial size as follows:

```
// create a new image made of 240 rows and 320 columns
cv::Mat image1(240,320,CV_8U,100);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this case, you also need to specify the type of each matrix element; `CV_8U` here, which corresponds to 1-byte pixel images. The letter `U` means it is unsigned. You can also declare signed numbers by using the letter `S`. For a color image, you would specify three channels (`CV_8UC3`). You can also declare integers (signed or unsigned) of size 16 and 32 (for example, `CV_16SC3`). You also have access to 32-bit and 64-bit floating-point numbers (for example, `CV_32F`).

Each element of an image (or a matrix) can be composed of more than one value (for example, the three channels of a color image); therefore, OpenCV has introduced a simple data structure that is used when pixel values are passed to functions. It is the `cv::Scalar` structure, which is generally used to hold one value or three values. For example, to create a color image initialized with red pixels, you will write the following code:

```
// create a red color image
// channel order is BGR
cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));
```

Similarly, the initialization of the gray-level image could also have been done using this structure by writing `cv::Scalar(100)`.

The image size also often needs to be passed to functions. We have already mentioned that the `cols` and `rows` attributes can be used to get the dimensions of a `cv::Mat` instance. The size information can also be provided through the `cv::Size` structure that simply contains the height and width of the matrix. The `size()` method allows you to obtain the current matrix size. It is the format that is used in many methods where a matrix size must be specified. For example, an image could be created as follows:

```
// create a non-initialized color image
cv::Mat image2(cv::Size(320,240),CV_8UC3);
```

The data block of an image can always be allocated or reallocated using the `create` method. When an image has been previously allocated, its old content is deallocated first. For reasons of efficiency, if the newly proposed size and type match the already existing size and type, then no new memory allocation is performed:

```
// re-allocate a new image
// (only if size or type are different)
image1.create(200,200,CV_8U);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

When no more references point to a given `cv::Mat` object, the allocated memory is automatically released. This is very convenient because it avoids the common memory leak problems often associated with dynamic memory allocation in C++. This is a key mechanism in OpenCV 2 that is accomplished by having the `cv::Mat` class implement reference counting and shallow copying. Therefore, when an image is assigned to another one, the image data (that is, the pixels) is not copied; both the images will point to the same memory block. This also applies to images passed by value or returned by value. A reference count is kept, such that the memory will be released only when all the references to the image will be destroyed or assigned to another image:

```
// all these images point to the same data block
cv::Mat image4(image3);
image1= image3;
```

Any transformation applied to one of the preceding images will also affect the other images. If you wish to create a deep copy of the content of an image, use the `copyTo` method. In that case, the `create` method is called on the destination image. Another method that produces a copy of an image is the `clone` method, which creates an identical new image as follows:

```
// these images are new copies of the source image
image3.copyTo(image2);
cv::Mat image5= image3.clone();
```

If you need to copy an image into another image that does not necessarily have the same data type, you have to use the `convertTo` method:

```
// convert the image into a floating point image [0,1]
image1.convertTo(image2,CV_32F,1/255.0,0.0);
```

In this example, the source image is copied into a floating-point image. The method includes two optional parameters—a scaling factor and an offset. Note that both the images must, however, have the same number of channels.

The allocation model for the `cv::Mat` objects also allows you safely to write functions (or class methods) that return an image:

```
cv::Mat function() {
  // create image
  cv::Mat ima(240,320,CV_8U,cv::Scalar(100));
  // return it
  return ima;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We also call this function from our `main` function, as follows:

```
// get a gray-level image
cv::Mat gray= function();
```

If we do this, then the `gray` variable will now hold the image created by the function without extra memory allocation. Indeed, as we explained, only a shallow copy of the image will be transferred from the returned `cv::Mat` instance to the `gray` image. When the `ima` local variable goes out of scope, this variable is deallocated, but since the associated reference counter indicates that its internal image data is being referred to by another instance (that is, the `gray` variable), its memory block is not released.

It's worth noting that in the case of classes, you should be careful and not return image `class` attributes. Here is an example of an error-prone implementation:

```
class Test {
  // image attribute
  cv::Mat ima;
  public:
  // constructor creating a gray-level image
 Test() : ima(240,320,CV_8U,cv::Scalar(100)) {}

  // method return a class attribute, not a good idea...
  cv::Mat method() { return ima; }
};
```

Here, if a function calls the method of this class, it obtains a shallow copy of the image attributes. If later this copy is modified, the `class` attribute will also be surreptitiously modified, which can affect the subsequent behavior of the class (and vice versa). To avoid these kinds of errors, you should instead return a clone of the attribute.

There's more...

While you are manipulating the `cv::Mat` class, you will discover that OpenCV also includes several other related classes. It will be important for you to become familiar with them.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The input and output arrays

If you look at the OpenCV documentation, you will see that many methods and functions accept parameters of the `cv::InputArray` type as the input. This type is a simple proxy class introduced to generalize the concept of arrays in OpenCV, and thus, avoid the duplication of several versions of the same method or function with different input parameter types. It basically means that you can supply a `cv::Mat` object or other compatible types as an argument. This class is just an interface, so you should never declare it explicitly in your code. It is interesting to know that `cv::InputArray` can also be constructed from the popular `std::vector` class. This means that such objects can be used as the input to OpenCV methods and functions (as long as it makes sense to do so). Other compatible types are `cv::Scalar` and `cv::Vec`; this later structure will be presented in `Chapter 2`, *Manipulating the Pixels*. There is also a `cv::OutputArray` proxy class that is used to designate the arrays returned by some methods or functions.

See also

- The complete OpenCV documentation can be found at `https://docs.opencv.org/`.
- `Chapter 2`, *Manipulating the Pixels*, will show you how to access and modify the pixel values of an image represented by the `cv::Mat` class efficiently.

The next recipe will explain how to define a **region of interest** (**ROI**) inside an image.

Defining regions of interest

Sometimes, a processing function needs to be applied only to a portion of an image. OpenCV incorporates an elegant and simple mechanism to define a subregion in an image and manipulate it as a regular image. This recipe will teach you how to define an ROI inside an image.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

Suppose we want to copy a small image onto a larger one. For example, let's say we want to insert the following small logo into our test image:

To do this, an ROI can be defined over which the copy operation can be applied. As we will see, the position of the ROI will determine where the logo will be inserted in the image.

How to do it...

Let's take a look at the following steps:

1. The first step consists of defining the ROI. We can use `Rect` to define the ROI:

```
cv::Rect myRoi= cv::Rect(image.cols-logo.cols, //ROI coordinates
                image.rows-logo.rows,
                logo.cols,logo.rows)
```

2. Once the ROI is defined, we can create a new mat applying the ROI to another mat and it can be manipulated as a regular `cv::Mat` instance. The key is that the ROI is indeed a `cv::Mat` object that points to the same data buffer as its parent image and has a header that specifies the coordinates of the ROI. Inserting the logo would then be accomplished as follows:

```
// define image ROI at image bottom-right
cv::Mat imageROI(image, myRoi);

// insert logo
logo.copyTo(imageROI);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Here, `image` is the destination image, and `logo` is the logo image (of a smaller size). The following image is then obtained by executing the previous code:

Now, let's go behind the scenes to understand the code better.

How it works...

One way to define an ROI is to use a `cv::Rect` instance. As the name indicates, it describes a rectangular region by specifying the position of the upper-left corner (the first two parameters of the constructor) and the size of the rectangle (the width and height are given in the last two parameters). In our example, we used the size of the image and the size of the logo in order to determine the position where the logo would cover the bottom-right corner of the image. Obviously, the ROI should always be completely inside the parent image.

The ROI can also be described using row and column ranges. A range is a continuous sequence from a start index to an end index (excluding both). The `cv::Range` structure is used to represent this concept. Therefore, an ROI can be defined from two ranges; in our example, the ROI could have been equivalently defined as follows:

```
imageROI= image(cv::Range(image.rows-logo.rows,image.rows),
                cv::Range(image.cols-logo.cols,image.cols));
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this case, the `operator()` function of `cv ::Mat` returns another `cv::Mat` instance that can then be used in subsequent calls. Any transformation of the ROI will affect the original image in the corresponding area because the image and the ROI share the same image data. Since the definition of an ROI does not include the copying of data, it is executed in a constant amount of time, no matter the size of the ROI.

If one wants to define an ROI made of some lines of an image, the following call could be used:

```
cv::Mat imageROI= image.rowRange(start,end);
```

Similarly, for an ROI made of some image columns, the following could be used:

```
cv::Mat imageROI= image.colRange(start,end);
```

There's more...

The OpenCV methods and functions include many optional parameters that are not discussed in the recipes of this book. When you wish to use a function for the first time, you should always take the time to look at the documentation to learn more about the possible options that this function offers. One very common option is the possibility to define image masks.

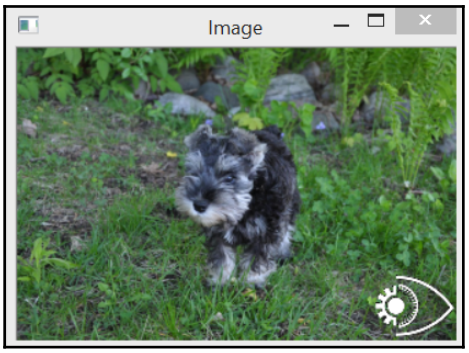
Using image masks

Some OpenCV operations allow you to define a mask that will limit the applicability of a given function or method, which is normally supposed to operate on all the image pixels. A mask is an 8-bit image that should be nonzero at all locations where you want an operation to be applied. At the pixel locations that correspond to the zero values of the mask, the image is untouched. For example, the `copyTo` method can be called with a mask. We can use it here to copy only the white portion of the logo shown previously, as follows:

```
// define image ROI at image bottom-right
imageROI= image(cv::Rect(image.cols-logo.cols,image.rows-logo.rows,
logo.cols,logo.rows));
// use the logo as a mask (must be gray-level)
cv::Mat mask(logo);

// insert by copying only at locations of non-zero mask
logo.copyTo(imageROI,mask);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The following image is obtained by executing the previous code:



The background of our logo was black (therefore, it had the value `0`); therefore, it was easy to use it as both the copied image and the mask. Of course, you can define the mask of your choice in your application; most OpenCV pixel-based operations give you the opportunity to use masks.

See also

- The `row` and `col` methods that will be used in the *Scanning an image with neighbor access* recipe of `Chapter 2`, *Manipulating the Pixels*. These are special cases of the `rowRange` and `colRange` methods in which the start and end indexes are equal in order to define a single-line or single-column ROI.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2
Manipulating the Pixels

In order to build computer vision applications, you need to be able to access the image content and, eventually, modify or create images. This chapter will teach you how to manipulate the picture elements (also known as **pixels**). You will learn how to scan an image and process each of its pixels. You will also learn how to do this efficiently, since even images of modest dimensions can contain hundreds of thousands of pixels.

Fundamentally, an image is a matrix of numerical values. This is why, as we learned in `Chapter 1`, *Playing with Images*, OpenCV 4 manipulates them using the `cv::Mat` data structure. Each element of the matrix represents one pixel. For a gray-level image (a black-and-white image), pixels are unsigned 8-bit values where `0` corresponds to black and `255` corresponds to white. In the case of color images, three primary color values are required in order to reproduce the different visible colors. This is a consequence of the fact that our human visual system is **trichromatic**; three types of cone cells on our retinas convey color information to our brains. This means that for a color image, three values must be associated with each pixel. In photography and digital imaging, the commonly used primary color channels are red, green, and blue. A matrix element is, therefore, made up of a triplet of 8-bit values in this case.

Note that even if 8-bit channels are generally sufficient, there are specialized applications where 16-bit channels are required (medical imaging, for example).

As we saw in the previous chapter, OpenCV also allows you to create matrices (or images) with pixel values of other types, for example, integer (`CV_32U` or `CV_32S`) and floating-point (`CV_32F`) numbers. These are very useful for storing, for example, intermediate values in some image-processing tasks. Most operations can be applied on matrices of any type; others require a specific type of work with only a given number of channels. Therefore, a good understanding of a function's or method's preconditions is essential in order to avoid common programming errors.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Throughout this chapter, we use the following color image as the input:

In this chapter, we will cover the following recipes:

- Accessing pixel values
- Scanning an image with pointers
- Scanning an image with iterators
- Writing efficient image-scanning loops
- Scanning an image with neighbor access
- Performing simple image arithmetic
- Remapping an image

Accessing pixel values

In order to access each individual element of a matrix, you just need to specify its row and column numbers. The corresponding element will be returned, which can be a single numerical value or a vector of values in the case of a multichannel image.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

To illustrate the direct access to pixel values, we will create a simple function that adds **salt-and-pepper noise** to an image. As the name suggests, salt-and-pepper noise is a particular type of noise in which some randomly selected pixels are replaced by a white or a black pixel. This type of noise can occur in faulty communications when the value of some pixels is lost during the transmission. In our case, we will simply randomly select a few pixels and assign a white color to them.

How to do it...

We are going to create a function for salt noise and how to use it. To do it, we are going to follow these steps:

1. Create a function that receives an input image and a number. This is the image that will be modified by our function. The second parameter is the number of pixels on which we want to overwrite white values:

```
void salt(cv::Mat image, int n) {
```

2. Define two variables that we are going to use to store a random position on an image:

```
int i,j;
```

3. Create a loop that iterates `n` number of times. This is the second parameter and defines the number of pixels on which we want to overwrite white values:

```
for (int k=0; k<n; k++) {
```

4. Generate two random image positions using the `std::rand()` function. We store the `x` value on an `i` variable and the `y` value on `j`. The `std::rand()` function returns a value between `0` and `RAND_MAX`, then we apply a module `%` to it with `cols` or `rows` of the image to return only values between `0` and the width or height:

```
// rand() is the random number generator
    i= std::rand()%image.cols;
    j= std::rand()%image.rows;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Using the `type` method, we distinguish between the two cases of gray-level and color images. In the case of a gray-level image, the number `255` is assigned to the single 8-bit value using the `Mat` function `at<type>(y,x)`:

```
if (image.type() == CV_8UC1) { // gray-level image

   image.at<uchar>(j,i)= 255;

 }
```

6. For a color image, you need to assign `255` to the three primary color channels in order to obtain a white pixel. To access each channel, we can use the array access (`nChannel`) where `nChannel` is the number of channels, `0` is blue, `1` is green, and `2` is blue in the BGR format:

```
else if (image.type() == CV_8UC3) { // color image

   image.at<cv::Vec3b>(j,i)[0]= 255;
   image.at<cv::Vec3b>(j,i)[1]= 255;
   image.at<cv::Vec3b>(j,i)[2]= 255;
 }
```

7. Finally, remember to close the loop and function brackets:

```
  }
}
```

8. To use this function, we can read an image from our disk:

```
// open the image
   cv::Mat image= cv::imread("boldt.jpg");
```

9. Now call the function using the image loaded and a number of pixels to change, for example, `3000`:

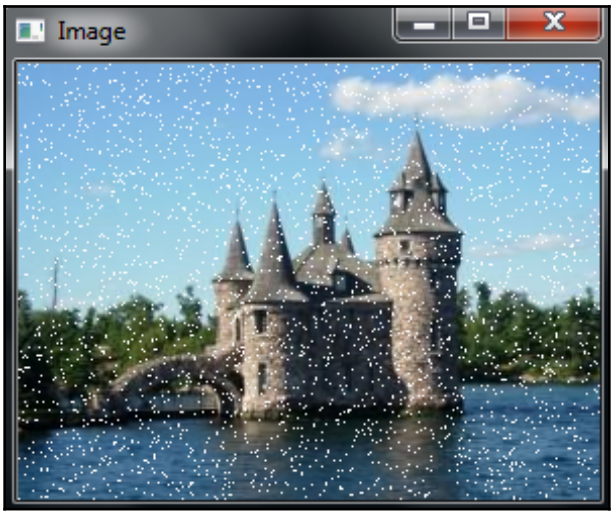
```
// call function to add noise
salt(image,3000);
```

10. Finally, display the image using the `cv::imshow` function:

```
// display image
cv::imshow("Image",image);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The resulting image will look as follows, as shown in this screenshot:



Let's see how the preceding instructions work when we execute them.

How it works...

The `cv::Mat` class includes several methods to access the different attributes of an image. The public member variables, `cols` and `rows`, give you the number of columns and rows in the image. For element access, `cv::Mat` has the `at` (`int y, int x`) method. However, the type returned by a method must be known at compile time, and since `cv::Mat` can hold elements of any type, the programmer needs to specify the return type that is expected. This is why the `at` method has been implemented as a template method. So, when you call it, you must specify the `image` element type as follows:

```
image.at<uchar>(j,i)= 255;
```

It is important to note that it is the programmer's responsibility to make sure that the type specified matches the type contained in the matrix. The `at` method does not perform any type of conversion.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In color images, each pixel is associated with three components—the red, green, and blue channels. Therefore, a `cv::Mat` class that contains a color image will return a vector of three 8-bit values. OpenCV has a defined type for such short vectors, and it is called `cv::Vec3b`. This is a vector of three **unsigned characters**. This explains why the element access to the pixels of a color pixel is written as follows:

```
image.at<cv::Vec3b>(j,i)[channel]= value;
```

The `channel` index designates one of the three color channels. OpenCV stores the `channel` values in the blue, green, red order (blue is, therefore, channel `0`).

Similar vector types also exist for two-element and four-element vectors (`cv::Vec2b` and `cv::Vec4b`) as well as for other element types. For example, for a two-element float vector, the last letter of the type name would be replaced by `f`, that is, `cv::Vec2f`. In the case of a short integer, the last letter is replaced by `s`, by `i` for an integer, and by `d` for a double-precision floating-point vector. All of these types are defined using the `cv::Vec<T,N>` template class, where `T` is the type and `N` is the number of vector elements.

On a last note, you might have been surprised by the fact that our image-modifying function uses a pass-by-value image parameter. This works because when images are copied, they still share the same image data. So, you do not necessarily have to transmit images by references when you want to modify their content. Incidentally, pass-by-value parameters often make code optimization easier for the compiler.

There's more...

The `cv::Mat` class has been made generic by defining it using C++ templates.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The cv::Mat_ template class

Using the `at` method of the `cv::Mat` class can sometimes be cumbersome because the returned type must be specified as a template argument in each call. In cases where the matrix type is known, it is possible to use the `cv::Mat_` class, which is a template subclass of `cv::Mat`. This class defines a few extra methods but no new data attributes so that pointers or references to one class can be directly converted to the other class. Among the extra methods, there is `operator()`, which allows direct access to matrix elements. Therefore, if an image is a `cv::Mat` variable that corresponds to a `uchar` matrix, then you can write the following code:

```
// use image with a Mat_ template
cv::Mat_<uchar> im2(image);
 im2(50,100)= 0; // access to row 50 and column 100
```

Since the type of the `cv::Mat_` elements are declared when the variable is created, the `operator()` method knows at compile time which type is to be returned. Other than the fact that it is shorter to write, using the `operator()` method provides exactly the same result as the `at` method.

See also

- The *There's more...* section of the *Scanning an image with pointers* recipe explains how to create a function with input and output parameters.
- The *Writing efficient image-scanning loops* recipe proposes a discussion on the efficiency of this method.

We've successfully learned how to access the pixel value. Now let's move on to the next recipe!

Scanning an image with pointers

In most image-processing tasks, you need to scan all the pixels of an image in order to perform a computation. Considering a large number of pixels will need to be visited, it is essential that you perform this task in an efficient way. This recipe and the next one will show you different ways of implementing efficient scanning loops. This recipe uses the pointer arithmetic.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

We will illustrate the image-scanning process by accomplishing a simple task—reducing the number of colors in an image.

Color images are composed of three-channel pixels. Each of these channels corresponds to the intensity value of one of the three primary colors: red, green, and blue. Since each of these values is an 8-bit unsigned character, the total number of colors is *256 x 256 x 256*, which is more than 16 million colors. Consequently, to reduce the complexity of analysis, it is sometimes useful to reduce the number of colors in an image. One way to achieve this goal is simply to subdivide the RGB space into cubes of equal sizes. For example, if you reduce the number of colors in each dimension by `8`, then you would obtain a total of *32 x 32 x 32* colors. Each color in the original image is then assigned a new color value in the color-reduced image that corresponds to the value in the center of the cube to which it belongs.

Therefore, the basic color reduction algorithm is simple. If `N` is the reduction factor, then divide the value by `N` (the integer division, therefore, the remainder is lost) for each pixel in the image and for each channel of this pixel. Then, multiply the result by `N`; this will give you the multiple of `N` just below the input pixel value. Just add `N/2` and you obtain the central position of the interval between two adjacent multiples of `N`. If you repeat this process for each 8-bit channel value, then you will obtain a total of `256/N x 256/N x 256/N` possible color values.

How to do it...

To reduce the number of colors in an image, follow the next steps:

1. The signature of our color reduction function will be as follows. The user provides an image and the per-channel reduction factor `div`:

```
void colorReduce(cv::Mat image, int div=64);
```

   Here, the processing is done **in-place**; that is, the pixel values of the input image are modified by the function. See the *There's more...* section of this recipe for a more general function signature with input and output arguments.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. The processing is done simply by creating a double loop that goes over all the pixel values. The first loop scans every row, getting the pointer of the row image data:

```
for (int j=0; j<image.rows; j++) {

    // get the address of row j
    uchar* data= image.ptr<uchar>(j);
```

3. The second loop goes over every column of the row pointer and applies the reduction of color with this formula:

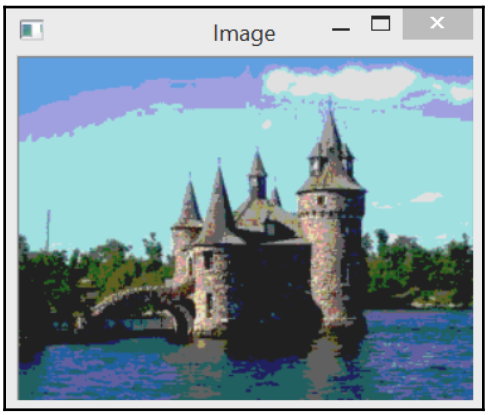
```
        for (int i=0; i<nc; i++) {

            // process each pixel ---------------------
                  data[i]=    data[i]/div*div + div/2;

            // end of pixel processing ----------------

        } // end of line
    }
}
```

4. This function can be tested by loading an image and calling the function as follows:

```
// read the image
image= cv::imread("boldt.jpg");
// process the image
colorReduce(image,64);
// display the image
cv::namedWindow("Image");
cv::imshow("Image",image);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This will give you, for example, the following image (refer to the book's graphics PDF to view this image in color), as shown in this screenshot:



Let's see how the preceding instructions work when we execute them.

How it works...

In a color image, the first three bytes of the image's data buffer give values of the upper-left pixel to the three-color channel. The next three bytes are the values of the second pixel of the first row, and so on (remember that OpenCV uses, by default, the BGR channel order). An image of width `W` and height `H` would then require a memory block of `WxHx3uchars`. However, for efficiency reasons, the length of a row can be padded with a few extra pixels. This is because some multimedia processor chips (for example, the Intel MMX architecture) can process images more efficiently when their rows are multiples of four or eight. Obviously, these extra pixels are not displayed or saved; their exact values are ignored. OpenCV designates the length of a padded row as the effective width. Obviously, if the image has not been padded with extra pixels, the effective width will be equal to the real image width. We have already learned that the `cols` and `rows` attributes give you the image's width and height; similarly, the `step` data attribute gives you the effective width in the number of bytes. Even if your image is of a type other than `uchar`, the `step` data will still give you the number of bytes in a row. The size of a pixel element is given by the `elemSize` method (for example, for a three-channel short integer matrix (`CV_16SC3`), `elemSize` will return `6`). Recall that the number of channels in the image is given by the `nchannels` method (which will be `1` for a gray-level image and `3` for a color image). Finally, the `total` method returns the total number of pixels (that is, the matrix entries) in the matrix.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The number of pixel values per row is then given by the following code:

```
int nc= image.cols * image.channels();
```

To simplify the computation of the pointer arithmetic, the `cv::Mat` class offers a method that gives you the address of an image row directly. This is the `ptr` method. It is a template method that returns the address of row `j`:

```
uchar* data= image.ptr<uchar>(j);
```

Note that in the processing statement, we could have equivalently used the pointer arithmetic to move from column to column. So, we could have written the following code:

```
*data= *data/div*div + div2; data++;
```

There's more...

The color reduction function presented in this recipe provides just one way of accomplishing this task. You could also use other color reduction formulas. A more general version of the function would also allow the specification of distinct input and output images. The image scanning can also be made more efficient by taking into account the continuity of the image data. Finally, it is also possible to use regular, low-level pointer arithmetic to scan the image buffer. All of these elements are discussed in the following subsections.

Other color reduction formulas

In our example, the color reduction is achieved by taking advantage of an integer division that floors the division result to the nearest lower integer as follows:

```
data[i]= (data[i]/div)*div + div/2;
```

The reduced color could have also been computed using the modulo operator that brings us to the nearest multiple of `div` (the per-channel reduction factor) as follows:

```
data[i]= data[i] - data[i]%div + div/2;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Another option would be to use bitwise operators. Indeed, if we restrict the reduction factor to a power of `2`, that is, `div=pow(2,n)`, then masking the first `n` bits of the pixel value would give us the nearest lower multiple of `div`. This mask would be computed by a simple bit shift as follows:

```
// mask used to round the pixel value
uchar mask= 0xFF<<n; // e.g. for div=16, mask= 0xF0
```

The color reduction would be given by the following code:

```
*data &= mask;      // masking
*data++ += div>>1;  // add div/2
```

In general, bitwise operations might lead to very efficient code, so they could constitute a powerful alternative when efficiency is a requirement.

Having input and output arguments

In our color reduction example, the transformation is directly applied to the input image, which is called an **in-place transformation**. This way, no extra image is required to hold the output result, which could save on memory usage when it is a concern. However, in some applications, the user wants to keep the original image intact. The user would then be forced to create a copy of the image before calling the function. Note that the easiest way to create an identical deep copy of an image is to call the `clone` method; for example, take a look at the following code:

```
// read the image
image= cv::imread("boldt.jpg");
// clone the image
cv::Mat imageClone= image.clone();
// process the clone
// orginal image remains untouched
colorReduce(imageClone);
// display the image result
cv::namedWindow("Image Result");
cv::imshow("Image Result",imageClone);
```

This extra overload can be avoided by defining a function that gives the user the option either to use or not use in-place processing. The signature of the method would then be as follows:

```
void colorReduce(const cv::Mat &image, // input image
                 cv::Mat &result,      // output image
                 int div=64);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that the input image is now passed as a `const` reference, which means that this image will not be modified by the function. The output image is passed as a reference so that the calling function will see the output argument modified by this call. When in-place processing is preferred, the same image is specified as the input and output:

```
colorReduce(image,image);
```

If not, another `cv::Mat` instance can be provided; for example, take a look at the following code:

```
cv::Mat result;
colorReduce(image,result);
```

The key here is first to verify whether the output image has an allocated data buffer with a size and pixel type that matches one of the input images. Very conveniently, this check is encapsulated inside the `create` method of `cv::Mat`. This is the method that is to be used when a matrix must be reallocated with a new size and type. If, by chance, the matrix already has the size and type specified, then no operation is performed and the method simply returns without touching the instance. Therefore, our function should simply start with a call to `create` unclear a matrix (if necessary) of the same size and type as the input image:

```
result.create(image.rows,image.cols,image.type());
```

The allocated memory block has a size of `total()*elemSize()`.The looping is then done with two pointers:

```
for (int j=0; j<nl; j++) {

    // get the addresses of input and output row j
    const uchar* data_in= image.ptr<uchar>(j);
    uchar* data_out= result.ptr<uchar>(j);

    for (int i=0; i<nc*nchannels; i++) {

        // process each pixel ---------------------
              data_out[i]= data_in[i]/div*div + div/2;

        // end of pixel processing ----------------

    } // end of line
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In cases where the same image is provided as the input and output, this function becomes completely equivalent to the first version presented in this recipe. If another image is provided as the output, the function will work correctly irrespective of whether the image has been allocated prior to the function call.

Efficient scanning of continuous images

We previously explained that, for efficiency reasons, an image can be padded with extra pixels at the end of each row. However, it is interesting to note that when the image is unpadded, this one can also be seen as a long one-dimensional array of `WxH` pixels. A convenient `cv::Mat` method can tell us whether the image has been padded. It is the `isContinuous` method that returns `true` if the image does not include padded pixels. Note that we could also check the continuity of the matrix by writing the following test:

```
// check if size of a line (in bytes)
// equals the number of columns times pixel size in bytes
image.step == image.cols*image.elemSize();
```

To be complete, this test should also check whether the matrix has only one line, in which case, it is continuous by definition. Nevertheless, always use the `isContinuous` method to test the continuity condition. In some specific processing algorithms, you can take advantage of the continuity of the image by processing it in one single (longer) loop. Our processing function would then be written as follows:

```
void colorReduce(cv::Mat &image, int div=64) {

     int nl= image.rows; // number of lines
     int nc= image.cols * image.channels();

     if (image.isContinuous())
     {
        // then no padded pixels
        nc= nc*nl;
        nl= 1;  // it is now a long 1D array
     }

     // this loop is executed only once
     // in case of continuous images
     for (int j=0; j<nl; j++) {

          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {

            // process each pixel ---------------------
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
                    data[i]= data[i]/div*div + div/2;

                // end of pixel processing ----------------

              } // end of line
        }
    }
```

Now when the continuity test tells us that the image does not contain padded pixels, we eliminate the outer loop by setting the width to `1` and the height to `WxH`. Note that there is also a `reshape` method that could have been used here. You would write the following in that case:

```
        if (image.isContinuous())
        {
           // no padded pixels
           image.reshape(1,   // new number of channels
                         1); // new number of rows
        }

        int nl= image.rows; // number of lines
        int nc= image.cols * image.channels();
```

The `reshape` method changes the matrix dimensions without requiring any memory copying or reallocation. The first parameter is the new number of channels and the second one is the new number of rows. The number of columns is readjusted accordingly.

In these implementations, the inner loop processes all image pixels in a sequence. This approach is mainly advantageous when several small images are scanned simultaneously into the same loop.

Low-level pointer arithmetics

In the `cv::Mat` class, the image data is contained in a memory block of unsigned characters. The address of the first element of this memory block is given by the `data` attribute that returns an unsigned character pointer. So, to start your loop at the beginning of the image, you could have written the following code:

```
uchar *data= image.data;
```

Moving from one row to the next could have been done by moving your row pointer using the effective width as follows:

```
data+= image.step;  // next line
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `step` method gives you the total number of bytes (including the padded pixels) in a line. In general, you can obtain the address of the pixel at row `j` and column `i` as follows:

```
// address of pixel at (j,i) that is &image.at(j,i)
data= image.data+j*image.step+i*image.elemSize();
```

However, even if this would work in our example, it is not recommended that you proceed in that way.

See also

- The *Writing efficient image-scanning loops* recipe in this chapter proposes a discussion concerning the efficiency of the scanning methods presented here

We've learned successfully how to scan an image with pointers. Now let's move on to the next recipe!

Scanning an image with iterators

In object-oriented programming, looping over a data collection is usually done using iterators. Iterators are specialized classes that are built to go over each element of a collection, hiding how the iteration over each element is specifically done for a given collection. This application of the information-hiding principle makes scanning a collection easier and safer. In addition, it makes it similar in form no matter what type of collection is used. The **Standard Template Library** (**STL**) has an iterator class associated with each of its collection classes. OpenCV then offers a `cv::Mat` iterator class that is compatible with the standard iterators found in the C++ STL.

Getting ready

In this recipe, we again use the color reduction example described in the previous recipe.

An iterator object for a `cv::Mat` instance can be obtained by first creating a `cv::MatIterator_` object. As is the case with `cv::Mat_`, the underscore indicates that this is a template subclass. Indeed, since image iterators are used to access the image elements, the return type must be known at the time of compilation. The iterator is then declared as follows:

```
cv::MatIterator_<cv::Vec3b> it;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Alternatively, you can also use the `iterator` type defined inside the `Mat_` template class as follows:

```
cv::Mat_<cv::Vec3b>::iterator it;
```

Now we are going to apply the iterators to the color reduction example.

How to do it...

To apply the iterators to the color reduction example, the following steps are performed:

1. We are going to loop over the pixels using the usual `begin` and `end` iterator methods, except that these ones are, again, template methods. Then, we have to get the `begin` position:

```
// obtain iterator at initial position
cv::Mat_<cv::Vec3b>::iterator it=
          image.begin<cv::Vec3b>();
```

2. Then, we must obtain the `end` position of the iterator:

```
// obtain end position
cv::Mat_<cv::Vec3b>::iterator itend=
          image.end<cv::Vec3b>();
```

3. Next, we must loop over the iterator until the `end` position:

```
// loop over all pixels
for ( ; it!= itend; ++it) {
```

4. Finally, apply the color reduction function to the pixel:

```
      // process each pixel ---------------------

     (*it)[0]= (*it)[0]/div*div + div/2;
     (*it)[1]= (*it)[1]/div*div + div/2;
     (*it)[2]= (*it)[2]/div*div + div/2;

      // end of pixel processing ----------------
   }
}
```

Remember that the iterator here returns a `cv::Vec3b` instance because we are processing a color image. Each color channel element is accessed using the dereferencing operator `[]`.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Working with iterators always follows the same pattern, no matter what kind of collection is scanned.

First, you create your `iterator` object using the appropriate specialized class, which, in our example, is `cv::Mat_<cv::Vec3b>::iterator` (or `cv::MatIterator_<cv::Vec3b>`).

You then obtain an iterator initialized at the starting position (in our example, the upper left corner of the image). This is done using a `begin` method. With a `cv::Mat` instance, you obtain it as `image.begin<cv::Vec3b>()`. You can also use arithmetic on the iterator. For example, if you wish to start at the second row of an image, you can initialize your `cv::Mat` iterator at `image.begin<cv::Vec3b>()+image.cols`. The `end` position of your collection is obtained similarly, but by using the `end` method. However, the iterator thus obtained is just outside your collection. This is why your iterative process must stop when it reaches the `end` position. You can also use arithmetic on this iterator; for example, if you wish to stop before the last row, your final iteration would stop when the iterator reaches `image.end<cv::Vec3b>()-image.cols`.

Once your iterator is initialized, you create a loop that goes over all elements until the end is reached. A typical `while` loop will look like the following code:

```
while (it!= itend) {

   // process each pixel ---------------------

   // end of pixel processing ----------------

   ++it;
}
```

The `++` operator is the one that is to be used to move to the next element. You can also specify the larger `step` size. For example, `it+=10` would be processed every `10` pixels.

Finally, inside the processing loop, you use the dereferencing `*` operator in order to access the current element. Using this, you can read (for example, `element= *it;`) or write (for example, `*it= element;`). Note that it is also possible to create constant iterators that you use if you receive a reference to `const cv::Mat` or if you wish to signify that the current loop does not modify the `cv::Mat` instance. These are declared as follows:

```
cv::MatConstIterator_<cv::Vec3b> it;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Or they are declared as follows:

```
cv::Mat_<cv::Vec3b>::const_iterator it;
```

There's more...

In this recipe, the start and end positions of the iterator were obtained using the `begin` and `end` template methods. As we did in the first recipe of this chapter, we could also have obtained them using a reference to a `cv::Mat_` instance. This would avoid the need to specify the iterator type in the `begin` and `end` methods since this one is specified when the `cv::Mat_` reference is created:

```
cv::Mat_<cv::Vec3b> cimage(image);
cv::Mat_<cv::Vec3b>::iterator it= cimage.begin();
cv::Mat_<cv::Vec3b>::iterator itend= cimage.end();
```

See also

- The *Writing efficient image-scanning loops* recipe proposes a discussion on the efficiency of iterators when scanning an image.
- Also, if you are not familiar with the concept of iterators in object-oriented programming and how they are implemented in ANSI C++, you should read a tutorial on STL iterators. Simply search the web with the keywords `STL Iterator`, and you will find numerous references to the subject.

We've successfully learned how to scan an image with iterators. Now let's move on to the next recipe!

Writing efficient image-scanning loops

In the previous recipes of this chapter, we presented different ways of scanning an image in order to process its pixels. In this recipe, we will compare the efficiency of these different approaches.

When you write an image-processing function, efficiency is often a concern. When you design your function, you will frequently need to check the computational efficiency of your code in order to detect any bottleneck in your processing that might slow down your program.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

However, it is important to note that, unless necessary, optimization should not be undertaken at the cost of reducing the program clarity. Simple code is indeed always easier to debug and maintain. Only code portions that are critical to a program's efficiency should be heavily optimized.

How to do it...

In order to measure the execution time of a function or a portion of code, there exists a very convenient OpenCV function called `cv::getTickCount()`. This function gives you the number of clock cycles that have occurred since the last time you started your computer. Since we want the execution time of a code portion given in seconds, we use another method, `cv::getTickFrequency()`. This gives us the number of cycles per second. The usual pattern to be used in order to obtain the computational time of a given function (or a portion of code) would be as follows:

1. Get the start time point:

```
const int64 start = cv::getTickCount();
```

2. Call your functions or the code you want to measure:

```
colorReduce(image); // a function call
```

3. Calculate the execution time, getting the actual time point minus the start point and dividing it by the number of cycles per second:

```
// elapsed time in seconds
double duration = (cv::getTickCount()-
start)/cv::getTickFrequency();
```

How it works...

The execution times of the different implementations of the `colorReduce` function from this chapter are reported here. The absolute runtime numbers would differ from one machine to another (here, we used a 2.83 GHz machine equipped with a 64-bit Intel Core 2 Quad Q9550 processor). It is rather interesting to look at their relative differences. These results are also dependent on the specific compiler that is used to produce the executable file. Our tests report the average time to reduce the colors of an image that has a resolution of *4,288 x 2,848* pixels.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

First, we compare the three ways of computing the color reduction as presented in the *There's more...* section of the *Scanning an image with pointers* recipe. It is interesting to observe that both the integer division formula and the one with bitwise operators take about the same execution time, that is, 31 ms. The version based on the modulo operator, however, takes 52 ms. This represents a difference of almost 50% between the fastest and the slowest! It is, therefore, important to take the time to identify the most efficient way of computing a result in an image loop, as the net impact can be very significant.

When an output image that needs to be reallocated is specified instead of in-place processing, the execution time becomes 33 ms. The extra duration represents the overhead for memory allocation.

In a loop, you should avoid repetitive computations of values that could be precomputed instead. This consumes time, obviously. For example, take the following inner loop of the color reduction function:

```
 int nc= image.cols * image.channels();
 uchar div2= div>>1;
      for (int i=0; i<nc; i++) {
```

Then, replace it with the following one:

```
      for (int i=0; i<image.cols * image.channels(); i++) {
// . . ....
       *data++ += div>>1;
```

The preceding code is a loop where you need to compute the total number of elements in a line and the `div>>1` result again and again; you will obtain a runtime of 61 ms, which is significantly slower than the original version, which took 31 ms. Note, however, that some compilers might be able to optimize these kinds of loops and still obtain an efficient code.

The version of the color reduction function that uses iterators, as shown in the *Scanning an image with iterators* recipe, gives slower results at 56 ms. The main objective of iterators is to simplify the image-scanning process and make it less prone to errors.

For completeness, we also implemented a version of the function that uses the `at` method for pixel access. The main loop of this implementation would then read simply as follows:

```
for (int j=0; j<nl; j++) {
  for (int i=0; i<nc; i++) {

    // process each pixel ---------------------
    image.at<cv::Vec3b>(j,i)[0]=
               image.at<cv::Vec3b>(j,i)[0]/div*div + div/2;
    image.at<cv::Vec3b>(j,i)[1]=
              image.at<cv::Vec3b>(j,i)[1]/div*div + div/2;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
      image.at<cv::Vec3b>(j,i)[2]=
                image.at<cv::Vec3b>(j,i)[2]/div*div + div/2;

      // end of pixel processing ----------------

    } // end of line
  }
```

This implementation is much slower when a runtime of 91 ms is obtained. This method should then be used only for the random access of image pixels but never when scanning an image.

A shorter loop with few statements is generally more efficiently executed than a longer loop over a single statement, even if the total number of elements processed is the same. Similarly, if you have `N` different computations to apply to a pixel, apply all of them in one loop rather than writing `N` successive loops, one for each computation.

We also performed the continuity test that produces one loop in the cases of continuous images instead of the regular double loop over lines and columns. For a very large image, such as the one we used in our tests, this optimization is not significant (29 ms instead of 31 ms), but, in general, it is always a good practice to use this strategy, since it can lead to a significant gain in speed.

There's more...

Multithreading is another way to increase the efficiency of your algorithms, especially since the advent of multicore processors. **OpenMP** and the **Intel Threading Building Blocks** (**TBB**) are two popular APIs that are used in concurrent programming to create and manage your threads. In addition, C++11 now offers built-in support for threads.

See also

- The *Performing simple image arithmetic* recipe presents an implementation of the color reduction function (described in the *There's more...* section) that uses the OpenCV 2 arithmetic image operators and has a runtime of 53 ms.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

- The *Applying look-up tables to modify image appearance* recipe of `Chapter 4`, *Counting the Pixels with Histograms*, describes an implementation of the color reduction function based on a lookup table. The idea is to precompute all intensity reduction values that lead to a runtime of 29 ms.

We've successfully learned how to write efficient image-scanning loops. Now let's move on to the next recipe!

Scanning an image with neighbor access

In image processing, it is common to have a processing function that computes a value at each pixel location based on the value of the neighboring pixels. When this neighborhood includes pixels of the previous and next lines, you then need to scan several lines of the image simultaneously. This recipe shows you how to do it.

Getting ready

To illustrate this recipe, we will apply a processing function that sharpens an image. It is based on the `Laplace` operator (which will be discussed in `Chapter 6`, *Filtering the Images*). It is indeed a well-known result in image processing that if you subtract its Laplacian from an image, the image edges are amplified, thereby giving a sharper image. This sharpened value is computed as follows:

```
sharpened_pixel= 5*current-left-right-up-down;
```

Here, `left` is the pixel that is immediately on the left-hand side of the current one, `up` is the corresponding one on the previous line, and so on.

How to do it...

To create the `sharpen` function, we are going to follow these steps:

1. We are going to create a `sharpen` function with an input and output image. This time, the processing cannot be accomplished in-place; the users need to provide an output image:

```
void sharpen(const cv::Mat &image, cv::Mat &result) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Allocate the output result image and get the number of channels of the input image with the `.channels()` function:

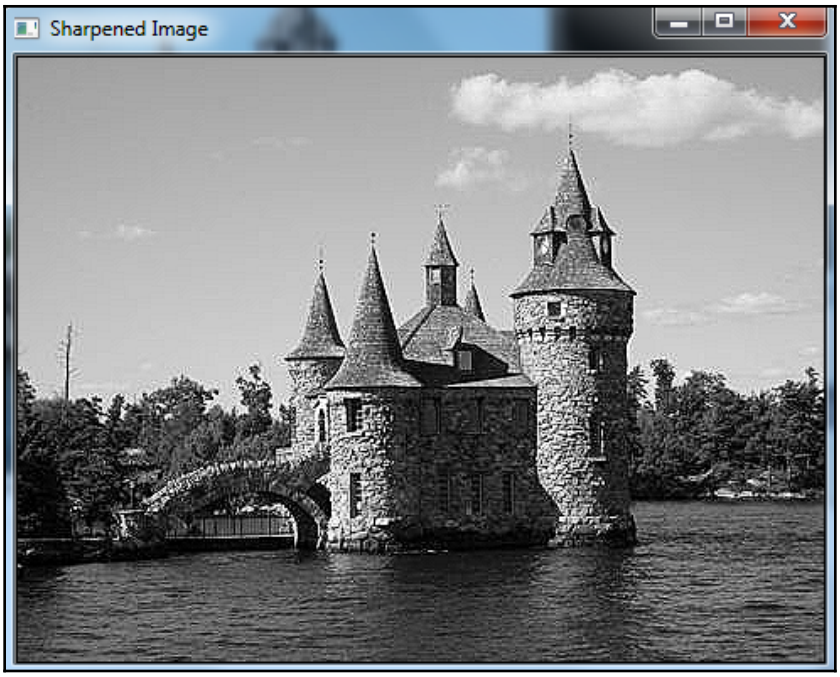
```
 // allocate if necessary
result.create(image.size(), image.type());
int nchannels= image.channels(); // get number of channels
```

3. We have to loop for every row. The image scanning is done using three pointers, one for the current line, one for the preceding line, and another one for the following line. Also, since each pixel computation requires access to its neighbors, it is not possible to compute a value for the pixels of the first and last rows of the image as well as the pixels of the first and last columns. The loop can then be written as follows:

```
 // for all rows (except first and last)
for (int j= 1; j<image.rows-1; j++) {

  const uchar* previous=
      image.ptr<const uchar>(j-1);     // previous row
  const uchar* current=
      image.ptr<const uchar>(j);       // current row
  const uchar* next=
      image.ptr<const uchar>(j+1);     // next row

  uchar* output= result.ptr<uchar>(j); // output row

  for (int i=nchannels; i<(image.cols-1)*nchannels; i++) {

     *output++= cv::saturate_cast<uchar>(
                5*current[i]-current[i-nchannels]-
                current[i+nchannels]-previous[i]-next[i]);
  }
 }
}
```

Note how we wrote the function so that it would work on both gray-level and color images. If we apply this function on a gray-level version of our test image, the following result is obtained:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Let's see how the instructions work when we execute them.

How it works...

In order to access the neighboring pixels of the previous and next rows, you must simply define additional pointers that are jointly incremented. You then access the pixels of these lines inside the scanning loop.

In the computation of the output pixel value, the `cv::saturate_cast` template function is called on the result of the operation. This is because it often happens that a mathematical expression applied on pixels leads to a result that goes outside the range of the permitted pixel values (that is, below `0` or over `255`). The solution is then to bring back the values inside this 8-bit range. This is done by changing negative values to `0` and values over `255` to `255`. This is exactly what the `cv::saturate_cast<uchar>` function is doing. In addition, if the input argument is a floating-point number, then the result is rounded to the nearest integer. You can obviously use this function with other types in order to guarantee that the result will remain within the limits defined by this type.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Border pixels that cannot be processed because their neighborhood is not completely defined need to be handled separately. Here, we simply set them to `0`. In other cases, it could be possible to perform a special computation for these pixels, but most of the time, there is no point in spending the time to process these very few pixels. In our function, these border pixels are set to `0` using two special methods. The first one is `row` and its dual is `col`. They return a special `cv::Mat` instance composed of a single-line ROI (or a single-column ROI) as specified in a parameter (remember, we discussed regions of interest in the previous chapter). No copy is made here because if the elements of this one-dimensional matrix are modified, they will also be modified in the original image. This is what we do when the `setTo` method is called. This method assigns a value to all the elements of a matrix. Take a look at the following statement:

```
result.row(0).setTo(cv::Scalar(0));
```

The preceding statement assigns the value of `0` to all pixels of the first line of the resulting image. In the case of a three-channel color image, you would use `cv::Scalar(a,b,c)` to specify the three values to be assigned to each channel of the pixel.

There's more...

When a computation is done over a pixel neighborhood, it is common to represent this with a kernel matrix. This kernel describes how the pixels involved in the computation are combined in order to obtain the desired result. For the sharpening filter used in this recipe, the kernel would be as follows:

0	-1	0
-1	5	-1
0	-1	0

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Unless stated otherwise, the current pixel corresponds to the center of the kernel. The value in each cell of the kernels represents a factor that multiplies the corresponding pixel. The result of the application of the kernel on a pixel is then given by the sum of all these multiplications. The size of the kernel corresponds to the size of the neighborhood (here, *3 x 3*). Using this representation, it can be seen that, as required by the sharpening filter, the four horizontal and vertical neighbors of the current pixel are multiplied by `-1`, while the current one is multiplied by `5`. Applying a kernel to an image is more than a convenient representation; it is the basis for the concept of convolution in signal processing. The kernel defines a filter that is applied to the image.

Since filtering is a common operation in image processing, OpenCV has defined a special function that performs this task—the `cv::filter2D` function. To use this, you just need to define a kernel (in the form of a matrix). The function is then called with the image and the kernel, and it returns the filtered image. Using this function, it is, therefore, easy to redefine our sharpening function as follows:

```
void sharpen2D(const cv::Mat &image, cv::Mat &result) {

   // Construct kernel (all entries initialized to 0)
   cv::Mat kernel(3,3,CV_32F,cv::Scalar(0));
   // assigns kernel values
   kernel.at<float>(1,1)= 5.0;
   kernel.at<float>(0,1)= -1.0;
   kernel.at<float>(2,1)= -1.0;
   kernel.at<float>(1,0)= -1.0;
   kernel.at<float>(1,2)= -1.0;

   //filter the image
   cv::filter2D(image,result,image.depth(),kernel);
}
```

This implementation produces exactly the same result as the previous one (and with the same efficiency). If you input a color image, then the same kernel will be applied to all three channels. Note that it is particularly advantageous to use the `filter2D` function with the large kernel, as it uses, in this case, a more efficient algorithm.

See also

- `Chapter 6`, *Filtering the Images*, provides us with more explanations concerning the concept of image filtering.

We've successfully learned how to scan an image with neighbor access. Now let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Performing simple image arithmetic

Images can be combined in different ways. Since they are regular matrices, they can be added, subtracted, multiplied, or divided. OpenCV offers various image arithmetic operators, and their use is discussed in this recipe.

Getting ready

Let's work with a second image that we will combine with our input image using an arithmetic operator. The following screenshot represents this second image:

Let's start with our recipe.

How to do it...

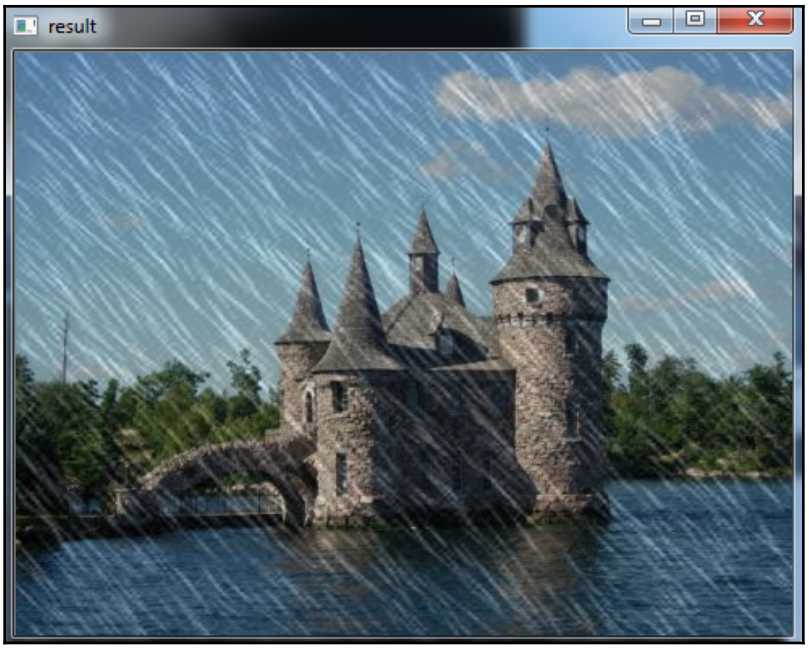
Here, we add two images. This is useful when you want to create some special effects or to overlay information over an image:

1. Call the `cv::add` function or, more precisely here, the `cv::addWeighted` function, since we want a weighted sum as follows:

```
cv::addWeighted(image1,0.7,image2,0.9,0.,result);
```

The operation results in a new image, as seen in the following screenshot:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Let's see how the preceding instructions work when we execute them.

How it works...

All binary arithmetic functions work the same way. Two inputs are provided and a third parameter specifies the output. In some cases, weights that are used as scalar multipliers in the operation can be specified. Each of these functions comes in several flavors; `cv::add` is a good example of a function that is available in many forms:

```
// c[i]= a[i]+b[i];
cv::add(imageA,imageB,resultC);
// c[i]= a[i]+k;
cv::add(imageA,cv::Scalar(k),resultC);
// c[i]= k1*a[1]+k2*b[i]+k3;
cv::addWeighted(imageA,k1,imageB,k2,k3,resultC);
// c[i]= k*a[1]+b[i];
cv::scaleAdd(imageA,k,imageB,resultC);
```

For some functions, you can also specify a mask:

```
// if (mask[i]) c[i]= a[i]+b[i];
cv::add(imageA,imageB,resultC,mask);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If you apply a mask, the operation is performed only on pixels for which the mask value is not null (the mask must be one-channel). Have a look at the different forms of the `cv::subtract`, `cv::absdiff`, `cv::multiply`, and `cv::divide` functions. Bitwise operators (operators applied to each individual bit of the pixels' binary representations) are also available: `cv::bitwise_and`, `cv::bitwise_or`, `cv::bitwise_xor`, and `cv::bitwise_not`. The `cv::min` and `cv::max` operators, which find the per-element maximum or minimum pixel values, are also very useful.

In all cases, the `cv::saturate_cast` function (see the preceding recipe) is always used to make sure that the results stay within the defined pixel value domain (that is, to avoid overflow or underflow).

The images must have the same size and type (the output image will be reallocated if it does not match the input size). Also, since the operation is performed per element, one of the input images can be used as the output.

Several operators that take a single image as the input are also available—`cv::sqrt`, `cv::pow`, `cv::abs`, `cv::cuberoot`, `cv::exp`, and `cv::log`. In fact, there exists an OpenCV function for almost any operation you have to apply on image pixels.

There's more...

It is also possible to use the usual C++ arithmetic operator on the `cv::Mat` instances or on the individual channels of `cv::Mat` instances. The following two subsections explain how to do this.

Overloaded image operators

Very conveniently, most arithmetic functions have their corresponding operator overloaded in OpenCV 2. Consequently, the call to `cv::addWeighted` can be written as follows:

```
result= 0.7*image1+0.9*image2;
```

The preceding code is a more compact form that is also easier to read. These two ways of writing the weighted sum are equivalent. In particular, the `cv::saturate_cast` function will still be called in both cases.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Most C++ operators have been overloaded. Among them are the bitwise operators `&`, `|`, `^`, and `~`; the `min`, `max`, and `abs` functions. The comparison operators `<`, `<=`, `==`, `!=`, `>`, and `>=` have also been overloaded, and they return an 8-bit binary image. You will also find the `m1*m2` matrix multiplication (where `m1` and `m2` are both `cv::Mat` instances), the `m1.inv()` matrix inversion, the `m1.t()` transpose, the `m1.determinant()` determinant, the `v1.norm()` vector norm, the `v1.cross(v2)` cross-product, the `v1.dot(v2)` dot product, and so on. When this makes sense, you also have the corresponding operator/assignment operator defined (the `+=` operator, for example).

In the *Writing efficient image-scanning loops* recipe, we presented a color reduction function that was written using loops that scan the image pixels to perform some arithmetic operations on them. From what we learned here, this function could be rewritten simply using arithmetic operators on the input image as follows:

```
image=(image&cv::Scalar(mask,mask,mask))
             +cv::Scalar(div/2,div/2,div/2);
```

The use of `cv::Scalar` is due to the fact that we are manipulating a color image. Performing the same test as we did in the *Writing efficient image-scanning loops* recipe, we obtain an execution time of 53 ms. Using the image operators makes the code so simple, and the programmer so productive, that you should consider their use in most situations.

Splitting the image channels

You'll sometimes want to process the different channels of an image independently. For example, you might want to perform an operation only on one channel of the image. You can, of course, achieve this in an image-scanning loop. However, you can also use the `cv::split` function that will copy the three channels of a color image into three distinct `cv::Mat` instances. Suppose we want to add our rain image to the blue channel only. The following is how we would proceed:

```
// create vector of 3 images
std::vector<cv::Mat> planes;
// split 1 3-channel image into 3 1-channel images
cv::split(image1,planes);
// add to blue channel
planes[0]+= image2;
// merge the 3 1-channel images into 1 3-channel image
cv::merge(planes,result);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `cv::merge` function performs the inverse operation; that is, it creates a color image from three one-channel images.

We've successfully learned how to perform simple image arithmetic. Now let's move on to the next recipe!

Remapping an image

In the recipes of this chapter, you learned how to read and modify the pixel values of an image. The last recipe will teach you how to modify the appearance of an image by moving its pixels. The pixel values are not changed by this process; it is rather the position of each pixel that is remapped to a new location. This is useful in order to create special effects on an image or to correct image distortions caused, for example, by a lens.

How to do it...

In order to use the OpenCV `remap` function, first you simply have to define the map to be used in the remapping process. Second, you have to apply this map on an input image. Obviously, it is the way you define your map that will determine the effect that will be produced. In our example, we define a transformation function that will create a wavy effect on the image. To do it, follow the next steps:

1. Create our `wave` function with two `params`, an input image, and an output image `result`:

```
// remapping an image by creating wave effects
void wave(const cv::Mat &image, cv::Mat &result) {
```

2. Create the two mapping variables, one for x positions and another for y positions, which store the new positions for remapping:

```
// the map functions
cv::Mat srcX(image.rows,image.cols,CV_32F);
cv::Mat srcY(image.rows,image.cols,CV_32F);
```

3. Loop over every pixel:

```
// creating the mapping
for (int i=0; i<image.rows; i++) {
  for (int j=0; j<image.cols; j++) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Assign to the map the *x* position the actual position, `j`:

```
srcX.at<float>(i,j)= j; // remain on same column
```

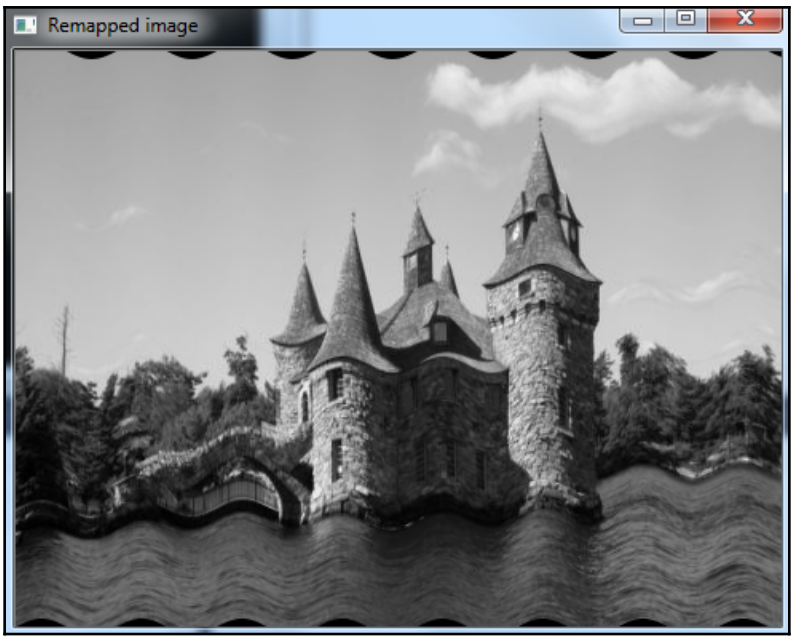
5. Calculate the new position for *y* using a sinusoid function using the *x* value as an input of `sin`:

```
srcY.at<float>(i,j)= i+5*sin(j/10.0);
```

6. Close the loop for calculating `map` variables and apply the `remap` function to the input image:

```
      }
    }

    // applying the mapping
    cv::remap(image, result, srcX, srcY, cv::INTER_LINEAR);
}
```

The result is as follows:



Let's see how the preceding instructions work when we execute them.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

The objective of remapping is to produce a new version of an image in which pixels have changed their positions. To construct this new image, we need to know what the original position for each pixel in the destination image is in the source image. The mapping function that is needed is, therefore, the one that will give us the original pixel positions as a function of the new pixel positions. This is called **backward mapping** because the transformation describes how the pixels of the new images are mapped back to the original image. In OpenCV, backward mapping is described using two maps—one for the *x* coordinates and one for the *y* coordinates. They are both represented by floating-point `cv::Mat` instances:

```
// the map functions
cv::Mat srcX(image.rows,image.cols,CV_32F); // x-map
cv::Mat srcY(image.rows,image.cols,CV_32F); // y-map
```

The size of these matrices will define the size of the destination image. The value of the `(i,j)` pixel of the destination image can then be read in the source image, by using the following line of code:

```
( srcX.at<float>(i,j) , srcY.at<float>(i,j) )
```

For example, a simple image flip effect such as the one we demonstrated in `Chapter 1`, *Playing with Images*, can be created by the following maps:

```
// creating the mapping
for (int i=0; i<image.rows; i++) {
  for (int j=0; j<image.cols; j++) {

    // horizontal flipping
    srcX.at<float>(i,j)= image.cols-j-1;
    srcY.at<float>(i,j)= i;
  }
}
```

To generate the resulting image, you simply call the OpenCV `remap` function:

```
// applying the mapping
cv::remap(image,  // source image
         result, // destination image
        srcX,   // x map
        srcY,   // y map
        cv::INTER_LINEAR); // interpolation method
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

It is interesting to note that the two maps contain floating-point values. Consequently, a pixel in the destination image can map back to a nonintegral value (that is, a location between pixels). This is very convenient because this allows us to define the mapping function of our choice. For instance, in our remapping example, we used a `sinusoidal` function to define our transformation. However, this also means that we have to interpolate the value of virtual pixels in between real pixels. There exist different ways of performing pixel interpolation, and the last parameter of the `remap` function allows us to select the method that will be used. Pixel interpolation is an important concept in image processing; this subject will be discussed in `Chapter 6`, *Filtering the Images*.

See also

- The *There's more...* section of the *Filtering images using low-pass filters* recipe of `Chapter 6`, *Filtering the Images*, explains the concept of pixel interpolation.
- The *Calibrating a camera* recipe of `Chapter 9`, *Describing and Matching Interest Points*, uses remapping to correct lens distortions in an image.
- The *Computing a homography between two images* recipe of `Chapter 9`, *Describing and Matching Interest Points*, uses perspective image warping to build an image panorama.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3
Processing Color Images with Classes

Good computer vision programs begin with good programming practices. Building a bug-free application is just the beginning. What you really want is an application that you, and the programmers working with you, will be able to adapt and evolve easily as new requirements come in. This chapter will show you how to make the best use of some object-oriented programming principles in order to build good-quality software programs. In particular, we will introduce a few important design patterns that will help you build applications with components that are easy to test, maintain, and reuse.

Design patterns are a well-known concept in software engineering. Basically, a design pattern is a sound, reusable solution to a generic problem that occurs frequently in software designing. Many software patterns have been introduced and documented. Good programmers should build a working knowledge of these existing patterns.

This chapter also has a secondary objective. It will teach you how to play with image colors. The example used throughout this chapter will show you how to detect the pixels of a given color, and the last two recipes will explain how to work with different color spaces.

In this chapter, we will cover the following recipes:

- Comparing colors using the strategy design pattern
- Segmenting an image with the GrabCut algorithm
- Converting color representations
- Representing colors with hue, saturation, and brightness

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Comparing colors using the strategy design pattern

Let's suppose that we want to build a simple algorithm that will identify all of the pixels in an image that have a given color. For this, the algorithm has to accept an image and a color as input and return a binary image showing the pixels that have the specified color. The tolerance with which we want to accept a color will be another parameter to be specified before running the algorithm.

In order to accomplish this objective, this recipe will use the **strategy design pattern**. This object-oriented design pattern constitutes an excellent way of encapsulating an algorithm in a class. It is easier than replacing a given algorithm with another one or chaining several algorithms together in order to build a more complex process. In addition, this pattern facilitates the deployment of an algorithm by hiding as much of its complexity as possible behind an intuitive programming interface.

How to do it...

Once an algorithm has been encapsulated in a class using the strategy design pattern, it can be deployed by creating an instance of that class. Typically, the instance will be created when the program is initialized. At the time of construction, the class instance will initialize the different parameters of the algorithm with their default values, such that it will immediately be ready to be used. The algorithm's parameter values can also be read and set by using appropriate methods. In the case of an application with a GUI, these parameters can be displayed and modified using different widgets (text fields, sliders, and so on) so that a user can play with them easily.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will show you the structure of a `Strategy` class in the next section; let's start with an example of how it can be deployed and used. Let's write a simple `main` function that will run our proposed color detection algorithm:

1. First, in the `main` function, we have to create an instance of our created class that we will cover in the next section:

```
int main()
{
   // 1. Create image processor object
   ColorDetector cdetect;
```

2. Read an image to process:

```
   // 2. Read input image
   cv::Mat image= cv::imread("boldt.jpg");
```

3. Check with the `empty` function whether we have loaded the image correctly; if it's empty, we must exit from our application:

```
   if (image.empty())
      return 0;
```

4. Use the new instance of `ColorDetector` to set the target color. This function is defined in the class that we will cover in the next section:

```
// 3. Set input parameters
   cdetect.setTargetColor(230,190,130); // here blue sky
```

5. Create a window to show the result of the image `process` with our instance. To do it, we have to use the function process of our instance:

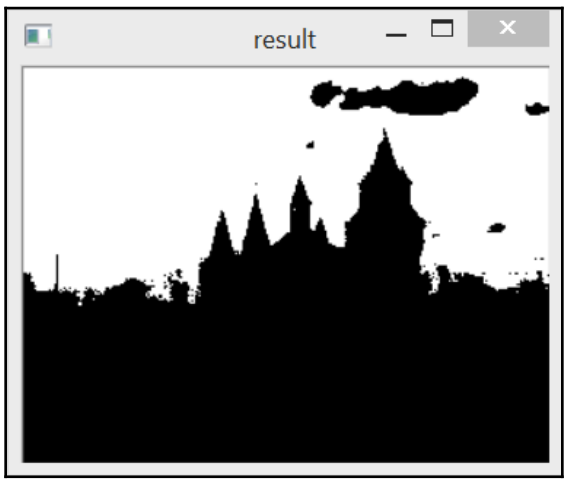
```
   cv::namedWindow("result");

   // 4. Process the image and display the result
   cv::imshow("result",cdetect.process(image));
```

6. Finally, wait for a key press before exiting:

```
cv::waitKey();

   return 0;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Running this program to detect a blue sky in the color version of the castle image presented in `Chapter 2`, *Manipulating the Pixels*, produces the following output:



Here, a white pixel indicates a positive detection of the sought color, and black indicates a negative detection.

Obviously, the algorithm we encapsulated in this class is relatively simple (as you will see next, it is composed of just one scanning loop and one tolerance parameter). The strategy design pattern becomes really powerful when the algorithm to be implemented is more complex, has many steps, and includes several parameters.

How it works...

The core process of this algorithm is easy to build. It is a simple scanning loop that goes over each pixel, comparing its color with the target color. Using what we learned in the *Scanning an image with iterators* recipe of `Chapter 2`, *Manipulating the Pixels*, this loop can be written as follows:

```
// get the iterators
cv::Mat_<cv::Vec3b>::const_iterator it=
                        image.begin<cv::Vec3b>();
cv::Mat_<cv::Vec3b>::const_iterator itend=
                        image.end<cv::Vec3b>();
cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

// for each pixel
for ( ; it!= itend; ++it, ++itout) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            // compute distance from target color
            if (getDistanceToTargetColor(*it)<=maxDist) {
                  *itout= 255;
            } else {
                 *itout= 0;
            }
      }
```

The `cv::Mat` variable's `image` refers to the input image, while `result` refers to the binary output image. Therefore, the first step consists of setting up the required iterators. The scanning loop then becomes easy to implement. The distance between the current pixel color and the target color is evaluated on each iteration, in order to check whether it is within the tolerance parameter defined by `maxDist`. If that is the case, the value `255` (white) is then assigned to the output image; if not, `0` (black) is assigned. To compute the distance to the target color, the `getDistanceToTargetColor` method is used. There are different ways to compute this distance. One could, for example, calculate the Euclidean distance between the three vectors that contain the RGB color values. To keep this computation simple, we simply sum the absolute differences of the RGB values (this is also known as the city-block distance) in our case. Note that in modern architecture, a floating-point Euclidean distance can be faster to compute than a simple city-block distance; this is also something to take into consideration in your design. Also, for more flexibility, we write the `getDistanceToTargetColor` method in terms of a `getColorDistance` method, as follows:

```
// Computes the distance from target color.
int getDistanceToTargetColor(const cv::Vec3b& color) const {
  return getColorDistance(color, target);
}

// Computes the city-block distance between two colors.
   int getColorDistance(const cv::Vec3b& color1,
                        const cv::Vec3b& color2) const {
  return abs(color1[0]-color2[0])+
               abs(color1[1]-color2[1])+
               abs(color1[2]-color2[2]);
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note how we used `cv::Vec3d` to hold the three unsigned characters that represent the RGB values of a color. The `target` variable obviously refers to the specified target color, and as you will see, it is defined as a `class` variable in the class algorithm that we will define. Now, let's complete the definition of the processing method. Users will provide an input image, and the result will be returned once the image scanning has completed:

```
cv::Mat ColorDetector::process(const cv::Mat &image) {

     // re-allocate binary map if necessary
     // same size as input image, but 1-channel
     result.create(image.size(),CV_8U);
     // processing loop above goes here
      ...

     return result;
}
```

Each time this method is called, it is important to check if the output image that contains the resulting binary map needs to be reallocated to fit the size of the input image. This is why we use the `create` method of `cv::Mat`. Remember that this method will only proceed to reallocation if the specified size and depth do not correspond to the current image structure.

Now that we have defined the core processing method, let's see what additional methods should be added in order to deploy this algorithm. We previously determined what input and output data our algorithm requires. Therefore, we will first define the class attributes that will hold this data:

```
class ColorDetector {

  private:

     // minimum acceptable distance
     int maxDist;

     // target color
     cv::Vec3b target;

     // image containing resulting binary map
     cv::Mat result;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In order to create an instance of the class that encapsulates our algorithm (which we have named `ColorDetector`), we need to define a constructor. Remember that one of the objectives of the strategy design pattern is to make algorithm deployment as easy as possible. The simplest constructor that can be defined is an empty one. It will create an instance of the class algorithm in a valid state. We then want the constructor to initialize all the input parameters to their default values (or the values that are generally known to give a good result). In our case, we decided that a distance of `100` is generally an acceptable tolerance parameter. We also set the default target color. We chose black, for no particular reason. The idea is to make sure we always start with predictable and valid input values:

```
// empty constructor
// default parameter initialization here
ColorDetector() : maxDist(100), target(0,0,0) {}
```

At this point, a user who creates an instance of our class algorithm can immediately call the `process` method with a valid image and obtain a valid output. This is another objective of the strategy pattern, that is, to make sure that the algorithm always runs with valid parameters. Obviously, the users of this class will want to use their own settings. This is done by providing the user with the appropriate getters and setters. Let's start with the `color` tolerance parameter:

```
// Sets the color distance threshold.
// Threshold must be positive,
// otherwise distance threshold is set to 0.
void setColorDistanceThreshold(int distance) {

   if (distance<0)
      distance=0;
   maxDist= distance;
}

// Gets the color distance threshold
int getColorDistanceThreshold() const {

   return maxDist;
}
```

Note how we first check the validity of the input. Again, this is to make sure that our algorithm will never be run in an invalid state. The target color can be set in a similar manner, as follows:

```
// Sets the color to be detected
void setTargetColor(uchar blue,
                    uchar green,
                    uchar red) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
      // BGR order
            target = cv::Vec3b(blue, green, red);
       }

       // Sets the color to be detected
       void setTargetColor(cv::Vec3b color) {

          target= color;
       }

       // Gets the color to be detected
       cv::Vec3b getTargetColor() const {

          return target;
       }
```

This time, it is interesting to note that we have provided the user with two definitions of the `setTargetColor` method. In the first version of the definition, the three color components are specified as three arguments, while in the second version, `cv::Vec3b` is used to hold the color values. Again, the objective is to facilitate the use of our class algorithm. The user can simply select the setter that best fits their needs.

There's more...

This recipe introduced you to the idea of encapsulating an algorithm in a class using the strategy design pattern. The example algorithm used in this recipe consisted of identifying the pixels of an image that have a color sufficiently close to a specified target color. This computation could have been done otherwise. Also, the implementation of a strategy design pattern could be complemented with function objects.

Computing the distance between two color vectors

To compute the distance between two color vectors, we used the following simple formula:

```
return abs(color[0]-target[0])+
       abs(color[1]-target[1])+
       abs(color[2]-target[2]);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

However, OpenCV includes a function to compute the Euclidean norm of a vector. Consequently, we could have computed our distance as follows:

```
return static_cast<int>(
   cv::norm<int,3>(cv::Vec3i(color[0]-target[0],
                             color[1]-target[1],
                             color[2]-target[2])));
```

A very similar result would then be obtained, using this definition of the `getDistance` method. Here, we use `cv::Vec3i` (a three-vector array of integers), because the result of the subtraction is an integer value.

It is also interesting to recall from `Chapter 2`, *Manipulating the Pixels*, that the OpenCV matrix and vector data structures include a definition of the basic arithmetic operators. Consequently, one could have proposed the following definition for distance computation:

```
return static_cast<int>(
   cv::norm<uchar,3>(color-target)); // wrong!
```

This definition may look right at first glance; however, it is wrong. This is because all these operators always include a call to `saturate_cast` (see the *Scanning an image with neighbor access* recipe in `Chapter 2`, *Manipulating the Pixels*) in order to ensure that the results stay within the domain of the input type (here, it is `uchar`). Therefore, in the cases where the target value is greater than the corresponding color value, the value `0` will be assigned instead of the negative value that one would have expected. A correct formulation would then be as follows:

```
cv::Vec3b dist;
cv::absdiff(color,target,dist);
return cv::sum(dist)[0];
```

However, using two function calls to compute the distance between two three-vector arrays is inefficient.

Using OpenCV functions

In this recipe, we used a loop with iterators in order to perform our computation. Alternatively, we could have achieved the same result by calling a sequence of OpenCV functions. The color detection method would then be written as follows:

```
cv::Mat ColorDetector::process(const cv::Mat &image) {
       cv::Mat output;
       // compute absolute difference with target color
       cv::absdiff(image,cv::Scalar(target),output);
       // split the channels into 3 images
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        std::vector<cv::Mat> images;
        cv::split(output,images);
        // add the 3 channels (saturation might occurs here)
        output= images[0]+images[1]+images[2];
        // apply threshold
        cv::threshold(output,  // input image
                      output,  // output image
                      maxDist, // threshold (must be < 256)
                      255,     // max value
        cv::THRESH_BINARY_INV); // thresholding mode
        return output;
    }
```

This method uses the `absdiff` function that computes the absolute difference between the pixels of an image and, in this case, a scalar value. Instead of a scalar value, another image can be provided as the second argument to this function. In the latter case, a pixel-by-pixel difference will be applied; consequently, the two images must be of the same size. The individual channels of the different image are then extracted using the `split` function (which is discussed in the *There's more...* section of the *Performing simple image arithmetic* recipe of `Chapter 2`, *Manipulating the Pixels*), in order to be able to add them together. It is important to note that the result of this sum may sometimes be greater than `255`, but because saturation is always applied, the result will be stopped at `255`. The consequence is that with this version, the `maxDist` parameter must also be less than `256`; this should be corrected if you consider this behavior unacceptable. The last step is to create a binary image by using the threshold function. This function is commonly used to compare all the pixels with a threshold value (the third parameter), and in the regular thresholding mode (`cv::THRESH_BINARY`), it assigns the defined maximum value (the fourth parameter) to all the pixels greater than the threshold and `0`. Here, we used the inverse mode (`cv::THRESH_BINARY_INV`), in which the defined maximum value is assigned to the pixels that have a value lower than, or equal to, the threshold. The `cv::THRESH_TOZERO_INV` and `cv::THRESH_TOZERO_INV` modes, which leave the pixels greater than or lower than the threshold unchanged, are also of interest.

Using OpenCV functions is always a good idea. You can quickly build complex applications and potentially reduce the number of bugs. The result is often more efficient (thanks to the optimization efforts invested by the OpenCV contributors). However, when many intermediate steps are performed, you may find that the resulting method consumes more memory.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The functor or function object

Using the C++ operator overloading, it is possible to create a class for which its instances behave like functions. The idea is to overload the `operator()` method, such that a call to the processing method of a class behaves exactly like a simple function call. The resulting class instance is called a **function object**, or a **functor**. Often, a functor includes a full constructor, such that it can be used immediately after being created. For example, you can add the following constructor to your `ColorDetector` class:

```
  // full constructor
  ColorDetector(uchar blue, uchar green, uchar red,
                int maxDist=100): maxDist(maxDist) {

  // target color
  setTargetColor(blue, green, red);
}
```

Obviously, you can still use the setters and getters that were defined previously. The functor method can be defined as follows:

```
  cv::Mat operator()(const cv::Mat &image) {

    // color detection code here ...
  }
```

To detect a given color with this functor method, simply write the following code snippet:

```
ColorDetector colordetector(230,190,130,  // color
    100); // threshold
  cv::Mat result= colordetector(image);   // functor call
```

As you can see, the call to the color detection method now looks like a function call. As a matter of fact, the `colordetector` variable can be used as if it were the name of a function.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The OpenCV base class for algorithms

OpenCV offers many algorithms that perform various computer vision tasks. To facilitate their use, most of these algorithms have been made a subclass of a generic base class called `cv::Algorithm`. This one implements some of the concepts dictated by the strategy design pattern. First, all of these algorithms are created dynamically, using a specialized static method that makes sure that the algorithm is always created in a valid state (that is, with valid default values for the unspecified parameters). Let's consider, for example, one of these subclasses, `cv::ORB`; this one is an interest point operator that will be discussed in the *Detecting FAST features at multiple scales* recipe, in `Chapter 8`, *Detecting Interest Points.* Here, we simply use it as an illustrative example of an algorithm.

An instance of this algorithm is therefore created as follows:

```
cv::Ptr<cv::ORB> ptrORB = cv::ORB::create(); // default state
```

Once it's created, the algorithm can be used. For example, the generic methods `read` and `write` can be used to load or store the state of the algorithm. The algorithms also have specialized methods (in the case of ORB, for example, the methods `detect` and `compute` can be used to trigger its main computational units). Algorithms also have specialized setter methods that allow for the specifying of their internal parameters. Note that we could have declared the pointer as `cv::Ptr<cv::Algorithm>`, but in this case, we would not be able to use its specialized methods.

See also

- The policy-based class design, introduced by A. Alexandrescu, is an interesting variant of the strategy design pattern, in which algorithms are selected at compile time
- The *Design Patterns: Elements of Reusable Object-Oriented Software* book, by Erich Gammaet, et al., Addison-Wesley, 1994, is one of the classic books on the subject
- The *Converting color representation* recipe introduces the concept of using perceptually uniform color spaces to achieve a more intuitive color comparison

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

You've successfully learned how to compare colors using the strategy design pattern. Now, let's move on to the next recipe.

Segmenting an image with the GrabCut algorithm

The previous recipe showed how color information can be useful to segment an image into an area corresponding to specific elements of a scene. Objects often have distinctive colors, and these ones can often be extracted by identifying areas of similar colors. OpenCV proposes an implementation of a popular algorithm for image segmentation—the **GrabCut** algorithm. GrabCut is a complex and computationally expensive algorithm, but it generally produces very accurate results. It is the best algorithm to use when you want to extract a foreground object in a still image (for example, to cut and paste an object from one picture to another).

How to do it...

