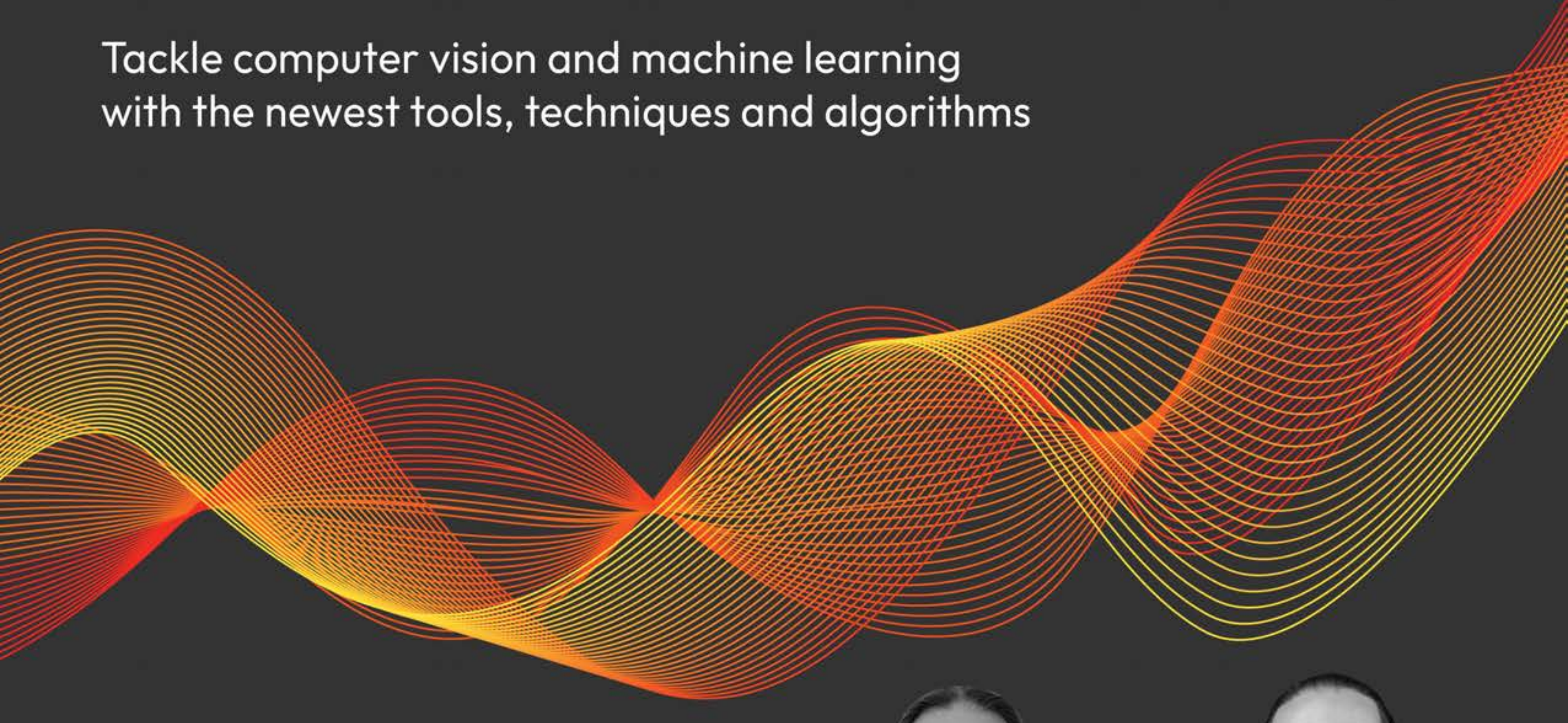
EXPERT INSIGHT

Learning OpenCV 5 Computer Vision with Python

Early Access

Tackle computer vision and machine learning with the newest tools, techniques and algorithms



Fourth Edition

Joseph Howse
Joe Minichino

‹packt›

Learning OpenCV 5 Computer Vision with Python

Early Access Publication: Learning OpenCV 5 Computer Vision with Python

Early Access Production Reference: B18454

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80323-022-1

www.packt.com

Table of Contents

Learning OpenCV 5 Computer Vision with Python, Fourth Edition: Tackle tools, techniques, and algorithms for computer vision and machine learning

5 Detecting and Recognizing Faces

Computer vision makes many futuristic-sounding tasks a reality. Two such tasks are face detection (locating faces in an image) and face recognition (identifying a face as belonging to a specific person). OpenCV implements several algorithms for face detection and recognition. These have applications in all sorts of real-world contexts, from security to entertainment.This chapter introduces some of OpenCV's face detection and recognition functionality, along with data files that define particular types of trackable objects. Specifically, in this chapter, we'll look at Haar cascade classifiers, which analyze the contrast between adjacent image regions to determine whether or not a given image or sub-image matches a known type. We consider how to combine multiple Haar cascade classifiers in a hierarchy so that one classifier identifies a parent region (for our purposes, a face) and other classifiers identify child regions (such as eyes).We also take a detour into the humble but important subject of rectangles. By drawing, copying, and resizing rectangular image regions, we can perform simple manipulations on image regions that we are tracking.We will cover the following topics:

- Understanding Haar cascades
- Finding the pre-trained Haar cascades that come with OpenCV. These include several face detectors
- Using Haar cascades to detect faces in still images and videos
- Gathering images to train and test a face recognizer
- Using several different face recognition algorithms: **Eigenfaces**, **Fisherfaces**, and **Local Binary Pattern Histograms** (**LBPHs**)
- Copying rectangular regions from one image to another, with or without a mask
- Using a depth camera to distinguish between a face and the background based on depth
- Swapping two people's faces in an interactive application

While this chapter focuses on classic approaches to face detection and recognition, we will go on to explore advanced new models by using OpenCV with a variety of other libraries in *Chapter 11*, *Neutral Networks with OpenCV – an Introduction*.

By the end of this chapter, we will have integrated face tracking and rectangle manipulations into Cameo, the interactive application that we have developed in previous chapters. Finally, we will have some face-to-face interaction!

Technical requirements

This chapter uses Python, OpenCV, and NumPy. As part of OpenCV, it uses the optional `opencv_contrib` modules, which include functionality for face recognition. Some parts of this chapter use OpenCV's optional support for OpenNI 2 to capture images from depth cameras. Please refer back to *Chapter 1*, *Setting Up OpenCV*, for installation instructions.The complete code for this chapter can be found in this book's GitHub repository, https://github.com/PacktPublishing/Learning-OpenCV-5-Computer-Vision-with-Python-Fourth-Edition, in the `chapter05` folder. Sample images are in the repository `images` folder.A subset of the chapter's sample code can be edited and run interactively in Google Colab at https://colab.research.google.com/github/PacktPublishing/Learning-OpenCV-5-Computer-Vision-with-Python-Fourth-Edition/blob/main/chapter05/chapter05.ipynb.

Conceptualizing Haar cascades

When we talk about classifying objects and tracking their location, what exactly are we hoping to pinpoint? What constitutes a recognizable part of an object?Photographic images, even from a webcam, may contain a lot of detail for our (human) viewing pleasure. However, image detail tends to be unstable with respect to variations in lighting, viewing angle, viewing distance, camera shake, and digital noise. Moreover, even real differences in physical detail might not interest us for classification. Joseph Howse, one of this book's authors, was taught in school that no two snowflakes look alike under a microscope. Fortunately, as a Canadian child, he had already learned how to recognize snowflakes without a microscope, as the similarities are more obvious in bulk.Hence, having some means of abstracting image detail is useful in producing stable classification and tracking results. The abstractions are called **features**, which are said to be **extracted** from the image data. There should be far fewer features than pixels, though any pixel might influence multiple features. A set of features is represented as a **vector** (conceptually, a set of coordinates in a multidimensional space), and the level of similarity between two images can be evaluated based on some measure of the distance between the images' corresponding feature vectors.

> Later, in *Chapter 6*, *Retrieving Images and Searching Using Image Descriptors*, we will explore several kinds of features, as well as advanced ways of describing and matching sets of features.

Haar-like features are one type of feature that is often applied to real-time face detection. They were first used for this purpose in the paper *Robust Real-Time Face Detection*, by Paul Viola and Michael Jones (*International Journal of Computer Vision 57(2), 137–154, Kluwer Academic Publishers, 2001*). An electronic version of this paper is available at http://comp3204.ecs.soton.ac.uk/cw/viola04ijcv.pdf.Each Haar-like feature describes the pattern of contrast among adjacent image regions. For example, edges, vertices, and thin lines each generate a kind of feature. Some features are distinctive in the sense that they typically occur in a certain class of object (such as a face) but not in other objects. These distinctive features can be organized into a hierarchy, called a **cascade**, in which the highest layers contain features of greatest distinctiveness, enabling a classifier to quickly reject subjects that lack these features. If a subject is a good match for the higher-layer features, then the classifier considers the lower-layer features too in order to weed out more false positives.For any given subject, the features may vary depending on the scale of the image and the size of the neighborhood (the region of nearby pixels) in which contrast is being evaluated. The neighborhood's size is called the **window size**. To make a Haar cascade classifier **scale-invariant** or, in other words, robust to changes in scale, the window size is kept constant but images are rescaled a number of times; hence, at some level of rescaling, the size of an object (such as a face) may match the window size. Together, the original image and the rescaled images are called an **image pyramid**, and each successive level in this pyramid is a smaller rescaled image. OpenCV provides a scale-invariant classifier that can load a Haar cascade from an XML file in a particular format. Internally, this classifier converts any given image into an image pyramid.Haar cascades, as implemented in OpenCV, are not robust to changes in rotation or perspective. For example, an upside-down face is not considered similar to an upright face and a face viewed in profile is not considered similar to a face viewed from the front. A more complex and resource-intensive implementation could improve a Haar cascade's robustness to rotation by considering multiple transformations of images as well as multiple window sizes. However, we will confine ourselves to the implementation in OpenCV.

Getting Haar cascade data

Your installation of OpenCV 5 should contain a subfolder called `data` . The path to this folder is stored in an OpenCV variable called `cv2.data.haarcascades` .The `data` folder contains XML files that can be loaded by an OpenCV class called `cv2.CascadeClassifier` . An instance of this class interprets a given XML file as a Haar cascade, which provides a detection model for a type of object such as a face. `cv2.CascadeClassifier` can detect this type of object in any image. As usual, we could obtain a still image from a file, or we could obtain a series of frames from a video file or a video camera.From the `data` folder, we will use the following cascade files:

```
haarcascade_frontalface_default.xml
haarcascade_eye.xml
```

As their names suggest, these cascades are for detecting faces and eyes. They require a frontal, upright view of the subject. We will use them later when building a face detector.

> If you are curious about how these cascade files are generated, you can find more information in Joseph Howse's book, *OpenCV 4 for Secret Agents* (Packt Publishing, 2019), specifically in *Chapter 3*, *Training a Smart Alarm to Recognize the Villain and His Cat*. With a lot of patience and a reasonably powerful computer, you can make your own cascades and train them for various types of objects.

Using OpenCV to perform face detection

With `cv2.CascadeClassifier` , it makes little difference whether we perform face detection on a still image or a video feed. The latter is just a sequential version of the former: face detection on a video is simply face detection applied to each frame. Naturally, with more advanced techniques, it would be possible to track a detected face continuously across multiple frames and determine that the face is the same one in each frame. However, it is good to know that a basic sequential approach also works.Let's go ahead and detect some faces.

Performing face detection on a still image

The first and most basic way to perform face detection is to load an image and detect faces in it. To make the result visually meaningful, we will draw rectangles around faces in the original image. Remembering that the face detector is designed for upright, frontal faces, we will use an image of a row of people, specifically woodcutters, standing shoulder to shoulder and facing the photographer or viewer.Let's go ahead and create the following basic script to perform face detection:

```
import cv2face_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_frontalface_default.xml')
img = cv2.imread('../images/woodcutters.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.08, 5)
for (x, y, w, h) in faces:
    img = cv2.rectangle(img, (x, y), (x+w, y+h), (255, 255255, 0), 2)

cv2.namedWindow('Woodcutters Detected!')
cv2.imshow('Woodcutters Detected!', img)
cv2.imwrite('./woodcutters_detected.pngpng', img)
cv2.waitKey(0)
```

Let's walk through the preceding code in small steps. First, we use the obligatory `cv2` import that you will find in every script in this book. Then, we declare a `face_cascade` variable, which is a `CascadeClassifier` object that loads a cascade for face detection:

```
face_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_frontalface_default.xml')
```

We then load our image file with `cv2.imread` and convert it into grayscale because `CascadeClassifier`, like many of OpenCV's classifiers, expects grayscale images. (If we try to use a color image, `CascadeClassifier` will internally convert it to grayscale anyway.) The next step, `face_cascade.detectMultiScale`, is where we perform the actual face detection:

```
img = cv2.imread('../images/woodcutters.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.08, 5)
```

The parameters of `detectMultiScale` include `scaleFactor` and `minNeighbors`. The `scaleFactor` argument, which should be greater than 1.0, determines the downscaling ratio of the image at each iteration of the face detection process. As we discussed earlier in the *Conceptualizing Haar cascades* section, this downscaling is intended to achieve scale invariance by matching various faces to the window size. The `minNeighbors` argument is the minimum number of overlapping detections that are required in order to retain a detection result. Normally, we expect that a face may be detected in multiple overlapping windows, and a greater number of overlapping detections makes us more confident that the detected face is truly a face.The value returned from the detection operation is a list of tuples that represent the face rectangles. OpenCV's `cv2.rectangle` function allows us to draw rectangles at the specified coordinates. `x` and `y` represent the left and top coordinates, while `w` and `h` represent the width and height of the face rectangle. We draw cyan rectangles around all of the faces we find by looping through the `faces` variable, making sure we use the original image for drawing, not the gray version:

```
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (255, 255255, 0), 2)
```

Lastly, we call `cv2.imshow` to display the resulting processed image and we call `cv2.imwrite` to save it. As usual, to prevent the image window from closing automatically, we insert a call to `waitKey`, which returns when the user presses any key:

```
cv2.imshow('Woodcutters Detected!', img)
cv2.imwrite('./woodcutters_detected.pngpng', img)
cv2.waitKey(0)
```

And there we go, three members of the band of woodcutters have been detected in our image, as shown in the following screenshot:

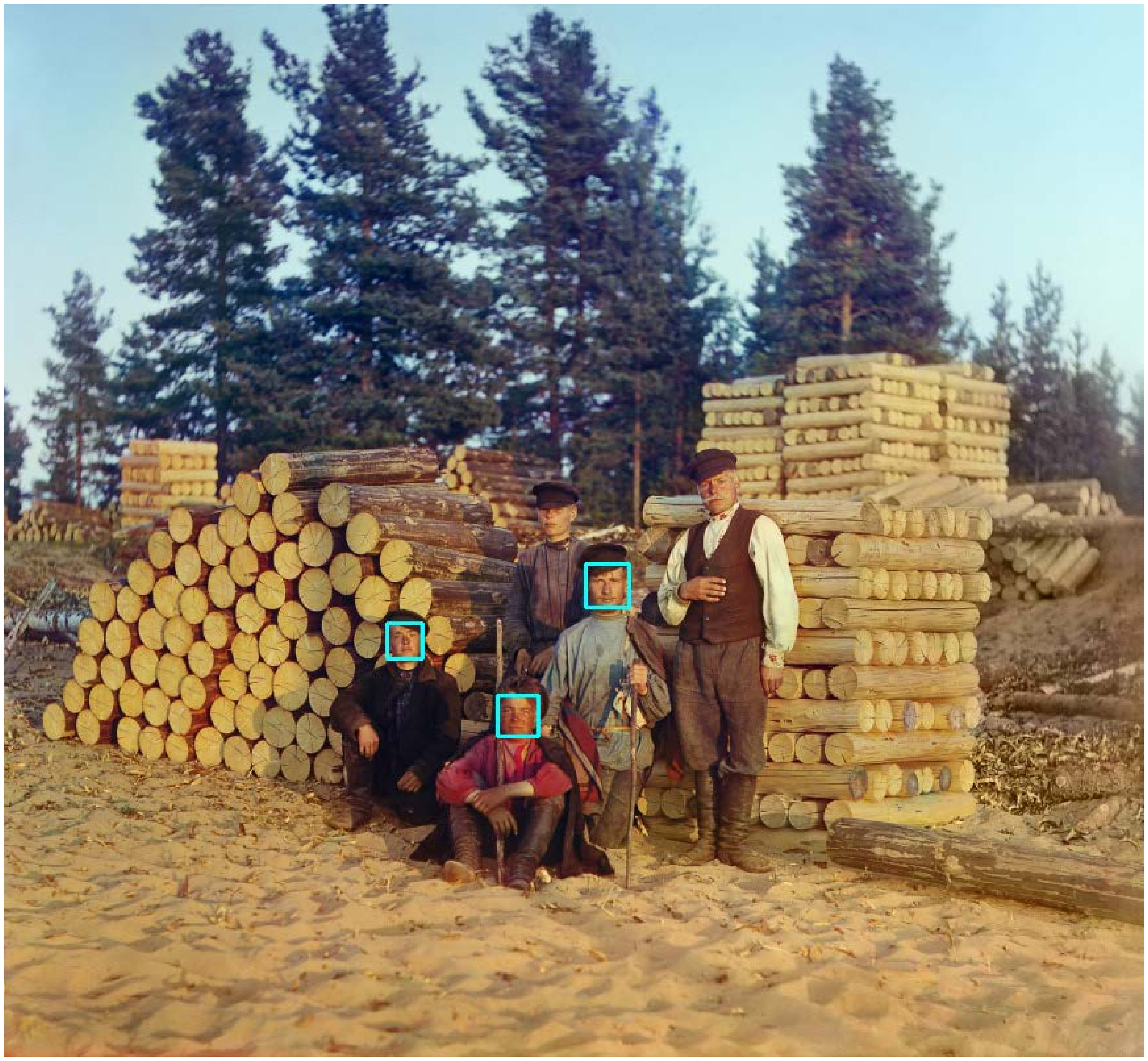


Figure 5.1: Face detection results of a photograph of woodcutters (Image credit: Prokudin-Gorsky)

The photograph in this example is the work of Sergey Prokudin-Gorsky (1863-1944), a pioneer of color photography. Tsar Nicholas II sponsored Prokudin-Gorsky to photograph people and places throughout the Russian Empire as a vast documentary project. Prokudin-Gorsky photographed these woodcutters near the Svir river, in northwestern Russia, in 1909.

Here, we have no false positive detections; all three rectangles really are woodcutters' faces. However, we do have two false negatives (woodcutters whose faces were not detected). Try adjusting the parameters of `face_cascade.detectMultiScale` to see how the results change. Then, let's proceed to a more interactive example.

Performing face detection on a video

We now understand how to perform face detection on a still image. As mentioned previously, we can repeat the process of face detection on each frame of a video (be it a camera feed or a pre-recorded video file).The next script will open a camera feed, read a frame, examine that frame for faces, and scan for eyes within the detected faces. Finally, it will draw blue rectangles around the faces and green rectangles around the eyes. Here is the script in its entirety:

```
import cv2
face_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_eye.xml')
camera = cv2.VideoCapture(0)
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(
            gray, 1.3, 5, minSize=(120, 120))
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            roi_gray = gray[y:y+h, x:x+w]
            eyes = eye_cascade.detectMultiScale(
                roi_gray, 1.11, 5, minSize=(40, 40))
            for (ex, ey, ew, eh) in eyes:
                cv2.rectangle(frame, (x+ex, y+ey),
                              (x+ex+ew, y+ey+eh), (0, 255, 0), 2)
        cv2.imshow('Face Detection', frame)
```

Let's break up the preceding sample into smaller, more digestible chunks:

1. As usual, we import the `cv2` module. After that, we initialize two `CascadeClassifier` objects, one for faces and another for eyes:

```
face_cascade = cv2.CascadeClassifier(    f'{cv2.data.haarcascades}haarcascade_frontalface_defaul
eye_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_eye.xml')
```

1. As in most of our interactive scripts, we open a camera feed and start iterating over frames. We continue until the user presses any key. Whenever we successfully capture a frame, we convert it into grayscale as our first step in processing it:

```
camera = cv2.VideoCapture(0)
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

1. We detect faces with the `detectMultiScale` method of our face detector. As we have previously done, we use the `scaleFactor` and `minNeighbors` arguments. We also use the `minSize` argument to specify a minimum size of a face, specifically 120x120. No attempt will be made to detect faces smaller than this. (Assuming that our user is sitting close to the camera, it is safe to say that the user's face will be larger than 120x120 pixels.) Here is the call to `detectMultiScale`:

```
faces = face_cascade.detectMultiScale(
    gray, 1.3, 5, minSize=(120, 120))
```

1. We iterate over the rectangles of the detected faces. We draw a blue border around each rectangle in the original color image. Then, within the same rectangular region of the grayscale image, we perform eye detection:

```
for (x, y, w, h) in faces:
    cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
    roi_gray = gray[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(
        roi_gray, 1.1, 5, minSize=(40, 40))
```

The eye detector is a bit less accurate than the face detector. You might see shadows, parts of the frames of glasses, or other regions of the face falsely detected as eyes. To improve the results, you could try defining `roi_gray` as a smaller region of the face, since we can make a good guess about the eyes' location in an upright face. You could also try using a `maxSize` argument to avoid false positives that are too large to be eyes. Also, you could adjust `minSize` and `maxSize` so that the dimensions are proportional to `w` and `h`, the size of the detected face. As an exercise, feel free to experiment with changes to these and other parameters.

1. We loop through the resulting eye rectangles and draw green outlines around them:

```
for (ex, ey, ew, eh) in eyes:
    cv2.rectangle(frame, (x+ex, y+ey),
                  (x+ex+ew, y+ey+eh), (0, 255, 0), 2)
```

1. Finally, we show the resulting frame in the window:

```
cv2.imshow('Face Detection', frame)
```

Run the script. If our detectors produce accurate results, and if any face is within the field of view of the camera, you should see a blue rectangle around the face and a green rectangle around each eye, as shown in this screenshot:

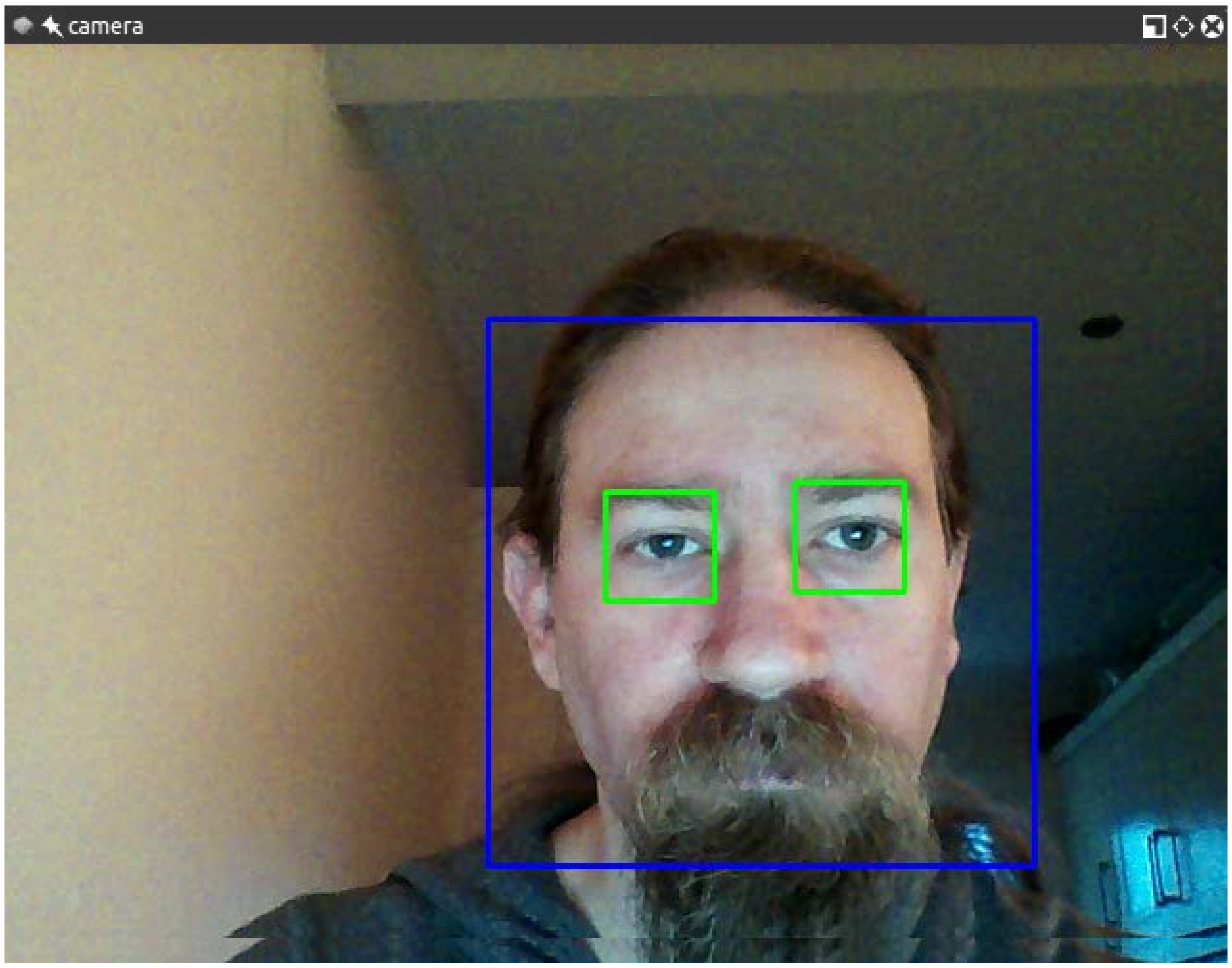


Figure 5.2: The resulting detection of a face and eyes in a photograph

Experiment with this script to see how the face and eye detectors perform under various conditions. Try a brighter or darker room. If you wear glasses, try removing them. Try various people's faces and various expressions. Adjust the detection parameters in the script to see how they affect the results. When you are satisfied, let's consider what else we can do with faces in OpenCV.

Performing face recognition

Detecting faces is a fantastic feature of OpenCV and one that constitutes the basis for a more advanced operation: face recognition. What is face recognition? It is the ability of a program, given an image or a video feed containing a person's face, to identify that person. One of the ways to achieve this (and the approach adopted by OpenCV) is to *train* the program by feeding it a set of classified pictures (a facial database) and perform recognition based on the features of those pictures.Another important feature of OpenCV's face recognition module is that each recognition has a confidence score, which allows us to set thresholds in real-life applications to limit the incidence of false identifications.Let's start from the very beginning; to perform face recognition, we need faces to recognize. We fulfill this requirement in two ways: supply the images ourselves or obtain freely available face databases. A large directory of face databases is available online at http://www.face-rec.org/databases/. Here are a few notable examples from the directory:

- Yale Face Database (Yalefaces): http://vision.ucsd.edu/content/yale-face-database
- Extended Yale Face Database B: http://vision.ucsd.edu/content/extended-yale-face-database-b-b
- Database of Faces (from AT&T Laboratories Cambridge): https://cam-orl.co.uk/facedatabase.html

If we trained a face recognizer on these samples, we would then have to run face recognition on an image that contains the face of one of the sampled people. This process might be educational, but perhaps not as satisfying as providing images of our own. You probably had the same thought that many computer vision learners have had: I wonder if I can write a program that recognizes my face with a certain degree of confidence. Indeed, you can and soon you will!

Generating the data for face recognition

Let's go ahead and write a script that will generate those images for us. A few images containing different expressions are all that we need, but it is preferable that the training images are square and are all the same size. Our sample script uses a size of 200x200, but most freely available datasets have smaller images than this.Here is the script itself:

```
import cv2
import os
output_folder = '../data/at/jm'
if not os.path.exists(output_folder):
    os.makedirs(output_folder)
face_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_frontalface_default.xml')
camera = cv2.VideoCapture(0)
count = 0
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(
            gray, 1.3, 5, minSize=(120, 120))
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            face_img = cv2.resize(gray[y:y+h, x:x+w], (200, 200))
            face_filename = '%s/%d.pgm' % (output_folder, count)
            cv2.imwrite(face_filename, face_img)
            count += 1
        cv2.imshow('Capturing Faces...', frame)
```

Here, we are generating sample images by building on our newfound knowledge of how to detect a face in a video feed. We are detecting a face, cropping that region of the grayscale-converted frame, resizing it to be 200x200 pixels, and saving it as a PGM file with a name in a particular folder (in this case, `jm`, one of the authors' initials;

you can use your own initials). Like many of our windowed applications, this one runs until the user presses any key.The `count` variable is present because we needed progressive names for the images. Run the script for a few seconds, change your facial expression a few times, and check the destination folder you specified in the script. You will find a number of images of your face, grayed, resized, and named with the format `<count>.pgm`.Modify the `output_folder` variable to make it match your name. For example, you might choose `'../data/at/my_name'`. Run the script, wait for it to detect your face in a number of frames (say, 20 or more), and then press any key to quit. Now, modify the `output_folder` variable again to make it match the name of a friend whom you also want to recognize. For example, you might choose `'../data/at/name_of_my_friend'`. Do not change the base part of the folder (in this case, `'../data/at'`) because later, in the *Loading the training data for face recognition* section, we will write code that loads the training images from all of the subfolders of this base folder. Ask your friend to sit in front of the camera, run the script again, let it detect your friend's face in a number of frames, and then quit. Repeat this process for any additional people you might want to recognize.Let's now move on to try and recognize the user's face in a video feed. This should be fun!

Choosing a face recognition algorithm

OpenCV 5 implements three different algorithms for recognizing faces: **Eigenfaces**, **Fisherfaces**, and **Local Binary Pattern Histograms** (**LBPHs**). Eigenfaces and Fisherfaces are derived from a more general-purpose algorithm called **Principal Component Analysis** (**PCA**). For a detailed description of the algorithms, refer to the following links:

- **PCA**: *A Tutorial on Principal Component Analysis* (2013), by Jonathon Shlens, is available at http://arxiv.org/pdf/1404.1100v1.pdf. This algorithm was invented in 1901 by Karl Pearson, and the original paper, *On Lines and Planes of Closest Fit to Systems of Points in Space*, is available at http://pca.narod.ru/pearson1901.pdf.
- **Eigenfaces**: The paper *Eigenfaces for Recognition* (1991), by Matthew Turk and Alex Pentland, is available at http://www.cs.ucsb.edu/~mturk/Papers/jcn.pdf.
- **Fisherfaces**: The seminal paper *The Use of Multiple Measurements in Taxonomic Problems* (1936), by R. A. Fisher, is available at http://onlinelibrary.wiley.com/doi/10.1111/j.1469-1809.1936.tb02137.x/pdf.
- **Local Binary Pattern**: The first paper describing this algorithm is *Performance evaluation of texture measures with classification based on Kullback discrimination of distributions* (1994), by T. Ojala, M. Pietikainen, and D. Harwood. It is available at https://ieeexplore.ieee.org/document/576366.

For this book's purposes, let's just take a high-level overview of the algorithms. First and foremost, they all follow a similar process; they take a set of classified observations (our face database, containing numerous samples per individual), train a model based on it, perform an analysis of face images (which may be face regions that we detected in an image or video), and determine two things: the subject's identity and a measure of confidence that this identification is correct. The latter is commonly known as the **confidence score**.Eigenfaces performs PCA, which identifies principal components of a certain set of observations (again, your face database), calculates the divergence of the current observation (the face being detected in an image or frame) compared to the dataset, and produces a value. The smaller the value, the smaller the difference between the face database and the detected face; hence, a value of 0 is an exact match.Fisherfaces also derives from PCA and evolves the concept, applying more complex logic. While computationally more intensive, it tends to yield more accurate results than Eigenfaces.LBPH instead divides a detected face into small cells and, for each cell, builds a histogram that describes whether the brightness of the image is increasing when comparing neighboring pixels in a given direction. This cell's histogram can be compared to the histogram of the corresponding cell in the model, producing a measure of similarity. Of the face recognizers in OpenCV, the implementation of LBPH is the only one that allows the model sample faces and the detected faces to be of a different shape and size. Hence, it is a convenient option, and the authors of this book find that its accuracy compares favorably to the other two options.Despite the algorithms' differences, OpenCV provides a similar interface for all three, as we shall soon see.

Loading the training data for face recognition

Regardless of our choice of face recognition algorithm, we can load the training images in the same way. Earlier, in the *Generating the data for face recognition* section, we generated training images and saved them in folders that were organized according to people's names or initials. For example, the following folder structure could contain sample face images of this book's authors, Joseph Howse (`jh`) and Joe Minichino (`jm`):

```
../
  data/
    at/
      jh/
      jm/
```

Let's write a script that loads these images and labels them in a way that OpenCV's face recognizers will understand. To work with the filesystem and the data, we will use the Python standard library's `os` module, as well as the `cv2` and `numpy` modules. Let's create a script that starts with the following `import` statements:

```
import os
import cv2
import numpy
```

Let's add the following `read_images` function, which walks through a directory's subdirectories, loads the images and shows each of them for user information purposes, resizes them to a specified size, and puts the resized images in a list. The order of this list has no significance in itself but, at the same time, our function builds two corresponding lists that describe the images: first, a corresponding list of people's names or initials (based on the subfolder names), and second, a corresponding list of labels or numeric IDs associated with the loaded images. For example, `jh` could be a name and `0` could be the label for all images that were loaded from the `jh` subfolder. Finally, the function converts the lists of images and labels into NumPy arrays, and it returns three variables: the list of names, the NumPy array of images, and the NumPy array of labels. Here is the function's implementation:

```
def read_images(path, image_size):
    names = []
    training_images, training_labels = [], []
    label = 0
    for dirname, subdirnames, filenames in os.walk(path):
        for subdirname in subdirnames:
            names.append(subdirname)
            subject_path = os.path.join(dirname, subdirname)
            for filename in os.listdir(subject_path):
                img = cv2.imread(os.path.join(subject_path, filename),
                                 cv2.IMREAD_GRAYSCALE)
                if img is None:
                    # The file cannot be loaded as an image.
                    # Skip it.
                    continue
                img = cv2.resize(img, image_size)
                cv2.imshow('training', img)
                cv2.waitKey(5)
                training_images.append(img)
                training_labels.append(label)
            label += 1
    training_images = numpy.asarray(training_images, numpy.uint8)
    training_labels = numpy.asarray(training_labels, numpy.int32)
    return names, training_images, training_labels
```

Let's call our `read_images` function by adding code such as the following:

```
path_to_training_images = '../data/at'
training_image_size = (200, 200)
names, training_images, training_labels = read_images(
    path_to_training_images, training_image_size)
```

Edit the `path_to_training_images` variable in the preceding code block to ensure that it matches the base folder of the `output_folder` variables you defined earlier in the code for the section *Generating the data for face recognition*.

So far, we have training data in a useful format but we have not yet created a face recognizer or performed any training. We will do so in the next section, where we continue the implementation of the same script.

Performing face recognition with Eigenfaces

Now that we have an array of training images and an array of their labels, we can create and train a face recognizer with just two more lines of code:

```
model = cv2.face.EigenFaceRecognizer_create()
model.train(training_images, training_labels)
```

What have we done here? We created the Eigenfaces face recognizer with OpenCV's `cv2.EigenFaceRecognizer_create` function, and we trained the recognizer by passing the arrays of images and labels (numeric IDs). Optionally, we could have passed two arguments to `cv2.EigenFaceRecognizer_create` :

- `num_components` : This is the number of components to keep for the PCA.
- `threshold` : This is a floating-point value specifying a confidence threshold. Faces with a confidence score below the threshold will be discarded. By default, the threshold is the maximum floating-point value so that no faces are discarded.

Having trained this model, we could save it to a file using code such as `model.write('my_model.xml')` . Later (for example, in another script), we could skip the training step and just load the pre-training model from the file using code such as `model.read('my_model.xml')` . However, for the sake of our simple demo, we will just train and test the model in one script without saving the model to a file.

To test this recognizer, let's use a face detector and a video feed from a camera. As we have done in previous scripts, we can use the following line of code to initialize the face detector:

```
face_cascade = cv2.CascadeClassifier(
    f'{cv2.data.haarcascades}haarcascade_frontalface_default.xml')
```

The following code initializes the camera feed, iterates over frames (until the user presses any key), and performs face detection and recognition on each frame:

```
camera = cv2.VideoCapture(0)
while (cv2.waitKey(1) == -1):
    success, frame = camera.read()
    if success:
        faces = face_cascade.detectMultiScale(frame, 1.3, 5)
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            roi_gray = gray[x:x+w, y:y+h]
            if roi_gray.size == 0:
                # The ROI is empty. Maybe the face is at the image edge.
                # Skip it.
                continue
            roi_gray = cv2.resize(roi_gray, training_image_size)
            label, confidence = model.predict(roi_gray)
            text = '%s, confidence=%.2f' % (names[label], confidence)
            cv2.putText(frame, text, (x, y - 20),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
        cv2.imshow('Face Recognition', frame)
```

Let's walk through the most important functionality of the preceding block of code. For each detected face, we convert and resize it so that we have a grayscale version that matches the expected size (in this case, 200x200 pixels as defined by the `training_image_size` variable in the previous section, *Loading the training data for face recognition*). Then, we pass the resized, grayscale face to the face recognizer's `predict` function. This returns a label and confidence score. We look up the person's name corresponding to the numeric label of that face. (Remember that we created the `names` array in the previous section, *Loading the training data for face recognition*.) We draw the name and confidence score in blue text above the recognized face. After iterating over all detected faces, we display the annotated image.

We have taken a simple approach to face detection and recognition, and it serves the purpose of enabling you to have a basic application running and understand the process of face recognition in OpenCV 5. To improve upon this approach and make it more robust, you could take further steps such as correctly aligning and rotating detected faces so that the accuracy of the recognition is maximized.

When you run the script, you should see something similar to the following screenshot:

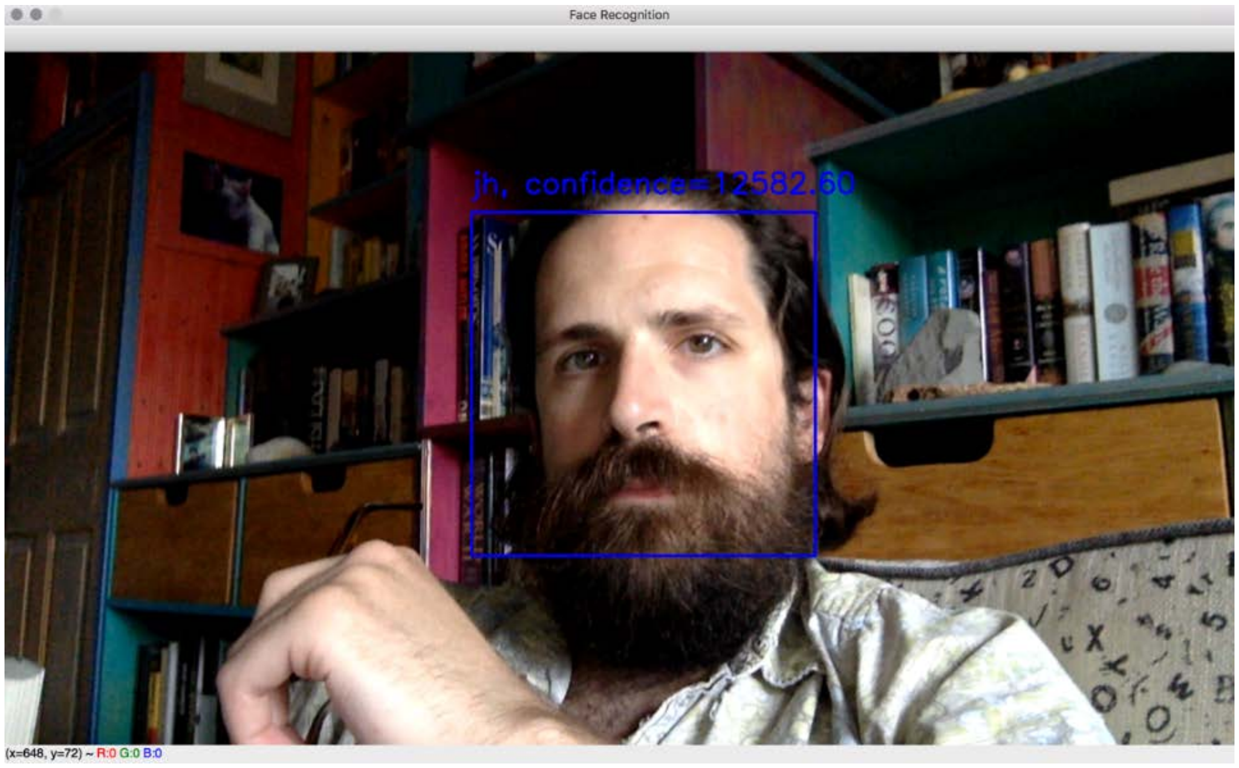


Figure 5.3: A face recognition result using live camera input

Next, let's consider how we would adapt these script to replace Eigenfaces with another face recognition algorithm.

Performing face recognition with Fisherfaces

What about Fisherfaces? The process does not change much; we simply need to instantiate a different algorithm. With default arguments, the declaration of our `model` variable would look like this:

```
model = cv2.face.FisherFaceRecognizer_create()
```

`cv2.face.FisherFaceRecognizer_create` takes the same two optional arguments as `cv2.createEigenFaceRecognizer_create` : the number of principal components to keep and the confidence threshold. As you might guess, these types of `*_create` functions are quite common in OpenCV for situations where several different algorithms share a common interface. Let’s look at one more example.

Performing face recognition with LBPH

For the LBPH algorithm, again, the process is similar. However, the algorithm factory takes the following optional parameters (in order):

- `radius` : The pixel distance between the neighbors that are used to calculate a cell's histogram (by default, 1)
- `neighbors` : The number of neighbors used to calculate a cell's histogram (by default, 8)
- `grid_x` : The number of cells into which the face is divided horizontally (by default, 8)
- `grid_y` : The number of cells into which the face is divided vertically (by default, 8)

- `confidence` : The confidence threshold (by default, the highest possible floating-point value so that no results are discarded)

With default arguments, the model declaration would look like this:

```
model = cv2.face.LBPHFaceRecognizer_create()
```

Note that, with LBPH, we do not need to resize images as the division into grids allows a comparison of patterns identified in each cell.

Having looked at the available face recognition algorithms, let’s now consider how to evaluate a recognition result.

Discarding results based on the confidence score

The `predict` method returns a tuple, in which the first element is the label of the recognized individual and the second is the confidence score. All algorithms come with the option of setting a confidence score threshold, which measures the distance of the recognized face from the original model; therefore, a score of 0 signifies an exact match.There may be cases in which you would rather retain all recognitions and then apply further processing, so you can come up with your own algorithms to estimate the confidence score of a recognition. For example, if you are trying to identify people in a video, you may want to analyze the confidence score in subsequent frames to establish whether the recognition was successful or not. In this case, you can inspect the confidence score obtained by the algorithm and draw your own conclusions.

The typical range of the confidence score depends on the algorithm. Eigenfaces and Fisherfaces produce values (roughly) in the range of 0 to 20,000, with any score below 4,000-5,000 being a quite confident recognition. For LBPH, a good recognition scores (roughly) below 50, and any value above 80 is considered a poor confidence score.

A normal custom approach would be to hold off drawing a rectangle around a recognized face until we have a number of frames with a good confidence score (where “good” is an arbitrary threshold we must choose, based on our algorithm and use case), but you have total freedom to use OpenCV's face recognition module to tailor your application to your needs. Next, let’s see how face detection and recognition work in a specialized use case.

Swapping faces in infrared

Face detection and recognition are not limited to the visible spectrum of light. With a **Near-Infrared** (**NIR**) camera and NIR light source, face detection and recognition are possible even when a scene appears totally dark to the human eye. This capability is quite useful in security and surveillance applications.We studied the basic usage of NIR depth cameras, such as the Asus Xtion PRO, in *Chapter 4*, *Depth Estimation and Segmentation*. We extended the object-oriented code of our interactive application, Cameo. We captured frames from a depth camera. Based on depth, we segmented each frame into a main layer (such as the user's face) and other layers. We painted the other layers black. This achieved the effect of hiding the background so that only the main layer (the user's face) appeared on-screen in the interactive video feed.Now, let's modify Cameo to do something that exercises our previous skills in depth segmentation and our new skills in face detection. Let's detect faces and, when we detect at least two faces in a frame, let's swap the faces so that one person's head appears atop another person's body. Rather than copying all pixels in a detected face rectangle, we will only copy the pixels that are part of the main depth layer for that rectangle. This should achieve the effect of swapping faces but not the background pixels surrounding the faces.Once the changes are complete, Cameo will be able to produce output such as the following screenshot:

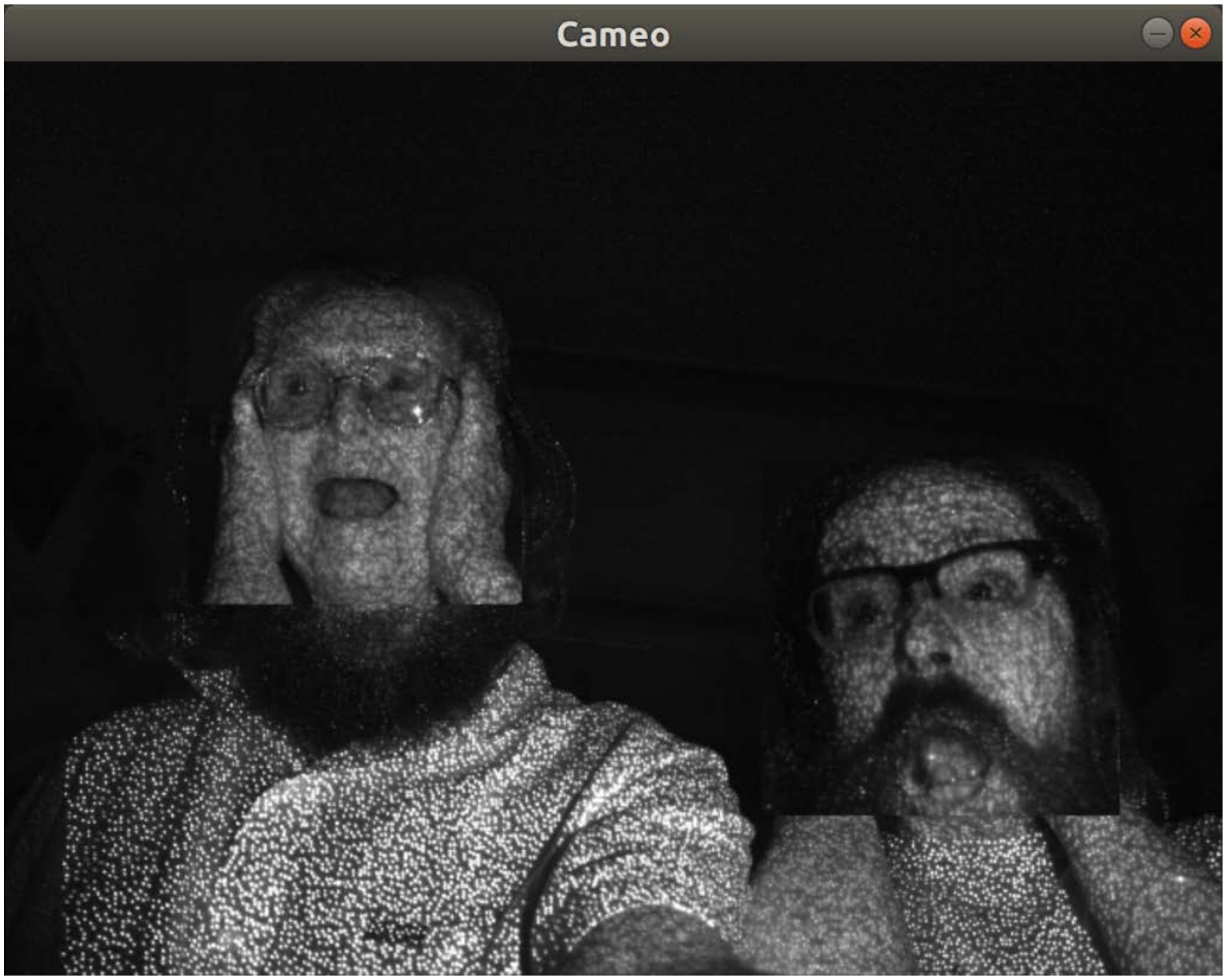


Figure 5.4: The result of swapping two faces in a frame from an infrared camera.

Here, we see the face of Joseph Howse swapped with the face of Janet Howse, his mother. Although Cameo is copying pixels from rectangular regions (and this is clearly visible at the bottom of the swapped regions, in the foreground), some of the background pixels within the rectangles are masked based on depth and thus are not swapped, so we do not see rectangular edges everywhere.You can find all of the relevant changes to the Cameo source code in this book's repository at https://github.com/PacktPublishing/Learning-OpenCV-5-Computer-Vision-with-Python-Fourth-Edition, specifically in the `chapter05/cameo` folder. For brevity, we will not discuss all of the changes here in this book, but we will cover some of the highlights in the next two subsections, *Modifying the application's loop* and *Masking a copy operation*.

Modifying the application's loop

To support face swapping, the Cameo project has two new modules called `rects` and `trackers`. The `rects` module contains functions for copying and swapping rectangles, with an optional mask to limit the copy or swap operation to particular pixels. The `trackers` module contains a class called `FaceTracker`, which adapts OpenCV's face detection functionality to an object-oriented style of programming.As we have covered OpenCV's face detection functionality earlier in this chapter, and we have demonstrated an object-oriented programming style in previous chapters, we will not go into the `FaceTracker` implementation here. Instead, you may look at it in this book's repository.Let's open `cameo.py` so that we can walk through the overall changes to the application:

1. Near the top of the file, we need to import our new modules, as `highlightedhighlighted` in the following code block:

```
import cv2
import depth
import filters
from managers import WindowManager, CaptureManager
import rects
from trackers import FaceTracker
```

1. Now, let's turn our attention to changes in the `__init__` method of our `CameoDepth` class. Our updated application uses an instance of `FaceTracker`. As part of its functionality, `FaceTracker` can draw rectangles around detected faces. Let's give Cameo's user the option to enable or disable the drawing of face rectangles. We will keep track of the currently selected option via a Boolean variable. The following code block `highlights` the necessary changes to initialize the `FaceTracker` object and the Boolean variable:

```
class CameoDepth(Cameo):
    def __init__(self):
        self._windowManager = WindowManager('Cameo',
                                            self.onKeypress)
        #device = cv2.CAP_OPENNI2 # uncomment for Kinect
        device = cv2.CAP_OPENNI2_ASUS # uncomment for Xtion
        self._captureManager = CaptureManager(
            cv2.VideoCapture(device), self._windowManager, True)
        self._faceTracker = FaceTracker()
        self._shouldDrawDebugRects = False
        self._curveFilter = filters.BGRPortraCurveFilter()
```

We make use of the `FaceTracker` object in the `run` method of `CameoDepth`, which contains the application's main loop that captures and processes frames. Every time we successfully capture a frame, we call methods of `FaceTracker` to update the face detection result and get the latest detected faces. Then, for each face, we create a mask based on the depth camera's disparity map. (Previously, in *Chapter 4*, *Depth Estimation and Segmentation*, we created such a mask for the entire image instead of a mask for each face rectangle.)

1. Then, we call a function, `rects.swapRects`, to perform a masked swap of the face rectangles. (We will look at the implementation of `swapRects` a little later, in the *Masking a copy operation* section.) Depending on the currently selected option, we might tell `FaceTracker` to draw rectangles around the faces. All of the relevant changes are `highlighted` in the following code block:

```
    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            # ... The logic for capturing a frame is unchanged ...
            if frame is not None:
                self._faceTracker.update(frame)
                faces = self._faceTracker.faces
                masks = [
                    depth.createMedianMask(
                        disparityMap, validDepthMask,
                        face.faceRect) \
                    for face in faces
                ]
                rects.swapRects(frame, frame,
                                [face.faceRect for face in faces],
                                masks)
                if self._captureManager.channel == cv2.CAP_OPENNI_BGR_IMAGE:
                    # A BGR frame was captured.
                    # Apply filters to it.
                    filters.strokeEdges(frame, frame)
                    self._curveFilter.apply(frame, frame)
                if self._shouldDrawDebugRects:
                    self._faceTracker.drawDebugRects(frame)
            self._captureManager.exitFrame()
            self._windowManager.processEvents()
```

1. Finally, let's modify the `onKeypress` method so that the user can hit the X key to start or stop displaying rectangles around detected faces. Again, the relevant changes are `highlighted` in the following code block:

```
    def onKeypress(self, keycode):
        """Handle a keypress.
        space -> Take a screenshot.
        tab -> Start/stop recording a screencast.
        x -> Start/stop drawing debug rectangles around faces.
        escape -> Quit.
        """
        if keycode == 32: # space
            self._captureManager.writeImage('screenshot.png')
        elif keycode == 9: # tab
            if not self._captureManager.isWritingVideo:
                self._captureManager.startWritingVideo(
                    'screencast.avi')
            else:
                self._captureManager.stopWritingVideo()
        elif keycode == 120: # x
            self._shouldDrawDebugRects = \
                not self._shouldDrawDebugRects
        elif keycode == 27: # escape
            self._windowManager.destroyWindow()
```

Next, let's look at the implementation of the `rects` module that we imported earlier in this section.

Masking a copy operation

The `rects` module is implemented in `rects.py`. We already saw a call to the `rects.swapRects` function in the previous section. However, before we consider the implementation of `swapRects`, we first need a more basic `copyRect` function.As far back as *Chapter 2*, *Handling Files, Cameras, and GUIs*, we learned how to copy data from one rectangular **region of interest** (**ROI**) to another using NumPy's slicing syntax. Outside the ROIs, the source and destination images were unaffected. Now, we want to apply further limits to this copy operation. We want to use a given mask that has the same dimensions as the source rectangle.We shall copy only those pixels in the source rectangle where the mask's value is not zero. Other pixels shall retain their old values from the destination image. This logic, with an array of conditions and two arrays of possible output values, can be expressed concisely with the `numpy.where` function.With this approach in mind, let's consider our `copyRect` function. As arguments, it takes a source and destination image, a source and destination rectangle, and a mask. The latter may be `None`, in which case, we simply resize the content of the source rectangle to match the destination rectangle and then assign the resulting resized content to the destination rectangle. Otherwise, we next ensure that the mask and the images have the same number of channels. We assume that the mask has one channel but the images may have three channels (BGR). We can add duplicate channels to mask using the `repeat` and `reshape` methods of `numpy.array`. Finally, we perform the copy operation using `numpy.where`. The complete implementation is as follows:

```
def copyRect(src, dst, srcRect, dstRect, mask = None,
             interpolation = cv2.INTER_LINEAR):
    """Copy part of the source to part of the destination."""
    x0, y0, w0, h0 = srcRect
    x1, y1, w1, h1 = dstRect
    # Resize the contents of the source sub-rectangle.
    # Put the result in the destination sub-rectangle.
    if mask is None:
        dst[y1:y1+h1, x1:x1+w1] = \
            cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                       interpolation = interpolation)
    else:
        if not utils.isGray(src):
            # Convert the mask to 3 channels, like the image.
            mask = mask.repeat(3).reshape(h0, w0, 3)
        # Perform the copy, with the mask applied.
        dst[y1:y1+h1, x1:x1+w1] = \
            numpy.where(cv2.resize(mask, (w1, h1),
                                   interpolation = \
                                   cv2.INTER_NEAREST),
                        cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                                   interpolation = interpolation),
                        dst[y1:y1+h1, x1:x1+w1])
```

We also need to define a `swapRects` function, which uses `copyRect` to perform a circular swap of a list of rectangular regions. `swapRects` has a `masks` argument, which is a list of masks whose elements are passed to the respective `copyRect` calls. If the value of the `masks` argument is `None`, we pass `None` to every `copyRect` call. The following code shows the full implementation of `swapRects`:

```
def swapRects(src, dst, rects, masks = None,
              interpolation = cv2.INTER_LINEAR):
    """Copy the source with two or more sub-rectangles swapped."""
    if dst is not src:
        dst[:] = src
    numRects = len(rects)
    if numRects < 2:
        return
    if masks is None:
        masks = [None] * numRects
    # Copy the contents of the last rectangle into temporary storage.
    x, y, w, h = rects[numRects - 1]
    temp = src[y:y+h, x:x+w].copy()
    # Copy the contents of each rectangle into the next.
    i = numRects - 2
    while i >= 0:
        copyRect(src, dst, rects[i], rects[i+1], masks[i],
                 interpolation)
        i -= 1
    # Copy the temporarily stored content into the first rectangle.
    copyRect(temp, dst, (0, 0, w, h), rects[0], masks[numRects - 1],
             interpolation)
```

Note that the `mask` argument in `copyRect` and the `masks` argument in `swapRects` both have a default value of `None`. If no mask is specified, these functions copy or swap the entire contents of the rectangle or rectangles.

Summary

By now, you should have a good understanding of how face detection and face recognition work and how to implement them in Python and OpenCV 5.The accuracy of detection and recognition algorithms heavily depends on the quality of the training data, so make sure you provide your applications with a large number of training images covering a variety of expressions, poses, and lighting conditions. Later in this book, in *Chapter 11*, *Neutral Networks with OpenCV – an Introduction*, we will look at how to use several robust, pre-trained face detection models that build atop advanced algorithms and large sets of training data.As human beings, we might be predisposed to think that human faces are particularly recognizable. We might even be overconfident in our own face recognition abilities. However, in computer vision, there is nothing very special about human faces, and we can just as readily use algorithms to find and identify other things. We will begin to do so next in *Chapter 6*, *Retrieving Images and Searching Using Image Descriptors*.

6 Retrieving Images and Searching Using Image Descriptors

Similar to the human eyes and brain, OpenCV can detect the main features of an image and extract them into so-called image descriptors. These features can then be used as a database, enabling image-based searches. Moreover, we can use key points to stitch images together and compose a bigger image. (Think of putting together many pictures to form a 360° panorama.)This chapter will show you how to detect the features of an image with OpenCV and make use of them to match and search images. Over the course of this chapter, we will take sample images and detect their main features, and then try to find a region of another image that matches the sample image. We will also find the homography or spatial relationship between a sample image and a matching region of another image.More specifically, we will cover the following tasks:

- Detecting keypoints and extracting local descriptors around the keypoints using any of the following algorithms: **Harris corners**, **SIFT**, **SURF**, or **ORB**
- Matching keypoints using **brute-force algorithms** or the **FLANN algorithm**
- Filtering out bad matches using **KNN** and the **ratio test**
- Finding the homography between two sets of matching keypoints
- Searching a set of images to determine which one contains the best match for a reference image

We will finish this chapter by building a proof-of-concept forensic application. Given a reference image of a tattoo, we will search for a set of images of people in order to find a person with a matching tattoo.

Technical requirements

This chapter uses Python, OpenCV, and NumPy. In regards to OpenCV, we use the optional `opencv_contrib` modules, which include additional algorithms for keypoint detection and matching. To enable the**SURF** algorithm (whichis patented and *not* free for commercial use), we must configure the `opencv_contrib` modules with the `OPENCV_ENABLE_NONFREE` flag in CMake. Please refer to *Chapter 1*, *Setting Up OpenCV*, for installation instructions. Additionally, if you have not already installed Matplotlib, install it by running `$ pip install matplotlib` (or `$ pip3 install matplotlib`, depending on your environment).The complete code for this chapter can be found in this book's GitHub repository, https://github.com/PacktPublishing/Learning-OpenCV-5-Computer-Vision-with-Python-Fourth-Edition, in the `chapter06` folder. The sample images can be found in the `images` folder.A subset of the chapter's sample code can be edited and run interactively in Google Colab at https://colab.research.google.com/github/PacktPublishing/Learning-OpenCV-5-Computer-Vision-with-Python-Fourth-Edition/blob/main/chapter06/chapter06.ipynb.

Understanding types of feature detection and matching

A number of algorithms can be used to detect and describe features, and we will explore several of them in this section. The most commonly used feature detection and descriptor extraction algorithms in OpenCV are as follows:

- **Harris**: This algorithm is useful for detecting corners.
- **SIFT**: This algorithm is useful for detecting blobs.
- **SURF**: This algorithm is useful for detecting blobs.
- **FAST**: This algorithm is useful for detecting corners.
- **BRIEF**: This algorithm is useful for detecting blobs.
- **ORB**: This algorithm stands for **Oriented FAST and Rotated BRIEF**. It is useful for detecting a combination of corners and blobs.

Matching features can be performed with the following methods:

- Brute-force matching
- FLANN-based matching

Spatial verification can then be performed with homography.We have just introduced a lot of new terminology and algorithms. Now, we will go over their basic definitions.

Defining features

What is a feature, exactly? Why is a particular area of an image classifiable as a feature, while others are not? Broadly speaking, a feature is an area of interest in the image that is unique or easily recognizable. **Corners** and regions with a high density of textural detail are good features, while patterns that repeat themselves a lot and low-density regions (such as a blue sky) are not. Edges are good features as they tend to divide two regions with abrupt changes in the intensity (gray or color) values of an image. A **blob** (a region of an image that greatly differs from its surrounding areas) is also an interesting feature.Most feature detection algorithms revolve around the identification of corners, edges, and blobs, with some also focusing on the concept of a **ridge**, which you can conceptualize as the axis of symmetry of an elongated object. (Think, for example, about identifying a road in an image.)Some algorithms are better at identifying and extracting features of a certain type, so it is important to know what your input image is so that you can utilize the best tool in your OpenCV belt.

Detecting Harris corners

Let's start by finding corners using the Harris corner detection algorithm. We will do this by implementing an example. If you continue to study OpenCV beyond this book, you will find that chessboards are a common subject of analysis in computer vision, partly because a checkered pattern is suited to many types of feature detection, and partly because chess is a popular pastime, especially in Russia, which is the country of origin of many of OpenCV's developers.Here is our sample image of a chessboard and chess pieces:

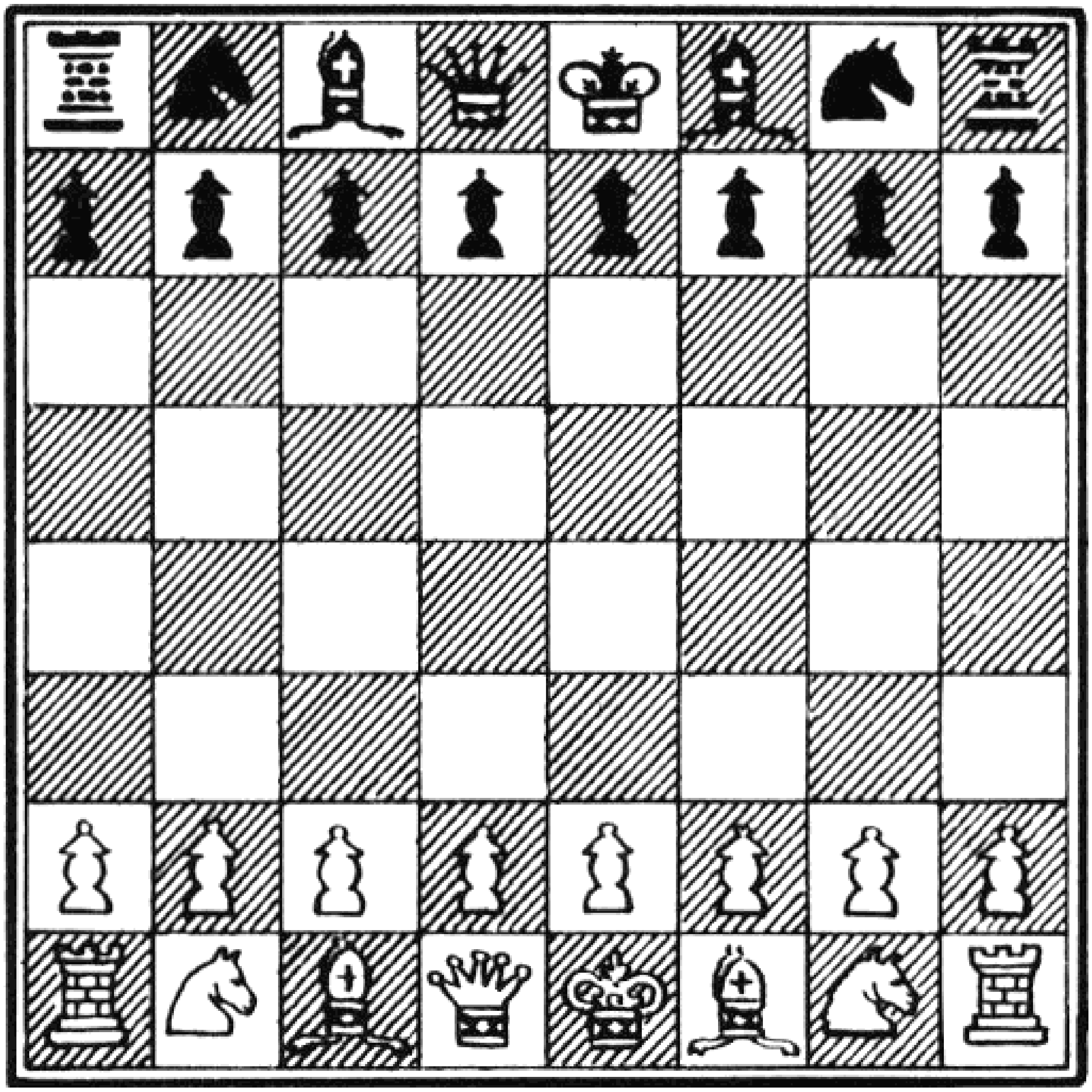


Figure 6.1: A chessboard, to be used for corner detection

OpenCV has a handy function called `cv2.cornerHarris`, which detects corners in an image. We can see this function at work in the following basic example:

```
import cv2
img = cv2.imread('../images/chess_board.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
dst = cv2.cornerHarris(gray, 2, 23, 0.04)
img[dst > 0.01 * dst.max()] = [0, 0, 255]
cv2.imshow('corners', img)
cv2.waitKey()
```

Let's analyze the code. After the usual imports, we load the chessboard image and convert it into grayscale. Then, we call the `cornerHarris` function:

```
dst = cv2.cornerHarris(gray, 2, 23, 0.04)
```

The most important parameter here is the third one, which defines the aperture or kernel size of the Sobel operator. The Sobel operator detects edges by measuring horizontal and vertical differences between pixel values in a neighborhood, and it does this using a kernel. The `cv2.cornerHarris` function uses a Sobel operator whose aperture is defined by this parameter. In plain English, the parameters define how sensitive corner detection is. It must be between 3 and 31 and be an odd value. With a low (highly sensitive) value of `3`, all those diagonal lines in the black squares of the chessboard will register as corners when they touch the border of the square. For a higher (less sensitive) value of `23`, only the corners of each square will be detected as corners. `cv2.cornerHarris` returns an image in floating-point format. Each value in this image represents a score for the corresponding pixel in the source image. A moderate or high score indicates that the pixel is likely to be a corner. Conversely, we can treat pixels with the lowest scores as non-corners. Consider the following line:

```
img[dst > 0.01 * dst.max()] = [0, 0, 255]
```

Here, we select pixels with scores that are at least 1% of the highest score, and we color these pixels red in the original image. Here is the result:

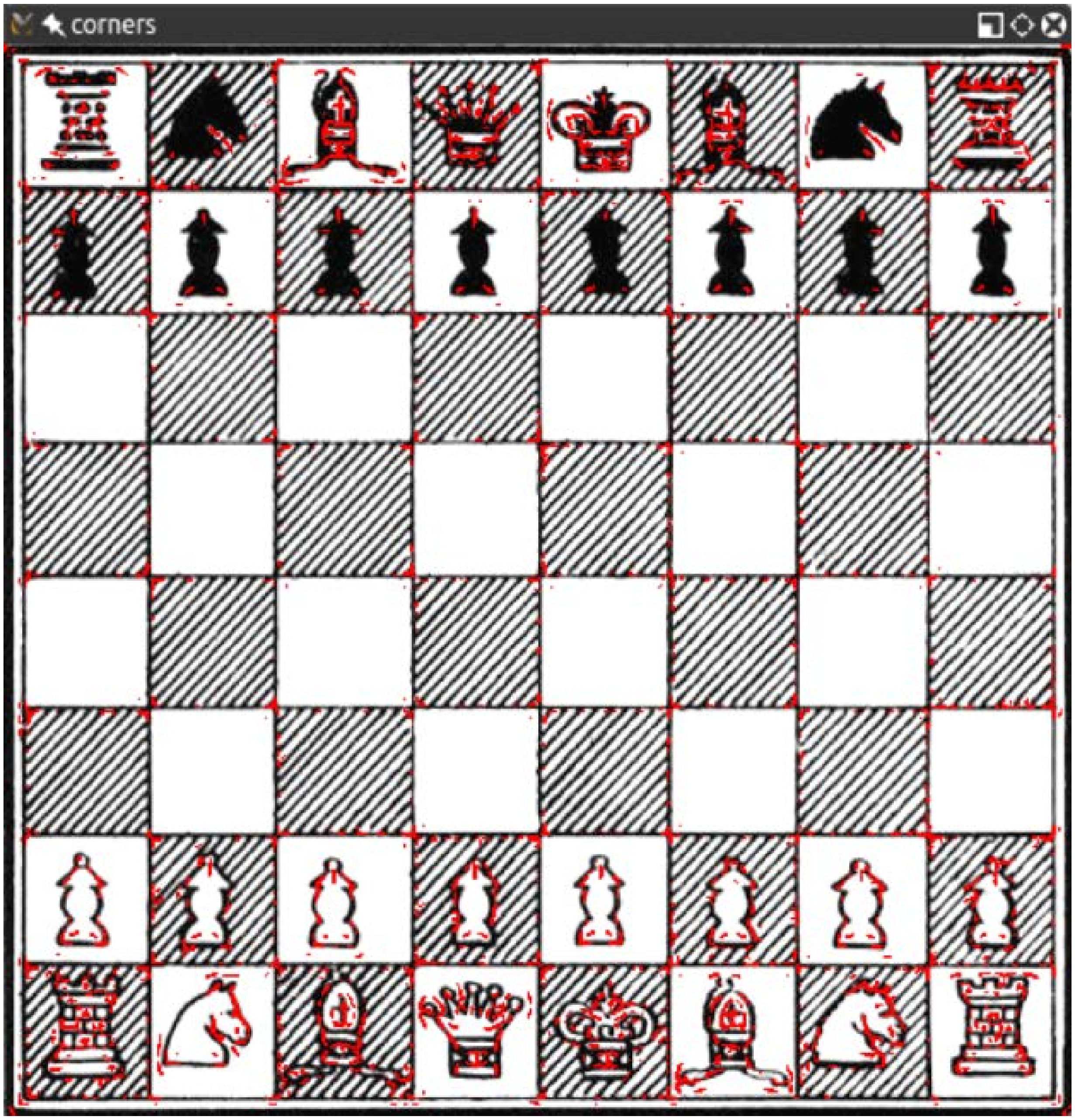


Figure 6.2: Corners detections in a chessboard

Great! Nearly all the detected corners are marked in red. The marked points include nearly all the corners of the chessboard's squares.

> If we tweak the second parameter in `cv2.cornerHarris` , we will see that smaller regions (for a smaller parameter value) or larger regions (for a larger parameter value) will be detected as corners. This parameter is called the block size.

Detecting DoG features and extracting SIFT descriptors

The preceding technique, which uses `cv2.cornerHarris` , is great for detecting corners and has a distinct advantage because even if the image is rotated corners are still the corners. However, if we scale an image to a smaller or larger size, some parts of the image may lose or even gain a corner quality.For example, take a look at the following corner detections in an image of the F1 Italian Grand Prix track:

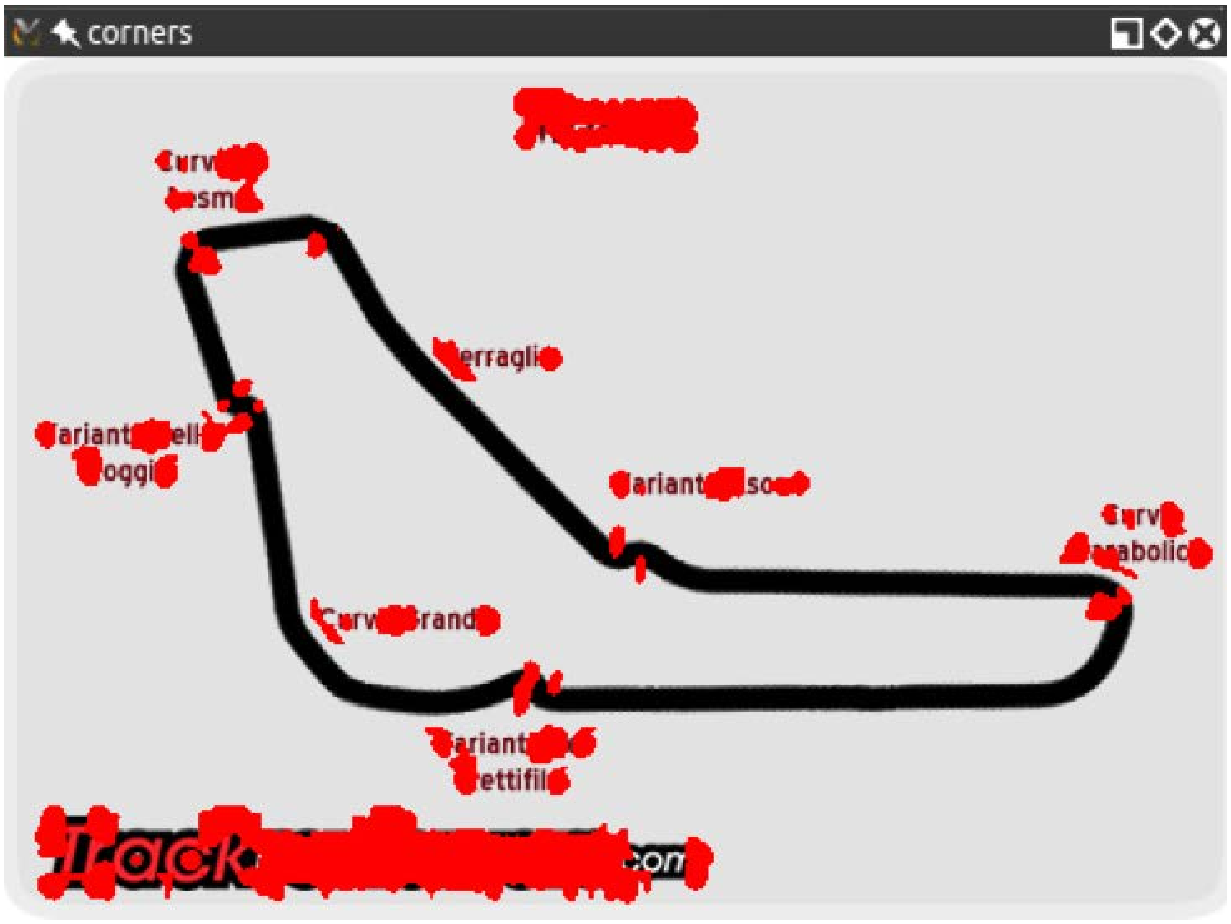


Figure 6.3: Corner detections in an image of the F1 Italian Grand Prix track

Here is the corner detection result with a smaller version of the same image:

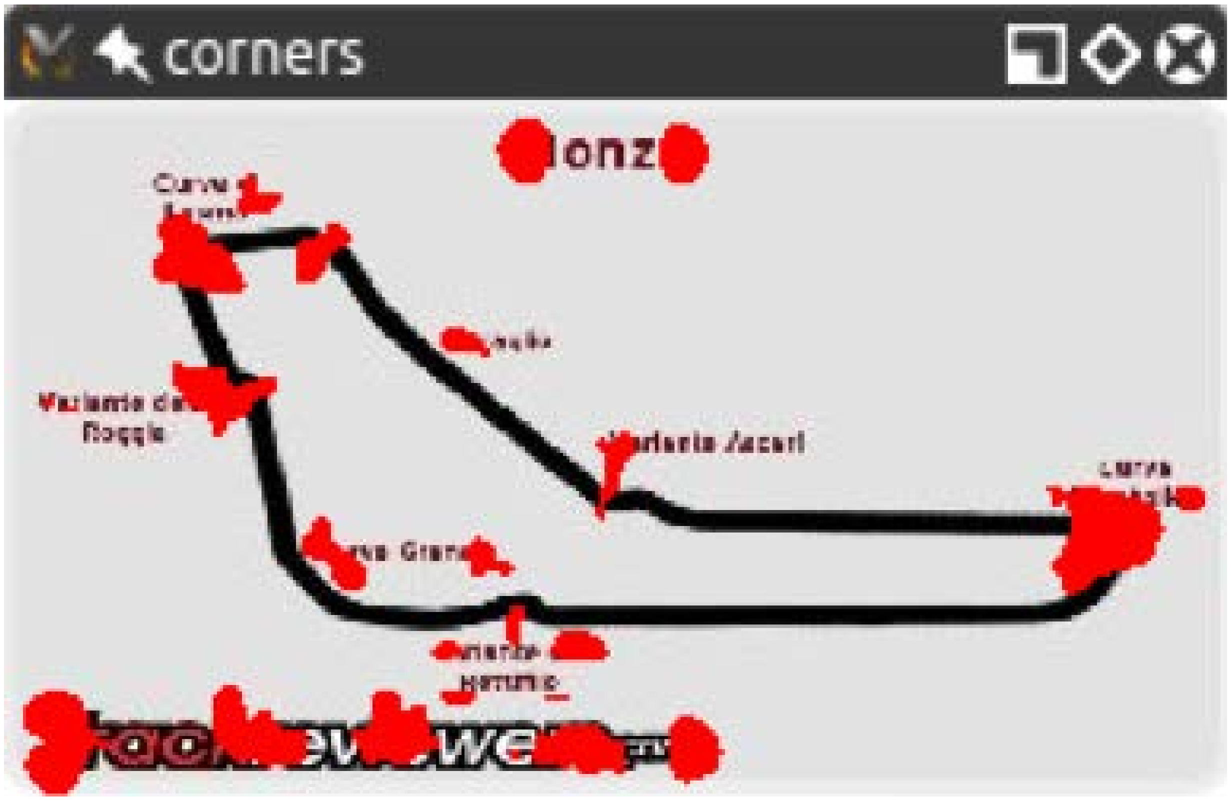


Figure 6.4: Corner detections in a smaller image of the F1 Italian Grand Prix track

You will notice how the corners are a lot more condensed; however, even though we gained some corners, we lost others! In particular, let's examine the **Variante Ascari** chicane, which looks like a squiggle at the end of the part of the track that runs straight from northwest to southeast. In the larger version of the image, both the entrance and the apex of the double bend were detected as corners. In the smaller image, the apex is not detected as such. If we further reduce the image, at some scale, we will lose the entrance to that chicane too.This loss of features raises an issue; we need an algorithm that works regardless of the scale of the image. Enter **Scale-Invariant Feature Transform** (**SIFT**). While the name may sound a bit mysterious, now that we know what problem we are trying to solve, it actually makes sense. We need a function (a transform) that will detect features (a feature transform) and will not output different results depending on the scale of the image (a scale-invariant feature transform). Note that SIFT does not detect keypoints. Keypoint detection is done with the **Difference of Gaussians** (**DoG**), while SIFT describes the region surrounding the keypoints by means of a feature vector.A quick introduction to the DoG is in order. Previously, in *Chapter 3*, *Processing Images with OpenCV*, we talked about low pass filters and blurring operations, and specifically the `cv2.GaussianBlur` function. DoG is the result of applying different Gaussian filters to the same image. Previously, we applied this type of technique for edge detection, and the idea is the same here. The final result of a DoG operation contains areas of interest (keypoints), which are then going to be described through SIFT.Let's see how DoG and SIFT behave in the following image, which is full of corners and features:

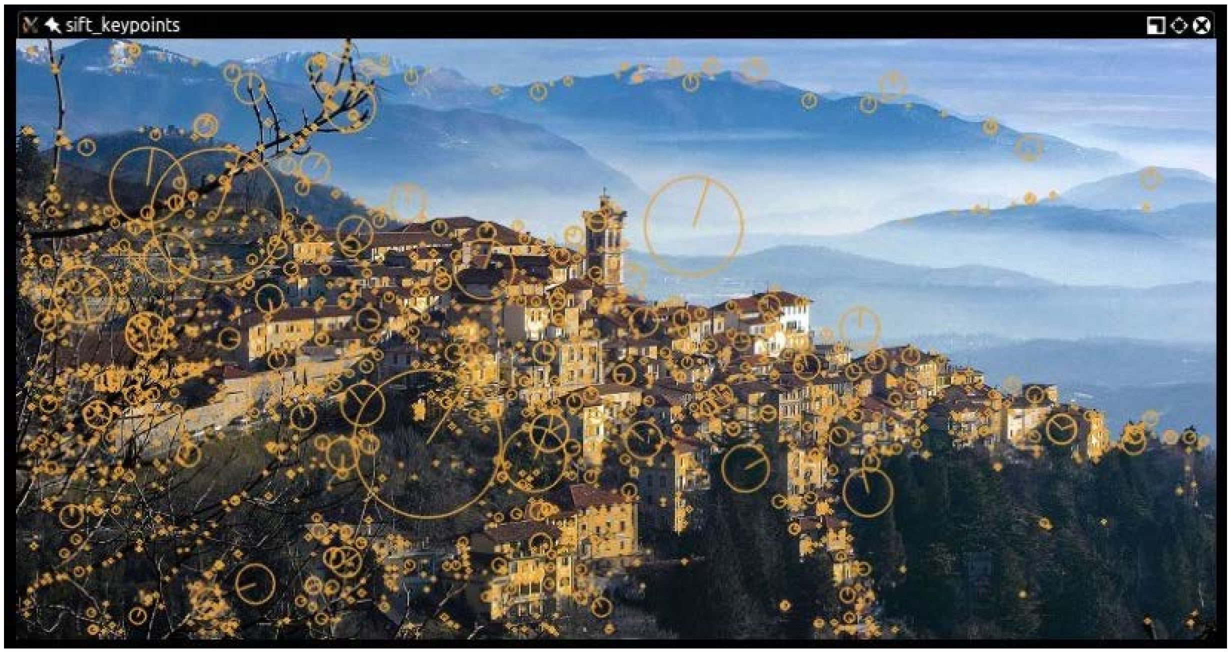


Figure 6.5: SIFT descriptors for an image of Varese, Lombardy, Italy

Here, the beautiful panorama of Varese (in Lombardy, Italy) gains a new type of fame as a subject of computer vision. Here is the code that produces this processed image:

```
import cv2
img = cv2.imread('../images/varese.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(gray, None)
cv2.drawKeypoints(img, keypoints, img, (51, 163, 236),
                  cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('sift_keypoints', img)
cv2.waitKey()
```

After the usual imports, we load the image we want to process. Then, we convert the image into grayscale. By now, you may have gathered that many methods in OpenCV expect a grayscale image as input. The next step is to create a SIFT detection object and compute the features and descriptors of the grayscale image:

```
sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(gray, None)
```

Behind the scenes, these simple lines of code carry out an elaborate process; we create a `cv2.SIFT` object, which uses DoG to detect keypoints and then computes a feature vector for the surrounding region of each keypoint. As the name of the `detectAndCompute` method clearly suggests, two main operations are performed: feature detection and the computation of descriptors. The return value of the operation is a tuple containing a list of keypoints and another list of the keypoints' descriptors.Finally, we process this image by drawing the keypoints on it with the `cv2.drawKeypoints` function and then displaying it with the usual `cv2.imshow` function. As one of its arguments, the `cv2.drawKeypoints` function accepts a flag that specifies the type of visualization we want. Here, we specify `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINT` in order to draw a visualization of the scale and orientation of each keypoint.

Anatomy of a keypoint

Each keypoint is an instance of the `cv2.KeyPoint` class, which has the following properties:

- The `pt` (point) property contains the *x* and *y* coordinates of the keypoint in the image.
- The `size` property indicates the diameter of the feature.
- The `angle` property indicates the orientation of the feature, as shown by the radial lines in the preceding processed image.
- The `response` property indicates the strength of the keypoint. Some features are classified by SIFT as stronger than others, and `response` is the property you would check to evaluate the strength of a feature.
- The `octave` property indicates the layer in the image pyramid where the feature was found. Let's briefly review the concept of an image pyramid, which we discussed previously in *Chapter 5*, *Detecting and Recognizing Faces*, in the *Conceptualizing Haar cascades* section. The SIFT algorithm operates in a similar fashion to face detection algorithms in that it processes the same image iteratively but alters the input at each iteration. In particular, the scale of the image is a parameter that changes at each iteration (`octave`) of the algorithm. Thus, the `octave` property is related to the image scale at which the keypoint was detected.
- Finally, the `class_id` property can be used to assign a custom identifier to a keypoint or a group of keypoints.

Detecting Fast Hessian features and extracting SURF descriptors

Computer vision is a relatively young branch of computer science, so many famous algorithms and techniques have only been invented recently. SIFT is, in fact, only 23 years old, having been published by David Lowe in 1999.SURF is a feature detection algorithm that was published in 2006 by Herbert Bay. SURF is several times faster than SIFT, and it is partially inspired by it.

> Note that SURF is a patented algorithm and, for this reason, is made available only in builds of `opencv_contrib` where the `OPENCV_ENABLE_NONFREE` CMake flag is used. SIFT was formerly a patented algorithm but its patent expired and now SIFT is available in standard builds of OpenCV.

It is not particularly relevant to this book to understand how SURF works under the hood, inasmuch as we can use it in our applications and make the best of it. What is important to understand is that `cv2.SURF` is an OpenCV class that performs keypoint detection with the Fast Hessian algorithm and descriptor extraction with SURF, much like the `cv2.SIFT` class performs keypoint detection with DoG and descriptor extraction with SIFT.Also, the good news is that OpenCV provides a standardized API for all its supported feature detection and descriptor extraction algorithms. Thus, with only trivial changes, we can adapt our previous code sample to use SURF instead of SIFT. Here is the modified code, with the changes `highlighted`:

```
import cv2
img = cv2.imread('../images/varese.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
surf = cv2.xfeatures2d.SURF_create(8000)
keypoints, descriptor = surf.detectAndCompute(gray, None)
cv2.drawKeypoints(img, keypoints, img, (51, 163, 236),
                  cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('surf_keypoints', img)
cv2.waitKey()
```

The parameter to `cv2.xfeatures2d.SURF_create` is a threshold for the Fast Hessian algorithm. By increasing the threshold, we can reduce the number of features that will be retained. With a threshold of `8000` , we get the following result:

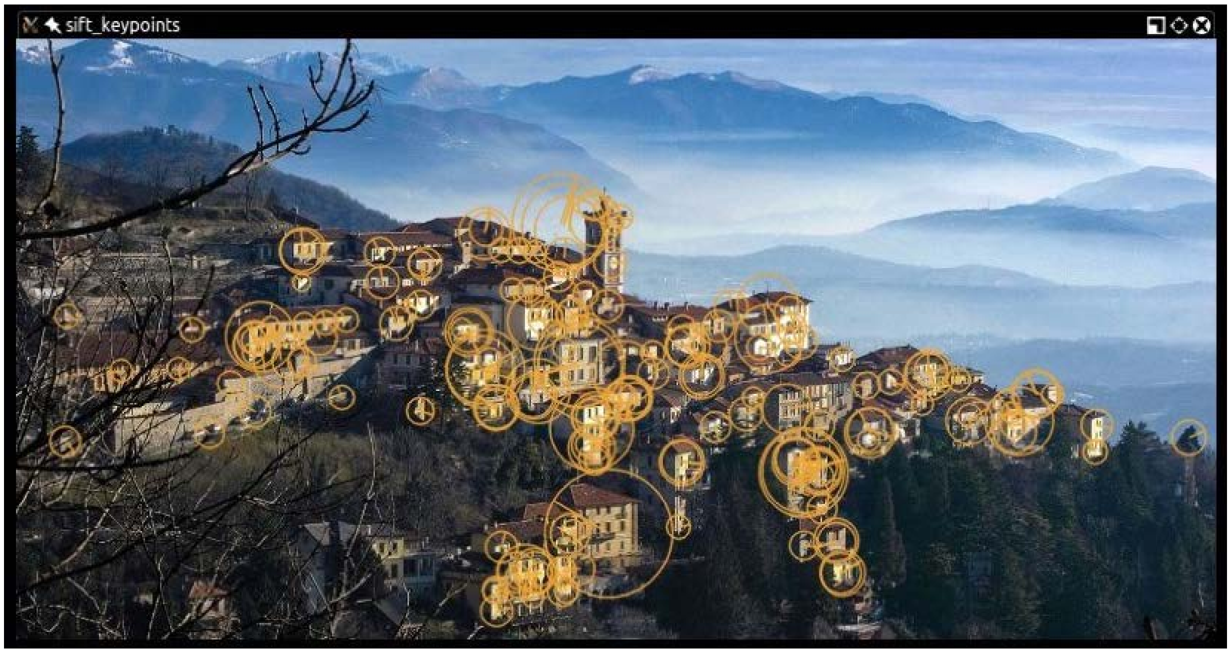


Figure 6.6: SURF descriptors for an image of Varese, Lombardy, Italy

Try adjusting the threshold to see how it affects the result. As an exercise, you may want to build a GUI application with a slider that controls the value of the threshold. This way, a user can adjust the threshold and see the number of features increase and decrease in an inversely proportional fashion. We built a GUI application with sliders in *Chapter 4*, *Depth Estimation and Segmentation*, in the *Depth estimation with a normal camera* section, so you may want to refer back to that section as a guide.Next, we'll examine the FAST corner detector, the BRIEF keypoint descriptor, and ORB (which uses FAST and BRIEF together).

Using ORB with FAST features and BRIEF descriptors

If SIFT is young, and SURF younger, ORB is in its infancy. ORB was first published in 2011 as a fast alternative to SIFT and SURF.The algorithm was published in the paper *ORB: an efficient alternative to SIFT or SURF*, available in PDF format at http://www.willowgarage.com/sites/default/files/orb_final.pdf.ORB mixes the techniques used in the FAST keypoint detector and the BRIEF keypoint descriptor, so it is worth taking a quick look at FAST and BRIEF first. Then, we will talk about brute-force matching – an algorithm used for feature matching – and look at an example of feature matching.

FAST

The **Features from Accelerated Segment Test** (**FAST**) algorithm works by analyzing circular neighborhoods of 16 pixels. It marks each pixel in a neighborhood as brighter or darker than a particular threshold, which is defined relative to the center of the circle. A neighborhood is deemed to be a corner if it contains a number of contiguous pixels marked as brighter or darker.FAST also uses a high-speed test, which can sometimes determine that a neighborhood is not a corner by checking just 2 or 4 pixels instead of 16. To understand how this test works, let's take a look at the following diagram, taken from the OpenCV documentation:

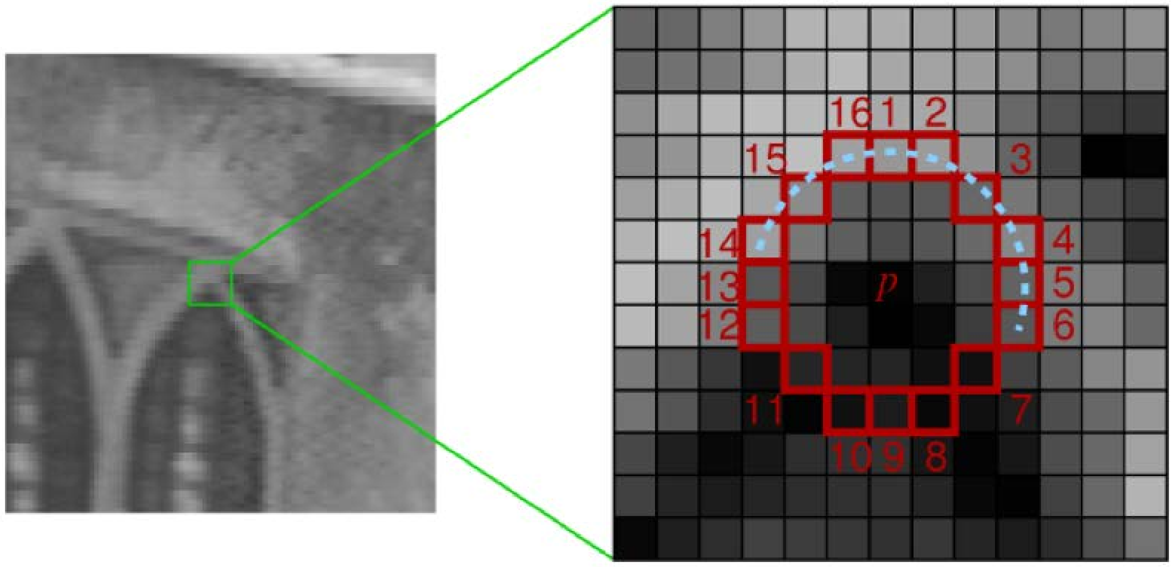


Figure 6.7: How the FAST algorithm analyzes a neighbourhood around a corner

Here, we can see a 16-pixel neighborhood at two different magnifications. The pixels at positions 1, 5, 9, and 13 correspond to the four cardinal points at the edge of the circular neighborhood. If the neighborhood is a corner, we expect that out of these four pixels, exactly three *or* exactly one will be brighter than the threshold. (Another way of saying this is that exactly one *or* exactly three of them will be darker than the threshold.) If exactly two of them are brighter than the threshold, then we have an edge, not a corner. If exactly four or exactly zero of them are brighter than the threshold, then we have a relatively uniform neighborhood that is neither a corner nor an edge.FAST is a clever algorithm, but it's not devoid of weaknesses, and to compensate for these weaknesses, developers analyzing images can implement a machine learning approach in order to feed a set of images (relevant to a given application) to the algorithm so that parameters such as the threshold are optimized. Whether the developer specifies parameters directly or provides a training set for a machine learning approach, FAST is an algorithm that is sensitive to the developer's input, perhaps more so than SIFT.

BRIEF

Binary Robust Independent Elementary Features (**BRIEF**), on the other hand, is not a feature detection algorithm, but a descriptor. Let's delve deeper into the concept of what a descriptor is, and then look at BRIEF.When we previously analyzed images with SIFT and SURF, the heart of the entire process was the call to the `detectAndCompute` function. This function performs two different steps – detection and computation – and they return two different results, coupled in a tuple.The result of detection is a set of keypoints; the result of the computation is a set of descriptors for those keypoints. This means that OpenCV's `cv2.SIFT` and `cv2.xfeatures2d.SURF` classes implement algorithms for both detection and description. Remember, though, that the original SIFT and SURF are *not* feature detection algorithms. OpenCV's `cv2.SIFT` implements DoG feature detection plus SIFT description, while OpenCV's `cv2.xfeatures2d.SURF` implements Fast Hessian feature detection plus SURF description.Keypoint descriptors are a representation of the image that serves as the gateway to feature matching because you can compare the keypoint descriptors of two images and find commonalities.BRIEF is one of the fastest descriptors currently available. The theory behind BRIEF is quite complicated, but suffice it to say that BRIEF adopts a series of optimizations that make it a very good choice for feature matching.

Brute-force matching

A brute-force matcher is a descriptor matcher that compares two sets of keypoint descriptors and generates a result that is a list of matches. It is called brute-force because little optimization is involved in the algorithm. For each keypoint descriptor in the first set, the matcher makes comparisons to every keypoint descriptor in the second set. Each comparison produces a distance value and the best match can be chosen on the basis of least distance.More generally, in computing, the term **brute-force** is associated with an approach that prioritizes the exhaustion of all possible combinations (for example, all the possible combinations of characters to crack a password of a known length). Conversely, an algorithm that prioritizes speed might skip some possibilities and try to take a shortcut to the solution that seems the most plausible.OpenCV provides a `cv2.BFMatcher` class that supports several approaches to brute-force feature matching.

Matching a logo in two images

Now that we have a general idea of what FAST and BRIEF are, we can understand why the team behind ORB (composed of Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski) chose these two algorithms as a foundation for ORB.In their paper, the authors set out to achieve the following results:

- The addition of a fast and accurate orientation component to FAST
- The efficient computation of oriented BRIEF features
- Analysis of variance and correlation of oriented BRIEF features
- A learning method to decorrelate BRIEF features under rotational invariance, leading to better performance in nearest-neighbor applications

The main points are quite clear: ORB aims to optimize and speed up operations, including the very important step of utilizing BRIEF in a rotation-aware fashion so that matching is improved, even in situations where a training image has a very different rotation to the query image.At this stage, though, perhaps you have had enough of the theory and want to sink your teeth into some feature matching, so let's look at some code. The following script attempts to match features in a logo to the features in a photograph that contains the logo:

```
import cv2
from matplotlib import pyplot as plt
# Load the images.
img0 = cv2.imread('../images/nasa_logo.png',
                  cv2.IMREAD_GRAYSCALE)
img1 = cv2.imread('../images/kennedy_space_center.jpg',
                  cv2.IMREAD_GRAYSCALE)
# Perform ORB feature detection and description.
orb = cv2.ORB_create()
kp0, des0 = orb.detectAndCompute(img0, None)
kp1, des1 = orb.detectAndCompute(img1, None)
# Perform brute-force matching.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des0, des1)
# Sort the matches by distance.
matches = sorted(matches, key=lambda x:x.distance)
# Draw the best 25 matches.
img_matches = cv2.drawMatches(
    img0, kp0, img1, kp1, matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
# Show the matches.
plt.imshow(img_matches)
plt.show()
```

Let's examine this code step by step. After the usual imports, we load two images (the query image and the scene) in grayscale format. Here is the query image, which is the NASA logo:

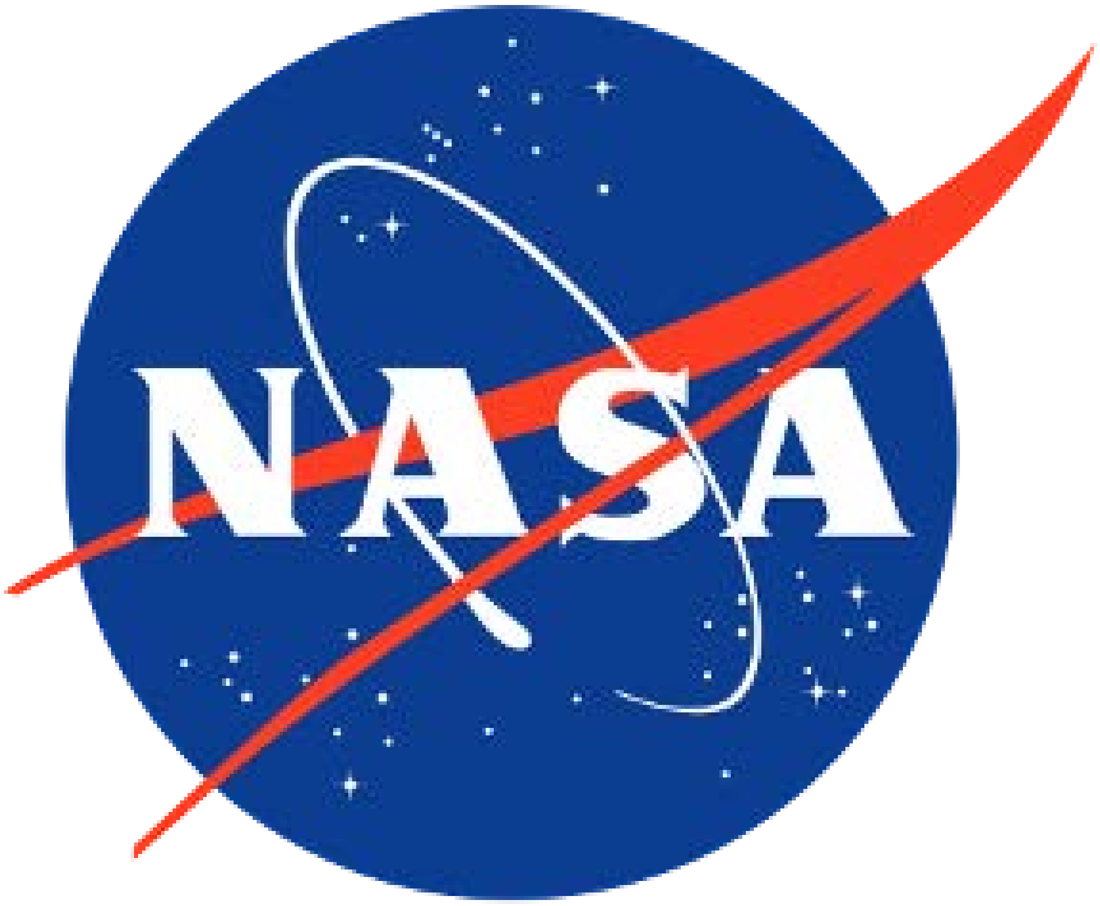


Figure 6.8: The NASA logo, to be used for keypoint descriptor matching

Here is the photo of the scene, which is the Kennedy Space Center:

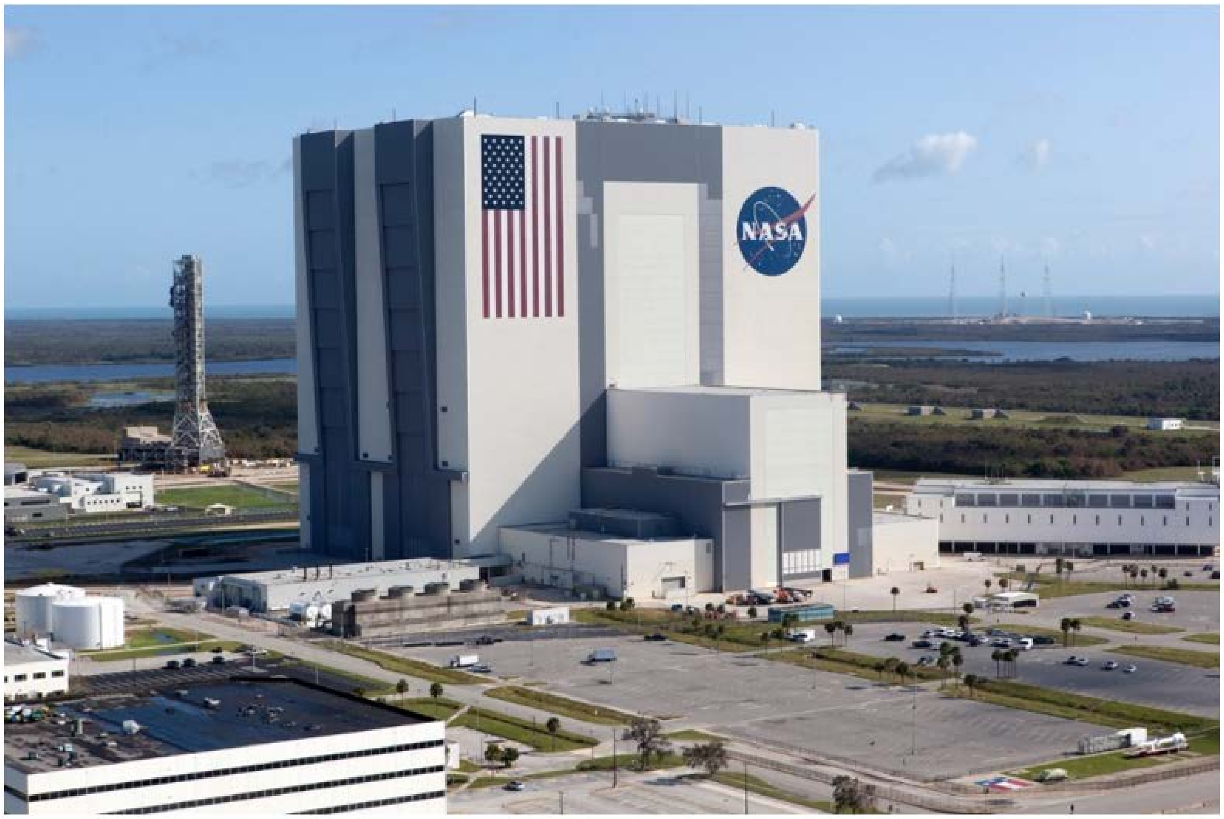


Figure 6.9: The Kennedy Space Center, with a NASA logo on its wall

Now, we proceed to create the ORB feature detector and descriptor:

```
# Perform ORB feature detection and description.
orb = cv2.ORB_create()
kp0, des0 = orb.detectAndCompute(img0, None)
kp1, des1 = orb.detectAndCompute(img1, None)
```

In a similar fashion to what we did with SIFT and SURF, we detect and compute the keypoints and descriptors for both images.From here, the concept is pretty simple: iterate through the descriptors and determine whether they are a match or not, and then calculate the quality of this match (distance) and sort the matches so that we can display the top *n* matches with a degree of confidence that they are, in fact, matching features on both images. `cv2.BFMatcher` does this for us:

```
# Perform brute-force matching.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des0, des1)
# Sort the matches by distance.
matches = sorted(matches, key=lambda x:x.distance)
```

At this stage, we already have all the information we need, but as computer vision enthusiasts, we place quite a bit of importance on visually representing data, so let's draw these matches in a `matplotlib` chart:

```
# Draw the best 25 matches.
img_matches = cv2.drawMatches(
    img0, kp0, img1, kp1, matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
# Show the matches.
plt.imshow(img_matches)
plt.show()
```

Python's slicing syntax is quite robust. If the `matches` list contains fewer than 25 entries, the `matches[:25]` slicing command will run without problems and give us a list with just as many elements as the original.

The result is as follows:

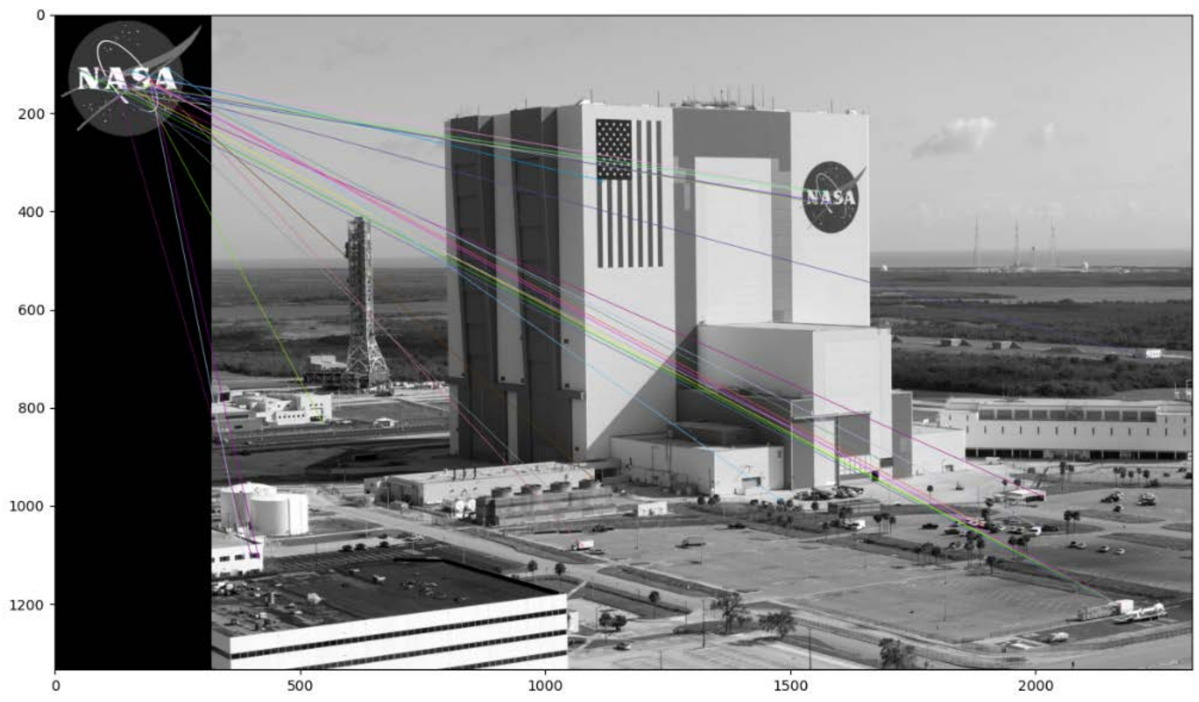


Figure 6.10: Matches between the NASA logo and the Kennedy Space Center, using brute-force matching

You might think that this is a disappointing result. Indeed, we can see that most of the matches are false matches. Unfortunately, this is quite typical. To improve our results, we need to apply additional techniques to filter out bad matches. We'll turn our attention to this task next.

Filtering matches using K-Nearest Neighbors and the ratio test

Imagine that a large group of renowned philosophers asks you to judge their debate on a question of great importance to life, the universe, and everything. You listen carefully as each philosopher speaks in turn. Finally, when all the philosophers have exhausted all their lines of argument, you review your notes and perceive two things, as follows:

- Every philosopher disagrees with every other
- No one philosopher is much more convincing than the others

From your first observation, you infer that *at most* one of the philosophers is right; however, it is possible that all the philosophers could be wrong. Then, from your second observation, you begin to fear that you are at risk of picking a philosopher who is wrong, *even if* one of the philosophers is correct. Any way you look at it, these people have made you late for dinner. You call it a tie and say that the debate's all-important question remains unresolved.We can compare our imaginary problem of judging the philosophers' debate to our practical problem of filtering out bad keypoint matches.First, let's assume that each keypoint in our query image has, at most, one correct match in the scene. By implication, if our query image is the NASA logo, we assume that the other image – the scene – contains, at most, one NASA logo. Given that a query keypoint has, at most, one correct or good match, when we consider all *possible* matches, we are primarily observing bad matches. Thus, a brute-force matcher, which computes a distance score for every possible match, can give us plenty of observations of the distance scores for bad matches. We expect that a good match will have a significantly better (lower) distance score than the numerous bad matches, so the scores for the bad matches can help us pick a threshold for a good match. Such a threshold does not necessarily generalize well across different query keypoints or different scenes, but at least it helps us on a case-by-case basis.Now, let's consider the implementation of a modified brute-force matching algorithm that adaptively chooses a distance threshold in the manner we have described. In the previous section's code sample, we used the `match` method of the `cv2.BFMatcher` class in order to get a list containing the single best (least-distance) match for each query keypoint. By doing so, we discarded information about the distance scores of all the worse possible matches – the kind of information we need for our adaptive approach. Fortunately, `cv2.BFMatcher` also provides a `knnMatch` method, which accepts an argument, `k`, that specifies the maximum number of best (least-distance) matches that we want to retain for each query keypoint. (In some cases, we may get fewer matches than the maximum.) **KNN** stands for **k-nearest neighbors**.We will use the `knnMatch` method to request a list of the two best matches for each query keypoint. Based on our assumption that each query keypoint has, at most, one correct match, we are confident that the second-best match is wrong. We multiply the second-best match's distance score by a value less than 1 in order to obtain the threshold.Then, we accept the best match as a good match only if its distant score is less than the threshold. This approach is known as the **ratio test**, and it was first proposed by David Lowe, the author of the SIFT algorithm. He describes the ratio test in his paper, *Distinctive Image Features from Scale-Invariant Keypoints*, which is available at https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf. Specifically, in the *Application to object recognition* section, he states the following:

> *"The probability that a match is correct can be determined by taking the ratio of the distance from the closest neighbor to the distance of the second closest."*

We can load the images, detect keypoints, and compute ORB descriptors in the same way as we did in the previous section's code sample. Then, we can perform brute-force KNN matching using the following two lines of code:

```
# Perform brute-force KNN matching.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
pairs_of_matches = bf.knnMatch(des0, des1, k=2)
```

`knnMatch` returns a list of lists; each inner list contains at least one match and no more than `k` matches, sorted from best (least distance) to worst. The following line of code sorts the outer list based on the distance score of the best matches:

```
# Sort the pairs of matches by distance.
pairs_of_matches = sorted(pairs_of_matches, key=lambda x:x[0].distance)
```

Let's draw the top 25 best matches, along with any second-best matches that `knnMatch` may have paired with them. We can't use the `cv2.drawMatches` function because it only accepts a one-dimensional list of matches; instead, we must use `cv2.drawMatchesKnn`. The following code is used to select, draw, and show the matches:

```
# Draw the 25 best pairs of matches.
img_pairs_of_matches = cv2.drawMatchesKnn(
    img0, kp0, img1, kp1, pairs_of_matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
# Show the pairs of matches.
plt.imshow(img_pairs_of_matches)
plt.show()
```

So far, we have not filtered out any bad matches – and, indeed, we have deliberately included the second-best matches, which we believe to be bad – so the result looks a mess. Here it is:

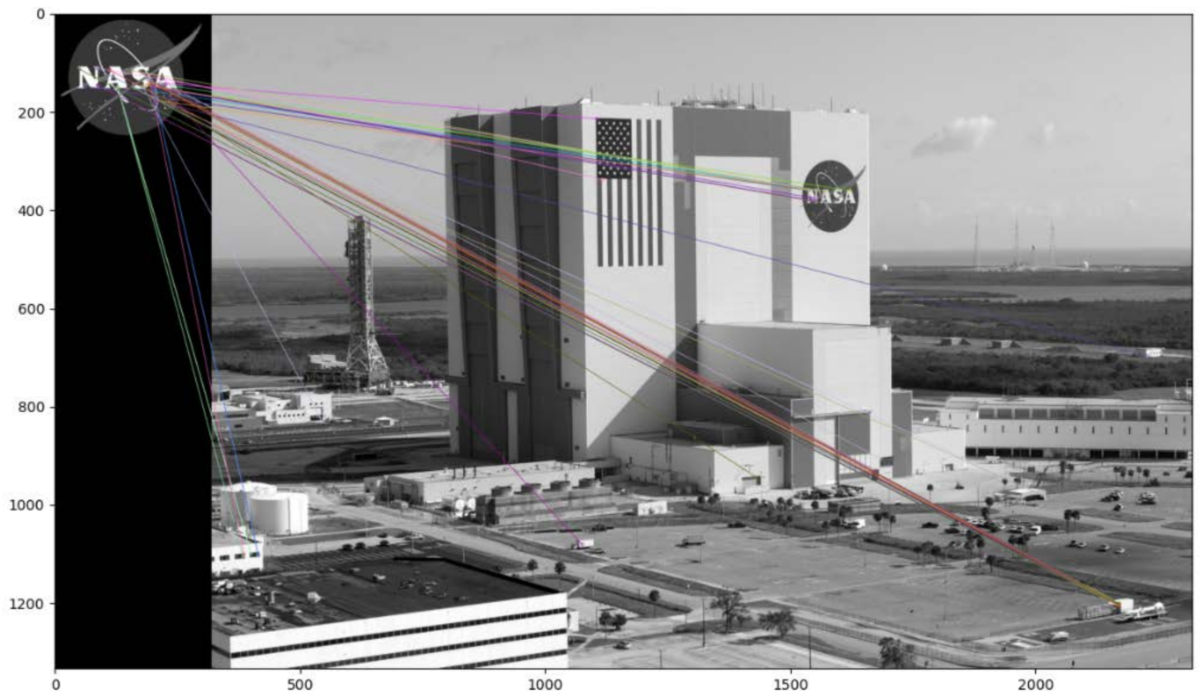


Figure 6.11: Matches between the NASA logo and the Kennedy Space Center, using brute-force KNN matching

Now, let's apply the ratio test. We will set the threshold at 0.8 times the distance score of the second-best match. If `knnMatch` has failed to provide a second-best match, we reject the best match anyway because we are unable to apply the test. The following code applies these conditions and provides us with a list of best matches that passed the test:

```
# Apply the ratio test.
matches = [x[0] for x in pairs_of_matches
           if len(x) > 1 and x[0].distance < 0.8 * x[1].distance]
```

Having applied the ratio test, now we are only dealing with best matches (not pairs of best and second-best matches), so we can draw them with `cv2.drawMatches` instead of `cv2.drawMatchesKnn`. Again, we will select the top 25 matches from the list. The following code is used to select, draw, and show the matches:

```
# Draw the best 25 matches.
img_matches = cv2.drawMatches(
    img0, kp0, img1, kp1, matches[:25], img1,
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
# Show the matches.
```

```
plt.imshow(img_matches)
plt.show()
```

Here, we can see the matches that passed the ratio test:

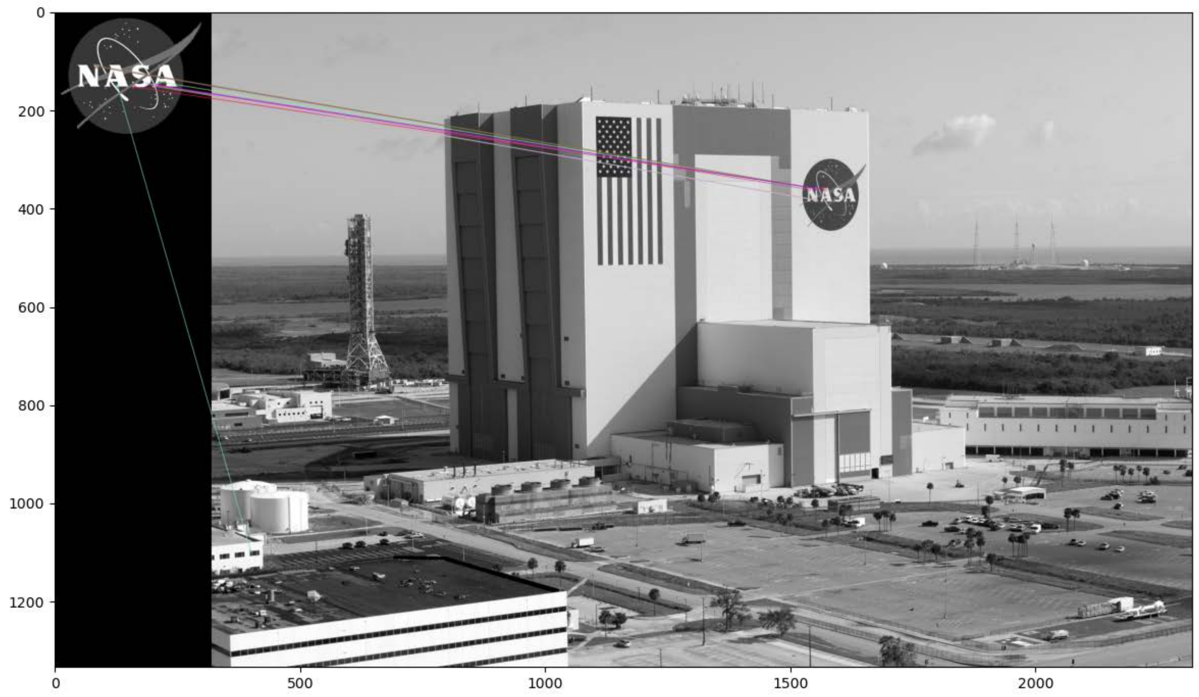


Figure 6.12: Matches between the NASA logo and the Kennedy Space Center, using brute-force KNN matching and the ratio test

Comparing this output image to the one in the previous section, we can see that KNN and the ratio test have allowed us to filter out many bad matches. The remaining matches are not perfect but nearly all of them point to the correct region – the NASA logo on the side of the Kennedy Space Center.We have made a promising start. Next, we will replace the brute-force matcher with a faster matcher called **FLANN**. After that, we will learn how to describe a set of matches in terms of homography – that is, a 2D transformation matrix that expresses the position, rotation, scale, and other geometric characteristics of the matched object.

Matching with FLANN

FLANN stands for **Fast Library for Approximate Nearest Neighbors**. It is an open source library under the permissive 2-clause BSD license and it is available on GitHub at https://github.com/flann-lib/flann. There, we find the following description:

"FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for the nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. FLANN is written in C++ and contains bindings for the following languages: C, MATLAB,Python, and Ruby."

In other words, FLANN has a big toolbox, it knows how to choose the right tools for the job, and it speaks several languages. These features make the library fast and convenient. Indeed, FLANN's authors claim that it is 10 times faster than other nearest-neighbor search software for many datasets.Although FLANN is also available as a standalone library, we will use it as part of OpenCV because OpenCV provides a handy wrapper for it.To begin our practical example of FLANN matching, let's import NumPy, OpenCV, and Matplotlib, and load two images from files. Here is the relevant code:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
img0 = cv2.imread('../images/gauguin_entre_les_lys.jpg',
                  cv2.IMREAD_GRAYSCALE)
img1 = cv2.imread('../images/gauguin_paintings.png',
                  cv2.IMREAD_GRAYSCALE)
```

Here is the first image – the query image – that our script is loading:

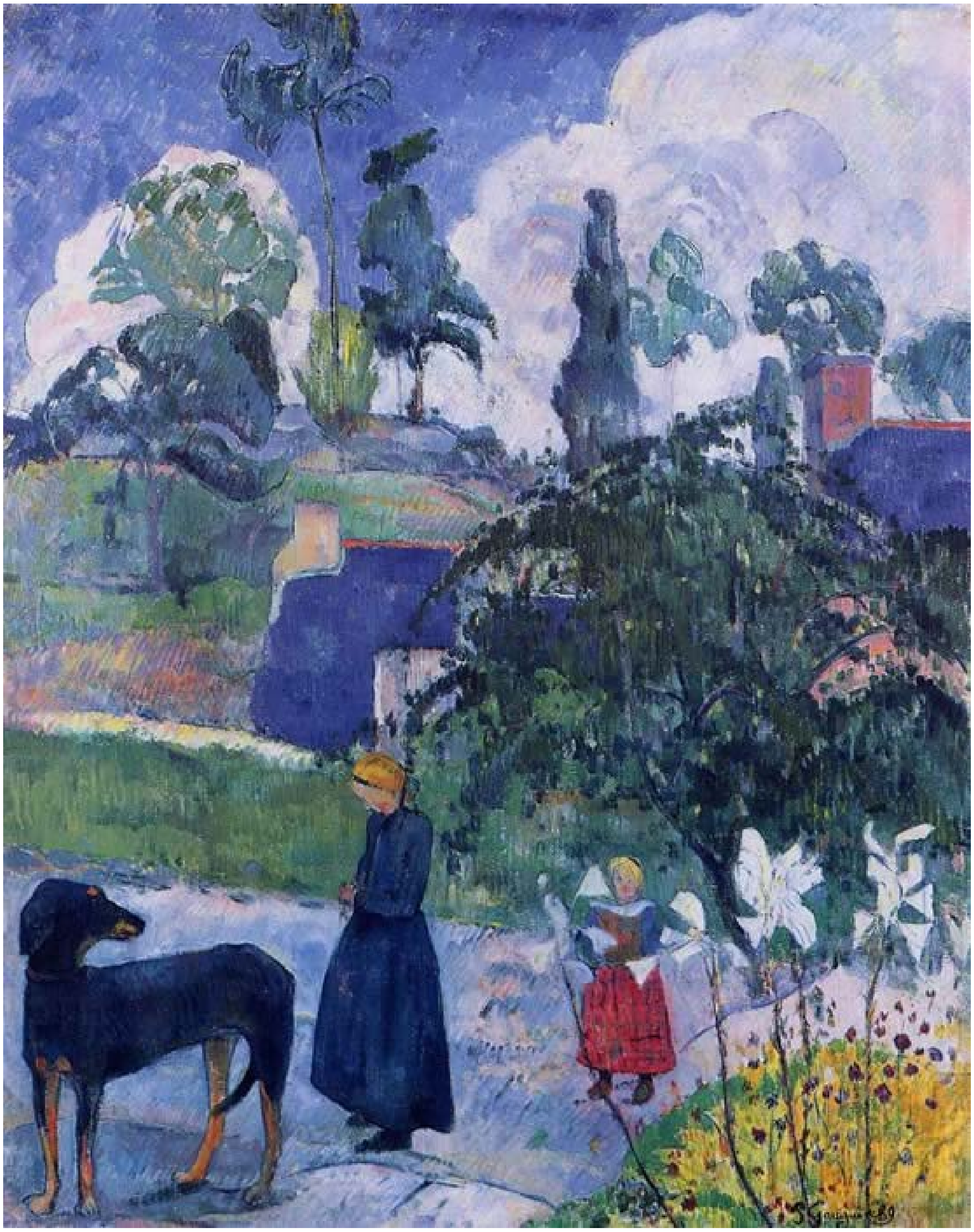


Figure 6.13: Paul Gaugin's painting Entre les lys (Among the lilies) , to be used for keypoint descriptor matching

This work of art is *Entre les lys* (*Among the lilies*), painted by Paul Gauguin in 1889. We will search for matching keypoints in a larger image that contains multiple works by Gauguin, alongside some haphazard shapes drawn by one of the authors of this book. Here is the larger image:

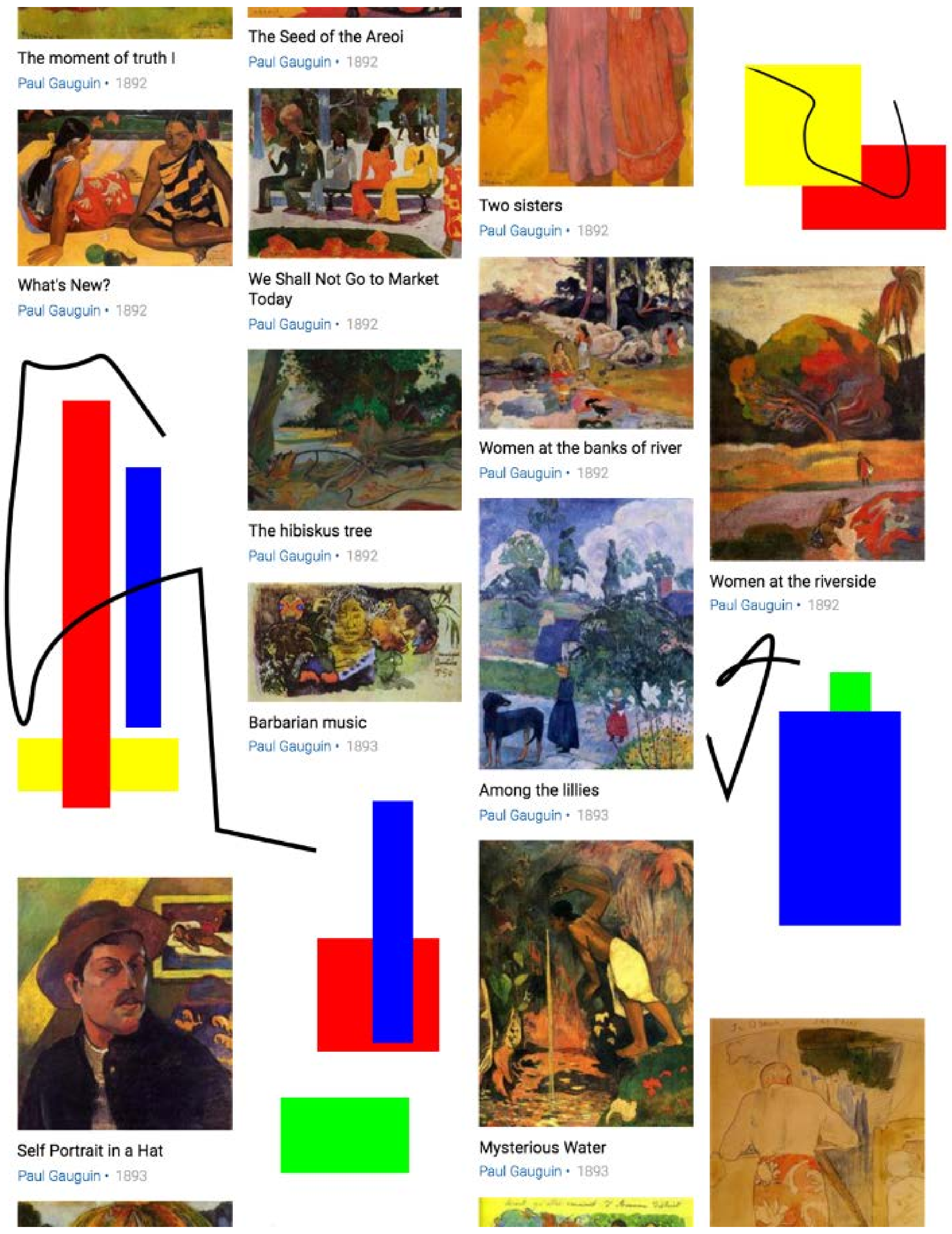


Figure 6.14: Haphazard shapes and various Gaugin paintings, to be used for keypoint descriptor matching

Within the larger image, *Entre les lys* appears in the third column, third row. The query image and the corresponding region of the larger image are not identical; they depict *Entre les lys* in slightly different colors and at a different scale. Nonetheless, this should be an easy case for our matcher.Let's detect the necessary keypoints and extract our features using the `cv2.SIFT` class:

```
# Perform SIFT feature detection and description.
sift = cv2.SIFT_create()
kp0, des0 = sift.detectAndCompute(img0, None)
kp1, des1 = sift.detectAndCompute(img1, None)
```

So far, the code should seem familiar, since we have already dedicated several sections of this chapter to SIFT and other descriptors. In our previous examples, we fed the descriptors to `cv2.BFMatcher` for brute-force matching. This time, we will use `cv2.FlannBasedMatcher` instead. The following code performs FLANN-based matching with custom parameters:

```
# Define FLANN-based matching parameters.
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
# Perform FLANN-based matching.
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des0, des1, k=2)
```

Here, we can see that the FLANN matcher takes two parameters: an `indexParams` object and a `searchParams` object. These parameters, passed in the form of dictionaries in Python (and structs in C++), determine the behavior of the index and search objects that are used internally by FLANN to compute the matches. We have chosen parameters that offer a reasonable balance between accuracy and processing speed. Specifically, we are using a **kernel density tree** (**kd-tree**) indexing algorithm with five trees, which FLANN can process in parallel. (The FLANN documentation recommends between one tree, which would offer no parallelism, and 16 trees, which would offer a high degree of parallelism if the system could exploit it.)We are performing 50 checks or traversals of each tree. A greater number of checks can provide greater accuracy but at a greater computational cost.After performing FLANN-based matching, we apply Lowe's ratio test with a multiplier of 0.7. To demonstrate a different coding style, we will use the result of the ratio test in a slightly different way compared to how we did in the previous section's code sample. Previously, we assembled a new list with just the good matches in it. This time, we will assemble a list called `mask_matches` , in which each element is a sublist of length `k` (the same `k` that we passed to `knnMatch`). If a match is good, we set the corresponding element of the sublist to `1`; otherwise, we set it to `0`.For example, if we have `mask_matches = [[0, 0], [1, 0]]`, this means that we have two matched keypoints; for the first keypoint, the best and second-best matches are both bad, while for the second keypoint, the best match is good but the second-best match is bad. Remember, we assume that all the second-best matches are bad. We use the following code to apply the ratio test and build the mask:

```
# Prepare an empty mask to draw good matches.
mask_matches = [[0, 0] for i in range(len(matches))]
# Populate the mask based on David G. Lowe's ratio test.
for i, (m, n) in enumerate(matches):
    if m.distance < 0.7 * n.distance:
        mask_matches[i]=[1, 0]
```

Now, it is time to draw and show the good matches. We can pass our `mask_matches` list to `cv2.drawMatchesKnn` as an optional argument, as `highlighted` in the following code segment:

```
# Draw the matches that passed the ratio test.
img_matches = cv2.drawMatchesKnn(
    img0, kp0, img1, kp1, matches, None,
    matchColor=(0, 255, 0), singlePointColor=(255, 0, 0),
    matchesMask=mask_matches, flags=0)
# Show the matches.
plt.imshow(img_matches)
plt.show()
```

`cv2.drawMatchesKnn` only draws the matches that we marked as good (with a value of `1`) in our mask. Let's unveil the result. Our script produces the following visualization of the FLANN-based matches:

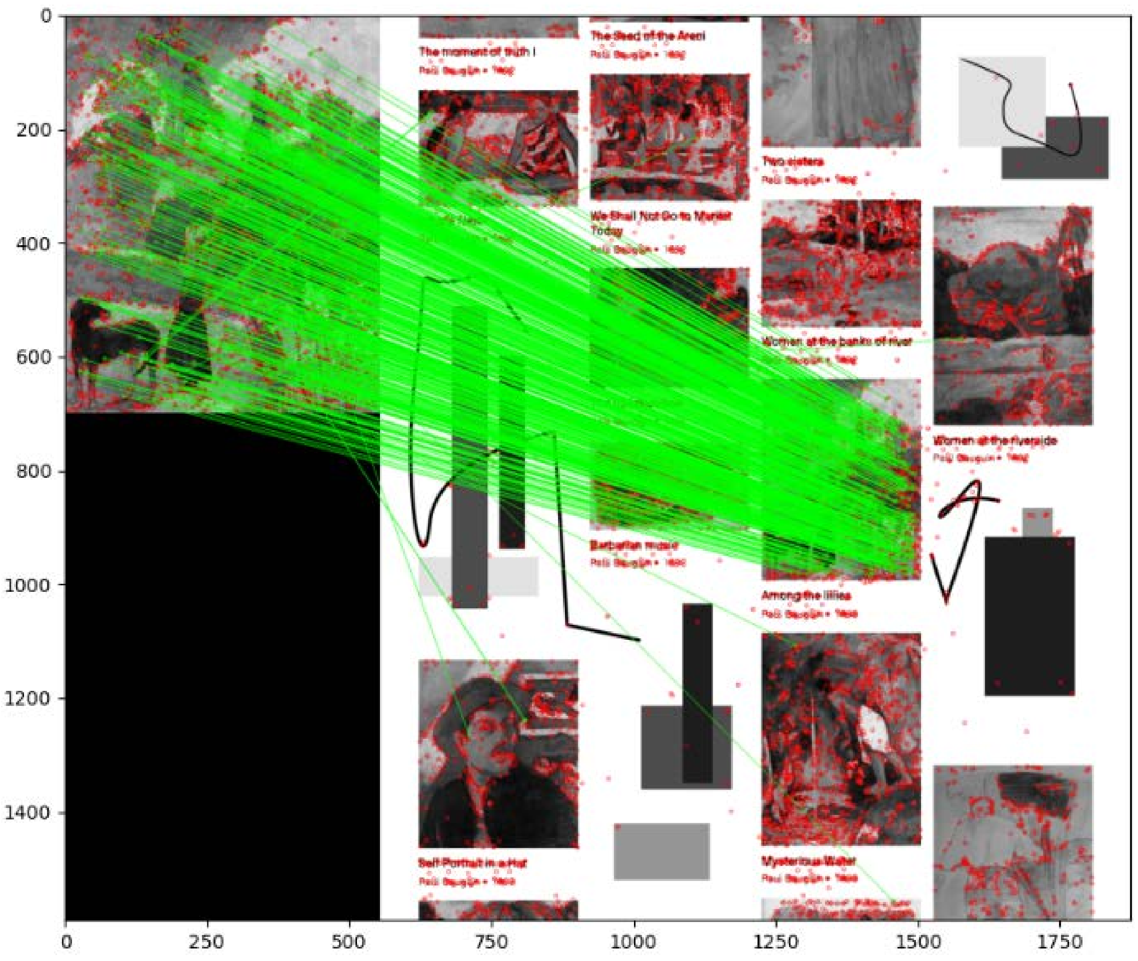


Figure 6.15: FLANN-based matching with Gaugin paintings

This is an encouraging picture: it appears that nearly all the matches fall in the right places. Next, let's try to reduce this type of result to a more succinct geometric representation – a homography – which would describe the pose of a whole matched object rather than a bunch of disconnected matched points.

Finding homography with FLANN-based matches

First of all, what is homography? Let's read a definition from the internet:

"A relation between two figures, such that to any point of the one corresponds one and but one point in the other, and vice versa. Thus, a tangent line rolling on a circle cuts two fixed tangents of the circle in two sets of points that are homographic."

If you – like the authors of this book – are none the wiser from the preceding definition, you will probably find the following explanation a bit clearer: homography is a condition in which two figures find each other when one is a perspective distortion of the other.First, let's take a look at what we want to achieve so that we can fully understand what homography is. Then, we will go through the code.Imagine that we want to search for the following tattoo:

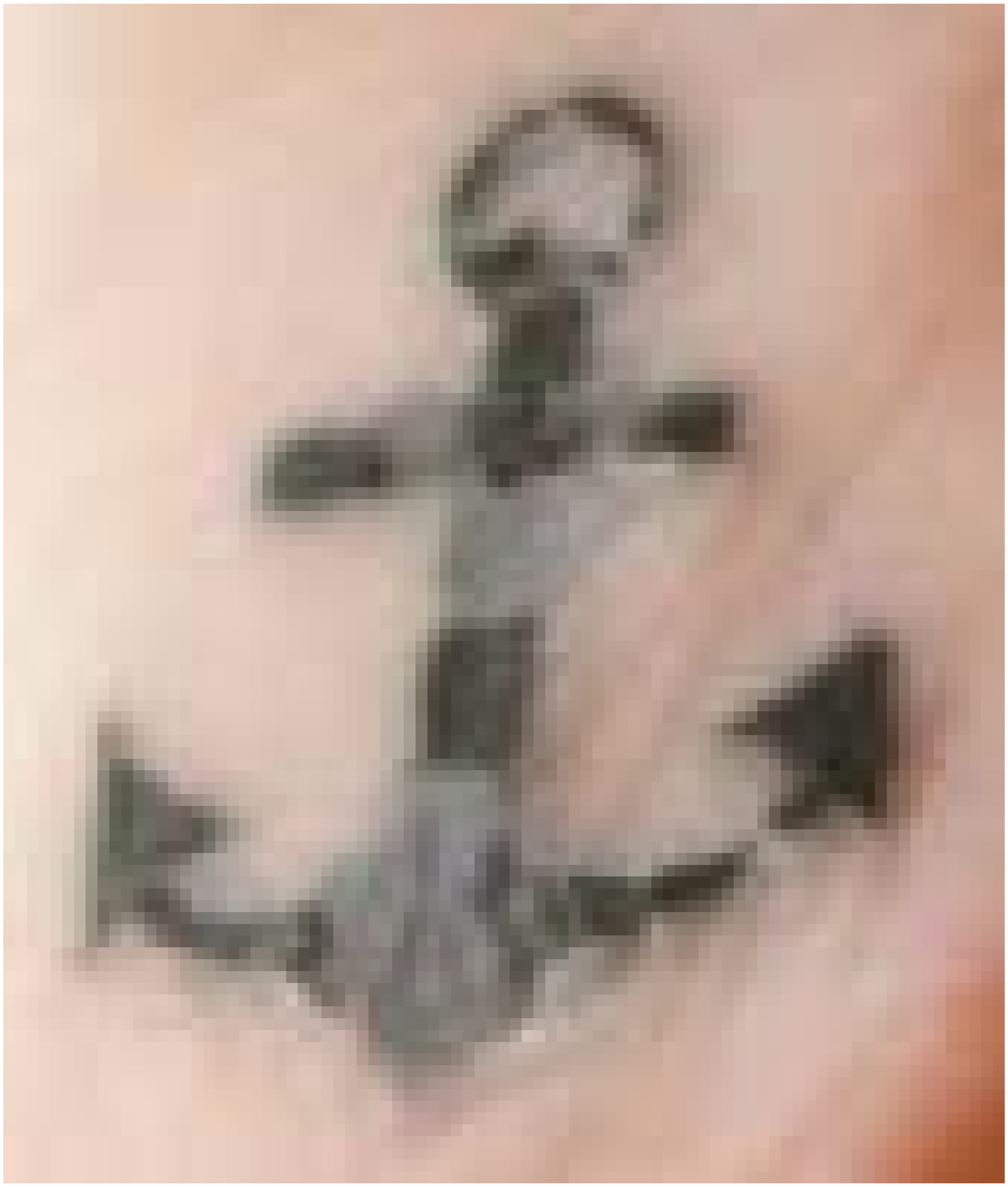


Figure 6.16: An anchor tattoo, to be used for keypoint matching and finding homography

We, as human beings, can easily locate the tattoo in the following image, despite there being a difference in rotation:

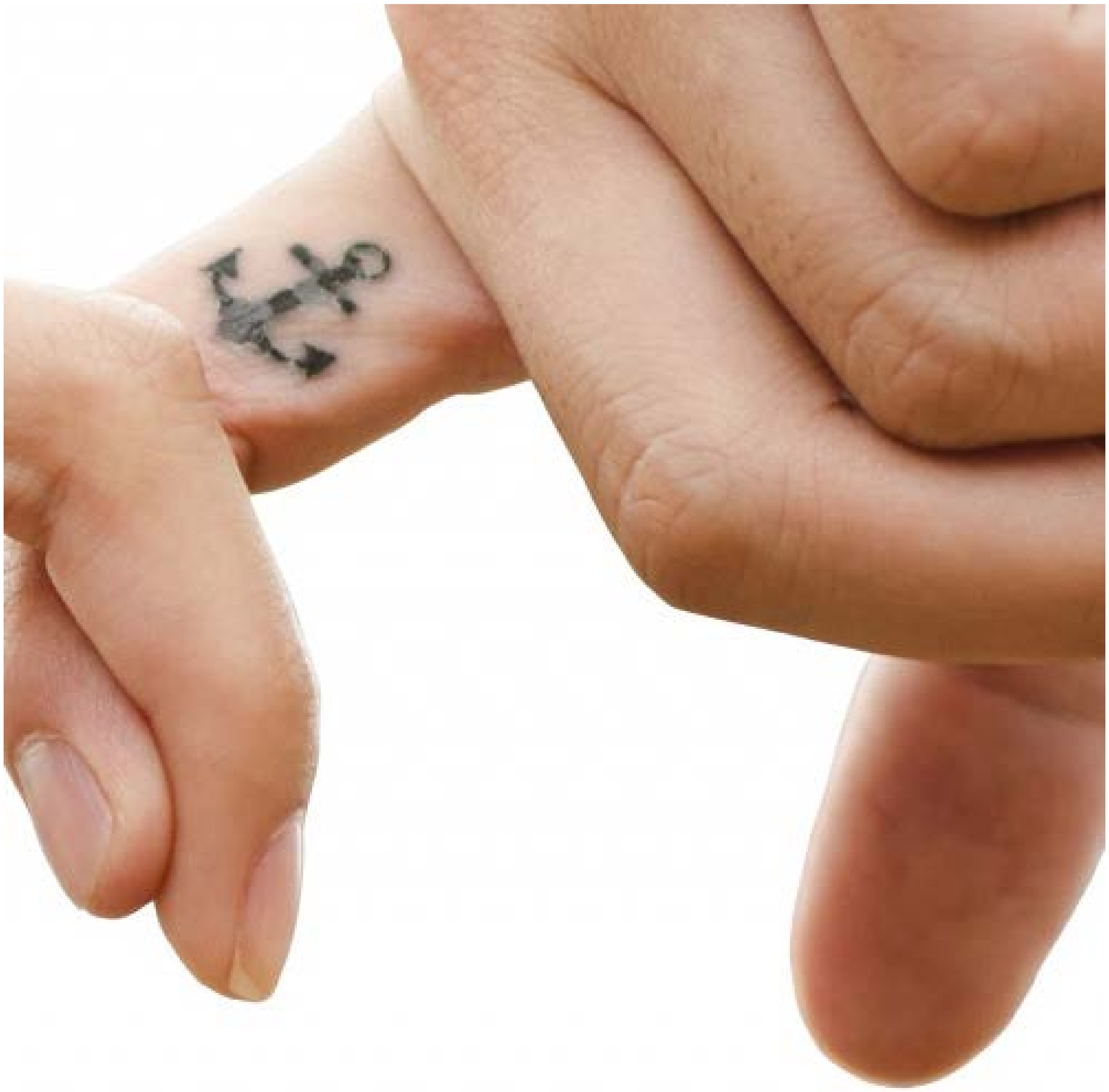


Figure 6.17: Fingers, with an anchor tattoo, to be used for keypoint matching and finding homography

