Python Image Processing Cookbook

Over 60 recipes to help you perform complex image processing and computer vision tasks with ease

Sandipan Dey

Packt
www.packt.com

Python Image Processing Cookbook

Over 60 recipes to help you perform complex image processing and computer vision tasks with ease

Sandipan Dey

Packt

BIRMINGHAM - MUMBAI

Python Image Processing Cookbook

Commissioning Editor: Pravin Dhandre
Acquisition Editor: Devika Battike
Content Development Editor: Athikho Sapuni Rishana
Senior Editor: Sofi Rogers
Technical Editor: Manikandan Kurup
Copy Editor: Safis Editing
Project Coordinator: Aishwarya Mohan
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Production Designer: Joshua Misquitta

First published: April 2020

Production reference: 1170420

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

ISBN 978-1-78953-714-7

www.packt.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

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

I dedicate this book to my beloved parents and all the teachers in my life from all the schools:

- Ballygunge Government High School, Kolkata, India
- Jadavpur University, Kolkata, India
- University of Maryland, Baltimore County, USA
- MOOC platforms like Coursera and edX

Contributors

About the author

Sandipan Dey is a data scientist with a wide range of interests, covering topics such as machine learning, deep learning, image processing, and computer vision. He has worked in numerous data science fields, working with recommender systems, predictive models for the events industry, sensor localization models, sentiment analysis, and device prognostics. He earned his master's degree in computer science from the University of Maryland, Baltimore County, and has published in a few IEEE data mining conferences and journals. He has earned certifications from 100+ MOOCs on data science, machine learning, deep learning, image processing, and related courses. He is a regular blogger (sandipanweb) and is a machine learning education enthusiast.

About the reviewers

Srinjoy Ganguly holds a master's in Artificial Intelligence from the University of Southampton, UK and has over 4 years of experience in AI. He completed his bachelor's in Electronics and Communications from GGSIPU, Delhi, and gained a thorough knowledge of signal processing algorithms and neural networks. He has actively pursued the theory and principles of machine learning, deep learning, computer vision, and reinforcement learning. His research interests include Bayesian machine learning, Bayesian deep learning, and quantum machine learning. He is also an active contributor to Julia language. He strongly believes that this book will be an extremely valuable asset into the hands of beginners in the field of AI and computer vision.

Shankar Jha is a software engineer who has helped a lot of beginners get to grips with Python through his articles and talks. He writes about Python, Django, tackling climate change, and poverty alleviation. He lives by the philosophy of Richard Feynman and Swami Vivekananda. His latest endeavor is to help facilitate the use of environment-friendly products. He believes this book will give you enough hands-on experience to jump start your career in computer vision.

Packt is searching for authors like you

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

Table of Contents

Preface

With advancements in wireless devices and mobile technology, there's an increasing demand for digital image processing skills to extract useful information from the ever-growing volume of images. This book provides comprehensive coverage of tools and algorithms, along with guiding you through analysis and visualization for image processing.

With the help of over 60 cutting-edge recipes, you'll address common challenges in image processing and learn how to perform complex tasks such as image detection, image segmentation, and image reconstruction using large hybrid datasets. Dedicated sections will also take you through implementing various image enhancement and image restoration techniques such as cartooning, gradient blending, and sparse dictionary learning. As you advance, you'll get to grips with face morphing and image segmentation techniques. With an emphasis on practical solutions, this book will help you apply deep learning-based techniques such as transfer learning and fine-tuning to solve real-world problems.

By the end of this Python book, you'll be proficient in applying image processing techniques effectively to leverage the capabilities of the Python ecosystem.

Who this book is for

This Python cookbook is for image processing engineers, computer vision engineers, software developers, machine learning engineers, or anyone who wants to become well-versed with image processing techniques and methods using a recipe-based approach. Although no image processing knowledge is expected, prior Python coding experience is necessary to understand key concepts covered in the book.

What this book covers

`Chapter 1`, *Image Manipulation and Transformation*, is where you will learn how to use different Python libraries (NumPy, SciPy, scikit-image, OpenCV, and Matplotlib) for image manipulation and transformation. You will learn how to write Python code to do point transforms (log/gamma transform, Gotham filter, colorspace transformation, and increasing brightness/contrast) and geometric transforms (swirl transform, perspective transform, and homography).

`Chapter 2`, *Image Enhancement*, is where you will learn how to use different Python libraries (NumPy, SciPy, scikit-image, OpenCV, PyWavelet, and MedPy) to denoise images (using linear/nonlinear filters, **Fast Fourier transform** (**FFT**), and autoencoders). You'll learn how to implement image enhancement techniques such as histogram equalization/matching, sketching/cartoonizing, pyramid blending/gradient blending, and edge detection with zero crossing.

`Chapter 3`, *Image Restoration*, is where you will learn how to implement image restoration (using NumPy, scikit-image, OpenCV, and scikit-learn) with deconvolution (inverse/weiner/LMS) filters. You'll learn how to implement image restoration with inpainting, variational methods, and sparse dictionary learning. You'll also learn how to implement steganography/steganalysis techniques with pysteg.

`Chapter 4`, *Binary Image Processing*, is where you will learn how to use different Python libraries (NumPy, SciPy, scikit-image, and OpenCV) for binary image processing (with mathematical morphology). You'll learn how to implement morphological operators, filters, and pattern matching and how to apply them in segmentation, fingerprint enhancement, counting objects, and blob separation.