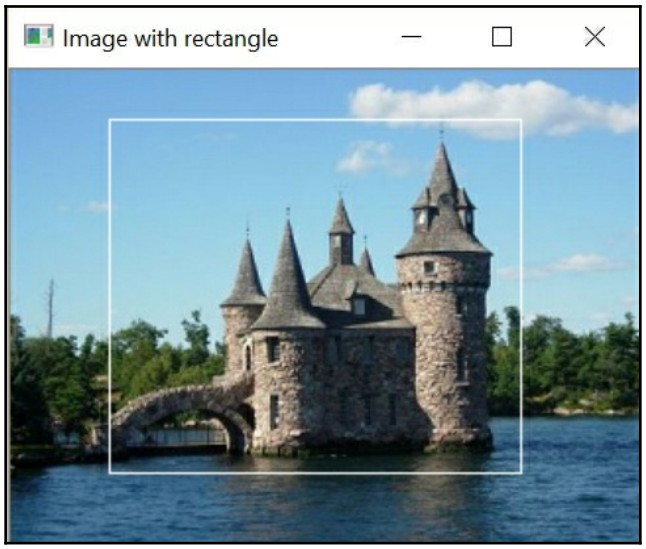
The `cv::grabCut` function is easy to use. You just need to input an image and label some of its pixels as belonging to the background or to the foreground. Based on this partial labeling, the algorithm will then determine a foreground/background segmentation for the complete image, as shown in the following steps:

1. One way to specify a partial foreground/background labeling for an input image is to define a rectangle inside which the foreground object is included:

```
// define bounding rectangle
// the pixels outside this rectangle
// will be labeled as background
cv::Rect rectangle(5,70,260,120);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This defines the following area in the image:



All the pixels outside this rectangle will then be marked as background, in addition to the input image and its segmentation image.

2. Calling the `cv::grabCut` function requires the definition of two matrices, which will contain the models built by the algorithm, as follows:

```
cv::Mat result; // segmentation (4 possible values)
// GrabCut segmentation
cv::grabCut(image, // input image
   result, // segmentation result
   rectangle,// rectangle containing foreground
   bgModel, fgModel, // models
   5, // number of iterations
   cv::GC_INIT_WITH_RECT); // use rectangle
```

Note how we specified that we are using the bounding rectangle mode with the `cv::GC_INIT_WITH_RECT` flag as the last argument of the function (the next section, *How it works...*, will discuss the other available mode).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The input/output segmentation image can have one of the following four values:

   - `cv::GC_BGD`: This is the value of the pixels that certainly belong to the background (for example, pixels outside the rectangle, in our example)
   - `cv::GC_FGD`: This is the value of the pixels that certainly belong to the foreground (there are none in our example)
   - `cv::GC_PR_BGD`: This is the value of the pixels that probably belong to the background
   - `cv::GC_PR_FGD`: This is the value of the pixels that probably belong to the foreground (that is, the initial value of the pixels inside the rectangle in our example)

4. We get a binary image of the segmentation by extracting the pixels that have a value equal to `cv::GC_PR_FGD`. This is accomplished with the following code:

```
// Get the pixels marked as likely foreground
cv::compare(result,cv::GC_PR_FGD,result,cv::CMP_EQ);
// Generate output image
cv::Mat foreground(image.size(),CV_8UC3,cv::Scalar(255,255,255));
image.copyTo(foreground,// bg pixels are notcopied
 result);
```

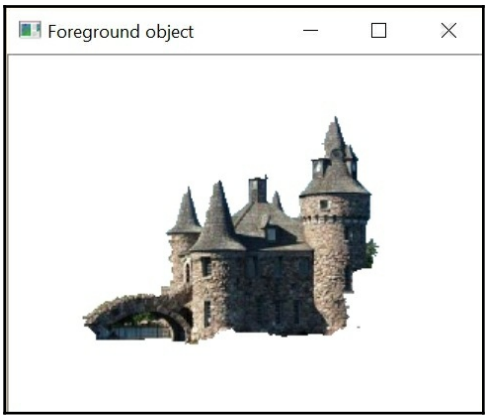
5. To extract all the foreground pixels, that is, with values equal to `cv::GC_PR_FGD` or `cv::GC_FGD`, it is possible to check the value of the first bit, as follows:

```
// checking first bit with bitwise-and
 result= result&1; // will be 1 if FG
```

   This is possible because these constants are defined as the values `1` and `3`, while the other two (`cv::GC_BGD` and `cv::GC_PR_BGD`) are defined as `0` and `2`. In our example, the same result is obtained because the segmentation image does not contain the `cv::GC_FGD` pixels (only the `cv::GC_BGD` pixels have been included).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The following image is then obtained:



Now, let's go behind the scenes to understand the code better.

How it works...

In the preceding example, the GrabCut algorithm was able to extract the foreground object by simply specifying a rectangle inside which the object of interest (the castle) was contained. Alternatively, one could also assign the values `cv::GC_BGD` and `cv::GC_FGD` to some specific pixels of the input image, which are provided by using a mask image as the second argument of the `cv::grabCut` function. You would then specify `GC_INIT_WITH_MASK` as the input mode flag. These input labels could be obtained, for example, by asking a user to mark a few elements of the image interactively. It is also possible to combine these two input modes.

Using this input information, the GrabCut algorithm creates the background/foreground segmentation by proceeding as follows. Initially, a foreground label (`cv::GC_PR_FGD`) is tentatively assigned to all the unmarked pixels. Based on the current classification, the algorithm groups the pixels into clusters of similar colors (that is, `K` clusters for the background and `K` clusters for the foreground). The next step is to determine a background/foreground segmentation by introducing boundaries between the foreground and background pixels. This is done through an optimization process that tries to connect pixels with similar labels, and that imposes a penalty for placing a boundary in the regions of relatively uniform intensity. This optimization problem can be solved efficiently using the Graph Cuts algorithm, a method that can find the optimal solution for a problem by representing it as a connected graph on which cuts are applied in order to compose an optimal configuration. The obtained segmentation produces new labels for the pixels.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The clustering process can then be repeated, and a new optimal segmentation found again, and so on. Therefore, the GrabCut algorithm is an iterative procedure that gradually improves the segmentation result. Depending on the complexity of the scene, a good solution can be found in higher or lower numbers of iterations (in easy cases, one iteration would be enough).

This explains the argument of the function where the user can specify the number of iterations to be applied. The two internal models maintained by the algorithm are passed as an argument of the function (and returned). Therefore, it is possible to call the function with the models of the last run again if one wishes to improve the segmentation result by performing additional iterations.

See also

- The article *GrabCut: Interactive Foreground Extraction Using Iterated Graph Cuts* in *ACM Transactions on Graphics (SIGGRAPH), Volume 23, Issue 3*, August 2004, by C. Rother, V. Kolmogorov, and A. Blake, describes the GrabCut algorithm in detail
- The *Segmenting images using watersheds* recipe in `Chapter 5`, *Transforming Images with Morphological Operations*, presents another image segmentation algorithm

You've successfully learned how to segment an image with the GrabCut algorithm. Now, let's move on to the next recipe.

Converting color representations

The earlier recipes taught you how to encapsulate an algorithm into a class. This way, the algorithm becomes easier to use through a simplified interface. Encapsulation also permits you to modify an algorithm's implementation without impacting the classes that use it. This principle is illustrated in this recipe, where we will modify the `ColorDetector` class algorithm in order to use another color space. Therefore, this recipe will also present an opportunity to introduce color conversions with OpenCV.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

The RGB color space is based on the use of the red, green, and blue additive primary colors. These have been selected because when they are combined together, they can produce a wide gamut of different colors. In fact, the human visual system is also based on the trichromatic perception of colors, with cone cell sensitivity located around the red, green, and blue spectrum. It is often the default color space in digital imagery because that is the way they are acquired. Captured light goes through the red, green, and blue filters. Additionally, in digital images, the red, green, and blue channels are adjusted such that when combined in equal amounts, a gray-level intensity is obtained, that is, from black `(0,0,0)` to white `(255,255,255)`.

Unfortunately, computing the distance between the colors using the RGB color space is not the best way to measure the similarity between two given colors. Indeed, RGB is not a perceptually uniform color space. This means that two colors at a given distance might look very similar, while two other colors separated by the same distance might look very different.

To solve this problem, other color representations that have the property of being perceptually uniform have been introduced. In particular, the CIE L*a*b* color space is one such color model. By converting our images to this representation, the Euclidean distance between an image pixel and the target color will then be a meaningful measure of the visual similarity between the two colors. In this recipe, we will show you how to modify the previous application in order to work with CIE L*a*b*.

How to do it...

The conversion of images between different color spaces is easily done through the use of the `cv::cvtColor` OpenCV function, as follows:

1. Let's convert the input image to the CIE L*a*b* color space at the beginning of the `process` method:

```
cv::Mat ColorDetector::process(const cv::Mat &image) {

     // re-allocate binary map if necessary
     // same size as input image, but 1-channel
     result.create(image.rows,image.cols,CV_8U);


     // Converting to Lab color space
     cv::cvtColor(image, converted, cv::COLOR_BGR2Lab);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
// get the iterators of the converted image
cv::Mat_<cv::Vec3b>::iterator it=
            converted.begin<cv::Vec3b>();
cv::Mat_<cv::Vec3b>::iterator itend=
            converted.end<cv::Vec3b>();
// get the iterator of the output image
cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

// for each pixel
for ( ; it!= itend; ++it, ++itout) {
...
```

2. The `converted` variable contains the image after color conversion. In the `ColorDetector` class, it is defined as a class attribute:

```
class ColorDetector {

  private:
     // image containing color converted image
     cv::Mat converted;
```

3. You also need to convert the input target color. You can do this by creating a temporary image that contains only one pixel. Note that you need to keep the same signature as in the earlier recipes; that is, the user continues to supply the target color in RGB:

```
// Sets the color to be detected
void setTargetColor(unsigned char red,
      unsigned char green, unsigned char blue) {

    // Temporary 1-pixel image
    cv::Mat tmp(1,1,CV_8UC3);
    tmp.at<cv::Vec3b>(0,0)= cv::Vec3b(blue, green, red);

    // Converting the target to Lab color space
    cv::cvtColor(tmp, tmp, cv::COLOR_BGR2Lab);

    target= tmp.at<cv::Vec3b>(0,0);
}
```

If the application of the preceding recipe is compiled with this modified class, it will now detect the pixels of the target color using the CIE L*a*b* color model.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

When an image is converted from one color space to another, a linear or nonlinear transformation is applied on each input pixel to produce the output pixels. The pixel type of the output image will match one of the input images. Even if you work with 8-bit pixels most of the time, you can also use a color conversion with floating-point images (in which case, the pixel values are generally assumed to vary between `0` and `1.0`) or with integer images (with pixels generally varying between `0` and `65535`). However, the exact domain of the pixel values depends on the specific color space and destination image type. For example, with the CIE L*a*b* color space, the `L` channel, which represents the brightness of each pixel, varies between `0` and `100`, and it is rescaled between `0` and `255` in the cases of 8-bit images. The `a` and `b` channels correspond to the chromaticity components. These channels contain information about the color of a pixel, independent of its brightness. Their values vary between `-127` and `127`; for 8-bit images, `128` is added to each value in order to make it fit within the `0` to `255` interval. However, note that the 8-bit color conversion will introduce rounding errors that will make the transformation imperfectly reversible.

Most commonly used color spaces are available. It is just a question of providing the right color space conversion code to the OpenCV function (for CIE L*a*b*, this code is `cv::COLOR_BGR2Lab`). Among these is YCrCb, which is the color space used in JPEG compression. To convert a color space from BGR to YCrCb, the code will be `cv::COLOR_BGR2YCrCb`. Note that all the conversions that involve the three regular primary colors, red, green, and blue, are available in the RGB and BGR order.

The *CIE L*u*v** color space is another perceptually uniform color space. You can convert from BGR to CIE L*u*v by using the `cv::COLOR_BGR2Luv` code. Both L*a*b* and L*u*v* use the same conversion formula for the brightness channel, but use a different representation for the chromaticity channels. Also, note that since these two color spaces distort the RGB color domain in order to make it perceptually uniform, these transformations are nonlinear (therefore, they are costly to compute).

There is also the CIE XYZ color space (converted using the `cv::COLOR_BGR2XYV` code). It is a standard color space used to represent any perceptible color in a device-independent way. This space is not usually directly used while processing. In the computation of the L*u*v and L*a*b color spaces, the XYZ color space is used as an intermediate representation. The transformation between RGB and XYZ is linear. It is also interesting to note that the `Y` channel corresponds to a gray-level version of the image.

HSV and HLS are interesting color spaces because they decompose the colors into their hue and saturation components, plus their value or luminance component, which is a more natural way for humans to describe colors.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can also convert color images to a gray-level intensity. The output will be a one-channel image:

```
cv::cvtColor(color, gray, cv::COLOR_BGR2Gray);
```

It is also possible to make conversions in another direction, but the three channels of the resulting color image will then be filled identically with the corresponding values in the gray-level image.

See also

- The *Using the mean shift algorithm to find an object* recipe, in `Chapter 4`, *Counting the Pixels with Histograms*, uses the HSV color space in order to find an object in an image.
- Many good references are available on the color space theory. Among them, the following is a complete reference: *The Structure and Properties of Color Spaces and the Representation of Color Images*, by E. Dubois, Morgan and Claypool Publishers, Oct. 2009.

You've successfully learned how to convert color representations. Now, let's move on to the next recipe.

Representing colors with hue, saturation, and brightness

In this chapter, we played with image colors. We used different color spaces and tried to identify image areas that have a specific color. The RGB color space, for instance, was considered, and although it is an effective representation for the capture and display of colors in electronic imaging systems, this representation is not very intuitive. This is not the way humans think about colors. We talk about colors in terms of their tints, brightness, or colorfulness (that is, whether a color is vivid or pastel). The **phenomenal color spaces**, based on the concepts of hue, saturation, and brightness, were introduced to help users to specify the colors using properties that are more intuitive to them. In this recipe, we will explore the concepts of hue, saturation, and brightness as a means to describe colors.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

The conversion of a BGR image into a phenomenal color space is done using the `cv::cvtColor` function that was explored in the previous recipe. The following steps will help us conduct the conversion:

1. Here, we will use the `cv::COLOR_BGR2HSV` conversion code:

```
// convert into HSV space
cv::Mat hsv;
cv::cvtColor(image, hsv, cv::COLOR_BGR2HSV);
```

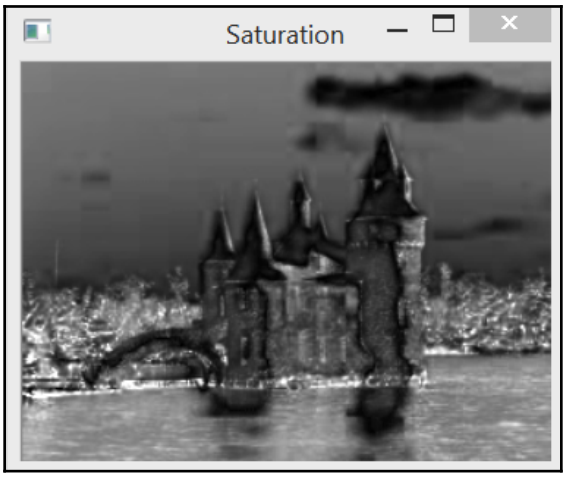
2. We can go back to the BGR space by using the `cv::COLOR_BGR2HSV` code. We can visualize each of the HSV components by splitting the converted image channels into three independent images, as follows:

```
// split the 3 channels into 3 images
std::vector<cv::Mat> channels;
cv::split(hsv,channels);
// channels[0] is the Hue
// channels[1] is the Saturation
// channels[2] is the Value
```

Since we are working on 8-bit images, OpenCV rescales the channel values to cover the `0` to `255` range (except for the hue, which is rescaled between `0` and `180`, as will be explained in the next section). This is very convenient, as we are able to display these channels as gray-level images. The value channel of the castle image will then look as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The same image in the saturation channel will look as follows:



Finally, the image in the hue channel looks as follows:

These images will be interpreted in the next section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

The phenomenal color spaces have been introduced because they correspond to the way humans tend to organize colors naturally. Indeed, humans prefer to describe colors with intuitive attributes, such as tints, colorfulness, and brightness. These three attributes are the basis of most phenomenal color spaces. **Hue** designates the dominant color; the names that we give to colors (such as green, yellow, blue, and red) correspond to the different hue values. **Saturation** tells us how vivid the color is; pastel colors have low saturation, while the colors of the rainbow are highly saturated. Finally, **brightness** is a subjective attribute that refers to the luminosity of a color. Other phenomenal color spaces use the concept of color **value** or color **lightness** as a way to characterize the relative color intensity.

These color components try to mimic the intuitive human perception of colors. In consequence, there is no standard definition for them. In the literature, you will find several different definitions and formulas for hue, saturation, and brightness. OpenCV proposes two implementations of phenomenal color spaces: the HSV and the HLS color spaces. The conversion formulas are slightly different, but they give very similar results.

The value component is probably the easiest to interpret. In the OpenCV implementation of the HSV space, it is defined as the maximum value of the three BGR components. It is a very simplistic implementation of the brightness concept. For a definition that matches the human visual system better, you should use the `L` channel of the L*a*b* or L*u*v* color spaces.

To compute the saturation, OpenCV uses a formula based on the minimum and maximum values of the BGR components:

$$S = (max(R,G,B) - min(R,G,B))/max(R,G,B)$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The idea is that a grayscale color in which the three R, G, and B components are all equal will correspond to a perfectly desaturated color; therefore, it will have a saturation value of `0`. Saturation is then a value between `0` and `1.0`. For 8-bit images, saturation is rescaled to a value between `0` and `255`, and when displayed as a gray-level image, brighter areas correspond to the colors that have a higher saturation. For example, from the saturation image in the previous section, it can be seen that the blue of the water is more saturated than the light blue pastel color of the sky, as expected. The different shades of gray have, by definition, a saturation value equal to zero (because in this case, all the three BGR components are equal). This can be observed on the different roofs of the castle, which are made of dark gray stone. Finally, in the saturation image, you may have noticed some white spots located at areas that correspond to very dark regions of the original image. These are a consequence of the definition of saturation used. Indeed, because saturation only measures the relative difference between the maximum and minimum BGR values, a triplet such as `(1,0,0)` gives a perfect saturation of `1.0`, even if this color would be seen as black. Consequently, the saturation values measured at dark regions are unreliable and should not be considered.

The hue of a color is generally represented by an angle value between `0` and `360`, with the red color at `0` degrees. In the case of an 8-bit image, OpenCV divides this angle by `2`, to fit within the single-byte range. Therefore, each hue value corresponds to a given color tint independent of its brightness and saturation. For example, both the sky and the water have the same hue value (approximately `200` degrees (intensity, `100`)), which corresponds to the blue shade; the green color of the trees in the background has a hue of around `90` degrees. It is important to note that hue is less reliable when evaluated for colors that have very low saturation.

The HSB color space is often represented by a cone, where each point inside corresponds to a particular color. The angular position corresponds to the hue of the color, the saturation is the distance from the central axis, and the brightness is given by the height. The tip of the cone corresponds to the black color for which the hue and saturation are undefined:

$$S = \frac{max(R,G,B) - min(R,G,B)}{max(R,G,B)}$$

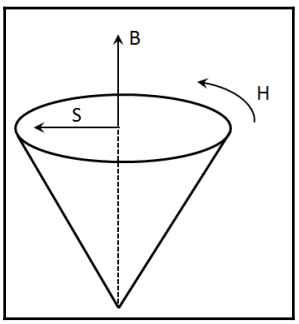
Interesting effects can be created by playing with the HSV values. Several color effects that can be created using photo-editing software are accomplished by this color space. For example, you may decide to modify an image by assigning a constant brightness to all the pixels of an image without changing the hue and saturation. This can be done as follows:

```
// convert into HSV space
cv::Mat hsv;
cv::cvtColor(image, hsv, cv::COLOR_BGR2HSV);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
// split the 3 channels into 3 images
std::vector<cv::Mat> channels;
cv::split(hsv,channels);
// Value channel will be 255 for all pixels
channels[2]= 255;
// merge back the channels
cv::merge(channels,hsv);
// reconvert to BGR
cv::Mat newImage;
cv::cvtColor(hsv,newImage,cv::COLOR_HSV2BGR);
```

This gives the following screenshot, which now looks like a drawing (see the book's graphics bundle to view this image in color):



The next section will give a few more topics you can take a look at while working on this recipe.

There's more...

The HSV color space can also be very convenient to use when you want to look for objects of specific colors.

Using colors for detection – skin tone detection

Color information can be very useful for the initial detection of specific objects. For example, the detection of road signs in a driver-assistance application could rely on the colors of standard signs in order to extract potential road sign candidates quickly. The detection of skin color is another example in which the detected skin regions could be used as an indicator of the presence of a human in an image; this approach is very often used in gesture recognition, where skin tone detection is used to detect hand positions.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In general, to detect an object using color, you need to first collect a large database of image samples that contain the object captured from different viewing conditions. These will be used to define the parameters of your classifier. You also need to select the color representation that you will use for classification. For skin tone detection, many studies have shown that skin color from the diversity of ethnic groups clusters well in the hue-saturation space. For this reason, we will simply use the hue and saturation values to identify the skin tones in the following image (see the book's graphics bundle to view this image in color):

Therefore, we have defined a function that classifies the pixels of an image as skin or non-skin, simply based on an interval of values (the minimum and maximum hues, and the minimum and maximum saturation values):

```
void detectHScolor(const cv::Mat& image,  // input image
  double minHue, double maxHue,  // Hue interval
  double minSat, double maxSat,  // saturation interval
  cv::Mat& mask) {               // output mask

  // convert into HSV space
  cv::Mat hsv;
  cv::cvtColor(image, hsv, cv::COLOR_BGR2HSV);

  // split the 3 channels into 3 images
  std::vector<cv::Mat> channels;
  cv::split(hsv, channels);
  // channels[0] is the Hue
  // channels[1] is the Saturation
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

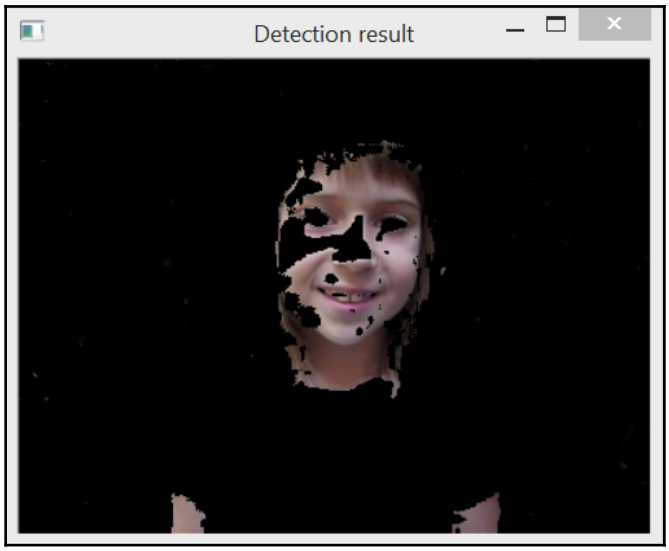
```
    // channels[2] is the Value

    // Hue masking
    cv::Mat mask1; // under maxHue
    cv::threshold(channels[0], mask1, maxHue, 255,
    cv::THRESH_BINARY_INV);
    cv::Mat mask2; // over minHue
    cv::threshold(channels[0], mask2, minHue, 255,
    cv::THRESH_BINARY);

    cv::Mat hueMask; // hue mask
    if (minHue < maxHue)
        hueMask = mask1 & mask2;
    else // if interval crosses the zero-degree axis
        hueMask = mask1 | mask2;

    // Saturation masking
    // under maxSat
    cv::threshold(channels[1], mask1, maxSat, 255,
    cv::THRESH_BINARY_INV);
    // over minSat
    cv::threshold(channels[1], mask2, minSat, 255,
    cv::THRESH_BINARY);

    cv::Mat satMask; // saturation mask
    satMask = mask1 & mask2;

    // combined mask
    mask = hueMask&satMask;
}
```

Having a large set of skin (and non-skin) samples at our disposal, we could have used a probabilistic approach for the likelihood of observing a given color in the skin class versus that of observing the same color in the non-skin class. Here, we defined an acceptable hue-saturation interval for our test image empirically (remember that the 8-bit version of the hue goes from `0` to `180` and the saturation goes from `0` to `255`):

```
    // detect skin tone
    cv::Mat mask;
    detectHScolor(image,
          160, 10, // hue from 320 degrees to 20 degrees
          25, 166, // saturation from ~0.1 to 0.65
          mask);

    // show masked image
    cv::Mat detected(image.size(), CV_8UC3, cv::Scalar(0, 0, 0));
    image.copyTo(detected, mask);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The following detection image is obtained as a result:



Note that for simplicity, we have not considered color saturation in the detection. In practice, excluding the colors with a high saturation would have reduced the possibility of the wrong detection of bright reddish colors as skin. Obviously, reliable and accurate detection of skin color would require a much more elaborate analysis that would have to be based on a large number of skin samples. It is also very difficult to guarantee good detection across different images, because many factors influence the color rendering in photography, such as white balancing and lighting conditions. Nevertheless, as shown in this chapter, only using hue information as an initial detector gives us acceptable results.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4
Counting the Pixels with Histograms

An image is composed of pixels of different values (colors). The distribution of pixel values across the image constitutes an important characteristic of this image. This chapter introduces the concept of image histograms. You will learn how to compute a histogram and how to use it to modify the image's appearance. Histograms can also be used to characterize the image's content and detect specific objects or textures in an image. Some of these techniques will be presented in this chapter.

In this chapter, we will cover the following recipes:

- Computing the image histogram
- Applying lookup tables to modify the image's appearance
- Equalizing the image histogram
- Backprojecting a histogram to detect the specific image content
- Using the mean shift algorithm to find an object
- Retrieving similar images using histogram comparison
- Counting pixels with integral images

Computing the image histogram

An image is composed of pixels and each pixel, can contain one value (one channel) that generates a gray image, or can contain three values (RGB or three channels) that generate a color image. Each channel contains values from `0` to `255` (from a black to a saturated channel, in the case of a one-channel pixel from black to white). Depending on the content of the image, you will get different amounts of each gray value.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

A **histogram** is a simple table that gives you the number of pixels that have a given value in an image (or sometimes, a set of images). The histogram of a gray-level image will, therefore, have `256` entries (or bins). Bin `0` gives you the number of pixels that have the value `0`, bin `1` gives you the number of pixels that have the value `1`, and so on. Obviously, if you sum all of the entries of a histogram, you should get the total number of pixels. Histograms can also be normalized so that the sum of the bins equals `1`. In this case, each bin gives you the percentage of pixels that have this specific value in the image.

Getting started

Computing a histogram with OpenCV can be done easily by using the `cv::calcHist` function. This is a general function that can compute the histogram of multiple channel images of any pixel value type and range. The first three recipes in this chapter will use the following image:

Now, let's look at the steps.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Here, we will make this simpler to use by specializing a class for the case of one-channel, gray-level images. For other types of images, you can always use the `cv::calcHist` function directly, which offers you all the flexibility that's required. The next section will explain each of its parameters. Then, we are going to implement our histogram class:

1. Create a `Histogram1D` class, as follows:

```
// To create histograms of gray-level images
class Histogram1D {

  private:

    int histSize[1];         // number of bins in histogram
    float hranges[2];        // range of values
    const float* ranges[1];  // pointer to the value ranges
    int channels[1];         // channel number to be examined

  public:

  Histogram1D() {

    // Prepare default arguments for 1D histogram
    histSize[0]= 256;   // 256 bins
    hranges[0]= 0.0;    // from 0 (inclusive)
    hranges[1]= 256.0;  // to 256 (exclusive)
    ranges[0]= hranges;
    channels[0]= 0;     // we look at channel 0
  }
```

2. With the defined member variables, computing a gray-level histogram can then be accomplished using the following method, which is implemented in the `Histogram1D` class:

```
// Computes the 1D histogram.
cv::Mat getHistogram(const cv::Mat &image) {

  cv::Mat hist;

  // Compute histogram
  cv::calcHist(&image,
    1,      // histogram of 1 image only
    channels,  // the channel used
    cv::Mat(),  // no mask is used
    hist,    // the resulting histogram
    1,      // it is a 1D histogram
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        histSize,  // number of bins
        ranges    // pixel value range
    );
        return hist;
}
```

3. Now, your program simply needs to open an image, create a `Histogram1D` instance, and call the `getHistogram` method:

```
// Read input image
cv::Mat image= cv::imread("group.jpg",
                          0); // open in b&w

// The histogram object
Histogram1D h;

// Compute the histogram
cv::Mat histo= h.getHistogram(image);
```

4. The `histo` object here is a simple one-dimensional array with `256` entries. Therefore, you can read each bin by simply looping over this array:

```
// Loop over each bin
for (int i=0; i<256; i++)
   cout << "Value " << i << " = " <<
               histo.at<float>(i) << endl;
```

With the image that was shown at the start of this chapter, some of the displayed values would read as follows:

```
...
Value 7 = 159
Value 8 = 208
Value 9 = 271
Value 10 = 288
Value 11 = 340
Value 12 = 418
Value 13 = 432
Value 14 = 472
Value 15 = 525
...
```

It is obviously difficult to extract any intuitive meaning from this sequence of values. For this reason, it is often convenient to display a histogram as a function, for example, using bar graphs.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Then, write the following methods to create a histogram graph:

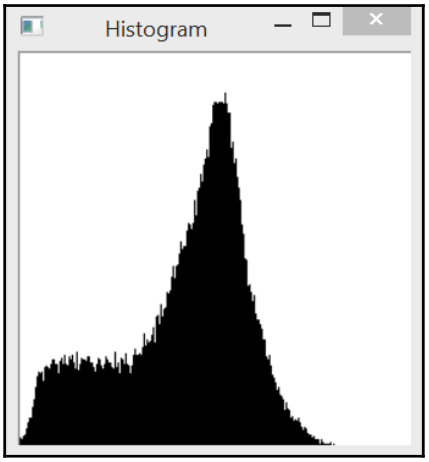
```
// Computes the 1D histogram and returns an image of it.
cv::Mat getHistogramImage(const cv::Mat &image,
                          int zoom=1){

  // Compute histogram first
  cv::Mat hist= getHistogram(image);

  // Creates image
  return getImageOfHistogram(hist, zoom);
}

// Create an image representing a histogram (static method)
cv::Mat Histogram1D::getImageOfHistogram(const cv::Mat &hist,
                                         int zoom) {
  // Get min and max bin values
  double maxVal = 0;
  double minVal = 0;
  cv::minMaxLoc(hist, &minVal, &maxVal, 0, 0);

  // get histogram size
  int histSize = hist.rows;

  // Square image on which to display histogram
  cv::Mat histImg(histSize*zoom,
                  histSize*zoom, CV_8U, cv::Scalar(255));

  // set highest point at 90% of nbins (i.e. image height)
  int hpt = static_cast<int>(0.9*histSize);

  // Draw vertical line for each bin
  for (int h = 0; h < histSize; h++) {

    float binVal = hist.at<float>(h);
    if (binVal>0) {
      int intensity = static_cast<int>(binVal*hpt / maxVal);
      cv::line(histImg, cv::Point(h*zoom, histSize*zoom),
        cv::Point(h*zoom, (histSize - intensity)*zoom),
             cv::Scalar(0), zoom);
    }
  }

  return histImg;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Using the `getImageOfHistogram` method, you can obtain an image of the histogram function in the form of a bar graph that is drawn using lines:

```
// Display a histogram as an image
cv::namedWindow("Histogram");
cv::imshow("Histogram", h.getHistogramImage(image));
```

The result is the following image:



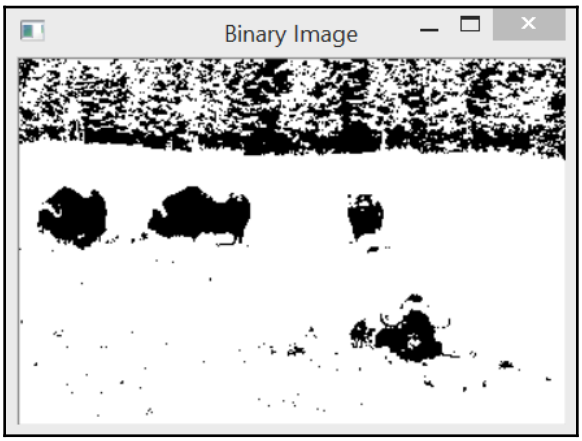
From the preceding histogram, it can be seen that the image exhibits a large peak of mid-gray-level values and a good quantity of darker pixels. Coincidentally, these two groups mostly correspond to the background and foreground of the image, respectively. This can be verified by thresholding the image at the transition between these two groups. A convenient OpenCV function can be used for this, namely, the `cv::threshold` function that was introduced in `Chapter 3`, *Processing Color Images with Classes*.

7. Here, to create our binary image, we threshold the image at the minimum value just before it increases toward the high peak of the histogram (gray value `60`):

```
cv::Mat thresholded; // output binary image
cv::threshold(image,thresholded,
            60,    // threshold value
          255,   // value assigned to
                    // pixels over threshold value
          cv::THRESH_BINARY); // thresholding type
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The resulting binary image clearly shows you the background/foreground segmentation:



Let's have a look at how these steps work.

How it works...

The `cv::calcHist` function has many parameters to permit its use in many contexts, which are as follows:

```
void calcHist(const Mat* images, int nimages,
const int* channels, InputArray mask, OutputArray hist,
int dims, const int* histSize, const float** ranges,
bool uniform=true, bool accumulate=false )
```

Most of the time, your histogram will be one of a single one-channel or three-channel image. However, the function allows you to specify a multiple-channel image distributed over several images. This is why an array of images is input into this function. The sixth parameter, `dims`, specifies the dimensionality of the histogram, for example, `1` for a 1D histogram. Even if you are analyzing a multichannel image, you do not have to use all of its channels in the computation of the histogram. The channels to be considered are listed in the `channel` array that has the specified dimensionality. In our class implementation, this single channel is the channel `0` by default. The histogram itself is described by the number of bins in each dimension (this is the `histSize` array of an integer) and by the minimum (inclusive) and maximum (exclusive) values in each dimension (given by the `ranges` array of two-element arrays). It is also possible to define a nonuniform histogram, in which case you need to specify the limits of each bin.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

As with many OpenCV functions, a mask can be specified, indicating which pixels you want to include in the count (all pixels for which the mask value is `0` are then ignored). Two additional optional parameters can be specified, both of which are Boolean values. The first one indicates whether the histogram is uniform or not (uniform is the default). The second allows you to accumulate the result of several histogram computations. If this last parameter is `true`, then the pixel count of the image will be added to the current values that were found in the input histogram. This is useful when you want to compute the histogram of a group of images.

The resulting histogram is stored in a `cv::Mat` instance. Indeed, the `cv::Mat` class can be used to manipulate general *N*-dimensional matrices. Recall from `Chapter 2`, *Manipulating the Pixels*, that this class defined the `at` method for matrices of dimensions 1, 2, and 3. This is why we were able to write the following code when accessing each bin of the 1D histogram in the `getHistogramImage` method:

```
float binVal = hist.at<float>(h);
```

Note that the values in the histogram are stored as `float` values.

There's more...

The `Histogram1D` class that was presented in this recipe has simplified the `cv::calcHist` function by restricting it to the 1D histogram. This is useful for gray-level images, but what about color images?

Computing histograms of color images

Using the same `cv::calcHist` function, we can compute histograms of multichannel images. For example, a class that computes histograms of color `BGR` images can be defined as follows:

```
class ColorHistogram {

  private:

    int histSize[3];        // size of each dimension
    float hranges[2];       // range of values
    const float* ranges[3]; // ranges for each dimension
    int channels[3];        // channel to be considered

  public:
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
ColorHistogram() {

  // Prepare default arguments for a color histogram
  // each dimension has equal size and range
  histSize[0]= histSize[1]= histSize[2]= 256;
  hranges[0]= 0.0;    // BRG range from 0 to 256
  hranges[1]= 256.0;
  ranges[0]= hranges; // in this class,
  ranges[1]= hranges; // all channels have the same range
  ranges[2]= hranges;
  channels[0]= 0;    // the three channels
  channels[1]= 1;
  channels[2]= 2;
}
```

In this case, the histogram will be three-dimensional. Therefore, we need to specify a range for each of the three dimensions. In the case of our `BGR` image, the three channels have the same `[0, 255]` range. With the arguments prepared, the color histogram is computed by the following method:

```
// Computes the histogram.
cv::Mat getHistogram(const cv::Mat &image) {

  cv::Mat hist;

  // BGR color histogram
  hranges[0]= 0.0;    // BRG range
  hranges[1]= 256.0;
  channels[0]= 0;    // the three channels
  channels[1]= 1;
  channels[2]= 2;

  // Compute histogram
  cv::calcHist(&image,
    1,         // histogram of 1 image only
    channels,  // the channel used
    cv::Mat(),  // no mask is used
    hist,       // the resulting histogram
    3,         // it is a 3D histogram
    histSize,  // number of bins
    ranges    // pixel value range
  );

  return hist;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

A three-dimensional `cv::Mat` instance is returned. When a histogram of `256` bins is selected, this matrix has `(256)^3` elements, which represents more than 16 million entries. In many applications, it would be better to reduce the number of bins in the computation of the histogram. It is also possible to use the `cv::SparseMat` data structure that is designed to represent large, sparse matrices (that is, matrices with very few nonzero elements) without consuming too much memory. The `cv::calcHist` function has a version that returns one such matrix. It is, therefore, simple to modify the previous method in order to use `cv::SparseMatrix`:

```
// Computes the histogram.
cv::SparseMat getSparseHistogram(const cv::Mat &image) {

  cv::SparseMat hist(3,        // number of dimensions
                     histSize, // size of each dimension
                  CV_32F);

  // BGR color histogram
  hranges[0]= 0.0;    // BRG range
  hranges[1]= 256.0;
  channels[0]= 0;     // the three channels
  channels[1]= 1;
  channels[2]= 2;

  // Compute histogram
  cv::calcHist(&image,
    1,         // histogram of 1 image only
    channels,  // the channel used
    cv::Mat(),  // no mask is used
    hist,       // the resulting histogram
    3,         // it is a 3D histogram
    histSize,  // number of bins
    ranges    // pixel value range
  );

  return hist;
}
```

Obviously, it is also possible to illustrate the color distribution in an image by showing the individual R, G, and B histograms.

See also

- The *Backprojecting a histogram to detect specific image content* recipe later in this chapter makes use of color histograms in order to detect specific image content.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've successfully learned how to compute the image histogram. Now, let's move on to the next recipe!

Applying lookup tables to modify the image's appearance

Image histograms capture the way a scene is rendered using the available pixel intensity values. By analyzing the distribution of the pixel values over an image, it is possible to use this information to modify and possibly improve an image. This recipe explains how you can use a simple mapping function, represented by a lookup table, to modify the pixel values of an image. As we will see, lookup tables are often defined from histogram distributions.

How to do it...

A **lookup table** is a simple one-to-one (or many-to-one) function that defines how pixel values are transformed into new values. It is a 1D array with, in the case of regular gray-level images, `256` entries:

1. Entry `i` of the table gives you the new intensity value of the corresponding gray level, which is as follows:

```
newIntensity= lookup[oldIntensity];
```

2. The `cv::LUT` function in OpenCV applies a lookup table to an image in order to produce a new image. We can add this function to our `Histogram1D` class:

```
cv::Mat applyLookUp(const cv::Mat& image, // input image
   const cv::Mat& lookup) { // 1x256 uchar matrix

   // the output image
   cv::Mat result;

   // apply lookup table
   cv::LUT(image,lookup,result);

   return result;
}
```

It's time to see how this works.

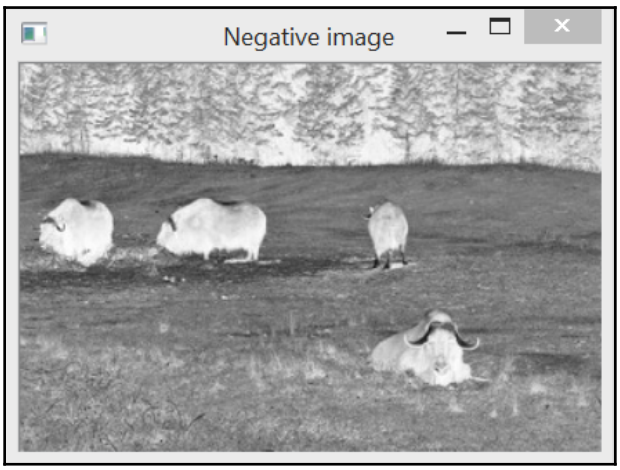
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

When a lookup table is applied to an image, it results in a new image where the pixel intensity values have been modified, as prescribed by the lookup table. A simple transformation could be the following:

```
// Create an image inversion table
int dim(256);
cv::Mat lut(1, // 1 dimension
   &dim,       // 256 entries
   CV_8U);     // uchar

for (int i=0; i<256; i++) {
   lut.at<uchar>(i)= 255-i;
}
```

This transformation simply inverts the pixel intensities, that is, intensity `0` becomes `255`, `1` becomes `254`, and so on. Applying such a lookup table on an image will produce the negative of the original image. For the image of the previous recipe, the result is seen here:



As you can see, we now have the pixel intensities reversed compared to the original image.

There's more...

Lookup tables are useful for any application in which all pixel intensities are given a new intensity value. The transformation, however, has to be global; that is, all pixels of each intensity value must undergo the same transformation.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Stretching a histogram to improve the image contrast

It is possible to improve an image's contrast by defining a lookup table that modifies the original image's histogram. For example, if you observe the histogram of the previous image that was shown in the first recipe, it is easy to notice that the full range of possible intensity values is not used (in particular, for this image, the brighter intensity values have not been used). You can, therefore, stretch the histogram in order to produce an image with an expanded contrast. To do so, the procedure uses a percentile threshold that defines the percentage of pixels that should be black and white in the stretched image. We must, therefore, find the lowest (`imin`) and the highest (`imax`) intensity value so that we have the required minimum number of pixels below or above the specified percentile. The intensity values can then be remapped so that the `imin` value is repositioned at intensity `0`, and the `imax` is assigned the value of `255`. The in-between `i` intensities are simply remapped linearly, as follows:

```
255.0*(i-imin)/(imax-imin);
```

Consequently, the complete image `stretch` method would look as follows:

```
cv::Mat stretch(const cv::Mat &image, int minValue=0) {

   // Compute histogram first
   cv::Mat hist= getHistogram(image);

   // find left extremity of the histogram
   int imin= 0;
   for( ; imin < histSize[0]; imin++ ) {
      // ignore bins with less than minValue entries
      if (hist.at<float>(imin) > minValue)
         break;
   }
   // find right extremity of the histogram
   int imax= histSize[0]-1;
   for( ; imax >= 0; imax-- ) {

      // ignore bins with less than minValue entries
      if (hist.at<float>(imax) > minValue)
         break;
   }
   // Create lookup table
   int dim(256);
   cv::Mat lookup(1,  // 1 dimension
         &dim,       // 256 entries
         CV_8U);     // uchar

   // Build lookup table
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        for (int i=0; i<256; i++) {
           // stretch between imin and imax
           if (i < imin) lookup.at<uchar>(i)= 0;
           else if (i > imax) lookup.at<uchar>(i)= 255;
           // linear mapping
        else lookup.at<uchar>(i)=
          cvRound(255.0*(i-imin)/(imax-imin));
        }

        // Apply lookup table
        cv::Mat result;
        result= applyLookUp(image,lookup);

        return result;
    }
```

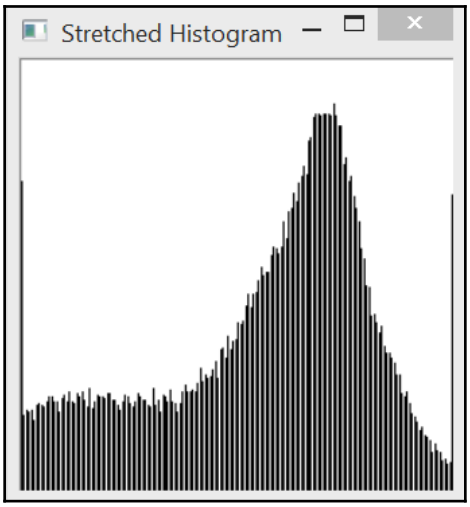
Note the call to our `applyLookUp` method once this one has been computed. Also, in practice, it could be advantageous not only to ignore bins with the `0` value, but also entries with negligible count, for example, less than a given value (defined here as `minValue`). The method is called as follows:

```
// setting 1% of pixels at black and 1% at white
cv::Mat streteched = h.stretch(image,0.01f);
```

The resulting stretched image is as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The expanded histogram then looks as follows:



Let's see what happens when we apply a lookup table on color images.

Applying a lookup table on color images

In `Chapter 2`, *Manipulating the Pixels*, we defined a color-reduction function that modifies the BGR values of an image in order to reduce the number of possible colors. We did this by looping through the image's pixels and applying the color-reduction function on each of them. In fact, it would be much more efficient to precompute all color reductions and then modify each pixel by using a lookup table. This is indeed very easy to accomplish from what we learned in the preceding recipe. The new color-reduction function would then be written as follows:

```
void colorReduce(cv::Mat &image, int div=64) {
    // creating the 1D lookup table
    cv::Mat lookup(1,256,CV_8U);
    // defining the color reduction lookup
    for (int i=0; i<256; i++)
    lookup.at<uchar>(i)= i/div*div + div/2;

    // lookup table applied on all channels
    cv::LUT(image,lookup,image);
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The color-reduction scheme is correctly applied here because when a one-dimensional lookup table is applied to a multichannel image, then the same table is individually applied to all channels. When a lookup table has more than one dimension, then it must be applied to an image with the same number of channels.

We've successfully learned how to apply lookup tables to modify the image's appearance. Now, let's move on to the next recipe!

Equalizing the image histogram

In the previous recipe, we showed you how the contrast of an image can be improved by stretching a histogram so that it occupies the full range of the available intensity values. This strategy indeed constitutes an easy fix that can effectively improve an image. However, in many cases, the visual deficiency of an image is not that it uses a too-narrow range of intensities. Rather, it is that some intensity values are used more frequently than others. The histogram that was shown in the first recipe of this chapter is a good example of this phenomenon. The middle-gray intensities are indeed heavily represented, while darker and brighter pixel values are rather rare. In fact, you can think that a good-quality image should make equal use of all available pixel intensities. This is the idea behind the concept of **histogram equalization**; that is, making the image histogram as flat as possible.

How to do it...

For this recipe, we need to perform the following steps:

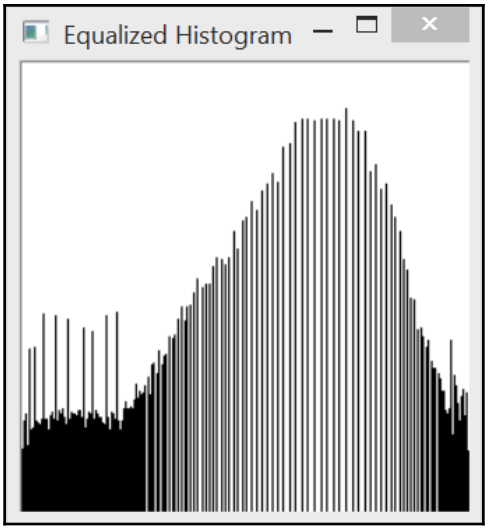
1. OpenCV offers an easy-to-use function that performs histogram equalization. It is called as follows:

```
cv::equalizeHist(image,result);
```

After applying it on our image, the following screenshot is the result:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This equalized image produces the following histogram:



Of course, the histogram cannot be perfectly flat because the lookup table is a global many-to-one transformation. However, it can be seen that the general distribution of the histogram is now more uniform than the original one.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

In a perfectly uniform histogram, all bins would have an equal number of pixels. This implies that 50% of the pixels should have an intensity lower than `128`, 25% should have an intensity lower than `64`, and so on. This observation can be expressed using the rule that, in a uniform histogram, *p%* of the pixels must have an intensity value lower than or equal to *255*p%*. The rule that's used to equalize a histogram is that the mapping of intensity `i` should be at the intensity that corresponds to the percentage of pixels that have an intensity value below `i`. Therefore, the required lookup table can be built from the following equation:

```
lookup.at<uchar>(i)= static_cast<uchar>(255.0*p[i]/image.total());
```

Here, `p[i]` is the number of pixels that have an intensity lower than or equal to `i`. The `p[i]` function is often referred to as a cumulative histogram; that is, it is a histogram that contains the count of pixels lower than or equal to a given intensity instead of containing the count of pixels that have a specific intensity value. Recall that `image.total()` returns the number of pixels in an image so that `p[i]/image.total()` is a percentage of pixels.

Generally, histogram equalization greatly improves the image's appearance. However, depending on the visual content, the quality of the result can vary from image to image.

We've successfully learned how to equalize the image histogram. Now, let's move on to the next recipe!

Backprojecting a histogram to detect specific image content

A histogram is an important characteristic of an image's content. If you look at an image area that shows a particular texture or a particular object, then the histogram of this area can be seen as a function that gives the probability that a given pixel belongs to this specific texture or object. In this recipe, you will learn how the concept of **histogram backprojection** can be used advantageously to detect specific image content.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Suppose you have an image and you wish to detect specific content inside it (for example, in the following image, the clouds in the sky):

1. The first thing to do is select a region of interest (ROI) that contains a sample of what you are looking for. This region is the one inside the rectangle that's been drawn on the following test image:

2. In our program, the ROI is obtained as follows:

```
cv::Mat imageROI;
imageROI= image(cv::Rect(216,33,24,30)); // Cloud region
```

3. You then extract the histogram of this ROI. This is easily accomplished using the `Histogram1D` class, which was defined in the first recipe of this chapter, as follows:

```
Histogram1D h;
cv::Mat hist= h.getHistogram(imageROI);
```

4. By normalizing this histogram, we obtain a function that gives us the probability that a pixel of a given intensity value belongs to the defined area, as follows:

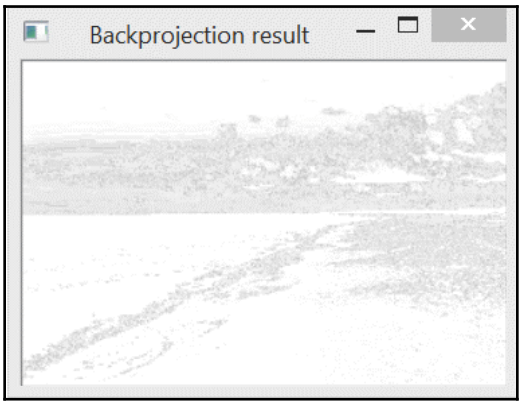
```
cv::normalize(histogram,histogram,1.0);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Backprojecting a histogram consists of replacing each pixel value in an input image with its corresponding probability value read in the normalized histogram. An OpenCV function performs this task as follows:

```
cv::calcBackProject(&image,
          1,          // one image
          channels,   // the channels used,
                      // based on histogram dimension
          histogram,  // the histogram we are backprojecting
          result,     // the resulting backprojection image
          ranges,       // the ranges of values
          255.0         // the scaling factor is chosen
          // such that a probability value of 1 maps to 255
     );
```

The result is the following probability map, with probabilities of belonging to the reference area ranging from bright (low probability) to dark (high probability):

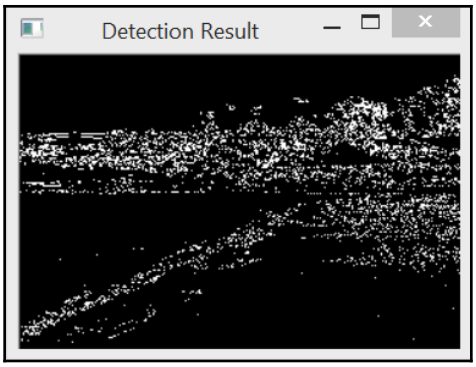


6. If we apply a threshold on this image, we obtain the most probable *cloud* pixels:

```
cv::threshold(result, result, threshold, 255, cv::THRESH_BINARY);
```

The result is shown in the following screenshot:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



We will now look at how these steps actually work.

How it works...

The preceding result can be disappointing because, in addition to the clouds, other areas have been incorrectly detected as well. It is important to understand that the probability function has been extracted from a simple gray-level histogram. Many other pixels in the image share the same intensities as the cloud pixels, and pixels of the same intensity are replaced with the same probability value when backprojecting the histogram. One solution to improve the detection result would be to use the color information. However, in order to do this, we need to modify the call to `cv::calBackProject`.

The `cv::calBackProject` function is similar to the `cv::calcHist` function. The first parameter specifies the input image. You then need to list the channel numbers you wish to use. The histogram that is passed to the function is, this time, an input parameter; its dimension should match the one of the channel list array. As with `cv::calcHist`, the ranges parameter specifies the bin boundaries of the input histogram in the form of an array of float arrays, each specifying the range (minimum and maximum values) of each channel. The resulting output is an image, which is the computed probability map. Since each pixel is replaced by the value found in the histogram at the corresponding bin position, the resulting image has values between `0.0` and `1.0` (assuming a normalized histogram has been provided as input). A last parameter allows you to optionally rescale these values by multiplying them by a given factor.

There's more...

Let's see how we can use the color information in the histogram backprojection algorithm.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Backprojecting color histograms

Multidimensional histograms can also be backprojected onto an image. Let's define a class that encapsulates the backprojection process. First, we define the required attributes and initialize the data, as follows:

```
class ContentFinder {

  private:

  // histogram parameters
  float hranges[2];
   const float* ranges[3];
   int channels[3];

  float threshold;           // decision threshold
  cv::Mat histogram;         // input histogram

  public:

  ContentFinder() : threshold(0.1f) {

    // in this class, all channels have the same range
    ranges[0]= hranges;
    ranges[1]= hranges;
    ranges[2]= hranges;
  }
```

Next, we define a threshold parameter that will be used to create the binary map that shows the detection result. If this parameter is set to a negative value, the raw probability map will be returned. Refer to the following code:

```
  // Sets the threshold on histogram values [0,1]
  void setThreshold(float t) {

     threshold= t;
  }

  // Gets the threshold
  float getThreshold() {

     return threshold;
  }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The input histogram is normalized (this is, however, not required) as follows:

```
// Sets the reference histogram
void setHistogram(const cv::Mat& h) {

   histogram= h;
   cv::normalize(histogram,histogram,1.0);
}
```

To backproject the histogram, you simply need to specify the image, the range (we assumed here that all channels have the same range), and the list of channels used. Refer to the following code:

```
// All channels used, with range [0,256[
cv::Mat find(const cv::Mat& image) {

  cv::Mat result;

  hranges[0]= 0.0;  // default range [0,256[
  hranges[1]= 256.0;
  channels[0]= 0;    // the three channels
  channels[1]= 1;
  channels[2]= 2;

  return find(image, hranges[0], hranges[1], channels);
}

// Finds the pixels belonging to the histogram
cv::Mat find(const cv::Mat& image,
             float minValue, float maxValue,
             int *channels) {

  cv::Mat result;

  hranges[0]= minValue;
  hranges[1]= maxValue;

    // histogram dim matches channel list
  for (int i=0; i<histogram.dims; i++)
      this->channels[i]= channels[i];

  cv::calcBackProject(&image,
          1,         // we only use one image at a time
          channels,  // vector specifying what histogram
          // dimensions belong to what image channels
          histogram,    // the histogram we are using
          result,       // the backprojection image
          ranges,       // the range of values,
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
                        //for each dimension
            255.0       // the scaling factor is chosen such
            // that a histogram value of 1 maps to 255
       );
    }

      // Threshold backprojection to obtain a binary image
    if (threshold>0.0)
      cv::threshold(result, result,
                255.0*threshold, 255.0, cv::THRESH_BINARY);

    return result;
  }
```

Now, let's use a BGR histogram on the color version of the image we used previously (see this book's website to see this image in color). This time, we will try to detect the blue-sky area. We will first load the color image, define the region of interest, and compute the 3D histogram on a reduced color space, as follows:

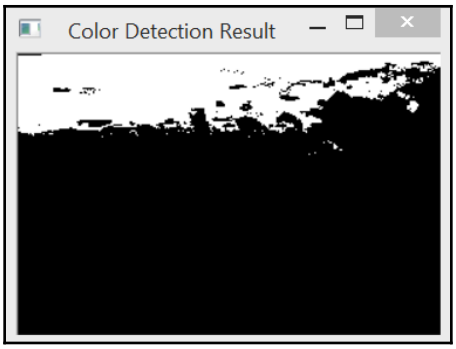
```
// Load color image
ColorHistogram hc;
cv::Mat color= cv::imread("waves.jpg");

// extract region of interest
imageROI= color(cv::Rect(0,0,165,75)); // blue sky area

// Get 3D color histogram (8 bins per channel)
hc.setSize(8); // 8x8x8
cv::Mat shist= hc.getHistogram(imageROI);
```

Next, you need to compute the histogram and use the `find` method to detect the sky portion of the image, as follows:

```
// Create the content finder
ContentFinder finder;
// set histogram to be back-projected
finder.setHistogram(shist);
finder.setThreshold(0.05f);

// Get back-projection of color histogram
Cv::Mat result= finder.find(color);
```

The result of the detection on the color version of the image in the previous section is shown here:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



The BGR color space is generally not the best one to identify color objects in an image. Here, to make it more reliable, we reduced the number of colors before computing the histogram (remember that the original BGR space counts more than 16 million colors). The histogram extracted represents the typical color distribution for a sky area. Try to backproject it onto another image. It should also detect the sky portion. Note that using a histogram built from multiple sky images should increase the accuracy of this detection.

Note that in this case, computing a sparse histogram would have been better in terms of memory usage. You should be able to redo this exercise using `cv::SparseMat` this time. Also, if you are looking for a bright-colored object, using the hue channel of the HSV color space would probably be more efficient. In other cases, the use of the chromaticity components of a perceptually uniform space (such as *L*a*b**) might constitute a better choice.

We've successfully learned how to backproject a histogram to detect specific image content. Now, let's move on to the next recipe!

Using the mean shift algorithm to find an object

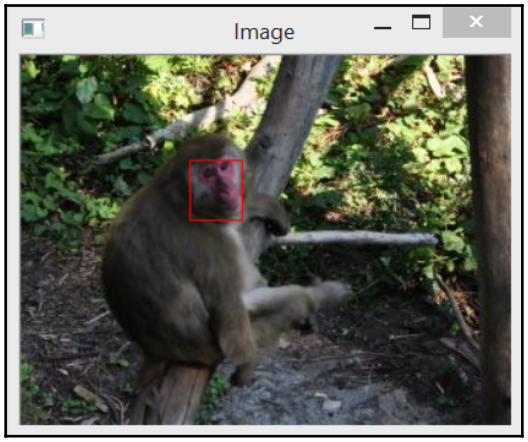
The result of a histogram backprojection is a probability map that expresses the probability that a given image's content is found at a specific image location. Suppose we now know the approximate location of an object in an image; the probability map can be used to find the exact location of the object. The most probable will be the one that maximizes this probability inside a given window. Therefore, if we start from an initial location and move around iteratively, it should be possible to find the exact object location. This is what is accomplished by the **mean shift algorithm**.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

The steps for this recipe are as follows:

1. Suppose we have identified an object of interest—here, a baboon's face—as shown in the following screenshot (refer to this book's graphics PDF to view this image in color):



2. This time, we will describe this object by using the hue channel of the HSV color space. This means that we need to convert the image into an HSV one and then extract the hue channel and compute the 1D hue histogram of the defined ROI. Refer to the following code:

```
// Read reference image
cv::Mat image= cv::imread("baboon1.jpg");
// Baboon's face ROI
cv::Mat imageROI= image(cv::Rect(110,260,35,40));
// Get the Hue histogram
int minSat=65;
ColorHistogram hc;
cv::Mat colorhist= hc.getHueHistogram(imageROI,minSat);
```

3. As we can see, the hue histogram is obtained using a convenient method that we have added to our `ColorHistogram` class as follows:

```
// Computes the 1D Hue histogram with a mask.
// BGR source image is converted to HSV
// Pixels with low saturation are ignored
cv::Mat getHueHistogram(const cv::Mat &image,
                        int minSaturation=0) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
  cv::Mat hist;

  // Convert to HSV color space
  cv::Mat hsv;
  cv::cvtColor(image, hsv, COLOR_BGR2HSV);

  // Mask to be used (or not)
  cv::Mat mask;

  if (minSaturation>0) {
    // Spliting the 3 channels into 3 images
    std::vector<cv::Mat> v;
    cv::split(hsv,v);

    // Mask out the low saturated pixels
    cv::threshold(v[1],mask,minSaturation,255,THRESH_BINARY);
  }

  // Prepare arguments for a 1D hue histogram
  hranges[0]= 0.0;    // range is from 0 to 180
  hranges[1]= 180.0;
  channels[0]= 0;     // the hue channel

  // Compute histogram
  cv::calcHist(&hsv,
    1,       // histogram of 1 image only
    channels, // the channel used
    mask,     // binary mask
    hist,     // the resulting histogram
    1,       // it is a 1D histogram
    histSize, // number of bins
    ranges   // pixel value range
  );

  return hist;
}
```

4. The resulting histogram is then passed to our `ContentFinder` class instance as follows:

```
ContentFinder finder;
finder.setHistogram(colorhist);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

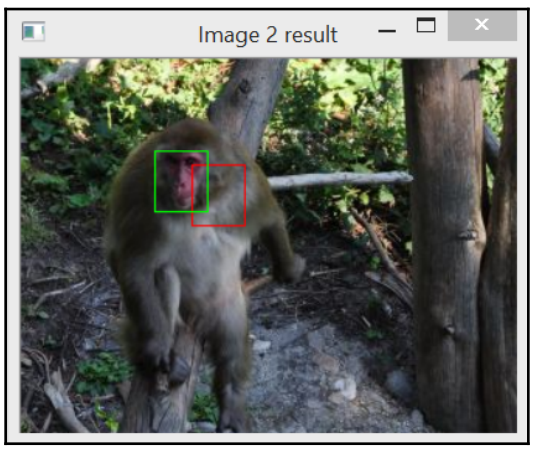
5. Now, let's open a second image where we want to locate the new position of the baboon's face. This image needs to be converted in to the HSV space first, and then we can backproject the histogram of the first image. Refer to the following code:

```
image= cv::imread("baboon3.jpg");
// Convert to HSV space
cv::cvtColor(image, hsv, COLOR_BGR2HSV);
// Get back-projection of hue histogram
int ch[1]={0};
finder.setThreshold(-1.0f); // no thresholding
cv::Mat result= finder.find(hsv,0.0f,180.0f,ch);
```

6. Now, from an initial rectangular area (that is, the position of the baboon's face in the initial image), the `cv::meanShift` algorithm of OpenCV will update the `rect` object at the new location of the baboon's face. Refer to the following code:

```
// initial window position
 cv::Rect rect(110,260,35,40);

// search objet with mean shift
 cv::TermCriteria criteria(cv::TermCriteria::MAX_ITER,
                           10,0.01);
 cv::meanShift(result,rect,criteria);
```

The initial (red) and new (green) face locations are displayed in the following screenshot (refer to this book's graphics PDF to view this image in color):



Let's have a look at how this steps work.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

In this example, we used the hue component of the HSV color space in order to characterize the object we were looking for. We made this choice because the baboon's face has a very distinctive pink color; consequently, considering the pixels' hue should make the face easily identifiable. The first step, therefore, is to convert the image in to the HSV color space. The hue component is the first channel of the resulting image when the `COLOR_BGR2HSV` flag is used. This is an 8-bit component that varies from `0` to `180` (with `cv::cvtColor`, the converted image is of the same type as the source image). In order to extract the hue image, the three-channel HSV image is split into three one-channel images using the `cv::split` function. The three images are put into a `std::vector` instance, and the hue image is the first entry of the vector (that is, at index `0`).

When using the hue component of a color, it is always important to take its saturation into account (which is the second entry of the vector). Indeed, when the saturation of a color is low, the hue information becomes unstable and unreliable. This is due to the fact that for low-saturated color, the B, G, and R components are almost equal. This makes it difficult to determine the exact color that is represented. Consequently, we decided to ignore the hue component of colors with low saturation. That is, they are not counted in the histogram (we did this by using the `minSat` parameter that masks out pixels with saturation below this threshold in the `getHueHistogram` method).

The mean shift algorithm is an iterative procedure that locates the local maxima of a probability function. It does this by finding the centroid, or weighted mean, of the data point inside a predefined window. The algorithm then moves the window center to the centroid location and repeats the procedure until the window center converges to a stable point. The OpenCV implementation defines two stopping criteria: a maximum number of iterations and a window center displacement value below which the position is considered to have converged to a stable point. These two criteria are stored in a `cv::TermCriteria` instance. The `cv::meanShift` function returns the number of iterations that have been performed. Obviously, the quality of the result depends on the quality of the probability map provided on the given initial position. Note that, here, we used a histogram of colors to represent an image's appearance; it is also possible to use histograms of other features to represent the object (for example, a histogram of edge orientation).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The mean-shift algorithm has been largely used for visual tracking. `Chapter 10`, *Estimating Projective Relations in Images*, will explore the problem of object tracking in more detail.
- The mean shift algorithm has been introduced in the article, *Mean Shift: a robust approach toward feature space analysis* by D. Comaniciu and P. Meer in *IEEE transactions on Pattern Analysis and Machine Intelligence, Volume 24, Number 5*, May 2002.

OpenCV also offers an implementation of the CamShift algorithm, which is an improved version of the mean shift algorithm in which the size and the orientation of the window can change.

We've successfully learned how to use the mean shift algorithm to find an object. Now, let's move on to the next recipe!

Retrieving similar images using histogram comparison

Content-based image retrieval is an important problem in computer vision. It consists of finding a set of images that present content that is similar to a given query image. Since we have learned that histograms constitute an effective way to characterize an image's content, it makes sense to think that they can be used to solve content-based retrieval problems.

The key here is to be able to measure the similarity between two images by simply comparing their histograms. A measurement function that will estimate how different, or how similar, two histograms are will need to be defined. Various measures have been proposed in the past, and OpenCV proposes a few of them in its implementation of the `cv::compareHist` function.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

In order to follow this recipe, follow these steps:

1. In order to compare a reference image with a collection of images and find the ones that are the most similar to this query image, we created an `ImageComparator` class. This class contains a reference to a query image and an input image, together with their histograms. In addition, since we will perform the comparison using color histograms, the `ColorHistogram` class is used as follows:

```
class ImageComparator {

  private:

  cv::Mat refH;       // reference histogram
  cv::Mat inputH;     // histogram of input image

  ColorHistogram hist; // to generate the histograms
  int nBins; // number of bins used in each color channel

  public:

  ImageComparator() :nBins(8) {

  }
```

2. To get a reliable similarity measure, the histogram should be computed over a reduced number of bins. Therefore, the class allows you to specify the number of bins to be used in each of the BGR channels. Refer to the following code:

```
// Set number of bins used when comparing the histograms
void setNumberOfBins( int bins) {

  nBins= bins;
}
```

3. The query image is specified using an appropriate setter that also computes the reference histogram, as follows:

```
// compute histogram of reference image
void setReferenceImage(const cv::Mat& image) {

  hist.setSize(nBins);
  refH= hist.getHistogram(image);
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Finally, a `compare` method compares the reference image with a given input image. The following method returns a score that indicates how similar the two images are:

```
// compare the image using their BGR histograms
double compare(const cv::Mat& image) {

  inputH= hist.getHistogram(image);

  return cv::compareHist(refH,inputH,HISTCMP_INTERSECT);
}
```

5. The preceding class can be used to retrieve images that are similar to a given query image. The following code is initially provided to the class instance:

```
ImageComparator c;
c.setReferenceImage(image);
```

6. Here, the query image we used is the color version of the beach image that was shown in the *Backprojecting a histogram to detect specific image content* recipe earlier in this chapter. This image was compared to the following series of images:

As you can see, the images are shown in an order, from the most similar to the least similar.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Most histogram comparison measures are based on bin-by-bin comparisons. This is why it is important to work with a reduced number of histogram bins when measuring the similarity of two color histograms. The call to `cv::compareHist` is straightforward. You just input the two histograms and the function returns the measured distance. The specific measurement method you want to use is specified using a flag. In the `ImageComparator` class, the intersection method is used (with the `CV_COMP_INTERSECT` flag). This method simply compares, for each bin, the two values in each histogram and keeps the minimum one. The similarity measure, then, is the sum of these minimum values. Consequently, two images that have histograms with no colors in common would get an intersection value of `0`, while two identical histograms would get a value that is equal to the total number of pixels.

The other available methods are the chi-square measure (the `HISTCMP_CHISQR` flag), which sums the normalized square difference between the bins, the correlation method (the `HISTCMP_CORREL` flag), which is based on the normalized cross-correlation operator that's used in signal processing to measure the similarity between two signals, and the Bhattacharyya measure (the `HISTCMP_BHATTACHARYYA` flag), which is used in statistics to estimate the similarity between two probabilistic distributions.

See also

- The OpenCV documentation provides a description of the exact formulas that are used in the different histogram comparison measures.
- The Earth Mover Distance is another popular histogram comparison method. It is implemented in OpenCV as the `cv::EMD` function. The main advantage of this method is that it takes into account the values found in adjacent bins to evaluate the similarity of two histograms. It is described in the article, *The Earth Mover's Distance as a Metric for Image Retrieval* by Y. Rubner, C. Tomasi, and L. J. Guibas in *Int. Journal of Computer Vision, Volume 40, Number 2., 2000, pp. 99-121*.

We've successfully learned how to retrieve similar images using the histogram comparison. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Counting pixels with integral images

In the previous recipes, we learned that a histogram is computed by going through all pixels of an image and cumulating a count of how often each intensity value occurs in this image. We have also seen that, sometimes, we are only interested in computing our histogram in some area of the image. In fact, having to cumulate a sum of pixels inside an image's subregion is a common task in many computer vision algorithms. Now, suppose you have to compute several such histograms over multiple regions of interest inside your image. All these computations could rapidly become very costly. In such a situation, there is a tool that can drastically improve the efficiency of counting pixels over image subregions—the **integral image**. Integral images have been introduced as an efficient way of summing pixels in image ROIs. They are widely used in applications that involve, for example, computations over sliding windows at multiple scales.

This recipe will explain the principle behind the integral image. Our objective here is to show how pixels can be summed over a rectangle region by using only three arithmetic operations. Once we have learned about this concept, the *There's more...* section of this recipe will show you two examples where integral images can be advantageously used.

How to do it...

This recipe will play with the following picture in which an ROI showing a girl on her bike is identified:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's have a look at the following code:

1. Integral images are useful when you need to sum pixels over several image areas. Normally, if you wish to get the sum of all pixels over an ROI, you would write the following code:

```
// Open image
cv::Mat image= cv::imread("bike55.bmp",0);
// define image roi (here the girl on bike)
int xo=97, yo=112;
int width=25, height=30;
cv::Mat roi(image,cv::Rect(xo,yo,width,height));
// compute sum
// returns a Scalar to work with multi-channel images
cv::Scalar sum= cv::sum(roi);
```

2. The `cv::sum` function simply loops over all the pixels of the region and accumulates the sum. Using the integral image, this can be achieved using only three additive operations. However, first, you need to compute the integral image, as follows:

```
// compute integral image
cv::Mat integralImage;
cv::integral(image,integralImage,CV_32S);
```

3. As will be explained in the next section, the same result can be obtained using this simple arithmetic expression on the computed integral image as follows:

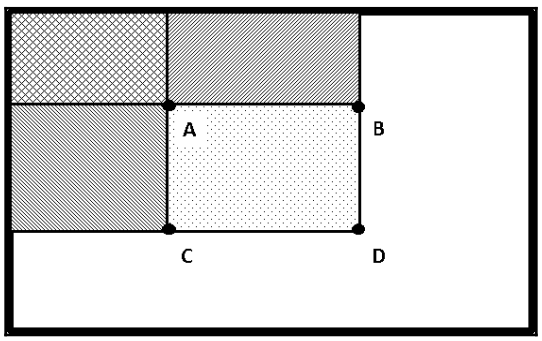
```
// get sum over an area using three additions/subtractions
 int sumInt= integralImage.at<int>(yo+height,xo+width)
          -integralImage.at<int>(yo+height,xo)
          -integralImage.at<int>(yo,xo+width)
          +integralImage.at<int>(yo,xo);
```

Both approaches give you the same result. However, computing the integral image is costly, since you have to loop over all the image pixels. The key is that once this initial computation is done, you will need to add only four pixels to get a sum over an ROI, no matter what the size of this region is. Integral images then become advantageous to use when multiple such pixel sums have to be computed over multiple regions of different sizes.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

In the previous section, you were introduced to the concept of integral images through a brief demonstration of the *magic* behind them; that is, how they can be used to compute the sum of pixels inside rectangular regions cheaply. To understand how they work, let's define what an integral image is. An integral image is obtained by replacing each pixel by the value of the sum of all the pixels located inside the upper left quadrant delimitated by this pixel. The integral image can be computed by scanning the image once, as the integral value of a current pixel is given by the integral value of the previously discussed pixel, plus the value of the cumulative sum of the current line. The integral image is therefore a new image containing pixel sums. To avoid overflows, this image is usually an image of `int` values (`CV_32S`) or float values (`CV_32F`). For example, in the following diagram, pixel **A** in this integral image would contain the sum of the pixels contained inside the upper left corner area, which is identified with a double-hatched pattern:



Once the integral image has been computed, any summation over a rectangular region can be easily obtained through four-pixel accesses, and here's why. Considering the preceding diagram again, we can see that the sum of the pixels inside the region delimited by the pixels **A**, **B**, **C**, and **D** can be obtained by reading the integral value at pixel **D**, from which you subtract the values of the pixels over **B** and to the left-hand side of **C**. However, by doing so, you have subtracted twice the sum of pixels located in the upper left corner of **A**; this is why you have to re-add the integral sum at **A**. Formally, then, the sum of pixels inside **A**, **B**, **C**, and **D** is given by *A-B-C+D*. If we use the `cv::Mat` method to access pixel values, this formula translates to the following:

```
// window at (xo,yo) of size width by height
return (integralImage.at<cv::Vec<T,N>>
                                  (yo+height,xo+width)
      -integralImage.at<cv::Vec<T,N>>(yo+height,xo)
      -integralImage.at<cv::Vec<T,N>>(yo,xo+width)
      +integralImage.at<cv::Vec<T,N>>(yo,xo));
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

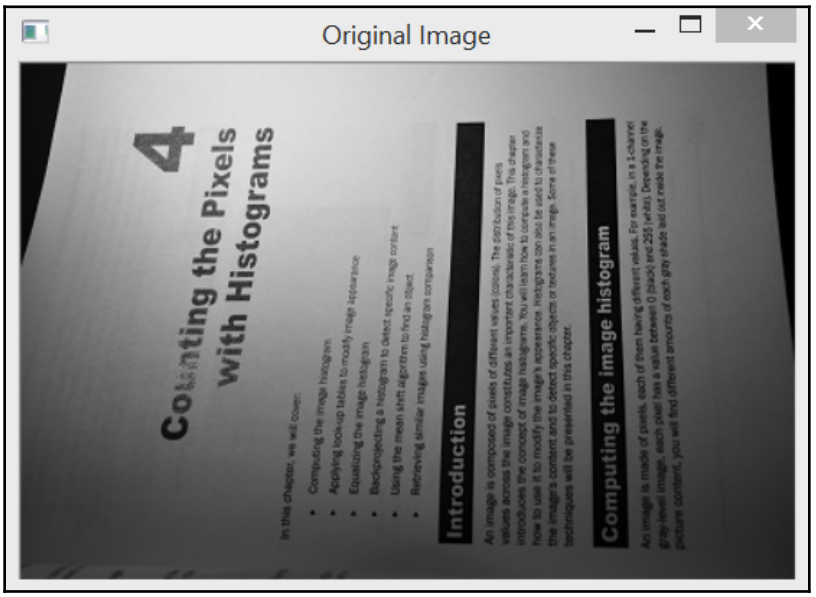
The complexity of this computation is, therefore, constant, no matter what the size of the ROI is. Note that for simplicity, we used the `at` method of the `cv::Mat` class, which is not the most efficient way to access pixel values (see `Chapter 2`, *Manipulating the Pixels*). This aspect will be discussed in the *There's more...* section of this recipe, which presents two applications that benefit from the efficiency of the integral image concept.

There's more...

Integral images are used whenever multiple pixel summations must be performed. In this section, we will illustrate the use of integral images by introducing the concept of adaptive thresholding. Integral images are also useful for the efficient computation of histograms over multiple windows. This is also explained in this section.

Adaptive thresholding

Applying a threshold on an image in order to create a binary image could be a good way to extract the meaningful elements of an image. Suppose that you have the following image of a book:

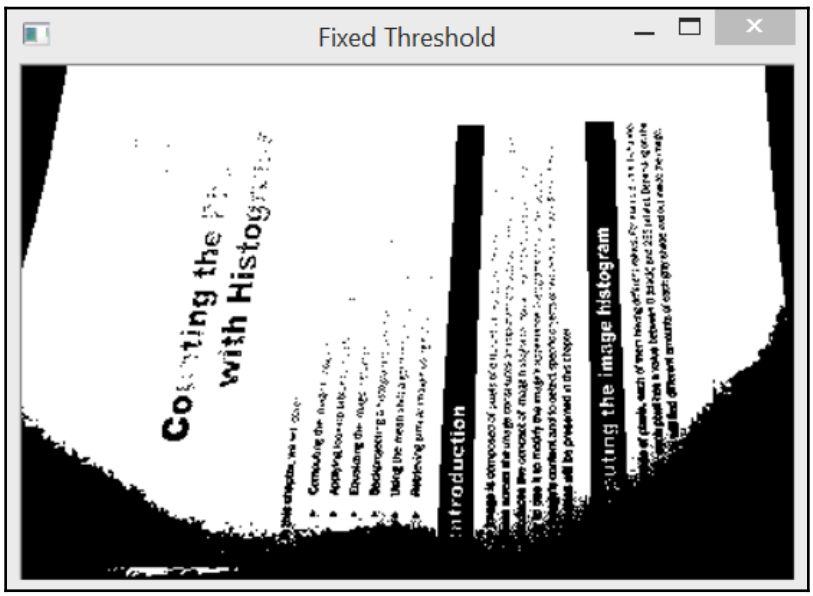


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Since you are interested in analyzing the text in this image, you apply a threshold to this image, as follows:

```
// using a fixed threshold
cv::Mat binaryFixed;
cv::threshold(image,binaryFixed,70,255,cv::THRESH_BINARY);
```

You obtain the following result:



In fact, no matter what value you choose for the threshold, in some parts of the image, you get missing text, whereas in other parts, the text disappears under the shadow. To overcome this problem, one possible solution consists of using a local threshold that is computed from each pixel's neighborhood. This strategy is called **adaptive thresholding**, and it consists of comparing each pixel with the mean value of the neighboring pixels. Pixels that clearly differ from their local mean will then be considered as outliers and will be cut off by the thresholding process.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Adaptive thresholding, therefore, requires the computation of a local mean around every pixel. This requires multiple image window summations that can be computed efficiently through the integral image. Consequently, the first step is to compute the following integral image:

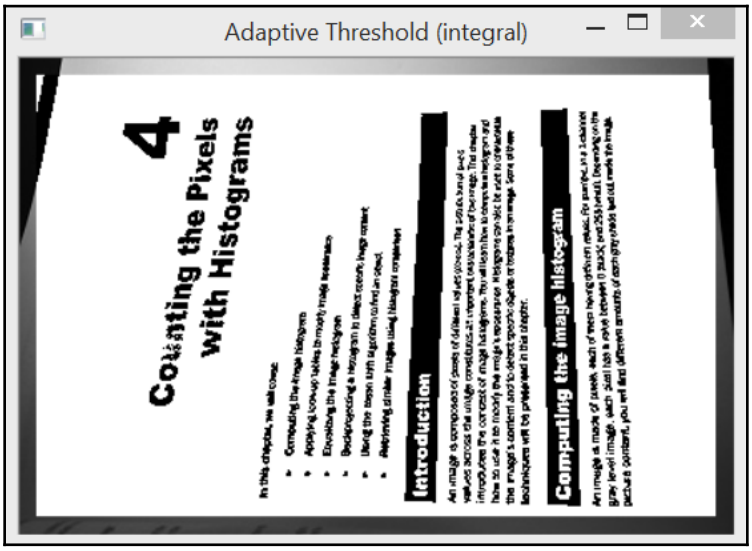
```
// compute integral image
cv::Mat iimage;
cv::integral(image,iimage,CV_32S);
```

Now, we can go through all pixels and compute the mean over a square neighborhood. We could use our `IntegralImage` class to do so, but this one uses the inefficient `at` method for pixel access. This time, let's be efficient by looping over the image using the pointer, like we did in `Chapter 2`, *Manipulating the Pixels*. This loop looks as follows:

```
int blockSize= 21; // size of the neighborhood
int threshold=10;  // pixel will be compared
                    // to (mean-threshold)

// for each row
int halfSize= blockSize/2;
  for (int j=halfSize; j<nl-halfSize-1; j++) {

    // get the address of row j
    uchar* data= binary.ptr<uchar>(j);
    int* idata1= iimage.ptr<int>(j-halfSize);
    int* idata2= iimage.ptr<int>(j+halfSize+1);

    // for pixel of a line
      for (int i=halfSize; i<nc-halfSize-1; i++) {
      // compute sum
      int sum= (idata2[i+halfSize+1]-
                  idata2[i-halfSize]-
              idata1[i+halfSize+1]+
                  idata1[i-halfSize])/
                     (blockSize*blockSize);

      // apply adaptive threshold
      if (data[i]<(sum-threshold))
        data[i]= 0;
      else
        data[i]=255;
        }
  }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this example, a neighborhood of size `21` x `21` is used. To compute each mean, we need to access the four integral pixels that delimit the square neighborhood—two are located on the line pointed by `idata1` and two are on the line pointed by `idata2`. The current pixel is compared to the mean that is thus computed, from which we subtract a threshold value (here, set to `10`); this is to make sure that rejected pixels clearly differ from their local mean. The following binary image is then obtained:



Clearly, this is a much better result that the one we got using a fixed threshold. Adaptive thresholding is a common image-processing technique. As such, it is also implemented in OpenCV as follows:

```
cv::adaptiveThreshold(image,           // input image
                      binaryAdaptive,  // output binary image
                255,             // max value for output
                      cv::ADAPTIVE_THRESH_MEAN_C, // method
                 cv::THRESH_BINARY, // threshold type
                 blockSize,       // size of the block
                 threshold);      // threshold used
```

This function call produces exactly the same result as the one we obtained using our integral image. In addition, instead of using the local mean for thresholding, this function allows you to use a Gaussian weighted sum (the method flag would be `ADAPTIVE_THRESH_GAUSSIAN_C`). It is interesting to note that our implementation is slightly faster than the `cv::adaptiveThreshold` call.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, it is worth mentioning that you can also write an adaptive thresholding procedure by using the OpenCV image operators. This would be done as follows:

```
cv::Mat filtered;
cv::Mat binaryFiltered;
cv::boxFilter(image,filtered,CV_8U,
              cv::Size(blockSize,blockSize));
filtered= filtered-threshold;
binaryFiltered= image>= filtered;
```

Image filtering will be covered in `Chapter 6`, *Filtering the Images*.

Visual tracking using histograms

As we learned in the previous recipes, a histogram constitutes a reliable global representation of an object's appearance. In this recipe, we will demonstrate the usefulness of integral images by showing you how we can locate an object in an image by searching for an image area that presents a histogram similar to a target object. We accomplished this in the *Using the Mean Shift algorithm to find an object* recipe by using the concepts of histogram backprojection and local search through mean shift. This time, we will find our object by performing an explicit search for regions of similar histograms over the full image.

In the special case where an integral image is used on a binary image made of `0` and `1` values, the integral sum gives you the number of pixels that have a value of `1` inside the specified region. We will exploit this fact in this recipe to compute the histogram of a gray-level image.

The `cv::integral` function also works for multichannel images. You can take advantage of this fact to compute histograms of image subregions using integral images. You simply need to convert your image into a multichannel image made of binary planes; each of these planes is associated with a bin of your histogram and shows you which pixels have a value that falls into this bin. The following function creates such multiplane images from a gray-level one:

```
// convert to a multi-channel image made of binary planes
// nPlanes must be a power of 2
void convertToBinaryPlanes(const cv::Mat& input,
                           cv::Mat& output, int nPlanes) {

    // number of bits to mask out
    int n= 8-static_cast<int>( log(static_cast<double>(nPlanes))/log(2.0));
    // mask used to eliminate least significant bits
    uchar mask= 0xFF<<n;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    // create a vector of binary images
    std::vector<cv::Mat> planes;
    // reduce to nBins by eliminating least significant bits
    cv::Mat reduced= input&mask;

    // compute each binary image plane
    for (int i=0; i<nPlanes; i++) {
      // 1 for each pixel equals to i<<shift
      planes.push_back((reduced==(i<<n))&0x1);
    }

    // create multi-channel image
    cv::merge(planes,output);
}
```

The integral image computations can also be encapsulated into one convenient `template` class, as follows:

```
template <typename T, int N>
class IntegralImage {

    cv::Mat integralImage;

  public:

    IntegralImage(cv::Mat image) {

    // (costly) computation of the integral image
    cv::integral(image,integralImage,cv::DataType<T>::type);
    }

    // compute sum over sub-regions of any size
     // from 4 pixel accesses
    cv::Vec<T,N> operator()(int xo, int yo,
                            int width, int height) {

      // window at (xo,yo) of size width by height
      return (integralImage.at<cv::Vec<T,N>>
                                        (yo+height,xo+width)
            -integralImage.at<cv::Vec<T,N>>(yo+height,xo)
            -integralImage.at<cv::Vec<T,N>>(yo,xo+width)
            +integralImage.at<cv::Vec<T,N>>(yo,xo));
    }

};
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We now want to find where the girl on the bicycle, whom we identified in the previous image, is in a subsequent image. Let's compute the histogram of the girl in this original image. We can accomplish this using the `Histogram1D` class we built in a recipe, *Retrieving similar images using histogram comparison*, of this chapter. Here, we produce a 16-bin histogram, as follows:

```
// histogram of 16 bins
Histogram1D h;
h.setNBins(16);
// compute histogram over image roi
cv::Mat refHistogram= h.getHistogram(roi);
```

The preceding histogram will be used as a referential representation to locate the target object (the girl on her bike) in a subsequent image.