As an exercise in computer vision, we want to write a script that produces the following visualization of keypoint matches and the homography:

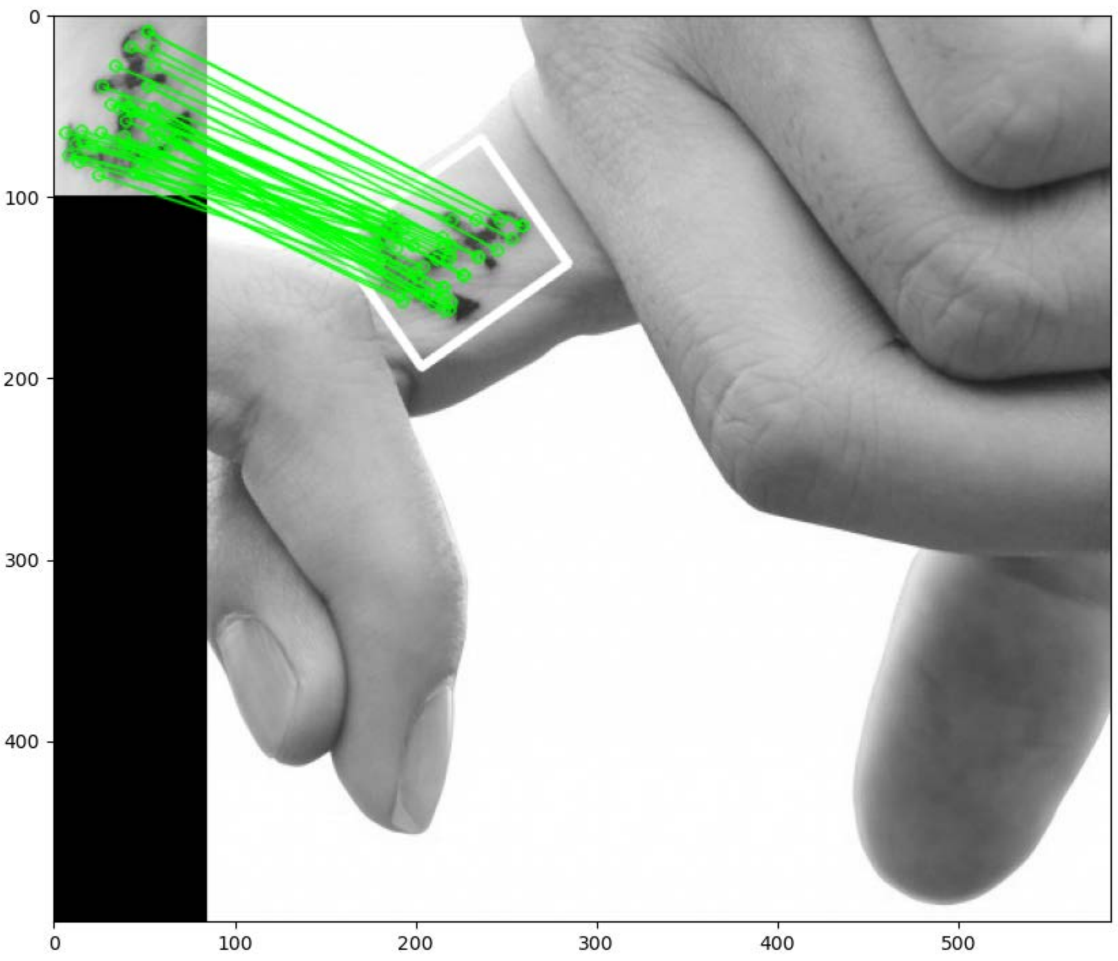


Figure 6.18: Keypoint matches and homography between two images of an anchor tattoo

As shown in the preceding screenshot, we took the subject in the first image, correctly identified it in the second image, drew matching lines between the keypoints, and even drew a white border showing the perspective deformation of the subject in the second image relative to the first image.You might have guessed – correctly – that the script's implementation starts by importing libraries, reading images in grayscale format, detecting features, and computing SIFT descriptors. We did all of this in our previous examples, so we will omit that here. Let's take a look at what we do next:

1. We proceed by assembling a list of matches that pass Lowe's ratio test, as shown in the following code:

```
# Find all the good matches as per Lowe's ratio test.
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)
```

1. Technically, we can calculate the homography with as few as four matches. However, if any of these four matches is flawed, it will throw off the accuracy of the result. A more practical minimum is `10`. Given the extra matches, the homography-finding algorithm can discard some outliers in order to produce a result that closely fits a substantial subset of the matches. Thus, we proceed to check whether we have at least `10` good matches:

```
MIN_NUM_GOOD_MATCHES = 10
if len(good_matches) >= MIN_NUM_GOOD_MATCHES:
```

1. If this condition has been satisfied, we look up the 2D coordinates of the matched keypoints and place these coordinates in two lists of floating-point coordinate pairs. One list contains the keypoint coordinates in the query image, while the other list contains the matching keypoint coordinates in the scene:

```
    src_pts = np.float32(
        [kp0[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
    dst_pts = np.float32(
        [kp1[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
```

1. Now, we find the homography:

```
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    mask_matches = mask.ravel().tolist()
```

Note that we create a `mask_matches` list, which will be used in the final drawing of the matches so that only points lying within the homography will have matching lines drawn.

1. At this stage, we have to perform a perspective transformation, which takes the rectangular corners of the query image and projects them into the scene so that we can draw the border:

```
    h, w = img0.shape
    src_corners = np.float32(
        [[0, 0], [0, h-1], [w-1, h-1], [w-1, 0]]).reshape(-1, 1, 2)
    dst_corners = cv2.perspectiveTransform(src_corners, M)
    dst_corners = dst_corners.astype(np.int32)
    # Draw the bounds of the matched region based on the homography.
    num_corners = len(dst_corners)
    for i in range(num_corners):
        x0, y0 = dst_corners[i][0]
        if i == num_corners - 1:
            next_i = 0
        else:
            next_i = i + 1
        x1, y1 = dst_corners[next_i][0]
        cv2.line(img1, (x0, y0), (x1, y1), 255, 3, cv2.LINE_AA)
```

Then, we proceed to draw the keypoints and show the visualization, as per our previous examples.

A sample application – tattoo forensics

Let's conclude this chapter with a real-life (or perhaps fantasy-life) example. Imagine you are working for the Gotham forensics department and you need to identify a tattoo. You have the original picture of a criminal's tattoo (perhaps captured in CCTV footage), but you don't know the identity of the person. However, you possess a database of tattoos, indexed with the name of the person that the tattoo belongs to.Let's divide this task into two parts:

- Build a database by saving image descriptors to files
- Load the database and scan for matches between a query image's descriptors and the descriptors in the database

We will cover these tasks in the next two subsections.

Saving image descriptors to file

The first thing we will do is save the image descriptors to an external file. This way, we don't have to recreate the descriptors every time we want to scan two images for matches.For the purposes of our example, let's scan a folder for images and create the corresponding descriptor files so that we have them readily available for future searches. To create descriptors, we will use a process we have already used a number of times in this chapter: namely, load an image, create a feature detector, detect features, and compute descriptors. To save the descriptors to a file, we will use a handy method of NumPy arrays called `save` , which dumps array data into a file in an optimized way.

The `pickle` module, in the Python standard library, provides more general-purpose serialization functionality that supports any Python object and not just

NumPy arrays. However, NumPy's array serialization is a good choice for numeric data.

Let's break our script up into functions. The main function will be named `create_descriptors` (plural, descriptors), and it will iterate over the files in a given folder. For each file, `create_descriptors` will call a helper function named `create_descriptor` (singular, descriptor), which will compute and save our descriptors for the given image file. Let's get started:

1. First, here is the implementation of `create_descriptors`:

```
import os
import numpy as np
import cv2
def create_descriptors(folder):
    feature_detector = cv2.SIFT_create()
    files = []
    for (dirpath, dirnames, filenames) in os.walk(folder):
        files.extend(filenames)
    for f in files:
        create_descriptor(folder, f, feature_detector)
```

Note that `create_descriptors` creates the feature detector because we only need to do this once, not every time we load a file. The helper function, `create_descriptor,` receives the feature detector as an argument.

1. Now, let's look at the latter function's implementation:

```
def create_descriptor(folder, image_path, feature_detector):
    if not image_path.endswith('png'):
        print('skipping %s' % image_path)
        return
    print('reading %s' % image_path)
    img = cv2.imread(os.path.join(folder, image_path),
                     cv2.IMREAD_GRAYSCALE)
    keypoints, descriptors = feature_detector.detectAndCompute(
        img, None)
    descriptor_file = image_path.replace('png', 'npy')
    np.save(os.path.join(folder, descriptor_file), descriptors)
```

Note that we save the descriptor files in the same folder as the images. Moreover, we assume that the image files have the `png` extension. To make the script more robust, you could modify it so that it supports additional image file extensions such as `jpg`. If a file has an unexpected extension, we skip it because it might be a descriptor file (from a previous run of the script) or some other non-image file.

1. We have finished implementing the functions. To complete the script, we will call `create_descriptors` with a folder name as an argument:

```
folder = '../images/tattoos'
create_descriptors(folder)
```

When we run this script, it produces the necessary descriptor files in NumPy's array file format, with the file extension `npy`. These files constitute our database of tattoo descriptors, indexed by name. (Each filename is a person's name.) Next, we'll write a separate script so that we can run a query against this database.

Scanning for matches

Now that we have descriptors saved to files, we just need to perform matching against each set of descriptors to determine which set best matches our query image.This is the process we will put in place:

1. Load a query image (`query.png`).
2. Scan the folder containing descriptor files. Print the names of the descriptor files.
3. Create SIFT descriptors for the query image.
4. For each descriptor file, load the SIFT descriptors and find FLANN-based matches. Filter the matches based on the ratio test. Print the person's name and the number of matches. If the number of matches exceeds an arbitrary threshold, print that the person is a suspect. (Remember, we are investigating a crime.)
5. Print the name of the prime suspect (the one with the most matches).

Let's consider the implementation:

1. First, the following code block loads the query image:

```
import os
import numpy as np
import cv2
# Read the query image.
folder = '../images/tattoos'
query = cv2.imread(os.path.join(folder, 'query.png'),
                   cv2.IMREAD_GRAYSCALE)
```

1. We proceed to assemble and print a list of the descriptor files:

```
# create files, images, descriptors globals
files = []
images = []
```

```
descriptors = []
for (dirpath, dirnames, filenames) in os.walk(folder):
    files.extend(filenames)
    for f in files:
        if f.endswith('npy') and f != 'query.npy':
            descriptors.append(f)
print(descriptors)
```

1. We set up our typical `cv2.SIFT` and `cv2.FlannBasedMatcher` objects, and we generate descriptors of the query image:

```
# Create the SIFT detector.
sift = cv2.SIFT_create()
# Perform SIFT feature detection and description on the
# query image.
query_kp, query_ds = sift.detectAndCompute(query, None)
# Define FLANN-based matching parameters.
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
# Create the FLANN matcher.
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

1. Now, we search for suspects, whom we define as people with at least 10 good matches for the query tattoo. Our search entails iterating over the descriptor files, loading the descriptors, performing FLANN-based matching, and filtering the matches based on the ratio test. We print a result for each person (each descriptor file):

```
# Define the minimum number of good matches for a suspect.
MIN_NUM_GOOD_MATCHES = 10
greatest_num_good_matches = 0
prime_suspect = None
print('>> Initiating picture scan...')
for d in descriptors:
    print('--------- analyzing %s for matches ------------' % d)
    matches = flann.knnMatch(
        query_ds, np.load(os.path.join(folder, d)), k=2)
    good_matches = []
    for m, n in matches:
        if m.distance < 0.7 * n.distance:
            good_matches.append(m)
    num_good_matches = len(good_matches)
    name = d.replace('.npy', '').upper()
    if num_good_matches >= MIN_NUM_GOOD_MATCHES:
        print('%s is a suspect! (%d matches)' % \
            (name, num_good_matches))
        if num_good_matches > greatest_num_good_matches:
            greatest_num_good_matches = num_good_matches
            prime_suspect = name
    else:
        print('%s is NOT a suspect. (%d matches)' % \
            (name, num_good_matches))
```

Note the use of the `np.load` method, which loads a specified `npy` file into a NumPy array.

1. In the end, we print the name of the prime suspect (if we found a suspect, that is):

```
if prime_suspect is not None:
    print('Prime suspect is %s.' % prime_suspect)
else:
    print('There is no suspect.')
```

Running the preceding script produces the following output:

```
>> Initiating picture scan...
--------- analyzing anchor-woman.npy for matches ------------
ANCHOR-WOMAN is NOT a suspect. (2 matches)
--------- analyzing anchor-man.npy for matches ------------
ANCHOR-MAN is a suspect! (44 matches)
--------- analyzing lady-featherly.npy for matches ------------
LADY-FEATHERLY is NOT a suspect. (2 matches)
--------- analyzing steel-arm.npy for matches ------------
STEEL-ARM is NOT a suspect. (0 matches)
--------- analyzing circus-woman.npy for matches ------------
CIRCUS-WOMAN is NOT a suspect. (1 matches)
Prime suspect is ANCHOR-MAN.
```

If we wanted, we could represent the matches and the homography graphically, as we did in the previous section.

Summary

In this chapter, we learned about detecting keypoints, computing keypoint descriptors, matching these descriptors, filtering out bad matches, and finding the homography between two sets of matching keypoints. We explored a number of algorithms that are available in OpenCV that can be used to accomplish these tasks, and we applied these algorithms to a variety of images and use cases.If we combine our new knowledge of keypoints with additional knowledge about cameras and perspective, we can track objects in 3D space. This will be the topic of *Chapter 9*, *Camera Models and Augmented Reality*. You can skip ahead to that chapter if you are particularly keen to reach the third dimension.If, instead, you think the next logical step is to round off your knowledge of two-dimensional solutions for object detection, recognition, and tracking, you can continue sequentially with *Chapter 7*, *Building Custom Object Detectors*, and then *Chapter 8*, *Tracking Objects*. It is good to know of a combination 2D and 3D techniques so that you can choose an approach that offers the right kind of output and the right computational speed for a given application.

7 Building Custom Object Detectors

This chapter delves deeper into the concept of object detection, which is one of the most common challenges in computer vision. Having come this far in the book, you are perhaps wondering when you will be able to put computer vision into practice on the streets. Do you dream of building a system to detect cars and people? Well, you are not too far from your goal, actually.We have already looked at some specific cases of object detection and recognition in previous chapters. We focused on upright, frontal human faces in *Chapter 5*, *Detecting and Recognizing Faces*, and on objects with corner-like or blob-like features in *Chapter 6*, *Retrieving Images and Searching Using Image Descriptors*. Now, in the current chapter, we will explore algorithms that have a good ability to generalize or extrapolate, in the sense that they can cope with the real-world diversity that exists within a given class of object. For example, different cars have different designs, and people can appear to be different shapes depending on the clothes they wear.Specifically, we will pursue the following objectives:

- Learning about another kind of feature descriptor: the **histogram of oriented gradients** (**HOG**) descriptor.
- Understanding **non-maximum suppression**, also called **non-maxima suppression** (**NMS**), which helps us choose the best of an overlapping set of detection windows.
- Gaining a high-level understanding of **support vector machines** (**SVMs**). These general-purpose classifiers are based on supervised machine learning, in a way that is similar to linear regression.
- Detecting people with a pre-trained classifier based on HOG descriptors.
- Training a **bag-of-words** (**BoW**) classifier to detect a car. For this sample, we will work with a custom implementation of an image pyramid, a sliding window, and NMS so that we can better understand the inner workings of these techniques.

Most of the techniques in this chapter are not mutually exclusive; rather, they work together as components of a detector. By the end of the chapter, you will know how to train and use classifiers that have practical applications on the streets!

Technical requirements

This chapter uses Python, OpenCV, and NumPy. Please refer back to *Chapter 1*, *Setting Up OpenCV*, for installation instructions.The completed code for this chapter can be found in this book's GitHub repository, at
https://github.com/PacktPublishing/Learning-OpenCV-5-Computer-Vision-with-

Python-Fourth-Edition, in the `chapter07` folder. Sample images can be found in the repository in the `images` folder.

Understanding HOG descriptors

HOG is a feature descriptor, so it belongs to the same family of algorithms as **scale-invariant feature transform** (**SIFT**), **speeded-up robust features** (**SURF**), and **Oriented FAST and rotated BRIEF** (**ORB**), which we covered in *Chapter 6*, *Retrieving Images and Searching Using Image Descriptors*. Like other feature descriptors, HOG is capable of delivering the type of information that is vital for feature matching, as well as for object detection and recognition. Most commonly, HOG is used for object detection. The algorithm – and, in particular, its use as a people detector – was popularized by Navneet Dalal and Bill Triggs in their paper *Histograms of Oriented Gradients for Human Detection* (INRIA, 2005), which is available online at https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf.HOG's internal mechanism is really clever; an image is divided into cells and a set of gradients is calculated for each cell. Each gradient describes the change in pixel intensities in a given direction. Together, these gradients form a histogram representation of the cell. We encountered a similar approach when we studied face recognition with the **local binary pattern histogram** (**LBPH**) in *Chapter 6*, *Detecting and Recognizing Faces*.Before diving into the technical details of how HOG works, let's first take a look at how HOG *sees* the world.

Visualizing HOG

Carl Vondrick, Aditya Khosla, Hamed Pirsiavash, Tomasz Malisiewicz, and Antonio Torralba have developed a HOG visualization technique called HOGgles (HOG goggles). For a summary of HOGgles, as well as links to code and publications, refer to Carl Vondrick's MIT web page at http://www.cs.columbia.edu/~vondrick/ihog/index.html. As one of their test images, Vondrick et al. use the following photograph of a truck:

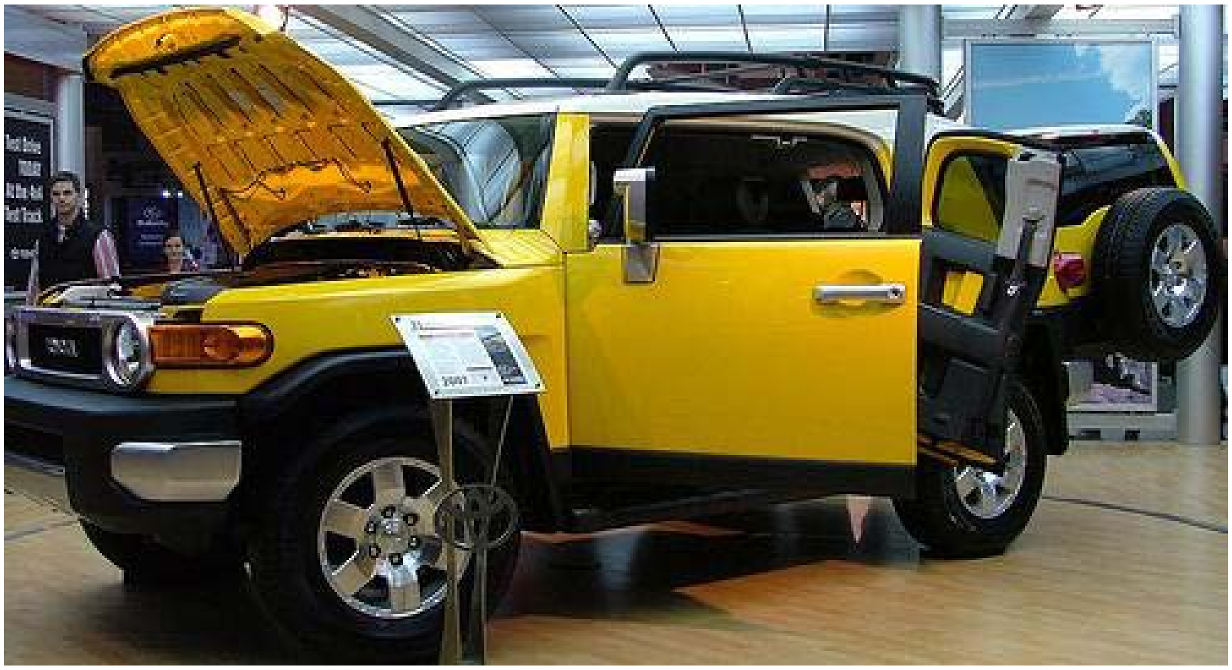


Figure 7.1: A truck, to be used as a test image for HOG visualization

Vondrick et al. produce the following visualization of the HOG descriptors, based on an approach from Dalal and Triggs' earlier paper:

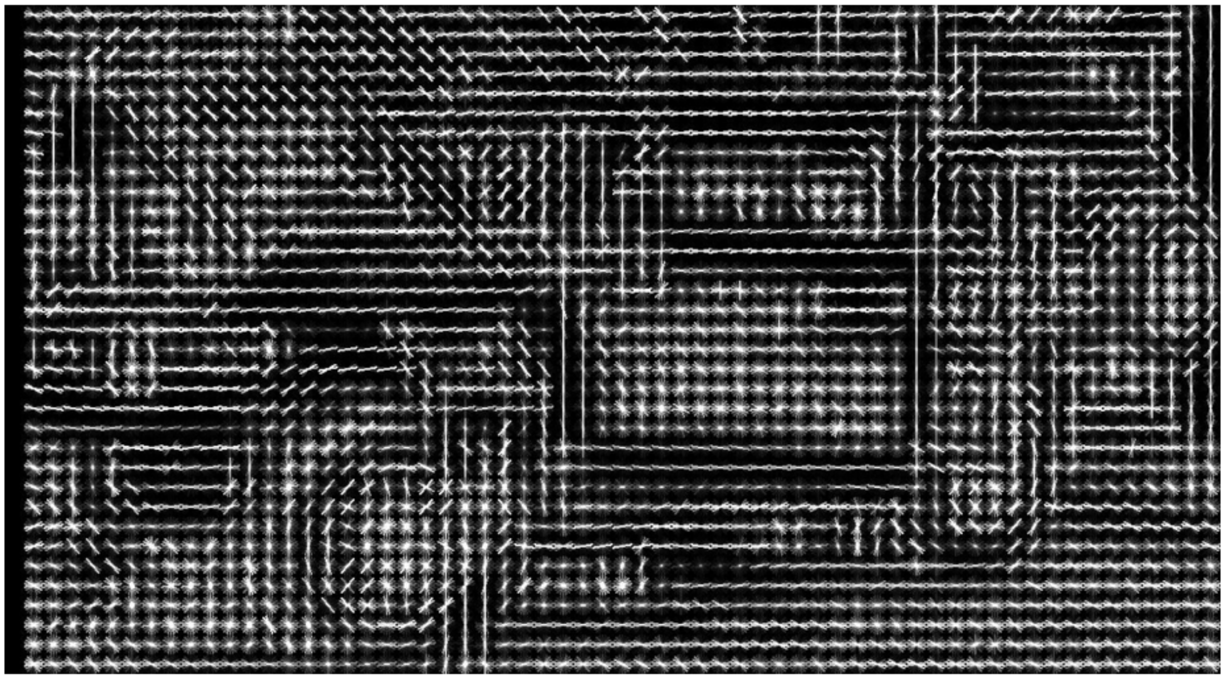


Figure 7.2: HOG cell gradients, visualized as lines

Then, applying HOGgles, Vondrick et al. invert the feature description algorithm to reconstruct the image of the truck as HOG sees it, as shown here:

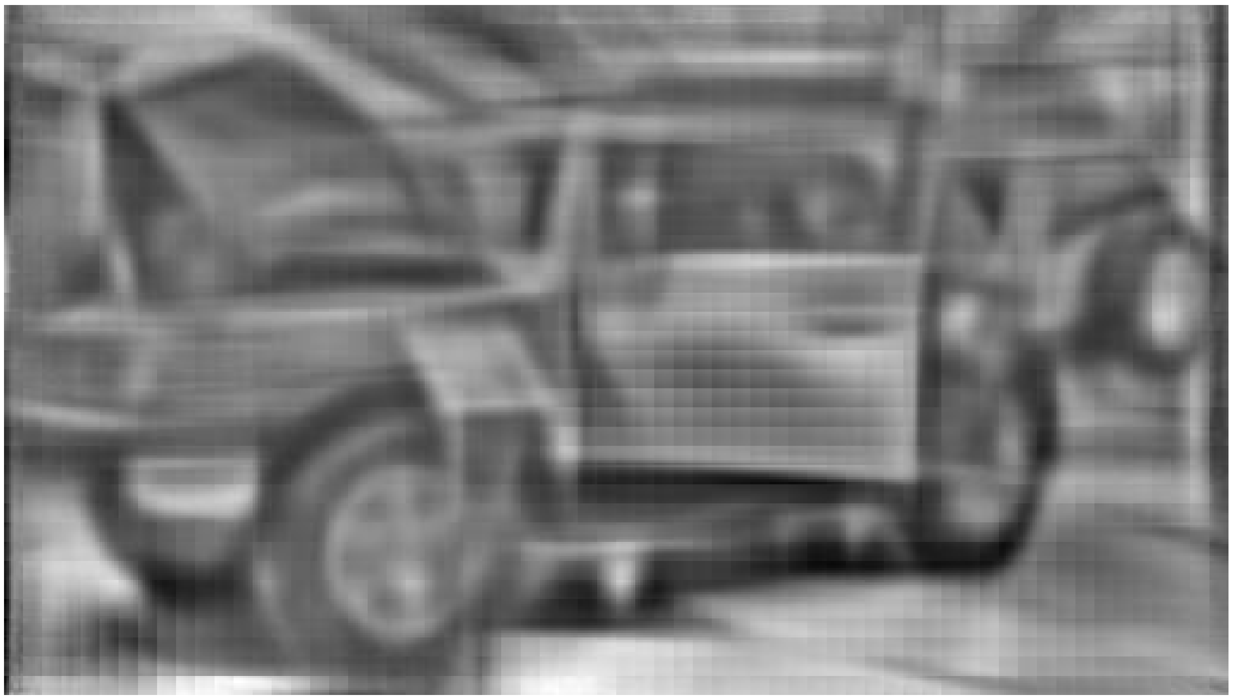


Figure 7.3: HOG cell gradients, visualized as smooth grayscale transitions

In both of these two visualizations, you can see that HOG has divided the image into cells, and you can easily recognize the wheels and the main structure of the vehicle. In the first visualization, the calculated gradients for each cell are visualized as a set of crisscrossing lines that sometimes look like an elongated star; the star's longer axes represent stronger gradients. In the second visualization, the gradients are visualized as a smooth transition in brightness, along various axes in a cell.Now, let's give further consideration to the way HOG works, and the way it can contribute to an object detection solution.

Using HOG to describe regions of an image

For each HOG cell, the histogram contains a number of bins equal to the number of gradients or, in other words, the number of axis directions that HOG considers. After calculating all the cells' histograms, HOG processes groups of histograms to produce higher-level descriptors. Specifically, the cells are grouped into larger regions, called blocks. These blocks can be made of any number of cells, but Dalal and Triggs found that 2x2 cell blocks yielded the best results when performing people detection. A block-wide vector is created so that it can be normalized, compensating for local variations in illumination and shadowing. (A single cell is too small a region to detect such variations.) This normalization improves a HOG-based detector's robustness, with respect to variations in lighting conditions.Like other detectors, a HOG-based detector needs to cope with variations in objects' location and scale. The need to search in various locations is addressed by moving a fixed-size sliding window across an image. The need to search at various scales is addressed by scaling the image to various sizes, forming a so-called image pyramid. We studied these techniques previously in *Chapter 5*, *Detecting and Recognizing Faces*, specifically in the *Conceptualizing Haar cascades* section. However, let's elaborate on one difficulty: how to handle multiple detections in overlapping windows.Suppose we are using a sliding window to perform people detection on an image. We slide our window in small steps, just a few pixels at a time, so we expect that it will frame any given person multiple times. Assuming that overlapping detections are indeed one person, we do not want to report multiple locations but, rather, only one location that we believe to be correct. In other words, even if a detection at a given location has a *good* confidence score, we might reject it if an overlapping detection has a *better* confidence score; thus, from a set of overlapping detections, we would choose the one with the *best* confidence score.This is where NMS comes into play. Given a set of overlapping regions, we can suppress (or reject) all the regions for which our classifier did not produce a maximal score.

Understanding NMS

The concept of NMS might sound simple. From a set of overlapping solutions, just pick the best one! However, the implementation is more complex than you might initially think. Remember the image pyramid? Overlapping detections can occur at different scales. We must gather up all our positive detections, and convert their bounds back to a common scale before we check for overlap. A typical implementation of NMS takes the following approach:

1. Construct an image pyramid.
2. Scan each level of the pyramid with the sliding window approach, for object detection. For each window that yields a positive detection (beyond a certain arbitrary confidence threshold), convert the window back to the original image's scale. Add the window and its confidence score to a list of positive detections.
3. Sort the list of positive detections by order of descending confidence score so that the best detections come first in the list.
4. For each window, *W*, in the list of positive detections, remove all subsequent windows that significantly overlap with *W*. We are left with a list of positive detections that satisfy the criterion of NMS.

Besides NMS, another way to filter the positive detections is to eliminate any subwindows. When we speak of a **subwindow** (or subregion), we mean a window (or region in an image) that is entirely contained inside another window (or region). To check for subwindows, we simply need to compare the corner coordinates of various window rectangles. We will take this simple approach in our first practical example, in the *Detecting people with HOG descriptors* section. Optionally, NMS and suppression of subwindows can be combined.

Several of these steps are iterative, so we have an interesting optimization problem on our hands. A fast sample implementation in MATLAB is provided by Tomasz Malisiewicz at http://www.computervisionblog.com/2011/08/blazing-fast-nmsm-from-exemplar-svm.html. A port of this sample implementation to Python is provided by Adrian Rosebrock at https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/. We will build atop the latter sample later in this chapter, in the *Detecting a car in a scene* section.Now, how do we determine the confidence score for a window? We need a classification system that determines whether a certain feature is present or not, and a confidence score for this classification. This is where **SVMs** come into play.

Understanding SVMs

Without going into details of how an SVM works, let's just try to grasp what it can help us accomplish in the context of machine learning and computer vision. Given labeled training data, an SVM learns to classify the same kind of data by finding an optimal *hyperplane*, which, in plain English, is the plane that divides differently labeled data by the largest possible margin. To aid our understanding, let's consider the following diagram, which is provided by Zach Weinberg under the Creative Commons Attribution-Share Alike 3.0 Unported License:

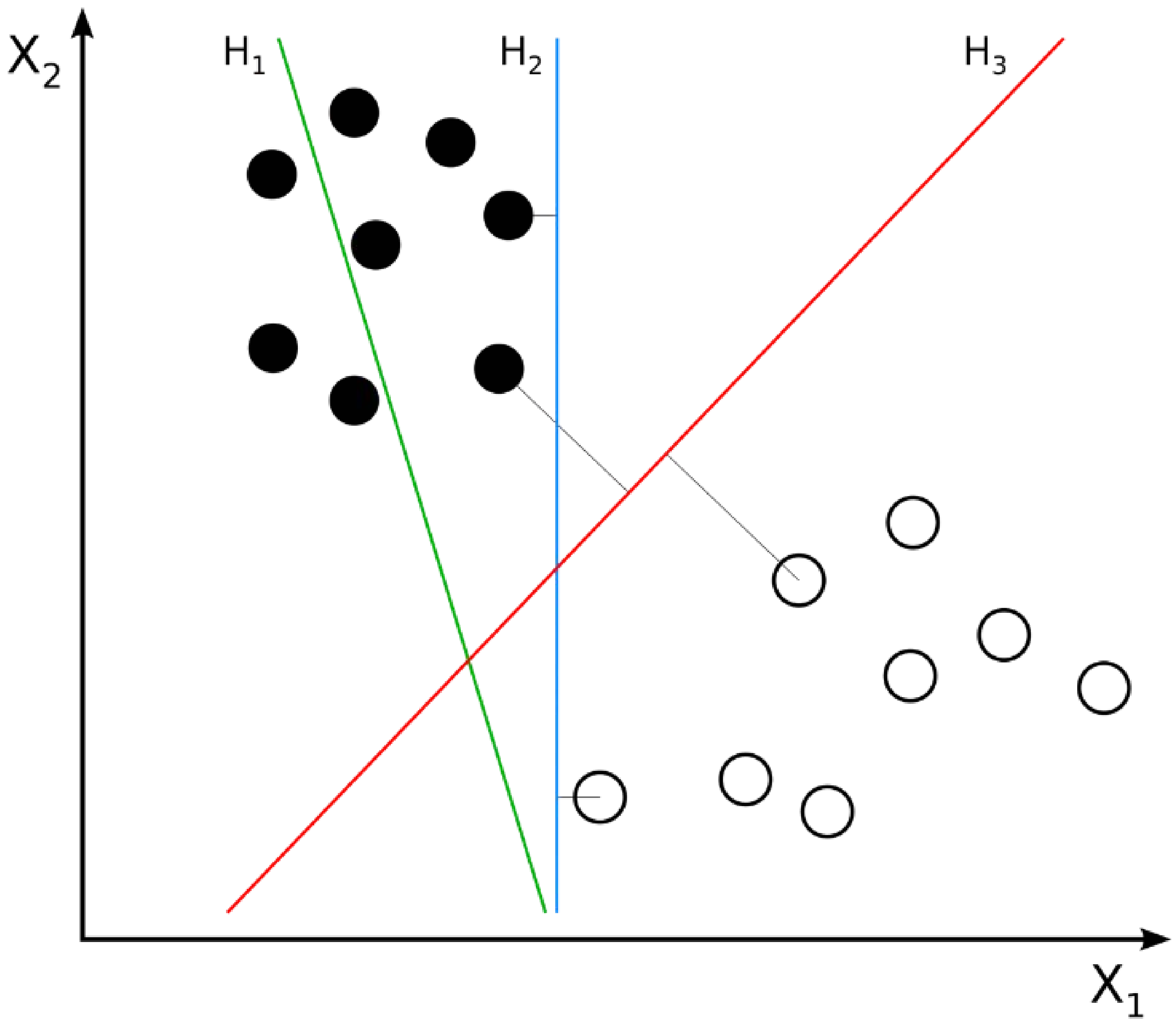


*Figure 7.4: Clusters and hyperplanes. An SVM would identify **H3** as the optimal hyperplane.*

Hyperplane **H1** (shown as a green line) does not divide the two classes (the black dots versus the white dots). Hyperplanes **H2** (shown as a blue line) and **H3** (shown as a red line) both divide the classes; however, only hyperplane **H3** divides the classes by a maximal margin.Let's suppose we are training an SVM as a people detector. We have two classes, *person* and *non-person*. As training samples, we provide vectors of HOG descriptors of various windows that do or do not contain a person. These windows may come from various images. The SVM learns by finding the optimal hyperplane that maximally divides the multidimensional HOG descriptor space into people (on one side of the hyperplane) and non-people (on the other side). Thereafter, when we give the trained SVM a vector of HOG descriptors for any other window in any image, the SVM can judge whether the window contains a person or not. The SVM can even give us a confidence value that relates to the vector's distance from the optimal hyperplane.The SVM model has been around since the early 1960s. However, it has undergone improvements since then, and the basis of modern SVM implementations can be found in the paper *Support-vector networks* (*Machine Learning*, 1995) by Corinna Cortes and Vladimir Vapnik. It is available at http://link.springer.com/article/10.1007/BF00994018.Now that we have a conceptual understanding of the key components we can combine to make an object detector, we can start looking at a few examples. We will start with one of OpenCV's ready-made object detectors, and then we will progress to designing and training our own custom object detectors.

Detecting people with HOG descriptors

OpenCV comes with a class called `cv2.HOGDescriptor`, which is capable of performing people detection. The interface has some similarities to the `cv2.CascadeClassifier` class that we used in *Chapter 5*, *Detecting and Recognizing Faces*. However, unlike `cv2.CascadeClassifier`, `cv2.HOGDescriptor` sometimes returns nested detection rectangles. In other words, `cv2.HOGDescriptor` might tell us that it detected one person whose bounding rectangle is located completely inside another person's bounding rectangle. This situation really is possible; for example, a child could be standing in front of an adult, and the child's bounding rectangle could be completely inside the adult's bounding rectangle. However, in a typical situation, nested detections are probably errors, so `cv2.HOGDescriptor` is often used along with code to filter out any nested detections.Let's begin our sample script by implementing a test to determine whether one rectangle is nested inside another. For this purpose, we will write a function, `is_inside(i, o)`, where `i` is the possible inner rectangle and `o` is the possible outer rectangle. The function will return `True` if `i` is inside `o`; otherwise, it will return `False`. Here is the start of the script:

```
import cv2
def is_inside(i, o):
    ix, iy, iw, ih = i
    ox, oy, ow, oh = o
    return ix > ox and ix + iw < ox + ow and \
        iy > oy and iy + ih < oy + oh
```

Now, we create an instance of `cv2.HOGDescriptor`, and we specify that it will use a default people detector that is built into OpenCV by running the following code:

```
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
```

Note that we specified the people detector with the `setSVMDetector` method. Hopefully, this makes sense based on the previous section of this chapter; an SVM is a classifier, so the choice of SVM determines the type of object our `cv2.HOGDescriptor` will detect.Now, we proceed to load an image – in this case, an old photograph of women working in a hayfield – and we attempt to detect people in the image by running the following code:

```
img = cv2.imread('../images/haying.jpg')
found_rects, found_weights = hog.detectMultiScale(
    img, winStride=(4, 4), scale=1.02, groupThreshold=1.9)
```

Note that `cv2.HOGDescriptor` has a `detectMultiScale` method, which returns two lists:

1. A list of bounding rectangles for detected objects (in this case, detected people).
2. A list of weights or confidence scores for detected objects. A higher value indicates greater confidence that the detection result is correct.

`detectMultiScale` accepts several optional arguments, including the following:

- `winStride`: This tuple defines the *x* and *y* distance that the sliding window moves between successive detection attempts. HOG works well with overlapping windows, so the stride may be small relative to the window size. A smaller value produces more detections, at a higher computational cost. The default stride has no overlap; it is the same as the window size, which is `(64, 128)` for the default people detector.
- `scale`: This scale factor is applied between successive levels of the image pyramid. A smaller value produces more detections, at a higher computational cost. The value must be greater than `1.0`. The default is `1.5`.
- `groupThreshold`: This value determines how stringent our detection criteria are. A smaller value is less stringent, resulting in more detections. The default is `2.0`.

Now, we can filter the detection results to remove nested rectangles. To determine whether a rectangle is a nested rectangle, we potentially need to compare it to every other rectangle. Note the use of our `is_inside` function in the following nested loop:

```
found_rects_filtered = []
found_weights_filtered = []
for ri, r in enumerate(found_rects):
    for qi, q in enumerate(found_rects):
        if ri != qi and is_inside(r, q):
            break
    else:
        found_rects_filtered.append(r)
        found_weights_filtered.append(found_weights[ri])
```

Finally, let's draw the remaining rectangles and weights in order to highlight the detected people, and let's show and save this visualization, as follows:

```
for ri, r in enumerate(found_rects_filtered):
    x, y, w, h = r
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 255), 2)
    text = '%.2f' % found_weights_filtered[ri]
    cv2.putText(img, text, (x, y - 20),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
cv2.imshow('Women in Hayfield Detected', img)
cv2.imwrite('./women_in_hayfield_detected.jpg', img)
cv2.waitKey(0)
```

If you run the script yourself, you will see rectangles around people in the image. Here is the result:

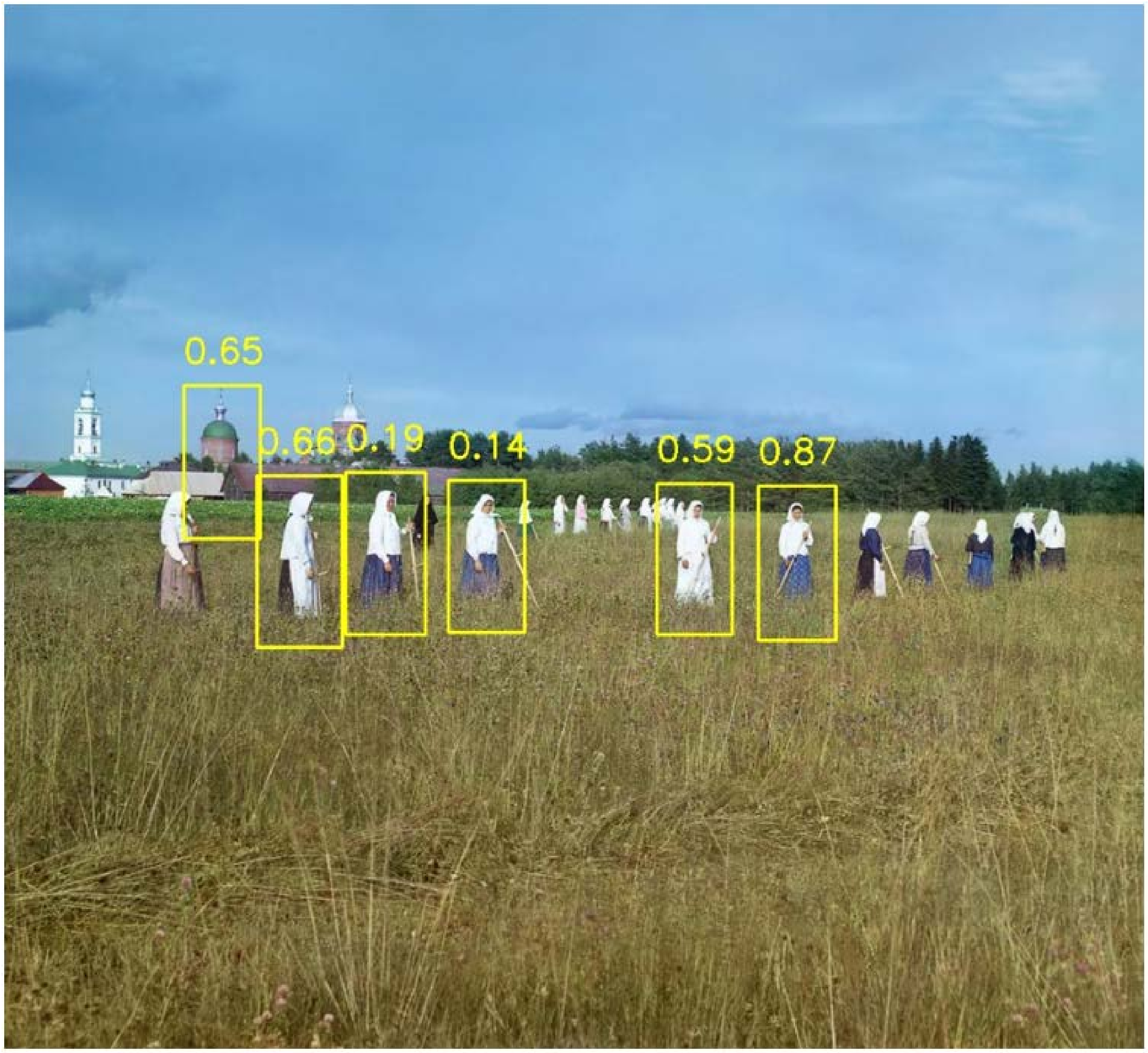


Figure 7.5: People detected using HOG. The photograph is another example of the work of Sergey Prokudin-Gorsky (1863-1944), a pioneer of color photography. Here, the scene is a field at the Leushinskii Monastery, in northwestern Russia, in 1909.

Of the six women who are nearest to the camera, five have been successfully detected. At the same time, a tower in the background has been falsely detected as a person. In many real-world applications, people-detection results can be improved by analyzing a series of frames in a video. For example, imagine that we are looking at a surveillance video of the Leushinskii Monastery's hayfield, instead of a single photograph. We should be able to add code to determine that the tower cannot be a person because it does not move. Also, we should be able to detect additional people in other frames and track each person's movement from frame to frame. We will look at a people-tracking problem in *Chapter 8*, *Tracking Objects*.Meanwhile, let's proceed to look at another kind of detector that we can train to detect a given class of objects.

Creating and training an object detector

Using a pre-trained detector makes it easy to build a quick prototype, and we are all very grateful to the OpenCV developers for making such useful capabilities as face detection and people detection readily available. However, whether you are a hobbyist or a computer vision professional, it is unlikely that you will only deal with people and faces.Moreover, if you are like the authors of this book, you will wonder how the people detector was created in the first place and whether you can improve it. Furthermore, you may also wonder whether you can apply the same concepts to detect diverse objects, ranging from cars to goblins.Indeed, in industry, you may have to deal with problems of detecting very specific objects, such as registration plates, book covers, or whatever thing may be most important to your employer or client.Thus, the question is, how do we come up with our own classifiers?There are many popular approaches. Throughout the remainder of this chapter, we will see that one answer lies in SVMs and the BoW technique.We have already talked about SVMs and HOG. Let's now take a closer look at BoW.

Understanding BoW

BoW is a concept that was not initially intended for computer vision; rather, we use an evolved version of this concept in the context of computer vision. Let's first talk about its basic version, which – as you may have guessed – originally belongs to the field of language analysis and information retrieval.

> Sometimes, in the context of computer vision, BoW is called **bag of visual words (BoVW)**. However, we will simply use the term BoW, since this is the term used by OpenCV.

BoW is the technique by which we assign a weight or count to each word in a series of documents; we then represent these documents with vectors of these counts. Let's look at an example, as follows:

- **Document 1**: I like OpenCV and I like Python.
- **Document 2**: I like C++ and Python.
- **Document 3**: I don't like artichokes.

These three documents allow us to build a dictionary – also called a **codebook** or vocabulary – with these values, as follows:

```
{
    I: 4,
    like: 4,
    OpenCV: 1,
    and: 2,
    Python: 2,
    C++: 1,
    don't: 1,
    artichokes: 1
}
```

We have eight entries. Let's now represent the original documents using eight-entry vectors. Each vector contains values representing the counts of all words in the dictionary, in order, for a given document. The vector representation of the preceding three sentences is as follows:

```
[2, 2, 1, 1, 1, 0, 0, 0]
[1, 1, 0, 1, 1, 1, 0, 0]
[1, 1, 0, 0, 0, 0, 1, 1]
```

These vectors can be conceptualized as a histogram representation of documents or as a descriptor vector that can be used to train classifiers. For example, a document can be classified as *spam* or *not spam* based on such a representation. Indeed, spam filtering is one of the many real-world applications of BoW.Now that we have a grasp of the basic concept of BoW, let's see how this applies to the world of computer vision.

Applying BoW to computer vision

We are by now familiar with the concepts of features and descriptors. We have used algorithms such as SIFT and SURF to extract descriptors from an image's features so that we can match these features in another image.We have also recently familiarized ourselves with another kind of descriptor, based on a codebook or dictionary. We know about an SVM, a model that can accept labeled descriptor vectors as training data, can find an optimal division of the descriptor space into the given classes, and can predict the classes of new data.Armed with this knowledge, we can take the following approach to build a classifier:

1. Take a sample dataset of images.

2. For each image in the dataset, extract descriptors (with SIFT, SURF, ORB, or a similar algorithm).
3. Add each descriptor vector to the BoW trainer.
4. Cluster the descriptors into *k* clusters whose centers (centroids) are our visual words. This last point probably sounds a bit obscure, but we will explore it further in the next section.

At the end of this process, we have a dictionary of visual words ready to be used. As you can imagine, a large dataset will help make our dictionary richer in visual words. Up to a point, the more words, the better!Having trained a classifier, we should proceed to test it. The good news is that the test process is conceptually very similar to the training process outlined previously. Given a test image, we can extract descriptors and **quantize** them (or reduce their dimensionality) by calculating a histogram of their distances to the centroids. Based on this, we can attempt to recognize visual words, and locate them in the image.This is the point in the chapter where you have built up an appetite for a deeper practical example, and are raring to code. However, before proceeding, let's take a quick but necessary digression into the theory of *k*-means clustering so that you can fully understand how visual words are created. Thereby, you will gain a better understanding of the process of object detection using BoW and SVMs.

k-means clustering

k-means clustering is a method of quantization whereby we analyze a large number of vectors in order to find a small number of clusters. Given a dataset, *k* represents the number of clusters into which the dataset is going to be divided. The term *means* refers to the mathematical concept of the mean or the average; when visually represented, the mean of a cluster is its **centroid** or the geometric center of points in the cluster.

> **Clustering** refers to the process of grouping points in a dataset into clusters.

OpenCV provides a class called `cv2.BOWKMeansTrainer`, which we will use to help train our classifier. As you might expect, the OpenCV documentation gives the following summary of this class:

> *A "kmeans-based class to train a visual vocabulary using the bag of words approach."*

After this long theoretical introduction, we can look at an example, and start training our custom classifier.

Detecting cars

To train any kind of classifier, we must begin by creating or acquiring a training dataset. We are going to train a car detector, so our dataset must contain positive samples that represent cars, as well as negative samples that represent other (non-car) things that the detector is likely to encounter while looking for cars. For example, if the detector is intended to search for cars on a street, then a picture of a curb, a crosswalk, a pedestrian, or a bicycle might be a more representative negative sample than a picture of the rings of Saturn. Besides representing the expected subject matter, ideally, the training samples should represent the way our particular camera and algorithm will see the subject matter.Ultimately, in this chapter, we intend to use a sliding window of fixed size, so it is important that our training samples conform to a fixed size, and that the positive samples are tightly cropped in order to frame a car without much background.Up to a point, we expect that the classifier's accuracy will improve as we keep adding good training images. On the other hand, a large dataset can make the training slow, and it is possible to overtrain a classifier such that it fails to extrapolate beyond the training set. Later in this section, we will write our code in a way that allows us to easily modify the number of training images so that we find a good size experimentally.Assembling a dataset of car images would be a time-consuming task if we were to do it all by ourselves (though it is entirely doable). To avoid reinventing the wheel – or the whole car – we can avail ourselves of ready-made datasets, such as the following:

- **UIUC Image Database for Car Detection:** https://cogcomp.seas.upenn.edu/Data/Car/
- **Stanford Cars Dataset**: http://ai.stanford.edu/~jkrause/cars/car_dataset.html

Let's use the UIUC dataset in our example. Several steps are involved in obtaining this dataset and using it in a script, so let's walk through them one by one, as follows:

1. Download the UIUC dataset from https://github.com/gcr/arc-evaluator/raw/master/CarData.tar.gz. Unzip it to some folder, which we will refer to as `<project_path>`. Now, the unzipped data should be located at `<project_path>/CarData`. Specifically, we will use some of the images in `<project_path>/CarData/TrainImages` and `<project_path>/CarData/TestImages`.
2. Also in `<project_path>`, let's create a Python script called `detect_car_bow_svm.py`. To begin the script's implementation, write the following code to check whether the `CarData` subfolder exists:

```
import cv2
import numpy as np
import os
if not os.path.isdir('CarData'):
    print(
        'CarData folder not found. Please download and unzip '
        'https://github.com/gcr/arc-evaluator/raw/master/CarData.tar.gz '
        'into the same folder as this script.')
    exit(1)
```

If you can run this script and it does not print anything, this means everything is in its correct place.

1. Next, let's define the following constants in the script:

```
BOW_NUM_TRAINING_SAMPLES_PER_CLASS = 10
SVM_NUM_TRAINING_SAMPLES_PER_CLASS = 11
BOW_NUM_CLUSTERS = 40
```

Note that our classifier will make use of two training stages: one stage for the BoW vocabulary, which will use a number of images as samples, and another stage for the SVM, which will use a number of BoW descriptor vectors as samples. Arbitrarily, we have defined a different number of training samples for each stage. At each stage, we could have also defined a different number of training samples for the two classes (*car* and *not car*), but instead, we will use the same number. We have also defined the number of BoW clusters. Note that the number of BoW clusters may be larger than the number of BoW training images, since each image has many descriptors.

1. We will use `cv2.SIFT` to extract descriptors and `cv2.FlannBasedMatcher` to match these descriptors. Let's initialize these algorithms with the following code:

```
sift = cv2SIFT_create()
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50))
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

Note that we have initialized SIFT and the **Fast Library for Appropriate Nearest Neighbors** (**FLANN**) in the same way as we did in *Chapter 6*, *Retrieving Images and Searching Using image Descriptors*. However, this time, descriptor matching is not our end goal; instead, it will be part of the functionality of our BoW.

1. OpenCV provides a class called `cv2.BOWKMeansTrainer` to train a BoW vocabulary, and a class called `cv2.BOWImgDescriptorExtractor` to convert some kind of lower-level descriptors – in our example, SIFT descriptors – into BoW descriptors. Let's initialize these objects with the following code:

```
bow_kmeans_trainer = cv2.BOWKMeansTrainer(BOW_NUM_CLUSTERSCLUSTERS)
bow_extractor = cv2.BOWImgDescriptorExtractor(sift, flann)
```

When initializing `cv2.BOWKMeansTrainer`, we must specify the number of clusters – in our example, `40`, as defined earlier. When initializing `cv2.BOWImgDescriptorExtractor`, we must specify a descriptor extractor and a descriptor matcher – in our example, the `cv2.SIFT` and `cv2.FlannBasedMatcher` objects that we created earlier.

1. To train the BoW vocabulary, we will provide samples of SIFT descriptors for various *car* and *not car* images. We will load the images from the `CarData/TrainImages` subfolder, which contains positive (*car*) images with names such as `pos-x.pgm`, and negative (*not car*) images with names such as `neg-x.pgm`, where `x` is a number starting at `1`. Let's write the following utility function to return a pair of paths to the `i` th positive and negative training images, where `i` is a number starting at `0`:

```
def get_pos_and_neg_paths(i):
    pos_path = 'CarData/TrainImages/pos-%d.pgm' % (i+1)
    neg_path = 'CarData/TrainImages/neg-%d.pgm' % (i+1)
    return pos_path, neg_path
```

Later in this section, we will call the preceding function in a loop, with a varying value of `i`, when we need to acquire a number of training samples.

1. For each path to a training sample, we will need to load the image, extract SIFT descriptors, and add the descriptors to the BoW vocabulary trainer. Let's write another utility function to do precisely this, as follows:

```
def add_sample(path):
    img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    if descriptors is not None:
        bow_kmeans_trainer.add(descriptors)
```

If no features are found in the image, then the `keypoints` and descriptors variables will be `None`.

1. At this stage, we have everything we need to start training the BoW vocabulary. Let's read a number of images for each class (*car* as the positive class and *not car* as the negative class) and add them to the training set, as follows:

```
for i in range(BOW_NUM_TRAINING_SAMPLES_PER_CLASS):
    pos_path, neg_path = get_pos_and_neg_paths(i)
    add_sample(pos_path)
    add_sample(neg_path)
```

1. Now that we have assembled the training set, we will call the vocabulary trainer's `cluster` method, which performs the *k*-means classification and returns the vocabulary. We will assign this vocabulary to the BoW descriptor extractor, as follows:

```
voc = bow_kmeans_trainer.cluster()
bow_extractor.setVocabulary(voc)
```

Remember that earlier, we initialized the BoW descriptor extractor with a SIFT descriptor extractor and FLANN matcher. Now, we have also given the BoW descriptor extractor a vocabulary that we trained with samples of SIFT descriptors. At this stage, our BoW descriptor extractor has everything it needs in order to extract BoW descriptors from **Difference of Gaussian** (**DoG**) features.

Remember that `cv2.SIFT` detects DoG features and extracts SIFT descriptors, as we discussed in *Chapter 6*, *Retrieving Images and Searching Using Image Descriptors*, specifically in the Detecting DoG features and extracting SIFT descriptors section.

1. Next, we will declare another utility function that takes an image and returns the descriptor vector, as computed by the BoW descriptor extractor. This involves extracting the image's DoG features, and computing the BoW descriptor vector based on the DoG features, as follows:

```
def extract_bow_descriptors(img):
    features = sift.detect(img)
    return bow_extractor.compute(img, features)
```

1. We are ready to assemble another kind of training set, containing samples of BoW descriptors. Let's create two arrays to accommodate the training data and labels, and populate them with the descriptors generated by our BoW descriptor extractor. We will label each descriptor vector with `1` for a positive sample and `-1` for a negative sample, as shown in the following code block:

```
training_data = []
training_labels = []
for i in range(SVM_NUM_TRAINING_SAMPLES_PER_CLASS):
    pos_path, neg_path = get_pos_and_neg_paths(i)
    pos_img = cv2.imread(pos_path, cv2.IMREAD_GRAYSCALE)
    pos_descriptors = extract_bow_descriptors(pos_img)
    if pos_descriptors is not None:
        training_data.extend(pos_descriptors)
        training_labels.append(1)
    neg_img = cv2.imread(neg_path, cv2.IMREAD_GRAYSCALE)
    neg_descriptors = extract_bow_descriptors(neg_img)
    if neg_descriptors is not None:
        training_data.extend(neg_descriptors)
        training_labels.append(-1)
```

Should you wish to train a classifier to distinguish between multiple positive classes, you can simply add other descriptors with other labels. For example, we could train a classifier that uses the label 1 for *car*, 2 for *person*, and -1 for *background*. There is no requirement to have a negative or background class but, if you do not, your classifier will assume that everything belongs to one of the positive classes.