Suppose that the only information we have is that the girl is moving more or less horizontally over the image. Since we will have many histograms to compute at various locations, we compute the integral image as a preliminary step. Refer to the following code:

```
// first create 16-plane binary image
cv::Mat planes;
convertToBinaryPlanes(secondIimage,planes,16);
// then compute integral image
IntegralImage<float,16> intHistogram(planes);
```

To perform the search, we loop over a range of possible locations and compare the current histogram with the referential one. Our goal is to find the location with the most similar histograms. Refer to the following code:

```
double maxSimilarity=0.0;
int xbest, ybest;
// loop over a horizontal strip around girl
 // location in initial image
for (int y=110; y<120; y++) {
  for (int x=0; x<secondImage.cols-width; x++) {
    // compute histogram of 16 bins using integral image
    histogram= intHistogram(x,y,width,height);
    // compute distance with reference histogram
    double distance= cv::compareHist(refHistogram,
                              histogram, HISTCMP_INTERSECT);
    // find position of most similar histogram
    if (distance>maxSimilarity) {

      xbest= x;
      ybest= y;
      maxSimilarity= distance;
    }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    }
  }
  // draw rectangle at best location
  cv::rectangle(secondImage,
                cv::Rect(xbest,ybest,width,height),0));
```

The location with the most similar histogram is then identified, like so:

The white rectangle represents the search area. Histograms of all windows that fit inside this area have been computed. We kept the window size constant, but it could have been a good strategy to search for slightly smaller or larger windows in order to take the eventual changes in scale into account. Note that in order to limit the complexity of this computation, the number of bins in the histograms to be computed should be kept low. In our example, we reduced this to `16` bins. Consequently, plane `0` of this multiplane image contains a binary image that shows you all the pixels that have a value between `0` and `15`, while plane `1` shows you pixels with values between `16` and `31`, and so on.

The search for an object consisted in computing the histograms of all windows of the given size over a predetermined range of pixels. This represents the computation of `3200` different histograms that have been efficiently computed from our integral image. All the histograms returned by our `IntegralImage` class are contained in a `cv::Vec` object (because of the use of the `at` method). We then use the `cv::compareHist` function to identify the most similar histogram (remember that this function, like most OpenCV functions, can accept either `cv::Mat` or `cv::Vec` objects through the convenient `cv::InputArray` generic parameter type).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- `Chapter 8`, *Detecting Interest Points*, will present the `SURF` operator that also relies on the use of integral images
- The article, *Robust Fragments-based Tracking using the Integral Histogram* by A. Adam, E. Rivlin, and I. Shimshoni in the proceedings of the *Int. Conference on Computer Vision and Pattern Recognition, 2006, pp. 798-805* describes an interesting approach that uses integral images to track objects in an image sequence

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5
Transforming Images with Morphological Operations

Mathematical morphology is a theory that was developed in the 1960s for the analysis and processing of discrete images. It defines a series of operators that transform an image by probing it with a predefined shape element. The way this shape element intersects the neighborhood of a pixel determines the result of the operation. This chapter presents the most important morphological operators. It also explores the problems of image segmentation and feature detection using algorithms based on morphological operators.

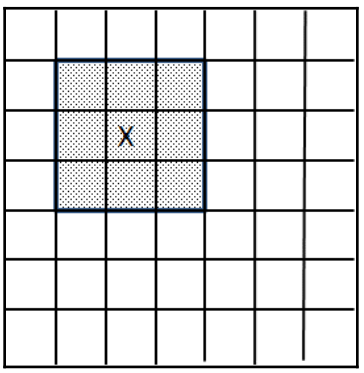
In this chapter, we will cover the following recipes:

- Eroding and dilating images using morphological filters
- Opening and closing images using morphological filters
- Detecting edges and corners using morphological filters
- Segmenting images using watersheds
- Extracting distinctive regions using MSER
- Extracting foreground objects with the GrabCut algorithm

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Eroding and dilating images using morphological filters

Erosion and dilation are the most fundamental morphological operators. Therefore, we will present these in the first recipe of this chapter. The fundamental component in mathematical morphology is the *structuring element*. A structuring element can be simply defined as a configuration of pixels (the square shape in the following diagram) on which an origin is defined (also called an **anchor point**). Applying a morphological filter consists of probing each pixel of the image using this structuring element. When the origin of the structuring element is aligned with a given pixel, its intersection with the image defines a set of pixels on which a particular morphological operation is applied (the nine shaded pixels in the following figure). In principle, the structuring element can be of any shape, but most often, a simple shape such as a square, circle, or diamond with the origin at the center is used (mainly for efficiency reasons):



Let's take a look at a few concepts before starting our recipe.

Getting ready

Since morphological filters often work on binary images, we will use the binary image that was created through thresholding in the first recipe of `Chapter 4`, *Counting the Pixels with Histograms*. However, since the convention is to have the foreground objects represented by high (white) pixel values and the background objects by low (black) pixel values in morphology, we have negated the image. In morphological terms, the following image is said to be the complement of the image that was created in `Chapter 4`, *Counting the Pixels with Histograms*:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's start with our recipe.

How to do it...

Erosion and dilation are implemented in OpenCV as simple functions, which are `cv::erode` and `cv::dilate`. Their usage is straightforward:

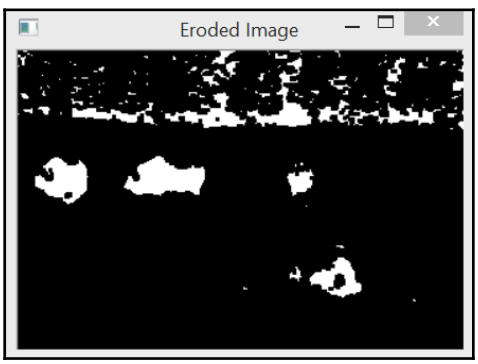
1. First, we have to read an image:

```
// Read input image
cv::Mat image= cv::imread("binary.bmp");
```

2. To apply the `cv::erode` function, we use the `erode` function, as follows:

```
// Erode the image
cv::Mat eroded;  // the destination image
cv::erode(image,eroded,cv::Mat());
```

We obtain the following result after applying the erode morphological filter:

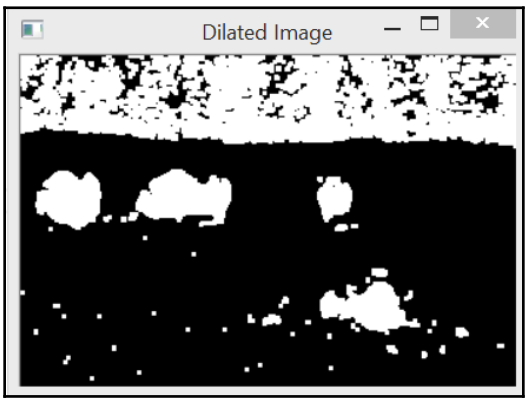


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. To dilat the image, we use `cv::dilate` as follows:

```
// Dilate the image
cv::Mat dilated;  // the destination image
cv::dilate(image,dilated,cv::Mat());
```

This is the final result of applying the `dilate` operation:



Now, let's go behind the scenes to understand the code better.

How it works...

As with all the other morphological filters, the two filters of this recipe operate on the set of pixels (or the neighborhood) around each pixel as defined by the structuring element. Recall that when applied to a given pixel, the anchor point of the structuring element is aligned with this pixel location, and all the pixels that intersect the structuring element are included in the current set. *Erosion* replaces the current pixel with the minimum pixel value found in the defined pixel set. *Dilation* is the complementary operator, and it replaces the current pixel with the maximum pixel value found in the defined pixel set. Since the input binary image contains only black (`0`) and white (`255`) pixels, each pixel is replaced by either a white or black pixel.

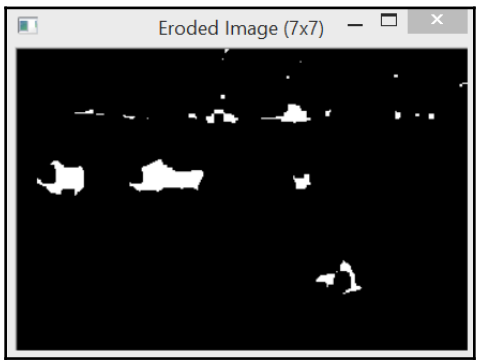
A good way to picture the effect of these two operators is to think in terms of background (black) and foreground (white) objects. With erosion, if the structuring element, when placed at a given pixel location, touches the background (that is, one of the pixels in the intersecting set is black), then this pixel will be sent to the background. In the case of dilation, if the structuring element on a background pixel touches a foreground object, then this pixel will be assigned a white value.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This explains why the size of the objects has been reduced (the shape has been eroded) in the eroded image. Note how some of the small objects (which can be considered as *noisy* background pixels) have also been completely eliminated. Similarly, the dilated objects are now larger, and some of the *holes* inside them have been filled. By default, OpenCV uses a 3 x 3 square structuring element. This default structuring element is obtained when an empty matrix (that is, `cv::Mat()`) is specified as the third argument in the function call, as it was in the preceding example. You can also specify a structuring element of the size (and shape) you want by providing a matrix in which the nonzero element defines the structuring element. In the following example, a 7 x 7 structuring element is applied:

```
cv::Mat element(7,7,CV_8U,cv::Scalar(1));
cv::erode(image,eroded,element);
```

The effect is much more destructive in this case, as shown in the following screenshot:



Another way to obtain the same result is to apply the same structuring element on an image, repetitively. The two functions have an optional parameter to specify the number of repetitions:

```
// Erode the image 3 times.
cv::erode(image,eroded,cv::Mat(),cv::Point(-1,-1),3);
```

The origin argument `cv::Point(-1,-1)` means that the origin is at the center of the matrix (default); it can be defined anywhere on the structuring element. The image that is obtained will be identical to the image we obtained with the 7 x 7 structuring element. Indeed, eroding an image twice is similar to eroding an image with a structuring element dilated with itself. This also applies to dilation.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, since the notion of background/foreground is arbitrary, we can make the following observation (which is a fundamental property of the erosion/dilation operators). Eroding the foreground objects with a structuring element can be seen as a dilation of the background part of the image. In other words, we can make the following observations:

- The erosion of an image is equivalent to the complement of the dilation of the complement image
- The dilation of an image is equivalent to the complement of the erosion of the complement image

There's more...

Note that even though we applied our morphological filters on binary images here, these filters can be applied on gray-level or even color images with the same definitions.

Also, note that the OpenCV morphological functions support in-place processing. This means that you can use the input image as the destination image, as follows:

```
cv::erode(image,image,cv::Mat());
```

OpenCV will create the required temporary image for you for this to work properly.

See also

- The *Opening and closing images using morphological filters* recipe applies the erosion and dilation filters in cascade to produce new operators
- The *Detecting edges and corners using morphological filters* recipe applies morphological filters on gray-level images

We've successfully learned how to erode and dilate images using morphological filters. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Opening and closing images using morphological filters

The previous recipe introduced you to the two fundamental morphological operators—dilation and erosion. From these, other operators can be defined. This and the following recipe will present some of them. The opening and closing operators are presented in this recipe.

How to do it...

In order to apply higher level morphological filters, you need to use the `cv::morphologyEx` function with the appropriate function code:

1. To create a closing or opening operator, we have to create a `cv::Mat` element that will be used as an opening/close kernel:

```
cv::Mat element5(5,5,CV_8U,cv::Scalar(1));
```

2. Now, we have to create another `cv::Mat` to store the result of applying our morphological operator:

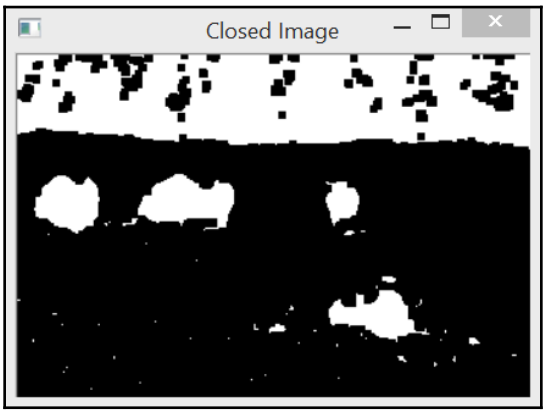
```
cv::Mat closed;
cv::Mat opened;
```

3. Finally, we can apply our closing or opening operator. To make a closing operator, we use the `cv::morphologyEx` function using the `cv::MORPH_CLOSE` parameter, as follows:

```
cv::morphologyEx(image,closed,cv::MORPH_CLOSE,element5);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

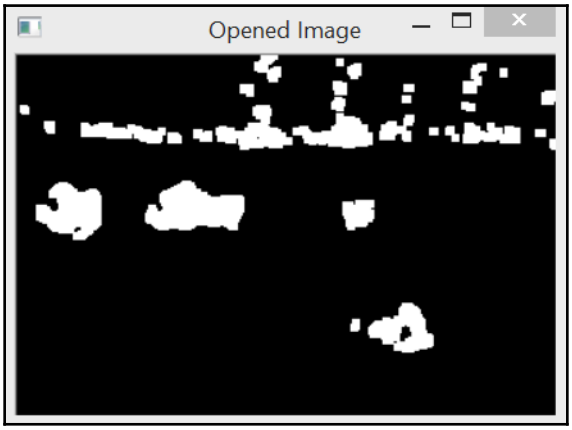
If we use the binary image of the preceding recipe as input, we will obtain an image similar to the image in the following screenshot for the closing operation:



4. If we use `cv::MORPH_OPEN` as a parameter, we will apply the opening morphological operation:

```
cv::morphologyEx(image,opened,cv::MORPH_OPEN,element5);
```

Similarly, applying the morphological opening operator will result in the following image:



Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

The opening and closing filters are simply defined in terms of the basic erosion and dilation operations. **Closing** is defined as the erosion of the dilation of an image. **Opening** is defined as the dilation of the erosion of an image.

Consequently, you can compute the closing of an image using the following calls:

```
// dilate original image
cv::dilate(image,result,cv::Mat());
// in-place erosion of the dilated image
cv::erode(result,result,cv::Mat());
```

The opening filter can be obtained by reverting these two function calls. While examining the result of the closing filter, it can be seen that the small holes of the white foreground objects have been filled. The filter also connects several adjacent objects together. Basically, any holes or gaps that are too small to contain the structuring element completely will be eliminated by the filter.

Reciprocally, the opening filter eliminated several small objects from the scene. All the objects that were too small to contain the structuring element have been removed.

These filters are often used in object detection. The closing filter connects the objects that have been erroneously fragmented into smaller pieces together, while the opening filter removes the small blobs that was introduced by the image noise. Therefore, it is advantageous to use them in a sequence. If our test binary image is successively closed and opened, we obtain an image that shows only the main objects in the scene, as shown in the following screenshot. You can also apply the opening filter before the closing filter if you wish to prioritize noise filtering, but this will be at the price of eliminating some fragmented objects:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that applying the same opening (and similarly, the closing) operator on an image several times has no effect. Indeed, since the holes have been filled by the first opening filter, an additional application of the same filter will not produce any other changes to the image. In mathematical terms, these operators are said to be idempotent.

See also

- The opening and closing operators are often used to clean up an image before extracting its connected components, as explained in the *Extracting the components' contours* recipe in `Chapter 7`, *Extracting Lines, Contours, and Components*.

We've successfully learned how to open and close images using morphological filters. Now, let's move on to the next recipe!

Detecting edges and corners using morphological filters

Morphological filters can also be used to detect specific features in an image. In this recipe, we will learn how to detect contours and corners in a gray-level image.

Getting ready

In this recipe, the following image will be used:

Let's start with our recipe.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Let's have a look at the following code:

1. The edges of an image to be detected can be extracted by using the appropriate filter of the `cv::morphologyEx` function. Refer to the following code:

```
// Get the gradient image using a 3x3 structuring element
cv::Mat result;
cv::morphologyEx(image,result,
                         cv::MORPH_GRADIENT,cv::Mat());

// Apply threshold to obtain a binary image
int threshold= 40;
cv::threshold(result, result,
                    threshold, 255, cv::THRESH_BINARY);
```

   The following image is obtained as a result:

2. In order to detect corners using morphology, we can define a class named `MorphoFeatures`, as follows:

```
class MorphoFeatures {

  private:

     // threshold to produce binary image
     int threshold;
     // structuring elements used in corner detection
    cv::Mat_<uchar> cross;
    cv::Mat_<uchar> diamond;
    cv::Mat_<uchar> square;
    cv::Mat_<uchar> x;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The detection of corners using morphological corners is a bit complex since it requires the successive application of several different morphological filters. This is a good example of the use of nonsquare structuring elements. Indeed, this requires four different structuring elements shaped as a square, diamond, cross, and X-shape to be defined in the constructor (all these structuring elements have a fixed 5 x 5 dimension for simplicity):

```
MorphoFeatures() : threshold(-1),
      cross(5, 5), diamond(5, 5), square(5, 5), x(5, 5) {
      // Creating the cross-shaped structuring element
      cross <<
        0, 0, 1, 0, 0,
        0, 0, 1, 0, 0,
        1, 1, 1, 1, 1,
        0, 0, 1, 0, 0,
        0, 0, 1, 0, 0;
      // Similarly creating the other elements
```

4. In the detection of corner features, all these structuring elements are applied in a cascade to obtain the resulting corner map:

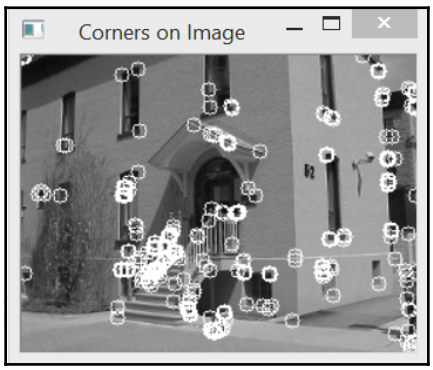
```
cv::Mat getCorners(const cv::Mat &image) {

   cv::Mat result;

   // Dilate with a cross
   cv::dilate(image,result,cross);

   // Erode with a diamond
   cv::erode(result,result,diamond);

   cv::Mat result2;
   // Dilate with a X
   cv::dilate(image,result2,x);

   // Erode with a square
   cv::erode(result2,result2,square);

   // Corners are obtained by differencing
   // the two closed images
   cv::absdiff(result2,result,result);

   // Apply threshold to obtain a binary image
   applyThreshold(result);

   return result;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. The corners are then detected on an image by using the following code:

```
// Get the corners
cv::Mat corners;
corners= morpho.getCorners(image);

// Display the corner on the image
morpho.drawOnImage(corners,image);
cv::namedWindow("Corners on Image");
cv::imshow("Corners on Image",image);
```

In the image, the detected corners are displayed as circles, as shown in the following screenshot:



You can obtain the full code from the book's GitHub repository.

How it works...

A good way to understand the effect of morphological operators on a gray-level image is to consider an image as a topological relief in which the gray levels correspond to elevation (or altitude). From this perspective, the bright regions correspond to mountains, while the dark areas correspond to the valleys of the terrain. Also, since edges correspond to a rapid transition between the dark and bright pixels, these can be pictured as abrupt cliffs. If an erosion operator is applied on such a terrain, the net result will be to replace each pixel by the lowest value in a certain neighborhood, thus reducing its height. As a result, cliffs will be *eroded* as the valleys expand. Dilation has the exact opposite effect; that is, cliffs will gain terrain over the valleys. However, in both cases, the plateaux (that is, the areas of constant intensity) will remain relatively unchanged.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

These observations lead to a simple way to detect the edges (or cliffs) of an image. This can be done by computing the differences between the dilated and eroded images. Since these two transformed images differ mostly at the edge locations, the image edges will be emphasized by the subtraction. This is exactly what the `cv::morphologyEx` function does when the `cv::MORPH_GRADIENT` argument is inputted. Obviously, the larger the structuring element is, the thicker the detected edges will be. This edge detection operator is also called the **Beucher** gradient (`Chapter 6`, *Filtering the images*, will discuss the concept of an image gradient in more detail). Note that similar results can also be obtained by simply subtracting the original image from the dilated one or the eroded image from the original. The resulting edges would be thinner.

Corner detection is a bit more complex since it uses four different structuring elements. This operator is not implemented in OpenCV, but we present it here to demonstrate how the structuring elements of various shapes can be defined and combined. The idea is to close the image by dilating and eroding it with two different structuring elements. These elements are chosen so that they leave straight edges unchanged, but because of their respective effects, the edges at corner points will be affected. Let's use the simple following image, made of a single white square, to understand the effect of this asymmetrical closing operation better:

The first square is the original image. When dilated with a cross-shaped structuring element, the square edges expand, except at the corner points where the cross shape does not hit the square. This is the result illustrated by the square in the middle. This dilated image is then eroded by a structuring element that has a diamond shape. This erosion brings back most edges to their original position but pushes the corners even further since they were not dilated. The rightmost square is then obtained, which (as can be seen) has lost its sharp corners. The same procedure is repeated with the X-shaped and square-shaped structuring elements. These two elements are the rotated versions of the previous elements and will consequently capture the corners at a 45-degree orientation. Finally, differencing the two results will extract the corner features.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The *Applying directional filters to detect edges* recipe in `Chapter 6`, *Filtering the Images*, describes the other filters that perform edge detection
- `Chapter 8`, *Detecting Interest Points*, presents different operators that perform corner detection
- *The Morphological gradients* by J.-F. Rivest, P. Soille, and S. Beucher, *ISET's symposium on electronic imaging science and technology, SPIE*, February 1992 article discusses the concept of morphological gradients in more detail
- The *A modified regulated morphological corner detector* by F.Y. Shih, C.-F. Chuang, and V. Gaddipati, *Pattern Recognition Letters, Volume 26, Issue 7*, May 2005 article gives more information on morphological corner detection

We've successfully learned how to detect edges and corners using morphological filters. Now, let's move on to the next recipe!

Segmenting images using watersheds

The watershed transformation is a popular image-processing algorithm that is used to segment an image into homogenous regions quickly. It relies on the idea that when the image is seen as a topological relief, the homogeneous regions correspond to relatively flat basins, delimited by steep edges. As a result of its simplicity, the original version of this algorithm tends to over-segment the image, which produces multiple small regions. This is why OpenCV proposes a variant of this algorithm that uses a set of predefined markers that guide the definition of the image segments.

How to do it...

The watershed segmentation is obtained through the use of the `cv::watershed` function. The input for this function is a 32-bit, signed, integer-marker image in which each nonzero pixel represents a label.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The idea is to mark some pixels of the image that are known to belong to a given region. From this initial labeling, the watershed algorithm will determine the regions to which the other pixels belong:

1. First, we will create the marker image as a gray-level image and then convert it into an image of integers. We have conveniently encapsulated this step into a `WatershedSegmenter` class. Refer to the following code:

```
class WatershedSegmenter {

  private:

     cv::Mat markers;

  public:

     void setMarkers(const cv::Mat& markerImage) {

      // Convert to image of ints
      markerImage.convertTo(markers,CV_32S);
     }

     cv::Mat process(const cv::Mat &image) {

      // Apply watershed
      cv::watershed(image,markers);

      return markers;
     }
```

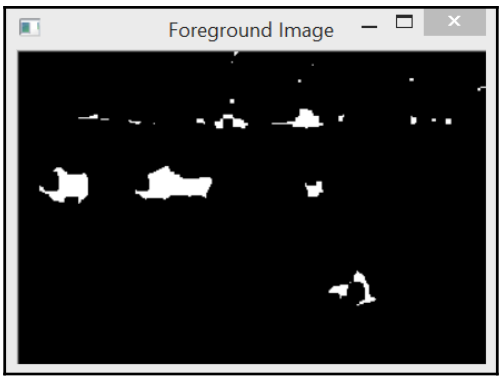
The way these markers are obtained depends on the application. For example, some preprocessing steps might have resulted in the identification of some pixels that belong to an object of interest. The watershed would then be used to delimit the complete object from that initial detection. In this recipe, we will simply use the binary image used throughout this chapter in order to identify the animals of the corresponding original image (this is the image that was shown at the beginning of `Chapter 4`, *Counting the Pixels with Histograms*). Therefore, from our binary image, we need to identify the pixels that belong to the foreground (the animals) and the pixels that belong to the background (mainly the grass). Here, we will mark the foreground pixels with the label `255` and the background pixels with the label `128` (this choice is totally arbitrary; any label number other than `255` will work). The other pixels, that is, the ones for which the labeling is unknown, are assigned the value `0`.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. As of now, the binary image includes too many white pixels that belong to the various parts of the image. We need to severely erode this image in order to retain only the pixels that belong to the important objects:

```
// Eliminate noise and smaller objects
cv::Mat fg;
cv::erode(binary,fg,cv::Mat(),cv::Point(-1,-1),4);
```

The result is the following image:

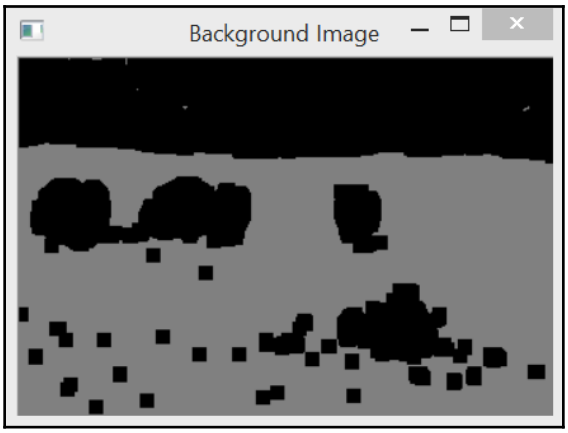


3. Note that a few pixels that belong to the background (forest) are still present. Let's keep them. Therefore, they will be considered to correspond to an object of interest. Similarly, we also select a few pixels of the background by a large dilation of the original binary image:

```
// Identify image pixels without objects
cv::Mat bg;
cv::dilate(binary,bg,cv::Mat(),cv::Point(-1,-1),4);
cv::threshold(bg,bg,1,128,cv::THRESH_BINARY_INV);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

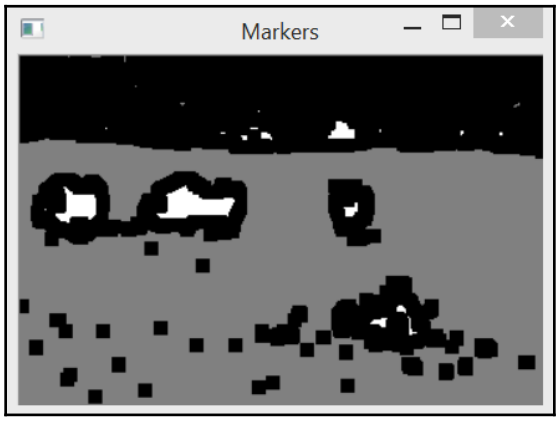
The resulting black pixels correspond to the background pixels. This is why the thresholding operation assigns the value `128` to these pixels immediately after the dilation. The following image is obtained:



4. These images are combined to form the marker image, as follows:

```
// Create markers im  age
cv::Mat markers(binary.size(),CV_8U,cv::Scalar(0));
markers= fg+bg;
```

Note how we used the overloaded + operator here in order to combine the images. The following image will be used as the input to the watershed algorithm:

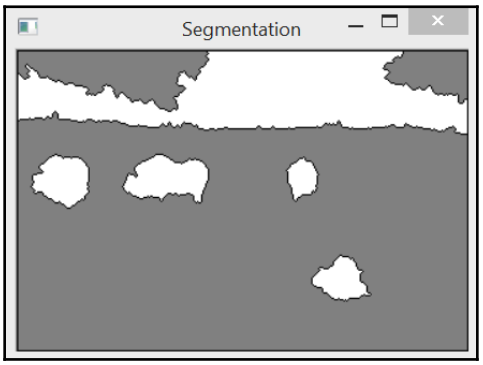


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

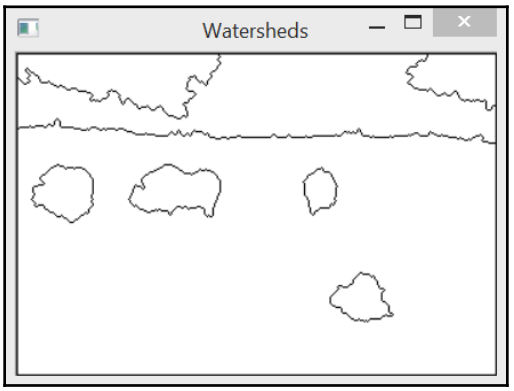
5. In this input image, the white areas belong, for sure, to the foreground objects, the gray areas are a part of the background, and the black areas have an unknown label. The segmentation is then obtained as follows:

```
// Create watershed segmentation object
WatershedSegmenter segmenter;

// Set markers and process
segmenter.setMarkers(markers);
segmenter.process(image);
```

The marker image is then updated so that each of the zero pixels are assigned one of the input labels, while the pixels that belong to the found boundaries have a value of `-1`. The resulting image of the labels is as follows:



The boundary image will be similar to the following screenshot:



Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Like we did in the preceding recipes, we will use the topological map analogy in the description of the watershed algorithm. In order to create a watershed segmentation, the idea is to flood the image progressively starting at level `0`. As the level of "water" progressively increases (to levels 1, 2, 3, and so on), catchment basins are formed. The size of these basins also gradually increases and, consequently, the water of two different basins will eventually merge. When this happens, a watershed is created in order to keep the two basins separate. Once the level of water has reached its maximal level, the sets of these created basins and watersheds form the watershed segmentation.

As expected, the flooding process initially creates many small individual basins. When all of these are merged, many watershed lines are created, which results in an over-segmented image. To overcome this problem, a modification to this algorithm has been proposed in which the flooding process starts from a predefined set of marked pixels. The basins that were created from these markers are labeled in accordance with the values assigned to the initial marks. When two basins having the same label merge, no watersheds are created, thus preventing over-segmentation. This is what happens when the `cv::watershed` function is called. The input marker image is updated to produce the final watershed segmentation. Users can input a marker image with any number of labels and pixels of unknown labeling left to value `0`. The marker image is chosen to be an image of a 32-bit signed integer in order to be able to define more than `255` labels. It also allows the special value, `-1`, to be assigned to the pixels associated with a watershed. This is returned by the `cv::watershed` function.

To facilitate the display of the result, we have introduced two special methods. The first method returns an image of the labels (with watersheds at value `0`). This is easily done through thresholding, as follows:

```
// Return result in the form of an image
cv::Mat getSegmentation() {
 cv::Mat tmp;
 // all segment with label higher than 255
 // will be assigned value 255
 markers.convertTo(tmp,CV_8U);

 return tmp;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Similarly, the second method returns an image in which the watershed lines are assigned the value `0`, and the rest of the image is at `255`. This time, the `cv::convertTo` method is used to achieve this result, as follows:

```
// Return watershed in the form of an image
cv::Mat getWatersheds() {
 cv::Mat tmp;
 // Each pixel p is transformed into
 // 255p+255 before conversion
 markers.convertTo(tmp,CV_8U,255,255);

 return tmp;
}
```

The linear transformation that is applied before the conversion allows the `-1` pixels to be converted into `0` (since *-1*255+255=0*).

Pixels with a value greater than `255` are assigned the value `255`. This is due to the saturation operation that is applied when signed integers are converted into unsigned chars.

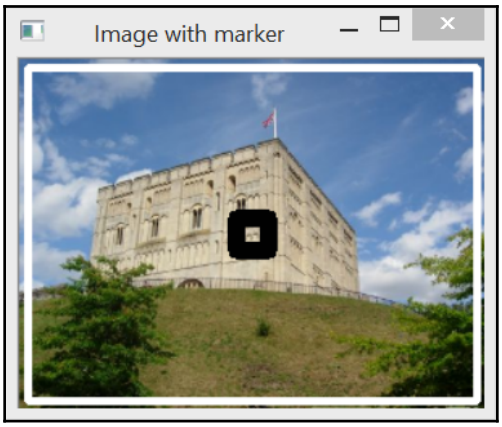
There's more...

Obviously, the marker image can be obtained in many different ways. For example, users can be interactively asked to paint areas on the objects and the background of a scene. Alternatively, in an attempt to identify an object located at the center of an image, you can also simply input an image with the central area marked with a certain label and the border of the image (where the background is assumed to be present) marked with another label. This marker image can be created as follows:

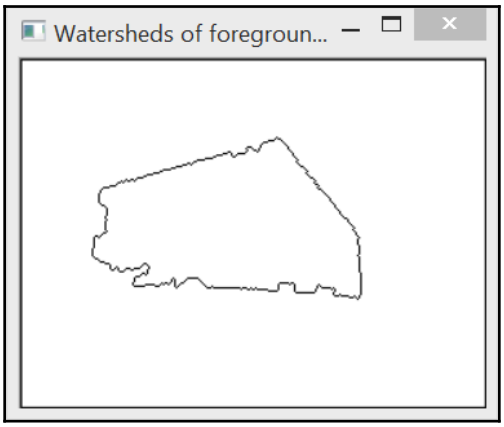
```
// Identify background pixels
cv::Mat imageMask(image.size(),CV_8U,cv::Scalar(0));
cv::rectangle(imageMask,cv::Point(5,5),
  cv::Point(image.cols-5, image.rows-5),
  cv::Scalar(255),3);
// Identify foreground pixels
// (in the middle of the image)
cv::rectangle(imageMask, cv::Point(image.cols/2-10,image.rows/2-10),
  cv::Point(image.cols/2+10,image.rows/2+10), cv::Scalar(1),10);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If we superimpose this marker image on a test image, we will obtain the following image:



The following is the resulting watershed image:



The next section gives some extra information that you can take a look at to enhance your knowledge.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The article, *The Viscous Watershed Transform* by C. Vachier and F. Meyer, *Journal of Mathematical Imaging and Vision, Volume 22, Issue 2-3*, May 2005, gives more information on the watershed transform
- The last recipe of this chapter, *Extracting foreground objects with the GrabCut algorithm*, presents another image segmentation algorithm that can also segment an image into background and foreground objects

We've successfully learned how to segment images using watersheds. Now, let's move on to the next recipe!

Extracting distinctive regions using MSER

In the previous recipe, you learned how an image can be segmented into regions by gradually flooding it and creating watersheds. The **maximally stable external regions** (**MSER**) algorithm uses the same immersion analogy in order to extract meaningful regions in an image. These regions will also be created by flooding the image level by level, but this time, we will be interested in the basins that remain relatively stable for a period of time during the immersion process. It will be observed that these regions correspond to some distinctive parts of the scene objects that are pictured in the image.

How to do it...

The basic class to compute the MSER of an image is `cv::MSER`. An instance of this class can be created by using the default empty constructor, as follows:

1. In our case, we chose to initialize it by specifying a minimum and maximum size for the detected regions in order to limit their number. Then, our call will be as follows:

```
// basic MSER detector
cv::MSER mser(5, // delta value for extremal region detection
              200,   // min acceptable area
             1500); // max acceptable area
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Now, the MSER can be obtained by a call to a functor, specifying the input image and an appropriate output data structure, as follows:

```
// vector of point sets
std::vector<std::vector<cv::Point>> points;
// detect MSER features
mser(image, points);
```

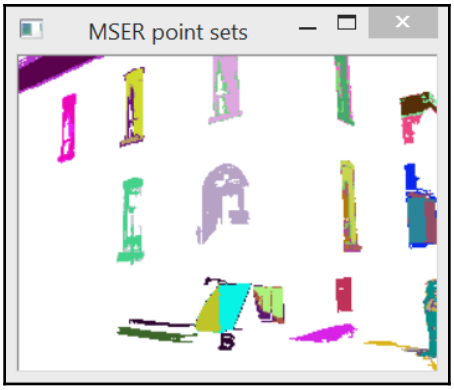
3. The result is a vector of regions represented by the pixel points that compose each of them. In order to visualize the results, we create a blank image on which we will display the detected regions in different colors (which are randomly chosen). This is done as follows:

```
// create white image
cv::Mat output(image.size(),CV_8UC3);
output= cv::Scalar(255,255,255);
// random number generator
cv::RNG rng;

// for each detected feature
for (std::vector<std::vector<cv::Point>>::
          iterator it= points.begin();
       it!= points.end(); ++it) {

// generate a random color
cv::Vec3b c(rng.uniform(0,255),
           rng.uniform(0,255),
           rng.uniform(0,255));

// for each point in MSER set
for (std::vector<cv::Point>::
             iterator itPts= it->begin();
          itPts!= it->end(); ++itPts) {

//do not overwrite MSER pixels
if (output.at<cv::Vec3b>(*itPts)[0]==255) {

    output.at<cv::Vec3b>(*itPts)= c;
        }
    }
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that the MSER form a hierarchy of regions. Therefore, to make all of these visible, we have chosen not to overwrite the small regions when they are included in larger ones. Let's say that the MSER are detected on the following image:

The resulting image would be as follows (refer to this book's graphics PDF to view this image in color):



These are the raw results of the detection. Nevertheless, it can be observed how this operator has been able to extract some meaningful regions (for example, the building's windows) from this image.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

The MSER algorithm uses the same mechanism as the watershed algorithm; that is, it proceeds by gradually flooding the image from level `0` to level `255`. As the level of water increases, you can observe that the sharply delimited, darker areas form the basins that have a relatively stable shape for a period of time (recall that under the immersion analogy, the water levels correspond to the intensity levels). These stable basins are the MSER. These are detected by considering the connected regions at each level and measuring their stability. This is done by comparing the current area of a region with the area it previously had when the level was down by a value of delta. When this relative variation reaches a local minimum, the region is identified as one of the MSER. The delta value that is used to measure the relative stability is the first parameter in the constructor of the `cv::MSER` class; its default value is `5`. In addition, the size of a region must be within a certain predefined range. The acceptable minimum and maximum region sizes are the next two parameters of the constructor. We must also ensure that the MSER is stable (the fourth parameter), that is, the relative variation of its shape is small enough. The stable regions can be included in the larger regions (called **parent regions**).

To be valid, a parent MSER must be sufficiently different from its child; this is the diversity criterion, and it is specified by the fifth parameter of the `cv::MSER` constructor. In the example we used in the previous section, the default value for these last two parameters were used. (The default values are `0.25` for the maximum allowable variation of an MSER and `0.2` for the minimum diversity of a parent MSER.)

The output of the MSER detector is a vector of point sets. Since we are generally more interested in a region as a whole rather than its individual pixel locations, it is common to represent an MSER by a simple geometrical shape that describes the MSER location and size. A bounding ellipse is a commonly used representation. In order to obtain these ellipses, we will make use of two convenient OpenCV functions. The first is the `cv::minAreaRect` function that finds the rectangle of minimum area and binds all the points in a set. This rectangle is described by a `cv::RotatedRect` instance. Once this bounding rectangle is found, it is possible to draw the inscribed ellipse on the image by using the `cv::ellipse` function. Let's encapsulate this complete process in one class. The constructor of this class basically repeats the one of the `cv::MSER` class. Refer to the following code:

```
class MSERFeatures {

  private:

    cv::MSER mser;        // mser detector
    double minAreaRatio;  // extra rejection parameter
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
  public:

    MSERFeatures(
           // aceptable size range
           int minArea=60, int maxArea=14400,
           // min value for MSER area/bounding-rect area
         double minAreaRatio=0.5,
           // delta value used for stability measure
        int delta=5,
           // max allowed area variation
        double maxVariation=0.25,
           // min size increase between child and parent
        double minDiversity=0.2):
              mser(delta,minArea,maxArea,
              maxVariation,minDiversity),
              minAreaRatio(minAreaRatio) {}
```

One extra parameter (`minAreaRatio`) has been added to eliminate the MSER for which the bounding rectangle has an area that differs greatly from the one of the MSER it represents. This is to remove the less interesting elongated shapes.

The list of representative bounding rectangles is computed by the following method:

```
// get the rotated bounding rectangles
// corresponding to each MSER feature
// if (mser area / bounding rect area) < areaRatio,
// the feature is rejected
void getBoundingRects(const cv::Mat &image,
                      std::vector<cv::RotatedRect> &rects) {

  // detect MSER features
  std::vector<std::vector<cv::Point>> points;
  mser(image, points);

  // for each detected feature
  for (std::vector<std::vector<cv::Point>>::
            iterator it= points.begin();
         it!= points.end(); ++it) {
        // Extract bouding rectangles
        cv::RotatedRect rr= cv::minAreaRect(*it);
        // check area ratio
        if (it->size() > minAreaRatio*rr.size.area())
        rects.push_back(rr);
  }
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The corresponding ellipses are drawn on the image using the following method:

```
// draw the rotated ellipses corresponding to each MSER
cv::Mat getImageOfEllipses(const cv::Mat &image,
            std::vector<cv::RotatedRect> &rects,
            cv::Scalar color=255) {

  // image on which to draw
  cv::Mat output= image.clone();

  // get the MSER features
  getBoundingRects(image, rects);

  // for each detected feature
  for (std::vector<cv::RotatedRect>::
            iterator it= rects.begin();
         it!= rects.end(); ++it) {

       cv::ellipse(output,*it,color);
  }

  return output;
}
```

The detection of the MSER is then obtained as follows:

```
  // create MSER feature detector instance
  MSERFeatures mserF(200,  // min area
                     1500, // max area
                0.5); // ratio area threshold
                          // default delta is used

  // the vector of bounding rotated rectangles
  std::vector<cv::RotatedRect> rects;

  // detect and get the image
  cv::Mat result= mserF.getImageOfEllipses(image,rects);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

By applying this function to the previously used image, we get the following image:

Comparing this result with the previous result should convince you that this later representation is easier to interpret. Note how the child and parent MSER are often represented by very similar ellipses. In some cases, it would then be interesting to apply a minimum variation criterion on these ellipses in order to eliminate these repeated representations.

See also

- The *Computing components' shape descriptors* recipe in `Chapter 7`, *Extracting Lines, Contours, and Components*, which will show you how to compute other properties of connected point sets
- `Chapter 8`, *Detecting Interest Points*, will explain how to use MSER as an interest point detector

We've successfully learned how to extract distinctive regions using MSER. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Extracting foreground objects with the GrabCut algorithm

OpenCV proposes the implementation of another popular algorithm for image segmentation—the **GrabCut** algorithm. This algorithm is not based on mathematical morphology, but we have presented it here since it shows some similarities in its use with the watershed segmentation algorithm we presented earlier in this chapter. GrabCut is computationally more expensive than watershed, but it generally produces more accurate results. It is the best algorithm to use when you want to extract a foreground object in a still image (for example, to cut and paste an object from one picture to another).

How to do it...

The `cv::grabCut` function is easy to use. You just need to input an image and label some of its pixels so that they belong to the background or to the foreground. Based on this partial labeling, the algorithm will then determine a foreground/background segmentation for the complete image. Let's have a look at the following steps:

1. One way to specify a partial foreground/background labeling for an input image is by defining a rectangle inside which the foreground object is included:

```
// define bounding rectangle
// the pixels outside this rectangle
// will be labeled as background
cv::Rect rectangle(5,70,260,120);
```

On executing the code, we get the following image:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. All the pixels outside this rectangle will then be marked as the background. In addition to the input image and its segmentation image, calling the `cv::grabCut` function requires the definition of two matrices, which will contain the models, built by the algorithm, as follows:

```
cv::Mat result; // segmentation (4 possible values)
cv::Mat bgModel,fgModel; // the models (internally used)
// GrabCut segmentation
cv::grabCut(image,    // input image
         result,      // segmentation result
         rectangle,   // rectangle containing foreground
         bgModel,fgModel, // models
         5,           // number of iterations
         cv::GC_INIT_WITH_RECT); // use rectangle
```

3. Note how we specified that we are using the bounding rectangle mode using the `cv::GC_INIT_WITH_RECT` flag as the last argument of the function (the next section will discuss the other available mode). The input/output segmentation image can have one of the following four values:

   - `cv::GC_BGD`: This is the value for the pixels that certainly belong to the background (for example, pixels outside the rectangle in our example)
   - `cv::GC_FGD`: This is the value for the pixels that certainly belong to the foreground (there are none in our example)
   - `cv::GC_PR_BGD`: This is the value for the pixels that probably belong to the background
   - `cv::GC_PR_FGD`: This is the value for the pixels that probably belong to the foreground (that is, the initial value for the pixels inside the rectangle in our example)

4. We get a binary image of the segmentation by extracting the pixels that have a value equal to `cv::GC_PR_FGD`. Refer to the following code:

```
// Get the pixels marked as likely foreground
cv::compare(result,cv::GC_PR_FGD,result,cv::CMP_EQ);
// Generate output image
cv::Mat foreground(image.size(),CV_8UC3,
                   cv::Scalar(255,255,255));
image.copyTo(foreground,// bg pixels are not copied
             result);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. To extract all the foreground pixels, that is, with values equal to `cv::GC_PR_FGD` or `cv::GC_FGD`, it is possible to check the value of the first bit, as follows:

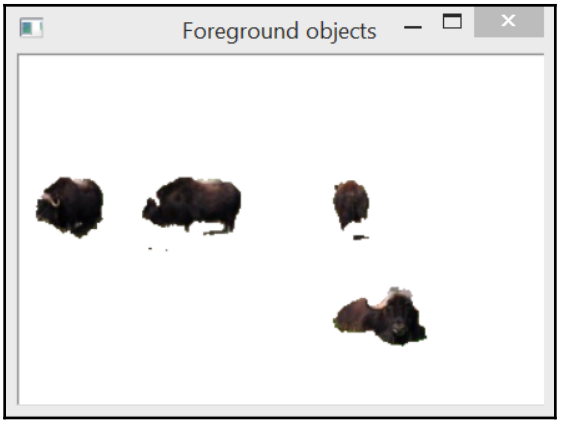
```
// checking first bit with bitwise-and
result= result&1; // will be 1 if FG
```

This is possible because these constants are defined as values `1` and `3`, while the other two (`cv::GC_BGD` and `cv::GC_PR_BGD`) are defined as `0` and `2`. In our example, the same result is obtained because the segmentation image does not contain the `cv::GC_FGD` pixels (only the `cv::GC_BGD` pixels have been inputted).

6. Finally, we obtain an image of the foreground objects (over a white background) by using the following copy operation with a mask:

```
// Generate output image
cv::Mat foreground(image.size(),CV_8UC3,
          cv::Scalar(255,255,255)); // all white image
image.copyTo(foreground,result); // bg pixels not copied
```

The following image is obtained as a result:



Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

In the preceding example, the GrabCut algorithm was able to extract the foreground objects by simply specifying a rectangle inside which these objects (the four animals) were contained. Alternatively, you could also assign the values `cv::GC_BGD` and `cv::GC_FGD` to some specific pixels of the segmentation image, which are provided as the second argument of the `cv::grabCut` function. You would then specify `GC_INIT_WITH_MASK` as the input mode flag. These input labels could be obtained, for example, by asking a user to mark a few elements of the image. It is also possible to combine these two input modes.

Using this input information, the GrabCut algorithm creates the background/foreground segmentation by proceeding as follows. Initially, a foreground label (`cv::GC_PR_FGD`) is tentatively assigned to all the unmarked pixels. Based on the current classification, the algorithm groups the pixels into clusters of similar colors (that is, `K` clusters for the background and `K` clusters for the foreground). The next step is to determine a background/foreground segmentation by introducing boundaries between the foreground and background pixels. This is done through an optimization process that tries to connect pixels with similar labels, and that imposes a penalty for placing a boundary in the regions of relatively uniform intensity. This optimization problem can be efficiently solved using the *Graph Cuts* algorithm, a method that can find the optimal solution of a problem by representing it as a connected graph on which cuts are applied in order to compose an optimal configuration. The obtained segmentation produces new labels for the pixels. The clustering process can then be repeated, and a new optimal segmentation is found again, and so on. Therefore, the GrabCut algorithm is an iterative procedure that gradually improves the segmentation result. Depending on the complexity of the scene, a good solution can be found in more or less numbers of iterations (in easy cases, one iteration would be enough).

This explains the argument of the function where the user can specify the number of iterations to be applied. The two internal models maintained by the algorithm are passed as an argument of the function (and returned). Therefore, it is possible to call the function with the models of the last run again if you wish to improve the segmentation result by performing additional iterations.

See also

- The *GrabCut: Interactive Foreground Extraction using Iterated Graph Cuts in ACM Transactions on Graphics (SIGGRAPH) Volume 23, Issue 3*, August 2004 article by C. Rother, V. Kolmogorov, and A. Blake describes the GrabCut algorithm in detail

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6
Filtering the Images

Filtering is one of the fundamental tasks in signal and image processing. It is a process aimed at selectively extracting certain aspects of an image that are considered to convey important information in the context of a given application. Filtering removes noise in images, extracts interesting visual features, allows image resampling, and so on. It finds its roots in the general **signals and systems** theory. We will not cover this theory in detail here. However, this chapter will present some of the important concepts related to filtering and will show you how filters can be used in image-processing applications. But first, let's begin with a brief explanation of the concept of **frequency domain analysis**.

When we look at an image, we observe how the different gray levels (or colors) are distributed over the image. Images differ from each other because they have different gray-level distributions. However, there exists another point of view under which an image can be analyzed. We can look at the gray-level variations that are present in an image. Some images contain large areas of almost constant intensity (for example, a blue sky), while in other images, the gray-level intensities vary rapidly over the image (for example, a busy scene crowded with many small objects). Therefore, observing the frequency of these variations in an image constitutes another way of characterizing an image. This point of view is referred to as the **frequency domain**, while characterizing an image by observing its gray-level distribution is referred to as the **spatial domain**.

The frequency domain analysis decomposes an image into its frequency content from the lowest to the highest frequencies. Areas where the image intensities vary slowly contain only low frequencies, while high frequencies are generated by rapid changes in intensities. Several well-known transformations exist, such as the Fourier transform or the cosine transform, which can be used to show the frequency content of an image explicitly. Note that since an image is a two-dimensional entity, it is made of both vertical frequencies (variations in the vertical directions) and horizontal frequencies (variations in the horizontal directions).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Under the frequency domain analysis framework, a **filter** is an operation that amplifies certain bands of the frequencies of an image while blocking (or reducing) other image frequency bands. A low-pass filter is, therefore, a filter that eliminates the high-frequency components of an image and reciprocally, a high-pass filter eliminates the low-pass components. This chapter will present some filters that are frequently used in image processing and will explain their effects when applied to an image.

In this chapter, we will cover the following recipes:

- Filtering images using low-pass filters
- Filtering images using a median filter
- Applying directional filters to detect edges
- Computing the Laplacian of an image

Filtering images using low-pass filters

In this first recipe, we will present some very basic low-pass filters. In the introductory section of this chapter, we learned that the objective of such filters is to reduce the amplitude of the image variations. One simple way to achieve this goal is to replace each pixel by the average value of the pixels around it. By doing this, the rapid intensity variations will be smoothed out and thus replaced by a more gradual transition.

How to do it...

The objective of the `cv::blur` function is to smooth an image by replacing each pixel with the average pixel value computed over a rectangular neighborhood. The following steps will help us as follows:

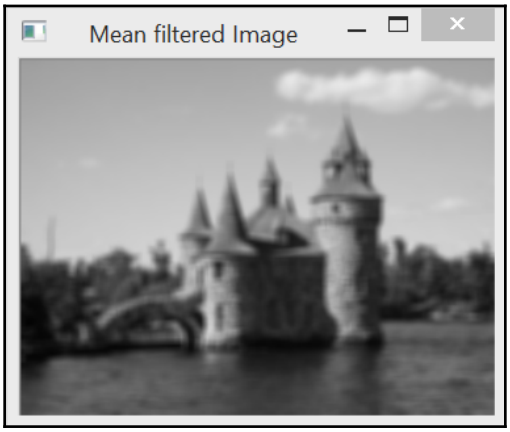
1. To apply a smooth blur function using the average of the `5x5` neighbor pixels that we have, we use the following snippet:

```
cv::blur(image,result, cv::Size(5,5)); // size of the filter
```

   This kind of filter is also called a **box filter**. Here, we applied it by using a `5x5` filter in order to make the filter's effect more visible. Take a look at the following screenshot:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The result of the filter being applied on the preceding image is the following screenshot:

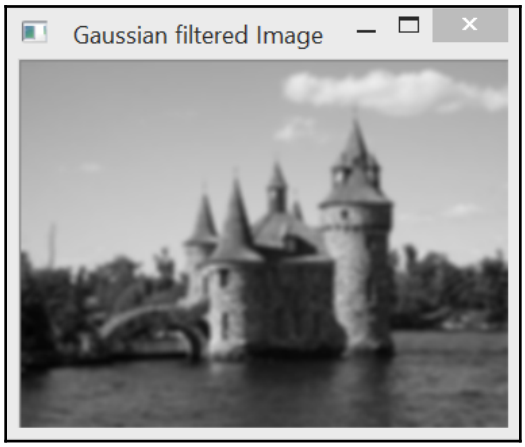


2. In some cases, it might be desirable to give more importance to the closer pixels in the neighborhood of a pixel. Therefore, it is possible to compute a weighted average in which nearby pixels are assigned a larger weight than ones that are further away. This can be achieved by using a weighted scheme that follows a Gaussian function (a *bell-shaped* function). The `cv::GaussianBlur` function applies such a filter and it is called as follows:

```
   cv::GaussianBlur(image,result,
cv::Size(5,5), // size of the filter
1.5); // parameter controlling the shape of the Gaussian
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The result is then seen in the following screenshot:



We will now see how these steps work in the background.

How it works...

A filter is said to be linear if its application corresponds to replacing a pixel with a weighted sum of neighboring pixels. This is the case with the mean filter in which a pixel is replaced by the sum of all pixels in a rectangular neighborhood and divided by the size of this neighborhood (to get the average value). This is like multiplying each neighboring pixel by `1` over the total number of pixels and summing all of these values. The different weights of a filter can be represented using a matrix that shows the multiplying factors associated with each pixel position in the considered neighborhood. The central element of the matrix corresponds to the pixel on which the filter is currently applied. Such a matrix is sometimes called a **kernel** or a **mask**. For a `3x3` mean filter, the corresponding kernel would be as follows:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

The `cv::boxFilter` function filters an image with a square kernel made of many instances of `1` only. It is similar to the mean filter but without dividing the result by the number of coefficients.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Applying a linear filter, then, corresponds to moving a kernel over each pixel of an image and multiplying each corresponding pixel by its associated weight. Mathematically, this operation is called a **convolution** and can be written formally as follows:

$$I_{out}(x,y) = \sum_i \sum_j I_{in}(x-i, y-j)K(i,j)$$

The preceding double summation aligns the current pixel at *(x,y)* with the center of the *K* kernel, which is assumed to be at coordinate (0,0).

Looking at the output images produced in this recipe, it can be observed that the net effect of a low-pass filter is to blur or smooth the image. This is not surprising since this filter attenuates the high-frequency components that correspond to the rapid variations visible on an object's edge.

In the case of a Gaussian filter, the weight associated with a pixel is proportional to its distance from the central pixel. Recall that the 1-D Gaussian function has the following form:

$$G(x) = Ae^{-x^2/2\sigma^2}$$

The normalizing coefficient *A* is chosen such that the different weights sum to 1. The σ (sigma) value controls the width of the resulting Gaussian function. The greater this value is, the flatter the function will be. For example, if we compute the coefficients of the 1-D Gaussian filter for the interval [-4, 0, 4] with σ = 0.5, we obtain the following coefficients:

```
[0.0 0.0 0.00026 0.10645 0.78657 0.10645 0.00026 0.0 0.0]
```

For σ=1.5, these coefficients are as follows:

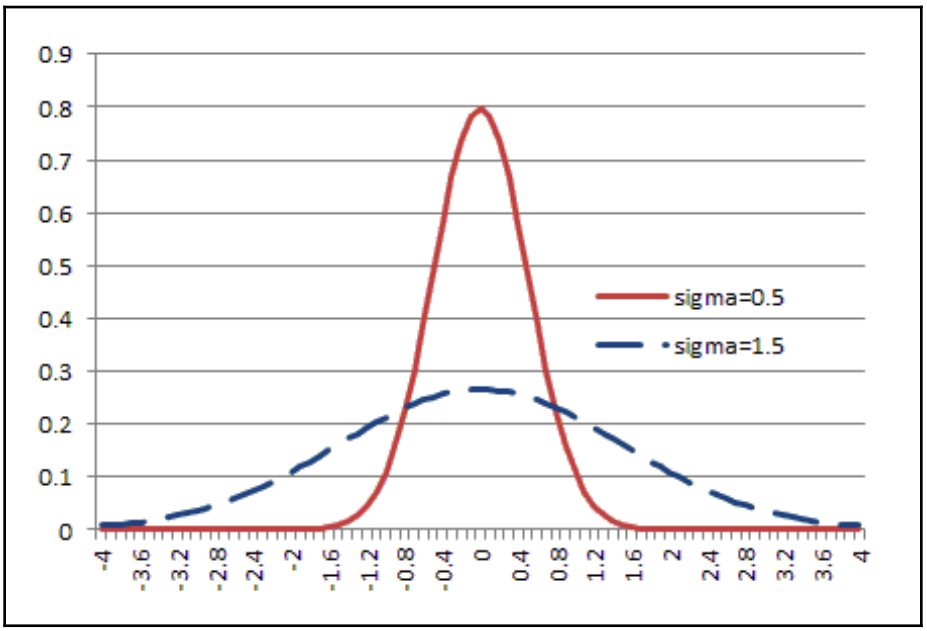
```
[0.00761 0.036075 0.10959 0.21345 0.26666
 0.21345 0.10959 0.03608 0.00761 ]
```

Note that these values were obtained by calling the `cv::getGaussianKernel` function with the appropriate σ value:

```
cv::Mat gauss= cv::getGaussianKernel(9, sigma,CV_32F);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The symmetrical bell shape of the Gaussian function makes it a good choice for filtering. Refer to the following screenshot:



Pixels farther from the center have a lower weight, which makes the pixel-to-pixel transitions smoother. This contrasts with the flat mean filter where pixels far away can cause sudden changes in the current mean value. In terms of frequencies, this implies that the mean filter does not remove all the high-frequency components.

To apply a 2-D Gaussian filter on an image, you can simply apply a 1-D Gaussian filter on the image lines first (to filter the horizontal frequencies), followed by the application of another 1-D Gaussian filter on the image columns (to filter the vertical frequencies). This is possible because the Gaussian filter is a separable filter (that is, the 2-D kernel can be decomposed into two 1-D filters). The `cv::sepFilter2D` function can be used to apply a general, separable filter. It is also possible to apply a 2-D kernel directly using the `cv::filter2D` function. In general, separable filters are faster to compute than non-separable ones because they require fewer multiplication operations.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

With OpenCV, the Gaussian filter to be applied on an image is specified by providing both the number of coefficients (the third parameter, which is an odd number) and the value of σ (the fourth parameter) to `cv::GaussianBlur`. You can also simply set the value of σ and let OpenCV determine the appropriate number of coefficients (you then input a value of `0` for the filter size). The opposite is also possible, where you input a size and a value of `0` for σ. The σ value that best fits the given size will be determined.

See also

- The *Downsampling images with filters* recipe explains how to reduce the size of an image using low-pass filters.
- The *There's more...* section of the *Scanning an image with neighbor access* recipe in `Chapter 2`, *Manipulating the Pixels*, introduces the `cv::filter2D` function. This function lets you apply a linear filter to an image by inputting the kernel of your choice.

We've successfully learned how to filter images using low-pass filters. Now let's move on to the next recipe!

Downsampling an image

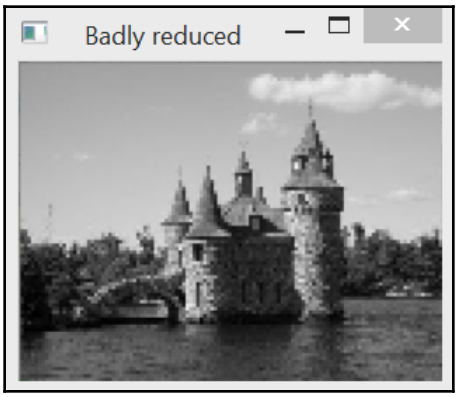
Images often need to be resized (resampled). The process of reducing the size of an image is often called **downsampling**, while increasing its size is called **upsampling**. The challenge in performing these operations is to ensure that the visual quality of the image is preserved as much as possible. To accomplish this objective, low-pass filters are often used; this recipe explains why.

How to do it...

You might think that you can reduce the size of an image by simply eliminating some of the columns and rows of the image. Unfortunately, the resulting image will not look very nice. The following screenshot illustrates this fact by showing you a test image that is reduced by a factor of `4` with respect to its original size by simply keeping `1` of every `4` columns and rows.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that to make the defects in this image more apparent, we zoom the image by displaying it with pixels that are two times larger (the next section explains how this can be done). Refer to the following screenshot:



Clearly, you can see that the image quality has degraded. For example, the oblique edges of the castle's roof in the original image now appear as a staircase on the reduced image. Other jagged distortions are also visible on the textured parts of the image (the brick walls, for instance).

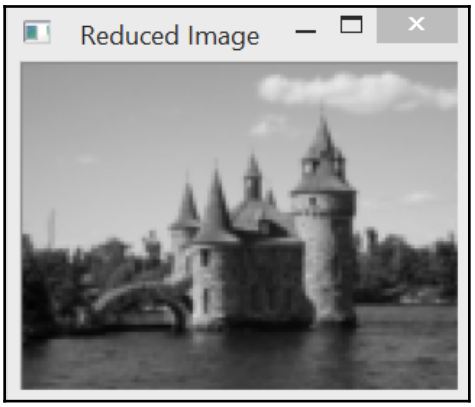
These undesirable artifacts are caused by a phenomenon called **spatial aliasing** that occurs when you try to include high-frequency components in an image that is too small to contain them. Indeed, smaller images (that is, images with fewer pixels) cannot represent fine textures and sharp edges as nicely as the higher resolution images (think of the difference between high-definition TV and conventional TV). Since fine details in an image correspond to high frequencies, we need to remove these higher frequency components in an image before reducing its size. We learned in this recipe that this can be done using a low-pass filter.

1. Consequently, to reduce the size of an image by `4` without adding annoying artifacts, you must first apply a low-pass filter to the original image before throwing away columns and rows. Here is how you would do this using OpenCV:

```
// first remove high frequency component
cv::GaussianBlur(image,image,cv::Size(11,11),2.0);
// keep only 1 of every 4 pixels
cv::Mat reduced2(image.rows/4,image.cols/4,CV_8U);
for (int i=0; i<reduced2.rows; i++)
 for (int j=0; j<reduced2.cols; j++)
   reduced2.at<uchar>(i,j)= image.at<uchar>(i*4,j*4);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The resulting image is shown in this screenshot:



Of course, some of the fine details of the image have been lost, but globally, the visual quality of the image is better preserved than in the previous case.

How it works...

In order to avoid undesirable aliasing effects, an image must always be low-pass filtered before reducing its size. As we explained previously, the role of the low-pass filter is to eliminate the high-frequency components that cannot be represented in the reduced image. The formal theory demonstrating this fact is well established and is often referred to as the **Nyquist-Shannon** theorem. In substance, the theory tells us that if you downsample an image by 2, then the bandwidth of the representable frequencies is also reduced by 2.

A special OpenCV function also performs image reduction. This is the `cv::pyrDown` function:

```
cv::Mat reducedImage;  // to contain reduced image
cv::pyrDown(image,reducedImage); // reduce image size by half
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The preceding function uses a 5 x 5 Gaussian filter to low-pass the image before reducing it by a factor of 2. The reciprocal `cv::pyrUp` function that doubles the size of an image also exists. It is interesting to note that, in this case, the upsampling is done by inserting the `0` values between every two columns and rows and then by applying the same 5 x 5 Gaussian filter (but with the coefficients multiplied by `4`) on the expanded image. Obviously, if you downsize an image and then upsize it, you will not recover the exact original image. What was lost during the downsizing process cannot be recovered. These two functions are used to create **image pyramids**. This is a data structure made of stacked versions of an image at different sizes (here, each level is twice smaller than the previous level, but the reduction factor can be less, for example, `1.2`) that is often built for efficient image analysis. For example, if you wish to detect an object in an image, the detection can first be accomplished on the small image at the top of the pyramid, and as you locate the object of interest, you can refine the search by moving to the lower levels of the pyramid that contains the higher resolution versions of the image.

Note that there is also a more general `cv::resize` function that allows you to specify the size you want for the resulting image. You simply call it by specifying a new size that could be smaller or larger than the original image:

```
cv::Mat resizedImage;  // to contain resized image
cv::resize(image,resizedImage,
     cv::Size(image.cols/4,image.rows/4)); // 1/4 resizing
```

It is also possible to specify resizing in terms of scale factors. In this case, an empty size instance is given as an argument followed by the desired scale factors:

```
cv::resize(image,resizedImage,
           cv::Size(), 1.0/4.0, 1.0/4.0); // 1/4 resizing
```

The last parameter allows you to select the interpolation method that is to be used in the resampling process. This is discussed in the following section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

There's more...

When an image is resized by a factional factor, it becomes necessary to perform some pixel interpolation in order to produce new pixel values at locations that fall in between the existing ones. General image remapping, as discussed in the *Remapping an image* recipe in `Chapter 2`, *Manipulating the Pixels*, is another situation where pixel interpolation is required.

Interpolating pixel values

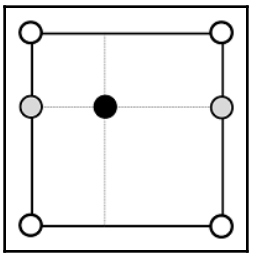
The most basic approach to performing interpolation is to use a **nearest neighbor strategy**. The new grid of pixels that must be produced is placed on top of the existing image, and each new pixel is assigned the value of its closest pixel in the original image. In the case of image upsampling (that is, when using a new grid denser than the original one), this implies that more than one pixel of the new grid will receive its value from the same original pixel. For example, if we rescale the reduced image of the previous section by `3` using nearest neighbor interpolation (which is done by using the interpolation flag `cv::INTER_NEAREST`), we obtain the following code:

```
cv::resize(reduced, newImage, cv::Size(), 3, 3,cv::INTER_NEAREST);
```

The result is shown in the following screenshot:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this case, the interpolation corresponds simply to multiplying the size of each pixel by `3` (this is how we produced the images of the previous section). A better approach consists of interpolating a new pixel value by combining the values of several neighboring pixels. Hence, we can interpolate a pixel value linearly by considering the four pixels around it, as illustrated by the following diagram:



This is done by first vertically interpolating two-pixel values to the left- and right-hand sides of the added pixel. Then, these two interpolated pixels (drawn in gray in the preceding figure) are used to interpolate the pixel value at the desired location horizontally. This bilinear interpolation scheme is the default approach used by `cv::resize` (that can also be explicitly specified by the flag `cv::INTER_LINEAR`):

```
cv::resize(reduced2, newImage,
           cv::Size(), 3, 3, cv::INTER_LINEAR);
```

The following is the result:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Other approaches also exist that can produce superior results. With **bicubic interpolation**, a neighborhood of 4 x 4 pixels is considered to perform the interpolation. However, since the approach uses more pixels and implies the computation of cubic terms, it is slower than bilinear interpolation.

See also

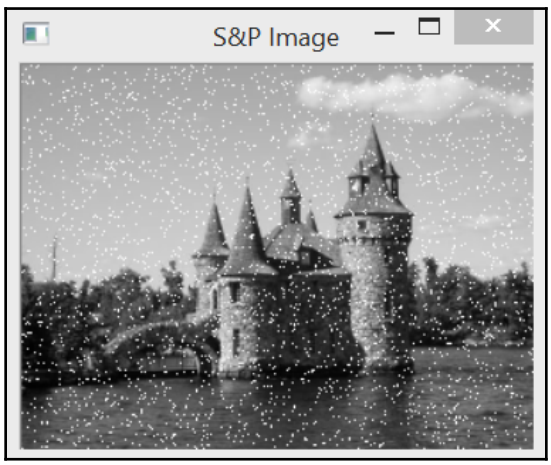
- The *There's more...* section of the *Scanning an image with neighbor access* recipe in `Chapter 2`, *Manipulating the Pixels*, introduces the `cv::filter2D` function. This function lets you apply a linear filter to an image by inputting the kernel of your choice.

We've successfully learned how to downsample an image. Now let's move on to the next recipe!

Filtering images using a median filter

The first recipe of this chapter introduced the concept of linear filters. Nonlinear filters also exist and can be used advantageously in image processing. One such filter is the median filter that we present in this recipe.

Since median filters are particularly useful in order to combat salt-and-pepper noise (or salt-only noise, in our case), we will use the image we created in the first recipe of `Chapter 2`, *Manipulating the Pixels*, and that is reproduced here:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's get started with the steps.

How to do it...

The step to take for this recipe is as follows:

1. The call to the median filtering function is done in a way that is similar to the other filters; we use the `medianBlur` function with kernel size, in our case a size of `5`:

```
cv::medianBlur(image,result,5); // size of the filter
```

The resulting image is shown in this screenshot:

It's time to understand the workings of this process.

How it works...

Since the median filter is not a linear filter, it cannot be represented by a kernel matrix. However, it also operates on a pixel's neighborhood in order to determine the output pixel value. The pixel and its neighborhood form a set of values and, as the name suggests, the median filter will simply compute the median value of this set, and the current pixel is then replaced with this median value (the median of a set is the value at the middle position when the set is sorted).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This explains why the filter is so efficient at eliminating the salt-and-pepper noise. Indeed, when an outlier black or white pixel is present in a given pixel neighborhood, it is never selected as the median value (rather, it is the maximal or minimal value), so it is always replaced by a neighboring value. By contrast, a simple mean filter would be greatly affected by such noise as can be observed in the following image that represents the mean-filtered version of our salt-and-pepper, corrupted image:

Clearly, the noisy pixels shifted the mean value of the neighboring pixels. As a result, the noise is still visible even if it has been blurred by the mean filter.

The median filter also has the advantage of preserving the sharpness of the edges. However, it washes out the textures in uniform regions (for example, the trees in the background). Because of the visual impact, it has on images, the median filter is often used to create special effects in photo-editing software tools. You should test it on a color image to see how it can produce *cartoon-like* images.

We've successfully learned how to filter images using a median filter. Now let's move on to the next recipe!

Applying directional filters to detect edges

The first recipe of this chapter introduced the idea of linear filtering using kernel matrices. The filters that were used had the effect of blurring an image by removing or attenuating its high-frequency components. In this recipe, we will perform the opposite transformation, that is, amplifying the high-frequency content of an image. As a result, the high-pass filters introduced here will perform **edge detection**.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

The filter that we will use here is called the **Sobel** filter. It is said to be a directional filter, because it only affects the vertical or the horizontal image frequencies depending on which kernel of the filter is used. OpenCV has a function that applies the Sobel operator on an image. The following steps will help us to apply the filters as follows:

1. To apply a Sobel operator in the horizontal direction, we have to use the following function and parameters:

```
cv::Sobel(image,    // input
          sobelX,   // output
      CV_8U,     // image type
      1, 0,      // kernel specification
      3,         // size of the square kernel
      0.4, 128); // scale and offset
```

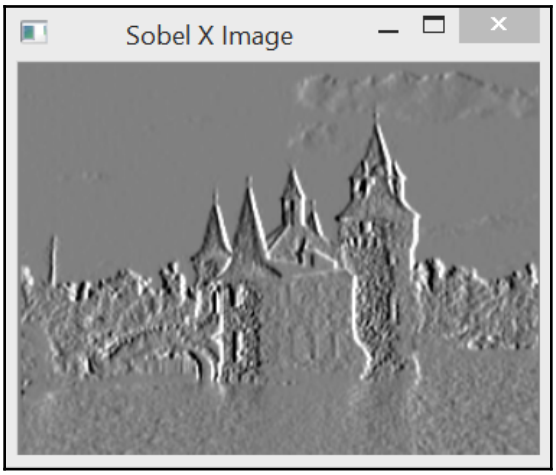
2. Vertical filtering is achieved by the following call (which is very similar to the horizontal filter):

```
cv::Sobel(image,    // input
      sobelY,    // output
      CV_8U,     // image type
      0, 1,      // kernel specification
      3,         // size of the square kernel
      0.4, 128); // scale and offset
```

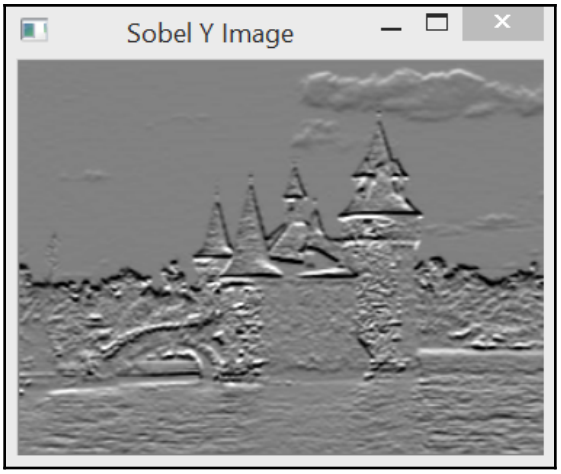
   Several integer parameters are provided to the function and these will be explained in the next section. Note that these have been chosen to produce an 8-bit image (`CV_8U`) representation of the output.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The result of the horizontal Sobel operator is as follows:



Since, as will be seen in the next section, the kernels of the Sobel operator contain both positive and negative values, the result of the Sobel filter is generally computed in a 16-bit signed integer image (`CV_16S`). To make the results displayable as an 8-bit image, as shown in the preceding screenshot, we used a representation in which a zero value corresponds to a gray level of `128`. Negative values are represented by darker pixels, while positive values are represented by brighter pixels. The vertical Sobel image is as follows:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If you are familiar with photo-editing software, the preceding images might remind you of the **image emboss** effect, and, indeed, this image transformation is generally based on the use of directional filters.

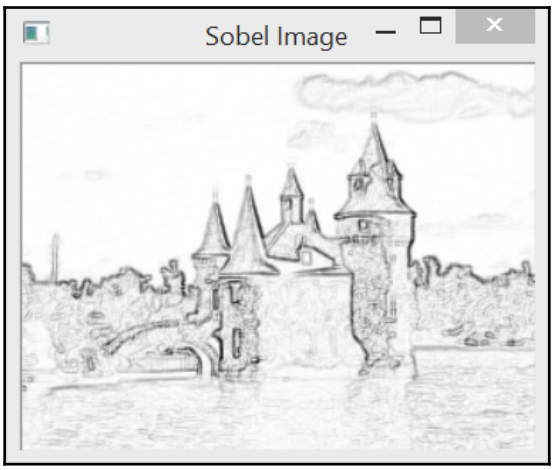
3. The two results (vertical and horizontal) can then be combined to obtain the norm of the Sobel filter:

```
// Compute norm of Sobel
cv::Sobel(image,sobelX,CV_16S,1,0);
cv::Sobel(image,sobelY,CV_16S,0,1);
cv::Mat sobel;
//compute the L1 norm
sobel= abs(sobelX)+abs(sobelY);
```

4. The Sobel norm can be displayed conveniently in an image using the optional rescaling parameter of the `convertTo` method in order to obtain an image in which zero values correspond to white, and higher values are assigned darker gray shades:

```
// Find Sobel max value
double sobmin, sobmax;
cv::minMaxLoc(sobel,&sobmin,&sobmax);
// Conversion to 8-bit image
// sobelImage = -alpha*sobel + 255
cv::Mat sobelImage;
sobel.convertTo(sobelImage,CV_8U,-255./sobmax,255);
```

The result can be seen in the following screenshot:



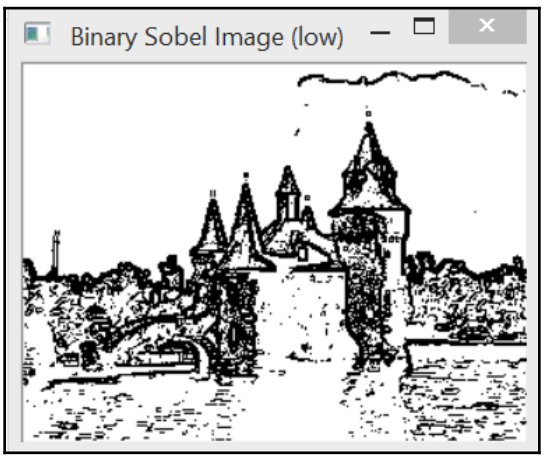
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Looking at this image, it is now clear why these kinds of operators are called edge detectors. It is then possible to threshold this image in order to obtain a binary map that shows you the image contour.

5. The following snippet creates the image that follows it:

```
cv::threshold(sobelImage, sobelThresholded,
                    threshold, 255, cv::THRESH_BINARY);
```

Here is the image after running the preceding snippet:



We will now discuss the steps taken to produce these results.

How it works...

The Sobel operator is a classic edge-detection linear filter that is based on a simple `3x3` kernel that has the following structure:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If we view the image as a two-dimensional function, the Sobel operator can then be seen as a measure of the variation of the image in the vertical and horizontal directions. In mathematical terms, this measure is called a **gradient**, and it is defined as a 2-D vector that is made from the function's first derivatives in two orthogonal directions:

$$grad(I) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right]^T$$

Therefore, the Sobel operator gives you an approximation of the image gradient by differencing pixels in the horizontal and vertical directions. It operates on a window around the pixel of interest in order to reduce the influence of noise. The `cv::Sobel` function computes the result of the convolution of the image with a Sobel kernel. Its complete specification is as follows:

```
cv::Sobel(image,  // input
          sobel,  // output
          image_depth,   // image type
          xorder,yorder, // kernel specification
          kernel_size,   // size of the square kernel
          alpha, beta);  // scale and offset
```

Therefore, you decide whether you wish to have the result written in an unsigned char, a signed integer, or a floating-point image. Of course, if the result falls outside the domain of the image pixel, saturation will be applied. This is where the last two parameters can be useful. Before storing the result in the image, the result can be scaled (multiplied) by `alpha` and an offset, `beta`, can be added. This is how, in the previous section, we generated an image for which the Sobel value `0` was represented by the mid-gray level `128`. Each Sobel mask corresponds to a derivative in one direction. Therefore, two parameters are used to specify the kernel that will be applied: the order of the derivative in the `x` and the `y` directions. For instance, the horizontal Sobel kernel is obtained by specifying `1` and `0` for the `xorder` and `yorder` parameters, and the vertical kernel will be generated with `0` and `1`. Other combinations are also possible, but these two are the ones that will be used most often (the case of second-order derivatives is discussed in the next recipe). Finally, it is also possible to use kernels of sizes that are larger than `3x3`. Values `1`, `3`, `5`, and `7` are possible choices for the kernel size. A kernel of size `1` corresponds to a 1-D Sobel filter (`1x3` or `3x1`). See the following *There's more...* section to learn why using a larger kernel might be useful.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Since the gradient is a 2-D vector, it has a norm and a direction. The norm of the gradient vector tells you what the amplitude of the variation is, and it is normally computed as a Euclidean norm (also called an **L2 norm**):

$$|grad(I)| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

However, in image processing, this norm is often computed as the sum of the absolute values. This is called the **L1 norm**, and it gives values that are close to the L2 norm but at a lower computational cost. This is what we did in this recipe:

```
//compute the L1 norm
sobel= abs(sobelX)+abs(sobelY);
```

The gradient vector always points in the direction of the steepest variation. For an image, this means that the gradient direction will be orthogonal to the edge, pointing in the darker to the brighter direction. Gradient angular direction is given by the following formula:

$$atan(grad(I))$$

Most often, for edge detection, only the norm is computed. However, if you require both the norm and the direction, then the following OpenCV function can be used:

```
// Sobel must be computed in floating points
cv::Sobel(image,sobelX,CV_32F,1,0);
cv::Sobel(image,sobelY,CV_32F,0,1);
// Compute the L2 norm and direction of the gradient
cv::Mat norm, dir;
cv::cartToPolar(sobelX,sobelY,norm,dir);
```

By default, the direction is computed in radians. Just add `true` as an additional argument in order to have it computed in degrees.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

A binary edge map has been obtained by applying a threshold on the gradient magnitude. Choosing the right threshold is not an obvious task. If the threshold value is too low, too many (thick) edges will be retained, while if we select a more severe (higher) threshold, then broken edges will be obtained. As an illustration of this trade-off situation, compare the preceding binary edge map with the following, which is obtained using a higher threshold value:

One way to get the best of both the lower and higher thresholds is to use the concept of hysteresis thresholding. This will be explained in the next chapter where we introduce the Canny operator.

There's more...

Other gradient operators also exist. We will present some of them in this section. It is also possible to apply a Gaussian smoothing filter before applying a derivative filter. This makes it less sensitive to noise, as explained in this section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Gradient operators

To estimate the gradient at a pixel location, the Prewitt operator defines the following kernels:

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

The Roberts operator is based on these simple `2x2` kernels:

1	0
0	-1

0	1
-1	0

The Scharr operator is preferred when more accurate estimates of the gradient orientation are required:

-3	0	3
-10	0	10
-3	0	3

-3	-10	-3
0	0	0
3	10	3

Note that it is possible to use the Scharr kernels with the `cv::Sobel` function by calling it with the `CV_SCHARR` argument:

```
cv::Sobel(image,sobelX,CV_16S,1,0, CV_SCHARR);
```

Or, equivalently, you can call the `cv::Scharr` function:

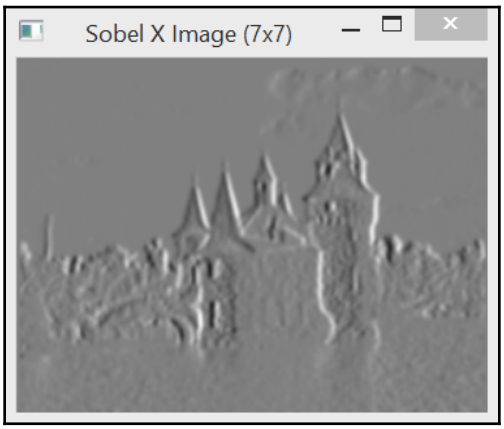
```
cv::Scharr(image,scharrX,CV_16S,1,0,3);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

All of these directional filters try to estimate the first-order derivative of the image function. Therefore, high values are obtained at areas where large-intensity variations in the filter direction are present, while flat areas produce low values. This is why filters that compute image derivatives are high-pass filters.

Gaussian derivatives

Derivative filters are high-pass filters. As such, they tend to amplify noise and small, highly contrasted details in an image. In order to reduce the impact of these higher-frequency elements, it is a good practice first to smooth the image before applying a derivative filter. You might think that this would be done in two steps, which are smoothing the image and then computing the derivative. However, a closer look at these operations reveals that it is possible to combine these two steps into one with a proper choice of the smoothing kernel. We learned previously that the convolution of an image with a filter can be expressed as a summation of terms. Interestingly, a well-known mathematical property is that the derivative of a summation of terms is equal to the summation of the terms' derivative. Consequently, instead of applying the derivative to the result of the smoothing, it is possible to derivate the kernel and then convolute it with the image. Since the Gaussian kernel is continuously derivable, it represents a particularly appropriate choice. This is what is done when you call the `cv::sobel` function with different kernel sizes. The function will compute a Gaussian derivative kernel with different σ values. As an example, if we select the `7x7` Sobel filter (that is `kernel_size=7`) in the *x*-direction, the following result is obtained:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If you compare this image with the one shown earlier, it can be seen that many fine details have been removed, giving them more emphasis on the more significant edges. Note that we now have a band-pass filter, the low frequencies having been removed by the Gaussian filter and the high frequencies having been removed by the Sobel filter.

See also

- The *Detecting edges using the Canny operator* recipe in `Chapter 7`, *Extracting Lines, Contours, and Components*, shows you how to obtain a binary edge map using two different threshold values.

We've successfully learned how to apply directional filters to detect edges. Now let's move on to the next recipe!

Computing the Laplacian of an image

The Laplacian is another high-pass linear filter that is based on the computation of the image derivatives. As will be explained, it computes second-order derivatives to measure the curvature of the image function.

How to do it...

The OpenCV function, `cv::Laplacian`, computes the Laplacian of an image. It is very similar to the `cv::Sobel` function. In fact, it uses the same basic function, `cv::getDerivKernels`, in order to obtain its kernel matrix. The only difference is that there are no derivative order parameters since these ones are, by definition, second-order derivatives. The following steps will help us in computing as follows:

1. For this operator, we will create a simple class that will encapsulate some useful operations related to the Laplacian. The basic methods are as follows:

```
class LaplacianZC {

  private:
    // laplacian
    cv::Mat laplace;
    // Aperture size of the laplacian kernel
    int aperture;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
public:

   LaplacianZC() : aperture(3) {}

   // Set the aperture size of the kernel
   void setAperture(int a) {
      aperture= a;
   }

   // Compute the floating point Laplacian
   cv::Mat computeLaplacian(const cv::Mat& image) {

      // Compute Laplacian
      cv::Laplacian(image,laplace,CV_32F,aperture);
      return laplace;
   }
```

2. The computation of the Laplacian is done here on a floating-point image. To get an image of the result, we perform a rescaling, as shown in the previous recipe. This rescaling is based on the Laplacian maximum absolute value, where value `0` is assigned a gray level of `128`. A method of our class allows the following image representation to be obtained:

```
// Get the Laplacian result in 8-bit image
// zero corresponds to gray level 128
// if no scale is provided, then the max value will be
// scaled to intensity 255
// You must call computeLaplacian before calling this
cv::Mat getLaplacianImage(double scale=-1.0) {
   if (scale<0) {
      double lapmin, lapmax;
    // get min and max laplacian values
      cv::minMaxLoc(laplace,&lapmin,&lapmax);
      // scale the laplacian to 127
      scale= 127/ std::max(-lapmin,lapmax);
   }

   // produce gray-level image
   cv::Mat laplaceImage;
   laplace.convertTo(laplaceImage,CV_8U,scale,128);
   return laplaceImage;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Using this class, the Laplacian image computed from a 7 x 7 kernel is obtained as follows:

```
// Compute Laplacian using LaplacianZC class
LaplacianZC laplacian;
laplacian.setAperture(7); // 7x7 laplacian
cv::Mat flap= laplacian.computeLaplacian(image);
laplace= laplacian.getLaplacianImage();
```

The resulting image is as shown in this screenshot:

Let's see how the code works.

How it works...

Formally, the Laplacian of a 2-D function is defined as the sum of its second derivatives:

$$laplacian(I) = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

In its simplest form, it can be approximated by the following 3 x 3 kernel:

0	1	0
1	-4	1
0	1	0

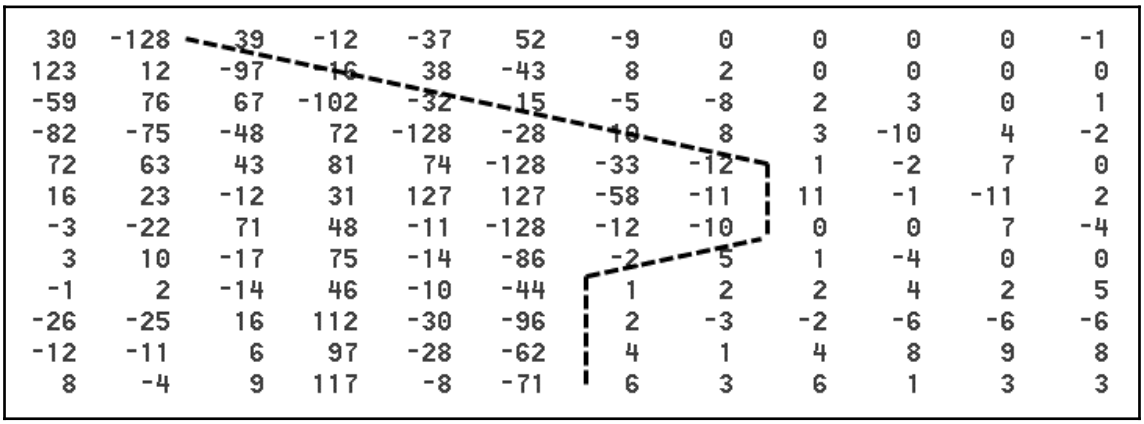
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

As with the Sobel operator, it is also possible to compute the Laplacian using larger kernels, and since this operator is even more sensitive to image noise, it is desirable to do so (unless computational efficiency is a concern). Since these larger kernels are computed using the second derivatives of the Gaussian function, the corresponding operator is often called the **Laplacian of Gaussian** (**LoG**). Note that the kernel values of a Laplacian always sum up to `0`. This guarantees that the Laplacian will be zero in areas of constant intensities. Indeed, since the Laplacian measures the curvature of the image function, it should be equal to `0` in flat areas.

At first glance, the effect of the Laplacian might be difficult to interpret. From the definition of the kernel, it is clear that any isolated pixel value (that is, a value that's very different than its neighbors) will be amplified by the operator. This is a consequence of the operator's high sensitivity to noise. However, it is more interesting to look at the Laplacian values around an image edge. The presence of an edge in an image is the result of a rapid transition between areas of different gray-level intensities. Following the evolution of the image function along an edge (for example, caused by a transition from dark to bright), you can observe that the gray-level ascension necessarily implies a gradual transition from a positive curvature (when the intensity values start to rise) to a negative curvature (when the intensity is about to reach its high plateau). Consequently, a transition between a positive and a negative Laplacian value (or reciprocally) constitutes a good indicator of the presence of an edge. Another way to express this fact is to say that edges will be located at the **zero-crossings** of the Laplacian function. We will illustrate this idea by looking at the values of a Laplacian in a small window of our test image. We select one that corresponds to an edge created by the bottom part of the roof of one of the castle's towers. A white box has been drawn in the following screenshot to show you the exact location of this ROI:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, looking at the Laplacian values (`7x7` kernel) inside this window, we have the following figure:



If, as illustrated, you carefully follow the zero-crossings of the Laplacian (located between pixels of different signs), you obtain a curve that corresponds to the edge that is visible in the image window. In the preceding figure, we drew dotted lines along the zero-crossings that corresponded to the edge of the tower that is visible in the selected image window. This implies that, in principle, you can even detect the image edges at sub-pixel accuracy.

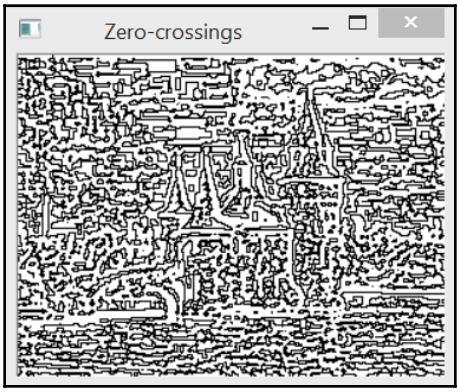
Following the zero-crossing curves in a Laplacian image is a delicate task. However, a simplified algorithm can be used to detect the approximate zero-crossing locations. This one proceeds by first thresholding the Laplacian at `0` such that it obtains a partition between the positive and negative values. The contours between these two partitions then correspond to our zero-crossings. Therefore, we use a morphological operation to extract these contours; that is, we subtract the dilated image from the Laplacian image (this is the Beucher gradient presented in the *Detecting edges and corners using morphological filters* recipe in `Chapter 5`, *Transforming Images with Morphological Operations*). This algorithm is implemented by the following method, which generates a binary image of zero-crossings:

```
// Get a binary image of the zero-crossings
// laplacian image should be CV_32F
cv::Mat getZeroCrossings(cv::Mat laplace) {

  // threshold at 0
  // negative values in black
  // positive values in white
  cv::Mat signImage;
  cv::threshold(laplace,signImage,0,255,cv::THRESH_BINARY);

  // convert the +/- image into CV_8U
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    cv::Mat binary;
    signImage.convertTo(binary,CV_8U);

    // dilate the binary image of +/- regions
    cv::Mat dilated;
    cv::dilate(binary,dilated,cv::Mat());
    // return the zero-crossing contours
    return dilated-binary;
}
```

The result is the following binary map:



As you can see, the zero-crossings of the Laplacian detect all edges. No distinction is made between strong edges and weaker edges. We also mentioned that the Laplacian is very sensitive to noise. Finally, some of these edges are due to compression artifacts. All these factors explain why so many edges are detected by the operator. In practice, the Laplacian is only used in conjunction with other operators to detect edges (for example, edges can be declared at zero-crossing locations of strong gradient magnitude). We will also learn in `Chapter 8`, *Detecting Interest Points*, that the Laplacian and other second-order operators are very useful in order to detect interest points at multiple scales.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

There's more...

The Laplacian is a high-pass filter. It is possible to approximate it by using a combination of low-pass filters. But, before that, let's quickly discuss image enhancement, which is a topic we have already discussed in `Chapter 2`, *Manipulating the Pixels*.

Enhancing the contrast of an image using the Laplacian

The contrast of an image can be enhanced by subtracting its Laplacian from it. This is what we did in the *Scanning an image with neighbor access* recipe in `Chapter 2`, *Manipulating the Pixels*, where we introduced the kernel:

0	-1	0
-1	5	-1
0	-1	0

This is equal to 1 minus the Laplacian kernel (that is, the original image minus its Laplacian).

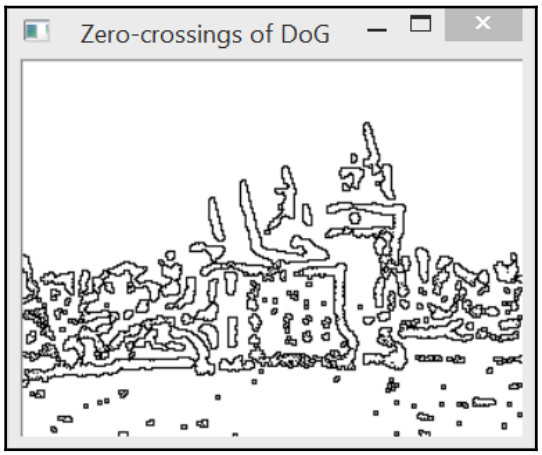
Difference of Gaussians

The Gaussian filter presented in the first recipe of this chapter extracts the low frequencies from an image. We learned that the range of frequencies that are filtered by a Gaussian filter depends on the parameter σ, which controls the width of the filter. Now, if we subtract the two images that result from the filtering of an image by two Gaussian filters of different bandwidths, then the resulting image will be composed of those higher frequencies that one filter has preserved, and not the other. This operation is called the **Difference of Gaussians** (**DoG**) and is computed as follows:

```
cv::GaussianBlur(image,gauss20,cv::Size(),2.0);
cv::GaussianBlur(image,gauss22,cv::Size(),2.2);

// compute a difference of Gaussians
cv::subtract(gauss22, gauss20, dog, cv::Mat(), CV_32F);

  // Compute the zero-crossings of DoG
zeros= laplacian.getZeroCrossings(dog);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In addition, we can also compute the zero-crossings of the DoG operator and then we obtain the following screenshot:



In fact, it can be demonstrated that with the proper choice of σ values, DoG operators can constitute a good approximation of LoG filters. Also, if you compute a series of a difference of Gaussians from consecutive pair values in an increasing sequence of σ values, you obtain a scale-space representation of the image. This multiscale representation is useful, for example, for scale-invariant image feature detection, as will be explained in `Chapter 8`, *Detecting Interest Points*.

See also

- The *Detecting scale-invariant features* recipe in `Chapter 8`, *Detecting Interest Points*, uses the Laplacian and DoG for the detection of scale-invariant features.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

7
Extracting Lines, Contours, and Components

In order to perform a content-based analysis of an image, it is necessary to extract meaningful features from the collection of pixels that constitute the image. Contours, lines, blobs, and so on, are fundamental image primitives that can be used to describe the elements contained in an image. This chapter will teach you how to extract some of these important image features.

In this chapter, we will cover the following recipes:

- Detecting image contours with the Canny operator
- Detecting lines in images with the Hough transform
- Fitting a line to a set of points
- Extracting the components' contours
- Computing components' shape descriptors

Detecting image contours with the Canny operator

Edges carry important visual information since they delineate the image elements. For this reason, they can be used, for example, in object recognition. However, simple binary edge maps suffer from two main drawbacks. First, the edges that are detected are unnecessarily thick; this makes the object's limit more difficult to identify. Second, and more importantly, it is often impossible to find a threshold that is sufficiently low in order to detect all important edges of an image and is, at the same time, sufficiently high in order to not include too many insignificant edges. This is a trade-off problem that the Canny algorithm tries to solve.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

The Canny algorithm is implemented in OpenCV by the `cv::Canny` function. As will be explained, this algorithm requires the specification of two thresholds:

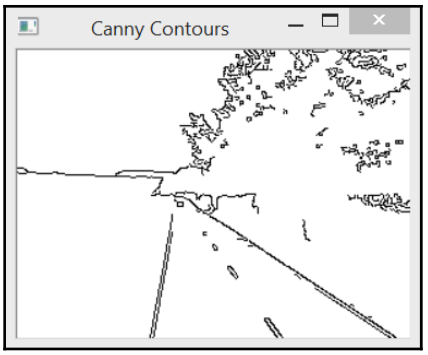
1. To apply a Canny algorithm to a previously loaded image, we have to create a new `cv::Mat` structure to store the result and call the `cv::Canny` function as follows:

```
// Apply Canny algorithm
cv::Mat contours;
cv::Canny(image,    // gray-level image
          contours, // output contours
          125,      // low threshold
          350);     // high threshold
```

   Take a look at the following screenshot:

2. When the algorithm is applied to the preceding screenshot, the result is as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

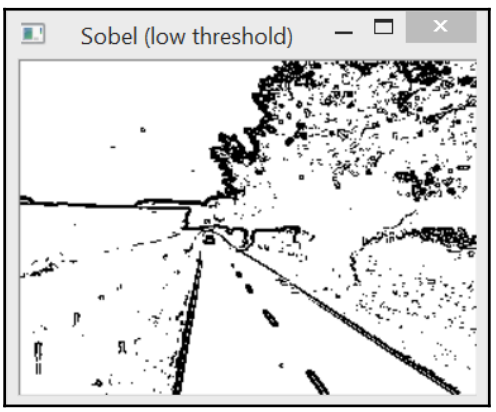


Note that, in order to obtain an image as shown in the preceding screenshot, we had to invert the black-and-white values since the normal result represents contours by nonzero pixels. The displayed image, then, is simply 255-contours.

How it works...

The Canny operator is generally based on the Sobel operator (presented in `Chapter 6`, *Filtering the Images),* although other gradient operators can also be used. The key idea here is to use two different thresholds in order to determine which point should belong to a contour – a low or a high threshold.

The low threshold should be chosen in a way that it includes all edge pixels that are considered to belong to a significant image contour. For example, by using the low-threshold value specified in the example of the preceding section and applying it on the result of a **Sobel** operator, the following edge map will be obtained:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

As we can see, the edges that delineate the road are very well defined. However, because a permissive threshold was used, more edges than what is ideally needed are also detected. The role of the second threshold, then, is to define the edges that belong to all important contours. It should exclude all edges considered as outliers. For example, the Sobel edge map that corresponds to the high-threshold used in our example is as follows:

We now have an image that contains broken edges, but the ones that are visible certainly belong to the significant contours of the scene.

The Canny algorithm combines these two edge maps in order to produce an *optimal* map of contours. It operates by keeping only the edge points of the low-threshold edge map for which a continuous path of edges exists, linking that edge point to an edge that belongs to the high-threshold edge map.

Consequently, all edge points of the high-threshold map are kept while all isolated chains of edge points in the low-threshold map are removed. The solution that is obtained constitutes a good compromise, allowing good quality contours to be obtained as long as appropriate threshold values are specified. This strategy, based on the use of two thresholds to obtain a binary map, is called **hysteresis thresholding** and can be used in any context where a binary map needs to be obtained from a thresholding operation. However, this is done at the cost of higher computational complexity.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In addition, the Canny algorithm uses an extra strategy to improve the quality of the edge map. Prior to the application of the hysteresis thresholding, all edge points for which the gradient magnitude is not a maximum in the gradient direction are removed. Recall that the gradient orientation is always perpendicular to the edge. Therefore, the local maximum of the gradient in this direction corresponds to the point of the maximum strength of the contour. This explains why thin edges are obtained in the Canny contour maps.

See also

- In `Chapter 6`, *Filtering the Images*, we learned how it is possible to detect the edges of an image. In particular, we showed you that by applying a threshold on the gradient magnitude, a binary map of the main edges of an image can be obtained.
- The classic article by J. Canny, *A Computational Approach to Edge Detection, IEEE Transactions on Pattern Analysis and Image Understanding, Vol. 18, Issue 6*, 1986.

We've successfully learned how to detect image contours with the Canny operator. Now, let's move on to the next recipe!

Detecting lines in images with the Hough transform

In our human-made world, planar and linear structures are in abundance. As a result, straight lines are frequently visible in images. These are meaningful features that play an important role in object recognition and image understanding. The **Hough transform** is a classic algorithm that is often used to detect these particular features in images. It was initially developed to detect lines in images and, as we will see, it can also be extended to detect other simple image structures.

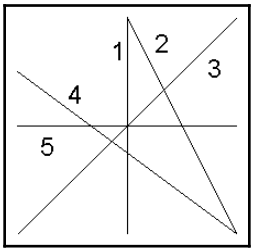
Getting ready

With the Hough transform, lines are represented using the following equation:

$$\rho = xcos(\theta) + ysin(\theta)$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The ϱ parameter is the distance between the line and the image origin (the upper-left corner), and θ is the angle of the perpendicular to the line. Under this representation, the lines visible in an image have a θ angle between 0 and π radians, while the ϱ radius can have a maximum value that equals the length of the image diagonal. Consider, for example, the following set of lines:



A vertical line (such as line **1)** has a θ angle value equal to zero, while a horizontal line (for example, line **5**) has its θ value equal to $\pi/2$. Therefore, line **3** has an angle θ equal to $\pi/4$, and line **4** is at 0.7π approximately. In order to be able to represent all possible lines with θ in the $[0, \pi]$ interval, the radius value can be made negative. This is the case of line **2**, which has a θ value equal to 0.8π with a negative value for ϱ.

How to do it...

OpenCV offers two implementations of the Hough transform for line detection. The basic version is `cv::HoughLines`. Its input is a binary map that contains a set of points (represented by nonzero pixels), some of which are aligned to form lines. Usually, this is an edge map obtained, for example, from the Canny operator. The output of the `cv::HoughLines` function is a vector of the `cv::Vec2f` elements, each of them being a pair of floating point values, which represents the parameters of a detected line, (ϱ, θ). The following is an example of using this function:

1. First, apply the Canny operator to obtain the image contours:

```
// Apply Canny algorithm
cv::Mat contours;
cv::Canny(image,contours,125,350);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Then, detect the lines using the Hough transform. Parameters three and four correspond to the step size for the line search. In our example, the function will search for lines of all possible radii by a step of `1` and all possible angles by a step of $\pi/180$. The role of the last parameter will be explained in the next section. With this particular choice of parameter values, 15 lines are detected on the road image of the preceding recipe:

```
// Hough tranform for line detection
   std::vector<cv::Vec2f> lines;
   cv::HoughLines(test,lines,
        1,PI/180,  // step size
        60);       // minimum number of votes
```

3. In order to visualize the result of the detection, it is interesting to draw these lines on the original image. However, it is important to note that this algorithm detects lines in an image and not line segments since the endpoints of each line are not given. Consequently, we will draw lines that traverse the entire image. To do this, for a vertically oriented line, we calculate its intersection with the horizontal limits of the image (that is, first and last rows) and draw a line between these two points. We proceed similarly with horizontally oriented lines, but by using the first and last columns. Lines are drawn using the `cv::line` function. Note that this function works well even with point coordinates outside the image limits. Therefore, there is no need to check whether the computed intersection points fall within the image. Lines are then drawn by iterating over the line vector, as follows:

```
std::vector<cv::Vec2f>::const_iterator it= lines.begin();
while (it!=lines.end()) {

   float rho= (*it)[0];   // first element is distance rho
   float theta= (*it)[1]; // second element is angle theta
   if (theta < PI/4.
        || theta > 3.*PI/4.) { // ~vertical line
      // point of intersection of the line with first row
      cv::Point pt1(rho/cos(theta),0);
      // point of intersection of the line with last row
      cv::Point pt2((rho-result.rows*sin(theta))/
                              cos(theta),result.rows);
      // draw a white line
      cv::line( image, pt1, pt2, cv::Scalar(255), 1);

   } else { // ~horizontal line

      // point of intersection of the
      // line with first column
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            cv::Point pt1(0,rho/sin(theta));
            // point of intersection of the line with last column
            cv::Point pt2(result.cols,
                    (rho-result.cols*cos(theta))/sin(theta));
            // draw a white line
            cv::line(image, pt1, pt2, cv::Scalar(255), 1);
        }

        ++it;
    }
```

We will then obtain the following result:

As we can see, the Hough transform simply looks for an alignment of edge pixels across the image. This can potentially create some false detections due to incidental pixel alignments, or multiple detections when several lines pass through the same alignment of pixels.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

To overcome some of these problems, and to allow line segments to be detected (that is, with endpoints), a variant of the transform has been proposed. This is the **Probabilistic Hough Transform**, and it is implemented in OpenCV as the `cv::HoughLinesP` function:

1. We use it here to create our `LineFinder` class that encapsulates the function parameters:

```
class LineFinder {

  private:

     // original image
     cv::Mat img;

     // vector containing the endpoints
     // of the detected lines
     std::vector<cv::Vec4i> lines;

     // accumulator resolution parameters
     double deltaRho;
     double deltaTheta;

     // minimum number of votes that a line
     // must receive before being considered
     int minVote;

     // min length for a line
     double minLength;

     // max allowed gap along the line
     double maxGap;

  public:

     // Default accumulator resolution is 1 pixel by 1 degree
     // no gap, no mimimum length
     LineFinder() : deltaRho(1), deltaTheta(PI/180),
                    minVote(10), minLength(0.), maxGap(0.) {}
```

2. Take a look at the corresponding setter methods:

```
// Set the resolution of the accumulator
void setAccResolution(double dRho, double dTheta) {

   deltaRho= dRho;
   deltaTheta= dTheta;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
// Set the minimum number of votes
void setMinVote(int minv) {

   minVote= minv;
}

// Set line length and gap
void setLineLengthAndGap(double length, double gap) {

   minLength= length;
   maxGap= gap;
}
```

3. With the preceding method, the method that performs Hough line segment detection is as follows:

```
// Apply Probabilistic Hough Transform
std::vector<cv::Vec4i> findLines(cv::Mat& binary) {

   lines.clear();
   cv::HoughLinesP(binary,lines,
                   deltaRho, deltaTheta, minVote,
                   minLength, maxGap);

   return lines;
}
```

4. This method returns a vector of `cv::Vec4i`, each containing the start and end point coordinates of each detected segment. The detected lines can then be drawn on an image with the following method:

```
// Draw the detected lines on an image
void drawDetectedLines(cv::Mat &image,
           cv::Scalar color=cv::Scalar(255,255,255)) {
   // Draw the lines
   std::vector<cv::Vec4i>::const_iterator it2=
                                        lines.begin();
   while (it2!=lines.end()) {
      cv::Point pt1((*it2)[0],(*it2)[1]);
      cv::Point pt2((*it2)[2],(*it2)[3]);

      cv::line( image, pt1, pt2, color);
      ++it2;
   }
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Now, using the same input image, lines can be detected with the following sequence:

```
// Create LineFinder instance
LineFinder finder;

// Set probabilistic Hough parameters
finder.setLineLengthAndGap(100,20);
finder.setMinVote(60);

// Detect lines and draw them
std::vector<cv::Vec4i> lines= finder.findLines(contours);
finder.drawDetectedLines(image);
cv::namedWindow("Detected Lines with HoughP");
cv::imshow("Detected Lines with HoughP",image);
```

The preceding code gives the following result:

Let's see how this works.

How it works...

The objective of the Hough transform is to find all lines in a binary image that passes through a sufficient number of points. It proceeds by considering each individual pixel point in the input binary map and identifying all possible lines that pass through it. When the same line passes through many points, it means that this line is significant enough to be considered.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Hough transform uses a two-dimensional accumulator in order to count how many times a given line is identified. The size of this accumulator is defined by the specified step sizes (as mentioned in the preceding section) of the (ϱ, θ) parameters of the adopted line representation. To illustrate the functioning of the transform, let's create a `180` by `200` matrix (corresponding to a step size of $\pi/180$ for θ and 1 for ϱ):

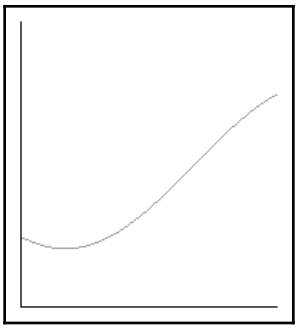
```
// Create a Hough accumulator
// here a uchar image; in practice should be ints
cv::Mat acc(200,180,CV_8U,cv::Scalar(0));
```

This accumulator is a mapping of different (ϱ, θ) values. Therefore, each entry of this matrix corresponds to one particular line. Now, if we consider one point, let's say one at coordinate `(50, 30)`, then it is possible to identify all lines that pass through this point by looping over all possible θ angles (with a step size of $\pi/180$) and computing the corresponding (rounded) ϱ value:

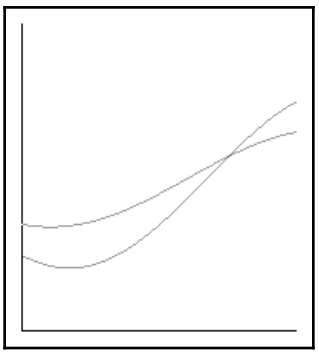
```
// Choose a point
int x=50, y=30;

// loop over all angles
for (int i=0; i<180; i++) {

   double theta= i*PI/180.;

   // find corresponding rho value
   double rho= x*std::cos(theta)+y*std::sin(theta);
   // j corresponds to rho from -100 to 100
   int j= static_cast<int>(rho+100.5);

   std::cout << i << "," << j << std::endl;

   // increment accumulator
   acc.at<uchar>(j,i)++;
}
```

The entries of the accumulator corresponding to the computed (ϱ, θ) pairs are then incremented, signifying that all of these lines pass through one point of the image (or, to say it another way, each point votes for a set of possible candidate lines). If we display the accumulator as an image (inverted and multiplied by `100` to make the count of `1` visible), we obtain the following:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



The preceding curve represents the set of all lines that pass through the considered point. Now, if we repeat the same exercise with, let's say, point `(30, 10)`, we now have the following accumulator:

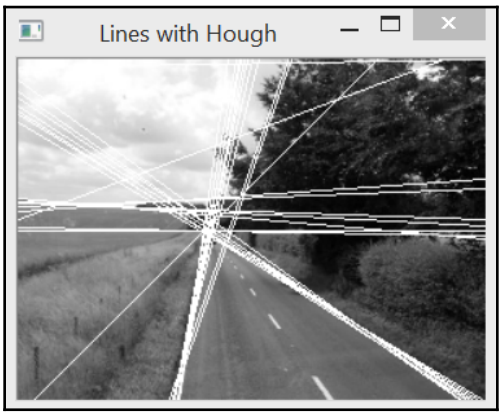


As we can see, the two resulting curves intersect at one point: the point that corresponds to the line that passes by these two points. The corresponding entry of the accumulator receives two votes, indicating that two points pass through this line. If the same process is repeated for all points of a binary map, then points aligned along a given line will increase a common entry of the accumulator many times. In the end, you just need to identify the local maxima in this accumulator that receives a significant number of votes in order to detect the lines (that is, point alignments) in the image. The last parameter specified in the `cv::HoughLines` function corresponds to the minimum number of votes that a line must receive to be considered as detected. For example, we lower this value at `50`, that is, the following:

```
cv::HoughLines(test,lines,1,PI/180,50);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

As a result of the previous code, more lines will be accepted for the example of the preceding section, as shown in the following screenshot:



The Probabilistic Hough Transform adds a few modifications to the basic algorithm. First, instead of systematically scanning the image row-by-row, points are chosen in random order in the binary map. Whenever an entry of the accumulator reaches the specified minimum value, the image is scanned along the corresponding line and all points that pass through it are removed (even if they have not voted yet). This scanning also determines the length of the segments that will be accepted. For this, the algorithm defines two additional parameters. One is the minimum length for a segment to be accepted, and the other is the maximum pixel gap that is permitted to form a continuous segment. This additional step increases the complexity of the algorithm, but this is partly compensated by the fact that fewer points will be involved in the voting process as some of them are eliminated by the line-scanning process.

There's more...

The Hough transform can also be used to detect other geometrical entities. In fact, any entity that can be represented by a parametric equation is a good candidate for the Hough transform.

Detecting circles

In the case of circles, the corresponding parametric equation is as follows:

$$r^2 = (x - x_0^2) + (y - y_0^2)$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This equation includes three parameters (the circle radius and center coordinates), which means that a three-dimensional accumulator would be required. However, it is generally found that the Hough transform becomes less reliable as the dimensionality of its accumulator increases. Indeed, in this case, a large number of entries of the accumulator will be incremented for each point, and, as a consequence, the accurate localization of local peaks becomes more difficult. Different strategies have been proposed in order to overcome this problem. The strategy used in the OpenCV implementation of the Hough circle detection uses two passes. During the first pass, a two-dimensional accumulator is used to find the candidate's circles locations. Since the gradient of points on the circumference of a circle should point in the direction of the radius, for each point, only the entries in the accumulator along the gradient direction are incremented (based on the predefined minimum and maximum radius values). Once a possible circle center is detected (that is, has received a predefined number of votes), a 1D histogram of the possible radius is built during the second pass. The peak value in this histogram corresponds to the radius of the detected circles.

The `cv::HoughCircles` function that implements the preceding strategy integrates both the Canny detection and the Hough transform:

```
cv::GaussianBlur(image,image,cv::Size(5,5),1.5);
std::vector<cv::Vec3f> circles;
cv::HoughCircles(image, circles, CV_HOUGH_GRADIENT,
   2,   // accumulator resolution (size of the image / 2)
   50,  // minimum distance between two circles
   200, // Canny high threshold
   100, // minimum number of votes
   25, 100); // min and max radius
```

Note that it is always recommended that you smooth the image before calling the `cv::HoughCircles` function in order to reduce the image noise that could cause several false circle detections. The result of the detection is given in a vector of `cv::Vec3f` instances. The first two values are the circle centers and the third is the radius.

The `CV_HOUGH_GRADIENT` argument was the only option available at the time of writing this book. It corresponds to the two-pass circle detection method. The fourth parameter defines the accumulator resolution. It is a divider factor; specifying a value of `2`, for example, makes the accumulator half the size of the image.

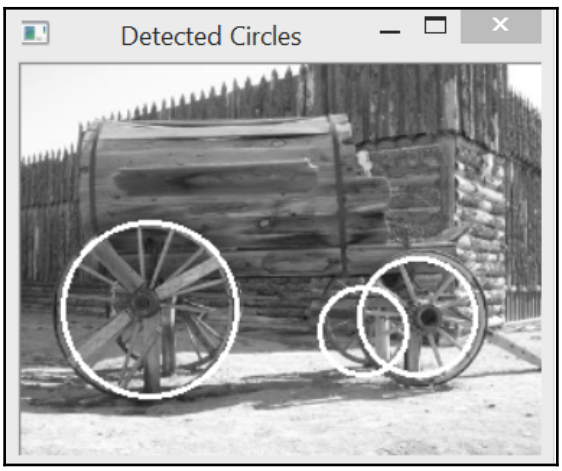
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The next parameter is the minimum distance in pixels between two detected circles. The other parameter corresponds to the high threshold of the Canny edge detector. The low-threshold value is always set at half this value. The seventh parameter is the minimum number of votes that a center location must receive during the first pass to be considered as a candidate circle for the second pass. Finally, the last two parameters are the minimum and maximum radius values for the circles to be detected. As we can see, the function includes many parameters that make it difficult to tune.

Once the vector of detected circles is obtained, these circles can be drawn on the image by iterating over the vector and calling the `cv::circle` drawing function with the found parameters:

```
std::vector<cv::Vec3f>::
       const_iterator itc= circles.begin();
while (itc!=circles.end()) {
  cv::circle(image,
     cv::Point((*itc)[0], (*itc)[1]), // circle centre
     (*itc)[2],       // circle radius
     cv::Scalar(255), // color
     2);              // thickness
  ++itc;
}
```

The following is the result obtained on a test image with the chosen arguments:



You can take a look at the articles given in the next section to know more about this recipe.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The article *Gradient-based Progressive Probabilistic Hough Transform, IEE Vision Image and Signal Processing, vol. 148 no 3, pp. 158-165,* by C. Galambos, J. Kittler, and J. Matas, 2002, is one of the numerous references on the Hough transform and describes the probabilistic algorithm implemented in OpenCV.
- The article *Comparative Study of Hough Transform Methods for Circle Finding, Image and Vision Computing, Vol. 8, No. 1, pp. 71-77, 1990* by H.K. Yuen, J. Princen, J. Illingworth, and J Kittler, describes different strategies for circle detection using the Hough transform.

We've successfully learned how to detect lines in images with the Hough transform. Now, let's move on to the next recipe!

Fitting a line to a set of points

In some applications, it could be important to not only detect lines in an image but also to obtain an accurate estimate of the line's position and orientation. This recipe will show you how to find the line that best fits a given set of points.

How to do it...

The first thing to do is to identify points in an image that seem to be aligned along a straight line. Let's use one of the lines we detected in the preceding recipe. The lines detected using `cv::HoughLinesP` are contained in `std::vector<cv::Vec4i>` called `lines`:

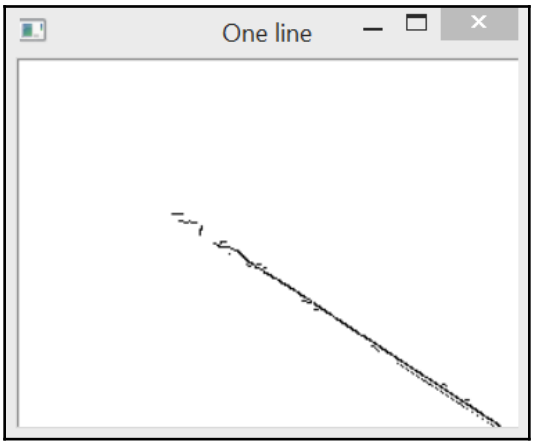
1. To extract the set of points that seem to belong to, let's say, the first of these lines, we can proceed as follows. We draw a white line on a black image and intersect it with the Canny image of contours used to detect our lines. This is simply achieved by the following statements:

```
int n=0; // we select line 0
// black image
cv::Mat oneline(contours.size(),CV_8U,cv::Scalar(0));
// white line
cv::line(oneline,
         cv::Point(lines[n][0],lines[n][1]),
         cv::Point(lines[n][2],lines[n][3]),
         cv::Scalar(255),
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        3); // line width
// contours AND white line
cv::bitwise_and(contours,oneline,oneline);
```

The result is an image that contains only the points that could be associated with the specified line. In order to introduce some tolerance, we draw a line of a certain thickness (here, `3`). All points inside the defined neighborhood are, therefore, accepted. The following is the image that is obtained (inverted for better viewing):



2. The coordinates of the points in this set can then be inserted in `std::vector` of the `cv::Point` functions (floating point coordinates, that is, `cv::Point2f`, can also be used) by the following double loop:

```
std::vector<cv::Point> points;

// Iterate over the pixels to obtain all point positions
for( int y = 0; y < oneline.rows; y++ ) {
   // row y
   uchar* rowPtr = oneline.ptr<uchar>(y);
   for( int x = 0; x < oneline.cols; x++ ) {
      // column x

      // if on a contour
      if (rowPtr[x]) {

         points.push_back(cv::Point(x,y));
      }
   }
 }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The best fit line is easily found by calling the `cv::fitLine` OpenCV function:

```
cv::Vec4f line;
cv::fitLine(points,line,
            CV_DIST_L2, // distance type
            0,          // not used with L2 distance
            0.01,0.01); // accuracy
```

   The preceding code gives us the parameters of the line equation in the form of a unit-directional vector (the first two values of `cv::Vec4f`) and the coordinates of one point on the line (the last two values of `cv::Vec4f`). For our example, these values are `(0.83, 0.55)` for the directional vector and `(366.1, 289.1)` for the point coordinates. The last two parameters specify the requested accuracy for the line parameters.

4. In general, the line equation will be used in the calculation of some properties (calibration is a good example where precise parametric representation is required). As an illustration, and to make sure we calculated the right line, let's draw the estimated line on the image. Here, we simply draw an arbitrary black segment that has a length of `100` pixels and a thickness of `3` pixels:

```
int x0= line[2];         // a point on the line
int y0= line[3];
int x1= x0+100*line[0]; // add a vector of length 100
int y1= y0+100*line[1]; // using the unit vector
// draw the line
cv::line(image,cv::Point(x0,y0),cv::Point(x1,y1),
         0,3); // color and thickness
```

   The result is then seen in the following screenshot:

We will now see how this works.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Fitting lines to a set of points is a classic problem in mathematics. The OpenCV implementation proceeds by minimizing the sum of the distances from each point to the line.

Several distance functions are proposed, and the fastest option is to use the Euclidean distance, which is specified by `CV_DIST_L2`. This choice corresponds to the standard least-squares line fitting. When outliers (that is, points that don't belong to the line) are included in the point set, other distance functions that give less influence to far points can be selected.

The minimization is based on the M-estimator technique that iteratively solves a weighted least-squares problem with weights that are inversely proportional to the distance from the line.

Using this function, it is also possible to fit a line to a 3D point set. The input is, in this case, a set of `cv::Point3i` or `cv::Point3f` functions and the output is a `std::Vec6f` function.

There's more...

The `cv::fitEllipse` function fits an ellipse to a set of 2D points. This returns a rotated rectangle (a `cv::RotatedRect` instance) inside which the ellipse is inscribed. In this case, you would write the following:

```
cv::RotatedRect rrect= cv::fitEllipse(cv::Mat(points));
cv::ellipse(image,rrect,cv::Scalar(0));
```

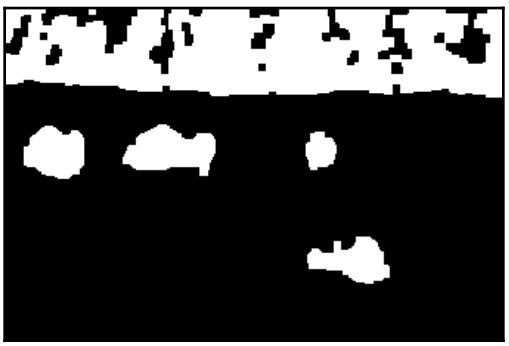
The `cv::ellipse` function is the one you would use to draw the computed ellipse.

We've successfully learned how to fit a line to a set of points. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Extracting the components' contours

Images generally contain representations of objects. One of the goals of image analysis is to identify and extract these objects. In object detection/recognition applications, the first step is often to produce a binary image that shows you where certain objects of interest could be located. No matter how this binary map is obtained (for example, from the histogram back projection as we did in `Chapter 4`, *Counting the Pixels with Histograms*, or from motion analysis as we will learn in `Chapter 11`, *Reconstructing 3D Scenes*), the next step is to extract the objects that are contained in this collection of one's and zero's. Consider, for example, the image of buffaloes in a binary form that we manipulated in `Chapter 5`, *Transforming Images with Morphological Operations*, as shown in the following image:



We obtained this image from a simple thresholding operation followed by the application of open and close morphological filters. This recipe will show you how to extract the objects of such images. More specifically, we will extract the **connected components**, that is, shapes made of a set of connected pixels in a binary image.

How to do it...

OpenCV offers a simple function that extracts the contours of the connected components of an image. This is the `cv::findContours` function:

1. To use this function we need a vector of points where we can store all output contours as follows:

```
// the vector that will contain the contours
std::vector<std::vector<cv::Point>> contours;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Now, we can use the `cv::findContours` to detect all the contours of the image and save them in the contours vector:

```
cv::findContours(image,
     contours, // a vector of contours
     CV_RETR_EXTERNAL, // retrieve the external contours
     CV_CHAIN_APPROX_NONE); // all pixels of each contours
```

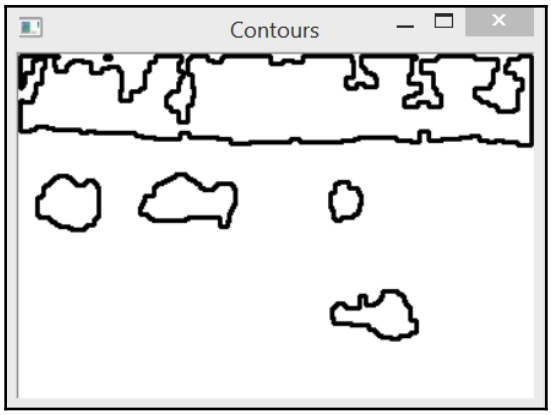
   The input is obviously the binary image. The output is a vector of contours, each contour is represented by a vector of `cv::Point` functions. This explains why the output parameter is defined as a `std::vector` function of the `std::vector` functions. In addition, two flags are specified. The first one indicates that only the external contours are required, that is, holes in an object will be ignored (the *There's more...* section will discuss the other options). The second flag is there to specify the format of the contour. With the current option, the vector will list all of the points in the contour. With the `CV_CHAIN_APPROX_SIMPLE` flag, only the endpoints would be included for horizontal, vertical, or diagonal contours. Other flags would give a more sophisticated chain approximation of the contours in order to obtain a more compact representation. With the preceding image, nine connected components are obtained as given by `contours.size()`.

3. Fortunately, there is a very convenient function that can draw the contours of those components on an image (here, a white image):

```
// draw black contours on a white image
cv::Mat result(image.size(),CV_8U,cv::Scalar(255));
cv::drawContours(result,contours,
   -1, // draw all contours
    0, // in black
    2);// with a thickness of 2
```

   If the third parameter of this function is a negative value, then all contours are drawn. Otherwise, it is possible to specify the index of the contour to be drawn. The result is shown in the following screenshot:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Let's see how the code works in the next section.

How it works...

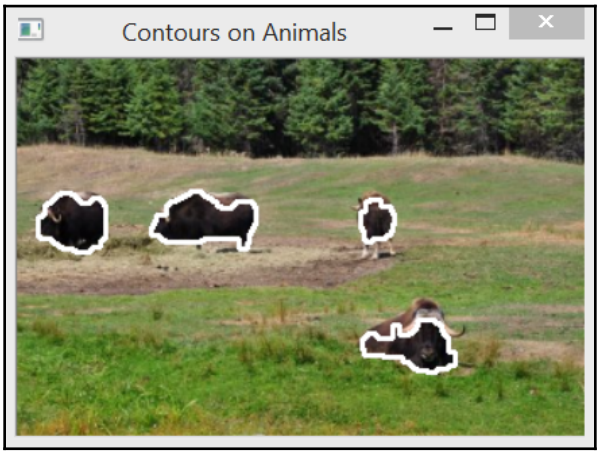
The contours are extracted by a simple algorithm that consists of systematically scanning the image until a component is hit. From this starting point on the component, its contour is followed, marking the pixels on its border. When the contour is completed, the scanning resumes at the last position until a new component is found.

The identified connected components can then be individually analyzed. For example, if some prior knowledge is available about the expected size of the objects of interest, it becomes possible to eliminate some of the components. Let's then use a minimum and a maximum value for the perimeter of the components. This is done by iterating over the vector of contours and eliminating the invalid components:

```
// Eliminate too short or too long contours
int cmin= 50;  // minimum contour length
int cmax= 1000; // maximum contour length
std::vector<std::vector<cv::Point>>::
           const_iterator itc= contours.begin();
// for all contours
while (itc!=contours.end()) {

   // verify contour size
   if (itc->size() < cmin || itc->size() > cmax)
      itc= contours.erase(itc);
   else
      ++itc;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that this loop could have been made more efficient since each erasing operation in an `std::vector` function is *O(N)*. However, considering the small size of this vector, the overall cost is not too high. This time, we draw the remaining contours on the original image and obtain the following result:



We were lucky enough to find a simple criterion that allowed us to identify all objects of interest in this image. In more complex situations, a more refined analysis of the components' properties is required. This is the object of the next recipe.

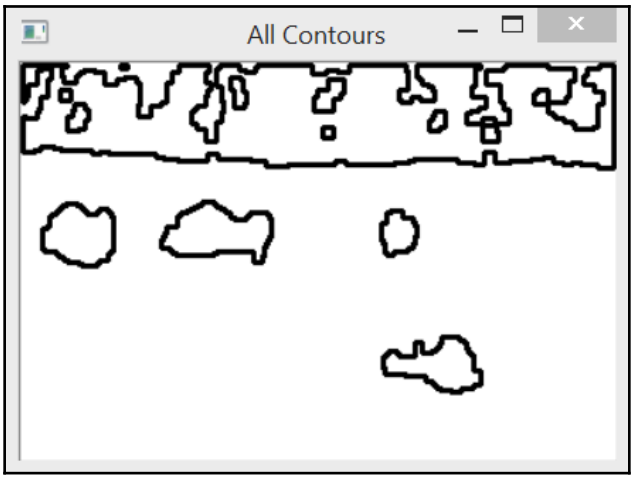
There's more...

With the `cv::findContours` function, it is also possible to include all closed contours in the binary map, including the ones formed by holes in the components. This is done by specifying another flag in the function call:

```
cv::findContours(image,
   contours, // a vector of contours
   CV_RETR_LIST, // retrieve all contours
   CV_CHAIN_APPROX_NONE); // all pixels of each contours
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

With this call, the following contours are obtained:



Notice the extra contours that were added in the background forest. It is also possible to have these contours organized into a hierarchy. The main component is the parent, holes in those are its children, and if there are components inside these holes, they become the children of the previous children, and so on. This hierarchy is obtained by using the `CV_RETR_TREE` flag, as follows:

```
std::vector<cv::Vec4i> hierarchy;
cv::findContours(image,
   contours, // a vector of contours
   hierarchy, // hierarchical representation
   CV_RETR_TREE, // retrieve all contours in tree format
   CV_CHAIN_APPROX_NONE); // all pixels of each contours
```

In this case, each contour has a corresponding hierarchy element at the same index, made of four integers. The first two integers give you the index of the next and the previous contours of the same level, and the next two integers give you the index of the first child and the parent of this contour. A negative index indicates the end of a contour list. The `CV_RETR_CCOMP` flag is similar but limits the hierarchy at two levels.

We've successfully learned how to extract the components' contours. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Computing components' shape descriptors

A connected component often corresponds to the image of some object in a pictured scene. To identify this object, or to compare it with other image elements, it can be useful to perform some measurements on the component in order to extract some of its characteristics. In this recipe, we will look at some of the shape descriptors available in OpenCV that can be used to describe the shape of a connected component.

How to do it...

Many OpenCV functions are available when it comes to shaping description. We will apply some of them on the components that we have extracted in the preceding recipe. In particular, we will use our vector of four contours corresponding to the four buffaloes we previously identified. In the following code snippets, we will compute a shape descriptor on the contours (`contours[0]` to `contours[3]`) and draw the result (with a thickness of `2`) over the image of the contours (with a thickness of `1`). This image is shown at the end of this section:

1. The first one is the bounding box, which is applied to the bottom-right component:

```
 // testing the bounding box
cv::Rect r0= cv::boundingRect(contours[0]);
// draw the rectangle
cv::rectangle(result,r0, 0, 2);
```

2. The minimum enclosing circle is similar. It is applied on the upper-right component:

```
 // testing the enclosing circle
 float radius;
 cv::Point2f center;
cv::minEnclosingCircle(contours[1],center,radius);
 // draw the circle
 cv::circle(result,cv::Point(center),
            static_cast<int>(radius),cv::Scalar(0),2);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The polygonal approximation of a component's contour is computed as follows (on the left-hand side component):

```
// testing the approximate polygon
std::vector<cv::Point> poly;
  cv::approxPolyDP(contours[2],poly,5,true);
// draw the polygon
cv::polylines(result, poly, true, 0, 2);
```

   Notice the polygon drawing function, `cv::polylines`. This operates similarly to the other drawing functions. The third Boolean parameter is used to indicate whether the contour is closed or not (if yes, the last point is linked to the first one).

4. The convex hull is another form of polygonal approximation (on the second component from the left):

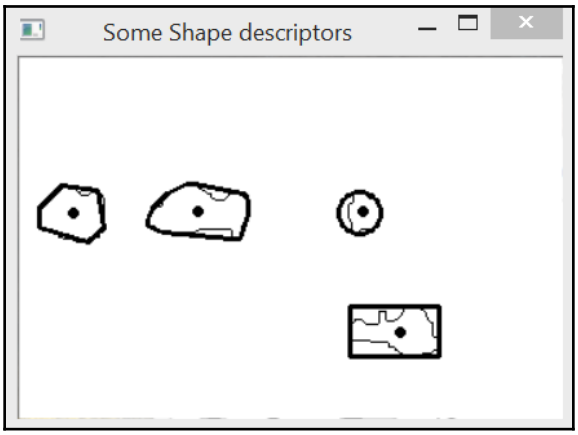
```
// testing the convex hull
std::vector<cv::Point> hull;
cv::convexHull(contours[3],hull);
// draw the polygon
cv::polylines(result, hull, true, 0, 2);
```

5. Finally, the computation of the moments is another powerful descriptor (the center of mass is drawn inside all components):

```
// testing the moments
// iterate over all contours
itc= contours.begin();
while (itc!=contours.end()) {

   // compute all moments
   cv::Moments mom= cv::moments(cv::Mat(*itc++));

   // draw mass center
   cv::circle(result,
      // position of mass center converted to integer
      cv::Point(mom.m10/mom.m00,mom.m01/mom.m00),
      2,cv::Scalar(0),2); // draw black dot
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The resulting image is as follows:



Let's take a look at how these steps work together.

How it works...

The *bounding box* of a component is probably the most compact way to represent and localize a component in an image. It is defined as the upright rectangle of a minimum size that completely contains the shape. Comparing the height and width of the box gives you an indication about the vertical or horizontal dimension of the object (for example, you could use a height-to-width ratio in order to distinguish the image of a car from one of a pedestrian). The *minimum enclosing circle* is generally used when only the approximate component size and location is required.

The *polygonal approximation* of a component is useful when you want to manipulate a more compact representation that resembles the component's shape. It is created by specifying an accuracy parameter, giving you the maximal acceptable distance between a shape and its simplified polygon. It is the fourth parameter in the `cv::approxPolyDP` function. The result is a vector of `cv::Point`, which corresponds to the vertices of the polygon. To draw this polygon, we need to iterate over the vector and link each point with the next one by drawing a line between them.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The **convex hull**, or convex envelope, of a shape, is the minimal convex polygon that encompasses a shape. It can be visualized as the shape that an elastic band would take if placed around the component. As it can be seen, the convex hull contour will deviate from the original one at the concave locations of the shape's contours. These locations are often designated as convexity defects and a special OpenCV function is available to identify them: the `cv::convexityDefects` function. It is called as follows:

```
std::vector<cv::Vec4i> defects;
cv::convexityDefects(contour, hull, defects);
```

The `contour` and `hull` arguments are, respectively, the original and the convex hull contours (both represented with `std::vector<cv::Point>` instances). The output is a vector of four-integer elements. The first two integers are the indices of the points on the contour, delimitating the defect; the third integer corresponds to the farthest point inside the concavity, and, finally, the last integer corresponds to the distance between this farthest point and the convex hull.

Moments are commonly used mathematical entities in the structural analysis of shapes. OpenCV has defined a data structure that encapsulates all computed moments of a shape. It is the object returned by the `cv::moments` function. Together, the moments represent a compact description of the shape of an object. They are commonly used, for example, in character recognition. We simply use this structure to obtain the mass center of each component that is computed from the first three spatial moments here.

There's more...

Other structural properties can be computed using the available OpenCV functions. The `cv::minAreaRect` function computes the minimum enclosed rotated rectangle (this one was used in `Chapter 5`, *Transforming Images with Morphological Operations,* in the *Extracting distinctive regions using MSER* recipe). The `cv::contourArea` function estimates the area of (the number of pixels inside) a contour. The `cv::pointPolygonTest` function determines whether a point is inside or outside a contour, and `cv::matchShapes` measure the resemblance between two contours. All these property measures can be advantageously combined in order to perform more advanced structural analysis.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Quadrilateral detection

The MSER features presented in `Chapter 5`, *Transforming Images with Morphological Operations*, constitute an efficient tool to extract shapes in an image. Considering the MSER result obtained in `Chapter 5`, *Transforming Images with Morphological Operations*, we will now build an algorithm to detect quadrilateral components in an image. In the case of the current image, this detection will allow us to identify the building's windows. A binary version of the MSER image is easily obtained as follows:

```
// create a binary version
components= components==255;
// open the image (white background)
cv::morphologyEx(components,components,
                 cv::MORPH_OPEN,cv::Mat(),
                 cv::Point(-1,-1),3);
```

In addition, we cleaned the image with some morphological filter. The image is then as follows:

The next step is to obtain the contours:

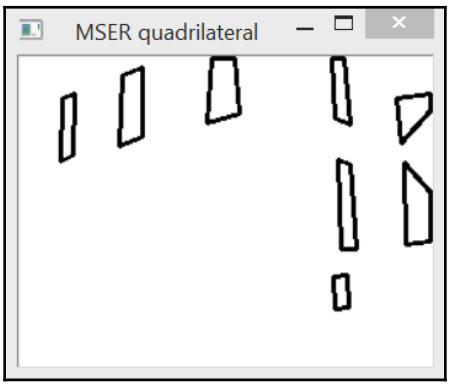
```
//invert image (background must be black)
cv::Mat componentsInv= 255-components;
// Get the contours of the connected components
cv::findContours(componentsInv,
  contours, // a vector of contours
  CV_RETR_EXTERNAL, // retrieve the external contours
  CV_CHAIN_APPROX_NONE);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, we go over all the contours and roughly approximate them with a polygon:

```
// white image
cv::Mat quadri(components.size(),CV_8U,255);

// for all contours
std::vector<std::vector<cv::Point>>::iterator
                            it= contours.begin();
while (it!= contours.end()) {
  poly.clear();
  // approximate contour by polygon
  cv::approxPolyDP(*it,poly,10,true);
  // do we have a quadrilateral?
  if (poly.size()==4) {
        // draw it
    cv::polylines(quadri, poly, true, 0, 2);
  }
  ++it;
}
```

The quadrilaterals are those polygons that have four edges. The detected ones are the following:



To detect rectangles, you can simply measure the angles between adjacent edges and reject the quadrilaterals that have angles that deviate too much from 90 degrees.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

8
Detecting Interest Points

In computer vision, the concept of interest points, also called **keypoints** or **feature points**, has been largely used to solve many problems in object recognition, image registration, visual tracking, 3D reconstruction, and more. Instead, of evaluating an image as a whole, it could be better to select points that can contain information that perform local analysis on the point to achieve results to apply local or globally. This approach works well as long as a sufficient number of such points are detected in the images of interest and as long as these points are distinct and stable features that can be accurately localized.

Because they are used to analyze image content, feature points should ideally be detected at the same scene or object location no matter from which viewpoint, scale, or orientation the image was taken. View invariance is a very desirable property in image analysis and has been the object of numerous studies. As we will see, different detectors have different invariance properties. This chapter focuses on the keypoint extraction process itself. The next two chapters will then show you how interest points can be put to work in different contexts such as image matching or image geometry estimation.

In this chapter, we will cover the following recipes:

- Detecting corners in an image
- Detecting features quickly
- Detecting scale-invariant features
- Detecting FAST features at multiple scales

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Detecting corners in an image

When searching for interesting feature points in images, corners come out as an interesting solution. They are features that can be easily localized in an image and they should abound in scenes of man-made objects (where they are produced by walls, doors, windows, tables, and so on). Corners are also interesting because they are two-dimensional features that can be accurately localized (even at sub-pixel accuracy), as they are at the junction of two or more edges. This is in contrast to points located on a uniform area or on the contour of an object and points that would be difficult to repeatedly localize precisely on other images of the same object. The Harris feature detector is a classical approach to detect corners in an image. We will explore this operator in this recipe.

How to do it...

The basic OpenCV function that is used to detect Harris corners is called `cv::cornerHarris` and is straightforward to use:

1. You call it on an input image, and the result is an image of floats saved on `cornerStrength` that gives you the corner strength at each pixel location, with the following code:

```
// Detect Harris Corners
cv::Mat cornerStrength;
cv::cornerHarris(image,      // input image
cornerStrength, // image of cornerness
                3,              // neighborhood size
                3,              // aperture size
                0.01);          // Harris parameter
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

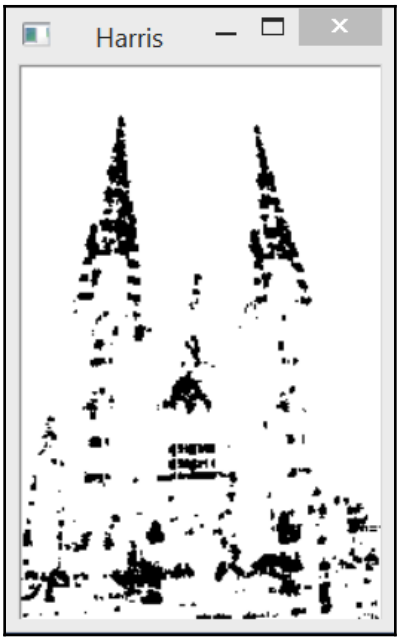
2. A threshold is then applied to this output image in order to obtain a set of detected corners. This is accomplished with the following code:

```
// threshold the corner strengths
cv::Mat harrisCorners;
double threshold= 0.0001;
cv::threshold(cornerStrength,harrisCorners,
threshold,255,cv::THRESH_BINARY);
```

Here is the original image:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The result is a binary map image shown in the following screenshot, which is inverted for better viewing (that is, we used `cv::THRESH_BINARY_INV` instead of `cv::THRESH_BINARY` to get the detected corners in black):



From the preceding function call, we observe that this interest point detector requires several parameters (these will be explained in the next section), which might make it difficult to tune. In addition, the corner map that is obtained contains many clusters of corner pixels, whereas we would like to detect well-localized points. Therefore, we will try to improve the corner-detection method by defining our own class to detect Harris corners.

The class encapsulates the Harris parameters with their default values and corresponding getter and setter methods (which are not shown here):

```
class HarrisDetector {

  private:

     // 32-bit float image of corner strength
     cv::Mat cornerStrength;
     // 32-bit float image of thresholded corners
     cv::Mat cornerTh;
     // image of local maxima (internal)
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    cv::Mat localMax;
    // size of neighborhood for derivatives smoothing
    int neighbourhood;
    // aperture for gradient computation
    int aperture;
    // Harris parameter
    double k;
    // maximum strength for threshold computation
    double maxStrength;
    // calculated threshold (internal)
    double threshold;
    // size of neighborhood for non-max suppression
    int nonMaxSize;
    // kernel for non-max suppression
    cv::Mat kernel;

  public:

    HarrisDetector() : neighbourhood(3), aperture(3),
                       k(0.01), maxStrength(0.0),
                       threshold(0.01), nonMaxSize(3) {
       // create kernel used in non-maxima suppression
       setLocalMaxWindowSize(nonMaxSize);
    }
```

3. To detect the Harris corners on an image, we proceed with two steps. First, the Harris values at each pixel are computed; this is shown in the following code:

```
// Compute Harris corners
void detect(const cv::Mat& image) {
   // Harris computation
   cv::cornerHarris(image,cornerStrength,
           neighbourhood,// neighborhood size
           aperture,     // aperture size
           k);           // Harris parameter
   // internal threshold computation
   cv::minMaxLoc(cornerStrength,
        0&maxStrength);

   // local maxima detection
   cv::Mat dilated;  // temporary image
   cv::dilate(cornerStrength,dilated,cv::Mat());
   cv::compare(cornerStrength,dilated,
               localMax,cv::CMP_EQ);
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Next, the feature points are obtained based on a specified threshold value. Since the range of possible values for Harris corners depends on the particular choices of its parameters, the threshold is specified as a quality level that is defined as a fraction of the maximal Harris value computed in the image. The following code shows how to do this:

```
// Get the corner map from the computed Harris values
cv::Mat getCornerMap(double qualityLevel) {

   cv::Mat cornerMap;

   // thresholding the corner strength
   threshold= qualityLevel*maxStrength;
   cv::threshold(cornerStrength,cornerTh,
                 threshold,255,cv::THRESH_BINARY);

   // convert to 8-bit image
   cornerTh.convertTo(cornerMap,CV_8U);
   // non-maxima suppression
   cv::bitwise_and(cornerMap,localMax,cornerMap);

   return cornerMap;
}
```

This method returns a binary corner map of the detected features. The fact that the detection of the Harris features has been split into two methods allows us to test the detection with a different threshold (until an appropriate number of feature points are obtained) without the need to repeat costly computations.

5. It is also possible to obtain the Harris features in the form of `std::vector` of `cv::Point`, as shown in the following code:

```
// Get the feature points from the computed Harris values
void getCorners(std::vector<cv::Point> &points,
                double qualityLevel) {

   // Get the corner map
   cv::Mat cornerMap= getCornerMap(qualityLevel);
   // Get the corners
   getCorners(points, cornerMap);
}

// Get the feature points from the computed corner map
void getCorners(std::vector<cv::Point> &points,
                const cv::Mat& cornerMap) {
   // Iterate over the pixels to obtain all features
   for( int y = 0; y < cornerMap.rows; y++ ) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            const uchar* rowPtr = cornerMap.ptr<uchar>(y);
            for( int x = 0; x < cornerMap.cols; x++ ) {

               // if it is a feature point
               if (rowPtr[x]) {

                  points.push_back(cv::Point(x,y));
               }
            }
         }
      }
```

This class also improves the detection of the Harris corners by adding a non-maximal suppression step, which will be explained in the next section.

6. The detected points can now be drawn on an image using the `cv::circle` function, as demonstrated by the following method:

```
// Draw circles at feature point locations on an image
void drawOnImage(cv::Mat &image,
   const std::vector<cv::Point> &points,
   cv::Scalar color= cv::Scalar(255,255,255),
   int radius=3, int thickness=1) {

   std::vector<cv::Point>::const_iterator it=
                                  points.begin();

   // for all corners
   while (it!=points.end()) {

      // draw a circle at each corner location
      cv::circle(image,*it,radius,color,thickness);
      ++it;
   }
}
```

7. Using the preceding class, the detection of the Harris points is accomplished as follows:

```
// Create Harris detector instance
HarrisDetector harris;
 // Compute Harris values
harris.detect(image);
 // Detect Harris corners
std::vector<cv::Point> pts;
harris.getCorners(pts,0.02);
// Draw Harris corners
harris.drawOnImage(image,pts);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This results in the following image:

Now, let's go behind the scenes to understand the code better.

How it works...

To define corners in images, Harris looks at the average change in directional intensity in a small window around a putative interest point. If we consider a displacement vector, `(u,v)`, the average intensity change is given by the following:

$$R = \sum (I(x+u, y+v) - I(x,y))^2$$

The summation is over a defined neighborhood around the considered pixel (the size of this neighborhood corresponds to the third parameter in the `cv::cornerHarris` function). This average intensity change can then be computed in all possible directions, which leads to the definition of a corner as a point for which the average change is high in more than one direction. From this definition, the Harris test is performed as follows. We first obtain the direction of the maximal average intensity change. Next, we check whether the average intensity change in the orthogonal direction is high as well. If it is the case, then we have a corner.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Mathematically, this condition can be tested by using an approximation of the preceding formula using the Taylor expansion:

$$R \approx \sum \left(I(x,y) + \frac{\delta I}{\delta x}u + \frac{\delta I}{\delta y}v - I(x,y) \right)^2 = \sum \left(\left(\frac{\delta I}{\delta x}u \right)^2 + \left(\frac{\delta I}{\delta y}v \right)^2 + 2\frac{\delta I}{\delta x}\frac{\delta I}{\delta y}uv \right)$$

This is then rewritten in the matrix form:

$$R \approx \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} \sum \left(\frac{\delta I}{\delta x} \right)^2 & \sum \frac{\delta I}{\delta x}\frac{\delta I}{\delta y} \\ \sum \frac{\delta I}{\delta x} - \frac{\delta I}{\delta y} & \sum \left(\frac{\delta I}{\delta y} \right)^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

This matrix is a covariance matrix that characterizes the rate of intensity change in all directions. This definition involves the image's first derivatives that are often computed using the Sobel operator. This is the case in the OpenCV implementation, which is the fourth parameter of the function that corresponds to the aperture used for the computation of the Sobel filters.

It can be shown that the two eigenvalues of the covariance matrix give you the maximal average intensity change and the average intensity change for the orthogonal direction. Then, if these two eigenvalues are low, we are in a relatively homogenous region. If one eigenvalue is high and the other is low, we must be on an edge.

Finally, if both eigenvalues are high, then we are at a corner location. Therefore, the condition for a point to be accepted as a corner is that it must have the smallest eigenvalue of the covariance matrix at a higher point than a given threshold.

The original definition of the Harris corner algorithm uses some properties of the eigendecomposition theory in order to avoid the cost of explicitly computing the eigenvalues. These properties are as follows:

- The product of the eigenvalues of a matrix is equal to its determinant.
- The sum of the eigenvalues of a matrix is equal to the sum of the diagonal of the matrix (also known as the **trace** of the matrix).

It then follows that we can verify whether the eigenvalues of a matrix are high by computing the following score:

$$Det(C) - kTrace^2(C)$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We can easily verify that this score will, indeed, be high only if both eigenvalues are high too. This is the score that is computed by the `cv::cornerHarris` function at each pixel location. The value of `k` is specified as the fifth parameter of the function. It could be difficult to determine what would be the best value for this parameter. However, in practice, it has been seen that value in the range of `0.05` and `0.5` generally gives good results.

To improve the result of the detection, the class described in the previous section adds an additional non-maximal suppression step. The goal here is to exclude Harris corners that are adjacent to others. Therefore, to be accepted, the Harris corner must not only have a score higher than the specified threshold, but it must also be a local maximum. This condition is tested by using a simple trick that consists of dilating the image of the Harris score in our `detect` method:

```
cv::dilate(cornerStrength,dilated,cv::Mat());
```

Since the dilation replaces each pixel value by the maximum in the defined neighborhood, the only points that will not be modified are the local maxima. This is what is verified by the following equality test:

```
cv::compare(cornerStrength,dilated,
            localMax,cv::CMP_EQ);
```

The `localMax` matrix will, therefore, be true (that is, nonzero) only at local maxima locations. We then use it in our `getCornerMap` method to suppress all non-maximal features (using the `cv::bitwise_and` function).

There's more...

Additional improvements can be made to the original Harris corner algorithm. This section describes another corner detector found in OpenCV, which expands the Harris detector to make its corners more uniformly distributed across the image. As we will see, this operator has an implementation for the feature detector in the OpenCV 2 common interface.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Good features to track

With the advent of floating-point processors, the mathematical simplification introduced to avoid the eigenvalue decomposition has become negligible, and, consequently, the detection of Harris corners can be made based on the explicitly computed eigenvalues. In principle, this modification should not significantly affect the result of the detection, but it avoids the use of the arbitrary `k` parameter. Note that two functions exist, allowing you to explicitly get the eigenvalues (and eigenvectors) of the Harris covariance matrix; these are `cv::cornerEigenValsAndVecs` and `cv::cornerMinEigenVal`.

A second modification addresses the problem of feature point clustering. Indeed, in spite of the introduction of the local maxima condition, interest points tend to be unevenly distributed across an image, showing concentrations at highly textured locations. A solution to this problem is to impose a minimum distance between two interest points. This can be achieved by the following algorithm.

Starting from the point with the strongest Harris score (that is, with the largest minimum eigenvalue), only accept interest points if they are located at, at least, a given distance from the already accepted points. This solution is implemented in OpenCV in the `cv::goodFeaturesToTrack` function, which is named as such because the features it detects can be used as a good starting set in visual tracking applications. This is called as follows:

```
    // Compute good features to track
    std::vector<cv::Point2f> corners;
    cv::goodFeaturesToTrack(image,  // input image
corners, // corner image
      500,     // maximum number of corners to be returned
      0.01,    // quality level
      10);     // minimum allowed distance between points
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In addition to the quality-level threshold value and the minimum-tolerated distance between interest points, the function also uses a maximum number of points that can be returned (this is possible since points are accepted in the order of strength). The preceding function call produces the following result:

This approach increases the complexity of the detection since it requires the interest points to be sorted by their Harris score, but it also clearly improves the distribution of the points across the image. Note that this function also includes an optional flag that requests that Harris corners are detected using the classical corner score definition (using the covariance matrix determinant and trace).

The feature detector's common interface

OpenCV 2 has introduced a common interface for its different interest point detectors. This interface allows for the easy testing of different interest point detectors within the same application.

The interface defines a `cv::Keypoint` class, which encapsulates the properties of each detected feature point. For the Harris corners, only the position of the keypoints and its response strength is relevant. The *Detecting scale-invariant features* recipe will discuss the other properties that can be associated with a keypoint.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `cv::FeatureDetector` abstract class imposes the existence of a `detect` operation with the following signatures:

```
void detect( const Mat& image, vector<KeyPoint>& keypoints,
             const Mat& mask=Mat() ) const;

void detect( const vector<Mat>& images,
             vector<vector<KeyPoint> >& keypoints,
             const vector<Mat>& masks=
                          vector<Mat>() ) const;
```

The second method allows interest points to be detected in a vector of images. The class also includes other methods that can read and write the detected points in a file.

The `cv::goodFeaturesToTrack` function has a wrapper class called `cv::GoodFeaturesToTrackDetector`, which inherits from the `cv::FeatureDetector` class. It can be used in a way that is similar to what we did with our Harris corners class, as follows:

```
 // vector of keypoints
 std::vector<cv::KeyPoint> keypoints;
 // Construction of the Good Feature to Track detector
cv::Ptr<cv::FeatureDetector> gftt=
   new cv::GoodFeaturesToTrackDetector(
  500,  // maximum number of corners to be returned
  0.01,  // quality level
  10);  // minimum allowed distance between points
// point detection using FeatureDetector method
gftt->detect(image,keypoints);
```

The result is the same as the one obtained previously since the same function is ultimately called by the wrapper. Note how we used the OpenCV 2 smart pointer class `(cv::Ptr)`, which, as explained in `Chapter 1`, *Playing with Images*, automatically releases the pointed object when the reference count drops to zero.

See also

- The classical article that describes the Harris operator by C. Harris and M.J. Stephens, *A combined corner and edge detector, Alvey Vision Conference, pp. 147-152,* 1988.
- The article by J. Shi and C. Tomasi, *Good Features to Track, Int. Conference on Computer Vision and Pattern Recognition, pp. 593-600, 1994*, introduces these features.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

- The article by K. Mikolajczyk and C. Schmid, *Scale & Affine Invariant Interest Point Detectors, International Journal of Computer Vision, Vol. 60, No. 1, pp. 63-86, 2004*, proposes a multi-scale and affine-invariant Harris operator.

We've successfully learned how to detect corners in an image. Now, let's move on to the next recipe!

Detecting features quickly

The Harris operator proposed a formal mathematical definition for corners (or, more generally, interest points) based on the rate of intensity changes in two perpendicular directions. Although this constitutes a sound definition, it requires the computation of the image derivatives, which is a costly operation, especially considering the fact that interest point detection is often just the first step in a more complex algorithm.

In this recipe, we present another feature point operator, called **FAST** (short for **Features from Accelerated Segment Test**). This one has been specifically designed to allow quick detection of interest points in an image; the decision to accept or not to accept a keypoint is based on only a few pixel comparisons.

How to do it...

Using the OpenCV 2 common interface for feature point detection makes the deployment of any feature point detectors easy. The detector presented in this recipe is the FAST detector. As the name suggests, it has been designed to be quick in order to compute the following. To detect keypoints using the FAST detector, we have to do the following:

1. Create a vector of keypoints to store the result:

```
// vector of keypoints
std::vector<cv::KeyPoint> keypoints;
```

2. Create a FAST detector with a threshold of 40 in our example:

```
// Construction of the Fast feature detector object
cv::Ptr<cv::FeatureDetector> fast=  new
cv::FastFeatureDetector(40);
```

3. Detect all the keypoints of an image loaded previously:

```
// feature point detection
fast->detect(image,keypoints);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that OpenCV also proposes a generic function to draw keypoints on an image:

```
cv::drawKeypoints(image,    // original image
     keypoints,               // vector of keypoints
     image,                  // the output image
     cv::Scalar(255,255,255), // keypoint color
     cv::DrawMatchesFlags::DRAW_OVER_OUTIMG); //drawing flag
```

By specifying the chosen drawing flag, the keypoints are drawn over the input image, producing the following output result:

An interesting option is to specify a negative value for the keypoint color. In this case, a different random color will be selected for each drawn circle.

Now, let's go behind the scenes to understand the code better.

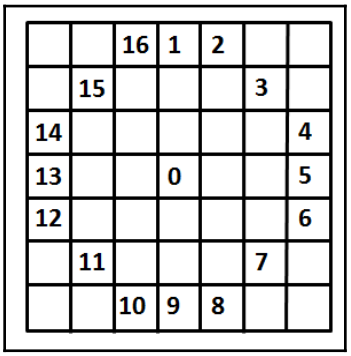
How it works...

As in the case with the Harris point, the FAST feature algorithm derives from the definition of what constitutes a *corner*. This time, this definition is based on the image intensity around a putative feature point. The decision to accept a keypoint is taken by examining a circle of pixels centered at a candidate point. If an arc of contiguous points of a length greater than 3/4 of the circle perimeter in which all pixels significantly differ from the intensity of the center point (all being either darker or brighter) is found, then a keypoint is declared.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This is a simple test that can be computed quickly. Moreover, in its original formulation, the algorithm uses an additional trick to further speed up the process. Indeed, if we first test four points separated by `90o` on the circle (for example, top, bottom, right, and left points), it can be easily shown that, in order to satisfy the condition expressed previously, at least three of these points must all be brighter or darker than the central pixel. If this is not the case, the point can be rejected immediately, without inspecting additional points on the circumference. This is a very effective test since, in practice, most of the image points will be rejected by this simple four-comparison test.

In principle, the radius of the circle of examined pixels could have been a parameter of the method. However, it has been found that in practice, a radius of **3** gives you both good results and high efficiency. There are, then, **16** pixels that need to be considered on the circumference of the circle, as follows:



The four points used for the pretest are the **1**, **5**, **9**, and **13** pixels and the required number of contiguous darker or brighter points is **12**. However, it has been observed that by reducing the length of the contiguous segment to **9**, better repeatability of the detected corners across images is obtained. This variant is often designated as the **FAST-9** corner detector, and this is the one that is used by OpenCV. Note that there exists a `cv::FASTX` function, which proposes another variant of the FAST detector.

To be considered as being significantly darker or brighter, the intensity of a point must differ from that of the central pixel by at least a given amount; this value corresponds to the threshold parameter specified in the function call. The larger this threshold is, the fewer corner points will be detected.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

As for Harris features, it is often better to perform non-maxima suppression on the corners that have been found. Therefore, a corner strength measure needs to be defined. Several alternatives to this can be considered, and the one that has been retained is the following. The strength of a corner is given by the sum of the absolute difference between the central pixel and the pixels on the identified contiguous arc. Note that the algorithm is also available through a direct function call. The code is as follows:

```
cv::FAST(image,     // input image
        keypoints, // output vector of keypoints
       40,        // threshold
       false);    // non-max suppression? (or not)
```

However, because of its flexibility, the use of the `cv::FeatureDetector` interface is recommended.

This algorithm results in very fast interest point detection and is, therefore, the feature of choice when speed is a concern. This is the case, for example, in real-time visual tracking or object-recognition applications where several points must be tracked or matched in a live video stream.

There's more...

To improve the detection of feature points, additional tools are offered by OpenCV. Indeed, a number of class adapters are available in order to better control the way the keypoints are extracted.

Adapted feature detection

If you wish to better control the number of detected points, a special subclass of the `cv::FeatureDetector` class, called `cv::DynamicAdaptedFeatureDetector`, is available. This allows you to specify the number of interest points that can be detected as an interval. In the case of the FAST feature detector, this is used as follows:

```
cv::DynamicAdaptedFeatureDetector fastD(
  new cv::FastAdjuster(40), // the feature detector
  150,   // min number of features
  200,   // max number of features
  50);   // max number of iterations
fastD.detect(image,keypoints); // detect points
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The interest points will then be iteratively detected. After each iteration, the number of detected points are checked and the detector threshold is adjusted accordingly in order to produce more or fewer points; this process is repeated until the number of detected points fits into the specified interval.