1. OpenCV provides a class called `cv2.ml_SVM`, representing an SVM. Let's create an SVM, and train it with the data and labels that we previously assembled, as follows:

```
svm = cv2.ml.SVM_create()
svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
```

Note that we must convert the training data and labels from lists to NumPy arrays before we pass them to the `train` method of `cv2.ml_SVM`.

1. Finally, we are ready to test the SVM by classifying some images that were not part of the training set. We will iterate over a list of paths to test images. For each path, we will load the image, extract BoW descriptors, and get the SVM's **prediction** or classification result, which will be either 1.0 (*car*) or -1.0 (*not car*), based on the training labels we used earlier. We will draw text on the image to show the classification result, and we will show the image in a window. After showing all the images, we will wait for the user to hit any key, and then the script will end. All of this is achieved in the following block of code:

```
for test_img_path in ['CarData/TestImages/test-0.pgm',
                      'CarData/TestImages/test-1.pgm',
                      '../images/car.jpg',
                      '../images/haying.jpg',
                      '../images/statue.jpg',
                      '../images/woodcutters.jpg']:
    img = cv2.imread(test_img_path)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    descriptors = extract_bow_descriptors(gray_img)
    prediction = svm.predict(descriptors)
    if prediction[1][0][0] == 1.0:
        text = 'car'
        color = (0, 255, 0)
    else:
        text = 'not car'
        color = (0, 0, 255)
    cv2.putText(img, text, (10, 30),
cv2.FONT_HERSHEY_SIMPLEX, 1,
                color, 2, cv2.LINE_AA)
    cv2.imshow(test_img_path, img)
cv2.waitKey(0)
```

Save and run the script. You should see six windows with various classification results. Here is a screenshot of one of the true positive results:

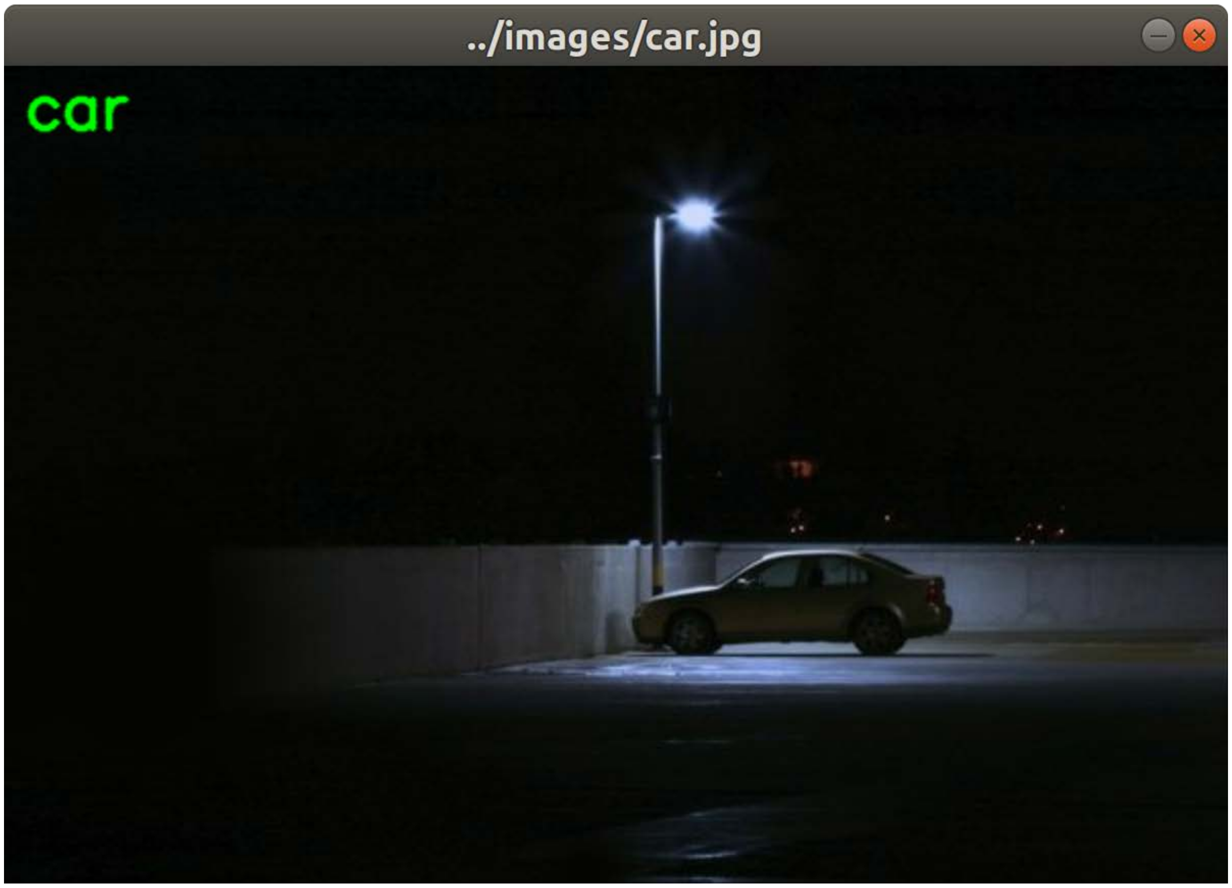


Figure 7.6: An image of a car, correctly classified by our SVM

The next screenshot shows one of the true negative results:

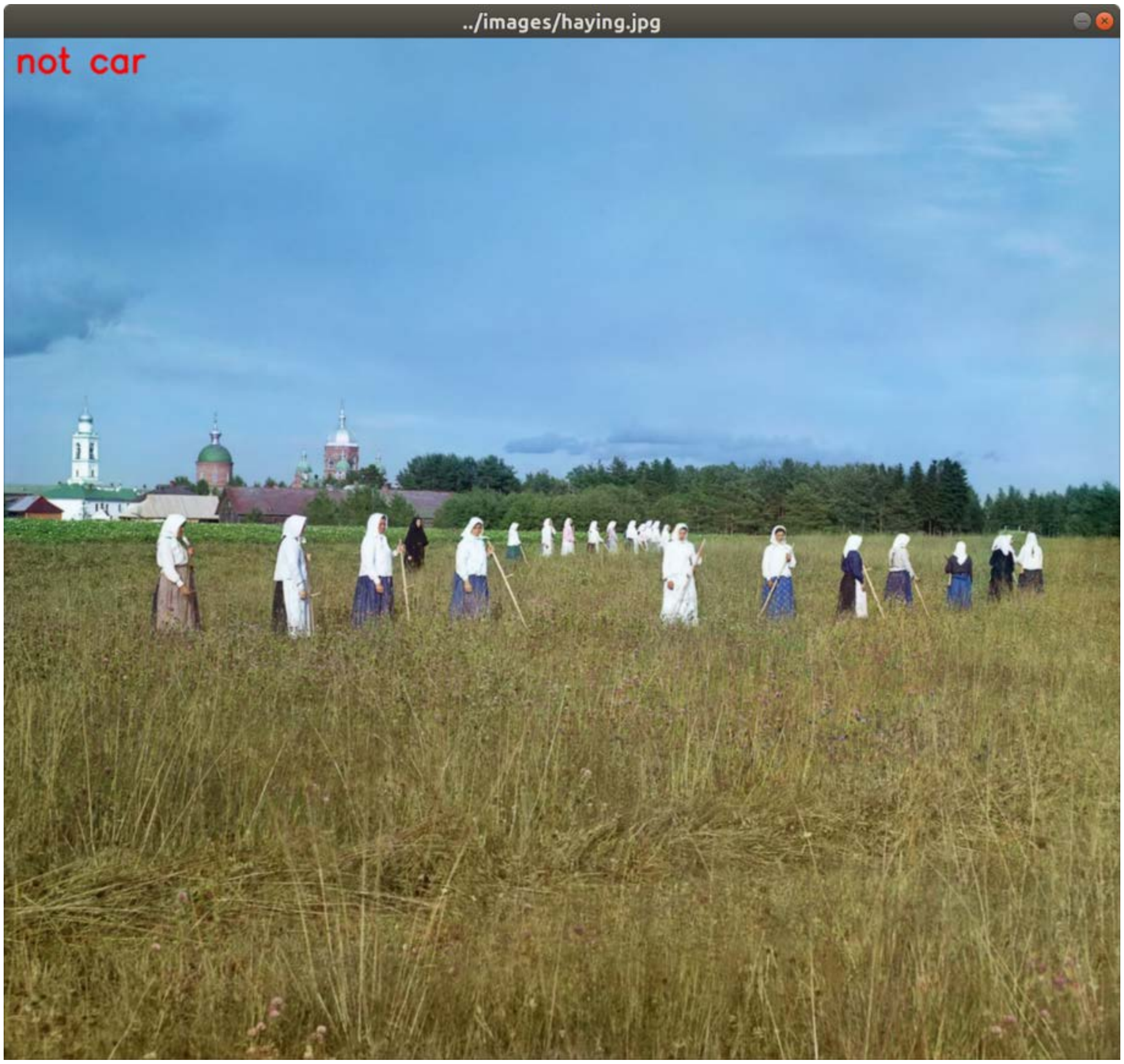


Figure 7.7: A non-car image, correctly classified by our SVM

Of the six images in our simple test, only the following one is incorrectly classified:

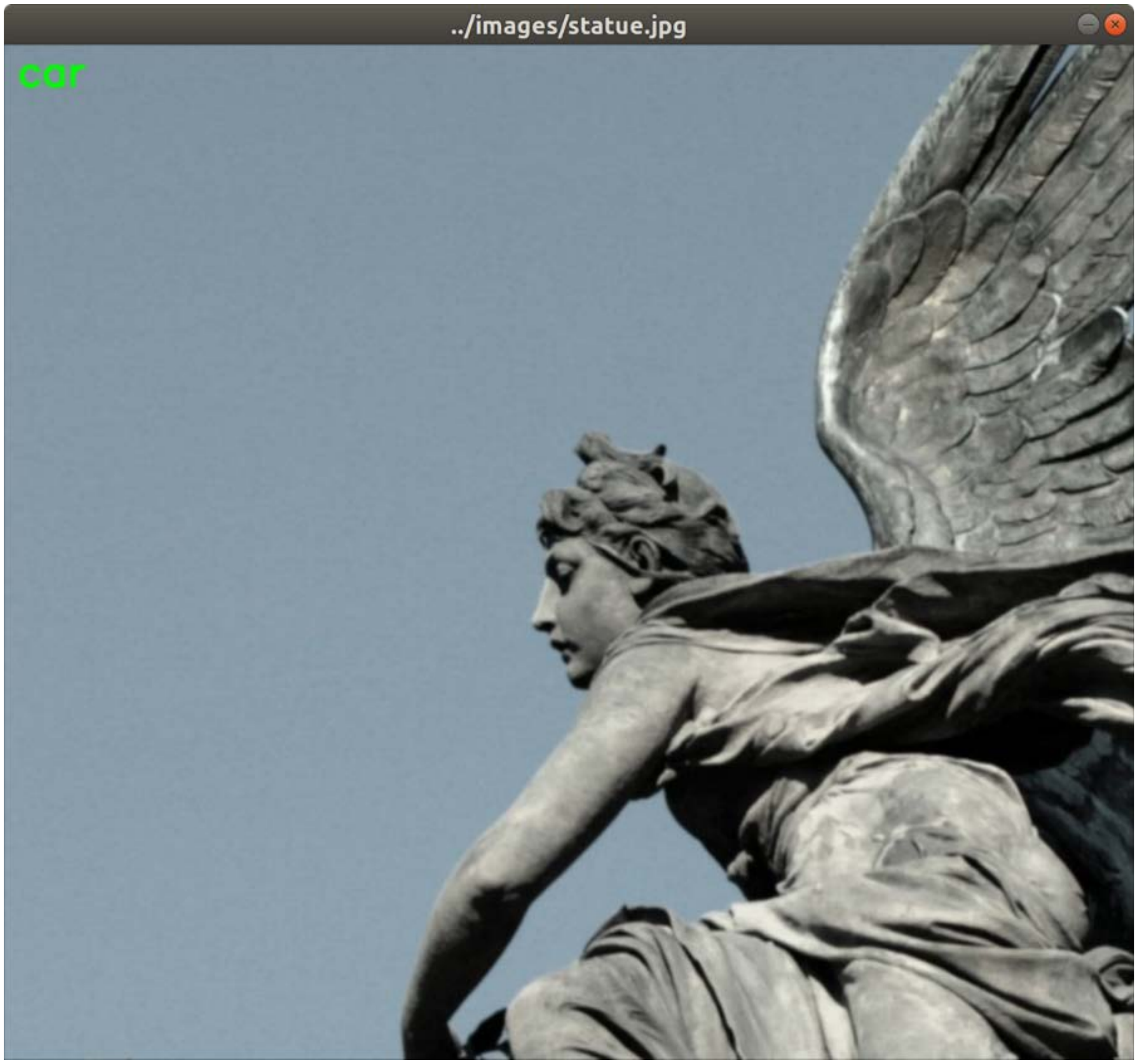


Figure 7.8: A non-car image, incorrectly classified as a car by our SVM

Incorrect classification solely depends upon the amount and nature of positive and negative images used for training and the robustness of the learning algorithm. For example, if the car position in every image of our training dataset is in the center of the image, the test image with a car position at the corner of the image may classify incorrectly. Experiment with adjusting the number of training samples and other parameters, and try testing the classifier on more images, to see what results you can get. Let's take stock of what we have done so far. We have used a mixture of SIFT, BoW, and SVMs to train a classifier to distinguish between two classes: *car* and *not car*. We have applied this classifier to whole images. The next logical step is to apply a sliding window technique so that we can narrow down our classification results to specific regions of an image.

Combining an SVM with a sliding window

By combining our SVM classifier with a sliding window technique and an image pyramid, we can achieve the following improvements:

- Detect multiple objects of the same kind in an image.
- Determine the position and size of each detected object in an image.

We will adopt the following approach:

1. Take a region of the image, classify it, and then move this window to the right by a predefined step size. When we reach the rightmost end of the image, reset the *x* coordinate to **0**, move down a step, and repeat the entire process.
2. At each step, perform a classification with the SVM that was trained with BoW.
3. Keep track of all the windows that are positive detections, according to the SVM.
4. After classifying every window in the entire image, scale the image down, and repeat the entire process of using a sliding window. Thus, we are using an image pyramid. Continue rescaling and classifying until we get to a minimum size.

When we reach the end of this process, we have collected important information about the content of the image. However, there is a problem: in all likelihood, we have found a number of overlapping blocks that each yield a positive detection with high confidence. That is to say, the image may contain one object that gets detected multiple times. If we reported these multiple detections, our report would be quite misleading, so we will filter our results using NMS.

> For a refresher, you may wish to refer back to the *Understanding NMS* section, earlier in this chapter.

Next, let's look at how to modify and extend our previous script, in order to implement the approach we have just described.

Detecting a car in a scene

We are now ready to apply all the concepts we learned so far by creating a car detection script that scans an image and draws rectangles around cars. Let's create a new Python script, `detect_car_bow_svm_sliding_window.py`, by copying our previous script, `detect_car_bow_svm.py`. (We covered the implementation of `detect_car_bow_svm.py` earlier, in the *Detecting cars* section.) Much of the new script's implementation will remain unchanged because we still want to train a BoW descriptor extractor and an SVM in almost the same way as we did previously. However, after the training is complete, we will process the test images in a new way.

Rather than classifying each image in its entirety, we will decompose each image into pyramid layers and windows, we will classify each window, and we will apply NMS to a list of windows that yielded positive detections.For NMS, we will rely on Malisiewicz and Rosebrock's implementation, as described earlier in this chapter, in the *Understanding NMS* section. You can find a slightly modified copy of their implementation in this book's GitHub repository, specifically in the Python script at `chapter07/non_max_suppression.py`. This script provides a function with the following signature:

```
def non_max_suppression_fast(boxes, overlapThresh):
```

As its first argument, the function takes a NumPy array containing rectangle coordinates and scores. If we have *N* rectangles, the shape of this array is *N*x5. For a given rectangle at index *i*, the values in the array have the following meanings:

- `boxes[i][0]` is the leftmost *x* coordinate.
- `boxes[i][1]` is the topmost *y* coordinate.
- `boxes[i][2]` is the rightmost *x* coordinate.
- `boxes[i][3]` is the bottommost *y* coordinate.
- `boxes[i][4]` is the score, where a higher score represents greater confidence that the rectangle is a correct detection result.

As its second argument, the function takes a threshold that represents the maximum proportion of overlap between rectangles. If two rectangles have a greater proportion of overlap than this, the one with the lower score will be filtered out. Ultimately, the function will return an array of the remaining rectangles.Now, let's turn our attention to the modifications to the `detect_car_bow_svm_sliding_window.py` script, as follows:

1. First, we want to add a new import statement for the NMS function, as `highlighted` in the following code:

```
import cv2
import numpy as np
import os
from non_max_suppression import non_max_suppression_fast as nms
```

1. Let's define some additional parameters near the start of the script, as `highlighted` here:

```
BOW_NUM_TRAINING_SAMPLES_PER_CLASS = 10
SVM_NUM_TRAINING_SAMPLES_PER_CLASS = 11
BOW_NUM_CLUSERS = 12
SVM_SCORE_THRESHOLD = 2.22
NMS_OVERLAP_THRESHOLD = 0.44
```

Note that we have reduced the number of *k*-means clusters from `40` to `12` (a number chosen arbitrarily based on experimentation). We will use `SVM_SCORE_THRESHOLD` as a threshold to distinguish between a positive window and a negative window. We will see how the score is obtained a little later in this section. We will use `NMS_OVERLAP_THRESHOLD` as the maximum acceptable proportion of overlap in the NMS step. Here, we have arbitrarily chosen 15%, so we will cull windows that overlap by more than this proportion. As you experiment with your SVMs, you may tweak these parameters to your liking until you find values that yield the best results in your application.

1. We will also adjust the parameters of the SVM, as follows:

```
svm = cv2.ml.SVM_create()
svm.setType(cv2.ml.SVM_C_SVC)
svm.setC(50)
svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
```

With the preceding changes to the SVM, we are specifying the classifier's level of strictness or severity. As the value of the `C` parameter increases, the risk of false positives decreases but the risk of false negatives increases. In our application, a false positive would be a window detected as a car when it is really *not* a car, and a false negative would be a car detected as a window when it really *is* a car.After the code that trains the SVM, we want to add two more helper functions. One of them will generate levels of the image pyramid, and the other will generate regions of interest, based on the sliding window technique. Besides adding these helper functions, we also need to handle the test images differently in order to make use of the sliding window and NMS. The following steps cover the changes:

1. First, let's look at the helper function that deals with the image pyramid. This function is shown in the following code block:

```
def pyramid(img, scale_factor=1.005, min_size=(10040100, 40),
            max_size=(600, 240240)):
    h, w = img.shape
    min_w, min_h = min_size
    max_w, max_h = max_size
    while w >= min_w and h >= min_h:
        if w <= max_w and h <= max_h:
            yield img
        w /= scale_factor
        h /= scale_factor
        img = cv2.resize(img, (int(w), int(h)),
                         interpolation=cv2.INTER_AREA)
```

The preceding function takes an image and generates a series of resized versions of it. The series is bounded by a maximum and minimum image size.

You will have noticed that the resized image is not returned with the `return` keyword but with the `yield` keyword. This is because this function is a so-called generator. It produces a series of images that we can easily use in a loop. If you are not familiar with generators, take a look at the official Python Wiki at https://wiki.python.org/moin/Generators.

1. Next up is the function to generate regions of interest, based on the sliding window technique. This function is shown in the following code block:

```
def sliding_window(img, step=20, window_size=(100, 40)):
    img_h, img_w = img.shape
    window_w, window_h = window_size
    for y in range(0, img_w, step):
        for x in range(0, img_h, step):
            roi = img[y:y+window_h, x:x+window_w]
            roi_h, roi_w = roi.shape
            if roi_w == window_w and roi_h == window_h:
                yield (x, y, roi)
```

Again, this is a generator. Although it is a bit deep-nested, the mechanism is very simple: given an image, return the upper-left coordinates and the sub-image representing the next window. Successive windows are shifted by an arbitrarily sized step from left to right until we reach the end of a row, and from the top to bottom until we reach the end of the image.

1. Now, let's consider the treatment of test images. As in the previous version of the script, we loop through a list of paths to test images, in order to load and process each one. The beginning of the loop is unchanged. For context, here it is:

```
for test_img_path in ['CarData/TestImages/test-0.pgm',
                      'CarData/TestImages/test-1.pgm',
                      '../images/car.jpg',
                      '../images/haying.jpg',
                      '../images/statue.jpg',
                      '../images/woodcutters.jpg']:
    img = cv2.imread(test_img_path)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

For each test image, we iterate over the pyramid levels, and for each pyramid level, we iterate over the sliding window positions. For each window or **region of interest** (**ROI**), we extract BoW descriptors and classify them using the SVM. If the classification produces a positive result that passes a certain confidence threshold, we add the rectangle's corner coordinates and confidence score to a list of positive detections. Continuing from the previous code block, we proceed to handle a given test image with the following code:

```
    pos_rects = []
    for resized in pyramid(gray_img):
        for x, y, roi in sliding_window(resized):
            descriptors = extract_bow_descriptors(roi)
            if descriptors is None:
                continue
            prediction = svm.predict(descriptors)
            if prediction[1][0][0] == 1.0:
                raw_prediction = svm.predict(
                    descriptors,
                    flags=cv2.ml.STAT_MODEL_RAW_OUTPUT)
                score = -raw_prediction[1][0][0]
                if score > SVM_SCORE_THRESHOLD:
                    h, w = roi.shape
                    scale = gray_img.shape[0] / \
                        float(resized.shape[0])
                    pos_rects.append([int(x * scale),
                                      int(y * scale),
                                      int((x+w) * scale),
                                      int((y+h) * scale),
                                      score])
```

Let's take note of a couple of complexities in the preceding code, as follows:

- To obtain a confidence score for the SVM's prediction, we must run the `predict` method with an optional flag, `cv2.ml.STAT_MODEL_RAW_OUTPUT`. Then, instead of returning a label, the method returns a score as part of its output. This score may be negative, and a low value represents a high level of confidence. To make the score more intuitive – and to match the NMS function's assumption that a higher score is better – we negate the score so that a high value represents a high level of confidence.
- Since we are working with multiple pyramid levels, the window coordinates do not have a common scale. We have converted them back to a common scale – the original image's scale – before adding them to our list of positive detections.

So far, we have performed car detection at various scales and positions; as a result, we have a list of detected car rectangles, including coordinates and scores. We expect a lot of overlap within this list of rectangles.

1. Now, let's call the NMS function, in order to cherry-pick the highest-scoring rectangles in the case of overlap, as follows:

   ```
   pos_rects = nms(np.array(pos_rects), NMS_OVERLAP_THRESHOLD)
   ```

   Note that we have converted our list of rectangle coordinates and scores to a NumPy array, which is the format expected by this function.

At this stage, we have an array of detected car rectangles and their scores, and we have ensured that these are the best non-overlapping detections we can select (within the parameters of our model).

1. Now, let's draw the rectangles and their scores by adding the following inner loop to the code:

```
for x0, y0, x1, y1, score in pos_rects:
    cv2.rectangle(img, (int(x0), int(y0)), (int(x1), int(y1)),
                  (0, 255, 255), 2)
    text = '%.2f' % score
    cv2.putText(img, text, (int(x0), int(y0) - 20),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
```

As in the previous version of this script, the body of the outer loop ends by showing the current test image, including the annotations we have drawn on it. After the loop runs through all the test images, we wait for the user to press any key; then, the program ends, as shown here:

```
    cv2.imshow(test_img_path, img)
cv2.waitKey(0)
```

Let's run the modified script, and see how well it can answer the eternal question: *Dude, where's my car?*The following screenshot shows a pair of successful detections:

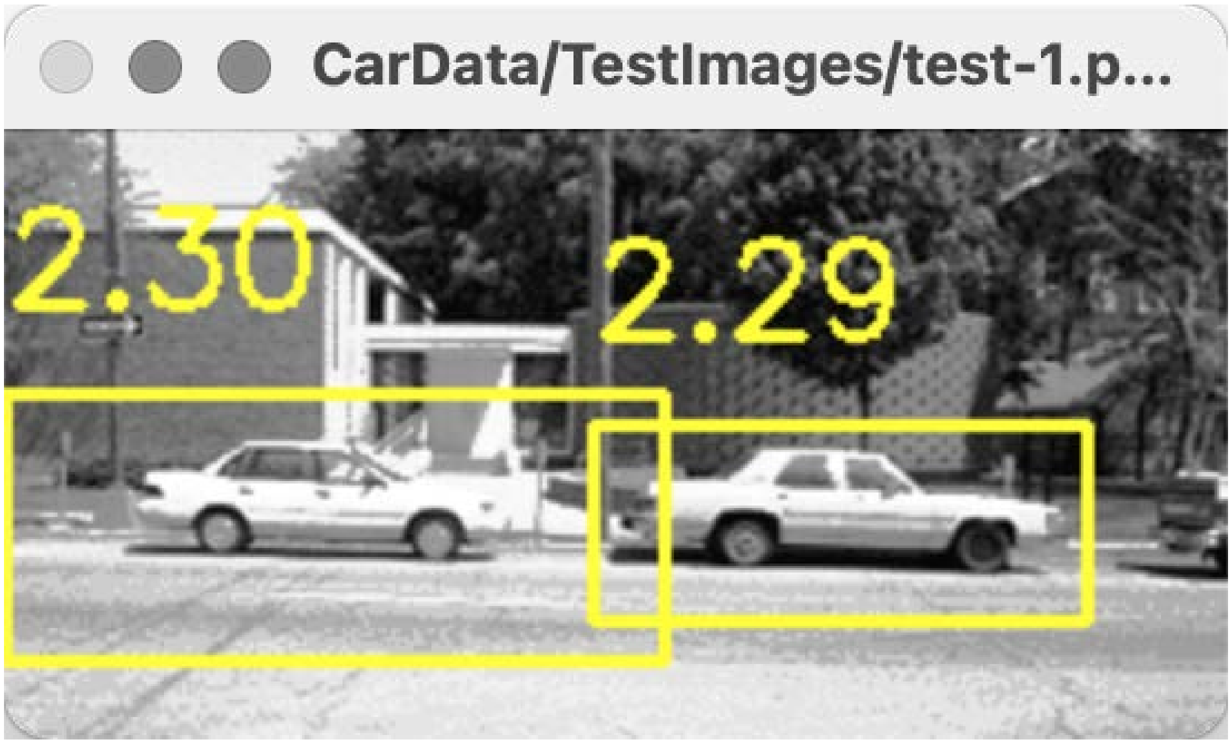


Figure 7.9: A pair of cars, correctly detected by our SVM with a sliding window approach

Among our other 5 test images, we have 2 false negatives (where a car is not detected as such) but no false positives.

> Remember that in this sample script, our training sets are small. Larger training sets, with more diverse backgrounds, could improve the results. Also, remember that the image pyramid and sliding window are producing a large number of ROIs. When we consider this, we should realize that aa low false *positive* rate is actually a significant accomplishment. Moreover, if we were performing detection on frames of a video, we could mitigate the problem of false *negatives* because we would have a chance to detect a car in multiple frames.

Feel free to experiment with the parameters and training sets of the preceding script. When you are ready, let's wrap up this chapter with a few closing notes.

Saving and loading a trained SVM

A final piece of advice on SVMs: you do not need to train a detector every time you want to use it – and, indeed, you should avoid doing so because training is slow. You can use code such as the following to save a trained SVM model to an XML file:

```
svm = cv2.ml.SVM_create()
svm.train(np.array(training_data), cv2.ml.ROW_SAMPLE,
          np.array(training_labels))
svm.save('my_svm.xml')
```

Subsequently, you can reload the trained SVM, using code such as the following:

```
svm = cv2.ml.SVM_create()
svm.load('my_svm.xml')
```

Typically, you might have one script that trains and saves your SVM model, and other scripts that load and use it for various detection problems.

Summary

In this chapter, we covered a wide range of concepts and techniques, including HOG, BoW, SVMs, image pyramids, sliding windows, and NMS. We learned that these techniques have applications in object detection, as well as other fields. We wrote a script that combined most of these techniques – BoW, SVMs, an image pyramid, a sliding window, and NMS – and we gained practical experience in machine learning through the exercise of training and testing a custom detector. Finally, we demonstrated that we can detect cars!Our new knowledge forms the foundation of the next chapter, in which we will utilize object detection and classification techniques on sequences of frames in videos. We will learn how to track objects and retain information about them – an important objective in many real-world applications.