A maximum number of iterations is specified in order to avoid the method that makes you spend too much time on multiple detections. For this method to be implemented in a generic way, the used `cv::FeatureDetector` class must implement the `cv::AdjusterAdapter` interface. This class includes a `tooFew` method and a `tooMany` method, both of which modify the internal threshold of the detector in order to produce more or less keypoints. There is also a good predicate method that returns `true` if the detector threshold can still be adjusted.

Using a `cv::DynamicAdaptedFeatureDetector` class can be a good strategy to obtain an appropriate number of feature points; however, you must understand that there is a performance price that you will have to pay for this benefit. Moreover, there is no guarantee that you will indeed obtain the requested number of features within the specified number of iterations.

You probably noticed that we passed as an argument, which is the address of a dynamically allocated object, to specify the feature detector that will be used by the adapter class. You might wonder whether you have to release the allocated memory at some point in order to avoid memory leaks. The answer is no, and this is because the pointer is transferred to a `cv::Ptr<FeatureDetector>` parameter, which automatically releases the pointed object.

A second useful class adapter is the `cv::GridAdaptedFeatureDetector` class. As the name suggests, it allows you to define a grid over the image. Each cell of this grid is then constrained to contain a maximum number of elements. The idea here is to spread the set of detected keypoints over the image in a better manner. When detecting keypoints in an image, it is, indeed, common to see a concentration of interest points in a specific textured area.

This is the case, for example, for the two towers of the church image on which a very dense set of FAST points have been detected. This class adapter is used as follows:

```
cv::GridAdaptedFeatureDetector fastG(
  new cv::FastFeatureDetector(10), // the feature detector
  1200,  // max total number of keypoints
  5,     // number of rows in grid
  2);    // number of cols in grid
fastG.detect(image,keypoints);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The class adapter proceeds by detecting feature points on each individual cell using the provided `cv::FeatureDetector` object. A maximum total number of points is also specified. Only the strongest points in each cell are kept in order to not exceed the specified maximum.

The `cv::PyramidAdaptedFeatureDetector` adapter proceeds by applying the feature detector on an image pyramid. The results are combined in the output vector of keypoints. This is called as follows:

```
cv::PyramidAdaptedFeatureDetector fastP(
  new cv::FastFeatureDetector(60), // the feature detector
  3);    // number of levels in the pyramid
fastP.detect(image,keypoints);
```

The coordinates of each point are specified in the original image coordinates. In addition, the special `size` attribute of the `cv::Keypoint` class is set such that points detected at half the original resolution are attributed a size that is the double that of the detected points in the original image. There is a special flag in the `cv::drawKeypoints` function that will draw the keypoints with a radius that is equal to the keypoint's `size` attribute:

The next section provides additional information on this recipe.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The *Matching points through correlation* recipe in `Chapter 9`, *Describing and Matching Interest Points*, shows you how to use FAST to match two images using simple correlation.
- The article by E. Rosten and T. Drummond, *Machine learning for high-speed corner detection, in In European Conference on Computer Vision, pp. 430-443, 2006*, describes the FAST feature algorithm and its variants in detail.

We've successfully learned how to detect features quickly. Now, let's move on to the next recipe!

Detecting scale-invariant features

The view invariance of feature detection was presented as an important concept in the introduction of this chapter. While orientation invariance, which is the ability to detect the same points even if an image is rotated, has been relatively well handled by the simple feature point detectors that have been presented so far, changes for invariance to scale are more difficult to achieve. To address this problem, the concept of scale-invariant features has been introduced in computer vision.

The idea here is to not only have a consistent detection of keypoints no matter at which scale an object is pictured, but to also have a scale factor associated with each of the detected feature points. Ideally, for the same object point featured at two different scales on two different images, the ratio of the two computed scale factors should correspond to the ratio of their respective scales. In recent years, several scale-invariant features have been proposed, and this recipe presents one of them, the SURF features. **SURF** stands for **Speeded Up Robust Features**, and, as we will see, they are not only scale-invariant features, but they also offer the advantage of being computed very efficiently.

How to do it...

The SURF feature detector is implemented in OpenCV in the `cv::SURF` function:

1. It is also possible to use this through `cv::FeatureDetector` as follows, as previous key detectors:

```
// Construct the SURF feature detector object
cv::Ptr<cv::FeatureDetector> detector = new cv::SURF(
2000.); // threshold
```

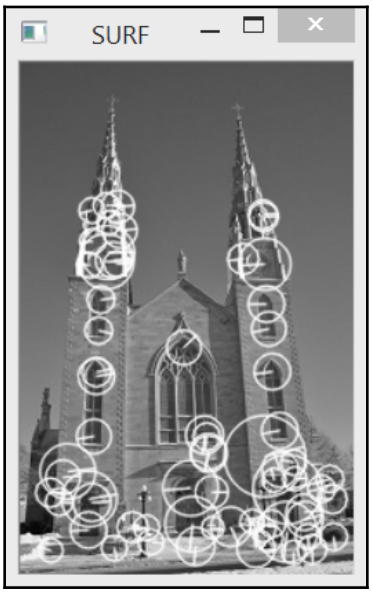

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
// Detect the SURF features
detector->detect(image,keypoints);
```

2. To draw these features, we again use the `cv::drawKeypoints` OpenCV function with the `DRAW_RICH_KEYPOINTS` flag so that we can visualize the associated scale factor:

```
// Draw the keypoints with scale and orientation information
cv::drawKeypoints(image,      // original image
   keypoints,                 // vector of keypoints
   featureImage,              // the resulting image
   cv::Scalar(255,255,255),   // color of the points
   cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS); //flag
```

The resulting image with the detected features is then as follows:



As explained in the previous recipe, the size of the keypoint circles resulting from the use of the `DRAW_RICH_KEYPOINTS` flag is proportional to the computed scale of each feature. The SURF algorithm also associates an orientation with each feature to make them invariant to rotations. This orientation is illustrated by a radial line inside each drawn circle.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If we take another picture of the same object, but at a different scale, the feature-detection result is as follows:

By carefully observing the detected keypoints on the two images, it can be seen that the change in the size of corresponding circles is often proportional to the change in scale. As an example, consider the bottom part of the upper-right window of the church. In both images, a SURF feature has been detected at that location, and the two corresponding circles (of different sizes) contain the same visual elements. Of course, this is not the case for all features, but as we will discover in `Chapter 9`, *Describing and Matching Interest Points*, the repeatability rate is sufficiently high to allow good matching between the two images.

Now, let's go behind the scenes to understand the code better.

How it works...

The derivatives of an image can be estimated using Gaussian filters (for more information, refer to `Chapter 6`, *Filtering the Images)*. These filters make use of a σ parameter, which defines the aperture (size) of the kernel. As we saw, this σ parameter corresponds to the variance of the Gaussian function used to construct the filter and it then implicitly defines a scale at which the derivative is evaluated. Indeed, a filter that has a larger σ value smooths out the finer details of the image. This is why we can say that it operates at a coarser scale.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

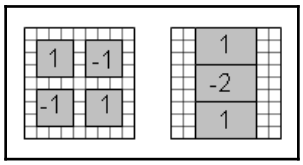
Now, if we compute, for instance, the Laplacian of a given image point using Gaussian filters at different scales, then different values are obtained. Looking at the evolution of the filter response for different scale factors, we obtain a curve that eventually reaches a maximum value at a σ value. If we extract this maximum value for two images of the same object taken at two different scales, the ratio of these two σ maxima will correspond to the ratio of the scales at which the images were taken. This important observation is at the core of the scale-invariant feature extraction process. That is, scale-invariant features should be detected as the local maxima in both the spatial space (in the image) and the scale space (as obtained from the derivative filters applied at different scales).

SURF implements this idea by proceeding as follows. First, to detect the features, the Hessian matrix is computed at each pixel. This matrix measures the local curvature of a function and is defined as follows:

$$H(x,y) = \begin{bmatrix} \frac{\delta^2 I}{\delta x^2} & \frac{\delta^2 I}{\delta x \delta y} \\ \frac{\delta^2 I}{\delta x \delta y} & \frac{\delta^2 I}{\delta y^2} \end{bmatrix}$$

The determinant of this matrix gives you the strength of this curvature. The idea, therefore, is to define corners as image points with high local curvature (that is, high variation in more than one direction). Since it is composed of second-order derivatives, this matrix can be computed using Laplacian of Gaussian kernels of a different scale, such as σ. This Hessian then becomes a function of three variables, which are `H(x,y,σ)`. Therefore, a scale-invariant feature is declared when the determinant of this Hessian reaches a local maximum in both spatial and scale space (that is, `3x3x3` non-maxima suppression needs to be performed). Note that, in order to be considered as a valid point, this determinant must have a minimum value as specified by the first parameter in the constructor of the `cv::SURF` class.

However, the calculation of all of these derivatives at different scales is computationally costly. The objective of the SURF algorithm is to make this process as efficient as possible. This is achieved by using approximated Gaussian kernels that involve only a few integer additions. These have the following structure:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The kernel on the left-hand side is used to estimate the mixed second derivatives, while the one on the right-hand side estimates the second derivative in the vertical direction. A rotated version of this second kernel estimates the second derivative in the horizontal direction. The smallest kernels have a size of `9x9` pixels, corresponding to $\sigma \approx 1.2$. To obtain a scale-space representation, kernels of increasing size are successively applied. The exact number of filters that are applied can be specified by additional parameters of the SURF class. By default, 12 different sizes of kernels are used (going up to size `99x99`). Note that the fact that integral images are used guarantees that the sum inside each lobe of each filter can be computed by using only three additions independent of the size of the filter.

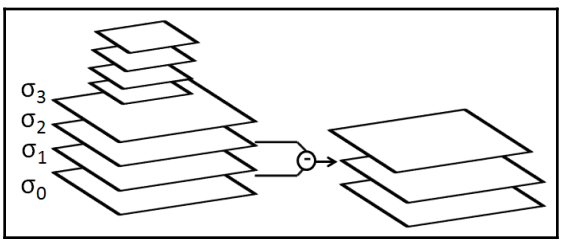
Once the local maxima are identified, the precise position of each detected interest point is obtained through interpolation in both scale and image space. The result is then a set of feature points that are localized at sub-pixel accuracy and with which a scale value is associated.

There's more...

The SURF algorithm has been developed as an efficient variant of another well-known scale-invariant feature detector called **SIFT** (short for **Scale-Invariant Feature Transform**).

The SIFT feature-detection algorithm

SIFT also detects features as local maxima in the image and scale space but uses the Laplacian filter response instead of the Hessian determinant. This Laplacian is computed at different scales (that is, increasing values of σ) using the difference of Gaussian filters, as explained in `Chapter 6`, *Filtering the Images*. To improve efficiency, each time the value of σ is doubled, the size of the image is reduced by two. Each pyramid level corresponds to an **octave**, and each scale is a **layer**. There are typically three layers per octave. The following diagram illustrates a pyramid of two octaves in which the four Gaussian-filtered images of the first octave produce three DoG layers:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

OpenCV has a class that detects these features, and it is called in a way that is similar to the SURF one:

```
// Construct the SIFT feature detector object
detector = new cv::SIFT();
// Detect the SIFT features
detector->detect(image,keypoints);
```

Here, we use all the default arguments to construct the detector, but you can specify the number of desired SIFT points (the strongest ones are kept), the number of layers per octave, and the initial value for σ. The result is similar to the one obtained with SURF:

However, since the computation of the feature point is based on floating-point kernels, SIFT is generally considered to be more accurate in terms of feature localization in the space and scale. For the same reason, it is also more computationally expensive, although this relative efficiency depends on each particular implementation.

As a final remark, you might have noticed that the SURF and SIFT classes have been placed in a `nonfree` package of the OpenCV distribution. This is because these algorithms have been patented, and, as such, their use in commercial applications might be subject to licensing agreements.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The *Computing the Laplacian of an image* recipe in `Chapter 6`, *Filtering the Images*, gives you more details on the Laplacian-of-Gaussian operator and the use of the difference of Gaussians.
- The *Describing local intensity patterns* recipe in `Chapter 9`, *Describing and Matching Interest Points*, explains how these scale-invariant features can be described for robust image matching.
- The article *SURF: Speeded Up Robust Features* by H. Bay, A. Ess, T. Tuytelaars and L. Van Gool in *Computer Vision and Image Understanding, Vol. 110, No. 3, pp. 346-359,* 2008, describes the SURF feature algorithm.
- The pioneering work by D. Lowe, *Distinctive Image Features from Scale Invariant Features* in *International Journal of Computer Vision, Vol. 60, No. 2, 2004, pp. 91-110,* describes the SIFT algorithm.

We've successfully learned how to detect scale-invariant features. Now, let's move on to the next recipe!

Detecting FAST features at multiple scales

FAST has been introduced as a quick way to detect keypoints in an image. With SURF and SIFT, the emphasis was on designing scale-invariant features. More recently, new interest point detectors have been introduced with the objective of achieving both fast detection and invariance-to-scale changes. This recipe presents the **Binary Robust Invariant Scalable Keypoints** (**BRISK**) detector. It is based on the FAST feature detector that we described in a previous recipe of this chapter. Another detector, called **ORB** (**Oriented FAST and Rotated BRIEF**), will also be discussed at the end of this recipe. These two feature point detectors constitute an excellent solution when fast and reliable image matching is required. They are especially efficient when they are used in conjunction with their associated binary descriptors, as will be discussed in `Chapter 9`, *Describing and Matching Interest Points*.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Following what we did in the previous recipes, the detection of keypoints with BRISK uses the `cv::FeatureDetector` abstract class. Let's have a look at the following steps:

1. We first create an instance of the detector, and then the `detect` method is called on an image:

```
// Construct the BRISK feature detector object
detector = new cv::BRISK();
// Detect the BRISK features
detector->detect(image,keypoints);
```

The image result shows you the keypoints that are detected at multiple scales:

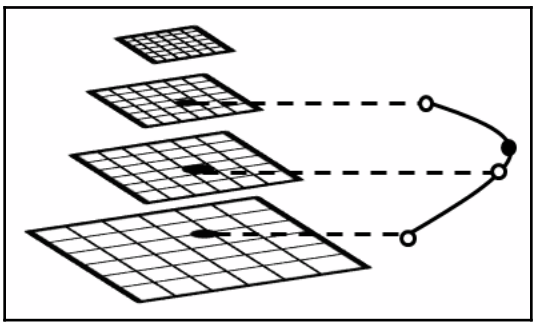
Now, let's go behind the scenes to understand the code better.

How it works...

BRISK is not only a feature point detector, but the method also includes a procedure that describes the neighborhood of each detected keypoint. This second aspect will be the subject of `Chapter 9`, *Describing and Matching interest Points*. We describe here how the quick detection of keypoints at multiple scales is performed using BRISK.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In order to detect interest points at different scales, the method first builds an image pyramid through two down-sampling processes. The first process starts from the original image size and downscales it by half at each layer (or octave). Secondly, in-between layers are created by down-sampling the original image by a factor of 1.5 and, from these in-between layers, the original image generates additional layers through successive half-sampling:



The FAST feature detector is then applied on all the images of this pyramid. Keypoint extraction is based on a criterion that is similar to the one used by SIFT. First, an acceptable interest point must be a local maximum when comparing its strength with one of its eight spatial neighbors. If this is the case, the point is then compared with the scores of the neighboring points in the layers above and below; if its score is higher in scale as well, then it is accepted as an interest point. A key aspect of BRISK resides in the fact that the different layers of the pyramid have different resolutions.

The method requires interpolation in both scale and space in order to locate each keypoint precisely. This interpolation is based on the FAST keypoint scores. In space, the interpolation is performed on 3 x 3 neighborhood. In scale, it is computed by fitting a 1D parabola along the scale axis through the current point and its two neighboring local keypoints in the layers above and below; this keypoint localization in scale is illustrated in the preceding diagram. As a result, even if the FAST keypoint detection is performed at discrete image scales, the resulting detected scales associated with each keypoint are a continuous value.

The `cv::BRISK` class proposes two optional parameters to control the detection of the keypoints. The first parameter is a threshold value that accepts FAST keypoints, and the second parameter is the number of octaves that will be generated in the image pyramid:

```
// Construct another BRISK feature detector object
detector = new cv::BRISK(
  20,  // threshold for FAST points to be accepted
  5);  // number of octaves
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

There's more...

BRISK is not the only multiscale fast detector that is proposed in OpenCV. The ORB feature detector can also perform efficient keypoint detection.

The ORB feature-detection algorithm

ORB stands for **Oriented FAST and Rotated BRIEF**. The first part of this acronym refers to the keypoint detection part, while the second part refers to the descriptor that is proposed by ORB. Here, we focus on the detection method; the descriptor will be presented in `Chapter 9`, *Describing and Matching Interest Points*.

As with BRISK, ORB first creates an image pyramid. This one is made of a number of layers in which each layer is a down-sampled version of the previous one by a certain scale factor (typically, `8` scales and `1.2` scale factor reduction; these are parameters in the `cv::ORB` function). The strongest `N` keypoints are then accepted where the keypoint score is defined by the Harris *cornerness* measure that was defined in the first recipe of this chapter (authors of this method found the Harris score to be a more reliable measure).

An original aspect of the ORB detector resides in the fact that orientation is associated with each detected interest point. As we will see in `Chapter 9`, *Describing and Matching Interest Points*, this information will be useful to align the descriptors of keypoints detected in different images. In the *Computing components' shape descriptors* recipe of `Chapter 7`, *Extracting Lines, Contours, and Components*, we introduced the concept of image moments, and, in particular, we showed you how the centroid of a component can be computed from its first three moments. ORB proposes that we use the orientation of the centroid of a circular neighborhood around the keypoint. Since FAST keypoints, by definition, always have a decentered centroid, the angle of the line that joins the central point and the centroid will always be well defined.

The ORB features are detected as follows:

```
// Construct the ORB feature detector object
detector = new cv::ORB(200, // total number of keypoints
                       1.2, // scale factor between layers
                   8);  // number of layers in pyramid
// Detect the ORB features
detector->detect(image,keypoints);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This call produces the following result:

As we can see, since the keypoints are independently detected on each pyramid layer, the detector tends to repeatedly detect the same feature point at different scales.

See also

- The *Describing keypoints with binary features* recipe in `Chapter 9`, *Describing and Matching Interest Points*, explains how simple binary descriptors can be used for the efficient robust matching of these features.
- The article *BRISK: Binary Robust Invariant Scalable Keypoint* by S. Leutenegger, M. Chli, and R. Y. Siegwart in *IEEE International Conference on Computer Vision, pp. 2448--2555*, 2011, describes the BRISK feature algorithm.
- The article *ORB: an efficient alternative to SIFT or SURF by* E. Rublee, V. Rabaud, K. Konolige, and G. Bradski *in IEEE International Conference on Computer Vision, pp.2564-2571,* 2011, describes the ORB feature algorithm.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

9
Describing and Matching Interest Points

In `Chapter 8`, *Detecting Interest Points,* we learned how to detect special points in an image with the objective of subsequently performing a local image analysis. These keypoints are chosen to be distinctive enough so that if a keypoint is detected on the image of an object, then the same point is expected to be detected in other images depicting the same object. We also described some more sophisticated interest point detectors that can assign a representative scale factor and/or an orientation to a keypoint. As we will see in this recipe, this additional information can be useful to normalize scene representations with respect to viewpoint variations.

In order to perform image analysis based on interest points, we now need to build rich representations that uniquely describe each of these keypoints. This chapter looks at the different approaches that have been proposed to extract **descriptors** from interest points. These ones are generally 1D or 2D vectors of binary, integer, or floating-point numbers that describe a keypoint and its neighborhood. A good descriptor should be distinctive enough to uniquely represent each keypoint of an image; it should be robust enough to have the same points represented similarly in spite of possible illumination changes or viewpoint variations. Ideally, it should also be compact to facilitate processing operations.

One of the most common operations accomplished with keypoints is image matching. The objective of performing this task could be, for example, to relate two images of the same scene or to detect the occurrence of a target object in an image. Here, we will study some basic matching strategies, a subject that will be further discussed in `Chapter 10`, *Estimating Projective Relations in Images*.

In this chapter, we will cover the following recipes:

- Matching local templates
- Describing local intensity patterns
- Describing keypoints with binary features

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Matching local templates

Feature point *matching* is the operation by which you can put in correspondence points from one image to points from another image (or points from an image set). Image points should match when they correspond to the image of the same scene element (or object point) in the real world.

A single pixel is certainly not sufficient to make a decision on the similarity of two keypoints. This is why an image *patch* around each keypoint must be considered during the matching process. If two patches correspond to the same scene element, then you might expect their pixels to exhibit similar values. A direct pixel-by-pixel comparison of pixel patches is the solution presented in this recipe. This is probably the simplest approach to feature point matching, but, as we will see, not the most reliable one. Nevertheless, in several situations, it can give good results.

How to do it...

Most often, patches are defined as squares of odd sizes centered at the keypoint position. The similarity between two square patches can then be measured by comparing the corresponding pixel intensity values inside the patches. A simple **Sum of Squared Differences** (**SSD**) is a popular solution. The feature matching strategy then works as follows. First, the keypoints are detected in each image. Here, let's use the FAST detector:

1. Define two keypoints vectors to store the points detected in for each image, as shown in the following code:

```
// Define keypoints vector
std::vector<cv::KeyPoint> keypoints1;
std::vector<cv::KeyPoint> keypoints2;
```

2. Create the FAST detector with a thresold of `80`:

```
cv::Ptr<cv::FeatureDetector> ptrDetector; // generic detector
ptrDetector= cv::FastFeatureDetector::create(80); // we select the
FAST detector
```

3. Detect all points in each image applying the FAST detector created:

```
ptrDetector->detect(image1,keypoints1);
ptrDetector->detect(image2,keypoints2);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. We then define a rectangle of the size `11x11` that will be used to define patches around each keypoint:

```
// Define a square neighborhood
const int nsize(11); // size of the neighborhood
cv::Rect neighborhood(0, 0, nsize, nsize); // 11x11
cv::Mat patch1;
cv::Mat patch2;
```

5. The keypoints in one image are compared with all the keypoints in the other image. For each keypoint of the first image, the most similar patch in the second image is identified. This process is implemented using two nested loops, as shown in the following code:

```
// For all keypoints in first image
// find best match in second image
cv::Mat result;
std::vector<cv::DMatch> matches;

//for all keypoints in image 1
for (int i=0; i<keypoints1.size(); i++) {
  // define image patch
  neighborhood.x = keypoints1[i].pt.x-nsize/2;
  neighborhood.y = keypoints1[i].pt.y-nsize/2;

  // if neighborhood of points outside image,
   // then continue with next point
  if (neighborhood.x<0 || neighborhood.y<0 ||
      neighborhood.x+nsize >= image1.cols ||
          neighborhood.y+nsize >= image1.rows)
      continue;

    //patch in image 1
    patch1 = image1(neighborhood);

    // reset best correlation value;
    cv::DMatch bestMatch;

    //for all keypoints in image 2
      for (int j=0; j<keypoints2.size(); j++) {

      // define image patch
      neighborhood.x = keypoints2[j].pt.x-nsize/2;
      neighborhood.y = keypoints2[j].pt.y-nsize/2;

      // if neighborhood of points outside image,
         // then continue with next point
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        if (neighborhood.x<0 || neighborhood.y<0 ||
          neighborhood.x + nsize >= image2.cols ||
               neighborhood.y + nsize >= image2.rows)
            continue;

        // patch in image 2
        patch2 = image2(neighborhood);

        // match the two patches
           cv::matchTemplate(patch1,patch2,result,
                             cv::TM_SQDIFF_NORMED);

        // check if it is a best match
        if (result.at<float>(0,0) < bestMatch.distance) {

          bestMatch.distance= result.at<float>(0,0);
          bestMatch.queryIdx= i;
          bestMatch.trainIdx= j;
        }
      }

      // add the best match
      matches.push_back(bestMatch);
  }
```

Note the use of the `cv::matchTemplate` function, which we will describe in the next section. It would suffice to say that this function computes the patch similarity score. When a potential match is identified, this one is represented through the use of a `cv::DMatch` object. This object stores the index of the two matching keypoints as well as the similarity score.

The more similar two image patches are, the higher the probability is that these patches correspond to the same scene point. This is why it is a good idea to sort the resulting of match point by their similarity scores.

6. To get the similarity scores, we can use the `nth_element` partial sorting algorithm from the `std` library like in the following code:

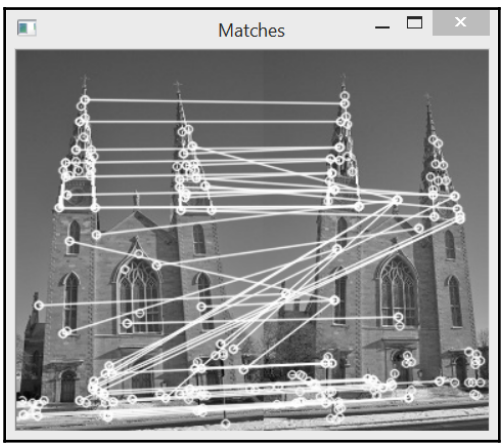
```
// extract the 25 best matches
std::nth_element(matches.begin(),
                 matches.begin()+25,matches.end());
matches.erase(matches.begin()+25,matches.end());
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. We can then simply retain the matches that pass a given similarity threshold. Here, we chose to keep only the `N` best matching points (we use `N=25` to facilitate the visualization of the matching results). Interestingly, there is an OpenCV function that can display the matching results by concatenating the two images and joining each corresponding point by a line. The function is used as follows:

```
// Draw the matching results
cv::Mat matchImage;
cv::drawMatches(image1,keypoints1, // first image
                image2,keypoints2, // second image
                matches,     // vector of matches
cv::Scalar(255,255,255),  // color of lines
cv::Scalar(255,255,255)); // color of points
```

Here is the match result:



Now, let's go behind the scenes to understand the code better.

How it works...

The results obtained are certainly not perfect, but a visual inspection of the matched image shows a number of successful matches. It can also be observed that the repetitive structures of the building cause some confusion. Also, since we tried to match all the points in the left image with the ones in the right image, we obtained cases where a point in the right image was matched with multiple points in the left image. This is an asymmetrical matching situation that can be corrected by, for example, keeping only the match with the best score for each point in the right image.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

To compare the image patches from each image, we used a simple criterion, that is, a pixel-per-pixel sum of the squared difference specified using the `TM_SQDIFF` flag. If we compare the `(x,y)` point of image `I1` with a putative match at `(x',y')` in image `I2`, then the similarity measure is given as follows:

$$\sum_{i,j}(I_1(x+i,y+j)-I_2(x'+i,y'+j))^2$$

Here, the sum of the `(i,j)` point provides the offset to cover the square template centered at each point. Since the difference between adjacent pixels in similar patches should be small, the best-matching patches should be the ones with the smallest sum. This is what is done in the main loop of the matching function; that is, for each keypoint in one image, we identify the keypoint in the other image that gives the lowest sum of the squared difference. We can also reject matches for which this sum is over a certain threshold value. In our case, we simply sort them from the most similar to the least similar ones.

In our example, the matching was done with square patches of size `11x11`. A larger neighborhood creates more distinctive patches, but it also make them more sensitive to local scene variations.

Comparing two image windows from a simple sum of square differences will work relatively well as long as the two images show the scene from similar points of view and similar viewing conditions. Indeed, a simple lighting change will increase or decrease all the pixel intensities of a patch, resulting in a large square difference. To make matching more invariant to lighting changes, other formulae that could be used to measure the similarity between two image windows exist. OpenCV offers a number of these. A very useful formula is the normalized sum of square differences (the `TM_SQDIFF_NORMED` flag):

$$\frac{\sum_{i,j}(I_1(x+i,y+j)-I_2(x'+i,y'+j))^2}{\sqrt{\sum_{i,j}I_1(x+i,y+j)^2}\sqrt{\sum_{i,j}I_2(x'+i,y'+j)^2}}$$

Other similarity measures are based on the concept of correlation, defined in the signal processing theory as follows (with the `TM_CCORR` flag):

$$\sum_{i,j}I_1(x+i,y+j)I_2(x'+i,y'+j)$$

This value will be maximal when two patches are similar.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The identified matches are stored in a vector of the `cv::DMatch` instances. Essentially, the `cv::DMatch` data structure contains the first index that refers to an element in the first vector of keypoints and the second index that refers to the matching feature in the second vector of keypoints. It also contains a real value that represents the distance between the two matched descriptors. This distance value is used in the definition of `operator<` when comparing two `cv::DMatch` instances.

When we drew the matches in the previous section, we wanted to limit the number of lines to make the results more readable. Therefore, we only displayed the `25` matches that had the lowest distance. To do this, we used the `std::nth_element` function, which positions the N^{th} element in sorted order at the Nth position, with all the smaller elements placed before this element. Once this is done, the vector is simply purged of its remaining elements.

There's more...

The `cv::matchTemplate` function is at the heart of our feature matching method. We used it here in a very specific way, which was to compare two image patches. However, this function has been designed to be used in a more generic way.

Template matching

A common task in image analysis is to detect the occurrence of a specific pattern or object in an image. This can be done by defining a small image of the object, a template, and searching for a similar occurrence in a given image. In general, the search is limited to a region of interest inside which we think the object can be found. The template is then slid over this region, and a similarity measure is computed at each pixel location. This is the operation performed by the `cv::matchTemplate` function. The input is a template image of a small size and an image over which the search is performed. The result is a `cv::Mat` function of floating point values that correspond to the similarity score at each pixel location. If the template is of the size `MxN` and the image is of the size `WxH`, then the resulting matrix will have a size of `W-N+1xH-N+1`. In general, you will be interested in the location of the highest similarity; so, the typical template matching code will look as follows (assuming that the target variable is our template):

```
// define search region
cv::Mat roi(image2,
  // here top half of the image
  cv::Rect(0,0,image2.cols,image2.rows/2));
// perform template matching
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
cv::matchTemplate(
  roi,    // search region
  target, // template
  result, // result
  CV_TM_SQDIFF); // similarity measure

// find most similar location
double minVal, maxVal;
cv::Point minPt, maxPt;
cv::minMaxLoc(result, &minVal, &maxVal, &minPt, &maxPt);

// draw rectangle at most similar location
// at minPt in this case
cv::rectangle(roi,
   cv::Rect(minPt.x, minPt.y, target.cols , target.rows),
   255);
```

Remember that this is a costly operation, so you should limit the search area and use a template to a size of only a few pixels.

See also

- The next recipe, *Describing local intensity patterns*, describes the `cv::BFMatcher` class, which implements the matching strategy that was used in this recipe.

We've successfully learned how to match local templates. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Describing local intensity patterns

The SURF and SIFT keypoint detection algorithms, discussed in `Chapter 8`, *Detecting Interest Points*, define a location, an orientation, and a scale for each of the detected features. The scale factor information is useful to define the size of a window of analysis around each feature point. So, the defined neighborhood would include the same visual information no matter what the scale of the object to which the feature belongs has been pictured. This recipe will show you how to describe an interest point's neighborhood using **feature descriptors**. In image analysis, the visual information included in this neighborhood can be used to characterize each feature point in order to make it distinguishable from the others. Feature descriptors are usually N-dimensional vectors that describe a feature point in a way that is invariant to changes in lighting and to small perspective deformations. Generally, descriptors can be compared using simple distance metrics, for example, the Euclidean distance. Therefore, they constitute a powerful tool that can be used in feature matching applications.

How to do it...

The `cv::Feature2D` abstract class defines a number of member functions that are used to compute the descriptors of a list of keypoints. As most feature-based methods include both a detector and a descriptor component, the associated classes include both a `detect` function (to detect the interest points) and a `compute` function (to compute their descriptors). This is the case of the `cv::xfeatures2d::SURF` and `cv::xfeature2d::SIFT` classes.

To create a SURF feature detector, we have to follow the next steps:

1. Create the SURF descriptor, as shown in the following code:

```
// Define feature detector
// Construct the SURF feature detector object
cv::Ptr<cv::Feature2D> ptrFeature2D =
cv::xfeatures2d::SURF::create(2000.0);
```

2. For each image, detect the keypoints:

```
// Keypoint detection
// Detect the SURF features
  detector->detect(image1,keypoints1);
  detector->detect(image2,keypoints2);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. For each of the keypoints detected in the previous step, extract their descriptors:

```
// Extract the descriptor
 cv::Mat descriptors1;
 cv::Mat descriptors2;
 ptrFeature2D->compute(image1,keypoints1,descriptors1);
 ptrFeature2D->compute(image2,keypoints2,descriptors2);
```

4. For SIFT, you will simply create a `SIFT` object instead.

   The result is a matrix (that is, a `cv::Mat` instance) that will contain as many rows as the number of elements in the keypoint vector. Each of these rows is an N-dimensional descriptor vector. In the case of the SURF descriptor, it has a default size of `64` and for SIFT, the default dimension is `128`. This vector characterizes the intensity pattern surrounding a feature point. The more similar the two feature points, the closer their descriptor vectors should be. These descriptors will now be used to match our keypoints.

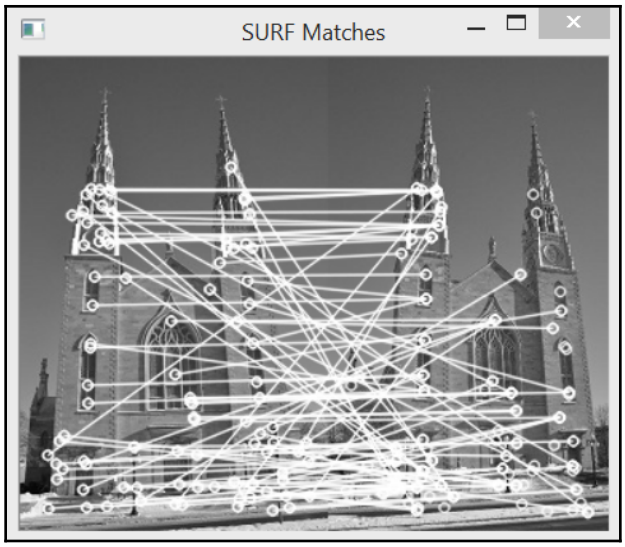
5. Exactly as we did in the previous recipe, each feature descriptor vector in the first image is compared to all the feature descriptors in the second image. The pair that obtains the best score (that is, the pair with the lowest distance between the two descriptor vectors) is then kept as the best match for that feature. This process is repeated for all the features in the first image. Very conveniently, this process is implemented in OpenCV in the `cv::BFMatcher` class, so we do not need to re-implement the double loops that we previously built. This class is used as follows:

```
// Construction of the matcher
cv::BFMatcher matcher(cv::NORM_L2);
// Match the two image descriptors
std::vector<cv::DMatch> matches;
matcher.match(descriptors1,descriptors2, matches);
```

This class is a subclass of the `cv::DescriptorMatcher` class, which defines the common interface for different matching strategies. The result is a vector of the `cv::DMatch` instances.

With the current Hessian threshold for SURF, we obtained `74` keypoints for the first image and `71` for the second. The brute-force approach will then produce `74` matches. Using the `cv::drawMatches` class as in the previous recipe produces the following image:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



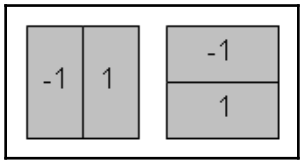
As can be seen, several of these matches correctly link a point on the left-hand side with its corresponding point on the right-hand side. You might notice some errors; these are due to the fact that the observed building has a symmetrical facade, which makes some of the local matches ambiguous. For SIFT, with the same number of keypoints, we obtained the following match result:

Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Good feature descriptors must be invariant to small changes in illumination, viewpoint, and to the presence of image noise. Therefore, they are often based on local intensity differences. This is the case for the SURF descriptors, which locally apply the following simple kernels around a keypoint:



The first kernel simply measures the local intensity difference in the horizontal direction (designated as `dx`), and the second measures this difference in the vertical direction (designated as `dy`). The size of the neighborhood used to extract the descriptor vector is generally defined as 20 times the scale factor of the feature (that is, `20`σ). This square region is then split into 4 x 4 smaller square subregions. For each subregion, the kernel responses (`dx` and `dy`) are computed at 5 x 5 regularly spaced locations (with the kernel size being `2`σ). All of these responses, are summed up, as follows, in order to extract four descriptor values for each subregion:

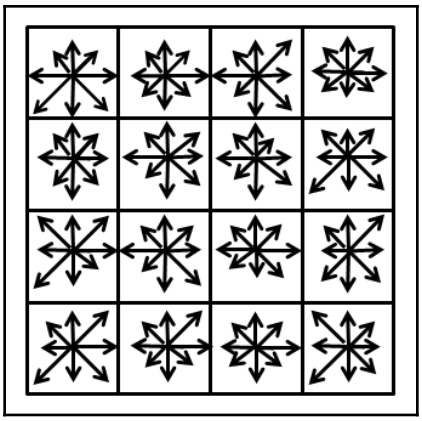
$$[\sum dx \quad \sum dy \quad \sum |dx| \quad \sum |dy|]$$

Since there are `4x4=16` subregions, we have a total of `64` descriptor values. Note that, in order to give more importance to the neighboring pixel, values closer to the keypoint, the kernel responses are weighted by a Gaussian centered at the keypoint location (with σ=`3.3`).

The `dx` and `dy` responses are also used to estimate the orientation of the feature. These values are computed (with a kernel size of `4`σ) within a circular neighborhood of radius `6`σ at locations regularly spaced by intervals of σ. For a given orientation, the responses inside a certain angular interval (π`/3`) are summed, and the orientation giving the longest vector is defined as the dominant orientation.

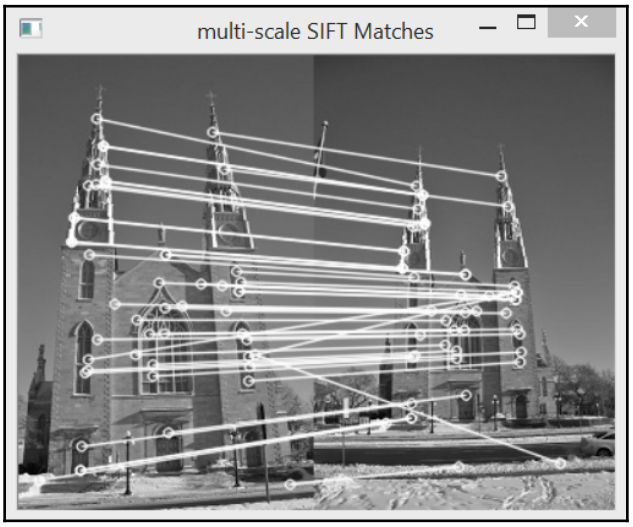
SIFT is a richer descriptor that uses an image gradient instead of simple intensity differences. It also splits the square neighborhood around each keypoint into 4 x 4 subregions (it is also possible to use 8 x 8 or 2 x 2 subregions). Inside each of these regions, a histogram of gradient orientations is built. The orientations are discretized into eight bins, and each gradient orientation entry is incremented by a value proportional to the gradient magnitude. This is illustrated by the following diagram, inside which each star-shaped arrow set represents a local histogram of a gradient orientation:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



These `16` histograms of eight bins each concatenated together then produce a descriptor of `128` dimensions. Note that, as for SURF, the gradient values are weighted by a Gaussian filter centered at the keypoint location in order to make the descriptor less sensitive to sudden changes in gradient orientations at the perimeter of the defined neighborhood. The final descriptor is then normalized to make the distance measurement more consistent.

With SURF and SIFT features and descriptors, scale-invariant matching can be achieved. Here is an example that shows the SURF match result for two images at different scales (here, the 50 best matches have been displayed):



Let's take a look at some other concepts that will be helpful in this recipe.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

There's more...

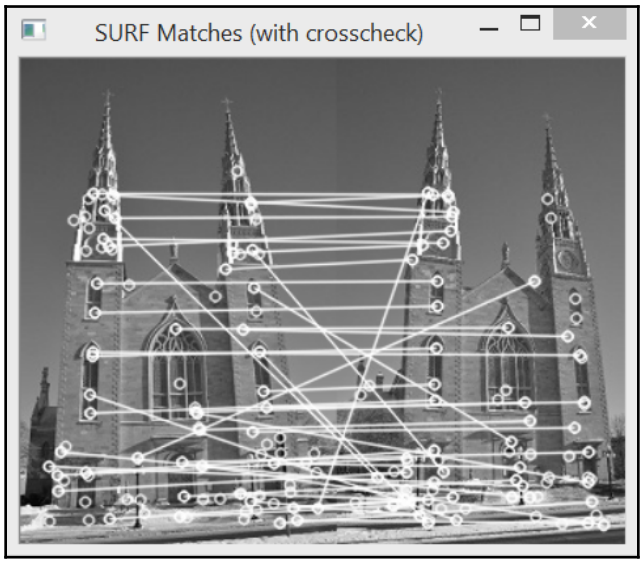
The match result produced by any matching algorithm always contains a significant number of incorrect matches. In order to improve the quality of the match set, there exist a number of strategies. Three of them are discussed here.

Cross-checking matches

A simple approach to validate the matches obtained is to repeat the same procedure a second time, but this time, each keypoint of the second image is compared with all the keypoints of the first image. A match is considered valid only if we obtain the same pair of keypoints in both directions (that is, each keypoint is the best match of the other). The `cv::BFMatcher` function gives us the option to use this strategy. It is included as a flag; when set to `true`, it forces the function to perform the reciprocal match crosscheck:

```
// Construction of the matcher with crosscheck
cv::BFMatcher matcher2(cv::NORM_L2, // distance measure
                       true);       // crosscheck flag
```

The improved match results are as shown in the following screenshot (in the case of SURF):



Let's take a look at the ratio test in the next section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The ratio test

We have already noted that repetitive elements in scene objects create unreliable results because of the ambiguity in matching visually similar structures. What happens in such cases is that a keypoint will match well with more than one other keypoint. Since the probability of selecting the wrong correspondence is high, it might be preferable to reject a match in this case.

To use this strategy, we then need to find the best two matching points of each keypoint. This can be done by using the `knnMatch` method of the `cv::DescriptorMatcher` class. Since we want only the two best matches, we specify `k=2`:

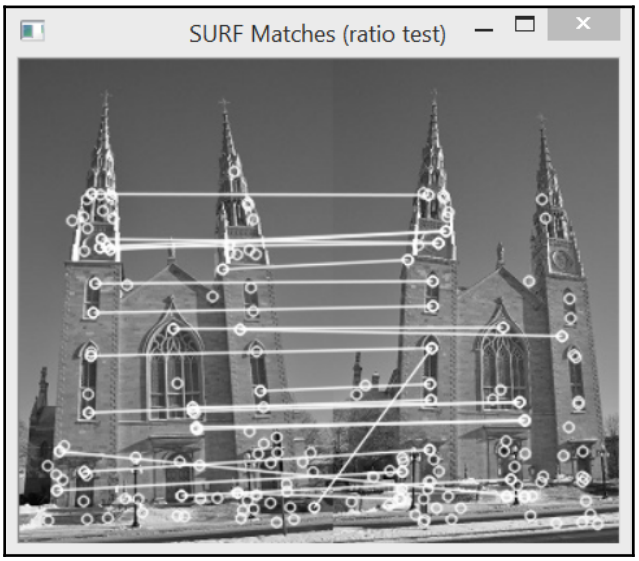
```
// find the best two matches of each keypoint
  std::vector<std::vector<cv::DMatch>> matches2;
  matcher.knnMatch(descriptors1,descriptors2, matches2,
              2); // find the k best matches
```

The next step is to reject all the best matches with a matching distance similar to that of their second best match. Since `knnMatch` produces a `std::vector` class of `std::vector` (this second vector is of the size `k`), we do this by looping over each keypoint match and perform a ratio test (this ratio will be one if the two best distances are equal). Here is how we can do it:

```
// perform ratio test
double ratio= 0.85;
std::vector<std::vector<cv::DMatch>>::iterator it;
for (it= matches2.begin(); it!= matches2.end(); ++it) {

  //   first best match/second best match
  if ((*it)[0].distance/(*it)[1].distance < ratio) {
    // it is an acceptable match
    matches.push_back((*it)[0]);
  }
}
// matches is the new match set
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The initial match set made up of `74` pairs is now reduced to `23` pairs; a good proportion of these are now correct matches:



Let's take a look at distance thresholding in the next section.

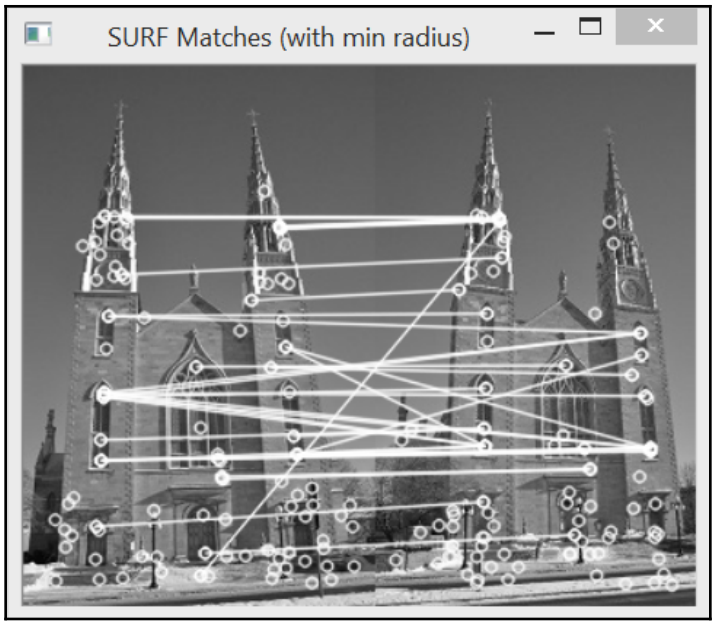
Distance thresholding

An even simpler strategy consists of rejecting matches for which the distance between their descriptors is too high. This is done using the `radiusMatch` method of the `cv::DescriptorMatcher` class:

```
// radius match
float maxDist= 0.4;
std::vector<std::vector<cv::DMatch>> matches2;
matcher.radiusMatch(descriptors1, descriptors2, matches2,
                    maxDist); // maximum acceptable distance
                              // between the 2 descriptors
```

The result is again an `std::vector` instance because the method will retain all the matches with a distance smaller than the specified threshold. This means that a given keypoint might have more than one matching point in the other image. Conversely, other keypoints will not have any matches associated with them (the corresponding inner `std::vector` class will then have a size of `0`). For our example, the result is a match set of `50` pairs:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Obviously, you can combine all these strategies in order to improve your matching results.

See also

- The *Detecting scale-invariant features* recipe in `Chapter 8`, *Detecting Interest Points*, presents the associated SURF and SIFT feature detectors and provides more references on the subject.
- The *Matching images using random sample consensus* recipe in `Chapter 10`, *Estimating Projective Relations in Images*, explains how to use the image and scene geometry in order to obtain a match set of even better quality.
- The *Matching Feature Points in Stereo Pairs: A Comparative Study of Some Matching Strategies* article by E. Vincent and R. Laganière in *Machine, Graphics and Vision, pp. 237-260, 2001*, describes other simple matching strategies that could be used to improve the quality of the match set.

We've successfully learned how to describe local intensity patterns. Now, let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Describing keypoints with binary features

In the previous recipe, we learned how to describe a keypoint using rich descriptors extracted from the image intensity gradient. These descriptors are floating-point vectors that have a dimension of `64`, `128`, or sometimes even longer. This makes them costly to manipulate. In order to reduce the memory and computational load associated with these descriptors, the idea of using binary descriptors has been recently introduced. The challenge here is to make them easy to compute and yet keep them robust to scene and viewpoint changes. This recipe describes some of these binary descriptors. In particular, we will look at the **ORB** (short for **Oriented FAST and Rotated BRIEF**) and **BRISK** (short for **Binary Robust Invariant Scalable Keypoints**) descriptors for which we presented their associated feature point detectors in `Chapter 8`, *Detecting Interest Points*.

How to do it...

Owing to the nice generic interface on the top of which the OpenCV detectors and the descriptors module are built, using a binary descriptor such as ORB is no different from using descriptors such as SURF and SIFT. The complete feature-based image matching sequence is as follows:

1. Define the keypoints vectors:

```
// Define keypoints vector
std::vector<cv::KeyPoint> keypoints1, keypoints2;
```

2. Create the `Mat` structure to store the descriptors:

```
cv::Mat descriptors1;
cv::Mat descriptors2;
```

3. Create the pointer to the `ORB` detector:

```
// Define feature detector/descriptor
// Construct the ORB feature object
cv::Ptr<cv::Feature2D> feature = cv::ORB::create(60);
// approx. 60 feature points
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Detect and compute the keypoints and descriptors for each image:

```
// Keypoint detection and description
// Detect the ORB features
feature->detectAndCompute(image1, cv::noArray(), keypoints1,
descriptors1);
feature->detectAndCompute(image2, cv::noArray(), keypoints2,
descriptors2);
```

5. Create `matcher` as follows:

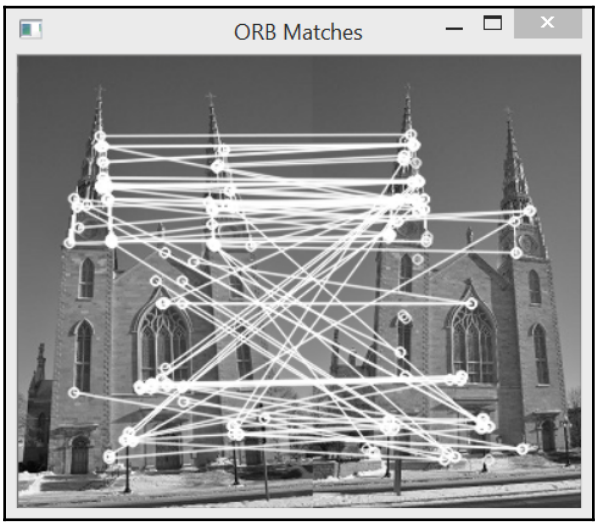
```
// Construction of the matcher
cv::BFMatcher matcher(cv::NORM_HAMMING); // always use hamming norm
for binary descriptors
```

6. Match the two image descriptors:

```
std::vector<cv::DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
```

The only difference resides in the use of the **Hamming** norm (the `cv::NORM_HAMMING` flag), which measures the distance between two binary descriptors by counting the number of bits that are dissimilar. On many processors, this operation is efficiently implemented by using an exclusive OR operation, followed by a simple bit count.

The following screenshot shows the result of the matching:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Similar results will be obtained with another popular binary feature detector/descriptor: BRISK. In this case, the `cv::DescriptorExtractor` instance is created by the new `cv::BRISK(40)` call. As we learned in `Chapter 8`, *Detecting Interest Points*, its first parameter is a threshold that controls the number of detected points.

Now, let's go behind the scenes to understand the code better.

How it works...

The ORB algorithm detects oriented feature points at multiple scales. Based on this result, the ORB descriptor extracts a representation of each keypoint by using simple intensity comparisons. In fact, ORB builds on a previously proposed descriptor called **BRIEF** (short for **Binary Robust Independent Elementary Features**). This later creates a binary descriptor by simply selecting a random pair of points inside a defined neighborhood around the keypoint. The intensity values of the two pixel points are then compared, and if the first point has a higher intensity, then the value `1` is assigned to the corresponding descriptor bit value. Otherwise, the value `0` is assigned. Repeating this test on a number of random pairs generates a descriptor that is made up of several bits; typically, `128` to `512` bits (pairwise tests) are used.

This is the scheme used by ORB. Then, the decision to be made is which set of point pairs should be used to build the descriptor. Indeed, even if the point pairs are randomly chosen, once they have been selected, the same set of binary tests must be performed to build the descriptor of all the keypoints in order to ensure consistency of the results. To make the descriptor more distinctive, intuition tells us that some choices must be better than the others. Also, the fact that the orientation of each keypoint is known introduces some bias in the intensity pattern distribution when this one is normalized with respect to this orientation (that is, when the point coordinates are given relative to this keypoint orientation). From these considerations and the experimental validation, ORB has identified a set of `256` point pairs with high variance and minimal pairwise correlation. In other words, the selected binary tests are the ones that have an equal chance of being `0` or `1` over a variety of keypoints and also those that are as independent from each other as possible.

In addition to the parameters that control the feature detection process, the `cv::ORB` constructor includes two parameters related to its descriptor. One parameter is used to specify the patch size inside which the point pairs are selected (the default is 31 x 31). The second parameter allows you to perform tests with a triplet or quadruplet of points instead of the default point pairs. Note that it is highly recommended that you use the default settings.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The descriptor of BRISK is very similar. It is also based on pairwise intensity comparisons with two differences. First, instead of randomly selecting the points from the 31 x 31 points of the neighborhood, the chosen points are selected from a sampling pattern of a set of concentric circles (made up of `60` points) with locations that are equally spaced. Second, the intensity at each of these sample points is a Gaussian-smoothed value with a σ value proportional to the distance from the central keypoint. From these points, BRISK selects `512` point pairs.

There's more...

Several other binary descriptors exist, and interested readers should take a look at the scientific literature to learn more on this subject. Since it is also available in OpenCV, we will describe one additional descriptor here.

FREAK

FREAK stands for **Fast Retina Keypoint**. This is also a binary descriptor, but it does not have an associated detector. It can be applied on any set of keypoints detected, for example, SIFT, SURF, or ORB.

Like BRISK, the FREAK descriptor is also based on a sampling pattern defined on concentric circles. However, to design their descriptor, the authors used an analogy with the human eye. They observed that on the retina, the density of the ganglion cells decreases with the increase in the distance to the fovea. Consequently, they built a sampling pattern made of `43` points in which the density of a point is much greater near the central point. To obtain its intensity, each point is filtered with a Gaussian kernel that has a size that also increases with the distance to the center.

In order to identify the pairwise comparisons that should be performed, an experimental validation has been performed by following a strategy similar to the one used for ORB. By analyzing several thousands of keypoints, the binary tests with the highest variance and lowest correlation are retained, resulting in `512` pairs.

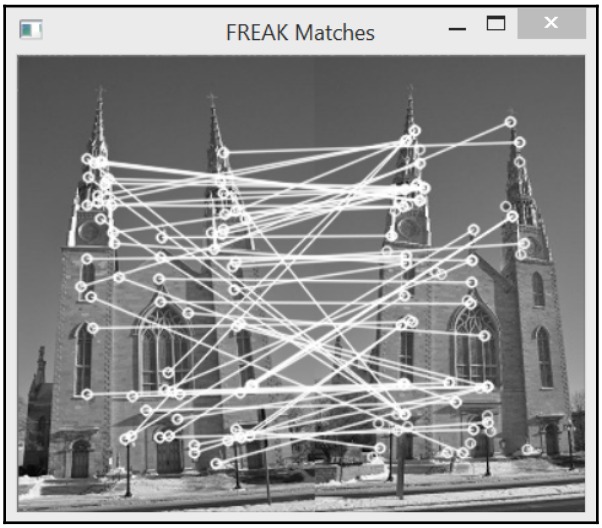
FREAK also introduces the idea of performing the descriptor comparisons in cascade. That is, the first `128` bits representing coarser information (corresponding to the tests performed at the periphery on larger Gaussian kernels) are performed first. Only if the compared descriptors pass this initial step will the remaining tests be performed.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

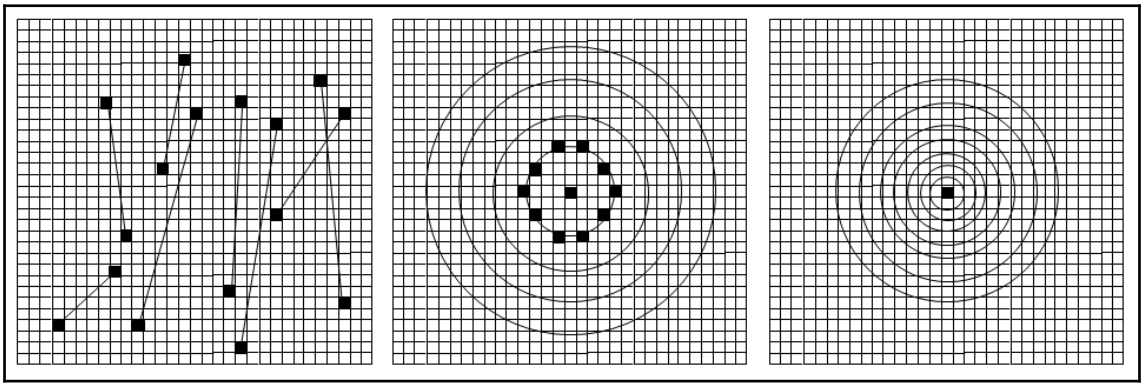
Using the keypoints detected with ORB, we extract the FREAK descriptors by simply creating the `cv::DescriptorExtractor` instance as follows:

```
cv::Ptr<cv::DescriptorExtractor> descriptor =
   new cv::FREAK(); // to describe with FREAK
```

The match result is as follows:



The following diagram illustrates the sampling pattern used for the three descriptors presented in this recipe:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The first square is the ORB descriptor in which point pairs are randomly selected on a square grid. Each pair of points linked by a line represent a possible test to compare the two pixel intensities. Here, we show only eight such pairs; the default ORB uses 256 pairs. The middle square corresponds to the BRISK sampling pattern. Points are uniformly sampled on the shown circles (for clarity, we only identify the points on the first circle here). Finally, the third square shows the log-polar sampling grid of FREAK. While BRISK has a uniform distribution of points, FREAK has a higher density of points closer to the center. For example, in BRISK, you find 20 points on the outer circle, while, in the case of FREAK, its outer circle includes only six points.

See also

- The *Detecting fast features at multiple scales* recipe in `Chapter 8`, *Detecting Interest Points,* presents the associated BRISK and ORB feature detectors and provides more references on the subject.
- The *BRIEF: Computing a Local Binary Descriptor Very Fast* article by E. M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, P. Fua in *IEEE Transactions on Pattern Analysis and Machine Intelligence, 2012*, describes the BRIEF feature descriptor that inspires the presented binary descriptors.
- The *FREAK: Fast Retina Keypoint* article by A.Alahi, R. Ortiz, P. Vandergheynst in *IEEE Conference on Computer Vision and Pattern Recognition, 2012*, describes the FREAK feature descriptor.

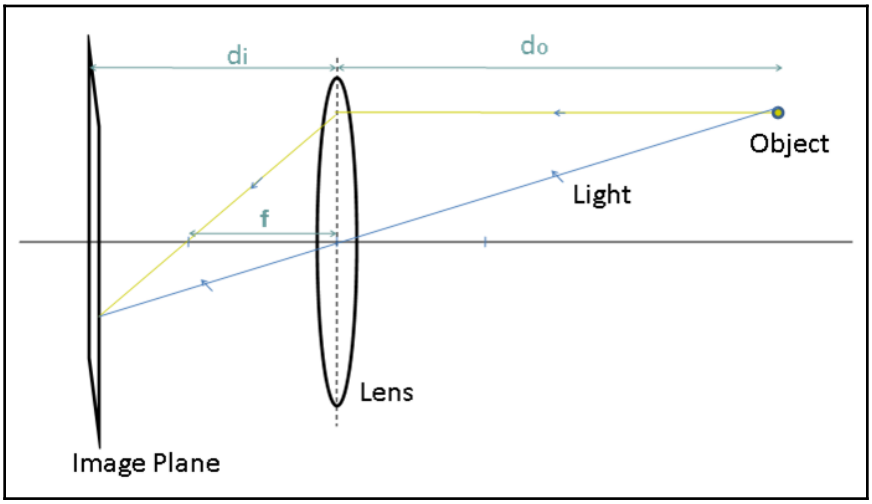
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

10
Estimating Projective Relations in Images

Images are generally produced using a digital camera, which captures a scene by projecting light going through its lens onto an image sensor. The fact that an image is formed by the projection of a 3D scene onto a 2D plane shows the existence of important relationships between a scene and its image and between different images of the same scene. Projective geometry is the tool that is used to describe and characterize, in mathematical terms, the process of image formation. In this chapter, we will introduce you to some of the fundamental projective relations that exist in multiview imagery and explain how they can be used in computer vision programming. You will learn how matching can be made more accurate through the use of projective constraints, or how a mosaic from multiple images can be composited using two-view relations. Before we start the recipes, let's explore the basic concepts related to scene projection and image formation.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Fundamentally, the process used to produce images has not changed since the beginning of photography. The light coming from an observed scene is captured by a camera through a frontal **aperture**; the captured light rays then hit an **image plane** (or an **image sensor**) located at the back of the camera. Additionally, a lens is used to concentrate the rays coming from different scene elements. This process is illustrated by the following diagram:

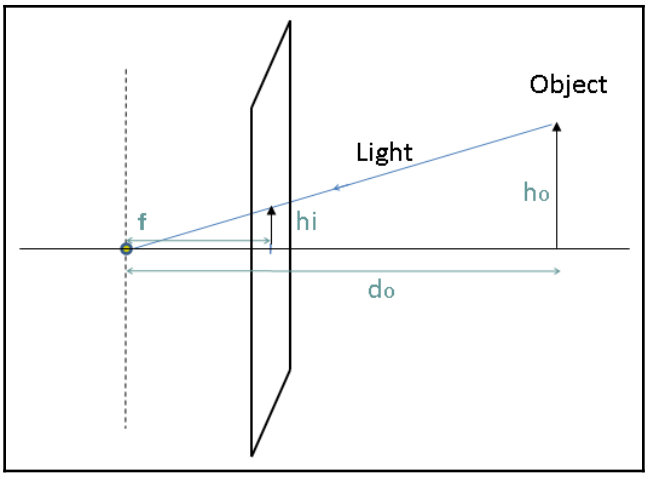


Here, **do** is the distance from the lens to the observed object, **di** is the distance from the lens to the image plane, and **f** is the focal length of the lens. These quantities are related by the so-called **thin lens equation**:

$$\frac{1}{f} = \frac{1}{do} + \frac{1}{di}$$

In computer vision, this camera model can be simplified in a number of ways. First, we can ignore the effect of the lens by considering that we have a camera with an infinitesimal aperture since, in theory, this does not change the image appearance (however, by doing so, we ignore the focusing effect by creating an image with an infinite **depth of field**). In this case, therefore, only the central ray is considered. Second, since most of the time we have **do>>di**, we can assume that the image plane is located at the focal distance. Finally, we can note from the geometry of the system that the image on the plane is inverted. We can obtain an identical but upright image by simply positioning the image plane in front of the lens. Obviously, this is not physically feasible, but from a mathematical point of view, this is the equivalent. This simplified model is often referred to as the **pin-hole camera** model, and it is represented as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



From this model, and using the law of similar triangles, we can easily derive the basic projective equation that relates a pictured object with its image:

$$hi = f\frac{ho}{do}$$

The size (*hi*) of the image of an object (of height *ho*) is, therefore, inversely proportional to its distance (*do*) from the camera, which is naturally true. In general, this relation describes where a 3D scene point will be projected on the image plane given the geometry of the camera. More specifically, if we assume that the reference frame is positioned at the focal point, then a 3D scene point located at position *(X, Y, Z)* will be projected onto the image plane at *(x,y)=(fX/Z,fY/Z)*. Here, the *Z* coordinate corresponds with the depth of the point (or distance to the camera, denoted by *do* in the previous equation). This relation can be rewritten in a simple matrix form through the introduction of homogeneous coordinates, in which 2D points are represented by three-vectors, and 3D points are represented by four-vectors (the extra coordinate is simply an arbitrary scale factor *S* that needs to be removed when a 2D coordinate needs to be extracted from a homogeneous three-vector):

$$S\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This *3 x 4* matrix is called the **projection matrix**. In cases where the reference frame is not aligned with the focal point, then the rotation `r` and translation `t` matrices must be introduced. The role of these is simply to express the projected 3D point into a camera-centric reference frame, which is as follows:

$$S\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

The first matrix of this equation is said to contain the intrinsic parameters of the camera (here, only the focal length, but the next chapter will introduce a few more intrinsic parameters). The second matrix contains the extrinsic parameters that are the parameters that relate the camera to the exterior world. It should be noted that, in practice, image coordinates are expressed in pixels, while 3D coordinates are expressed in world measurements (for example, meters). This aspect will be explored in `Chapter 11`, *Reconstructing 3D Scenes*.

In this chapter, we will cover the following recipes:

- Computing the fundamental matrix of an image pair
- Matching images using a random sample consensus
- Computing a homography between two images
- Detecting a planar target in images

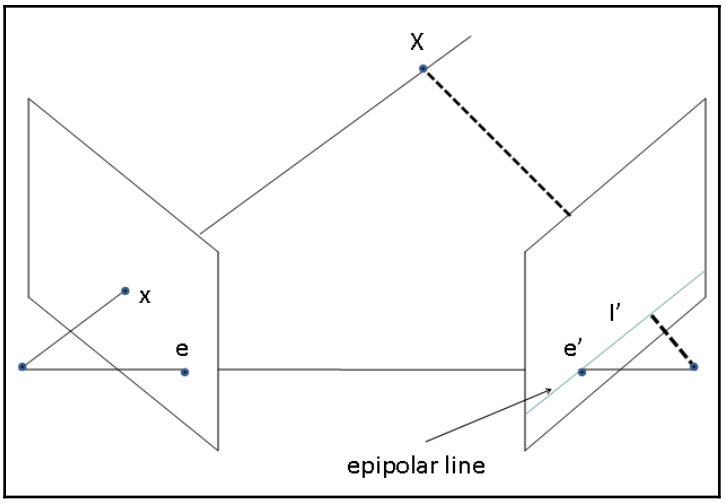
Computing the fundamental matrix of an image pair

In this recipe, we will explore the projective relationship that exists between two images that display the same scene. These two images could have been obtained by moving a camera at two different locations to take pictures from two viewpoints, or by using two cameras, each of them taking a different picture of the scene. When these two cameras are separated by a rigid baseline, we use the term **stereovision**.

Getting ready

Let's now consider two cameras observing a given scene point, as shown in the following diagram:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



We learned that we can find the image **x** of a 3D point **X** by tracing a line joining this 3D point with the camera's center. Conversely, the scene point that has its image at the position **x** on the image plane can be located anywhere on this line in the 3D space. This implies that if we want to find the corresponding point of a given image point in another image, we need to search along with the projection of this line onto the second image plane. This imaginary line is called the **epipolar line** of point **X**. It defines a fundamental constraint that must satisfy two corresponding points; that is, the match of a given point must lie on the epipolar line of this point in the other view, and the exact orientation of this epipolar line depends on the respective position of the two cameras. In fact, the configuration of the epipolar line characterizes the geometry of a two-view system.

Another observation that can be made from the geometry of this two-view system is that all the epipolar lines pass through the same point. This point corresponds to the projection of one camera's center onto the other camera. This special point is called an **epipole**.

Mathematically, the relationship between an image point and its corresponding epipolar line can be expressed using a *3 x 3* matrix as follows:

$$\begin{bmatrix} l'_1 \\ l'_2 \\ l'_3 \end{bmatrix} = F \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In projective geometry, a 2D line is also represented by a 3-vector. It corresponds to the set of 2D points, *(x',y')*, that satisfies the `l1'x'+ l2'y'+ l3'=0` equation (the prime superscript denotes that this line belongs to the second image). Consequently, the `F` matrix, called the fundamental matrix, maps a 2D image point in one view to an epipolar line in the other view.

How to do it...

The fundamental matrix of an image pair can be estimated by solving a set of equations that involve a certain number of known matched points between the two images. The minimum number of such matches is seven, as described in the following steps:

1. In order to illustrate the fundamental matrix estimation process and using the image pair from `Chapter 9`, *Describing and Matching Interest Points*, we selected seven good matches from the matching results of SIFT features, as presented in `Chapter 9`, *Describing and Matching Interest Points*.

   These matches will be used to compute the fundamental matrix using the `cv::findFundamentalMat` OpenCV function. The image pair with its selected matches are shown in the following screenshot:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. If we have the image points in each image as the `cv::keypoint` instances (for example, if they were detected using a keypoint detector as in `Chapter 8`, *Detecting Interest Points*), they first need to be converted into `cv::Point2f` in order to be used with `cv::findFundamentalMat`. An OpenCV function can be used to this end:

```
// Convert keypoints into Point2f
std::vector<cv::Point2f> selPoints1, selPoints2;
std::vector<int> pointIndexes1, pointIndexes2;
cv::KeyPoint::convert(keypoints1,selPoints1,pointIndexes1);
cv::KeyPoint::convert(keypoints2,selPoints2,pointIndexes2);
```

3. The two vectors, `selPoints1` and `selPoints2`, contain the corresponding points in the two images. The keypoint instances are `keypoints1` and `keypoints2`. The `pointIndexes1` and `pointIndexes2` vectors contain the indexes of the keypoints to be converted. The call to the `cv::findFundamentalMat` function is then as follows:

```
// Compute F matrix from 7 matches
cv::Mat fundamental= cv::findFundamentalMat(
   selPoints1,    // 7 points in first image
   selPoints2,    // 7 points in second image
   CV_FM_7POINT); // 7-point method
```

4. One way to visually verify the validity of the fundamental matrix is to draw the epipolar lines of some selected points. Another OpenCV function allows the epipolar lines of a given set of points to be computed. Once these are computed, they can be drawn using the `cv::line` function. The following lines of code accomplish these two steps (that is, computing and drawing epipolar lines in the image on the right from the points in the image on the left):

```
// draw the left points corresponding epipolar
// lines in right image
std::vector<cv::Vec3f> lines1;
cv::computeCorrespondEpilines(
   selPoints1,  // image points
   1,           // in image 1 (can also be 2)
   fundamental, // F matrix
   lines1);     // vector of epipolar lines

// for all epipolar lines
for (vector<cv::Vec3f>::const_iterator it= lines1.begin();
    it!=lines1.end(); ++it) {
       // draw the line between first and last column
       cv::line(image2,
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
                cv::Point(0,-(*it)[2]/(*it)[1]),
                cv::Point(image2.cols,-((*it)[2]+
                          (*it)[0]*image2.cols)/(*it)[1]),
                          cv::Scalar(255,255,255));
    }
```

The result can be seen in the following screenshot:

Remember that the epipole is at the intersection of all the epipolar lines, and it is the projection of the other camera's center. This epipole is visible in the preceding image. Often, the epipolar lines intersect outside the image boundaries. In the case of our example, it is at the location where the first camera would be visible if the two images were taken at the same instant. Note that the results can be quite unstable when the fundamental matrix is computed from seven matches. Indeed, substituting one match for another could lead to a significantly different set of epipolar lines.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

We previously explained that for a point in one image, the fundamental matrix gives the equation of the line on which its corresponding point in the other view should be found. If the corresponding point of a `p(x, y)` point (expressed in homogenous coordinates) is `p'(x', y'x)` and if `F` is the fundamental matrix between the two views, then since `p'(x', y')` lies on the `Fp` epipolar line, we have the following equation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}^T F \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

This equation expresses the relationship between two corresponding points and is known as the **epipolar constraint**. Using this equation, it becomes possible to estimate the entries of the matrix using known matches. Since the entries of the `F` matrix are given up to a scale factor, there are only eight entries to be estimated (the ninth can be arbitrarily set to `1`). Each match contributes to one equation. Therefore, with eight known matches, the matrix can be fully estimated by solving the resulting set of linear equations. This is what is done when you use the `CV_FM_8POINT` flag with the `cv::findFundamentalMat` function. Note that, in this case, it is possible (and preferable) to input more than eight matches. The obtained over-determined system of linear equations can then be solved in a mean-square sense.

To estimate the fundamental matrix, an additional constraint can also be exploited. Mathematically, the `F` matrix maps a 2D point to a 1D pencil of lines (that is, lines that intersect at a common point). The fact that all these epipolar lines pass through this unique point (that is, the epipole) imposes a constraint on the matrix. This constraint reduces the number of matches required to estimate the fundamental matrix to seven. Unfortunately, in this case, the set of equations become nonlinear with up to three possible solutions (in this case, `cv::findFundamentalMat` will return a fundamental matrix of the size *9 x 3*, that is, three *3 x 3* matrices stacked up). The seven-match solution of the `F` matrix estimation can be invoked in OpenCV by using the `CV_FM_7POINT` flag. This is what we did in the example of the preceding section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Lastly, we would like to mention that the choice of an appropriate set of matches in the image is important to obtain an accurate estimation of the fundamental matrix. In general, the matches should be well distributed across the image and include points at different depths in the scene. Otherwise, the solution will become unstable or degenerate configurations. In particular, the selected scene points should not be coplanar, as the fundamental matrix (in this case) becomes degenerated.

See also

- *Multiple View Geometry in Computer Vision*, *Cambridge University Press*, 2004, R. Hartley and A. Zisserman, is the most complete reference on projective geometry in computer vision.
- The *Computing a homography between two images* recipe explains why a fundamental matrix cannot be computed when the matched points are coplanar or are the result of a pure rotation.

We've successfully learned how to compute the fundamental matrix of an image pair. Now, let's move on to the next recipe!

Matching images using a random sample consensus

When two cameras observe the same scene, they see the same elements but under different viewpoints. We have already studied the feature point matching problem in `Chapter 8`, *Detecting Interest Points*. In this recipe, we come back to this problem, and we will learn how to exploit the epipolar constraint between two views to match image features more reliably.

The principle that we will follow is simple – when we match feature points between two images, we only accept those matches that fall on the corresponding epipolar lines. However, to be able to check this condition, the fundamental matrix must be known, but we need good matches to estimate this matrix. This seems to be a chicken-and-egg problem. However, in this recipe, we propose a solution in which the fundamental matrix and a set of good matches will be jointly computed.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

The objective is to be able to compute a fundamental matrix and a set of good matches between two views. To do so, all the found feature point correspondences will be validated using the epipolar constraint introduced in the previous recipe:

1. To this end, we have created a class that encapsulates the different steps of the proposed robust matching process:

```
class RobustMatcher {
  private:
    // pointer to the feature point detector object
    cv::Ptr<cv::FeatureDetector> detector;
    // pointer to the feature descriptor extractor object
    cv::Ptr<cv::DescriptorExtractor> extractor;
    int normType;
    float ratio; // max ratio between 1st and 2nd NN
    bool refineF; // if true will refine the F matrix
    double distance; // min distance to epipolar
    double confidence; // confidence level (probability)

  public:
    RobustMatcher(std::string detectorName, // specify by name
                  std::string descriptorName)
      : normType(cv::NORM_L2), ratio(0.8f),
          refineF(true), confidence(0.98), distance(3.0) {

      // construct by name
      if (detectorName.length()>0) {
      detector= cv::FeatureDetector::create(detectorName);
      extractor= cv::DescriptorExtractor::
                           create(descriptorName);
      }
    }
```

Note how we used the `create` methods of the `cv::FeatureDetector` and `cv::DescriptorExtractor` interfaces so that a user can select the `create` methods by their names. Note that the `create` methods can also be specified using the defined `setFeatureDetector` and `setDescriptorExtractor` setter methods.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. The main method is our `match` method, which returns matches, detected keypoints, and the estimated fundamental matrix. The method proceeds in four distinct steps (explicitly identified in the comments of the following code) that we will now explore:

```
// Match feature points using RANSAC
// returns fundamental matrix and output match set
cv::Mat match(cv::Mat& image1, cv::Mat& image2, // input images
              std::vector<cv::DMatch>& matches,      // output
matches
              std::vector<cv::KeyPoint>& keypoints1, // output
keypoints
              std::vector<cv::KeyPoint>& keypoints2) {
    // 1. Detection of the feature points
    detector->detect(image1,keypoints1);
    detector->detect(image2,keypoints2);

    // 2. Extraction of the feature descriptors
    cv::Mat descriptors1, descriptors2;
    extractor->compute(image1,keypoints1,descriptors1);
    extractor->compute(image2,keypoints2,descriptors2);

    // 3. Match the two image descriptors
    //    (optionnally apply some checking method)
    // Construction of the matcher with crosscheck
    cv::BFMatcher matcher(normType,   //distance measure
                          true);     // crosscheck flag
    // match descriptors
      std::vector<cv::DMatch> outputMatches;
    matcher.match(descriptors1,descriptors2,outputMatches);

    // 4. Validate matches using RANSAC
    cv::Mat fundaental= ransacTest(outputMatches,
                           keypoints1, keypoints2, matches);

    // return the found fundamental matrix
    return fundamental;
  }
```

The first two steps simply detect the feature points and compute their descriptors. Next, we proceed to feature matching using the `cv::BFMatcher` class, as we did in `Chapter 8`, *Detecting Interest Points*. We use the `crosscheck` flag to obtain matches of better quality.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The fourth step is the new concept introduced in this recipe. It consists of an additional filtering test that will, this time, use the fundamental matrix in order to reject matches that do not obey the epipolar constraint.

3. To do this test based on the RANSAC method that can compute the fundamental matrix even when outliers are still present in the match set, we write the following function (this method will be explained in the next section):

```
// Identify good matches using RANSAC
// Return fundamental matrix and output matches
cv::Mat ransacTest(const std::vector<cv::DMatch>& matches,
                   const std::vector<cv::KeyPoint>& keypoints1,
                const std::vector<cv::KeyPoint>& keypoints2,
              std::vector<cv::DMatch>& outMatches) {

// Convert keypoints into Point2f
  std::vector<cv::Point2f> points1, points2;
  for (std::vector<cv::DMatch>::const_iterator it=
matches.begin(); it!= matches.end(); ++it) {

       // Get the position of left keypoints
       points1.push_back(keypoints1[it->queryIdx].pt);
       // Get the position of right keypoints
       points2.push_back(keypoints2[it->trainIdx].pt);
    }

  // Compute F matrix using RANSAC
  std::vector<uchar> inliers(points1.size(),0);
  cv::Mat fundamental= cv::findFundamentalMat(
    points1,points2, // matching points
      inliers,      // match status (inlier or outlier)
      CV_FM_RANSAC, // RANSAC method
      distance,     // distance to epipolar line
      confidence);  // confidence probability
  // extract the surviving (inliers) matches
  std::vector<uchar>::const_iterator itIn= inliers.begin();
  std::vector<cv::DMatch>::const_iterator itM= matches.begin();
  // for all matches
  for ( ;itIn!= inliers.end(); ++itIn, ++itM) {
    if (*itIn) { // it is a valid match
      outMatches.push_back(*itM);
    }
  }
  return fundamental;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. This code is a bit long because the keypoints need to be converted into `cv::Point2f` before the `F` matrix computation. Using this class, the robust matching of an image pair is then easily accomplished by the following calls:

```
// Prepare the matcher (with default parameters)
RobustMatcher rmatcher("SURF"); // we use SURF features here
// Match the two images
std::vector<cv::DMatch> matches;
std::vector<cv::KeyPoint> keypoints1, keypoints2;
cv::Mat fundamental= rmatcher.match(image1,image2,
                        matches, keypoints1, keypoints2);
```

This results in `62` matches that are shown in the following screenshot:

Interestingly, almost all these matches are correct, even if a few false matches remain; later, these accidentally fell on the corresponding epipolar lines of the computed fundamental matrix.

How it works...

In the preceding recipe, we learned that it is possible to estimate the fundamental matrix associated with an image pair from a number of feature point matches. Obviously, to be exact, this matching set must be made up of only good matches. However, in a real context, it is not possible to guarantee that a matching set obtained by comparing the descriptors of the detected feature points will be completely exact. This is why a fundamental matrix estimation method based on the **RANSAC** (**RANdom SAmpling Consensus**) strategy has been introduced.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The RANSAC algorithm aims at estimating a given mathematical entity from a dataset that may contain a number of outliers. The idea is to randomly select some data points from the set and perform the estimation only with these. The number of selected points should be the minimum number of points required to estimate the mathematical entity. In the case of the fundamental matrix, eight matched pairs are the minimum number (in fact, it could be seven matches, but the eight-point linear algorithm is faster to compute). Once the fundamental matrix is estimated from these eight random matches, all the other matches in the match set are tested against the epipolar constraint that derives from this matrix. All the matches that fulfill this constraint (that is, matches for which the corresponding feature is at a short distance from its epipolar line) are identified. These matches form the **support set** of the computed fundamental matrix.

The central idea behind the RANSAC algorithm is that the larger the support set is, the higher the probability that the computed matrix is the right one. Conversely, if one (or more) of the randomly selected matches is an incorrect match, then the computed fundamental matrix will also be incorrect, and its support set is expected to be small. This process is repeated a number of times, and, in the end, the matrix with the largest support will be retained as the most probable one.

Therefore, our objective is to pick eight random matches several times so that eventually we select eight good ones, which should give us a large support set. Depending on the number of wrong matches in the entire dataset, the probability of selecting a set of eight correct matches will differ. We, however, know that the more selections we make, the higher our confidence will be that we have at least one good match set among those selections. More precisely, if we assume that the match set is made of `w%` inliers (good matches), then the probability that we select eight good matches is `w8`. Consequently, the probability that a selection contains at least one wrong match is `(1-w8)`. If we make `k` selections, the probability of having one random set that contains good matches is only `1-(1-w8)k`. This is the confidence probability, `c`, and we want this probability to be as high as possible, since we need at least one good set of matches in order to obtain the correct fundamental matrix. Therefore, when running the RANSAC algorithm, you need to determine the number of `k` selections that need to be made in order to obtain a given confidence level.

When using the `cv::findFundamentalMat` function with the `CV_FM_RANSAC` method, two extra parameters are provided. The first parameter is the confidence level, which determines the number of iterations to be made (by default, it is `0.99`). The second parameter is the maximum distance to the epipolar line for a point to be considered as an inlier. All the matched pairs in which a point is at a greater distance from its epipolar line than the distance specified will be reported as an outlier. The function also returns `std::vector` of the character value, indicating that the corresponding match in the input set has been identified as an outlier (`0`) or as an inlier (`1`).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The more good matches you have in your initial match set, the higher the probability is that RANSAC will give you the correct fundamental matrix. This is why we applied the `crosscheck` filter when matching the feature points. You could have also used the ratio test presented in the previous recipe in order to further improve the quality of the final match set. It is just a question of balancing the computational complexity, the final number of matches, and the required level of confidence that the obtained match set will contain only exact matches.

There's more...

The result of the robust matching process presented in this recipe is an estimate of the fundamental matrix computed using the eight selected matches that have the largest support and the set matches included in this support set. Using this information, it is possible to refine these results in two ways.

Refining the fundamental matrix

Since we now have a good quality matching set, as in the last step, it might be a good idea to use all of them to re-estimate the fundamental matrix. We already mentioned that there exists a linear eight-point algorithm to estimate this matrix. We can, therefore, obtain an over-determined system of equations that will solve the fundamental matrix in a least-squares sense. This step can be added at the end of our `ransacTest` function:

```
if (refineF) {
// The F matrix will
  // be recomputed with all accepted matches

  // Convert keypoints into Point2f
  points1.clear();
  points2.clear();
  for (std::vector<cv::DMatch>::
              const_iterator it= outMatches.begin();
     it!= outMatches.end(); ++it) {

     // Get the position of left keypoints
     points1.push_back(keypoints1[it->queryIdx].pt);
     // Get the position of right keypoints
     points2.push_back(keypoints2[it->trainIdx].pt);
  }

  // Compute 8-point F from all accepted matches
  fundamental= cv::findFundamentalMat(
    points1,points2, // matching points
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        CV_FM_8POINT); // 8-point method solved using SVD
    }
```

The `cv::findFundamentalMat` function accepts more `8` matches by solving the linear system of equations using singular value decomposition.

Refining the matches

We learned that in a two-view system, every point must lie on the epipolar line of its corresponding point. This is the epipolar constraint expressed by the fundamental matrix. Consequently, if you have a good estimate of a fundamental matrix, you can use this epipolar constraint to correct the obtained matches by forcing them to lie on their epipolar lines. This can be easily done by using the `cv::correctMatches` OpenCV function:

```
std::vector<cv::Point2f> newPoints1, newPoints2;
// refine the matches
correctMatches(fundamental,             // F matrix
               points1, points2,        // original position
              newPoints1, newPoints2); // new position
```

This function proceeds by modifying the position of each corresponding point so that it satisfies the epipolar constraint while minimizing the cumulative (squared) displacement.

We've successfully learned how to cmatch images using a random sample consensus. Now let's move on to the next recipe!

Computing a homography between two images

The second recipe of this chapter showed you how to compute the fundamental matrix of an image pair from a set of matches. In projective geometry, another very useful mathematical entity also exists. This one can be computed from multiview imagery, and, as we will see, is a matrix with special properties.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

Again, let's consider the projective relation between a 3D point and its image on a camera, which we introduced in the first recipe of this chapter. Basically, we learned that this equation relates a 3D point with its image using the intrinsic properties of the camera and the position of this camera (specified with a rotation and a translation component). If we now carefully examine this equation, we realize that there are two special situations of particular interest. The first situation is when two views of a scene are separated by a pure rotation. It can then be observed that the fourth column of the extrinsic matrix will be made up of *0*'s (that is, the translation is null):

$$S\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & 0 \\ r_4 & r_5 & r_6 & 0 \\ r_7 & r_8 & r_9 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

As a result, the projective relation in this special case becomes a *3 x 3* matrix. A similarly interesting situation also occurs when the object we observe is a plane. In this specific case, we can assume that the points on this plane will be located at `Z=0`, without the loss of generality. As a result, we obtain the following equation:

$$S\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$$

This zero coordinate of the scene points will then cancel the third column of the projective matrix, which will then again become a *3 x 3* matrix. This special matrix is called a **homography** and it implies that, under special circumstances (here, pure rotation or a planar object), a point is related to its image by a linear relation of the following form:

$$S\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Here, *H* is a *3 x 3* matrix. This relation holds up to a scale factor represented here by the *s* scalar value. Once this matrix is estimated, all the points in one view can be transferred to a second view using this relation. Note that as a side-effect of the homography relation, the fundamental matrix becomes undefined in these cases.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

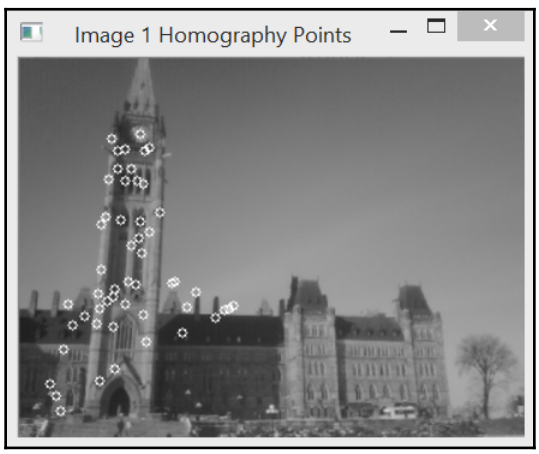
How to do it...

Suppose that we have two images separated by a pure rotation. This happens, for example, when you take pictures of a building or a landscape by rotating yourself; as you are sufficiently far away from your subject, the translational component is negligible. These two images can be matched using the features of your choice and the `cv::BFMatcher` function. Then, as we did in the previous recipe, we will apply a RANSAC step that will, this time, involve the estimation of a homography based on a matching set (which obviously contains a good number of outliers):

1. This is done by using the `cv::findHomography` function, which is very similar to the `cv::findFundamentalMat` function:

```
// Find the homography between image 1 and image 2
std::vector<uchar> inliers(points1.size(),0);
cv::Mat homography= cv::findHomography(
  points1, points2, // corresponding points
  inliers,    // outputed inliers matches
  CV_RANSAC, // RANSAC method
  1.);       // max distance to reprojection point
```

2. You will recall that a homography exists (instead of a fundamental matrix) because our two images are separated by a pure rotation. The images are shown here. We also displayed the inlier keypoints as identified by the `inliers` argument of the function. Refer to the following screenshot:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The second image is shown as follows:

3. The resulting inliers that comply with the found homography have been drawn on these images using the following loop:

```
// Draw the inlier points
std::vector<cv::Point2f>::const_iterator itPts=
                                          points1.begin();
std::vector<uchar>::const_iterator itIn= inliers.begin();
while (itPts!=points1.end()) {

   // draw a circle at each inlier location
   if (*itIn)
       cv::circle(image1,*itPts,3,
                  cv::Scalar(255,255,255));
   ++itPts;
   ++itIn;
}
```

4. The homography is a *3 x 3* invertible matrix; therefore, once it has been computed, you can transfer image points from one image to the other. In fact, you can do this for every pixel of an image. Consequently, you can transfer a complete image to the point of view of a second image. This process is called **image mosaicking**, and it is often used to build a large panorama from multiple images. An OpenCV function that does exactly this is given as follows:

```
// Warp image 1 to image 2
cv::Mat result;
cv::warpPerspective(image1, // input image
   result,         // output image
```

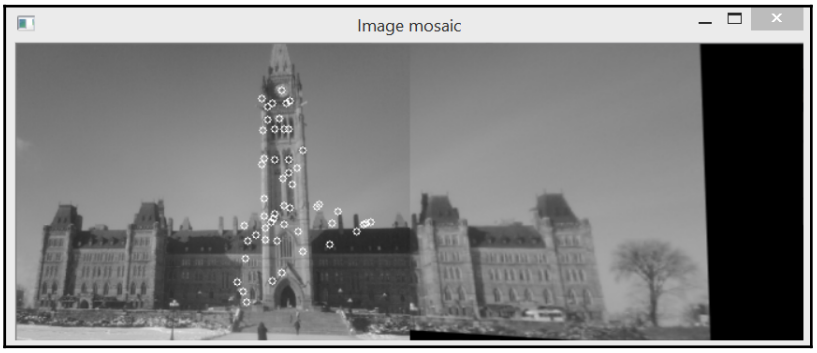

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            homography,      // homography
            cv::Size(2*image1.cols,
                     image1.rows)); // size of output image
```

5. Once this new image is obtained, it can be appended to the other image in order to expand the view (since the two images are now from the same point of view):

```
// Copy image 1 on the first half of full image
cv::Mat half(result,cv::Rect(0,0,image2.cols,image2.rows));
image2.copyTo(half); // copy image2 to image1 roi
```

The following image is the result:



We will now see how this works.

How it works...

When two views are related by a homography, it becomes possible to determine where a given scene point on one image is found on the other image. This property becomes particularly interesting for the points in one image that fall outside the image boundaries of the other. Indeed, since the second view shows a portion of the scene that is not visible in the first image, you can use the homography in order to expand the image by reading the color value of the additional pixels in the other image. That's how we were able to create a new image that is an expansion of our second image in which extra columns were added to the right-hand side.

The homography computed by `cv::findHomography` is the one that maps the points in the first image to the points in the second image. This homography can be computed from a minimum of four matches, and the RANSAC algorithm is used again here. Once the homography with the best support is found, the `cv::findHomography` method refines it using all the identified inliers.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, in order to transfer the points of image `1` to image `2`, what we need is, in fact, inverse homography. This is exactly what the `cv::warpPerspective` function is doing by default; that is, it uses the inverse of the homography provided as the input to get the color value of each point of the output image (this is what we called **backward mapping** in `Chapter 2`, *Manipulating the Pixels*). When an output pixel is transferred to a point outside the input image, a black value (`0`) is simply assigned to this pixel. Note that a `cv::WARP_INVERSE_MAP` flag can be specified as the optional fifth argument in `cv::warpPerspective` if you want to use direct homography instead of the inverted one during the pixel transfer process.

There's more...

A homography also exists between two images of a plane. We can then make use of this to recognize a planar object in an image.

We've successfully learned how to compute a homography between two images. Now, let's move on to the next recipe!

Detecting planar targets in an image

In the previous recipe, we explained how homographies can be used to stitch together images separated by a pure rotation to create a panorama. In this recipe, we also learned that different images of a plane also generate homographies between views. We will now see how we can make use of this fact to recognize a planar object in an image.

How to do it...

Suppose you want to detect the occurrence of a planar object in an image. This object could be a poster, painting, signage, book cover (as in the following example), and so on. Based on what we learned in this chapter, the strategy would consist of detecting feature points on this object and to trying to match them with the feature points in the image. These matches would then be validated using a robust matching scheme similar to the one we used in the previous recipe, but this time based on a homography:

1. Let's define a `TargetMatcher` class very similar to our `RobustMatcher` class:

```
class TargetMatcher {
 private:
   // pointer to the feature point detector object
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    cv::Ptr<cv::FeatureDetector> detector;
    // pointer to the feature descriptor extractor object
    cv::Ptr<cv::DescriptorExtractor> extractor;
    cv::Mat target; // target image
    int normType;
    double distance; // min reprojection error
```

2. Here, we simply add a `target` attribute that represents the reference image of the planar object to be matched. The matching methods are identical to the ones of the `RobustMatcher` class, except that they include `cv::findHomography` instead of `cv::findFundamentalMat` in the `ransacTest` method. We also added a method to initiate target matching and find the position of the target:

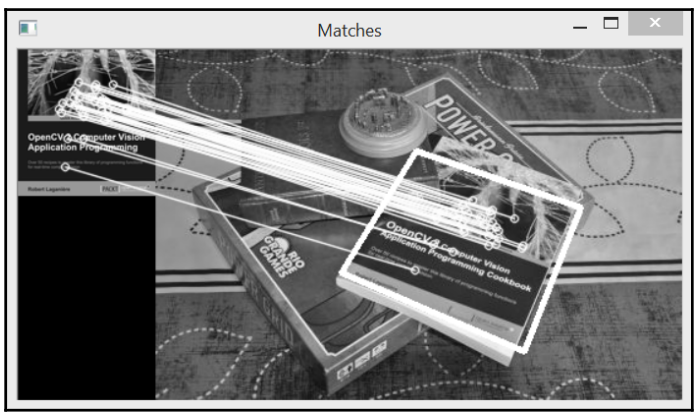
```
  // detect the defined planar target in an image
  // returns the homography
  // the 4 corners of the detected target
  // plus matches and keypoints
  cv::Mat detectTarget(const cv::Mat& image,
    // position of the target corners (clock-wise)
    std::vector<cv::Point2f>& detectedCorners,
    std::vector<cv::DMatch>& matches,
    std::vector<cv::KeyPoint>& keypoints1,
    std::vector<cv::KeyPoint>& keypoints2) {

    // find a RANSAC homography between target and image
      cv::Mat homography= match(target, image, matches, keypoints1,
keypoints2);
    // target corners
    std::vector<cv::Point2f> corners;
    corners.push_back(cv::Point2f(0,0));
    corners.push_back(cv::Point2f(target.cols-1,0));
    corners.push_back(cv::Point2f(target.cols-1,target.rows-1));
    corners.push_back(cv::Point2f(0,target.rows-1));

    // reproject the target corners
    cv::perspectiveTransform(corners,detectedCorners,
                                    homography);
    return homography;
  }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Once the homography has been found by the `match` method, we define the four corners of the target (that is, the four corners of its reference image). These are then transferred to the image using the `cv::perspectiveTransform` function. This function simply multiplies each point in the input vector by the homography matrix. This gives us the coordinates of these points in the other image. Target matching is then performed as follows:

```
// Prepare the matcher
TargetMatcher tmatcher("FAST","FREAK");
tmatcher.setNormType(cv::NORM_HAMMING);

// definition of the output data
std::vector<cv::DMatch> matches;
std::vector<cv::KeyPoint> keypoints1, keypoints2;
std::vector<cv::Point2f> corners;
// the reference image
tmatcher.setTarget(target);
// match image with target
tmatcher.detectTarget(image,corners,matches,
                          keypoints1,keypoints2);
// draw the target corners on the image
cv::Point pt= cv::Point(corners[0]);
cv::line(image,cv::Point(corners[0]),cv::Point(corners[1]),
cv::Scalar(255,255,255),3);
cv::line(image,cv::Point(corners[1]),cv::Point(corners[2]),
cv::Scalar(255,255,255),3);
cv::line(image,cv::Point(corners[2]),cv::Point(corners[3]),
cv::Scalar(255,255,255),3);
cv::line(image,cv::Point(corners[3]),cv::Point(corners[0]),
cv::Scalar(255,255,255),3);
```

Using the `cv::drawMatches` function, we display the results as follows:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

You can also use homographies to modify the perspectives of planar objects. For example, if you have several pictures from different points of view of the flat facade of a building, you can compute the homography between these images and build a large mosaic of the facade by wrapping the images and assembling them together, as we did in this recipe. A minimum of four matched points between two views is required to compute a homography. The `cv::getPerspectiveTransform` function allows such a transformation from four corresponding points to be computed.

See also

- The *Remapping an image* recipe in `Chapter 2`, *Manipulating the Pixels*, discusses the concept of backward mapping.
- The *Automatic Panoramic Image Stitching using Invariant Features* article *by* M. Brown and D. Lowe in *International Journal of Computer Vision, Volume 74, Issue 1*, 2007, describes the complete method to build panoramas from multiple images.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

11
Reconstructing 3D Scenes

In the previous chapter, we learned how a camera captures a 3D scene by projecting light rays on a 2D sensor plane. The image produced is an accurate representation of what the scene looks like from a particular point of view in the instant that the image is captured. However, by its nature, the process of image formation eliminates all of the information concerning the depth of the represented scene elements. This chapter will examine how, under specific conditions, the 3D structure of the scene and the 3D pose of the cameras that captured it can be recovered. We will demonstrate how a good understanding of the concepts of projective geometry allows us to devise methods that enable 3D reconstruction. We will, therefore, revisit the principle of image formation that was introduced in the previous chapter; in particular, we will now take into consideration that our image is composed of pixels.

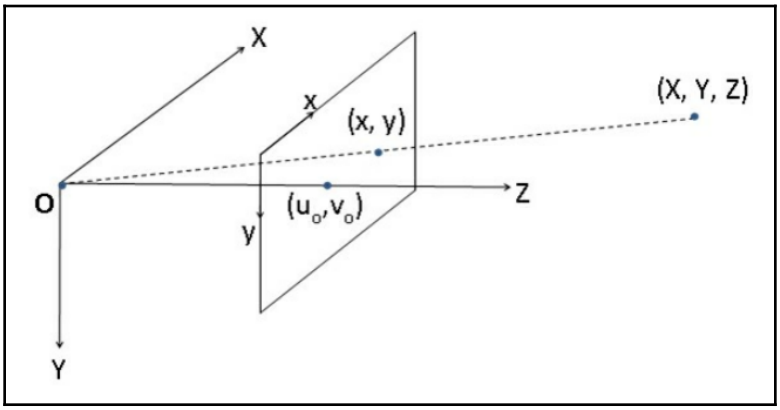
In this chapter, we will cover the following recipes:

- Calibrating a camera
- Recovering the camera pose
- Reconstructing a 3D scene from calibrated cameras
- Computing depth from a stereo image

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Digital image formation

Let's redraw a new version of the diagram shown in `Chapter 10`, *Estimating Projective Relations in Images*, describing the pinhole camera model. More specifically, we want to demonstrate the relationship between a point in 3D at its position *(X, Y, Z)* and its image (*x*, *y*) on a camera, specified in pixel coordinates:



Notice the changes that have been made to the original diagram. First, we added a reference frame that we positioned at the center of the projection. Second, we have the *y* axis pointing downward in order to ensure that the coordinate system is compatible with the usual convention that places the image origin in the upper-left corner of the image. Finally, we also identified a special point on the image plane—considering the line coming from the focal point is orthogonal to the image plane, then the point (*u0*, *v0*) is the pixel position at which this line pierces the image plane. This point is called the **principal point**. It is logical to assume that this principal point is at the center of the image plane, but, in practice, this one might be off by a few pixels, depending on the precision with which the camera has been manufactured.

In `Chapter 10`, *Estimating Projective Relations in Images*, we learned that the essential parameters of a camera in the pinhole model are its focal length and the size of the image plane (which defines the field of view of the camera). In addition, since we are dealing with digital images, the number of pixels on the image plane (that is, its resolution) is another important characteristic of a camera. We also previously learned that a 3D point (*X*, *Y*, *Z*) will be projected onto the image plane at (*fX*/*Z*, *fY*/*Z*).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, if we want to translate this coordinate into pixels, we need to divide the 2D image position by the pixel width (*px*) and pixel height (*py*), respectively. We notice that by dividing the focal length given in world units (which is generally given in millimeters) by *px*, we obtain the focal length expressed in (horizontal) pixels. Let's define this term, then, as *fx*. Similarly, *fy = f/py* is defined as the focal length expressed in vertical pixel units. The complete projective equation is, therefore, as follows:

$$x = \frac{f_x X}{Z} + u_0$$

$$y = \frac{f_y Y}{Z} + v_0$$

Recall that (*u0*, *v0*) is the principal point that is added to the result in order to move the origin to the upper-left corner of the image. Note also that the physical size of a pixel can be obtained by dividing the size of the image sensor (which is generally in millimeters) by the number of pixels (horizontally or vertically). In modern sensors, pixels are generally square, that is, they have the same horizontal and vertical size.

The preceding equations can be rewritten in matrix form as we did in `Chapter 10`, *Estimating Projective Relations in Images*. Here is the complete projective equation in its most general form:

$$s \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r1 & r2 & r3 & t1 \\ r4 & r5 & r6 & t2 \\ r7 & r8 & r9 & t3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

The next recipe shows how to calibrate a camera.

Calibrating a camera

Camera calibration is the process by which the different camera parameters are obtained. You can obviously use the specifications provided by the camera manufacturer, but for some tasks, such as 3D reconstruction, these specifications are not accurate enough. Camera calibration works by showing known patterns to the camera and analyzing the obtained images. An optimization process will then determine the optimal parameter values that explain the observations. This is a complex process that has been made easy by the availability of OpenCV's calibration functions.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

To calibrate a camera, you show it a set of scene points where the 3D positions are known. Then, you need to observe where these points project on the image. With the knowledge of a sufficient number of 3D points and associated 2D image points, the exact camera parameters can be inferred from the projective equation. Obviously, for accurate results, we need to observe as many points as possible. One way to achieve this is to take one picture of a scene with many known 3D points, but, in practice, this is rarely feasible. A more convenient way is to take several images of a set of 3D points from different viewpoints. This approach is simpler, but it requires you to compute the position of each camera view in addition to the computation of the internal camera parameters, which, fortunately, is feasible.

OpenCV proposes that you use a chessboard pattern to generate the set of 3D scene points that are required for calibration. This pattern creates points in the corners of each square, and since this pattern is flat, we can freely assume that the board is located at $Z = 0$, with the x and y axes well-aligned according to the grid. In this case, the calibration process simply consists of showing the chessboard pattern to the camera from different viewpoints. Here is one example of a 6×4 calibration pattern image:

The good thing is that OpenCV has a function that automatically detects the corners of this chessboard pattern.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

We are going to use a number of OpenCV algorithms that detect the inner points of a chessboard and compute the camera calibration; to do this, perform the following steps to calibrate the camera:

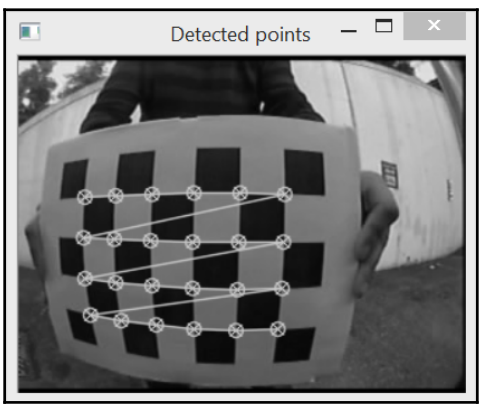
1. You simply provide an image and the size of the chessboard used (that is, the number of horizontal and vertical inner corner points). The function will return the position of these chessboard corners on the image. If the function fails to find the pattern, then it simply returns `false`:

```
// output vectors of image points
std::vector<cv::Point2f> imageCorners;
// number of inner corners on the chessboard
cv::Size boardSize(6,4);
// Get the chessboard corners
bool found = cv::findChessboardCorners(image,
                                boardSize, imageCorners);
```

2. The output parameter, `imageCorners`, will simply contain the pixel coordinates of the detected inner corners of the shown pattern. Note that this function accepts additional parameters if you need to fine-tune the algorithm, which is not discussed here. There is also a special function that draws the detected corners on the chessboard image, with lines connecting them in a sequence:

```
//Draw the corners
cv::drawChessboardCorners(image,
                boardSize, imageCorners,
                found); // corners have been found
```

The following image is obtained:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The lines that connect the points show the order in which the points are listed in the vector of detected image points. To perform a calibration, we now need to specify the corresponding 3D points.

3. You can specify these points in the units of your choice (for example, in centimeters or in inches); however, the simplest way to do this is to assume that each square represents one unit. In this case, the coordinate of the first point will be (*0*, *0*, *0*)—assuming that the board is located at a depth of *Z = 0*, then, the second point will be *(1*, *0*, *0*), and so on, until the last point is at (*5*, *3*, *0*). There are a total of *24* points in this pattern, which are not enough to obtain an accurate calibration.
4. To get more points, you need to show more images of the same calibration pattern from various points of view. To do so, you can either move the pattern in front of the camera or move the camera around the board; from a mathematical point of view, this completely equivalent. The OpenCV calibration function assumes that the reference frame is fixed on the calibration pattern and will calculate the rotation and translation of the camera with respect to the reference frame. Let's now encapsulate the calibration process in a `CameraCalibrator` class. The attributes of this class are as follows:

```
class CameraCalibrator {

    // input points:
    // the points in world coordinates
    std::vector<std::vector<cv::Point3f>> objectPoints;
    // the point positions in pixels
    std::vector<std::vector<cv::Point2f>> imagePoints;
    // output Matrices
    cv::Mat cameraMatrix;
    cv::Mat distCoeffs;
   // flag to specify how calibration is done
   int flag;
```

5. Note that the input vectors of the scene and image points are, in fact, `std::vector` classes of `std::vector` classes; each vector element is a vector of the points from one view. Here, we decided to add the calibration points by specifying a vector of the chessboard image filename as the input of our `addChessBoardPoints` class function:

```
// Open chessboard images and extract corner points
int CameraCalibrator::addChessboardPoints(
         const std::vector<std::string>& filelist,
         cv::Size & boardSize) {
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Then, in this function, we first have to initialize the vectors that we mention and set up the 3D scene points of our chessboard:

```
// the points on the chessboard
 std::vector<cv::Point2f> imageCorners;
 std::vector<cv::Point3f> objectCorners;

// 3D Scene Points:
// Initialize the chessboard corners
// in the chessboard reference frame
// The corners are at 3D location (X,Y,Z)= (i,j,0)
for (int i=0; i<boardSize.height; i++) {
   for (int j=0; j<boardSize.width; j++) {
      objectCorners.push_back(cv::Point3f(i, j, 0.0f));
   }
 }
```

7. Now we have to read each image of the input list and find the corners of the chessboard using the `cv::findChessboardCorners` function:

```
 // 2D Image points:
 cv::Mat image; // to contain chessboard image
 int successes = 0;
 // for all viewpoints
 for (int i=0; i<filelist.size(); i++) {
     // Open the image
     image = cv::imread(filelist[i],0);
     // Get the chessboard corners
     bool found = cv::findChessboardCorners(
                     image, boardSize, imageCorners);
```

8. Moreover, in order to obtain a more accurate image point location, the `cv::cornerSubPix` function can be used and, as the name suggests, the image points will then be localized at a subpixel level of accuracy. The termination criterion that is specified by the `cv::TermCriteria` object defines the maximum number of iterations and the minimum level of accuracy in the subpixel coordinates. The first of these two conditions that are reached will stop the corner refinement process:

```
     // Get subpixel accuracy on the corners
     cv::cornerSubPix(image, imageCorners,
               cv::Size(5,5),
               cv::Size(-1,-1),
      cv::TermCriteria(cv::TermCriteria::MAX_ITER +
                       cv::TermCriteria::EPS,
          30,      // max number of iterations
          0.1));  // min accuracy
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            //If we have a good board, add it to our data
            if (imageCorners.size() == boardSize.area()) {
                // Add image and scene points from one view
                addPoints(imageCorners, objectCorners);
                successes++;
            }
        }
       return successes;
    }
```

9. When a set of chessboard corners have been successfully detected, these points are then added to our vectors of the image and scene points using the `addPoints` method. Once a sufficient number of chessboard images have been processed (and consequently, a large number of 3D scene point or 2D image point correspondences are available), we can initiate the computation of the calibration parameters as follows:

```
// Calibrate the camera
// returns the re-projection error
double CameraCalibrator::calibrate(cv::Size &imageSize)
{
   //Output rotations and translations
    std::vector<cv::Mat> rvecs, tvecs;

   // start calibration
   return
     calibrateCamera(objectPoints, // the 3D points
                imagePoints,  // the image points
                imageSize,    // image size
                cameraMatrix, // output camera matrix
                distCoeffs,   // output distortion matrix
                rvecs, tvecs, // Rs, Ts
                flag);        // set options
}
```

In practice, 10 to 20 chessboard images are sufficient, but these must be taken from different viewpoints at different depths. The two important outputs of this function are the camera matrix and the distortion parameters. These will be described in the next section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

In order to explain the results of the calibration, we need to go back to the projective equation presented in the introduction to this chapter. This equation describes the transformation of a 3D point into a 2D point through the successive application of two matrices. The first matrix includes all of the camera parameters, which are called the intrinsic parameters of the camera. The *3* x *3* matrix is one of the output matrices returned by the `cv::calibrateCamera` function. There is also a function called `cv::calibrationMatrixValues` that explicitly returns the value of the intrinsic parameters given by a calibration matrix.

The second matrix is there to express the input points as camera-centric coordinates. It is composed of a rotation vector (a *3* x *3* matrix) and a translation vector (a *3* x *1* matrix). Remember that in our calibration example, the reference frame was placed on the chessboard. Therefore, there is a rigid transformation (made of a rotation component represented by the matrix entries from `r1` to `r9` and a translation component represented by `t1`, `t2`, and `t3`) that must be computed for each view. These are in the output parameter list of the `cv::calibrateCamera` function. The rotation and translation components are often called the **extrinsic parameters** of the calibration, and they are different for each view. The intrinsic parameters remain constant for a given camera or lens system.

The calibration results provided by `cv::calibrateCamera` are obtained through an optimization process. This process aims to find the intrinsic and extrinsic parameters that minimize the difference between the predicted image point position, as computed from the projection of the 3D scene points, and the actual image point position, as observed on the image. The sum of this difference for all the points specified during the calibration is called the **reprojection error**.

The intrinsic parameters of our test camera obtained from a calibration based on *27* chessboard images are `fx=409` pixels, `fy=408` pixels, `u0=237` pixels, and `v0=171` pixels. Our calibration images have a size of *536* x *356* pixels. From the calibration results, you can see that, as expected, the principal point is close to the center of the image, yet it is off by a few pixels. The calibration images were taken using a Nikon D500 camera with an 18 mm lens. Looking at the manufacturer specifications, we find that the sensor size of this camera is 23.5 mm x 15.7 mm, which gives us a pixel size of 0.0438 mm. The estimated focal length is expressed in pixels, so multiplying the result by the pixel size gives us an estimated focal length of 17.8 mm, which is consistent with the actual lens that we used.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

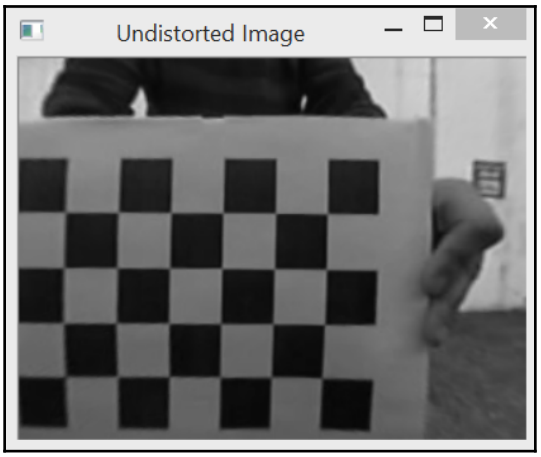
Let's now turn our attention to the distortion parameters. So far, we have mentioned that in the pinhole camera model, we can neglect the effect of the lens. However, this is only possible if the lens that is used to capture an image does not introduce important optical distortions. Unfortunately, this is not the case with lenses of a lower quality or lenses that have a very short focal length. You may have already noticed that the chessboard pattern shown in the image we used for our example is clearly distorted—the edges of the rectangular board are curved in the image. Also, note that this distortion becomes more significant as we move away from the center of the image. This is a typical distortion that is observed with a fish-eye lens, and it is called **radial distortion**.

It is possible to compensate for these deformations by introducing an appropriate distortion model. The idea is to represent the distortions, introduced by a lens, by using a set of mathematical equations. Once established, these equations can then be reverted in order to undo the distortions that are visible on the image. Fortunately, the exact parameters of the transformation that will correct the distortions can be obtained with the other camera parameter during the calibration phase. Once this is done, any image from the newly-calibrated camera will be undistorted. Therefore, we have added an additional method to our calibration class:

```
// remove distortion in an image (after calibration)
cv::Mat CameraCalibrator::remap(const cv::Mat &image) {

   cv::Mat undistorted;

   if (mustInitUndistort) { // called once per calibration
    cv::initUndistortRectifyMap(
      cameraMatrix,  // computed camera matrix
      distCoeffs,    // computed distortion matrix
      cv::Mat(),     // optional rectification (none)
      cv::Mat(),     // camera matrix to generate undistorted
            image.size(),  // size of undistorted
            CV_32FC1,      // type of output map
            map1, map2);   // the x and y mapping functions

    mustInitUndistort= false;
   }

   // Apply mapping functions
   cv::remap(image, undistorted, map1, map2,
      cv::INTER_LINEAR); // interpolation type

   return undistorted;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Running this code results in the following screenshot of the image:



As you can see, once the image is undistorted, we obtain a regular perspective image.

To correct the distortion, OpenCV uses a polynomial function that is applied to the image point in order to move them at their undistorted position. By default, five coefficients are used; a model made of eight coefficients is also available. Once these coefficients are obtained, it is possible to compute two `cv::Mat` mapping functions (one for the *x* coordinate, and one for the *y* coordinate), which will give the new undistorted position of an image point on a distorted image. This is computed by the `cv::initUndistortRectifyMap` function; the `cv::remap` function remaps all the points of an input image to a new image. Note that because of the nonlinear transformation, some pixels of the input image now fall outside the boundary of the output image. You can expand the size of the output image to compensate for this loss of pixels, but you will now obtain output pixels that have no values in the input image (they will then be displayed as black pixels).

There's more...

There are more options that are available when it comes to camera calibration. Let's take a look at them in the next section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Calibration with known intrinsic parameters

When a good estimate of the camera's intrinsic parameters is known, it could be advantageous to input them into the `cv::calibrateCamera` function. They will then be used as initial values in the optimization process. To do so, you just need to add the `CALIB_USE_INTRINSIC_GUESS` flag and input these values in the calibration matrix parameter. It is also possible to impose a fixed value for the principal point (`CALIB_FIX_PRINCIPAL_POINT`), which can often be assumed to be the central pixel. You can also impose a fixed ratio for the focal lengths of `fx` and `fy` (`CALIB_FIX_RATIO`)—in this case, you can assume the pixels to be square-shaped.

Using a grid of circles for calibration

Instead of the usual chessboard pattern, OpenCV also offers the possibility to calibrate a camera by using a grid of circles. In this case, the centers of the circles are used as calibration points. The corresponding function is very similar to the function that we used to locate the chessboard corners:

```
      cv::Size boardSize(7,7);
std::vector<cv::Point2f> centers;
      bool found = cv:: findCirclesGrid(
                          image, boardSize, centers);
```

See also

- The *A Flexible New Technique for Camera Calibration* article by Z. Zhang in *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 22, No. 11*, 2000, is a classic paper on the problem of camera calibration.

We've successfully learned how to calibrate a camera. Now let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Recovering the camera pose

When a camera is calibrated, it becomes possible to relate the captured images to the outside world. We previously explained that if the 3D structure of an object is known, then you can predict how the object will be projected on to the sensor of the camera. The process of image formation is described by the projective equation presented at the beginning of this chapter. When most of the terms of this equation are known, then it becomes possible to infer the value of the other elements (2D or 3D) through the observation of some images. In this recipe, we will look at the camera pose recovery problem when a known 3D structure is observed.

How to do it...

Let's consider a simple object—a bench in a park. We took an image of the following bench using the camera and lens system that we calibrated in the previous recipe. We have also manually identified eight distinct image points on the bench that we will use for our camera pose estimation:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

By having access to this object, it is possible to take a number of physical measurements. The bench is composed of a seat that is 242.5 cm x 53.5 cm x 9 cm and a back that is 242.5 cm x 24 cm x 9 cm, which is fixed 12 cm over the seat. Using this information, we can easily derive the 3D coordinates of the eight identified points in an object-centric reference frame (here, we fixed the origin at the left extremity of the intersection between the two planes):

1. So, the first thing we have to do in our algorithm is to create a `cv::Point3f` vector containing these coordinates:

```
// Input object points
std::vector<cv::Point3f> objectPoints;
objectPoints.push_back(cv::Point3f(0, 45, 0));
objectPoints.push_back(cv::Point3f(242.5, 45, 0));
objectPoints.push_back(cv::Point3f(242.5, 21, 0));
objectPoints.push_back(cv::Point3f(0, 21, 0));
objectPoints.push_back(cv::Point3f(0, 9, -9));
objectPoints.push_back(cv::Point3f(242.5, 9,
-9));objectPoints.push_back(cv::Point3f(242.5, 9, 44.5));
objectPoints.push_back(cv::Point3f(0, 9, 44.5));
```

2. The question now is where the camera was with respect to these points when the preceding picture was taken. Since the coordinates of the image of these known points on the 2D image plane are also known, then it becomes easy to answer this question by using the `cv::solvePnP` function. Here, the correspondence between the 3D and the 2D points has been established manually, but you should be able to come up with methods that will allow you to obtain this information automatically:

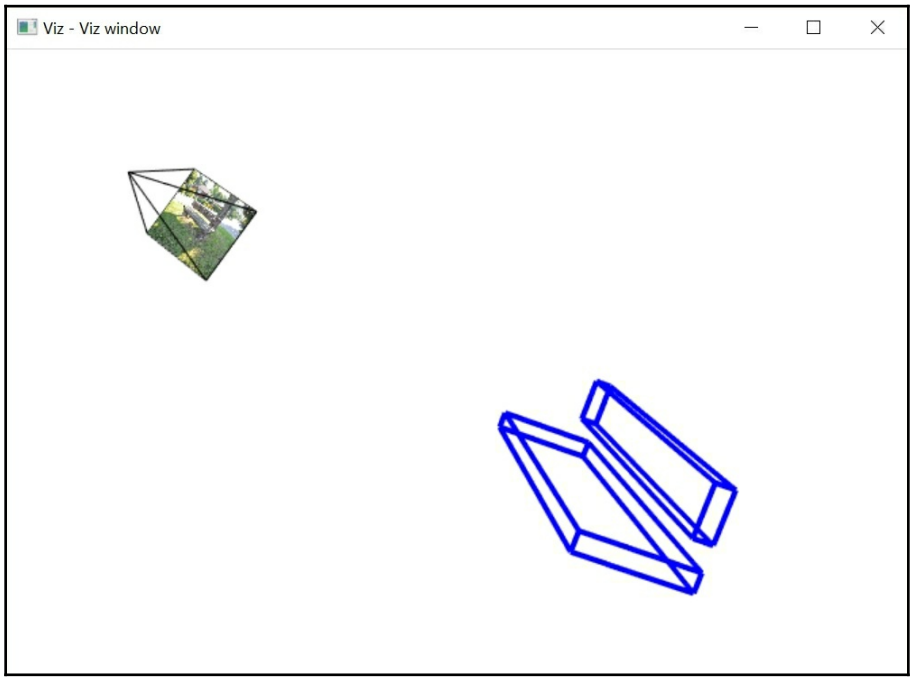
```
// Input image points
 std::vector<cv::Point2f> imagePoints;
 imagePoints.push_back(cv::Point2f(136, 113));
 imagePoints.push_back(cv::Point2f(379, 114));
 imagePoints.push_back(cv::Point2f(379, 150));
 imagePoints.push_back(cv::Point2f(138, 135));
 imagePoints.push_back(cv::Point2f(143, 146));
 imagePoints.push_back(cv::Point2f(381, 166));
 imagePoints.push_back(cv::Point2f(345, 194));
 imagePoints.push_back(cv::Point2f(103, 161));

// Get the camera pose from 3D/2D points
cv::Mat rvec, tvec;
cv::solvePnP(objectPoints, imagePoints, cameraMatrix,
cameraDistCoeffs,  rvec, tvec);

cv::Mat rotation;
// convert vector-3 rotation
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
// to a 3x3 rotation matrix
cv::Rodrigues(rvec, rotation);
```

This function, in fact, computes the rigid transformation (that is, the rotation and translation components) that brings the object coordinates in to the camera-centric reference frame (that is, the one that has its origin at the focal point). It is also important to note that the rotation computed by this function is given in the form of a 3D vector. This is a compact representation in which the rotation that is to be applied is described by a unit vector (that is, an axis of rotation) around which the object is rotated by a certain angle. This axis-angle representation is also called **Rodrigues' rotation formula.** In OpenCV, the angle of the rotation corresponds to the norm of the output rotation vector, the latter being aligned with the axis of rotation. This is why the `cv::Rodrigues` function is used to obtain the 3D matrix of rotation that appears in our projective equation.

3. The pose recovery procedure described here is simple, but how do we know that we obtained the right camera and object pose information? Well, we can visually assess the quality of the results by using the `cv::viz` module, which gives us the ability to visualize 3D information. The use of this module is explained in the last section of this recipe, but let's display a simple 3D representation of our object and the camera that captured it:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

It might be difficult to judge the quality of the pose recovery just by looking at this image, but if you test the example of this recipe on your computer, then you will have the option of moving this representation in 3D using your mouse, which should give you a better sense of the solution obtained.

How it works...

In this recipe, we assumed that the 3D structure of the object was known, as well as the correspondence between the sets of object points and image points. The camera's intrinsic parameters were also known through calibration. If you look at our projective equation, presented at the end of the *Digital image formation* section in the introduction to this chapter, this means that we have points in which the coordinates of (X, Y, Z) and (x, y) are known. We also know the elements of the first matrix (that is, the intrinsic parameters). Only the second matrix is unknown; this is the one that contains the extrinsic parameters of the camera (that is, the camera/object pose information). Our objective, then, is to recover these unknown parameters from the observation of 3D scene points. This problem is known as the **Perspective-n-Point** (**PnP**) problem.

The rotation has three degrees of freedom (the angles of rotation around the three axes), as does the translation. We, therefore, have a total of six unknowns. For each object point and image point correspondence, the projective equation gives us three algebraic equations, but since the projective equation is up to a scale factor, we only have two independent equations. A minimum of three points is, therefore, required to solve this system of equations. Obviously, more points provide a more reliable estimate.

In practice, many different algorithms have been proposed to solve this problem and OpenCV proposes a number of different implementations in its `cv::solvePnP` function. The default method consists of optimizing what is called the **reprojection error**. Minimizing this type of error is considered to be the best strategy for getting accurate 3D information from camera images. In our problem, it corresponds to finding the optimal camera position that minimizes the 2D distance between the projected 3D points (as obtained by applying the projective equation) and the observed image points given as input.

Note that OpenCV also has a `cv::solvePnPRansac` function. As the name suggests, this function uses the **RANSAC** algorithm in order to solve the PnP problem. This means that some of the object point and image point correspondences may be wrong and the function will return the ones that have been identified as outliers. This is very useful when these correspondences have been obtained through an automatic process that can fail for some points.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

There's more...

When working with 3D information, it is often difficult to validate the solutions obtained. To this end, OpenCV offers a simple, yet powerful, visualization module that facilitates the development and debugging of 3D vision algorithms. It allows inserting points, lines, cameras, and other objects in a virtual 3D environment that you can interactively visualize from various points of views.

cv::Viz – a 3D visualizer module

`cv::Viz` is an extra module of the OpenCV library that is built on top of the **Visualization Toolkit** (**VTK**) open source library. This is a powerful framework used for 3D computer graphics. With `cv::viz`, you can create a 3D virtual environment in which you can add a variety of objects. A visualization window is created that displays the environment from a given point of view. You saw, in this recipe, an example of what can be displayed in a `cv::viz` window. This window responds to mouse events that are used to navigate inside the environment (through rotations and translations). This section describes the basic use of the `cv::viz` module.

The first thing to do is to create the visualization window. Here, we use a white background, as follows:

```
// Create a viz window
cv::viz::Viz3d visualizer("Viz window");
visualizer.setBackgroundColor(cv::viz::Color::white());
```

Next, you create your virtual objects and insert them into the scene. There are a variety of predefined objects. One of them is particularly useful for us; it is the one that creates a virtual pin-hole camera:

```
// Create a virtual camera
 cv::viz::WCameraPosition cam(cMatrix, // matrix of intrinsics
 image, // image displayed on the plane
 30.0, // scale factor
 cv::viz::Color::black());
// Add the virtual camera to the environment
visualizer.showWidget("Camera", cam);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The `cMatrix` variable is the `cv::Matx33d` (that is, `cv::Matx<double,3,3>`) instance containing the intrinsic camera parameters that were obtained from the calibration. By default, this camera is inserted at the origin of the coordinate system. To represent the bench, we used two rectangular cuboid objects, as follows:

```
// Create a virtual bench from cuboids
 cv::viz::WCube plane1(cv::Point3f(0.0, 45.0, 0.0),
 cv::Point3f(242.5, 21.0, -9.0),
 true, // show wire frame
 cv::viz::Color::blue());
 plane1.setRenderingProperty(cv::viz::LINE_WIDTH, 4.0);
 cv::viz::WCube plane2(cv::Point3f(0.0, 9.0, -9.0),
 cv::Point3f(242.5, 0.0, 44.5),
 true, // show wire frame
 cv::viz::Color::blue());
 plane2.setRenderingProperty(cv::viz::LINE_WIDTH, 4.0);
 // Add the virtual objects to the environment
 visualizer.showWidget("top", plane1);
 visualizer.showWidget("bottom", plane2);
```

This virtual bench is also added at the origin; it then needs to be moved to its camera-centric position, which was obtained in the `cv::solvePnP` function. It is the responsibility of the `setWidgetPose` method to perform this operation. This one simply applies the rotation and translation components of the estimated motion:

```
cv::Mat rotation;
 // convert vector-3 rotation
 // to a 3x3 rotation matrix
 cv::Rodrigues(rvec, rotation);

// Move the bench
 cv::Affine3d pose(rotation, tvec);
 visualizer.setWidgetPose("top", pose);
 visualizer.setWidgetPose("bottom", pose);
```

The final step is to create a loop that keeps displaying the visualization window; the `1` ms pause is included in order to listen to mouse events:

```
// visualization loop
 while(cv::waitKey(100)==-1 && !visualizer.wasStopped())
 {
    visualizer.spinOnce(1,  true); // redraw
 }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This loop will stop when the visualization window is closed or when a key is pressed over an OpenCV image window. Try applying motion to an object (using `setWidgetPose`) inside this loop; this is how animation can be created.

See also

- *Model-Based Object Pose in 25 Lines of Code* by D. DeMenthon and L. S. Davis, in *European Conference on Computer Vision*, 1992, pp.335-343, is a famous method for recovering the camera pose from scene points.
- The *Matching images using random sample consensus* recipe in `Chapter 10`, *Estimating Projective Relations in Images* describes the RANSAC algorithm.
- The *Installing the OpenCV library* recipe in `Chapter 1`, *Playing with Images* explains how to install the RANSAC `cv::viz` extra module.

We've successfully learned how to recover a camera pose. Now let's move on to the next recipe!

Reconstructing a 3D scene from calibrated cameras

In the previous recipe, we saw that it is possible to recover the position of a camera that is observing a 3D scene when the camera is calibrated. The approach that was described took advantage of the fact that, sometimes, the coordinates of some 3D points visible in the scene might be known. We will now learn that if a scene is observed from more than one point of view, a 3D pose and structure can be reconstructed even if no information about the 3D scene is available. This time, we will use correspondences between image points in the different views in order to infer 3D information. We will introduce a new mathematical entity encompassing the relationship between two views of a calibrated camera, and we will discuss the principle of triangulation in order to reconstruct 3D points from 2D images.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Let's again use the camera that we calibrated in the first recipe of this chapter and take two pictures of a scene. We can match the feature points between these two views using the **SIFT** (short for **Scale Invariant Feature Transform**) detector and descriptor presented in `Chapter 8`, *Detecting Interest Points*, and `Chapter 9`, *Describing and Matching Interest Points*.

The fact that the calibration parameters of the camera are available allows us to work in world coordinates and, therefore, establish a physical constraint between the camera poses and the position of the corresponding points. Essentially, we introduce a new mathematical entity called the **essential matrix**, which is the calibrated version of the fundamental matrix that was introduced in the previous chapter. Therefore, there is a `cv::findEssentialMat` function that's identical to the `cv::findFundametalMat` function that was used in the *Computing the fundamental matrix of an image pair* recipe in `Chapter 10`, *Estimating Projective Relations in Images*.

We can call the `cv::findEssentialMat` function with the established point correspondences and, through a RANSAC scheme, filter out the outlier points to retain only the matches that comply with the found geometry. The following steps will help us in reconstructing a 3D image:

1. First, we have to get a large number of keypoints in each image using the SIFT detector and descriptor:

```
    // vector of keypoints and descriptors
    std::vector<cv::KeyPoint> keypoints1;
    std::vector<cv::KeyPoint> keypoints2;
    cv::Mat descriptors1, descriptors2;

    // Construction of the SIFT feature detector
    cv::Ptr<cv::Feature2D> ptrFeature2D =
                cv::xfeatures2d::SIFT::create(500);

    // Detection of the SIFT features and associated descriptors
    ptrFeature2D->detectAndCompute(image1, cv::noArray(),
keypoints1, descriptors1);
    ptrFeature2D->detectAndCompute(image2, cv::noArray(),
keypoints2, descriptors2);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Using `cv::BFMatcher`, match the keypoints that are detected using their descriptors and store them as float points:

```
// Match the two image descriptors
// Construction of the matcher with crosscheck
cv::BFMatcher matcher(cv::NORM_L2, true);
std::vector<cv::DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);

// Convert keypoints into Point2f
std::vector<cv::Point2f> points1, points2;
for (std::vector<cv::DMatch>::const_iterator it =
       matches.begin(); it != matches.end(); ++it)
  {

  // Get the position of left keypoints
  float x = keypoints1[it->queryIdx].pt.x;
  float y = keypoints1[it->queryIdx].pt.y;
  points1.push_back(cv::Point2f(x, y));
  // Get the position of right keypoints
  x = keypoints2[it->trainIdx].pt.x;
  y = keypoints2[it->trainIdx].pt.y;
  points2.push_back(cv::Point2f(x, y));
}
```

3. Using the `cv::findEssentialMat` function, we can obtain the camera matrix with the image keypoints that matches in both images:

```
// Find the essential between image 1 and image 2
cv::Mat inliers;
cv::Mat essential = cv::findEssentialMat(points1,
     points2,
     Matrix,      // intrinsic parameters
     cv::RANSAC, 0.9, 1.0, // RANSAC method
     inliers);    // extracted inliers
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

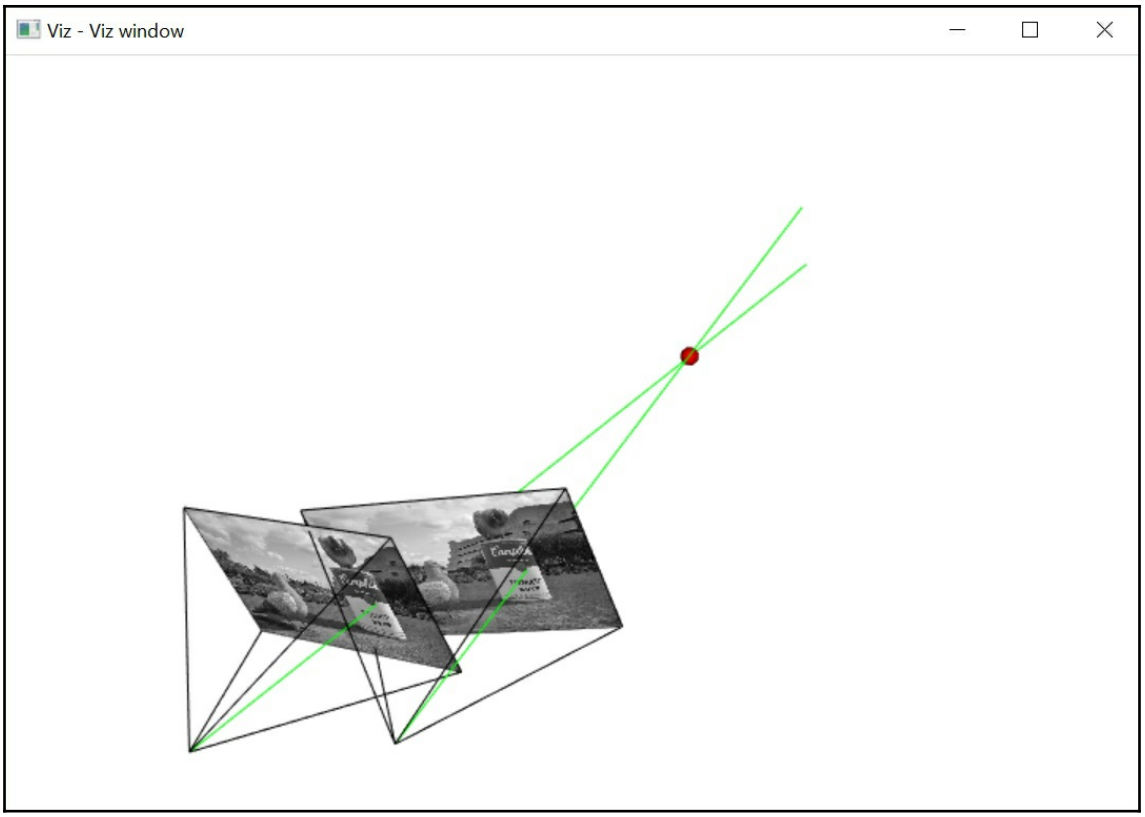
The resulting set of **Inliers matches** is as follows:

4. As will be explained in the next section, the essential matrix encapsulates the rotation and translation components that separate the two views. It is, therefore, possible to recover the relative pose between our two views directly from this matrix. OpenCV has a function that performs this operation; it is the `cv::recoverPose` function, and is used as follows:

```
// recover relative camera pose from essential matrix
cv::Mat rotation, translation;
cv::recoverPose(essential, // the essential matrix
         points1, points2,  // the matched keypoints
         cameraMatrix,      // matrix of intrinsics
         rotation, translation,  // estimated motion
         inliers);          //inliers matches
```

5. Now that we have the relative pose between the two cameras, it is possible to estimate the location of points for which we established a correspondence between the two views. The following screenshot illustrates how this is possible. It shows the two cameras at their estimated positions (the left one is placed at the origin). We also selected a pair of corresponding points and, for these image points, we traced a ray that, according to the projective geometry model, corresponds to all possible locations of the associated 3D point:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Clearly, since these two image points have been generated by the same 3D point, the two rays must intersect at one location, that is, the location of the 3D point. The method that consists of intersecting the lines of projection of two corresponding image points when the relative position of two cameras is known is called **triangulation**. This process first requires the two projection matrices and can be repeated for all matches. Remember, however, that these ones must be expressed in world coordinates; this is done here by using the `cv::undistortPoints` function.

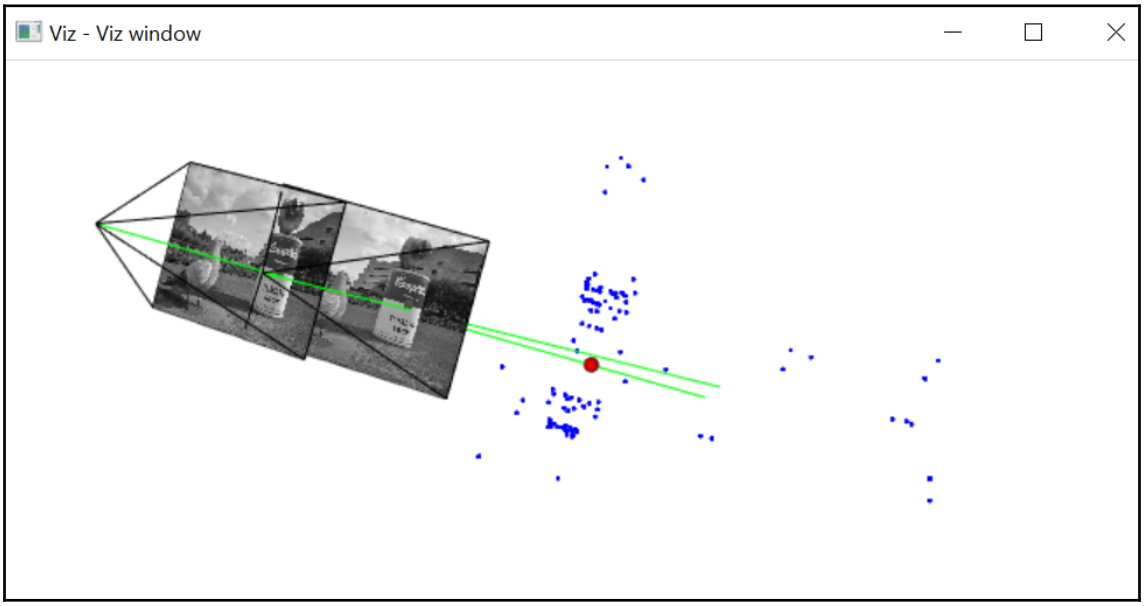
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Finally, we call our triangulate function, which computes the position of the triangulated point (which will be described in the next section):

```
  // compose projection matrix from R,T
  cv::Mat projection2(3, 4, CV_64F); // the 3x4 projection matrix
  rotation.copyTo(projection2(cv::Rect(0, 0, 3, 3)));
  translation.copyTo(projection2.colRange(3, 4));

  // compose generic projection matrix
  cv::Mat projection1(3, 4, CV_64F, 0.);// the 3x4 projection
matrix
  cv::Mat diag(cv::Mat::eye(3, 3, CV_64F));
  diag.copyTo(projection1(cv::Rect(0, 0, 3, 3)));

  // to contain the inliers
  std::vector<cv::Vec2d> inlierPts1;
  std::vector<cv::Vec2d> inlierPts2;

  // create inliers input point vector for triangulation
  int j(0);
  for (int i = 0; i < inliers.rows; i++) {
    if (inliers.at<uchar>(i)) {
       inlierPts1.push_back(cv::Vec2d(points1[i].x, points1[i].y));
       inlierPts2.push_back(cv::Vec2d(points2[i].x, points2[i].y));
    }
  }

  // undistort and normalize the image points
  std::vector<cv::Vec2d> points1u;
  cv::undistortPoints(inlierPts1, points1u,
                      cameraMatrix, cameraDistCoeffs);
  std::vector<cv::Vec2d> points2u;
  cv::undistortPoints(inlierPts2, points2u,
                      cameraMatrix, cameraDistCoeffs);

  // triangulation
  std::vector<cv::Vec3d> points3D;
  triangulate(projection1, projection2,
              points1u, points2u, points3D);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. A cloud of 3D points located on the surface of the scene elements is found, as follows:

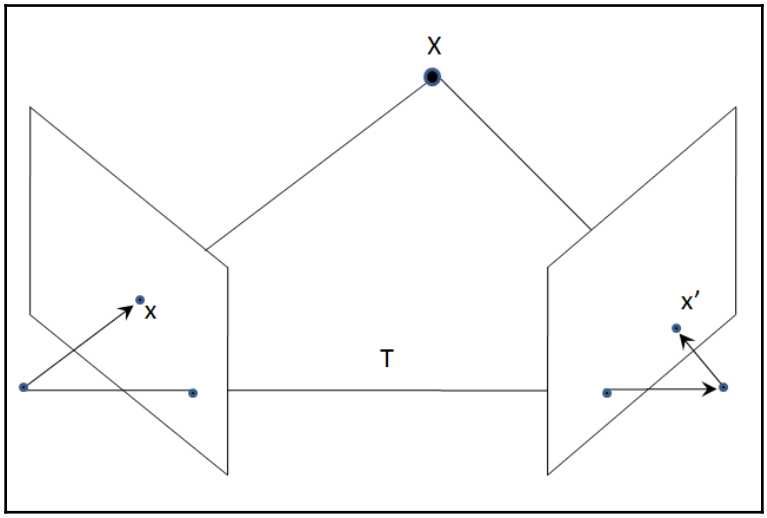


Note that from this new point of view, we can see that the two rays we drew do not intersect as they were supposed to. This fact will be discussed in the next section.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

The calibration matrix is an entity that allows us to transform pixel coordinates into world coordinates. By doing so, we can more easily relate image points to the 3D points that have produced them. This is demonstrated in the following diagram, which we will now use to demonstrate the relationship between a world point and its images:



The preceding diagram shows two cameras separated by a rotation of **R** and a translation of **T**. It is interesting to note that the translation vector, **T**, joins the centers of the projection of the two cameras. We also have a vector of **x** joining the first camera center to an image point and a vector of **x'** joining the second camera center to the corresponding image point. Since we have the relative motion between the two cameras, we can express the orientation of **x**, in terms of the second camera reference, as `Rx`. Now, if you carefully observe the geometry of the image points shown, you will observe that the **T**, **Rx**, and **x'** vectors are all coplanar. This fact can be expressed by the following mathematical equation:

$$x'.(T \times Rx) = x'Ex = 0$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

It was possible to reduce the first relation to a single *3* x *3* matrix of *E* because a cross-product can also be expressed by a matrix operation. This *E* matrix is called the essential matrix, and the associated equation is the calibrated equivalent of the epipolar constraint presented in the *Computing the fundamental matrix of an image pair* recipe in `Chapter 10`, *Estimating Projective Relations in Images*. We can then estimate this essential matrix from image correspondences, as we did for the fundamental matrix, but, this time, expressing them in world coordinates. Additionally, as demonstrated, the essential matrix is built from the rotation and translation components of the motion between the two cameras. This means that once this matrix has been estimated, it can be decomposed to obtain the relative pose between the cameras. This is what we did by using the `cv::recoverPose` function. This function calls the `cv::decomposeEssentialMat` function, which produces four possible solutions for the relative pose. The right one is identified by looking at the set of provided matches to determine the solution that is physically possible.

Once the relative pose between the cameras has been obtained, the position of any point corresponding to a matching pair is recovered through triangulation. Different methods have been proposed to solve the triangulation problem. The simplest solution likely consists of considering the two projection matrices, *P* and *P'*. Matrices seek 3D points in homogenous coordinates can be expressed as `X=[X,Y,Z,1]`^{T}, as we already know `x=PX` and `x'=P'X`. Each of these two homogenous equations brings two independent equations, which is sufficient to solve the three unknowns of the 3D point position. This overdetermined system of the equation can be solved using a least-square approach, which can be accomplished using a convenient OpenCV utility function called `cv::solve`. The complete function is as follows:

```
// triangulate using Linear LS-Method
cv::Vec3d triangulate(const cv::Mat &p1,
                      const cv::Mat &p2,
                      const cv::Vec2d &u1,
                      const cv::Vec2d &u2) {

// system of equations assuming image=[u,v] and X=[x,y,z,1]
// from u(p3.X)= p1.X and v(p3.X)=p2.X
cv::Matx43d A(u1(0)*p1.at<double>(2, 0) - p1.at<double>(0, 0),
              u1(0)*p1.at<double>(2, 1) - p1.at<double>(0, 1),
              u1(0)*p1.at<double>(2, 2) - p1.at<double>(0, 2),
              u1(1)*p1.at<double>(2, 0) - p1.at<double>(1, 0),
              u1(1)*p1.at<double>(2, 1) - p1.at<double>(1, 1),
              u1(1)*p1.at<double>(2, 2) - p1.at<double>(1, 2),
              u2(0)*p2.at<double>(2, 0) - p2.at<double>(0, 0),
              u2(0)*p2.at<double>(2, 1) - p2.at<double>(0, 1),
              u2(0)*p2.at<double>(2, 2) - p2.at<double>(0, 2),
              u2(1)*p2.at<double>(2, 0) - p2.at<double>(1, 0),
              u2(1)*p2.at<double>(2, 1) - p2.at<double>(1, 1),
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            u2(1)*p2.at<double>(2, 2) - p2.at<double>(1, 2));

    cv::Matx41d B(p1.at<double>(0, 3) - u1(0)*p1.at<double>(2, 3),
                  p1.at<double>(1, 3) - u1(1)*p1.at<double>(2, 3),
                  p2.at<double>(0, 3) - u2(0)*p2.at<double>(2, 3),
                  p2.at<double>(1, 3) - u2(1)*p2.at<double>(2, 3));

    // X contains the 3D coordinate of the reconstructed point
    cv::Vec3d X;
    // solve AX=B
    cv::solve(A, B, X, cv::DECOMP_SVD);
    return X;
}
```

We have noted in the previous section that very often, because of noise and digitization, the projection lines that should normally intersect do not intersect in practice. The least-square solution will, therefore, find a solution somewhere around the point of intersection. Additionally, this method will not work if you try to reconstruct a point at infinity. This is because, for such a point, the fourth element of the homogenous coordinates should be at `0` and not at `1`, as is assumed.

Finally, it is important to understand that this 3D reconstruction is done up to a scale factor only. If you need to make real measurements, you need to know at least one physical distance; for example, the real distance between the two cameras or the height of one of the visible objects.

There's more...

The 3D reconstruction is a rich field of research in computer vision, and there is much more to explore in the OpenCV library on the subject.

Decomposing a homography

In this recipe, we learned that an essential matrix can be decomposed in order to recover the rotation and translation components between two cameras. We also learned in `Chapter 10`, *Estimating Projective Relations in Images*, that a homography exists between two views of a plane. In this case, this homography also contains the rotational and translational components. Additionally, it contains information about the plane, namely, its normal with respect to each camera. The `cv::decomposeHomographyMat` function can be used to decompose this matrix; the condition, however, is to have a calibrated camera.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Bundle adjustment

In this recipe, we first estimated the camera position from matches and then reconstructed the associated 3D points through triangulation. It is possible to generalize this process by using any number of views. For each of these views, feature points are detected and are matched with the other views. By using this information, it is possible to write equations that relate the rotations and translations between the views, the set of 3D points, and the calibration information. All of these unknowns can be optimized together through a large optimization process that aims at minimizing the reprojection errors of all points in each view where they are visible. This combined optimization procedure is called a **bundle adjustment**. Take a look at the `cv::detail::BundleAdjusterReproj` class, which implements the camera parameter's refinement algorithm that minimizes the sum of the reprojection error squares.

See also

- *Triangulation* by R. Hartley and P. Sturm in *Computer Vision and Image Understanding Vol. 68, No. 2*, 1997, presents a formal analysis of different triangulation methods.
- *Modeling the World from Internet Photo Collections* by N. Snavely, S.M. Seitz, and R. Szeliski in *International Journal of Computer Vision, Vol. 80, No. 2*, 2008, describes a large-scale application of 3D reconstruction through bundle adjustment.

We've successfully learned how to reconstruct a 3D scene from calibrated cameras. Now let's move on to the next recipe!

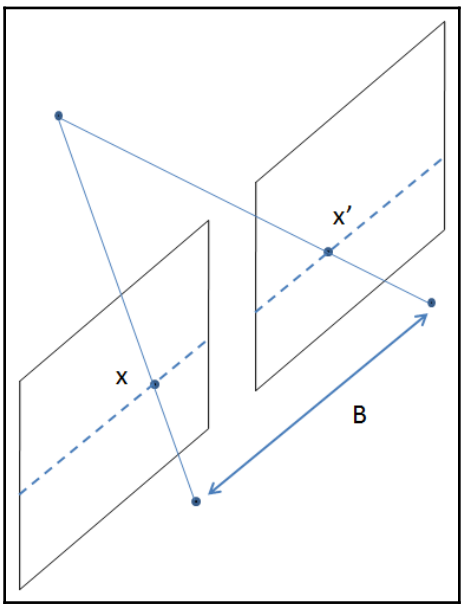
Computing depth from a stereo image

Humans view the world in three dimensions using their two eyes. Robots can do the same when they are equipped with two cameras; this is called **stereo vision**. A stereo rig is a pair of cameras mounted on a device, looking at the same scene and separated by a fixed baseline (that is, the distance between the two cameras). This recipe will demonstrate how a depth map can be computed from two stereo images by computing the depth correspondence between the two views.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

A stereo vision system is generally made of two side-by-side cameras looking toward the same direction. The following diagram illustrates such a stereo system in a perfectly-aligned configuration:



Under this ideal configuration, the cameras are only separated by a horizontal translation and, therefore, all epipolar lines are horizontal. This means that corresponding points have the same *y* coordinates, which reduces the search for matches on a 1D line. The difference in the *x* coordinates depends on the depth of the points. Points at infinity have image points at the same (*x*, *y*) coordinates, and the closer the points are to the stereo rig, then the greater the difference of their *x* coordinates. This fact can be demonstrated formally by looking at the projective equation. When cameras are separated by a pure horizontal translation, then the projective equation of the second camera (the one on the right) will be as follows:

$$s\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -B \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Here, for simplicity, we assume square pixels and the same calibration parameters for both cameras. Now, if you compute the difference of x-x' (do not forget to divide by s to normalize the homogenous coordinates) and isolate the z coordinate, you obtain the following equation:

$$Z = f\frac{(x - x')}{B}$$

The (x-x') term is called the **disparity**. To compute the depth map of a stereo vision system, the disparity of each pixel must be estimated. This recipe will show you how to do this.

How to do it...

The ideal configuration shown in the previous section is, in practice, very difficult to realize. Even if they are accurately positioned, the cameras of the stereo rig will unavoidably include some additional translational and rotational components. Fortunately, however, the images can be rectified so as to produce horizontal epilines. This can be achieved by computing the fundamental matrix of the stereo system using, for example, the robust matching algorithm of `Chapter 10`, *Estimating Projective Relations in Images*. This is what we did for the following stereo pair (with epipolar lines drawn on it):

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

OpenCV offers a rectifying function that uses a homographic transformation to project the image plane of each camera onto the perfectly aligned virtual plane, as follows:

1. This transformation is computed from a set of matched points and a fundamental matrix. Once computed, these homographies are then used to wrap the images:

```
// Compute homographic rectification
cv::Mat h1, h2;
cv::stereoRectifyUncalibrated(points1, points2,
                              fundamental,
                              image1.size(), h1, h2);

// Rectify the images through warping
cv::Mat rectified1;
cv::warpPerspective(image1, rectified1, h1, image1.size());
cv::Mat rectified2;
cv::warpPerspective(image2, rectified2, h2, image1.size());
```

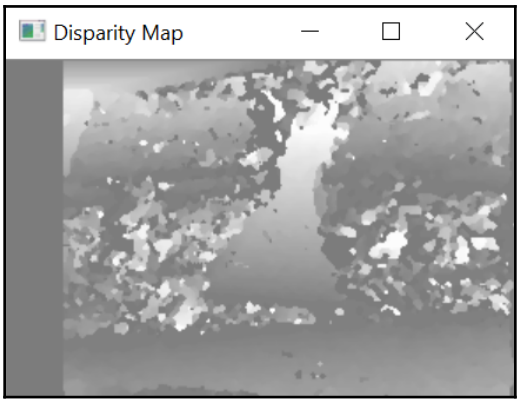
For our example, the rectified image pair is as follows:

2. Computing the disparity map can then be accomplished using methods that assume the parallelism of the cameras (and, consequently, the horizontal epipolar lines):

```
//Compute disparity
cv::Mat disparity;
cv::Ptr<cv::StereoMatcher> pStereo =
     cv::StereoSGBM::create(0,   //minimum disparity
                            32,  //maximum disparity
                            5);  //block size
pStereo->compute(rectified1, rectified2, disparity);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The obtained disparity map can then be displayed as an image. Bright values correspond to high disparities and, from what we learned earlier in this recipe, those high disparity values correspond to proximal objects:



The quality of the computed disparity mainly depends on the appearance of the different objects that compose the scene. Highly-textured regions tend to produce more accurate disparity estimates since they can be non-ambiguously matched. Additionally, a larger baseline increases the range of detectable depth values. However, enlarging the baseline also makes disparity computation more complex and less reliable.

How it works...

Computing disparities is a pixel-matching exercise. We already mentioned that when the images are properly rectified, then the search space is conveniently aligned with the image rows. The difficulty, however, is that in stereo vision, we are generally seeking a dense disparity map. That is, we want to match every pixel of one image with the pixels of the other image. This can be more challenging than selecting a few distinctive points in an image and finding their corresponding points in the other image. Disparity computation is, therefore, a complex process that is generally composed of four steps:

1. Matching the cost calculation
2. Cost aggregation
3. Disparity computation and optimization
4. Disparity refinement

These steps are detailed in the following paragraph.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Assigning a disparity to one pixel is putting a pair of points in correspondence in a stereo set. Finding the best disparity map is often posed as an optimization problem. From this perspective, matching two points has a cost that must be computed following a defined metric. This can be, for example, a simple absolute or the squared difference of intensities, colors, or gradients. In the search for an optimal solution, the matching cost is generally aggregated over a region in order to cope with local noise ambiguity. The global disparity map can then be estimated by evaluating an energy function that includes terms to smooth the disparity map, take into account any possible occlusion, and enforce a uniqueness constraint. Finally, a post-processing step is often applied in order to refine the disparity estimates during which, for example, planar regions are detected or depth discontinuities are detected.

OpenCV implements a number of disparity computation methods. Here, we used the `cv::StereoSGBM` approach. The simplest method is `cv::StereoBM`, which is based on block matching.

Finally, it should be noted that a more accurate rectification can be performed if you are ready to undergo a full calibration process. The `cv::stereoCalibrate` and `cv::stereoRectify` functions are, in this case, used in conjunction with a calibration pattern. The rectification mapping then computes new projection matrices for the cameras instead of simple homographies.

See also

- *A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms* by D. Scharstein and R. Szeliski in *International Journal of Computer Vision, Vol. 47,* 2002, is a classic reference on disparity computation methods.
- *Stereo Processing by Semi-Global Matching and Mutual Information* by H. Hirschmuller in *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 30, No. 2, pp. 328-341,* 2008, describes the approach used for computing the disparity in this recipe.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

12
Processing Video Sequences

Video signals constitute a rich source of visual information. They are made of a sequence of images, called **frames**, that are taken at regular time intervals (specified as the **frame rate**, generally expressed in frames per second) and show a scene in motion. With the advent of powerful computers, it is now possible to perform advanced visual analysis on video sequences—sometimes at rates close to, or even faster than, the actual video frame rate. This chapter will show you how to read, process, and store video sequences.

We will see that, once the individual frames of a video sequence have been extracted, the different image processing functions presented in this book can be applied to each of them. In addition, we will also look at a few algorithms that perform a temporal analysis of the video sequence, compare adjacent frames to track objects, or cumulate image statistics over time in order to extract foreground objects.

In this chapter, we will cover the following recipes:

- Reading video sequences
- Processing video frames
- Writing video sequences
- Extracting the foreground objects in a video

Reading video sequences

In order to process a video sequence, we need to be able to read each of its frames. OpenCV has put in place an easy-to-use framework that can help us perform frame extraction from video files or even from USB or IP cameras. This recipe shows you how to use it.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Basically, all you need to do in order to read the frames of a video sequence is create an instance of the `cv::VideoCapture` class. You then create a loop that will extract and read each video frame. Here, we are going to explain how to code the `main` function to show each frame. The following steps will help us to read video sequences:

1. Open a video capture with the following code:

```
int main()
{
  // Open the video file
  cv::VideoCapture capture("bike.avi");
```

2. Check whether the video has successfully opened; if not, return and exit:

```
  if (!capture.isOpened())
    return 1;
```

3. Get some video properties, such as the frame rate:

```
// Get the frame rate
  double rate= capture.get(CAP_PROP_FPS);
```

4. Create a variable to store each `frame` and a new window to show it:

```
  bool stop(false);
  cv::Mat frame; // current video frame
  cv::namedWindow("Extracted Frame");
```

5. Calculate `delay` between each frame to show in milliseconds:

```
  int delay= 1000/rate;
```

6. Loop until `stop`:

```
  // for all frames in video
  while (!stop) {
```

7. In each iteration, read a frame from the capture. If it's not possible, read more frames and exit from the loop:

```
    // read next frame if any
  if (!capture.read(frame))
    break;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

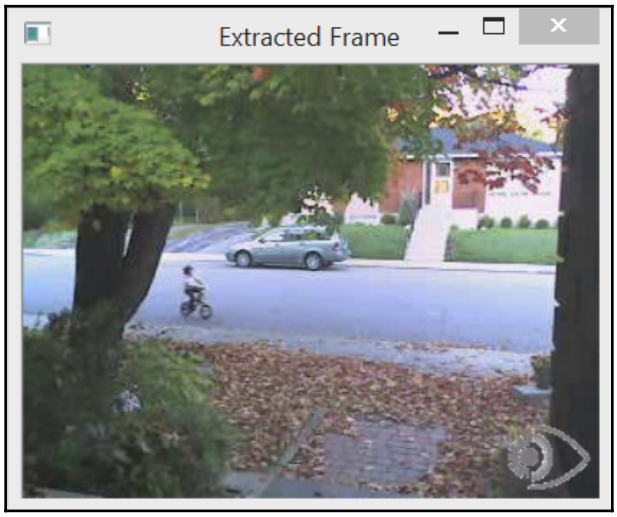
8. Show the image and wait for the delay for a key press to exit:

```
    cv::imshow("Extracted Frame",frame);

    // introduce a delay
    // or press key to stop
    if (cv::waitKey(delay)>=0)
      stop= true;
  }
```

9. Finally, when finishing the loop, close and release the video capture:

```
  // Close the video file.
  // Not required since called by destructor
  capture.release();
  return 0;
}
```

A window will appear on which the video will play, as shown in the following screenshot:



Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

To open a video, you simply need to specify the video filename. This can be done by providing the name of the file in the constructor of the `cv::VideoCapture` object. It is also possible to use the `open` method if `cv::VideoCapture` has already been created. Once the video is successfully opened (this can be verified through the `isOpened` method), it is possible to start the frame extraction. It is also possible to query the `cv::VideoCapture` object for information associated with the video file by using its `get` method with the appropriate flag. In the preceding example, we obtained the frame rate using the `CAP_PROP_FPS` flag. Since it is a generic function, it always returns a double, even if another type would be expected in some cases. For example, the total number of frames in the video file would be obtained (as an integer) as follows:

```
long t= static_cast<long>(
              capture.get(CAP_PROP_FRAME_COUNT));
```

Have a look at the different flags that are available in the OpenCV documentation in order to find out what information can be obtained from the video.

There is also a `set` method, which allows you to input some parameters to the `cv::VideoCapture` instance. For example, you can request to move to a specific frame using the `CAP_PROP_POS_FRAMES` flag:

```
// goto frame 100
double position= 100.0;
capture.set(CAP_PROP_POS_FRAMES, position);
```

You can also specify the position in milliseconds using `CAP_PROP_POS_MSEC`, or you can specify the relative position inside the video using `CAP_PROP_POS_AVI_RATIO` (with `0.0` corresponding to the beginning of the video and `1.0` to the end). The method returns `true` if the requested parameter setting is successful. Note that the possibility to get or set a particular video parameter largely depends on the codec that is used to compress and store the video sequence. If you are unsuccessful with some parameters, that could be simply due to the specific codec you are using.

Once the captured video is successfully opened, the frames can be sequentially obtained by repetitively calling the `read` method as we did in the example of the previous section. You can equivalently call the overloaded reading operator:

```
capture >> frame;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

It is also possible to call the two basic methods:

```
capture.grab();
capture.retrieve(frame);
```

Also note how, in our example, we introduced a delay in displaying each frame. This is done using the `cv::waitKey` function. Here, we set the delay at a value that corresponds to the input video frame rate (if fps is the number of frames per second, then 1,000/fps is the delay between two frames in milliseconds). You can obviously change this value to display the video at a slower or faster speed. However, if you are going to display the video frames, it is important that you insert such a delay if you want to make sure that the window has sufficient time to refresh (since it is a process of low priority, it will never refresh if the CPU is too busy). The `cv::waitKey` function also allows us to interrupt the reading process by pressing any key. In such a case, the function returns the ASCII code of the key that is pressed. Note that if the delay specified to the `cv::waitKey` function is `0`, then it will wait indefinitely for the user to press a key. This is very useful if someone wants to trace a process by examining the results frame by frame.

The final statement calls the `release` method, which will close the video file. However, this call is not required since `release` is also called by the `cv::VideoCapture` destructor.

It is important to note that, in order to open the specified video file, your computer must have the corresponding codec installed; otherwise, `cv::VideoCapture` will not be able to decode the input file. Normally, if you are able to open your video file with a video player on your machine (such as Windows Media Player), then OpenCV should also be able to read this file.

There's more...

You can also read the video stream capture of a camera that is connected to your computer (a USB camera, for example). In this case, you simply specify an ID number (an integer) instead of a filename to the `open` function. Specifying `0` for the ID will open the default installed camera. In this case, the role of the `cv::waitKey` function, which stops the processing, becomes essential since the video stream from the camera would otherwise be infinitely read.

Finally, it is also possible to load a video from the web. In this case, all you have to do is provide the correct address, for example by using the following code:

```
cv::VideoCapture capture("http://www.laganiere.name/bike.avi");
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The *Writing video sequences* recipe in this chapter has more information on video codecs.
- The `ffmpeg.org` website presents a completely open source and cross-platform solution for audio/video reading, recording, converting, and streaming. The OpenCV classes that manipulate video files are built on top of this library.

We've successfully learned how to read video sequences. Now, let's move on to the next recipe!

Processing video frames

In this recipe, our objective is to apply some processing function to each of the frames of a video sequence. We will do this by encapsulating the OpenCV video capture framework into our own class. Among other things, this class will allow us to specify a function that will be called each time a new frame is extracted.

How to do it...

What we want is to be able to specify a processing function (a `callback` function) that will be called for each frame of a video sequence. This function can be defined as receiving a `cv::Mat` instance and outputting a processed frame as follows:

1. Therefore, in our framework, the processing function must have the following signature to be a valid callback:

```
void processFrame(cv::Mat& img, cv::Mat& out);
```

2. As an example of such a processing function, consider the following simple function that computes the `Canny` edges of an input image:

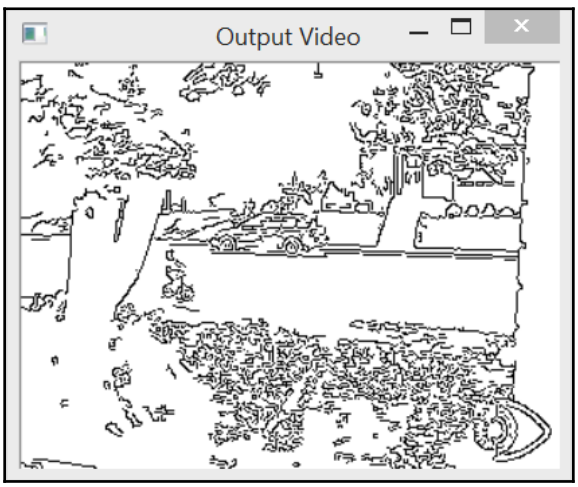
```
void canny(cv::Mat& img, cv::Mat& out) {
  // Convert to gray
  if (img.channels()==3)
    cv::cvtColor(img,out,COLOR_BGR2GRAY);
  // Compute Canny edges
  cv::Canny(out,out,100,200);
  // Invert the image
  cv::threshold(out,out,128,255,cv::THRESH_BINARY_INV);
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Our `VideoProcessor` class encapsulates all aspects of a video-processing task. Using this class, the procedure will be to create a class instance, specify an input video file, attach the `callback` function to it, and then start the process. Programmatically, these steps are accomplished using our proposed class, as follows:

```
// Create instance
VideoProcessor processor;
// Open video file
processor.setInput("bike.avi");
// Declare a window to display the video
processor.displayInput("Current Frame");
processor.displayOutput("Output Frame");
// Play the video at the original frame rate
processor.setDelay(1000./processor.getFrameRate());
// Set the frame processor callback function
processor.setFrameProcessor(canny);
// Start the process
processor.run();
```

If this code is run, then two windows will play the input video and the output result at the original frame rate (a consequence of the delay introduced by the `setDelay` method). For example, considering the input video for which a frame is shown in the previous recipe, the output window will look as follows:



Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

As we did in other recipes, our objective was to create a class that encapsulates the common functionalities of a video-processing algorithm. As you might expect, the class includes several member variables that control the different aspects of the video frame processing:

```
class VideoProcessor {
  private:
  // the OpenCV video capture object
  cv::VideoCapture capture;
  // the callback function to be called
  // for the processing of each frame
  void (*process)(cv::Mat&, cv::Mat&);
  // a bool to determine if the
  // process callback will be called
  bool callIt;
  // Input display window name
  std::string windowNameInput;
  // Output display window name
  std::string windowNameOutput;
  // delay between each frame processing
  int delay;
  // number of processed frames
  long fnumber;
  // stop at this frame number
  long frameToStop;
  // to stop the processing
  bool stop;
```

The first member variable is the `cv::VideoCapture` object. The second attribute is the `process` function pointer, which will point to the callback function. This function can be specified using the corresponding setter method:

```
  // set the callback function that
  // will be called for each frame
  void setFrameProcessor(
    void (*frameProcessingCallback)
      cv::Mat&, cv::Mat&)) {
        process= frameProcessingCallback;
      }
```

The following method opens the video file:

```
  // set the name of the video file
  bool setInput(std::string filename) {
    fnumber= 0;
    // In case a resource was already
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    // associated with the VideoCapture instance
    capture.release();
    // Open the video file
    return capture.open(filename);
}
```

It is generally interesting to display the frames as they are processed. Therefore, two methods are used to create the display windows:

```
// to display the input frames
void displayInput(std::string wn) {
  windowNameInput= wn;
  cv::namedWindow(windowNameInput);
}

// to display the processed frames
void displayOutput(std::string wn) {
  windowNameOutput= wn;
  cv::namedWindow(windowNameOutput);
}
```

The main method, called `run`, is the one that contains the frame extraction loop:

```
// to grab (and process) the frames of the sequence
void run() {

  // current frame
  cv::Mat frame;
  // output frame
  cv::Mat output;

  // if no capture device has been set
  if (!isOpened())
    return;

    stop= false;

    while (!isStopped()) {

    // read next frame if any
    if (!readNextFrame(frame))
      break;

    // display input frame
    if (windowNameInput.length()!=0)
      cv::imshow(windowNameInput,frame);

    // calling the process function
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    if (callIt) {

      // process the frame
      process(frame, output);
      // increment frame number
      fnumber++;

      else { // no processing
        output= frame;
    }

      // display output frame
      if (windowNameOutput.length()!=0)
        cv::imshow(windowNameOutput,output);

      // introduce a delay
      if (delay>=0 && cv::waitKey(delay)>=0)
        stopIt();

      // check if we should stop
      if (frameToStop>=0 &&
        getFrameNumber()==frameToStop)
          stopIt();
    }
  }

  // Stop the processing
  void stopIt() {
    stop= true;
  }

  // Is the process stopped?
  bool isStopped() {
    return stop;
  }

  // Is a capture device opened?
  bool isOpened() {
    capture.isOpened();
  }

  // set a delay between each frame
  // 0 means wait at each frame
  // negative means no delay
  void setDelay(int d) {

    delay= d;
  }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This method uses a `private` method that reads the frames:

```
// to get the next frame
// could be: video file or camera
bool readNextFrame(cv::Mat& frame) {
  return capture.read(frame);
}
```

The `run` method proceeds by first calling the `read` method of the `cv::VideoCapture` OpenCV class. There is then a series of operations that are executed, but, before each of them is invoked, a check is made to determine whether it has been requested. The input window is displayed only if an input window name has been specified (using the `displayInput` method); the `callback` function is called only if one has been specified (using `setFrameProcessor`). The output window is displayed only if an output window name has been defined (using `displayOutput`); a delay is introduced only if one has been specified (using the `setDelay` method). Finally, the current frame number is checked if a stop frame has been defined (using `stopAtFrameNo`).

You might also wish to simply open and play the video file (without calling the `callback` function). Therefore, we have two methods that specify whether or not we want the `callback` function to be called:

```
// process callback to be called
void callProcess() {
  callIt= true;
}

// do not call process callback
void dontCallProcess() {
  callIt= false;
}
```

Finally, the class also offers us the possibility to stop at a certain frame number:

```
void stopAtFrameNo(long frame) {
  frameToStop= frame;
}

// return the frame number of the next frame
long getFrameNumber() {
  // get info of from the capture device
  long fnumber= static_cast<long>(
    capture.get(CAP_PROP_POS_FRAMES));
  return fnumber;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The class also contains a number of getter and setter methods that are basically a wrapper over the general `set` and `get` methods of the `cv::VideoCapture` framework.

There's more...

Our `VideoProcessor` class is there to facilitate the deployment of a video-processing module. Few additional refinements can be made to it.

Processing a sequence of images

Sometimes, the input sequence is made of a series of images that are individually stored in distinct files. Our class can be easily modified to accommodate such input. You just need to add a member variable that will hold a vector of image filenames and its corresponding iterator:

```
// vector of image filename to be used as input
std::vector<std::string> images;
// image vector iterator
std::vector<std::string>::const_iterator itImg;
```

A new `setInput` method is used to specify the filenames to be read:

```
// set the vector of input images
bool setInput(const std::vector<std::string>& imgs) {
  fnumber= 0;
  // In case a resource was already
  // associated with the VideoCapture instance
  capture.release();

  // the input will be this vector of images
  images= imgs;
  itImg= images.begin();

  return true;
}
```

The `isOpened` method becomes as follows:

```
// Is a capture device opened?
bool isOpened() {
  return capture.isOpened() || !images.empty();
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The last method that needs to be modified is the private `readNextFrame` method that will read from the video or from the vector of filenames, depending on the input that has been specified. The test is that if the vector of image filenames is not empty, then that is because the input is an image sequence. The call to `setInput` with a video filename clears this vector:

```
// to get the next frame
// could be: video file; camera; vector of images
bool readNextFrame(cv::Mat& frame) {
  if (images.size()==0)
    return capture.read(frame);
    else {
      if (itImg != images.end()) {
        frame= cv::imread(*itImg);
        itImg++;
        return frame.data != 0;
      } else {
      return false;
    }
  }
```

Using a frame processor class

In an object-oriented context, it might make more sense to use a frame processing class instead of a frame processing function. Indeed, a class would give the programmer much more flexibility in the definition of a video-processing algorithm. We can, therefore, define an interface that any class that wishes to be used inside `VideoProcessor` will need to implement:

```
// The frame processor interface
class FrameProcessor {
  public:
  // processing method
  virtual void process(cv:: Mat &input, cv:: Mat &output)= 0;
};
```

A setter method allows you to input a `FrameProcessor` instance to the `VideoProcessor` framework and assign it to the added `frameProcessor` member variable that is defined as a pointer to a `FrameProcessor` object:

```
// set the instance of the class that
// implements the FrameProcessor interface
void setFrameProcessor(FrameProcessor* frameProcessorPtr)
{
  // invalidate callback function
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    process= 0;
    // this is the frame processor instance
    // that will be called
    frameProcessor= frameProcessorPtr;
    callProcess();
  }
```

When a frame processor class instance is specified, it invalidates any frame processing function that could have been set previously. The same obviously applies if a frame processing function is specified instead. The `while` loop of the `run` method is modified to take into account this modification:

```
  while (!isStopped()) {
    // read next frame if any
    if (!readNextFrame(frame))
      break;

    // display input frame
    if (windowNameInput.length()!=0)
      cv::imshow(windowNameInput,frame);

    // ** calling the process function or method **
    if (callIt) {

      // process the frame
      if (process) // if call back function
        process(frame, output);
      else if (frameProcessor)
        // if class interface instance
        frameProcessor->process(frame,output);
      // increment frame number
      fnumber++;

    } else {

      output= frame;
    }
     // display output frame
     if (windowNameOutput.length()!=0)
       cv::imshow(windowNameOutput,output);
     // introduce a delay
     if (delay>=0 && cv::waitKey(delay)>=0)
       stopIt();
     // check if we should stop
     if (frameToStop>=0 && getFrameNumber()==frameToStop)
       stopIt();
  }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The *Tracking feature points in a video* recipe in this chapter gives you an example of how to use the `FrameProcessor` class interface

We've successfully learned how to process video frames. Now, let's move on to the next recipe!

Writing video sequences

In the previous recipes, we learned how to read a video file and extract its frames. This recipe will show you how to write frames and, therefore, create a video file. This will allow us to complete the typical video-processing chain—reading an input video stream, processing its frames, and then storing the results in a new video file.

How to do it...

Let's perform the following steps:

1. Writing video files in OpenCV is done using the `cv::VideoWriter` class. An instance is constructed by specifying the filename, the frame rate at which the generated video should play, the size of each frame, and whether or not the video will be created in color:

```
writer.open(outputFile, // filename
    codec,          // codec to be used
    framerate,      // frame rate of the video
    frameSize,      // frame size
    isColor);       // color video?
```

2. In addition, you must specify the way you want the video data to be saved. This is the `codec` argument; this one will be discussed at the end of this recipe.
3. Once the video file is opened, frames can be added to it by repetitively calling the `write` method:

```
writer.write(frame); // add the frame to the video file
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Using the `cv::VideoWriter` class, our `VideoProcessor` class introduced in the previous recipe can easily be expanded in order to give it the ability to write video files. A simple program that will read a video, process it, and write the result to a video file would then be written as follows:

```
// Create instance
VideoProcessor processor;

// Open video file
processor.setInput("bike.avi");
processor.setFrameProcessor(canny);
processor.setOutput("bikeOut.avi");
// Start the process
processor.run();
```

5. Proceeding as we did in the preceding recipe, we also want to give the user the possibility to write the frames as individual images. In our framework, we adopt a naming convention that consists of a prefix name followed by a number made of a given number of digits. This number is automatically incremented as frames are saved. Then, to save the output result as a series of images, you would change the preceding statement by this one:

```
processor.setOutput("bikeOut",  //prefix
".jpg",     // extension
3,          // number of digits
0)// starting index
```

6. Using the specified number of digits, this call will create the `bikeOut000.jpg`, `bikeOut001.jpg`, and `bikeOut002.jpg` files, and so on.

Now, let's go behind the scenes to understand the code better.

How it works...

Let's now describe how to modify our `VideoProcessor` class in order to give it the ability to write video files. First, a `cv::VideoWriter` variable member must be added to our class (plus a few other attributes):

```
class VideoProcessor {

  private:

  ...
  // the OpenCV video writer object
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
cv::VideoWriter writer;
// output filename
std::string outputFile;
// current index for output images
int currentIndex;
// number of digits in output image filename
int digits;
// extension of output images
std::string extension;
```

An extra method is used to specify (and open) the output video file:

```
// set the output video file
// by default the same parameters than
// input video will be used
bool setOutput(const std::string &filename, int codec=0,
   double framerate=0.0, bool isColor=true) {

  outputFile= filename;
  extension.clear();

  if (framerate==0.0)
    framerate= getFrameRate(); // same as input

  char c[4];
  // use same codec as input
  if (codec==0) {
    codec= getCodec(c);
  }

  // Open output video
  return writer.open(outputFile, // filename
  codec,          // codec to be used
  framerate,      // frame rate of the video
  getFrameSize(), // frame size
  isColor);       // color video?
}
```

A private method, called the `writeNextFrame` method, handles the frame writing procedure (in a video file or as a series of images):

```
// to write the output frame
// could be: video file or images
void writeNextFrame(cv::Mat& frame) {

  if (extension.length()) { // then we write images

    std::stringstream ss;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
      // compose the output filename
      ss << outputFile << std::setfill('0') << std::setw(digits)
         << currentIndex++ << extension;
      cv::imwrite(ss.str(),frame);

    } else { // then write to video file
      writer.write(frame);
    }
  }
```

For the case where the output is made of individual image files, we need an additional setter method:

```
// set the output as a series of image files
// extension must be ".jpg", ".bmp" ...
bool setOutput(const std::string &filename, // prefix
  const std::string &ext, // image file extension
  int numberOfDigits=3,   // number of digits
  int startIndex=0) {     // start index

  // number of digits must be positive
  if (numberOfDigits<0)
    return false;

  // filenames and their common extension
  outputFile= filename;
  extension= ext;

  // number of digits in the file numbering scheme
  digits= numberOfDigits;
  // start numbering at this index
  currentIndex= startIndex;

  return true;
}
```

Finally, a new step is then added to the video capture loop of the `run` method:

```
    while (!isStopped()) {

      // read next frame if any
      if (!readNextFrame(frame))
        break;

      // display input frame
      if (windowNameInput.length()!=0)
        cv::imshow(windowNameInput,frame);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
      // calling the process function or method
      if (callIt) {

        // process the frame
        if (process)
          process(frame, output);
        else if (frameProcessor)
          frameProcessor->process(frame,output);
        // increment frame number
        fnumber++;

      } else {

        output= frame;
      }

      // ** write output sequence **
      if (outputFile.length()!=0)
        writeNextFrame(output);

      // display output frame
      if (windowNameOutput.length()!=0)
        cv::imshow(windowNameOutput,output);
      // introduce a delay
      if (delay>=0 && cv::waitKey(delay)>=0)
        stopIt();

      // check if we should stop
      if (frameToStop>=0 && getFrameNumber()==frameToStop)
        stopIt();
    }
  }
```

There's more...

When a video is written to a file, it is saved using a codec. A **codec** is a software module that is capable of encoding and decoding video streams. The codec defines both the format of the file and the compression scheme that is used to store the information. Obviously, a video that has been encoded using a given codec must be decoded with the same codec. For this reason, four-character codes have been introduced to uniquely identified codecs. This way, when a software tool needs to write a video file, it determines the codec to be used by reading the specified four-character code.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The codec four-character code

As the name suggests, the four-character code is made of four ASCII characters that can also be converted into an integer by appending them together. Using the `CAP_PROP_FOURCC` flag of the `get` method of an opened `cv::VideoCapture` instance, you can obtain this code of an opened video file. We can define a method in our `VideoProcessor` class to return the four-character code of an input video:

```
// get the codec of input video
int getCodec(char codec[4]) {

  // undefined for vector of images
  if (images.size()!=0) return -1;

  union { // data structure for the 4-char code
    nt value;
    char code[4]; } returned;

  // get the code
  returned.value= static_cast<int>
     (capture.get(CAP_PROP_FOURCC));

  // get the 4 characters
  codec[0]= returned.code[0];
  codec[1]= returned.code[1];
  codec[2]= returned.code[2];
  codec[3]= returned.code[3];

  // return the int value corresponding to the code
  return returned.value;
}
```

The `get` method always returns a `double` value that is then cast into an integer. This integer represents the code from which the four characters can be extracted using a `union` data structure. If we open our test video sequence, then we have the following statements:

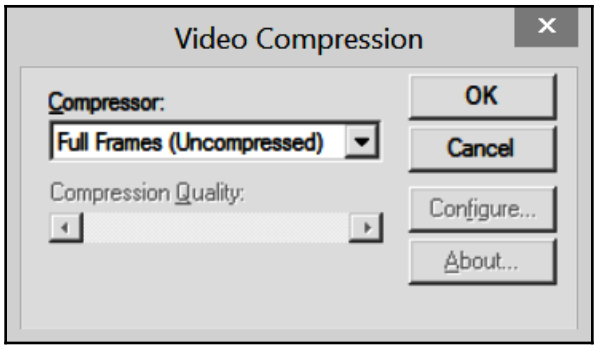
```
char codec[4];
processor.getCodec(codec);
std::cout << "Codec: " << codec[0] << codec[1] << codec[2] <<
   codec[3] << std::endl;
```

From the preceding statements, we obtain the following:

```
Codec : XVID
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

When a video file is written, the codec must be specified using its four-character code. This is the second parameter in the `open` method of the `cv::VideoWriter` class. You can use, for example, the same one as the input video (this is the default option in our `setOutput` method). You can also pass the value `-1` and the method will pop up a window that will ask you to select one codec from the list of available codecs, as shown here:



The list you will see on this window corresponds to the list of installed codecs on your machine. The code of the selected codec is then automatically sent to the `open` method.

See also

- The `Xvid.org` website offers you an open source video codec library based on the MPEG-4 standard for video compression. Xvid also has a competitor called DivX, which offers proprietary but free codec and software tools.

We've successfully learned to write video sequences. Now, let's move on to the next recipe!

Extracting the foreground objects in a video

When a fixed camera observes a scene, the background remains mostly unchanged. In this case, the interesting elements are the moving objects that evolve inside this scene. In order to extract these foreground objects, we need to build a model of the background, and then compare this model with a current frame in order to detect any foreground objects. This is what we will do in this recipe. Foreground extraction is a fundamental step in intelligent surveillance applications.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

If we had an image of the background of the scene (that is, a frame that contains no foreground objects) at our disposal, then it would be easy to extract the foreground of a current frame through a simple image difference:

```
// compute difference between current image and background
cv::absdiff(backgroundImage,currentImage,foreground);
```

Each pixel for which this difference is high enough would then be declared as a foreground pixel. However, most of the time, this background image is not readily available. Indeed, it could be difficult to guarantee that no foreground objects are present in a given image, and, in busy scenes, such situations might rarely occur. Moreover, the background scene often evolves over time because, for instance, the lighting condition changes (for example, from sunrise to sunset) or because new objects can be added or removed from the background.

Therefore, it is necessary to dynamically build a model of the background scene. This can be done by observing the scene for a period of time. If we assume that, most often, the background is visible at each pixel location, then it could be a good strategy to simply compute the average of all of the observations. However, this is not feasible for a number of reasons. First, this would require a large number of images to be stored before computing the background. Second, while we are accumulating images to compute our average image, no foreground extraction will be done. This solution also raises the problem of when and how many images should be accumulated to compute an acceptable background model. In addition, the images where a given pixel is observing a foreground object would have an impact on the computation of the average background.

A better strategy is to dynamically build the background model by regularly updating it. This can be accomplished by computing what is called a **running average** (also called **moving average**). This is a way to compute the average value of a temporal signal that takes into account the latest received values. If *pt* is the pixel value at a given time *t* and $\mu t\text{-}1$ is the current average value, then this average is updated using the following formula:

$$\mu_t = (1 - \alpha)\mu_{t-1} + \alpha p_t$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The α parameter is called the **learning rate**, and it defines the influence of the current value over the currently estimated average. The larger this value is, the faster the running average will adapt to changes in the observed values. To build a background model, you just have to compute a running average for every pixel of the incoming frames. The decision to declare a foreground pixel is then simply based on the difference between the current image and the background model.

How to do it...

The following steps will help us to extract the foreground images as follows:

1. Let's build a class that will learn about a `background` model using moving averages and that will extract `foreground` objects by subtraction. The required attributes are the following:

```
class BGFGSegmentor : public FrameProcessor {

  cv::Mat gray;         // current gray-level image
  cv::Mat background;   // accumulated background
  cv::Mat backImage;    // current background image
  cv::Mat foreground;   // foreground image
  // learning rate in background accumulation
  double learningRate;
  int threshold;        // threshold for foreground extraction
```

2. The main process consists of comparing the current `frame` with the `background` model and then updating this model:

```
  // processing method
  void process(cv:: Mat &frame, cv:: Mat &output) {

    // convert to gray-level image
    cv::cvtColor(frame, gray, COLOR_BGR2GRAY);

    // initialize background to 1st frame
    if (background.empty())
      gray.convertTo(background, CV_32F);

    // convert background to 8U
    background.convertTo(backImage,CV_8U);

    // compute difference between image and background
    cv::absdiff(backImage,gray,foreground);

    // apply threshold to foreground image
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

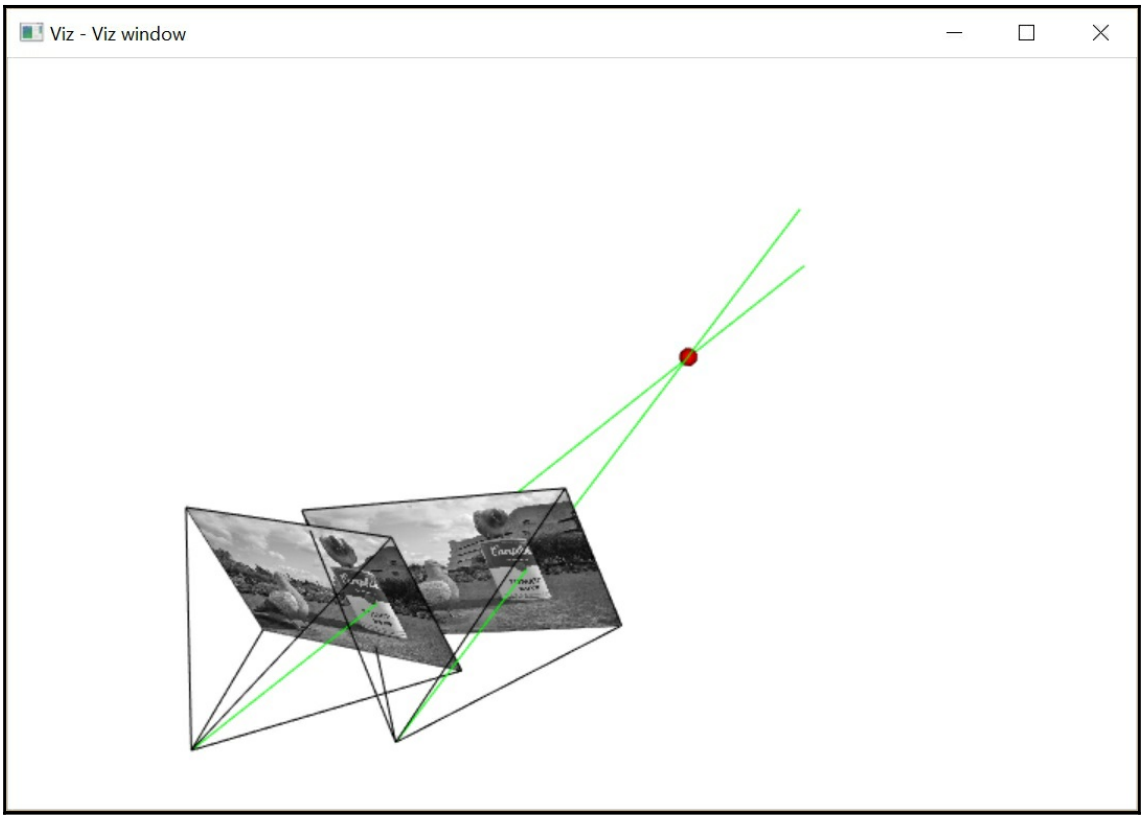
```
    cv::threshold(foreground,output,threshold,255,cv::
       THRESH_BINARY_INV);

    // accumulate background
    cv::accumulateWeighted(gray, background,
      // alpha*gray + (1-alpha)*background
      learningRate,  // alpha
      output);       // mask

    }
```

3. Using our video-processing framework, the foreground extraction program will be built as follows:

```
int main()
{
  // Create video procesor instance
  VideoProcessor processor;

  // Create background/foreground segmentor
  BGFGSegmentor segmentor;
  segmentor.setThreshold(25);

   // Open video file
   processor.setInput("bike.avi");

  // set frame processor
  processor.setFrameProcessor(&segmentor);

  // Declare a window to display the video
  processor.displayOutput("Extracted Foreground");

  // Play the video at the original frame rate
  processor.setDelay(1000./processor.getFrameRate());

  // Start the process
  processor.run();
}
```

One of the resulting binary foreground images that will be displayed is as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Now, let's go behind the scenes to understand the code better.

How it works...

Computing the running average of an image is easily accomplished through the `cv::accumulateWeighted` function, which applies the running average formula to each pixel of the image. Note that the resulting image must be a floating point image. This is why we had to convert the background model into a background image before comparing it with the current frame. A simple thresholded absolute difference (computed by `cv::absdiff` followed by `cv::threshold`) extracts the foreground image. Note that we then used the foreground image as a mask to `cv::accumulateWeighted`, in order to avoid the updating of pixels, declared as foreground. This works because our foreground image is defined as being `false` (that is, `0`) at foreground pixels (which also explains why the foreground objects are displayed as black pixels in the resulting image).

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, it should be noted that, for simplicity, the background model that is built by our program is based on the gray-level version of the extracted frames. Maintaining a color background would require the computation of a running average in some color space. However, the main difficulty in the presented approach is to determine the appropriate value for the threshold that would give good results for a given video.

There's more...

The preceding simple method to extract foreground objects in a scene works well for simple scenes that show a relatively stable background. However, in many situations, the background scene might fluctuate in certain areas between different values, thus causing frequent false foreground detections. These might be due to, for example, a moving background object (for example, tree leaves) or a glaring effect (for example, on the surface of the water). Casted shadows also pose a problem, since these ones are often detected as part of a moving object. In order to cope with these problems, more sophisticated background modeling methods have been introduced.

The mixture of Gaussian method

One of these algorithms is the **mixture of Gaussian** method. It proceeds in a way that is similar to the method presented in this recipe but adds a number of improvements.

First, the method maintains more than one model per pixel (that is, more than one running average). This way, if a background pixel fluctuates between, let's say, two values, two running averages are then stored. A new pixel value will be declared as the foreground only if it does not belong to any of the most frequently observed models. The number of models used is a parameter of the method and a typical value is five.

Second, not only is the running average maintained for each model, but also for the running variance. This one is computed as follows:

$$\sigma_t^2 = (1-\alpha)\sigma_{t-1}^2 + \alpha(p_t - \mu_t)^2$$

These computed average and variances are used to build a Gaussian model from which the probability of a given pixel value to belonging to the background can be estimated. This makes it easier to determine an appropriate threshold since it is now expressed as a probability rather than an absolute difference. Consequently, in areas where the background values have larger fluctuations, a greater difference will be required to declare a foreground object.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, when a given Gaussian model is not hit sufficiently often, it is excluded as being part of the background model. Reciprocally, when a pixel value is found to be outside the currently maintained background models (that is, it is a foreground pixel), a new Gaussian model is created. If in the future, this new model becomes a hit, then it becomes associated with the background.

This more sophisticated algorithm is obviously more complex to implement than our simple background/foreground segmentor. Fortunately, an OpenCV implementation exists, called `cv::BackgroundSubtractorMOG`, and is defined as a subclass of the more general `cv::BackgroundSubtractor` class. When used with its default parameter, this class is very easy to use:

```
int main()
{
  // Open the video file
  cv::VideoCapture capture("bike.avi");
  // check if video successfully opened
  if (!capture.isOpened())
    return 0;
  // current video frame
  cv::Mat frame;
  // foreground binary image
  cv::Mat foreground;
  cv::namedWindow("Extracted Foreground");
  // The Mixture of Gaussian object
  // used with all default parameters
  cv::BackgroundSubtractorMOG mog;
  bool stop(false);
  // for all frames in video
  while (!stop) {
    // read next frame if any
    if (!capture.read(frame))
      break;
    // update the background
    // and return the foreground
    mog(frame,foreground,0.01)
    // learning rate
    // Complement the image
    cv::threshold(foreground,foreground,
       128,255,cv::THRESH_BINARY_INV);
    // show foreground
    cv::imshow("Extracted Foreground",foreground);

    // introduce a delay
    // or press key to stop
    if (cv::waitKey(10)>=0)
      stop= true;
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    }
  }
```

As can be seen in the preceding code snippet, it is just a matter of creating the class instance and calling the method that simultaneously updates the background and returns the foreground image (the extra parameter being the learning rate). Also, note that the background model is computed in color here. The method implemented in OpenCV also includes a mechanism to reject shadows by checking whether the observed pixel variation is simply caused by a local change in brightness (if so, then it is probably due to a shadow) or whether it also includes some change in chromaticity.

A second implementation is also available and is simply called `cv::BackgroundSubtractorMOG2`. One of the improvements is that the number of appropriate Gaussian models per pixel to be used is now determined dynamically. You can use this one in place of the previous one in the preceding example. You should run these different methods on a number of videos in order to appreciate their respective performances. In general, you will observe that `cv::BackgroundSubtractorMOG2` is much faster.

See also

- The article by C. Stauffer and W.E.L. Grimson, *Adaptive background mixture models for real-time tracking*, in *Conf. on Computer Vision and Pattern Recognition*, 1999, gives you a more complete description of the mixture of Gaussian algorithm.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

13
Tracking Visual Motion

Video sequences are interesting because they show scenes and objects in motion. The preceding chapter introduced the tools for reading, processing, and saving videos. In this chapter, we will look at different algorithms that track the visible motion in a sequence of images. This visible or **apparent motion** can be caused by objects that move in different directions and at various speeds, or by the motion of the camera (or a combination of both).

Tracking apparent motion is of utmost importance for many applications. It allows you to follow specific objects while they are moving in order to estimate their speed and determine where they are going. It also permits you to stabilize videos taken from handheld cameras by removing or reducing the amplitude of camera jitters. Motion estimation is also used in video coding to compress a video sequence in order to facilitate its transmission or storage. This chapter will present a number of algorithms that track the motion in an image sequence, and as you will see, this tracking can be achieved either sparsely (that is, in fewer locations of an image—this is **sparse motion**) or densely (that is, at every pixel of an image—this is **dense motion**).

In this chapter, we will cover the following recipes:

- Tracing feature points in a video
- Estimating the optical flow
- Tracking an object in a video

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Tracing feature points in a video

We learned in previous chapters that analyzing an image through some of its most distinctive points can lead to effective and efficient computer vision algorithms. This is also true for image sequences, in which the motion of some interest points can be used to understand how the different elements of a captured scene move. In this recipe, you will learn how to perform a temporal analysis of a sequence by tracking feature points as they move from frame to frame.

How to do it...

We will now start with the tracing of feature points in a video, frame by frame. Let's take a look at the following steps:

1. To start the tracking process, the first thing to do is to detect the feature points in an initial frame. You then try to track these points in the next frame.
2. Obviously, since we are dealing with a video sequence, there is a good chance that the object, on which the feature points are found, has moved (this motion can also be due to camera movement). Therefore, you must search around a point's previous location in order to find its new location in the next frame. This is what the `cv::calcOpticalFlowPyrLK` function accomplishes. That is, you input two consecutive frames and a vector of feature points in the first image; the function then returns a vector of the new point locations. To track the points over a complete sequence, you repeat this process from frame to frame. Note that as you follow the points across the sequence, you will unavoidably lose track of some of them such that the number of tracked feature points will gradually reduce. Therefore, it could be a good idea to detect new features from time to time.

3. We will now take advantage of the video-processing framework that we defined in `Chapter 12`, *Processing Video Sequences*, and we will define a class that implements the `FrameProcessor` interface, which will be introduced in the *Processing the video frames* recipe of this chapter. The data attributes of this class include the variables that are required to perform both the detection of feature points and their tracking:

```
class FeatureTracker : public FrameProcessor {

  cv::Mat gray;      // current gray-level image
  cv::Mat gray_prev; // previous gray-level image
  // tracked features from 0->1
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
std::vector<cv::Point2f> points[2];
// initial position of tracked points
std::vector<cv::Point2f> initial;
std::vector<cv::Point2f> features;  // detected features
int max_count;     // maximum number of features to detect
double qlevel;     // quality level for feature detection
double minDist;    // min distance between two points
std::vector<uchar> status; // status of tracked features
std::vector<float> err;    // error in tracking

public:

FeatureTracker() : max_count(500), qlevel(0.01), minDist(10.) {}
```

4. Next, we define the `process` method that will be called for each frame of the sequence. Essentially, we need to proceed as follows:
    1. First, the feature points are detected if necessary.
    2. Next, these points are tracked.
    3. You reject the points that you cannot track or you no longer want to track. You are now ready to handle the successfully tracked points.
    4. Finally, the current frame and its points become the previous frame and points for the next iteration; you can do this as follows:

```
void process(cv:: Mat &frame, cv:: Mat &output) {

  // convert to gray-level image
  cv::cvtColor(frame, gray, COLOR_BGR2GRAY);
  frame.copyTo(output);

  // 1. if new feature points must be added
  if(addNewPoints()){
    // detect feature points
    detectFeaturePoints();
    // add the detected features to
    // the currently tracked features
    points[0].insert(points[0].end(),
                     features.begin(), features.end());
    initial.insert(initial.end(),
                   features.begin(), features.end());
  }

  // for first image of the sequence
  if(gray_prev.empty())
    gray.copyTo(gray_prev);

  // 2. track features
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        cv::calcOpticalFlowPyrLK(
              gray_prev, gray, // 2 consecutive images
              points[0],       // input point positions in
    first image
              points[1],       // output point positions in the
    2nd image
              status,          // tracking success
              err);            // tracking error

        // 3. loop over the tracked points to reject some
        int k=0;
        for( int i= 0; i < points[1].size(); i++ ) {

          // do we keep this point?
          if (acceptTrackedPoint(i)) {
            // keep this point in vector
            initial[k]= initial[i];
            points[1][k++] = points[1][i];
          }
        }

        // eliminate unsuccesful points
        points[1].resize(k);
        initial.resize(k);

        // 4. handle the accepted tracked points
        handleTrackedPoints(frame, output);

        // 5. current points and image become previous ones
        std::swap(points[1], points[0]);
        cv::swap(gray_prev, gray);
      }
```

5. This method makes use of four utility methods. It should be easy for you to change any of these methods in order to define new behavior for your own tracker. The first of these methods detect the feature points; note that we have already discussed the `cv::goodFeatureToTrack` function in the first recipe of `Chapter 8`, *Detecting Interest Points*:

```
// feature point detection
void detectFeaturePoints() {

  // detect the features
  cv::goodFeaturesToTrack(gray, // the image
            features,    // the output detected features
            max_count,   // the maximum number of features
            qlevel,      // quality level
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
                minDist);    // min distance between two features
}
```

6. The second method determines whether new feature points should be detected. This will happen when a negligible number of tracked points remain:

```
// determine if new points should be added
bool addNewPoints() {

  // if too few points
  return points[0].size()<=10;
}
```

7. The third method rejects some of the tracked points based on a criterion defined by the application. Here, we decided to reject the points that do not move (in addition to those that cannot be tracked by the `cv::calcOpticalFlowPyrLK` function). We consider that the non-moving points belong to the background scene and are, therefore, uninteresting:

```
  //determine which tracked point should be accepted
  bool acceptTrackedPoint(int i) {

   return status[i] &&  //status is false if unable to track point
i
           // if point has moved
           (abs(points[0][i].x-points[1][i].x)+
                (abs(points[0][i].y-points[1][i].y))>2);
  }
```

8. Finally, the fourth method handles the tracked feature points by drawing all the tracked points with a line that joins them to their initial position (that is, the position where they were detected the first time) on the current frame:

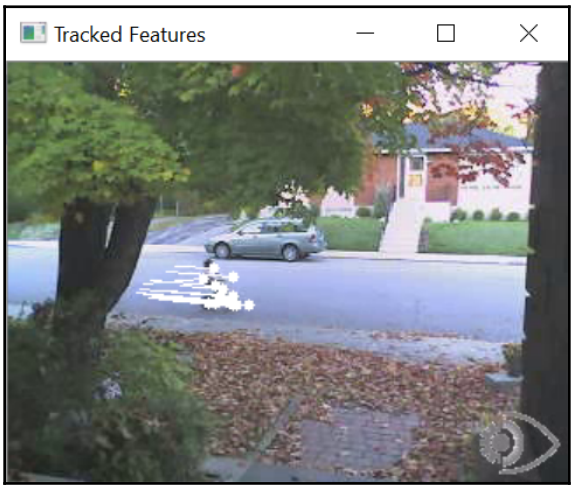
```
  // handle the currently tracked points
  void handleTrackedPoints(cv:: Mat &frame, cv:: Mat &output) {

    // for all tracked points
    for (int i= 0; i < points[1].size(); i++ ) {

      // draw line and circle
      cv::line(output, initial[i],  // initial position
               points[1][i],        // new position
               cv::Scalar(255,255,255));
               cv::circle(output, points[1][i], 3,
                          cv::Scalar(255,255,255),-1);
    }
  }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. A simple `main` function to track the feature points in a video sequence will then be written, as follows:

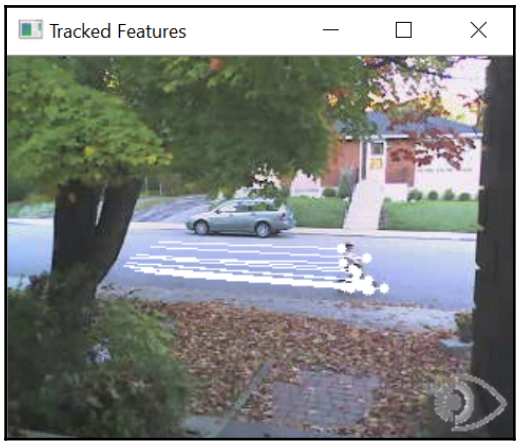
```
int main() {
  // Create video procesor instance
  VideoProcessor processor;

  // Create feature tracker instance
  FeatureTracker tracker;
  // Open video file
  processor.setInput("bike.avi");

  // set frame processor
  processor.setFrameProcessor(&tracker);

  // Declare a window to display the video
  processor.displayOutput("Tracked Features");

  // Play the video at the original frame rate
  processor.setDelay(1000./processor.getFrameRate());

  //Start the process
  processor.run();
}
```

The resulting program will show you the evolution of the moving tracked features over time. Here, for example, are two such frames at two different instants. In this video, the camera is fixed; the young cyclist is, therefore, the only moving object. Here is the result that is obtained after a few frames have been processed:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

A few seconds later, we obtain the following frame:



Now, let's go behind the scenes to understand the code better.

How it works...

To track the feature points from frame to frame, we must locate the new position of a feature point in the subsequent frame. If we assume that the intensity of the feature point does not change from one frame to the next one, then we are looking for a displacement of (u, v), as follows:

$$I_t(x, y) = I_{t+1}(x + u, y + v)$$

Here, I_t and I_{t+1} are the current frame and the following frame, respectively. This intensity-constant assumption generally holds for small displacements in images that are taken at two respective instants. We can then use the Taylor expansion in order to approximate this equation by using an equation that involves the image derivatives:

$$I_{t+1}(x + u, y + v) \approx I_t(x, y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t}$$

This latter equation leads us to another equation (as a consequence of the constant intensity assumption that cancels the two intensity terms):

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v = -\frac{\partial I}{\partial t}$$

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

This constraint is the fundamental **optical flow** constraint equation and is also known as the **brightness constancy equation**.

This constraint is exploited by the **Lukas-Kanade feature tracking** algorithm. In addition to using this constraint, the Lukas-Kanade algorithm also makes an assumption that the displacement of all the points in the neighborhood of the feature point is the same. We can, therefore, impose the optical flow constraint on all of these points using a unique (u, v) unknown displacement. This gives us more equations than the number of unknowns (two) and, therefore, we can solve this system of equations in a mean-square sense. In practice, it is solved iteratively, and the OpenCV implementation also offers us the possibility to perform this estimation at a different resolution in order to make the search more efficient and more tolerant to a larger displacement. By default, the number of image levels is `3` and the window size is `15`. These parameters can, of course, be changed. You can also specify the termination criteria, which define the conditions that stop the iterative search. The sixth parameter of `cv::calcOpticalFlowPyrLK` contains the residual mean squared error that can be used to assess the quality of the tracking. The fifth parameter contains binary flags that tell us whether tracking the corresponding point was considered successful or not.

The preceding description represents the basic principles behind the Lukas-Kanade tracker. The current implementation contains other optimizations and improvements that make the algorithm more efficient in computing the displacement of a large number of feature points.

See also

- `Chapter 8`, *Detecting Interest Points*, particularly the discussion on feature point detection.
- The *Tracking an object in a video* recipe of this chapter uses feature point tracking in order to track objects.
- The classic article by B. Lucas and T. Kanade, *An Iterative Image Registration Technique with an Application to Stereo Vision*, at the *Int. Joint Conference in Artificial Intelligence*, pp. 674-679, 1981, describes the original feature point tracking algorithm.
- The article by J. Shi and C. Tomasi, *Good Features to Track*, at the *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 593-600, 1994, describes an improved version of the original feature point tracking algorithm.

We've successfully learned how to track feature points in a video. Now let's move on to the next recipe!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Estimating the optical flow

When a scene is observed by a camera, the observed brightness pattern is projected on the image sensor and thus forms an image. In a video sequence, we are often interested in capturing the motion pattern, that is, the projection of the 3D motion of the different scene elements on an image plane. This image of projected 3D motion vectors is called the **motion field**. However, it is not possible to directly measure the 3D motion of scene points from a camera sensor. All we observe is a brightness pattern that is in motion from frame to frame. This apparent motion of the brightness pattern is called the **optical flow**. You might think that the motion field and optical flow should be equal, but this is not always true. An obvious case can be the observation of a uniform object; for example, if a camera moves in front of a white wall, then no optical flow is generated. Another classical example is the illusion produced by a rotating barber pole:

In this case, the motion field should show motion vectors in the horizontal direction as the vertical cylinder rotates around its main axis. However, observers perceive this motion as red and blue strips moving up, and this is what the optical flow will show. In spite of these differences, the optical flow is considered to be a valid approximation of the motion field. This recipe will explain how the optical flow of an image sequence can be estimated.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Getting ready

Estimating the optical flow means quantifying the apparent motion of the brightness pattern in an image sequence. So, let's consider one frame of the video at one given instant. If we look at one particular pixel of (x, y) on the current frame, then we want to know where this point is moving to in the subsequent frames. This means that the coordinates of this point are moving over time—a fact that can be expressed as ($x(t)$, $y(t)$)—and our goal is to estimate the velocity of this point (dx/dt, dy/dt). The brightness of this particular point at a given time, t, can be obtained by looking at the corresponding frame of the sequence, that is, $I(x(t), y(t), t)$. From our **image brightness constancy** assumption, we can write that the brightness of this point does not vary with respect to time as follows:

$$\frac{dI(x(t), y(t), t)}{dt} = 0$$

The chain rule allows us to write the following equation:

$$\frac{dI}{dx}\frac{dx}{dt} + \frac{dI}{dy}\frac{dy}{dt} + \frac{dI}{dt} = 0$$

This equation is known as the brightness constancy equation and it relates the optical flow components (that is, the derivatives of x and y with respect to time) with the image derivatives. This is exactly the same equation that we derived in the previous recipe; we simply demonstrated it differently.

This single equation (which is composed of two unknowns) is, however, insufficient for computing the optical flow at a pixel location. We, therefore, need to add an additional constraint. A common choice is to assume the smoothness of the optical flow, which means that the neighboring optical flow vectors should be similar. Any departure from this assumption should, therefore, be penalized. One particular formulation for this constraint is based on the Laplacian of the optical flow:

$$\frac{\partial^2}{\partial x^2}\frac{dx}{dt} + \frac{\partial^2}{\partial y^2}\frac{dy}{dt}$$

The objective is, therefore, to find the optical flow field that minimizes both the deviations from the brightness constancy equation and the Laplacian of the flow vectors.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Several approaches have been proposed to solve the dense optical flow estimation problem, and OpenCV implements a few of them. Let's use the `cv::DualTVL1OpticalFlow` class that is built as a subclass of the generic `cv::Algorithm` base class. The following steps help us to visualize the optical flow:

1. Following the implemented pattern, the first thing to do is to create an instance of this class and obtain a pointer for it:

```
//Create the optical flow algorithm
cv::Ptr<cv::DualTVL1OpticalFlow> tvl1 =
                          cv::createOptFlow_DualTVL1();
```

2. Since the object that we just created is in a ready-to-use state, we simply call the method that calculates an optical flow field between the two frames:

```
cv::Mat oflow;   // image of 2D flow vectors
//compute optical flow between frame1 and frame2
tvl1->calc(frame1, frame2, oflow);
```

   The result is an image of 2D vectors (`cv::Point`) that represents the displacement of each pixel between the two frames. In order to display the result, we must, therefore, show these vectors. This is why we created a function that generates an image map for an optical flow field.

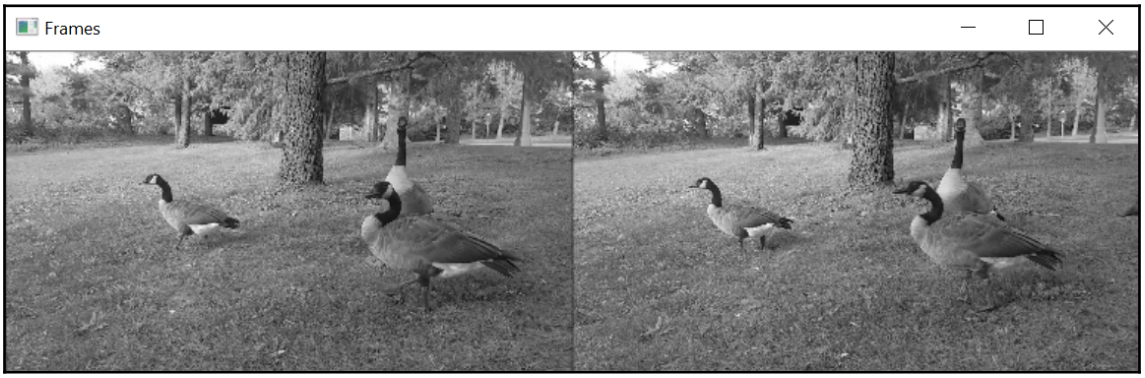
3. To control the visibility of the vectors, we used two parameters. The first one is a `stride` value that is defined so that only one vector over a certain number of pixels will be displayed. This `stride` value makes space for the display of the vectors. The second parameter is a `scale` factor that extends the vector length to make it more apparent. Each drawn optical flow vector is then a simple line that ends with a plain circle to symbolize the tip of an arrow. Our mapping function is, therefore, as follows:

```
// Drawing optical flow vectors on an image
void drawOpticalFlow(const cv::Mat& oflow,  // the optical flow
  cv::Mat& flowImage,      // the produced image
  int stride,  // the stride for displaying the vectors
  float scale, // multiplying factor for the vectors
  const cv::Scalar& color) // the color of the vectors
{
  // create the image if required
  if (flowImage.size() != oflow.size()) {
    flowImage.create(oflow.size(), CV_8UC3);
    flowImage = cv::Vec3i(255,255,255);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    }

    //for all vectors using stride as a step
    for (int y = 0; y < oflow.rows; y += stride)
      for (int x = 0; x < oflow.cols; x += stride) {
        //gets the vector
        cv::Point2f vector = oflow.at< cv::Point2f>(y, x);
        // draw the line
        cv::line(flowImage, cv::Point(x,y),
                 cv::Point(static_cast<int>(
                                  x + scale*vector.x + 0.5),
                           static_cast<int>(
                                  y + scale*vector.y + 0.5)), color);
        // draw the arrow tip
        cv::circle(flowImage,
                   cv::Point(static_cast<int>(
                                  x + scale*vector.x + 0.5),
                             static_cast<int>(
                                  y + scale*vector.y + 0.5)),
                    1, color, -1);
      }
}
```

4. Now consider the following two frames:



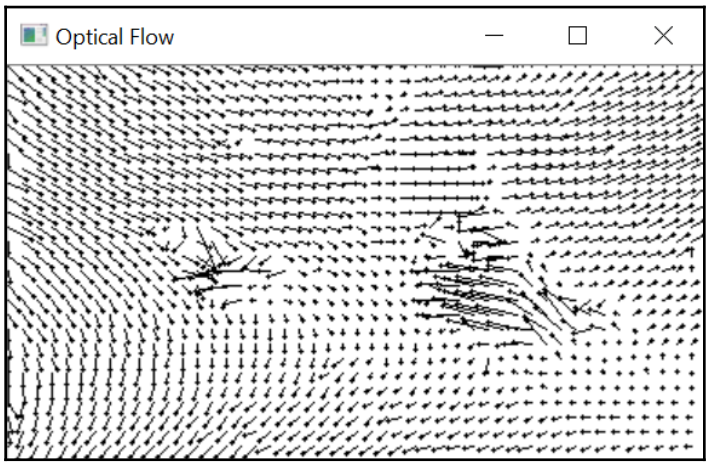
5. If these frames are used, then the estimated optical flow field can be visualized by calling our drawing function:

```
// Draw the optical flow image
cv::Mat flowImage;
drawOpticalFlow(oflow,     // input flow vectors
                flowImage, // image to be generated
                8,         // display vectors every 8 pixels
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
                                2,         // multiply size of vectors by 2
                                cv::Scalar(0, 0, 0)); // vector color
```

The result is as follows:



Now, let's go behind the scenes in order to understand the code better.

How it works...

In the first section of this recipe, we explained that an optical flow field can be estimated by minimizing a function that combines the brightness constancy constraint and a smoothness function. The equations we then presented constitute the classical formulation of the problem, and it has been improved in many ways.

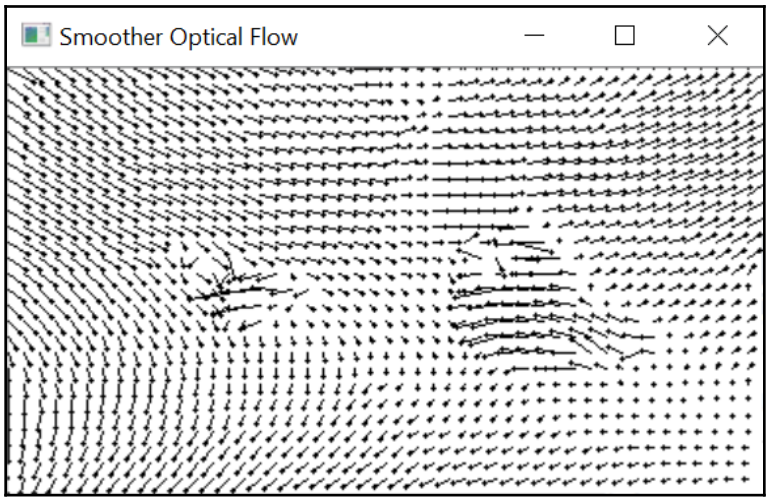
The method that we used in the previous section is known as the `Dual TV L1` method. It has two main ingredients; the first one is the use of a smoothing constraint that aims at minimizing the absolute value of the optical flow gradient (instead of the square of it). This choice reduces the impact of the smoothing term, especially at regions of discontinuity where, for example, the optical flow vectors of a moving object are quite different from the ones in the background. The second ingredient is the use of a **first-order Taylor approximation**; this linearizes the formulation of the brightness constancy constraint. We will not enter into the details of this formulation here; it suffices to say that this linearization facilitates the iterative estimation of the optical flow field. However, since the linear approximation is only valid for small displacements, the method requires a coarse-to-fine estimation scheme.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this recipe, we used the `Dual TV L1` method with its default parameters. A number of setters and getters methods allow you to modify the parameters that can have an impact on the quality of the solution and on the speed of computation. For example, you can modify the number of scales used in the pyramidal estimation or specify a, more or less, strict stopping criterion to be adopted during each iterative estimation step. Another important parameter is the weight that is associated with the brightness constancy constraint versus the smoothness constraint. For example, if we reduce the importance that is given to the brightness constancy constraint by two, then we can obtain a smoother optical flow field:

```
// compute a smoother optical flow between 2 frames
tvl1->setLambda(0.075);
tvl1->calc(frame1, frame2, oflow);
```

The following screenshot displays the output of the preceding code:



The next section provides some additional information that you can take a look at later.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The article by B.K.P. Horn and B.G. Schunck, *Determining Optical Flow, in Artificial Intelligence*, 1981, is the classical reference on optical flow estimation.
- The article by C. Zach, T. Pock, and H. Bischof, *A Duality Based Approach for Realtime TV-L1 Optical Flow*, at *IEEE Conference on Computer Vision and Pattern Recognition*, 2007, describes the details of the `Dual TV-L1` method.

We've successfully learned how to estimate optical flow. Now let's move on to the next recipe!

Tracking an object in a video

In the previous two recipes, we learned how to track the motion of points and pixels in an image sequence. In many applications, however, the usual requirement is to track a specific moving object in a video. An object of interest is first identified, and then it must be followed over a long sequence. This is challenging because, as it evolves in the scene, the image of this object will undergo many changes in appearance due to viewpoint and illumination variations, non-rigid motion, occlusion, and more.

This recipe presents some of the object-tracking algorithms implemented in the OpenCV library. These implementations are based on a common framework, which facilitates the substitution of one method by another. Contributors have also made a number of new methods available. Note that we have already presented a solution to the object-tracking problem in the *Counting pixels with integral images* recipe in `Chapter 4`, *Counting the Pixels with Histograms*; this method was based on the use of histograms computed through integral images.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

The visual object-tracking problem generally assumes that no prior knowledge about the objects to be tracked is available. Tracking is, therefore, initiated by identifying the object in a frame, and tracking must start at this point. The initial identification of the object is achieved by specifying a bounding box inside which the target is inscribed. The objective of the tracker module is to reidentify this object in a subsequent frame.

The `cv::Tracker` class of OpenCV that defines the object-tracking framework has, therefore, two `main` methods. The first one is the `init` method that is used to define the initial target bounding box. The second one is the `update` method that outputs a new bounding box, given a new frame. Both methods accept a frame (that is, a `cv::Mat` instance) and a bounding box (that is, a `cv::Rect2D` instance) as arguments; in the first method, the bounding box is an input parameter, while for the second method, the bounding box is an output parameter. Let's take a look at the following steps:

1. In order to test one of the proposed object tracker algorithms, we use the video-processing framework that has been presented in the previous chapter. In particular, we define a frame-processing subclass that will be called by our `VideoProcessor` class when each frame of the image sequence is received. This subclass has the following attributes:

```
class VisualTracker : public FrameProcessor {

  cv::Ptr<cv::Tracker> tracker;
  cv::Rect2d box;
  bool reset;

  public:
  // constructor specifying the tracker to be used
  VisualTracker(cv::Ptr<cv::Tracker> tracker) :
                    reset(true), tracker(tracker) {}
```

2. The `reset` attribute is set to `true` whenever the tracker has been reinitiated through the specification of a new target's bounding box; it is the `setBoundingBox` method that is used to store a new object position:

```
// set the bounding box to initiate tracking
void setBoundingBox(const cv::Rect2d& bb) {
  box = bb;
  reset = true;
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The `callback` method that is used to process each frame simply calls the appropriate method of the tracker, and then draws the new computed bounding box on the frame that is to be displayed:

```
// callback processing method
void process(cv:: Mat &frame, cv:: Mat &output) {

  if (reset) { // new tracking session
    reset = false;
    tracker->init(frame, box);

  } else {
    // update the target's position
    tracker->update(frame, box);
  }

  // draw bounding box on current frame
  frame.copyTo(output);
  cv::rectangle(output, box, cv::Scalar(255, 255, 255), 2);
}
```

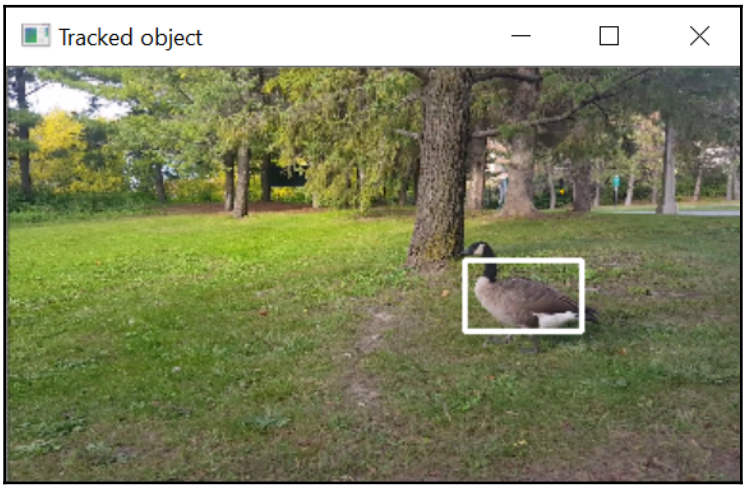
4. To demonstrate how an object can be tracked using the `VideoProcessor` and `FrameProcessor` instances, we use the **median flow tracker**, which is defined in OpenCV:

```
int main(){
  // Create video procesor instance
  VideoProcessor processor;

  // generate the filename
  std::vector<std::string> imgs;
  std::string prefix = "goose/goose";
  std::string ext = ".bmp";

  // Add the image names to be used for tracking
  for (long i = 130; i < 317; i++) {

    std::string name(prefix);
    std::ostringstream ss; ss << std::setfill('0') <<
                 std::setw(3) << i; name += ss.str();
    name += ext;
    imgs.push_back(name);
  }

  // Create feature tracker instance
  VisualTracker tracker(cv::TrackerMedianFlow::createTracker());
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    // Open video file
    processor.setInput(imgs);

    // set frame processor
    processor.setFrameProcessor(&tracker);

    // Declare a window to display the video
    processor.displayOutput("Tracked object");

    // Define the frame rate for display
    processor.setDelay(50);

    // Specify the original target position
    tracker.setBoundingBox(cv::Rect(290,100,65,40));

    // Start the tracking
    processor.run();
}
```

5. The first bounding box identifies one goose in our test image sequence; this is then automatically tracked in the subsequent frames:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Unfortunately, as the sequence progresses, the tracker will unavoidably make errors. The accumulation of these small errors will cause the tracker to slowly drift from the real target position. Here, for example, is the estimated position of our target after 130 frames have been processed:

7. Eventually, the tracker will lose track of the object. The ability of a tracker to follow an object over a long period of time is the most important criterion that characterizes the performance of an object tracker.

Now, let's go behind the scenes to understand the code better.

How it works...

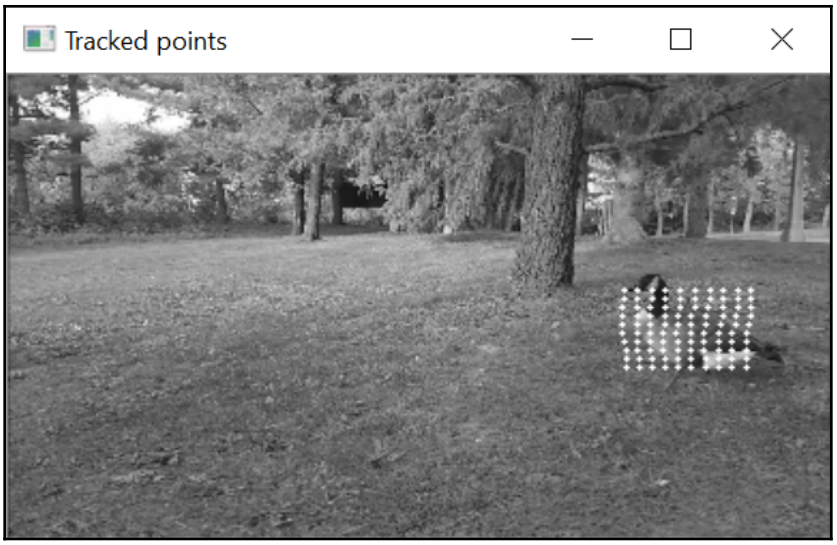
In this recipe, we showed how the generic `cv::Tracker` class can be used to track an object in an image sequence. We selected the median flow tracker algorithm to illustrate the tracking result. This is a simple, but effective, method to track a textured object, as long as its motion is not too rapid and it is not too severely occluded.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The median flow tracker is based on feature point tracking. It first starts by defining a grid of points over the object to be tracked. You could, instead, choose to detect interest points on the object using, for instance, the FAST operator presented in `Chapter 8`, *Detecting Interest Points*. However, using points at predefined locations has a number of advantages: it saves time by avoiding the computation of interest points; it guarantees that a sufficient number of points will be available for tracking; and it also makes sure that these points will be well distributed over the whole object. The median flow implementation uses, by default, a grid of *10* x *10* points, as follows:

The next step is to use the Lukas-Kanade feature-tracking algorithm presented in the first recipe of this chapter (*Tracing feature points in a video*). Here, each point of the grid is then tracked over the next frame:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



The median flow algorithm then estimates the errors that were made when tracking these points. These errors can be estimated, for example, by computing the sum of absolute pixel difference in a window around the point at its initial and tracked position. This is the type of error that is conveniently computed and returned by the `cv::calcOpticalFlowPyrLK` function. Another error measure proposed by the median flow algorithm is to use the forward-backward error. After the points have been tracked between one frame and the next, these points, at their new position, are backward-tracked to check whether they will return to their original position in the initial image. The difference between the newly-obtained forward-backward position and the initial one is the error in tracking.

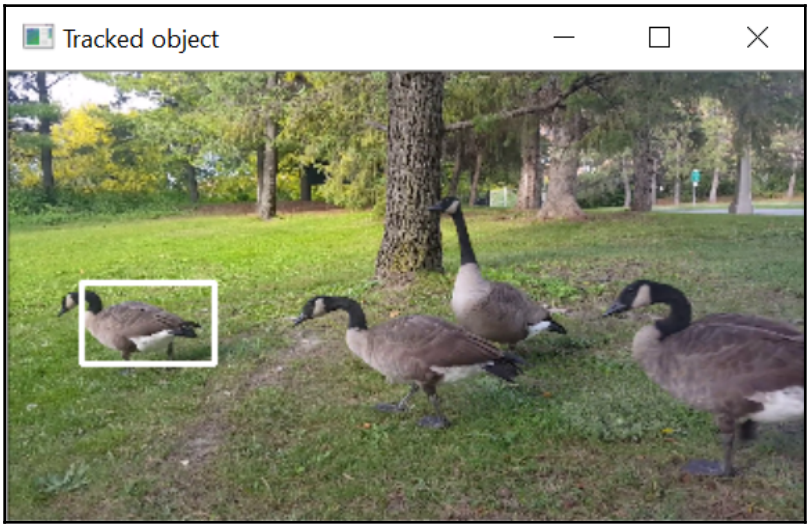
Once the tracking error of each point has been computed, only 50 percent of the points that have the smallest error is considered. This group of points is then used to compute the new position of the bounding box in the next image. Each of these points votes for a displacement value and the median of these possible displacements is retained. For the change in scale, the points are considered in pairs. The ratio of the distance between the two points in the initial frame and the next one is estimated. Again, it is the median of these scales that is finally applied.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The median flow tracker is one of many other visual object trackers based on feature point tracking. Another family of solutions is the one that is based on template matching, a concept we discussed in the *Matching local templates* recipe in `Chapter 9`, *Describing and Matching Interest Points*. A good representative of these kinds of approaches is the **Kernelized Correlation Filter** (**KCF**) algorithm, which is implemented as the `cv::TrackerKCF` class in OpenCV:

```
VisualTracker tracker(cv::TrackerKCF::createTracker());
```

Essentially, this algorithm uses the target's bounding box as a template to search for the new object position in the next view. This is normally computed through a simple correlation, but KCF uses a special trick based on the Fourier transform that we briefly mentioned in the introduction of `Chapter 6`, *Filtering the Images*. Without entering into any details, the signal-processing theory tells us that correlating a template over an image corresponds to a simple image multiplication in the frequency domain. This considerably speeds up the identification of the matching window in the next frame and makes KCF one of the fastest and more robust trackers. As an example, here is the position of the bounding box after tracking of 130 frames using KCF:



The next section provides some additional information that you can take a look at later.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

See also

- The article by Z. Kalal, K. Mikolajczyk, and J. Matas, *Forward-Backward Error: Automatic Detection of Tracking Failures*, in *Int. Conf. on Pattern Recognition*, 2010, describes the median flow algorithm
- The article by Z. Kalal, K. Mikolajczyk, and J. Matas, *Tracking-Learning-Detection*, in *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 34, No. 7*, 2012, is an advanced tracking method that uses the median flow algorithm
- The article by J.F. Henriques, R. Caseiro, P. Martins, and J. Batista, *High-Speed Tracking with Kernelized Correlation Filters*, in *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 37, No. 3*, 2014, describes the KCF tracker algorithm

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

14
Learning from Examples

Nowadays, machine learning is very often used to solve difficult machine vision problems. In fact, it is a rich field of research encompassing many important concepts that would deserve an entire cookbook by themselves. This chapter surveys some of the main machine learning techniques and explains how these can be deployed in computer vision systems using OpenCV.

At the core of machine learning is the development of computer systems that can learn, by themselves, how to react to data inputs. Instead of being explicitly programmed, machine learning systems automatically adapt and evolve when examples of desired behaviors are presented to them. Once a successful training phase is completed, it is expected that the trained system will output the correct response to new unseen queries.

Machine learning can solve many types of problems; however, our focus here will be on classification problems. Usually, in order to build a classifier that can recognize instances of a specific class of concepts, this one must be trained with a large set of annotated samples. In a 2-class problem, this set will be made of positive samples representing instances of the class to be learned, and of negative samples made of counter-examples of instances not belonging to the class of interest. From these observations, a decision function predicting the correct class of any input instances has to be learned.

In computer vision, these samples are images (or video segments). The first thing to do, therefore, is to find a representation that will ideally describe the content of each image in a compact and distinctive way. One simplistic representation could be to use a fixed-size thumbnail image. The row-by-row succession of the pixels of this thumbnail image forms a vector that can then be used as a training sample presented to a machine learning algorithm. Other alternatives, and probably more effective representations can also be used. The recipes of this chapter describe different image representations, and introduce some well-known machine learning algorithms. We should emphasize that we will not be able to cover in detail all of the theoretical aspects of the different machine learning techniques, which are discussed in the recipes; our objective is rather to present the main principles that govern their functioning.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this chapter, we will cover the following recipes:

- Recognizing faces using the nearest neighbors of local binary patterns
- Finding objects and faces using a cascade of Haar features
- Detecting objects and people using **Support Vector Machines** (**SVMs**) and histograms of oriented gradients

Recognizing faces using the nearest neighbors of local binary patterns

Our first exploration of machine learning techniques will start with what is probably the simplest approach, namely, **nearest neighbor classification**. We will also present the **local binary pattern** (**LBP**) feature, which is a popular representation encoding the textural patterns and contours of an image in a contrasting and unique way.

Our illustrative example will concern the face recognition problem. This is a very challenging problem that has been the object of numerous researches over the past 20 years. The basic solution we present here is one of the face recognition methods that is implemented in OpenCV. You will quickly realize that this solution is not very robust and works only under very favorable conditions. Nevertheless, this approach constitutes an excellent introduction to machine learning and to the face recognition problem.

How to do it...

The OpenCV library proposes a number of face recognition methods implemented as a subclass of the generic `cv::face::FaceRecognizer` class. In this recipe, we will examine the `cv::face::LBPHFaceRecognizer` class, which is interesting to us because it is based on a simple, but often very effective, classification approach, that is, the nearest neighbor classifier.

Moreover, the image representation it uses is built from the LBP feature, which is a very popular way of describing image patterns. Let's take a look at the following steps:

1. In order to create an instance of the `cv::face::LBPHFaceRecognizer` class, its static `create` method is called as follows:

```
cv::Ptr<cv::face::FaceRecognizer> recognizer =
cv::face::LBPHFaceRecognizer::create(1,     // radius of LBP
pattern
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    8,      // the number of neighboring pixels to consider
    8, 8,   // grid size
    200.8); // minimum distance to nearest neighbor
```

2. As will be explained in the next section, the first two arguments that are provided serve to describe the characteristic of the LBP feature to be used. The next step is to feed the recognizer with a number of reference face images. This is done by providing two vectors: one containing the face images, and the other one containing the associated labels. Each label is an arbitrary integer value identifying a particular individual. The idea is to train the recognizer by showing it different images of each of the people to be recognized. As you may imagine, the more representative the images you provide are, then the better the chances that the correct person will be identified. In our very simplistic example, we simply provide two images of two reference persons. Here, the `train` method is the one to call:

```
// vectors of reference image and their labels
std::vector<cv::Mat> referenceImages;
std::vector<int> labels;
// open the reference images
referenceImages.push_back(
     cv::imread("face0_1.png", cv::IMREAD_GRAYSCALE));
labels.push_back(0); //person 0
referenceImages.push_back(
     cv::imread("face0_2.png", cv::IMREAD_GRAYSCALE));
labels.push_back(0); //person 0
referenceImages.push_back(
     cv::imread("face1_1.png", cv::IMREAD_GRAYSCALE));
labels.push_back(1); //person 1
referenceImages.push_back(
     cv::imread("face1_2.png", cv::IMREAD_GRAYSCALE));
labels.push_back(1); //person 1

// train the recognizer by computing the LBPHs
recognizer->train(referenceImages, labels);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

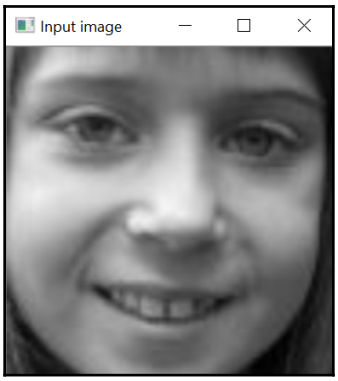
3. The images used are shown in the following screenshot, with the top row being images of person `0`, and the second row of images being person `1`:

4. The quality of these reference images is also very important. In addition, it is a good idea to have them normalized, that is, to have the main facial features appear at standardized locations. For example, having the tip of the nose located in the middle of the image, and the two eyes horizontally aligned at a specific image row. Facial feature detection methods exist that can be used to automatically normalize facial images in this way. This was not done in our example, and the robustness of the recognizer will suffer from this. Nevertheless, this one is ready to be used, an input image can be provided, and it will try to predict the label to which this face image corresponds:

```
// predict the label of this image
recognizer->predict(inputImage,      // face image
  predictedLabel,  // predicted label of this image
  confidence);     // confidence of the prediction
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Our input image is displayed in the following screenshot:



Not only does the recognizer return the predicted label, but it also returns a confidence score. In the case of the `cv::face::LBPHFaceRecognizer` class, the lower this confidence value is, then the more confident the recognizer is of its prediction. Here, we obtain a correct label prediction (`1`) with a confidence value of `90.3`.

How it works...

In order to understand the functioning of the face recognition approach presented in this recipe, we need to explain its two main components – the image representation that is used, and the classification method that is applied.

As its name indicates, the `cv::face::LBPHFaceRecognizer` algorithm makes use of the LBP feature. This is a contrasting and unique way of describing the image patterns present in an image. It is a local representation that transforms every pixel into a binary representation by encoding the pattern of image intensities found in a neighborhood. To achieve this goal, a simple rule is applied; a local pixel is compared to each of its selected neighbors. If its value is greater than that of its neighbor, then *0* is assigned to the corresponding bit position; if not, then *1* is assigned. In its simplest and most common form, each pixel is compared to its *8* immediate neighbors, which generates an 8-bit pattern. For example, let's consider the following local pattern:

87	98	17
21	26	89
19	24	90

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Applying the rule described previously generates the following binary values:

1	1	0
0		1
0	0	1

Taking an initial position in the top-left pixel and moving clockwise, the central pixel will be replaced by the binary sequence of 11,011,000. Generating a complete 8-bit LBP image is then easily achieved by looping over all pixels of an image to produce all corresponding LBP bytes. This is accomplished by using the following function:

```
//compute the Local Binary Patterns of a gray-level image
void lbp(const cv::Mat &image, cv::Mat &result) {

  result.create(image.size(), CV_8U); //allocate if necessary

  for (int j = 1; j<image.rows - 1; j++) {
    //for all rows (except first and last)

    // pointers to the input rows
    const uchar* previous = image.ptr<const uchar>(j - 1);
    const uchar* current  = image.ptr<const uchar>(j);
    const uchar* next     = image.ptr<const uchar>(j + 1);
    uchar* output = result.ptr<uchar>(j);        //output row

    for (int i = 1; i<image.cols - 1; i++) {

      //compose local binary pattern
      *output =  previous[i - 1] > current[i] ? 1 : 0;
      *output |= previous[i] > current[i] ?     2 : 0;
      *output |= previous[i + 1] > current[i] ? 4 : 0;
      *output |= current[i - 1] > current[i] ?  8 : 0;
      *output |= current[i + 1] > current[i] ? 16 : 0;
      *output |= next[i - 1] > current[i] ?    32 : 0;
      *output |= next[i] > current[i] ?        64 : 0;
      *output |= next[i + 1] > current[i] ?   128 : 0;
      output++; // next pixel
    }
  }
  // Set the unprocess pixels to 0
  result.row(0).setTo(cv::Scalar(0));
  result.row(result.rows - 1).setTo(cv::Scalar(0));
  result.col(0).setTo(cv::Scalar(0));
  result.col(result.cols - 1).setTo(cv::Scalar(0));
}
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The body of the loop compares each pixel with its eight neighbors and the bit values are assigned through simple bit shifts:

An LBP image is obtained and can be displayed as a gray-level image, as follows:

This gray-level representation is not really interpretable, but it does illustrate the encoding process that occurred.

Returning to our `cv::face::LBPHFaceRecognizer` class, we can see that the first two parameters of the `create` method specify the size (that is, the radius in pixels) and dimension (that is, the number of pixels along the circle, possibly applying interpolation) of the neighborhood to be considered. Once the LBP image is generated, the image is divided into a grid. The size of this grid is specified as the third parameter of the `create` method.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

For each block of this grid, a histogram of LBP values is constructed. A global image representation is finally obtained by concatenating the bin counts of all these histograms into one large vector. Using an 8×8 grid, the set of computed 256-bin histograms then forms a 16384-dimensional vector.

The `train` method of the `cv::face::LBPHFaceRecognizer` class, therefore, generates this long vector for each of the reference images provided. Each face image can then be seen as a point in a very high dimensional space. When a new image is submitted to the recognizer using the `predict` method, the closest reference point to this image is found. The label associated with this point is, therefore, the predicted label, and the confidence value will be the computed distance. This is the principle that defines the nearest neighbor classifier. One more ingredient is generally added; if the nearest neighbor of the input point is too far from it, then this could mean that this point, in fact, does not belong to any of the reference classes. So, how far away must this point be to be considered as an outlier? Well, this is specified by the fourth parameter of the `create` method of the `cv::face::LBPHFaceRecognizer` class.

As you can see, this is a very simple idea and it turns out to be very effective when the different classes generate distinct clouds of points in the representational space. Another benefit of this approach is that the method implicitly handles multiple classes, as it simply reads the predicted class from its nearest neighbors. The main drawback is its computational cost. Finding the nearest neighbor in such a large space, possibly composed of many reference points, can take time. Storing all these reference points is also costly in memory.

See also

- The article by T. Ahonen, A. Hadid, and M. Pietikainen, *Face Description with Local Binary Patterns: Application to Face Recognition* in *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 2006, describes the use of LBP for face recognition.
- The article by B. Froba and A. Ernst, *Face Detection with the Modified Census Transform* in *IEEE Conference on Automatic Face and Gesture Recognition*, 2004, proposes a variant of the LBP feature.
- The article by M. Uricar, V. Franc, and V. Hlavac, *Detector of Facial Landmarks Learned by the Structured Output SVM* in *International Conference on Computer Vision Theory and Applications*, 2012, describes a facial feature detector that is based on the SVMs discussed in the last recipe of this chapter.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've successfully learned how to recognize faces using the nearest neighbors of local binary patterns. Now, let's move on to the next recipe!

Finding objects and faces with a cascade of Haar features

In the previous recipe, we learned about some of the basic concepts of machine learning. We demonstrated how a classifier can be built by collecting samples of the different classes of interest. However, for the approach that was considered in this previous recipe, training a classifier simply consists of storing all the samples' representations. From there, the label of any new instance can be predicted by looking at the closest (nearest neighbor) labeled point. For most machine learning methods, training is a relatively iterative process, during which machinery is built by looping over the samples. The performance of the classifier produced gradually improves as more samples are presented. Learning eventually stops when a certain performance criterion is reached, or when no more improvements can be obtained from the current training dataset. This recipe will present a machine learning algorithm that follows this procedure, that is, the cascade of boosted classifiers.

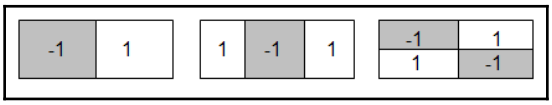
However, before we look at this classifier, we will first turn our attention to the Haar feature image representation. Now, we understand that a good representation is an essential ingredient in the production of a robust classifier. LBPs, as described in the previous *Recognizing faces using the nearest neighbors of local binary patterns* recipe, constitute one possible choice; the next section describes another popular representation.

Getting ready

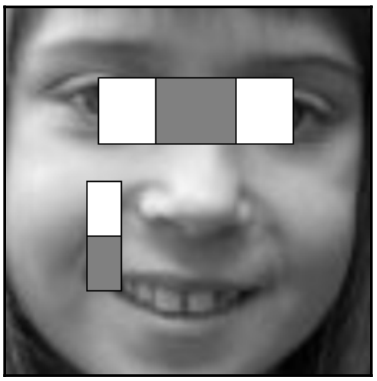
The first step in the generation of a classifier is to assemble a (preferably) large collection of image samples showing different instances of the classes of objects to be identified. The way in which these samples are represented has been shown to have an important impact on the performance of the classifier that is to be built from them. Pixel-level representations are generally considered to be too low-level to robustly describe the intrinsic characteristics of each class of objects. Instead, representations that can describe, at various scales, the distinctive patterns present in an image are preferable. This is the objective of Haar features, which are also called **Haar-like** features because they derive from Haar transform basis functions.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Haar features define small rectangular areas of pixels, which are later compared through simple subtractions. Three different configurations are generally considered, namely, the 2-rectangle, the 3-rectangle, and the 4-rectangle features; this is demonstrated in the following diagram:



These features can be of any size and applied to any area of the image to be represented. For example, here are two Haar features applied to an image of a face:



Building a Haar representation consists of selecting a number of Haar features of any given type, size, and location, and applying them to images. The specific set of values obtained from the chosen set of Haar features constitutes the image representation. The challenge is to then determine which set of features to select. Indeed, to distinguish one class of objects from another, some Haar features must be more relevant than others. For example, in the case of the class of face images, applying a 3-rectangle Haar feature between the eyes (as shown in the preceding screenshot) could be a good idea, as we expect all face images to consistently produce a high value. Of course, since there are hundreds of thousands of possible Haar features in existence, it would certainly be difficult to manually make a good selection. So, we need a machine learning method that will select the most relevant features for a given class of objects.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

In this recipe, we will learn how we can build, using OpenCV, a boosted cascade of features to produce a 2-class classifier. However, before we do, let's explain the terminology that is used here. A 2-class classifier is one that can identify the instances of one class (for example, face images) from the rest (for example, images that do not contain faces). In this case, therefore, we have the *positive samples* (that is, face images) and the *negative samples* (that is, non-face images); these latter images are also called background images. The classifier of this recipe will be made of a cascade of simple classifiers that will be sequentially applied. Each stage of the cascade will make a quick decision about rejecting, or not, the object shown based on the values obtained for a small subset of features. This cascade is boosted in the sense that each stage improves (or boosts) the performance of the previous ones by making more accurate decisions. The main advantage of this approach is that the early stages of the cascade are composed of simple tests that can then quickly reject instances that do not belong to the class of interest. These early rejections make the cascade classifier quicker because, when searching for a class of object by scanning an image, most subwindows to be tested will not belong to the class of interest. In this way, only a few windows will have to pass through all the stages before being accepted or rejected, as described in the following steps:

1. In order to train a boosted classifier cascade for a specific class, OpenCV offers a software tool that will perform all the required operations. When you install the library, you should have two executable modules created and located in the appropriate `bin` directory – these are `opencv_createsamples.exe` and `opencv_traincascade.exe`. Make sure your system `PATH` points to this directory so that you can execute these tools from anywhere.
   When training a classifier, the first thing to do is to collect the samples. The positive ones are made of images showing instances of the target class. In our simple example, we decided to train a classifier to recognize stop signs; here are the few positive samples we collected:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. The list of the positive samples to be used must be specified in a text file, which we have named `stop.txt`. It contains image filenames and bounding box coordinates:

```
stop00.png 1 0 0 64 64
stop01.png 1 0 0 64 64
stop02.png 1 0 0 64 64
stop03.png 1 0 0 64 64
stop04.png 1 0 0 64 64
stop05.png 1 0 0 64 64
stop06.png 1 0 0 64 64
stop07.png 1 0 0 64 64
```

3. The first number after the filename is the number of positive samples visible in the image. Next, is the upper-left coordinate of the bounding box containing this positive sample and, finally, its width and height. In our case, the positive samples have already been extracted from their original images – this is why we always have one sample per file and the upper-left coordinates at *(0, 0)*. Once this file is available, you can then create the positive sample file by running the extractor tool:

```
opencv_createsamples –info stop.txt –vec stop.vec –w 24 –h 24 –num
8
```

4. This will create a `stop.vec` output file, which will contain all the positive samples specified in the input text file. Note that we made the sample size smaller (that is, 24 × 24) than the original size (that is, 64 × 64). The extractor tool resizes all the samples to the specified size. Usually, Haar features work better with smaller templates, but this is something that has to be validated on a case-by-case basis.
   The negative samples are simply background images containing no instances of the class of interest (no stop signs, in our case). However, these images should show a good variety of what the classifier is expected to see. These negative images could be of any size, since the training tool will extract random negative samples from them. Here is one example of a background image that we can use:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Once the positive and negative sample sets are in place, the classifier cascade is ready to be trained; calling the tool is done as follows:

```
opencv_traincascade  -data classifier -vec stop.vec
                     -bg neg.txt -numPos 9  -numNeg 20
                     -numStages 20 -minHitRate 0.95
                     -maxFalseAlarmRate 0.5 -w 24 -h 24
```

The parameters used here will be explained in the next section. Note that this training process can take a very long time; in a complex case involving thousands of samples, it can even take days to execute. As it runs, the cascade trainer will print out performance reports each time the training of a stage is completed. In particular, the classifier will tell you what the current **hit rate** (**HR**) is; this is the percentage of positive samples that are currently accepted by the cascade (that is, they are correctly recognized as positive instances – they are also called *true positives*). You want this number to be as close as possible to 1.0. It will also give you the current **false alarm rate** (**FAR**), which is the number of tested negative samples that are wrongly classified as positive instances (also called *false positives*). You want this number to be as close as possible to 0.0. These numbers are reported for each of the features introduced in each stage.

Our simple example took only a few seconds. The structure of the classifier produced is described in an XML file that results from the training phase. The classifier is then ready to be used! You can submit any sample to it and it will tell you whether it believes that it is a positive or a negative one.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. In our example, we trained our classifier with 24 × 24 images, but, in general, what you want is to find out whether there are any instances of your class of objects somewhere in an image (of any size). To achieve this objective, you simply have to scan the input image and extract all the possible windows of the sample size. If your classifier is accurate enough, only the windows that contain the seek objects will return a positive detection. However, this only works as long as the visible positive samples have the appropriate size. To detect instances at multiple scales, you then have to build a pyramid of images by reducing the size of the original image by a certain factor at each level of the pyramid. In this way, bigger objects will eventually fit the trained sample size as we go down the pyramid. This is a long process, but the good news is that OpenCV provides a class that implements this process. Its use is actually pretty straightforward; first, you construct the classifier by loading the appropriate XML file:

```
cv::CascadeClassifier cascade;
if (!cascade.load("stopSamples/classifier/cascade.xml")) {
  std::cout << "Error when loading the cascade classfier!"
            << std::endl;
  return -1;
}
```

7. Then, you call the `detection` method using an input image:

```
cascade.detectMultiScale(inputImage, // input image
  detections, // detection results
  1.1,        // scale reduction factor
  2,          // number of required neighbor detections
  0,          // flags (not used)
  cv::Size(48, 48),    // minimum object size to be detected
  cv::Size(128, 128)); // maximum object size to be detected
```

8. The result is provided as a vector of `cv::Rect` instances. To visualize the detection results, you just have to draw these rectangles on your input image:

```
for (int i = 0; i < detections.size(); i++)
cv::rectangle(inputImage, detections[i],
              cv::Scalar(255, 255, 255), 2);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

When our classifier is tested on an image, here is the result we obtain:

Let's take a look at how these steps work in the following section.

How it works...

In the previous section, we explained how it is possible to build an OpenCV cascade of classifiers using positive and negative samples of a class of objects. We will now examine the basic steps of the learning algorithm that is used to train this cascade. Our cascade has been trained using the Haar features that were described in the introductory section of this recipe; however, as you will see, any other simple feature can be used to build a boosted cascade. Since the theory and principles of boosted learning are pretty complex, we will not cover all of their aspects in this recipe; interested readers should refer to the articles listed in the last section.

Let's first restate that there are two core ideas behind the cascade of boosted classifiers. The first one is that a strong classifier can be built by combining together several weak classifiers (that is, those based on simple features). Secondly, in machine vision, negative instances are found much more frequently than the positive ones, so effective classification can be performed in stages. The early stages quickly reject the obvious negative instances, and more refined decisions can be made at later stages for more difficult samples. Based on these two ideas, we can now describe the boosted cascade learning algorithm. Our explanations are based on the variant of boosting called **AdaBoost**, which is the one that is most often used. Our description will also allow us to explain some of the parameters used in the `opencv_traincascade` tool.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

In this recipe, we use the Haar features in order to build our weak classifier. When one Haar feature is applied (of given type, size, and location), a value is obtained. A simple classifier is then obtained by finding the threshold value that would best classify the negative and positive class instances based on this feature value. To find this optimal threshold, we have at our disposal a number of positive and negative samples (the number of positive and negative samples to be used in this step by `opencv_traincascade` is given by the `-numPos` and `-numNeg` parameters). Since we have a large number of possible Haar features, we examine all of them and select the one that best classifies our sample set. Obviously, this very basic classifier will make errors (that is, misclassify several samples); this is why we need to build several of these classifiers. These classifiers are added iteratively, each time searching for the new Haar feature giving the best classification. But since, at each iteration, we want to focus on the samples that are currently misclassified, the classification performance is measured by giving a higher weight to the misclassified samples. A set of simple classifiers is, therefore, obtained and a strong classifier is then built from a weighted sum of these weak classifiers (that is, classifiers with better performance are given a higher weight). Following this approach, a strong classifier with good performance can be obtained by combining hundreds of simple features.

However, in order to build a cascade of classifiers in which early rejection is a central mechanism, we do not want a strong classifier made of a large number of weak classifiers. Instead, we need to find very small classifiers that will use only a handful of Haar features in order to quickly reject the obvious negative samples while keeping all the positive ones. In its classical form, AdaBoost aims to minimize the total classification error by counting the number of false negatives (that is, a positive sample that is classified as a negative one) and false positives (that is, a negative sample that is classified as a positive one). In the present case, we need to have most, if not all, the positive samples correctly classified while minimizing the false positive rate. Fortunately, it is possible to modify AdaBoost so that the true positives are rewarded more strongly. Consequently, when training each stage of a cascade, two criteria must be set – the minimum hit rate and the maximum false alarm rate; in `opencv_traincascade`, these are specified using the `-minHitRate` (0.995 is the default value) and `-maxFalseAlarmRate` (0.5 is the default value) parameters. Haar features are added to the stage until these two performance criteria are met. The minimum hit rate must be set relatively high to make sure that the positive instances will go through to the next stage; remember that if a positive instance is rejected by a stage, then this error cannot be recovered. Therefore, to facilitate the generation of a classifier of low complexity, you should set the maximum false alarm rate relatively high. Otherwise, your stage will need many Haar features in order to meet the performance criteria, which contradicts the idea of early rejection in order to compute classifier stages.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

A good cascade, therefore, will be made up of the early stages using fewer features – with the number of features per stage growing as you go up the cascade. In `opencv_traincascade`, the maximum number of features per stage is set using the `-maxWeakCount` (the default is 100) parameter, and the number of stages is set using the `-numStages` (the default is 20) parameter.

When the training of a new stage starts, new negative samples must be collected. These are extracted from the background images provided. The difficulty here is to find negative samples that pass through all the previous stages (that is, those that are wrongly classified as positives). The more stages you have trained, the more difficult it will be to collect these negative samples. This is why it is important to provide the classifier with a large variety of background images. It will then be able to extract patches from these that are difficult to classify (because they resemble the positive samples). Additionally, note that if, at a given stage, the two performance criteria are met without adding any new features, then the cascade training is stopped at this point (this means that you can use it as it is, or retrain it by providing more difficult samples). Reciprocally, if the stage is unable to meet the performance criteria, then the training will also be stopped; in this case, you should try a new training process using easier performance criteria.

With a cascade that is made of *n* stages, it can easily be shown that the global performance of the classifier will be at least better than `minHitRate`^{n} and `maxFalseAlarmRate`^{n}. This is the result of each stage being built on top of the results of the previous cascade of stages. For example, if we consider the default values of `opencv_traincascade`, we expect our classifier to have an accuracy (hit rate) of 0.995^{20} and a false alarm rate of 0.5^{20}. This means that 90% of the positive instances will be correctly identified and 0.001% of negative samples will be incorrectly classified as positive. Note that a consequence of the fact that a fraction of the positive samples will be lost as we go up the cascade is that you always have to provide more positive samples than the specified number of samples to use in each stage. In the numerical example we just gave, we need `numPos` to be set at 90% of the number of available positive samples.

One important question is *How many samples should be used for training?*. This is difficult to answer but, obviously, your positive sample set must be large enough in order to cover a wide range of possible appearances of your class instances. Your background images should also be relevant. In the case of our stop sign detector example, we included urban images as stop signs that are expected to be seen in that context. A usual rule of thumb is to have *numNeg= 2 * numPos*, but this has to be validated on your own dataset.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, we explained in this recipe how to build a cascade of classifiers using Haar features. Such features can also be built using other features, such as the LBP feature discussed in the previous recipe or the histograms of oriented gradients that will be presented in the next recipe. Here, `opencv_traincascade` has a `-featureType` parameter, allowing the selection of different feature types.

There's more...

The OpenCV library proposes a number of pretrained cascades that you can use to detect faces, facial features, people, and more. You will find these cascades in the form of XML files in the data directory of the library source directory.

Face detection with a Haar cascade

The pretrained models are ready to be used; all you have to do is create an instance of the `cv::CascadeClassifier` class using the appropriate XML file:

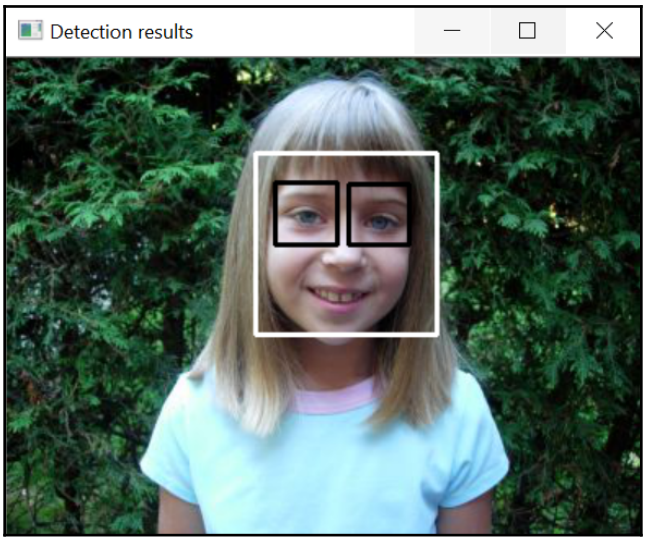
```
cv::CascadeClassifier faceCascade;
if (!faceCascade.load("haarcascade_frontalface_default.xml")) {
  std::cout << "Error when loading the face cascade classfier!"
            << std::endl;
  return -1;
}
```

Then, to detect faces with Haar features, you proceed with the following code:

```
faceCascade.detectMultiScale(picture, // input image
  detections,  // detection results
  1.1,         // scale reduction factor
  3,           // number of required neighbor detections
  0,           // flags (not used)
  cv::Size(48, 48),    // minimum object size to be detected
  cv::Size(128, 128)); // maximum object size to be detected

// draw detections on image
for (int i = 0; i < detections.size(); i++)
  cv::rectangle(picture, detections[i],
                cv::Scalar(255, 255, 255), 2);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

The same process can be repeated for an eye detector, as follows:



More information pertaining to this recipe is provided in the next section.

See also

- The *Describing and matching local intensity patterns* recipe in `Chapter 9`, *Describing and Matching Interest Points*, describes the SURF descriptor, which also uses Haar-like features.
- The *Rapid Object Detection Using a Boosted Cascade of Simple Features* article by P. Viola and M. Jones in *Computer Vision and Pattern Recognition* conference, 2001, is the classical paper that describes the cascade of boosted classifiers and Haar features.
- The *A Short Introduction to Boosting* article by Y. Freund and R.E. Schapire in *Journal of Japanese Society for Artificial Intelligence*, 1999, describes the theoretical foundations of boosting.
- The *Filtered Channel Features for Pedestrian Detection* article by S. Zhang, R. Benenson and B. Schiele in *IEEE Conf. on Computer Vision and Pattern Recognition*, 2015, presents features that are similar to Haar and that can produce highly accurate detections.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've successfully learned how to find objects and faces with a cascade of Haar features. Now, let's move on to the next recipe!

Detecting objects and people using SVMs and histograms of oriented gradients

This recipe presents another machine learning method, the SVM, which can produce accurate 2-class classifiers from training data. They have been largely used to solve many computer vision problems. This time, classification is solved by using a mathematical formula that looks at the geometry of the problem in high-dimension spaces.

In addition, we will also present a new image representation that is often used in conjunction with SVMs to produce robust object detectors.

Getting ready

Images of objects are mainly characterized by their shape and textual content. This is the aspect that is captured by the **Histogram of Oriented Gradients** (**HOG**) representation. As its name suggests, this representation is based on building histograms from image gradients. In particular, because we are more interested in shapes and textures, it is the distribution of the gradient orientations that is analyzed. In addition, in order to take into consideration the spatial distribution of these gradients, multiple histograms are computed over a grid that divides the image into regions.

The first step in building an HOG representation is, therefore, to compute the gradient of an image. The image is then subdivided into small cells (for example, 8 × 8 pixels) and histograms of gradient orientations are built for each of these cells. The range of possible orientations must, therefore, be divided into bins. Most often, only the gradient orientations are considered but not their directions (these are called unsigned gradients). In this case, the range of possible orientations is from 0 to 180 degrees. A 9-bin histogram, in this case, will divide the possible orientations into intervals of 20 degrees. Each gradient vector in a cell contributes to a bin with a weight corresponding to the magnitude of this gradient.

The cells are then grouped into blocks. A block is then made of a certain number of cells. These blocks that cover the image can overlap each other (that is, they can share cells). For example, if blocks are made up of *2* × *2* cells, then a new block can be defined every one cell; this will represent a block stride of *1* cell and each cell (except the last one in a row) will then contribute to *2* blocks. Conversely, with a block stride of *2* cells, the blocks will not overlap at all.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

A block contains a certain number of cell histograms (for example, *4* in the case of a block made up of *2 × 2* cells). These histograms are simply concatenated together to form a long vector (for example, *4* histograms of *9* bins each will produce a vector of length *36*). In order to make the representation invariant to changes, this vector is then normalized (for example, each element is divided by the magnitude of the vector). Finally, you also concatenate together all the vectors that are associated with all the blocks of the image (row order) into a very large one (for example, in a *64 × 64* image, you will have a total of seven *16 × 16* blocks when a stride of *1* is applied on cells of size *8 × 8*; this represents a final vector of *49 x 36 = 1764* dimensions). This long vector is the HOG representation of the image.

As you can see, the HOG of an image leads to a vector of very high dimension (see the *There's more...* section of this recipe that proposes a way in which to visualize an HOG representation). This vector characterizes the image and can then be used to classify images of different classes of objects. To achieve this goal, we, therefore, need a machine learning method that can handle vectors of very high dimension.

How to do it...

In this recipe, we will build another stop sign classifier. This is obviously just a toy example that serves to illustrate the learning procedure:

1. As we explained in the previous recipe, the first step is to collect samples for training. In our example, the set of positive samples that we will be using is as follows:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. And our (very small) set of negative samples is as follows:

3. We will now learn how to differentiate between these two classes using SVM, as implemented in the `cv::svm` class. To build a robust classifier, we will represent our class instances using HOG, as described in the introductory section of this recipe. More precisely, we will use *8* × *8* blocks made up of *2* × *2* cells with a block stride of *1* cell:

```
cv::HOGDescriptor hogDesc(positive.size(),  // size of the window
  cv::Size(8, 8),     // block size
  cv::Size(4, 4),     // block stride
  cv::Size(4, 4),     // cell size
  9);                 // number of bins
```

4. With 9-bin histograms and *64* × *64* samples, this configuration produces HOG vectors (made of *225* blocks) of size *8,100*. We compute this descriptor for each of our samples and then transfer them into a single matrix (one HOG per row):

```
// compute first descriptor
std::vector<float> desc;
hogDesc.compute(positives[0], desc);

// the matrix of sample descriptors
int featureSize = desc.size();
int numberOfSamples = positives.size() + negatives.size();

// create the matrix that will contain the samples HOG
cv::Mat samples(numberOfSamples, featureSize, CV_32FC1);
// fill first row with first descriptor
for (int i = 0; i < featureSize; i++)
  samples.ptr<float>(0)[i] = desc[i];

// compute descriptor of the positive samples
for (int j = 1; j < positives.size(); j++) {
  hogDesc.compute(positives[j], desc);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    // fill the next row with current descriptor
    for (int i = 0; i < featureSize; i++)
      samples.ptr<float>(j)[i] = desc[i];
  }
  // compute descriptor of the negative samples
  for (int j = 0; j < negatives.size(); j++) {
    hogDesc.compute(negatives[j], desc);
    // fill the next row with current descriptor
    for (int i = 0; i < featureSize; i++)
      samples.ptr<float>(j + positives.size())[i] = desc[i];
  }
```

5. Note how we computed the first HOG in order to obtain the size of the descriptor and then created the matrix of descriptors. A second matrix is then created to contain the labels associated with each sample. In our case, the first rows are the positive samples (and must be assigned a label of `1`), while the remaining rows are the negative samples (labeled `-1`):

```
// Create the labels
cv::Mat labels(numberOfSamples, 1, CV_32SC1);
// labels of positive samples
labels.rowRange(0, positives.size()) = 1.0;
// labels of negative samples
labels.rowRange(positives.size(), numberOfSamples) = -1.0;
```

6. The next step is to build the SVM classifier that will be used for training; we also select the type of `SVM` and the kernel to be used (these parameters will be discussed in the next section):

```
// create SVM classifier
cv::Ptr<cv::ml::SVM> svm = cv::ml::SVM::create();
svm->setType(cv::ml::SVM::C_SVC);
svm->setKernel(cv::ml::SVM::LINEAR);
```

7. We are now ready for training. The labeled samples are first provided to the classifier and the `train` method is called:

```
// prepare the training data
cv::Ptr<cv::ml::TrainData> trainingData =
 cv::ml::TrainData::create(samples,
 cv::ml::SampleTypes::ROW_SAMPLE, labels);
// SVM training
svm->train(trainingData);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Once the training phase is completed, any sample of the unknown class can be submitted to the classifier, which will try to predict the class to which it belongs (here, we test four samples):

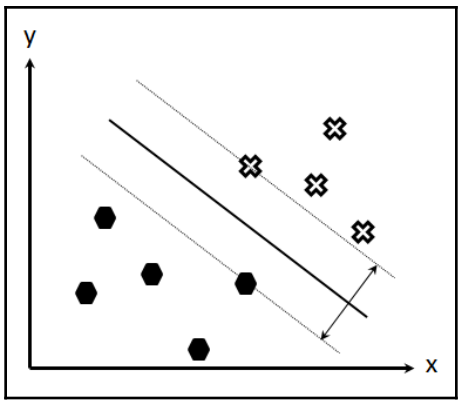
```
cv::Mat queries(4, featureSize, CV_32FC1);

// fill the rows with query descriptors
hogDesc.compute(cv::imread("stop08.png",
                           cv::IMREAD_GRAYSCALE), desc);
for (int i = 0; i < featureSize; i++)
  queries.ptr<float>(0)[i] = desc[i];
hogDesc.compute(cv::imread("stop09.png",
                           cv::IMREAD_GRAYSCALE), desc);
for (int i = 0; i < featureSize; i++)
  queries.ptr<float>(1)[i] = desc[i];
hogDesc.compute(cv::imread("neg08.png",
                           cv::IMREAD_GRAYSCALE), desc);
for (int i = 0; i < featureSize; i++)
  queries.ptr<float>(2)[i] = desc[i];
hogDesc.compute(cv::imread("neg09.png",
                           cv::IMREAD_GRAYSCALE), desc);
for (int i = 0; i < featureSize; i++)
  queries.ptr<float>(3)[i] = desc[i];
cv::Mat predictions;

// Test the classifier
svm->predict(queries, predictions);
for (int i = 0; i < 4; i++)
  std::cout << "query: " << i << ": " <<
               ((predictions.at<float>(i,) < 0.0)?
                     "Negative" : "Positive") << std::endl;
```

If the classifier has been trained with representative samples, then it should be able to correctly predict the label of a new instance.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

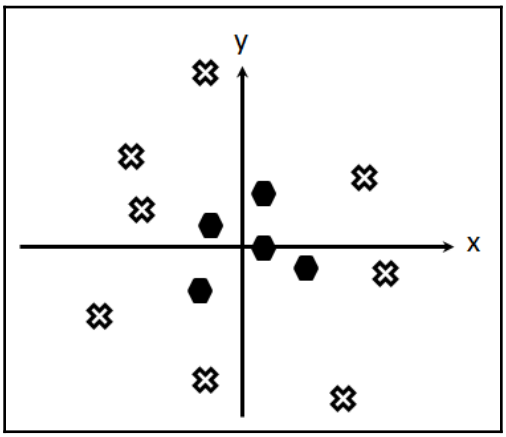
In our stop sign recognition example, each instance of our class is represented by a point in a 8100-dimensional HOG space. It is obviously impossible to visualize such a large space, but the idea behind SVMs is to trace a boundary in that space, which will segregate points that belong to one class from points belonging to another class. More specifically, this boundary will, in fact, be just a simple hyperplane. This idea is better explained considering a 2D space, where each instance is represented as a 2D point. The hyperplane is, in this case, a simple line:



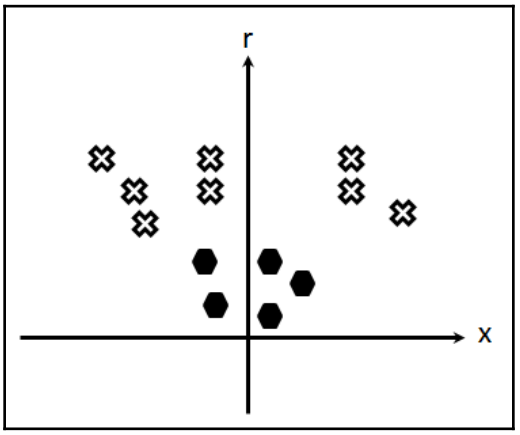
This is a trivial example but, conceptually, working in a two-dimensional space, or in a 8100-dimensional space, is the same thing. The preceding diagram demonstrates how a simple line can separate the points of the two classes. In the case illustrated here, we can see that many other lines could also achieve this perfect class separation. One question is, therefore, which exact line should we choose? To answer this question, you must first realize that the samples we used to build our classifier constitute just a small snapshot of all the possible instances that will need to be classified when the classifier is used in a target application. This means that we want our classifier to not only be able to correctly separate the provided sample set, but we also want this one to make the best decision regarding the future instances that are shown to it. This concept is often referred to as the *generalization* power of a classifier. Intuitively, it would be reasonable to believe that our separating hyperplane should be located in between the two classes, not closer to one class than the other. More formally, SVMs propose setting the hyperplane at a position that maximizes the margin around the defined boundary. This *margin* is defined as the minimum distance between the separating hyperplane and the closest point in the positive sample set, plus the distance between the hyperplane and the closest negative sample. The closest points (the ones that define the margin) are called the **support vectors**. The mathematics behind SVM defines an optimization function aimed at identifying these support vectors.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

But the proposed solution to the classification problem cannot be that simple. Consider what will happen if the distribution of the sample points is as follows:



In this case, a simple hyperplane (a line here) cannot achieve a proper separation. SVM solves this problem by introducing artificial variables that bring the problem into a higher-dimensional space through some non-linear transformations. For example, in the preceding example, you might propose to add the distance to the origin as an additional variable, that is, to compute *r= sqrt(x2+y2)* for each point. We now have a three-dimensional space; for simplicity, let's just draw the points on the *(r, x)* plane:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

As you can see, our set of sample points can now be separated by a simple hyperplane. This implies that you now have to find the support vectors in this new space. In fact, in the SVM formulation, you do not have to bring all the points into that new space; you just have to define a way to measure the point-to-hyperplane distance in that augmented space. SVM, therefore, defines *kernel functions* that allow you to measure this distance in the higher space without having to explicitly compute the point coordinates in that space. This is just a mathematical trick that explains why support vectors producing the maximal margin can be efficiently computed in very high (artificial) dimensional space. This also explains why, when you want to use SVMs, you need to specify which kernel you want to use. It is by applying these kernels that you will make non-linearly separable in order to become separable in the kernel space.

However, since with SVMs, we often work with features of very high dimension (for example, the *8,100* dimension in our HOG example), then it may be that our samples will be made separable using a simple hyperplane. This is why it makes sense to not use non-linear kernels (or, more precisely, to use a linear kernel, that is, `cv::ml::SVM::LINEAR`) and work in the original feature space. The resulting classifier will then be computationally simpler. But for more challenging classification problems, kernels remain a very effective tool. OpenCV offers you a number of standard kernels (for example, radial basis functions, and sigmoid functions); the objective of these is to send the samples into a larger non-linear space that will make the classes separable by a hyperplane. SVM has a number of variants; the most common is the C-SVM, which adds a penalty for each outlier sample that does not lie on the right side of the hyperplane.

Finally, we insist on the fact that, because of their strong mathematical foundations, SVMs work very well with features of very high dimension. In fact, they have been shown to operate best when the number of dimensions of the feature space is larger than the number of samples. They are also memory efficient, as they just have to store the support vectors (in comparison to a method such as nearest neighbors that requires keeping in memory all sample points).

There's more...

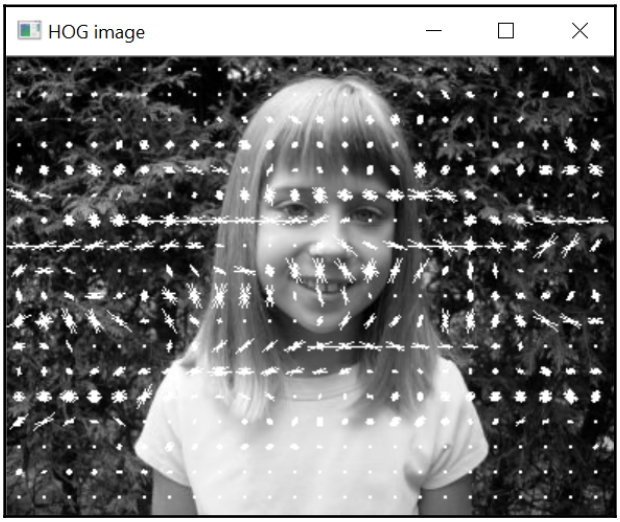
Histograms of oriented gradients and SVM form a good combination for the construction of good classifiers. One of the reasons for this success is the fact that HOG can be viewed as a robust high-dimensional descriptor that captures the essential aspects of an object class. HOG-SVM classifiers have been used successfully in many applications; pedestrian detection is one of them.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Finally, since this is the last recipe of this book, we will, therefore, end it with a perspective on a recent trend in machine learning that is revolutionizing computer vision and artificial intelligence.

HOG visualization

HOGs are built from cells that are combined in overlapping blocks. It is, therefore, difficult to visualize this descriptor. Nevertheless, they are often represented by displaying the histograms associated with each cell. In this case, instead of aligning the orientation bins in a regular bar graph, a histogram of orientation can be more intuitively drawn in a star-shape, where each line has the orientation associated to `bin` that it represents and the length of the line is proportional to that `bin` count. These HOG representations can then be displayed over an image, as follows:



Each cell HOG representation can be produced by a simple function that accepts an iterator pointing to a histogram. Lines of proper orientation and length are then drawn for each `bin`:

```
//draw one HOG over one cell
void drawHOG(std::vector<float>::const_iterator hog,
             // iterator to the HOG
  int numberOfBins,       // number of bins inHOG
  cv::Mat &image,         // image of the cell
  float scale=1.0) {      // length multiplier
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    const float PI = 3.1415927;
    float binStep = PI / numberOfBins;
    float maxLength = image.rows;
    float cx = image.cols / 2.;
    float cy = image.rows / 2.;

    // for each bin
    for (int bin = 0; bin < numberOfBins; bin++) {

      // bin orientation
      float angle = bin*binStep;
      float dirX = cos(angle);
      float dirY = sin(angle);
      // length of line proportion to bin size
      float length = 0.5*maxLength* *(hog+bin);

      // drawing the line
      float x1 = cx - dirX * length * scale;
      float y1 = cy - dirY * length * scale;
      float x2 = cx + dirX * length * scale;
      float y2 = cy + dirY * length * scale;
      cv::line(image, cv::Point(x1, y1), cv::Point(x2, y2),
               CV_RGB(255, 255, 255), 1);
    }
  }
```

An HOG visualization function will then call this preceding function for each cell:

```
  // Draw HOG over an image
  void drawHOGDescriptors(const cv::Mat &image,  // the input image
    cv::Mat &hogImage,     // the resulting HOG image
    cv::Size cellSize,     // size of each cell (blocks are ignored)
    int nBins) {           // number of bins

    // block size is image size
    cv::HOGDescriptor hog(
        cv::Size((image.cols / cellSize.width) * cellSize.width,
          (image.rows / cellSize.height) * cellSize.height),
          cv::Size((image.cols / cellSize.width) * cellSize.width,
          (image.rows / cellSize.height) * cellSize.height),
        cellSize,    // block stride (only 1 block here)
        cellSize,    // cell size
        nBins);      // number of bins

    //compute HOG
    std::vector<float> descriptors;
    hog.compute(image, descriptors);
    ...
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
    float scale= 2.0 / *
            std::max_element(descriptors.begin(),descriptors.end());
    hogImage.create(image.rows, image.cols, CV_8U);
    std::vector<float>::const_iterator itDesc= descriptors.begin();

    for (int i = 0; i < image.rows / cellSize.height; i++) {
      for (int j = 0; j < image.cols / cellSize.width; j++) {
        //draw each cell
           hogImage(cv::Rect(j*cellSize.width, i*cellSize.height,
                     cellSize.width, cellSize.height));
        drawHOG(itDesc, nBins,
                   hogImage(cv::Rect(j*cellSize.width,
                                     i*cellSize.height,
                                     cellSize.width, cellSize.height)),
                   scale);
        itDesc += nBins;
      }
    }
  }
```

This function computes an HOG descriptor that has the specified cell size but is made of only one large block (that is, a block that has the size of the image). This representation, therefore, ignores the effect of normalization that occurs at each block level.

People detection

OpenCV offers a pretrained people detector based on HOG and SVM. As for the classifier cascades of the previous recipe, this SVM classifier can be used to detect instances in a full image by scanning a window across the image at multiple scales. You then just have to construct the classifier and perform the detection on an image:

```
// create the detector
std::vector<cv::Rect> peoples;
cv::HOGDescriptor peopleHog;
peopleHog.setSVMDetector(
    cv::HOGDescriptor::getDefaultPeopleDetector());
// detect peoples oin an image
peopleHog.detectMultiScale(myImage, // input image
  peoples, // ouput list of bounding boxes
  0,       // threshold to consider a detection to be positive
  cv::Size(4, 4),    // window stride
  cv::Size(32, 32),  // image padding
  1.1,               // scale factor
  2);                // grouping threshold
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Here, the window stride defines how the *128 × 64* template is moved over the image (which is every *4* pixels horizontally and vertically in our example). Longer strides make the detection faster (because fewer windows are evaluated), but you may then miss some people falling in-between tested windows. The image padding parameter simply adds pixels on the border of the image so that people at the edge of the image can be detected. The standard threshold for an SVM classifier is *0* (since 1 is the value assigned to positive instances and *-1* to the negative ones). But if you really want to be certain that what you detect is a person, then you can raise this threshold value (this means that you want *high precision* at the price of missing some people in the image). Reciprocally, if you want to be certain of detecting all people (that is, you want a *high recall* rate), then you can lower the threshold; more false detections will occur in that case.

Here is an example of the detection results that are obtained:

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

It is important to note that when a classifier is applied to a full image, the multiple windows applied at successive locations will often lead to multiple detections around a positive sample. The best thing when two or more bounding boxes overlap at the same location is to retain only one of them. There is a function called `cv::groupRectangles` that simply combines rectangles of similar size at similar locations (this function is automatically called by `detectMultiScale`). In fact, obtaining a group of detections at a particular location can even be seen as an indicator confirming that we indeed have a positive instance at this location. This is why the `cv::groupRectangles` function allows us to specify the minimum size for a detection cluster to be accepted as a positive detection (that is, isolated detection should be discarded). This is the last parameter of the `detectMultiScale` method. Setting this at *0* will keep all detections (no grouping is done here), which, in our example, leads to the following result:

Let's take a look at deep learning and convolutional networks in the next section.

Deep learning and convolutional neural networks (CNNs)

We cannot conclude this chapter on machine learning without mentioning deep CNNs. The application of these to computer vision classification problems has led to impressive results. In fact, their outstanding performance when applied to real-world problems is such that they now open the door to a new family of applications that could not be envisioned previously.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Deep learning is based on the theory of neural networks that was introduced in the late 1950s. So, why are they generating such great interest today? Essentially, this is for two reasons; first, the computational power that is available nowadays allows the deployment of neural networks of a size that makes them able to solve challenging problems. While the first neural network (the perceptron) has only one layer and few weight parameters to tune, today's networks can have hundreds of layers and millions of parameters to be optimized (hence the name deep networks). Second, the large amount of data available today makes their training possible. In order to perform well, deep networks require thousands, if not millions, of annotated samples (this is because of the very large number of parameters that need to be optimized).

The most popular deep networks are CNNs. As the name suggests, they are based on convolution operations (see `Chapter 6`, *Filtering the Images*). The parameters to learn, in this case, are the values inside the kernel of all filters that compose the network. These filters are organized into layers in which the early layers extract the fundamental shapes, such as lines and corners, while the higher layers progressively detect more complex patterns (such as, for example, the presence of eyes, mouth, and hair, in a human detector).

OpenCV3 has a deep neural network module, but this is mainly for importing deep networks that have been trained using other tools such as TensorFlow, Caffe, or Torch. When building your future computer vision applications, you will certainly have to have a look at the deep learning theory and its related tools.

See also

- The *Describing and matching local intensity patterns* recipe in `Chapter 9`, *Describing and Matching Interest Points,* describes the SIFT descriptor, which is similar to the HOG descriptor.
- The *Histograms of Oriented Gradients for Human Detection* article by N. Dalal and B. Triggs in *Computer Vision and Pattern Recognition* conference, 2005, is the classical paper that introduces histograms of oriented gradients for people detection.
- The *Deep Learning* article by Y. LeCun, Y. Bengio, and G. Hinton in *Nature, no 521*, 2015, is a good starting point for exploring the world of deep learning.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

15
OpenCV Advanced Features

Deep learning is a subfield of machine learning, based on traditional neural networks and convolutional neural networks. It's state-of-the-art pattern recognition. Deep learning is reaching the best accuracy in fields such as speech recognition, text recognition, and image classification. Actually, OpenCV adds the deep learning module as a basic module in its core algorithms and is making big efforts to increase its performance using CPUs and GPUs.

In this chapter of advanced features, we are going to cover how to enable *Halide*, a new language that OpenCV is using to improve the performance of deep learning inferences.

Finally, we are going to explore the new possibilities of using OpenCV in a web browser as a JavaScript library, which allows us to use computer vision in a web client without installing anything on the client computer.

In this chapter, we are going to explore the following recipes:

- Face detection using deep learning
- Object detection using YOLOv3
- Enabling Halide to improve efficiency
- OpenCV.js introduction

Face detection using deep learning

In this recipe, we are going to learn how to use a trained deep learning model for a face detection algorithm in OpenCV. To do this, we are going to download pre-trained face detection models and use OpenCV methods to import the model and also see how to convert an input image or frame into the required deep learning structure.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Using deep learning in OpenCV is very easy, and the only files that we require are the pre-trained models and know the basic configuration of it. We can download tested, pre-trained OpenCV models from `https://github.com/opencv/open_model_zoo`.

To create the face detector algorithm, follow these steps:

1. Download and save the model of the face detector in the `data` folder (for convenience, we downloaded the required models and saved them in the `data` folder). It requires two files normally: the weights and the network structure. In our case, we are going to download the `deploy.prototxt` file, which defines the structure of the network,
   and `res10_300x300_ssd_iter_140000.caffemodel`, which contains the weights of the network.
2. Let's write our **deep neural network** (**DNN**) code. Create a `.cpp` file and include the DNN OpenCV library and the other required libraries:

```
#include <opencv2/dnn.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;
using namespace cv::dnn;
```

3. Let's declare some of the global variables to be used in our DNN algorithm that defines the input network, preprocessing data, and the filenames to load:

```
// Set up configuration
float confidenceThreshold = 0.5;
String modelConfiguration = "data/deploy.prototxt";
String modelBinary =
"data/res10_300x300_ssd_iter_140000.caffemodel";
const size_t inWidth = 300;
const size_t inHeight = 300;
const double inScaleFactor = 1.0;
const Scalar meanVal(104.0, 177.0, 123.0);
```

4. Create the `main` function and load the models in the OpenCV `dnn::Net` class:

```
int main(int argc, char** argv)
{
    // Init DNN
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
   dnn::Net net = readNetFromCaffe(modelConfiguration,
modelBinary);
```

5. Ask the `empty` function whether the DNN is correctly loaded:

```
if (net.empty())
 {
   cerr << "Can't load network by using the following files: " <<
endl;
   cerr << "prototxt: " << modelConfiguration << endl;
   cerr << "caffemodel: " << modelBinary << endl;
   cerr << "Models are available here:" << endl;
   cerr << "<OPENCV_SRC_DIR>/samples/dnn/face_detector" << endl;
   cerr << "or here:" << endl;
   cerr <<
"https://github.com/opencv/opencv/tree/master/samples/dnn/face_dete
ctor" << endl;
   exit(-1);
 }
```

6. If the DNN is correctly loaded, we can start to capture frames. Check the number of input arguments in our application to load the default value or a video file to process:

```
VideoCapture cap;
  if (argc==1)
  {
    cap = VideoCapture(0);
    if(!cap.isOpened())
    {
      cout << "Couldn't find default camera" << endl;
      return -1;
    }
   }
   else
   {
     cap.open(argv[1]);
     if(!cap.isOpened())
     {
       cout << "Couldn't open image or video: " << argv[1] <<
endl;
       return -1;
     }
 }
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. If the capturer is correctly opened, we can start the main loop to grab each frame until we get an empty frame:

```
for(;;)
{
  Mat frame;
  cap >> frame; // get a new frame from camera/video or read image
  if (frame.empty())
  {
     waitKey();
     break;
  }
```

8. Let's process the image in the DNN algorithm. Prepare the image to send to the DNN algorithm. To do so, we have to convert the OpenCV `Mat` structure into a DNN structure called blob using the `blobFromImage` function. OpenCV uses the `Mat` class to store the blobs:

```
//! [Prepare blob]
 Mat inputBlob = blobFromImage(frame, inScaleFactor,
 Size(inWidth, inHeight), meanVal, false, false); //Convert Mat to
batch of images
```

9. Once the frame is converted into a blob, set it as an input of our DNN and make detections using the `forward` function:

```
//! [Set input blob]
 net.setInput(inputBlob, "data"); //set the network input
 //! [Make forward pass]
 Mat detection = net.forward("detection_out"); //compute output
 Mat detectionMat(detection.size[2], detection.size[3], CV_32F,
detection.ptr<float>());
```

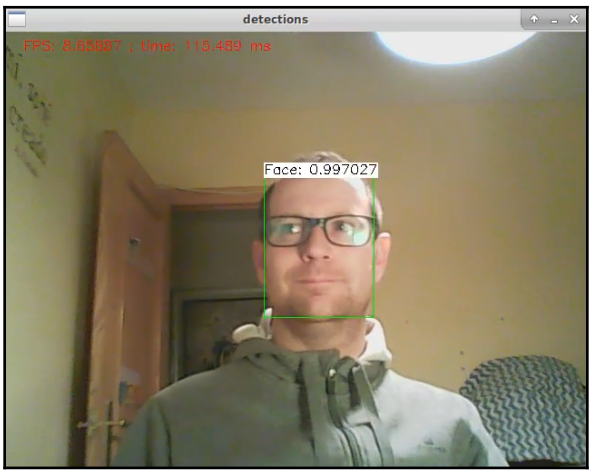
10. For each detected face, we are going to draw a rectangle into the image frame and draw its confidence:

```
for(int i = 0; i < detectionMat.rows; i++)
{
  float confidence = detectionMat.at<float>(i, 2);

  if(confidence > confidenceThreshold)
  {
    int xLeftBottom = static_cast<int>(detectionMat.at<float>(i, 3)
* frame.cols);
    int yLeftBottom = static_cast<int>(detectionMat.at<float>(i, 4)
* frame.rows);
    int xRightTop = static_cast<int>(detectionMat.at<float>(i, 5) *
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
frame.cols);
    int yRightTop = static_cast<int>(detectionMat.at<float>(i, 6) *
frame.rows);

    Rect object((int)xLeftBottom, (int)yLeftBottom,
 (int)(xRightTop - xLeftBottom),
 (int)(yRightTop - yLeftBottom));

    rectangle(frame, object, Scalar(0, 255, 0));

    stringstream ss;
    ss.str("");
    ss << confidence;
    String conf(ss.str());
    String label = "Face: " + conf;
    int baseLine = 0;
    Size labelSize = getTextSize(label, FONT_HERSHEY_SIMPLEX, 0.5,
1, &baseLine);
    rectangle(frame, Rect(Point(xLeftBottom, yLeftBottom -
labelSize.height),
    Size(labelSize.width, labelSize.height + baseLine)),
    Scalar(255, 255, 255), FILLED);
    putText(frame, label, Point(xLeftBottom, yLeftBottom),
    FONT_HERSHEY_SIMPLEX, 0.5, Scalar(0,0,0));
   }
 }
```

We can see the result of our code in the next screenshot:



Now, let's go behind the scenes to understand the code better.

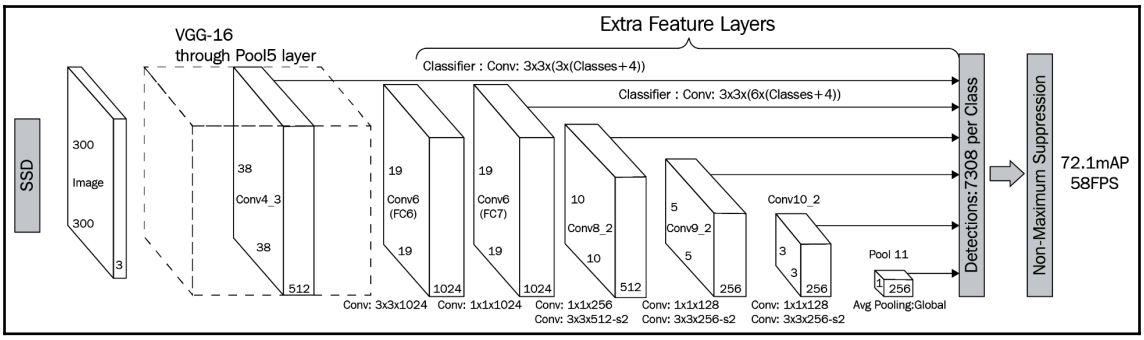
EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

Let's now explore how the DNN face detector works. In our example, we use the **Single-Shot Detector** (**SSD**) DNN algorithm, which simultaneously predicts the bounding boxes and the class when the image is processed by the DNN. Basically, the DNN structure is defined as follows:

- A *300 x 300* input image.
- The input image is passed through multiple convolutional layers, obtaining different features at different scales.
- For each feature map obtained in 2, we use a *3 x 3* convolutional filter to evaluate a small set of default bounding boxes.
- For each default box evaluated, the bounding box offsets and class probabilities are predicted.

The model architecture looks like this:



SSD is a DNN algorithm developed to classify multiple classes and not only faces, but we use a modified network to perform an accurate execution of face detection.

In OpenCV, the most important functions that define and use DNN are `blobFomImage`, `readNetFrom`, `setInput`, and `forward`.

When we need to convert the input image into a blob, we use the `blobFromImage` function, which has the following structure:

```
blobFromImage(image, scaleFactor,
 size, mean, swapRB, crop);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Here is the meaning of each parameter of the `blobFromImage` function:

- `image`: Input image (with 1, 3, or 4 channels).
- `size`: Spatial size for the output image.
- `mean`: Scalar with mean values that are subtracted from channels. Values are intended to be in (mean-R, mean-G, mean-B) order if the image has BGR ordering and `swapRB` is true.
- `scalefactor`: Multiplier for image values.
- `swapRB`: This is the flag that indicates that swapping the first and last channels in a 3-channel image is necessary.
- `crop`: This is the flag that indicates whether the image will be cropped after resize.

To load a model, we can import it from any of these types using its `readFrom[type]` importers, as follows:

- **Caffe**
- **DarkNet**
- **ONNX** (short for **Open Neural Network Exchange**)
- **TensorFlow**
- **Torch**

Once we have imported the files and we have an input blob, we can set the input blob to the network using the `setInput` function of the `Net` class, where the first `param` is the blob input and the second `param` is a string of the input name layer if more than one exists.

The last import function is `forward`, which executes the network for the input blob and returns the predictions in a `Mat` format.

In the face detection algorithm, the returned `Mat` has the following structure, where `detection.size[2]` is the number of detected objects and `detection.size[3]` is the number of results per detection (bounding box data and confidence) and each row has the following structure:

- `Column 0`: Confidence of the object being present
- `Column 1`: Confidence of bounding box
- `Column 2`: Confidence of face detected
- `Column 3`: X bottom-left bounding box
- `Column 4`: Y bottom-left bounding box

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

- `Column 5`: X top-right bounding box
- `Column 6`: Y top-right bounding box

The bounding box is relative (zero to one) to the image size because when we generate the rectangle to draw, we multiply by frame size.

See also

- For more information on the information covered in this recipe, check out *SSD: Single Shot MultiBox Detector* by Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Arxiv 2016 (`https://arxiv.org/abs/1512.02325`).

We've successfully learned how to detect a face using deep learning. Now let's move on to the next recipe.

Object detection with YOLOv3

In this recipe, we are going to use the YOLOv3 DNN algorithm, which performs object detection. Object detection in an image is a common task in computer vision; with deep learning techniques, we can achieve highly accurate detections. YOLOv3 reaches a 60.6 mAP in the COCO dataset (80 classes and more than 300,000 images), which has a very good precision with 20 fps or 33 mAP with 220 fps.

How to do it...

In this recipe, we are going to use the same functions and classes to load the model, convert the images, and predict, but we are going to introduce one more important function – non-maximum suppression – and a few `util` functions to draw the predictions with their labels:

1. Create a `.cpp` file that starts by including some required headers and the OpenCV headers. Initialize the required global variables for the minimum threshold and a vector to store the labels of each class:

```
#include <fstream>
#include <sstream>
#include <iostream>
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
#include <opencv2/core.hpp>
#include <opencv2/dnn.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>

using namespace cv;
using namespace dnn;
using namespace std;

// Initialize the parameters
float confThreshold = 0.5; // Confidence threshold
float nmsThreshold = 0.4; // Non-maximum suppression threshold
int inpWidth = 416; // Width of network's input image
int inpHeight = 416; // Height of network's input image
vector<string> classes;
```

2. Let's start with our `main` function. Read the file that stores, line by line, all the classes that we can predict:

```
int main(int argc, char** argv)
{
  // Load names of classes
  string classesFile = "data/coco.names";
  ifstream ifs(classesFile.c_str());
  string line;
  while (getline(ifs, line)) classes.push_back(line);
```

3. Load the model using its weights and definition:

```
// Give the configuration and weight files for the model
 String modelConfiguration = "data/yolov3.cfg";
 String modelWeights = "data/yolov3.weights";

// Load the network
 Net net = readNetFromDarknet(modelConfiguration, modelWeights);
```

4. Load an image and convert it into a blob:

```
Mat input, blob;

 input= imread(argv[1]);

// Stop the program if reached end of video
 if (input.empty()) {
 cout << "No input image" << endl;
 return 0;
 }
 // Create a 4D blob from a frame.
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
 blobFromImage(input, blob, 1/255.0, Size(inpWidth, inpHeight),
Scalar(0,0,0), true, false);
```

5. Let's detect all objects and their classes using the `setInput` and `forward` functions:

```
//Sets the input to the network
 net.setInput(blob);

 // Runs the forward pass to get output of the output layers
 vector<Mat> outs;
 net.forward(outs, getOutputsNames(net));
```

6. Post-process the output results, draw over an image, and show it:

```
// Remove the bounding boxes with low confidence
 postprocess(input, outs);

 imshow("Deep Learning. Chapter 12", input);
 waitKey(0);
```

7. In the `postprocess` function, store all boxes that have a prediction confidence higher than `confThreshold`:

```
for (size_t i = 0; i < outs.size(); ++i)
 {
    //Only the bounding boxes with high confidence scores are kept
after scanning through the
    //output. The class with the highest score for the box is
assigned the box's class label
   float* data = (float*)outs[i].data;
   for (int j = 0; j < outs[i].rows; ++j, data += outs[i].cols)
   {
     Mat scores = outs[i].row(j).colRange(5, outs[i].cols);
     Point classIdPoint;
     double confidence;
     // Get the value and location of the maximum score
     minMaxLoc(scores, 0, &confidence, 0, &classIdPoint);
     if (confidence > confThreshold)
     {
       int centerX = (int)(data[0] * frame.cols);
       int centerY = (int)(data[1] * frame.rows);
       int width = (int)(data[2] * frame.cols);
       int height = (int)(data[3] * frame.rows);
       int left = centerX - width / 2;
       int top = centerY - height / 2;

       classIds.push_back(classIdPoint.x);
```

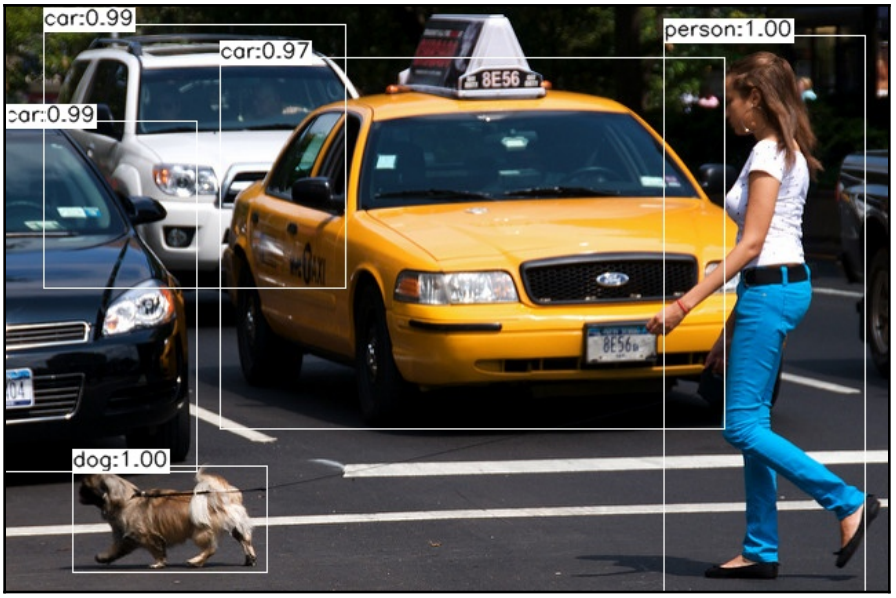

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            confidences.push_back((float)confidence);
            boxes.push_back(Rect(left, top, width, height));
          }
        }
      }
```

8. Apply non-maximum suppression with the `NMSBoxes` function to get only the non-overlapping boxes with a high confidence and draw them:

```
vector<int> indices;
NMSBoxes(boxes, confidences, confThreshold, nmsThreshold, indices);
for (size_t i = 0; i < indices.size(); ++i)
{
  int idx = indices[i];
  Rect box = boxes[idx];
  drawPred(classIds[idx], confidences[idx], box.x, box.y,
 box.x + box.width, box.y + box.height, frame);
}
```

The result of using YOLOv3 looks as follows:

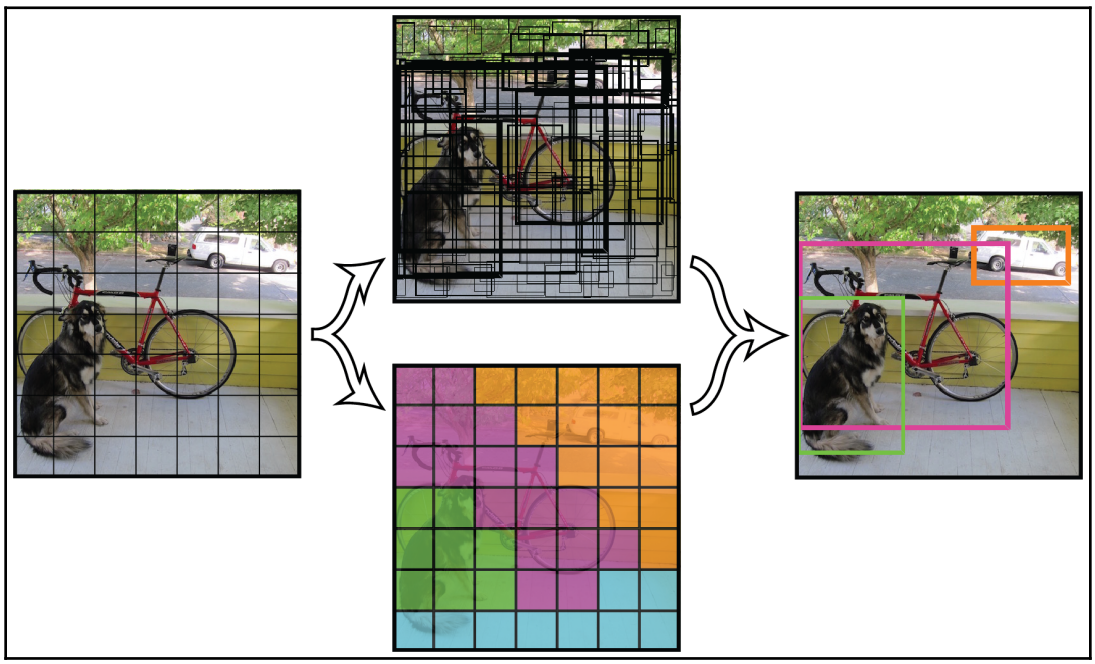


Now, let's go behind the scenes to understand the code better.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

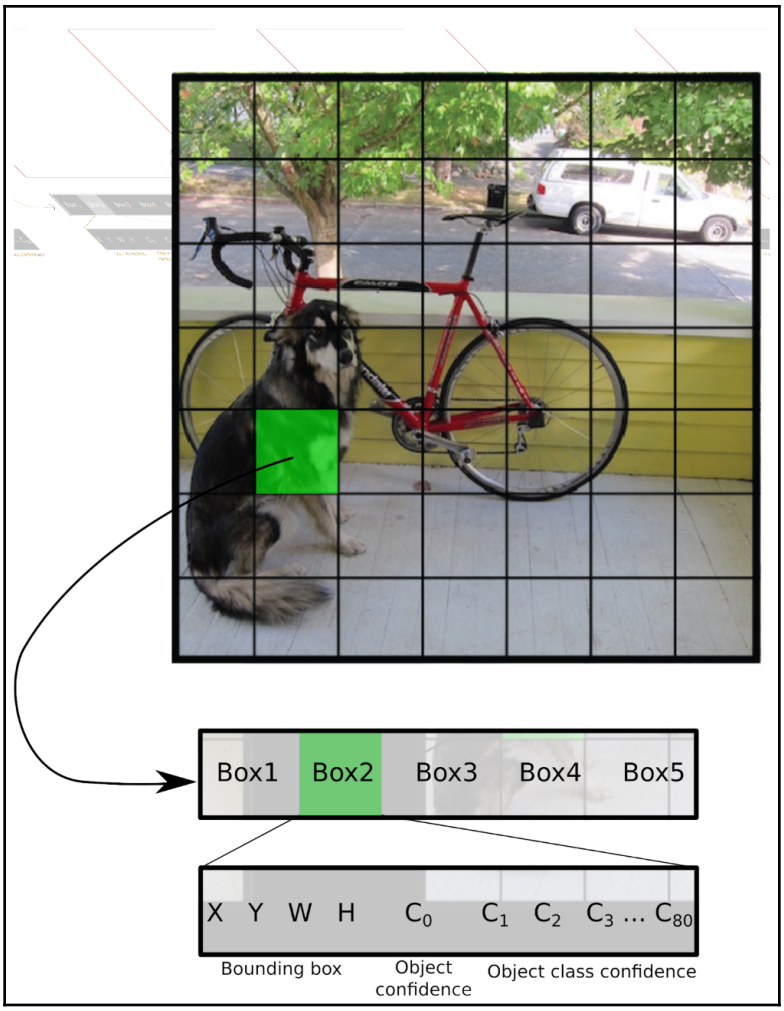
How it works...

YOLO is a non-trivial deep learning neural network; it uses a different approach from other object-detection algorithms. YOLO divides the input image into an *S x S* grid. For each grid, YOLO checks for *B* bounding boxes, and then the deep learning model extracts the bounding boxes for each patch, the confidence to contain a possible object, and the confidence of each category in the training dataset per each box. The following screenshot shows the *S x S* grid:



YOLO is trained with a grid of *19* and *5* bounding boxes per grid using 80 categories. Then, the output result is 19 x 19 x 425, where 425 comes from the data of the bounding box *(x, y, width, height)*, the object confidence, and the 80 classes, confidence multiplied by the number of boxes per grid; *5_bounding boxes*(x,y,w,h,object_confidence, classify_confidence*[*80*])=*5*(*4 + 1 + 80*):

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



The YOLOv3 architecture is based on DarkNet, which contains 53 layer networks, and YOLO adds 53 more layers for a total of 106 network layers. If you want a faster architecture, you can check out Version 2 or TinyYOLO versions, which use fewer layers.

See also

- COCO dataset: `http://cocodataset.org/#home`
- YOLO website: `https://pjreddie.com/darknet/yolo/`

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

We've successfully learned how to detect an object with YOLOv3. Now let's move on to the next recipe.

Enabling Halide to improve efficiency

Halide is a new programming language written for high-performance image processing on new computers. Halide currently targets the following:

- **CPU architectures**: X86, ARM, MIPS, Hexagon, and PowerPC
- **Operating systems**: Linux, Windows, macOS, Android, iOS, and Qualcomm QuRT
- **GPU Compute APIs**: CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal, and Microsoft Direct X 12

In this recipe, we are going to show how to compile OpenCV with Halide and how to enable it in your DNN algorithms.

How to do it...

To enable Halide in OpenCV, follow these steps. It will take a while to compile it all:

1. Download and unpack the LLVM source code from `http://releases.llvm.org/4.0.0/llvm-4.0.0.src.tar.xz`. We are going to call `llvm_root` as an `llvm root` directory of the source code.
2. Create a directory, `llvm_root/tools/clang`.
3. Download CLang with the same version of LLVM (from `http://releases.llvm.org/4.0.0/cfe-4.0.0.src.tar.xz`) and extract it into `llvm_root/tools/clang/`.
4. Go into the `llvm_root` folder:

```
cd llvm_root
```

5. Create the `build` folder and go into it:

```
$ mkdir build
$ cd build
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Configure the project to compile with `cmake`:

```
cmake -DLLVM_ENABLE_TERMINFO=OFF -DLLVM_TARGETS_TO_BUILD="X86" -
DLLVM_ENABLE_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=Release ..
```

7. Compile LLVM as follows:

```
make -j8
```

8. When it finishes, we have to continue compiling Halide. To do so, download it from GitHub (`https://github.com/halide/Halide`).
9. Access the Halide folder and create the `build` folder:

```
$ cd halide_root
$ mkdir build
$ cd build
```

10. Configure the project to compile Halide. Remember to change `llvm_root` to your correct path:

```
cmake -DLLVM_DIR=llvm_root/build/lib/cmake/llvm -
DCMAKE_BUILD_TYPE=Release -DLLVM_VERSION=40 -DWITH_TESTS=OFF -
DWITH_APPS=OFF -DWITH_TUTORIALS=OFF ..
```

11. Compile Halide as follows:

```
make -j8
```

12. Compile OpenCV. Download it and create a `build` folder:

```
$ git clone https://github.com/opencv/opencv.git
$ cd opencv
$ mkdir build
$ cd build
```

13. Configure the project with Halide flags enabled:

```
cmake -DWITH_HALIDE=ON -DHALIDE_ROOT_DIR=halide_root
```

14. Compile OpenCV with Halide with the `make` command:

```
make -j8
```

Finally, in your DNN projects, to enable the Halide backend, you only have to add the following line after loading your models:

```
net.setPreferableBackend(DNN_BACKEND_HALIDE);
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How it works...

The OpenCV integration of Halide is in progress and is not mature yet, but the expectations of Halide in OpenCV are high to improve DNN modules in different architectures. In the first stages of testing, developers reach in some DNN algorithms, such as ENet, a 320% speed using Halide on CPU and 470% better speed in GPU. In other cases, it is not high-performing, but core developers are working to improve it.

See also

- Halide language website: `https://halide-lang.org/`
- *Differentiable Programming for Image Processing and Deep Learning in Halide* by Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley, SIGGRAPH 2018
- The first presentation of OpenCV and Halide: `https://es.slideshare.net/embeddedvision/making-opencv-code-run-fast-a-presentation-from-intel`

We've successfully learned how to enable Halide to improve efficiency. Now let's move on to the next recipe.

OpenCV.js introduction

This recipe introduces a new way to develop computer vision algorithms with OpenCV using JavaScript for the **World Wide Web** (**WWW**). Before OpenCV.js, developers that needed some computer vision tasks on the web had to develop them in C++ in servers or Java applets clients, which were not optimized, but with OpenCV.js, the new paradigm of client applications using web browsers has grown with unlimited of possibilities.

In this recipe, we are going to create a very simple case where we load an image into an `img` HTML tag, change the color image, and present it to a `canvas` element.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

How to do it...

Before starting to use OpenCV.js, we need a web page as a user interface. To get a user interface, we are going to create an HTML web page:

1. Create an HTML page with the `img`, `canvas`, and `button` elements:

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <style>
  #container {min-height:300px;}
  #canvasOutput, #imageSrc{
    background:#ccc;
    min-width:300px;
    min-height:300px;
    display:block;
    float:left;
    margin-left:20px;
  }
    </style>
    <title>OpenCV Computer vision on Web. Packt Publishing.</title>
  </head>
  <body>
  <div id="status" class="alert alert-primary" role="alert">Loading
OpenCV...</div>
  <div id="container">
    <img id="imageSrc" alt="No Image" class="small"
src="img/gray.png">
          <canvas id="canvasOutput" class="small"
height="300px"></canvas>
  </div>
  <input type="file" id="fileInput" name="file" accept="image/*">
```

2. At the bottom of the web page, load the OpenCV JavaScript library:

```
<!-- OPENCV -->
    <script async="" src="data/opencv.js" type="text/javascript"
onload="onOpenCvReady();" onerror="onOpenCvError();"></script>
```


EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. In a `script` wrapper, let's create our application. We need to get the references as variables of the input button and the image element:

```
<script type="text/javascript">
        let imgElement = document.getElementById('imageSrc');
        let inputElement = document.getElementById('fileInput');
```

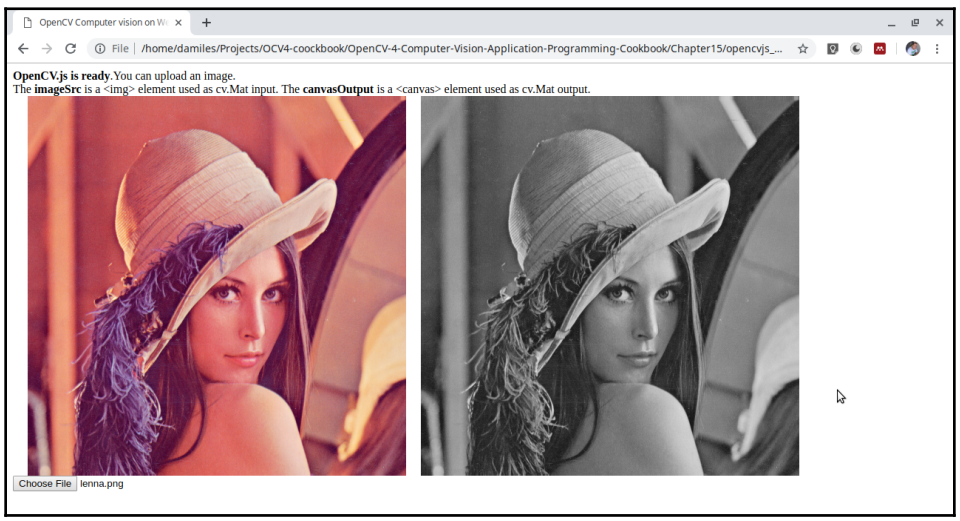
4. When the button is clicked, we are going to load the selected image into the `img` element:

```
inputElement.addEventListener('change', (e) => {
            imgElement.src =
URL.createObjectURL(e.target.files[0]);
        }, false);
```

5. When the image is loaded in the `img` element, we use OpenCV functions (as in C++) and apply the operations we want. In our case, we are going to change from RGB to gray and we are going to use the `imshow` function to show the result in `canvas` element by passing it to the function the ID of `canvas`:

```
imgElement.onload = function() {
            let mat = cv.imread(imgElement);
            cv.cvtColor(mat, mat, cv.COLOR_BGR2GRAY);
            cv.imshow('canvasOutput', mat);
            mat.delete();
        };
```

Here is the result:



EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, let's go behind the scenes to understand the code better.

How it works...

OpenCV.js is a port of some OpenCV functions using new technology that compiles C++ code into JavaScript. OpenCV uses Emscripten to compile C++ functions into Asm.js or WebAssembly targets.

Asm.js is highly optimized and it is designed to be near-native code, achieving speeds 2x slower (depending on the browser and computer) than the same native executable application.

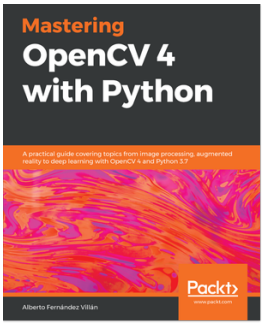
WebAssembly is a new technology and web standard that defines a binary format for executing code in web pages. It is developed to complement JavaScript to speed up applications that must run like native code. This technology is the best choice to increase the performance of computer vision and port OpenCV to JavaScript. WebAssembly is highly optimized for speed and achieves near-native code that is just 1.5x slower.

In our case, we develop a simple web page structure where, with a simple button, we can load an image into an `img` element that we require to use as an input image. Reading this `img` element content, we can load into an OpenCV `Mat` and apply almost any computer vision functions that OpenCV can offer. In our example, we convert the image from color to gray. OpenCV.js changes the `imshow` functionality to allow us to show `cv::Mat` in a `canvas` element instead of new windows, like, C++, enabling us to interact with the web page.

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

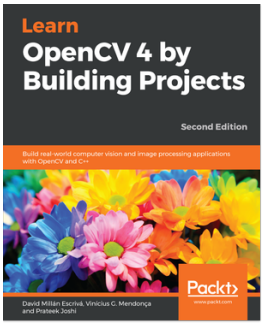


Mastering OpenCV 4 with Python
Alberto Fernández Villán

ISBN: 9781789344912

- Handle files and images, and explore various image processing techniques
- Explore image transformations, including translation, resizing, and cropping
- Gain insights into building histograms
- Brush up on contour detection, filtering, and drawing
- Work with Augmented Reality to build marker-based and markerless applications
- Work with the main machine learning algorithms in OpenCV
- Explore the deep learning Python libraries and OpenCV deep learning capabilities
- Create computer vision and deep learning web applications

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use



Learn OpenCV 4 By Building Projects - Second Edition
David Millán Escrivá, Vinícius G. Mendonça, Prateek Joshi

ISBN: 9781789341225

- Install OpenCV 4 on your operating system
- Create CMake scripts to compile your C++ application
- Understand basic image matrix formats and filters
- Explore segmentation and feature extraction techniques
- Remove backgrounds from static scenes to identify moving objects for surveillance
- Employ various techniques to track objects in a live video
- Work with new OpenCV functions for text detection and recognition with Tesseract
- Get acquainted with important deep learning tools for image classification

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

Index

3

3D scene
  reconstructing, from calibrated cameras 336, 337, 339, 342, 343, 345

A

AdaBoost 417
adapted feature detection 256, 257, 258
adaptive thresholding 136
anchor point
  images, eroding 145
apparent motion 380

B

backward mapping 68
Beucher gradient 157
bicubic interpolation 189
Binary Robust Independent Elementary Features (BRIEF) 289
Binary Robust Invariant Scalable Keypoints (BRISK) detector 265
brightness constancy equation 387
brightness
  colors, representing 90, 93
bundle adjustment 346

C

callback function 20
camera calibration
  about 321, 322, 324, 328, 329
  grid of circles, using 329
  with known intrinsic parameters 329
  working 326, 328
camera pose
  recovering 330, 332, 333, 334
circles
  detecting 222, 223, 224
CMake tool
  reference 10
codec 370
codec four-character code 371, 372
color histograms
  backprojecting 120, 122
color reduction formulas 45, 46
color representations
  converting 86, 88, 90
colors
  used, for skin tone detection 95, 97
components' contours
  extracting 229, 230, 231, 232, 233
components' shape descriptors
  computing 234, 235, 236
  quadrilateral, detecting 238, 239
connected components 229
continuous images
  efficient scanning 48, 49
convex hull 237
convolution 181
convolutional neural networks (CNNs)
  deep learning, using 434, 435
corners
  about 241
  detecting, in image 241, 242, 243, 245, 247, 248, 249
cross-checking matches 283

D

deep learning
  used, for face detection 436, 437, 439, 441, 442
  with convolutional neural networks (CNNs) 434, 435
dense motion 380

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

depth map
  computing, from stereo image 346, 348, 350
descriptors 270
Difference of Gaussians (DoG) 207, 208
digital image formation 319, 320
directional filters
  applying, to detect edges 191, 194, 195, 196, 197, 198
distance thresholding 285, 286
distinctive regions
  extracting, with MSER 166, 168, 169, 172
downsampling 183

E

edge detection 191
efficient image-scanning loops
  writing 53, 55
essential matrix 337
extrinsic parameters 326

F

face detection
  deep learning used 436, 437, 439, 441, 443
faces
  detecting, Haar cascade used 420, 421
  finding, Haar features cascade used 411, 412, 413, 414, 415, 417, 418, 419
  recognizing, nearest neighbors of local binary patterns used 404, 405, 406, 407, 408, 409, 410
false alarm rate (FAR) 415
false positives 415
FAST (Features from Accelerated Segment Test) 253
FAST detector
  using 253
  working 254, 255, 256
FAST features
  detecting, at multiple scales 265, 266
  working 267
feature descriptors 278
feature detector's common interface 251, 252
feature points
  about 240
  in video, tracing 381, 385, 386, 387
features
  detecting quickly 253
  detecting, with FAST detector 253, 254
filter 178
foreground objects
  extracting, with GrabCut algorithm 173, 174, 176
  in video, extracting 372, 373, 374, 377
frame processor class
  using 364
frame rate 352
frames 352
FREAK (Fast Retina Keypoint) 290, 291, 292
frequency domain 177
fundamental matrix
  computing. of image pair 296, 297, 299, 301, 302
  refining 308

G

gaussian derivatives 200, 201
GNU Lesser General Public License (LGPL) 13
GrabCut algorithm
  used, for extracting foreground objects 173, 174, 176
  used, for image segmentation 82, 84, 86
gradient 196
gradient operators 199

H

Haar cascade
  used, for detecting faces 420, 421
Haar features cascade
  used, for detecting objects and faces 412, 413, 416, 418, 420
  used, for finding objects and faces 411
Haar-like feature 411
Halide
  enabling, to improve efficiency 449, 450, 451
Hamming norm 288
Harris corners
  detecting 250
histogram comparison
  used, for retrieving similar images 128, 130, 131
Histogram of Oriented Gradients (HOG) 422

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

histogram
  about 100
  backprojecting, for specific image content detection 116
  stretching, for image contrast improvement 111
histograms, color images
  computing 106, 108
histograms
  backprojecting, for specific image content detection 119
  used, for visual tracking 139, 142
hit rate (HR) 415
HOG visualization 430
homography between two images
  computing 309, 310, 312, 313, 314
homography
  decomposing 345
Hough transform
  about 213
  circles, detecting 222, 223, 224
  used, for detecting lines in image 213, 214, 215, 218, 219, 220, 222
hue
  colors, representing 90, 93
hysteresis thresholding 212

I

image brightness constancy assumption 389
image channels
  splitting 65
image contours
  detecting, with Canny operator 209, 211, 212, 213
image edges
  detecting, with morphological filters 153, 157
image emboss 194
image histogram
  computing 99, 101, 104, 106
  equalizing 114, 116
  working 116
image masks
  using 33, 34
image mosaicking 312
image pair
  fundamental matrix, computing 296, 297, 299, 301, 302
image pyramids 186
image
  contrast enhancing, Laplacian used 207
  Laplacian, computing 204, 205
  lines, detecting with Hough transform 220, 222
images sequence
  processing 363, 364
images
  clicking on 20, 21
  closing, with morphological filters 150, 153
  dilating, with morphological filters 145
  displaying 16, 19
  downsampling 183, 184, 185, 186, 187
  drawing on 21
  eroding, with morphological filters 145, 147, 148
  filtering, low-pass filters used 178, 179, 180, 181, 182, 183
  filtering, median filter used 189, 190, 191
  input arguments 46
  Laplacian, computing 201, 203
  lines, detecting with Hough transform 213, 214, 215, 218, 219
  loading 15, 16, 18
  matching, with random sample consensus 302, 304, 306, 307, 308
  opening, with morphological filters 150, 153
  output arguments 46
  pixel values, interpolating 187, 188, 189
  planar targets, detecting 314, 316, 317
  remapping 66, 68, 69
  saving 18, 19
  scanning, with iterators 50, 51, 52
  scanning, with neighbor access 57, 58, 59
  scanning, with pointers 41, 42, 44
in-place transformation 46
input array 30
integral images
  adaptive thresholding 135, 138
  used, for counting pixels 132, 134
integrated development environment (IDE) 9
Intel Threading Building Blocks (TBB) 56
interest points
  about 240
  corner, detecting in image 241

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

iterators
  image, scanning with 50, 51, 52

K

kernel 180
Kernelized Correlation Filter (KCF) 401
keypoints
  about 240
  describing, with binary features 287, 289

L

L1 norm 197
L2 norm 197
Laplacian of Gaussian (LoG) 204
Laplacian
  computing, of image 201, 203, 204, 205
  used, for enhancing image contrast 207
learning rate 374
line
  fitting, to set of points 225, 226, 227, 228
local binary pattern (LBP)
  about 404
  faces recognizing, nearest neighbors used 404, 405, 406, 407, 409, 410
local intensity patterns
  describing 278, 279, 280
  working 281, 282
local templates
  matching 271, 272, 274, 276
  working 274, 275, 276
lookup tables
  applying, for modification of image appearance 109, 110
  applying, on color images 113
low-level pointer arithmetics 49
low-pass filters
  used, for filtering images 178, 179, 180, 181, 182, 183
Lukas-Kanade feature tracking 387

M

mask 180
Mat data structure
  exploring 23, 24, 25, 27, 28, 29
Mat_ template class 41
matches
  refining 309
mathematical morphology 144
maximally stable external regions (MSER)
  about 166
  used, for extracting distinctive regions 166, 168, 169, 172
  working 169
mean shift algorithm
  used, for finding object 123, 125, 127
median filter
  used, for filtering image 189, 190, 191
mixture of Gaussian method 377, 378
moments 237
morphological filters
  about 145
  images, dilating 145, 147, 148
  images, eroding 147, 148
  used, for closing images 150, 152
  used, for detecting corners 153, 156, 158
  used, for detecting edges 153, 156, 158
  used, for opening images 150, 152
motion field 388
moving average 373

N

nearest neighbor classification 404
nearest neighbor strategy 187
nearest neighbors, local binary patterns
  used, for recognizing faces 404, 405, 406, 408, 409
nearest neighbors, of local binary patterns
  used, for recognizing faces 411
neighbor access
  image, scanning with 57, 58, 60
Nyquist-Shannon 185

O

object detection
  YOLOv3 used 443, 447, 449
objects
  detecting, histograms of oriented gradients used 422
  detecting, SVMs used 422, 423, 424, 426, 428, 430

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

finding, Haar features cascade used 411, 412, 413, 414, 416, 417, 418, 419, 420
in video, tracking 394, 396, 398, 399, 400, 401
OpenCV developer site
reference 15
OpenCV developments
Qt, using for 13
OpenCV library
installing 8, 9, 10, 11, 12
modules 12
OpenCV.js 451, 452, 453, 454
OpenCV
reference 9
OpenMP 56
optical flow
about 387, 388
estimating 388, 389, 390, 393
ORB (Oriented FAST and Rotated BRIEF) 265
ORB feature-detection algorithm 268, 269
output array 30
overloaded image operators 64

P

people
detecting 432
detecting, histograms of oriented gradients used 422, 424, 426, 428
Perspective-n-Point (PnP) 333
phenomenal color spaces 90
pixel values
accessing 36, 37, 38, 39, 40
pixels 35
planar targets
detecting, in image 314, 316, 317
pointers
images, scanning with 41, 42, 44
principal point 319
Probabilistic Hough Transform 217
projection matrix 296

Q

Qt
download link 14
OpenCV applications, running 22, 23
using, for OpenCV developments 13
quadrilateral
detecting 238, 239

R

radial distortion 327
random sample consensus
used, for matching images 302, 304, 306, 307, 308
ratio test 284, 285
regions of interest
defining 30, 31, 32
reprojection error 326
running average 373

S

salt-and-pepper noise 37
saturation
colors, representing 90, 93
Scale Invariant Feature Transform (SIFT) 337
scale-invariant features
detecting 259
detecting, with SURF detector 259, 261, 262, 263
SIFT feature-detection algorithm 263, 264
signals and systems theory 177
simple image arithmetic
performing 62, 63, 64
skin tone detection
colors, using 95, 98
Sobel filter 192
Sobel operator 211
sparse motion 380
spatial aliasing 184
spatial domain 177
Standard Template Library (STL) 50
stereo image
depth map, computing from 346, 347, 348, 350, 351
stereo vision 346
strategy design pattern
about 71
distance between two color vectors, computing 77
function object 80
functor 80

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use

OpenCV base class, for algorithms 81
OpenCV functions, using 78
used, for comparing colors 71, 72, 74, 77
Sum of Squared Differences (SSD) 271
Support Vector Machines (SVMs) 404
support vectors 427
SURF (Speeded Up Robust Features) 259
SURF detector
working 261

T

thin lens equation 294
triangulation 340
trichromatic 35

U

upsampling 183

V

video frames
processing 357, 358, 359, 362, 363
processor class, using 364
sequence of images, processing 363, 364
video sequences
codec four-character code 371, 372
reading 352, 353, 354, 355, 356
writing 366, 367, 370
video
feature points, tracing 381, 385, 386
feature points, tracking 384, 387
foreground objects, extracting 372, 373, 376, 377
mixture of Gaussian method 377
object, tracking 394, 395, 396, 398
Visualization Toolkit (VTK) 334
Viz module 334, 336

W

watersheds
used, for segmenting images 158, 161, 164, 166

Y

YOLOv3
used, for object detection 443, 447

Z

zero-crossings 204

EBSCOhost - printed on 2/9/2023 10:15 AM via . All use subject to https://www.ebsco.com/terms-of-use