`Chapter 5`, *Image Registration*, is where you will learn how to use different Python libraries (NumPy, scikit-image, OpenCV, and PyStasm) for image matching/registration/stitching. You'll learn how to implement image registration techniques with warping/feature (SIFT/SURF/ORB)-based methods and the RANSAC algorithm. You'll also learn how to implement panorama image creation, and face morphing, as well as how to implement a basic image search engine.

`Chapter 6`, *Image Segmentation*, is where you will learn how to use different Python libraries (NumPy, scikit-image, OpenCV, SimpleITK, and DeepLab) for image segmentation. You'll learn how to implement image segmentation techniques with graph-based methods/clustering methods, super-pixelation, and machine learning algorithms. You'll also learn how to implement semantic segmentation with DeepLab.

`Chapter 7`, *Image Classification*, is where you will learn how to use different Python libraries (scikit-learn, OpenCV, TensorFlow, Keras, and PyTorch) for image classification. You'll learn how to implement deep learning-based techniques such as transfer learning/fine-tuning. You'll learn how to implement panorama image creation and face morphing. You'll also learn how to implement deep learning-based classification techniques for hand gestures and traffic signals.

`Chapter 8`, *Object Detection in Images*, is where you will learn how to use different Python libraries (scikit-learn, OpenCV, TensorFlow, Keras, and PyTorch) for object detection in images. You'll learn how to implement classical machine learning (HOG/SVM) techniques as well as deep learning models to detect objects. You'll also learn how to implement barcode detection and text detection from images.

`Chapter 9`, *Face Detection and Recognition*, is where you will learn how to use different Python libraries (scikit-learn, OpenCV, dlib, TensorFlow, Keras, PyTorch, DeepFace, and FaceNet) for face detection in images. You'll also learn how to implement facial keypoint recognition and facial/emotion/gender recognition with deep learning.

To get the most out of this book

Basic knowledge of Python and image processing is required to understand and run the code, along with access to a few online image datasets and the book's GitHub link.

Python 3.5+ (Python 3.7.4 was used to test the code) is needed with Anaconda preferably installed for the Windows users, along with Jupyter (to view/run notebooks).

All the code was tested on Windows 10 (Pro) with 32 GB RAM and an Intel i7-series processor. However, the code should require little/no change to be run on Linux.

You will need to install all the required Python packages using pip3.

Access to a GPU is recommended to run the recipes involving training with deep learning (that is, training that involves libraries such as TensorFlow, Keras, and PyTorch) much faster. The code that is best run with a GPU was tested on an Ubuntu 16.04 machine with an Nvidia Tesla K80 GPU (with CUDA 10.1).

A basic math background is also needed to understand the concepts in the book.

Software/hardware covered in the book	**OS requirements**
Python 3.7.4.	Windows 10.
Anaconda version 2019.10 (py37_0).	Windows 10.
For the GPU, you will need an NVIDIA graphics card or access to an AWS GPU instance (`https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html`) or Google Colab (`https://colab.research.google.com/`).	Windows 10/Linux (Ubuntu 16).

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

To access the notebooks and images, clone the repository from this URL: `https://github.com/PacktPublishing/Python-Image-Processing-Cookbook`.

Install Python 3.7 and the necessary libraries as and when required. Install Anaconda/Jupyter and open the notebooks for each chapter. Run the code for each recipe. Follow the instructions for each recipe for any additional steps (for instance, you may need to download a pre-trained model or an image dataset).

Some additional exercises are provided for most of the recipes in a *There's more...* section to test your understanding. Perform them independently and have fun!

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Python-Image-Processing-Cookbook`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781789537147_ColorImages.pdf`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Implement a `bilinear_interpolate()` function, which interpolates over every image channel."

A block of code is set as follows:

```
def get_grid_coordinates(points):
  xmin, xmax = np.min(points[:, 0]), np.max(points[:, 0]) + 1
  ymin, ymax = np.min(points[:, 1]), np.max(points[:, 1]) + 1
  return np.asarray([(x, y) for y in range(ymin, ymax)
         for x in range(xmin, xmax)], np.uint32)
```

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

```
$ pip install mtcnn
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "**Face alignment** is a data normalization process—an essential preprocessing step for many facial recognition algorithms."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

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

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

Getting ready

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

How to do it…

This section contains the steps required to follow the recipe.

How it works…

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

There's more…

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

See also

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

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

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

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

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

Reviews

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

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

1
Image Manipulation and Transformation

Image transformation is the art of transforming an image. With image transformation and manipulation, we can enhance the appearance of an image. The transformation and manipulation operation can also be used as preprocessing steps for more complex image processing tasks, such as classification or segmentation, which you will get more acquainted with in later chapters. In this chapter, you are going to learn how to use different Python libraries (NumPy, SciPy, scikit-image, OpenCV-Python, Mahotas, and Matplotlib) for image manipulation and transformation. Different recipes will help you to learn how to write Python code to implement color space transformation, geometric transformations, perspective transforms/homography, and so on.

In this chapter, we will cover the following recipes:

- Transforming color space (RGB → Lab)
- Applying affine transformation
- Applying perspective transformation and homography
- Creating pencil sketches from images
- Creating cartoonish images
- Simulating light art/long exposure
- Object detection using color in HSV

Technical requirements

To run the codes without any errors, you need to first install Python 3 (for example, 3.6) and the required libraries, if they are not already installed. If you are working on Windows, you are recommended to install the Anaconda distribution. You also need to install the `jupyter` library to work with the notebooks.

All of the code files in this book are available in the GitHub repository at `https://github.com/PacktPublishing/Python-Image-Processing-Cookbook`. You should clone the repository (to your working directory). Corresponding to each chapter, there is a folder and each folder contains a notebook with the complete code (for all of the recipes for each chapter); a subfolder named `images`, which contains all the input images (and related files) required for that chapter; and (optionally) another sub-folder named `models`, which contains the models and related files to be used for the recipes in that chapter.

Transforming color space (RGB → Lab)

The **CIELAB** (abbreviated as **Lab**) color space consists of three color channels, expressing the color of a pixel as three tuples **(L, a, b)**, where the **L** channel stands for **luminosity/illumination/intensity** (lightness). The **a** and **b** channels represent the **green-red** and **blue-yellow** color components, respectively. This color model separates the *intensity* from the *colors* completely. It's device-independent and has a large *gamut*. In this recipe, you will see how to convert from RGB into the Lab color space and vice versa and the usefulness of this color model.

Getting ready

In this recipe, we will use a flower RGB image as the input image. Let's start by importing the required libraries with the following code block:

```
import numpy as np
from skimage.io import imread
from skimage.color import rgb2lab, lab2rgb
import matplotlib.pylab as plt
```

How to do it...

In this recipe, you will see a few remarkable uses of the Lab color space and how it makes some image manipulation operations easy and elegant.

Converting RGB image into grayscale by setting the Lab space color channels to zero

Perform the following steps to convert an RGB color image into a grayscale image using the Lab color space and `scikit-image` library functions:

1. Read the input image. Perform a color space transformation—from RGB to Lab color space:

```
im = imread('images/flowers.png')
im1 = rgb2lab(im)
```

2. Set the color channel values (the second and third channels) to zeros:

```
im1[...,1] = im1[...,2] = 0
```

3. Obtain the grayscale image by converting the image back into the RGB color space from the Lab color space:

```
im1 = lab2rgb(im1)
```

4. Plot the input and output images, as shown in the following code:

```
plt.figure(figsize=(20,10))
plt.subplot(121), plt.imshow(im), plt.axis('off'),
plt.title('Original image', size=20)
plt.subplot(122), plt.imshow(im1), plt.axis('off'), plt.title('Gray
scale image', size=20)
plt.show()
```

The following screenshot shows the output of the preceding code block:

Changing the brightness of the image by varying the luminosity channel

Perform the following steps to change the brightness of a colored image using the Lab color space and `scikit-image` library functions:

1. Convert the input image from RGB into the Lab color space and increase the first channel values (by `50`):

```
im1 = rgb2lab(im)
im1[...,0] = im1[...,0] + 50
```

2. Convert it back into the RGB color space and obtain a brighter image:

```
im1 = lab2rgb(im1)
```

3. Convert the RGB image into the Lab color space and decrease only the first channel values (by `50`, as seen in the following code) and then convert back into the RGB color space to get a darker image instead:

```
im1 = rgb2lab(im)
im1[...,0] = im1[...,0] - 50
im1 = lab2rgb(im1)
```

If you run the preceding code and plot the input and output images, you will get an output similar to the one shown in the following screenshot:

How it works...

The `rgb2lab()` function from the scikit-image `color` module was used to convert an image from RGB into the Lab color space.

The modified image in the Lab color space was converted back into RGB using the `lab2rgb()` function from the scikit-image `color` module.

Since the color channels are separated in the a and b channels and in terms of intensity in the L channel by setting the color channel values to zero, we can obtain the grayscale image from a colored image in the Lab space.

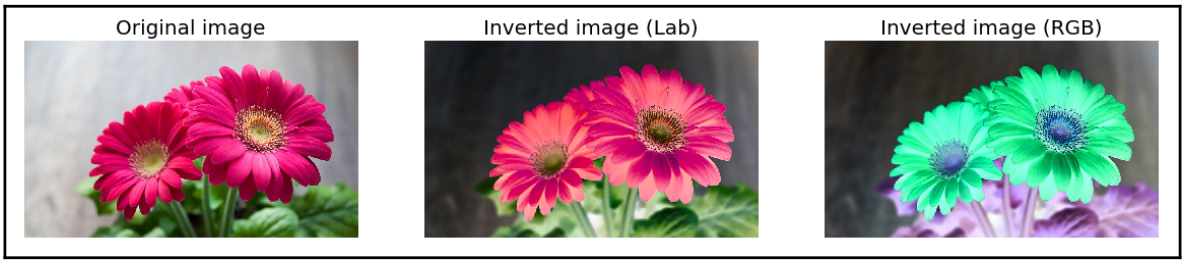
The brightness of the input color image was changed by changing only the L channel values in the Lab space (unlike in the RGB color space where all the channel values need to be changed); there is no need to touch the color channels.

There's more...

There are many other uses of the Lab color space. For example, you can obtain a more natural inverted image in the Lab space since only the luminosity channel needs to be inverted, as demonstrated in the following code block:

```
im1 = rgb2lab(im)
im1[...,0] = np.max(im1[...,0]) - im1[...,0]
im1 = lab2rgb(im1)
```

If you run the preceding code and display the input image and the inverted images obtained in the RGB and the Lab space, you will get the following screenshot:



As you can see, the **Inverted image** in the **Lab** color space appears much more natural than the **Inverted image** in the **RGB** color space.

Applying affine transformation

An **affine transformation** is a geometric transformation that preserves points, straight lines, and planes. Lines that are parallel before the transform remain parallel post-application of the transform. For every pixel x in an image, the affine transformation can be represented by the mapping, $x \mapsto Mx+b$, where M is a linear transform (matrix) and b is an offset vector.

In this recipe, we will use the `scipy ndimage` library function, `affine_transform()`, to implement such a transformation on an image.

Getting ready

First, let's import the libraries and the functions required to implement an affine transformation on a grayscale image:

```
import numpy as np
from scipy import ndimage as ndi
from skimage.io import imread
from skimage.color import rgb2gray
```

How to do it...

Perform the following steps to apply an affine transformation to an image using the `scipy.ndimage` module functions:

1. Read the color image, convert it into grayscale, and obtain the grayscale image shape:

```
img = rgb2gray(imread('images/humming.png'))
w, h = img.shape
```

2. Apply identity transform:

```
mat_identity = np.array([[1,0,0],[0,1,0],[0,0,1]])
img1 = ndi.affine_transform(img, mat_identity)
```

3. Apply reflection transform (along the *x* axis):

```
mat_reflect = np.array([[1,0,0],[0,-1,0],[0,0,1]]) @
np.array([[1,0,0],[0,1,-h],[0,0,1]])
img1 = ndi.affine_transform(img, mat_reflect) # offset=(0,h)
```

4. Scale the image (`0.75` times along the *x* axis and `1.25` times along the *y* axis):

```
s_x, s_y = 0.75, 1.25
mat_scale = np.array([[s_x,0,0],[0,s_y,0],[0,0,1]])
img1 = ndi.affine_transform(img, mat_scale)
```

5. Rotate the image by 30° counter-clockwise. It's a composite operation—first, you will need to shift/center the image, apply rotation, and then apply inverse shift:

```
theta = np.pi/6
mat_rotate = np.array([[1,0,w/2],[0,1,h/2],[0,0,1]]) @
np.array([[np.cos(theta),np.sin(theta),0],[np.sin(theta),-
np.cos(theta),0],[0,0,1]]) @ np.array([[1,0,-w/2],[0,1,-
h/2],[0,0,1]])
img1 = ndi.affine_transform(img1, mat_rotate)
```

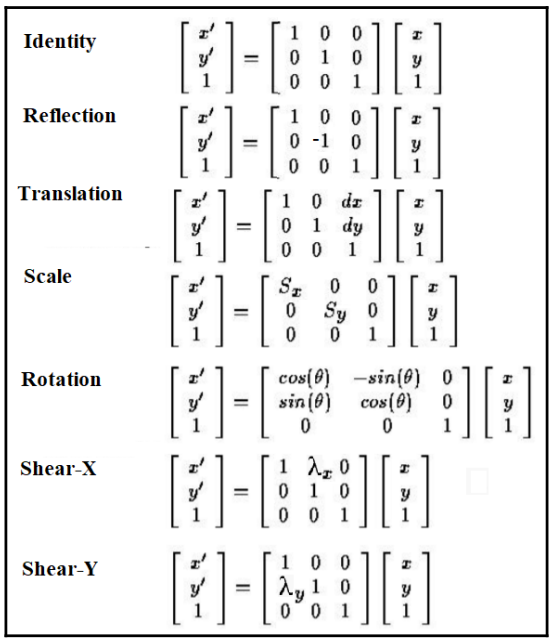
6. Apply shear transform to the image:

```
lambda1 = 0.5
mat_shear = np.array([[1,lambda1,0],[lambda1,1,0],[0,0,1]])
img1 = ndi.affine_transform(img1, mat_shear)
```

7. Finally apply all of the transforms together, in sequence:

```
mat_all = mat_identity @ mat_reflect @ mat_scale @ mat_rotate @
mat_shear
ndi.affine_transform(img, mat_all)
```

The following screenshot shows the matrices (*M*) for each of the affine transformation operations:



How it works...

Note that, for an image, the *x* axis is the vertical (**+ve downward**) axis and the *y* axis is the horizontal (**+ve left-to-right**) axis.

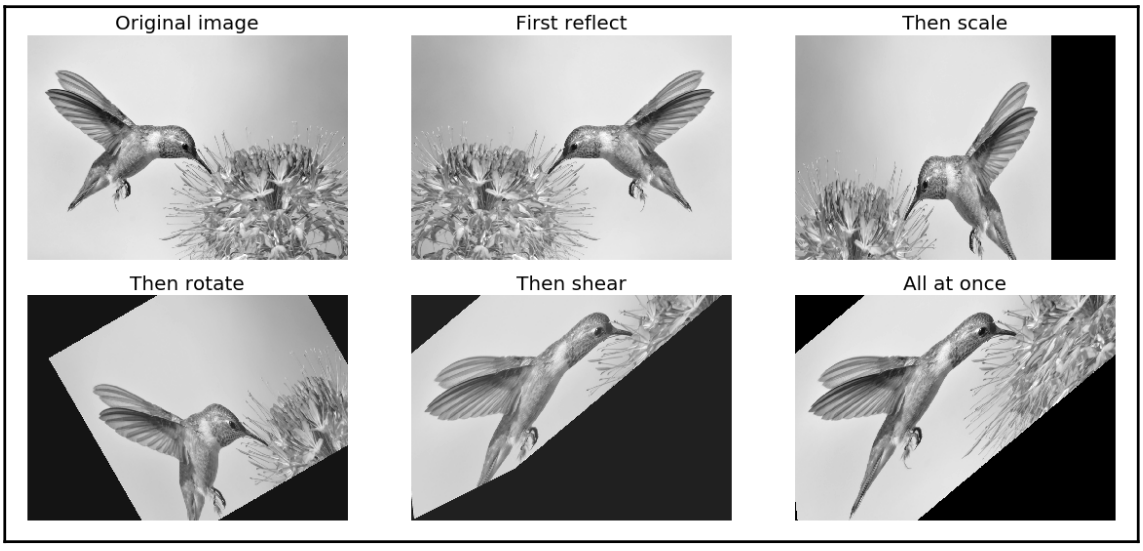
With the `affine_transform()` function, the pixel value at location **o** in the output (transformed) image is determined from the pixel value in the input image at *position np.dot(matrix, o) + offset*. Hence, the matrix that needs to be provided as input to the function is actually the inverse transformation matrix.

In some of the cases, an additional matrix is used for translation, to bring the transformed image within the frame of visualization.

The preceding code snippets show how to implement different affine transformations such as **reflection**, **scaling**, **rotation**, and **shear** using the `affine_transform()` function. We need to provide the proper transformation matrix, *M* (shown in the preceding diagram) for each of these cases (homogeneous coordinates are used).

We can use the product of all of the matrices to perform a combination of all of the affine transformations at once (for instance, if you want transformation *T1* followed by *T2*, you need to multiply the input image by the matrix *T2.T1*).

If all of the transformations are applied in sequence and the transformed images are plotted one by one, you will obtain an output like the following screenshot:

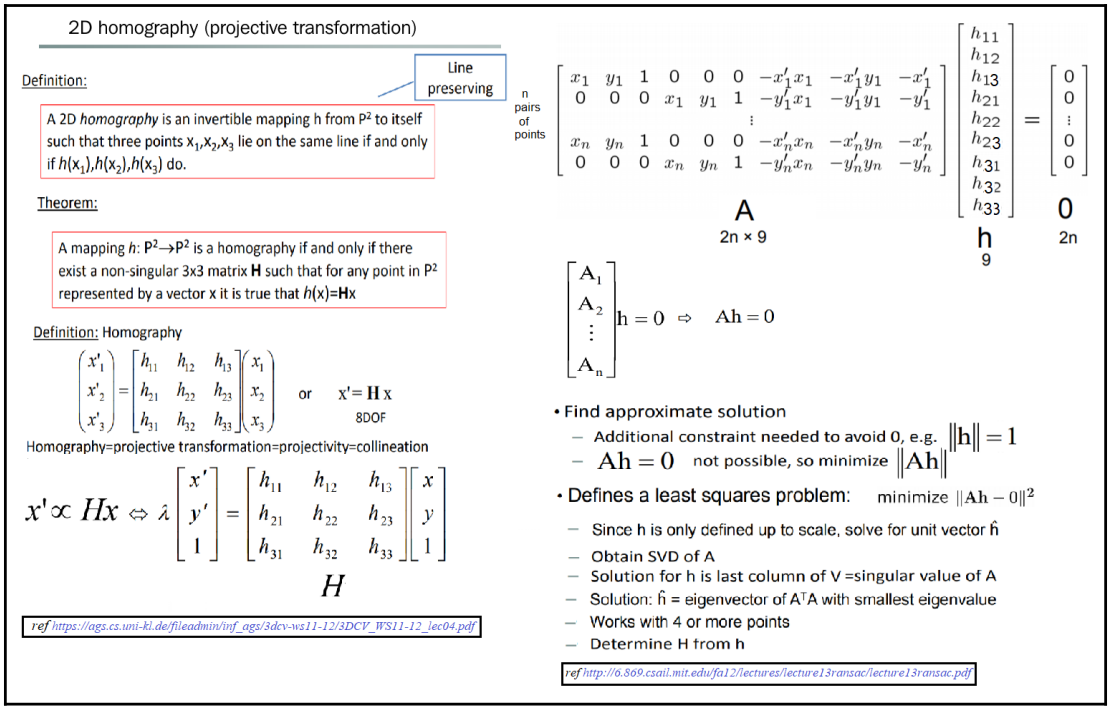


There's more...

Again, in the previous example, the `affine_transform()` function was applied to a grayscale image. The same effect can be obtained with a color image also, such as by applying the mapping function to each of the image channels simultaneously and independently. Also, the `scikit-image` library provides the `AffineTransform` and `PiecewiseAffineTransform` classes; you may want to try them to implement affine transformation as well.

Applying perspective transformation and homography

The goal of perspective (projective) transform is to estimate homography (a matrix, **H**) from point correspondences between two images. Since the matrix has a **Depth Of Field** (**DOF**) of eight, you need at least four pairs of points to compute the homography matrix from two images. The following diagram shows the basic concepts required to compute the homography matrix:



Fortunately, we don't need to compute the SVD and the *H* matrix is computed automatically by the `ProjectiveTransform` function from the scikit-image `transform` module. In this recipe, we will use this function to implement homography.

Getting ready

We will use a humming bird's image and an image of an astronaut on the moon (taken from NASA's public domain images) as input images in this recipe. Again, let's start by importing the required libraries as usual:

```
from skimage.transform import ProjectiveTransform
from skimage.io import imread
import numpy as np
import matplotlib.pylab as plt
```

How to do it...

Perform the following steps to apply a projective transformation to an image using the `transform` module from scikit-image:

1. First, read the source image and create a destination image with the `np.zeros()` function:

```
im_src = (imread('images/humming2.png'))
height, width, dim = im_src.shape
im_dst = np.zeros((height, width, dim))
```

2. Create an instance of the `ProjectiveTransform` class:

```
pt = ProjectiveTransform()
```

3. You just need to provide four pairs of matching points between the source and destination images to estimate the homography matrix, *H*, automatically for you. Here, the four corners of the destination image and the four corners of the input hummingbird are provided as matching points, as shown in the following code block:

```
src = np.array([[ 295., 174.],
 [ 540., 146. ],
 [ 400., 777.],
 [ 60., 422.]])
dst = np.array([[ 0., 0.],
 [height-1, 0.],
 [height-1, width-1],
 [ 0., width-1]])
```

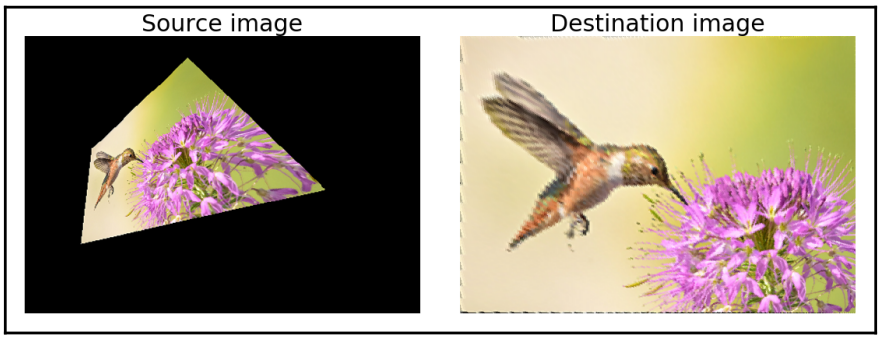
4. Obtain the source pixel index corresponding to each pixel index in the destination:

```
x, y = np.mgrid[:height, :width]
dst_indices = np.hstack((x.reshape(-1, 1), y.reshape(-1,1)))
src_indices = np.round(pt.inverse(dst_indices), 0).astype(int)
valid_idx = np.where((src_indices[:,0] < height) &
(src_indices[:,1] < width) &
                     (src_indices[:,0] >= 0) & (src_indices[:,1] >=
0))
dst_indicies_valid = dst_indices[valid_idx]
src_indicies_valid = src_indices[valid_idx]
```

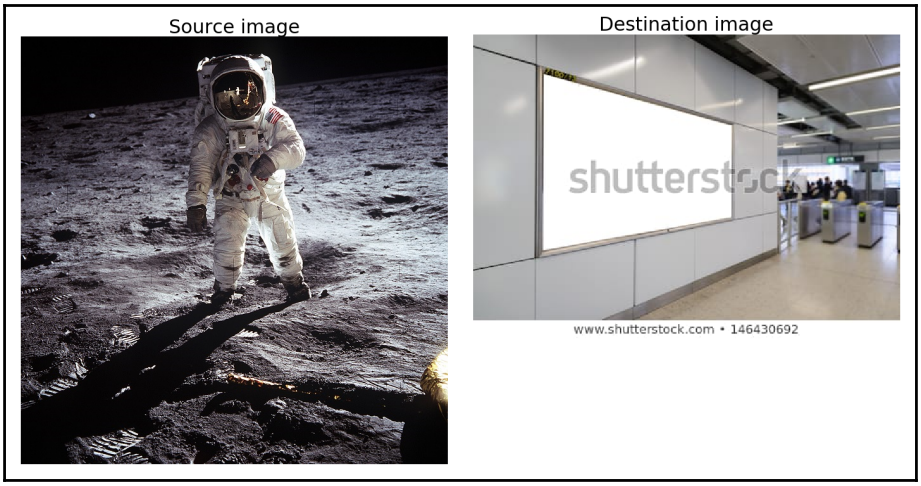
5. Copy pixels from the source to the destination images:

```
im_dst[dst_indicies_valid[:,0],dst_indicies_valid[:,1]] =
im_src[src_indicies_valid[:,0],src_indicies_valid[:,1]]
```

If you run the preceding code snippets, you will get an output like the following screenshot:



The next screenshot shows the source image of an astronaut on the moon and the destination image of the canvas. Again, by providing four pairs of mapping points in between the source (corner points) and destination (corners of the canvas), the task is pretty straightforward:



The following screenshot shows the output image after the projective transform:

How it works...

In both of the preceding cases, the input image is projected onto the desired location of the output image.

A `ProjectiveTransform` object is needed to be created first to apply perspective transform to an image.

A set of 4-pixel positions from the source image and corresponding matching pixel positions in the destination image are needed to be passed to the `estimate()` function along with the object instance and this computes the homography matrix, *H* (and returns `True` if it can be computed).

The `inverse()` function is to be called on the object and this will give you the source pixel indices corresponding to all destination pixel indices.

There's more...

You can use the `warp()` function (instead of the `inverse()` function) to implement homography/projective transform.

See also

For more details, refer to the following links:

- `https://www.youtube.com/watch?v=YwIB9PbQkEM`
- `https://www.youtube.com/watch?v=2ggjHjRx2SQ`
- `https://www.youtube.com/watch?v=vviNh5y71ss`

Creating pencil sketches from images

Producing sketches from images is all about detecting edges in images. In this recipe, you will learn how to use different techniques, including the difference of **Gaussian** (and its extended version, **XDOG**), anisotropic diffusion, and dodging (applying Gaussian blur + invert + thresholding), to obtain sketches from images.

Getting ready

The following libraries need to be imported first:

```
import numpy as np
from skimage.io import imread
from skimage.color import rgb2gray
from skimage import util
from skimage import img_as_float
import matplotlib.pylab as plt
from medpy.filter.smoothing import anisotropic_diffusion
from skimage.filters import gaussian, threshold_otsu
```

How to do it...

The following steps need to be performed:

1. Define the `normalize()` function to implement min-max normalization in an image:

```
def normalize(img):
    return (img-np.min(img))/(np.max(img)-np.min(img))
```

2. Implement the `sketch()` function that takes an image and the extracted edges as input:

```
def sketch(img, edges):
    output = np.multiply(img, edges)
    output[output>1]=1
    output[edges==1]=1
    return output
```

3. Implement a function to extract the edges from an image with anisotropic diffusion:

```
def edges_with_anisotropic_diffusion(img, niter=100, kappa=10,
gamma=0.1):
    output = img - anisotropic_diffusion(img, niter=niter, \
            kappa=kappa, gamma=gamma, voxelspacing=None, \
            option=1)
    output[output > 0] = 1
    output[output < 0] = 0
    return output
```

4. Implement a function to extract the edges from an image with the `dodge` operation (there are two implementations):

```
def sketch_with_dodge(img):
 orig = img
 blur = gaussian(util.invert(img), sigma=20)
 result = blur / util.invert(orig)
 result[result>1] = 1
 result[orig==1] = 1
 return result

def edges_with_dodge2(img):
 img_blurred = gaussian(util.invert(img), sigma=5)
 output = np.divide(img, util.invert(img_blurred) + 0.001)
 output = normalize(output)
 thresh = threshold_otsu(output)
 output = output > thresh
 return output
```

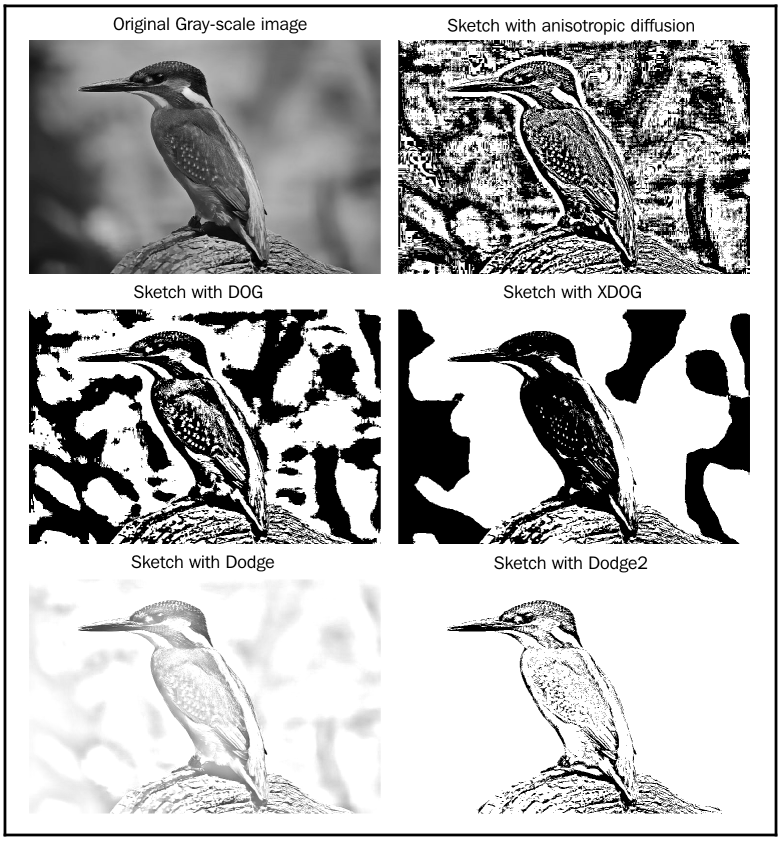
5. Implement a function to extract the edges from an image with a **Difference of Gaussian** (**DOG**) operation:

```
def edges_with_DOG(img, k = 200, gamma = 1):
    sigma = 0.5
    output = gaussian(img, sigma=sigma) - gamma*gaussian(img, \
                      sigma=k*sigma)
    output[output > 0] = 1
    output[output < 0] = 0
    return output
```

6. Implement a function to produce sketches from an image with an **Extended Difference of Gaussian** (**XDOG**) operation:

```
def sketch_with_XDOG(image, epsilon=0.01):
    phi = 10
    difference = edges_with_DOG(image, 200, 0.98).astype(np.uint8)
    for i in range(0, len(difference)):
        for j in range(0, len(difference[0])):
            if difference[i][j] >= epsilon:
                difference[i][j] = 1
            else:
                ht = np.tanh(phi*(difference[i][j] - epsilon))
            difference[i][j] = 1 + ht
    difference = normalize(difference)
    return difference
```

If you run the preceding code and plot all of the input/output images, you will obtain an output like the following screenshot:



How it works...

As you can see from the previous section, many of the sketching techniques work by blurring the edges (for example, with Gaussian filter or diffusion) in the image and removing details to some extent and then subtracting the original image to get the sketch outlines.

The `gaussian()` function from the scikit-image filters module was used to blur the images.

The `anisotropic_diffusion()` function from the `filter.smoothing` module of the `medpy` library was used to find edges with anisotropic diffusion (a variational method).

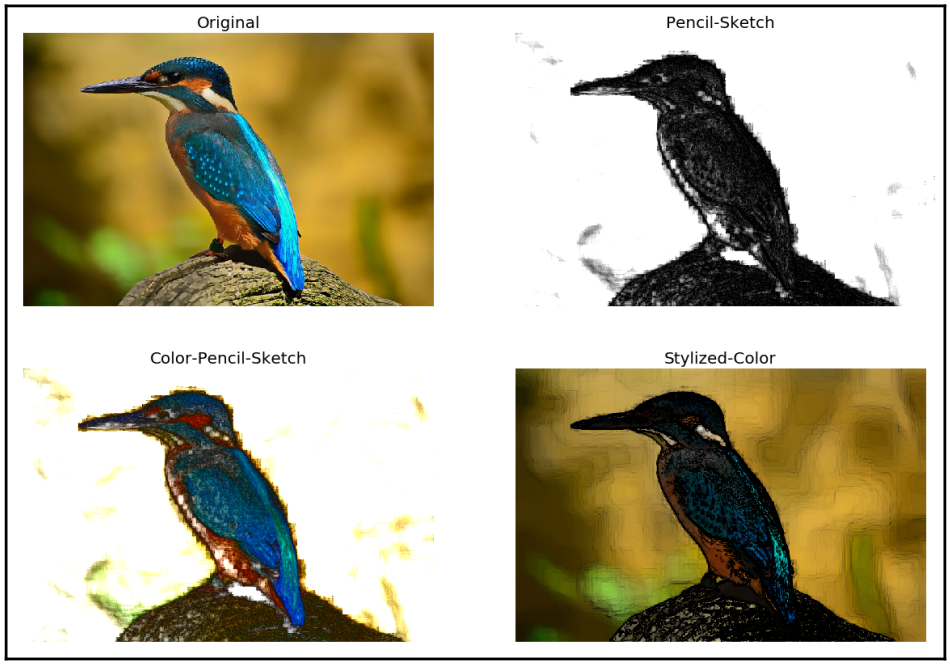
The dodge operation divides (using `np.divide()`) the image by the inverted blurred image. This highlights the boldest edges in the image.

There's more...

There are a few more edge detection techniques, such as Canny (with hysteresis thresholds), that you can try to produce sketches from images. You can try them on your own and compare the sketches obtained using different algorithms. Also, by using OpenCV-Python's `pencilSketch()` and `sylization()` functions, you can produce black and white and color pencil sketches, as well as watercolor-like stylized images, with the following few lines of code:

```
import cv2
import matplotlib.pylab as plt
src = cv2.imread('images/bird.png')
#dst = cv2.detailEnhance(src, sigma_s=10, sigma_r=0.15)
dst_sketch, dst_color_sketch = cv2.pencilSketch(src, sigma_s=50,
sigma_r=0.05, shade_factor=0.05)
dst_water_color = cv2.stylization(src, sigma_s=50, sigma_r=0.05)
```

If you run this code and plot the images, you will get a diagram similar to the following screenshot:



See also

For more details, refer to the following link:

- `https://www.youtube.com/watch?v=Zyl1gAIROxg`

Creating cartoonish images

In this recipe, you will learn how to create cartoonish flat-textured images from an image. Again, there is more than one way to do the same; here, we will learn how to do it with edge-preserving bilateral filters.

Getting ready

The following libraries need to be imported first:

```
import cv2
import numpy as np
import matplotlib.pylab as plt
```

How to do it...

For this recipe, we will be using the `bilateralFilter()` function from OpenCV-Python. We need to start by downsampling the image to create an image pyramid (you will see more of this in the next chapter), followed by repeated application of small bilateral filters (to remove unimportant details) and upsampling the image to its original size. Next, you need to apply the median blur (to flatten the texture) followed by masking the original image with the binary image obtained by adaptive thresholding. The following code demonstrates the steps:

1. Read the input image and initialize the parameters to be used later:

```
img = plt.imread("images/bean.png")

num_down = 2 # number of downsampling steps
num_bilateral = 7 # number of bilateral filtering steps

w, h, _ = img.shape
```

2. Use the Gaussian pyramid's downsampling to reduce the image size (and make the subsequent operations faster):

```
img_color = np.copy(img)
for _ in range(num_down):
 img_color = cv2.pyrDown(img_color)
```

3. Apply bilateral filters (with a small *diameter* value) iteratively. The `d` parameter represents the diameter of the neighborhood for each pixel, where the `sigmaColor` and `sigmaSpace` parameters represent the filter sigma in the color and the coordinate spaces, respectively:

```
for _ in range(num_bilateral):
 img_color = cv2.bilateralFilter(img_color, d=9, sigmaColor=0.1,
sigmaSpace=0.01)
```

4. Use upsampling to enlarge the image to the original size:

```
for _ in range(num_down):
 img_color = cv2.pyrUp(img_color)
```

5. Convert to the output image obtained from the last step and blur the image with the `median` filter:

```
img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
img_blur = cv2.medianBlur(img_gray, 7)
```

6. Detect and enhance the edges:

```
img_edge = cv2.adaptiveThreshold((255*img_blur).astype(np.uint8), \
            255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, \
            blockSize=9, C=2)
```

7. Convert the grayscale edge image back into an RGB color image and compute bitwise `AND` with the RGB color image to get the final output cartoonish image:

```
img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
img_cartoon = cv2.bitwise_and(img_color, img_edge)
```

How it works...

As explained earlier, the `bilateralFilter()`, `medianBlur()`, `adaptiveThreshold()`, and `bitwise_and()` functions from OpenCV-Python were the key functions used to first remove weak edges, then convert into flat texture, and finally enhance the prominent edges in the image.

The `bilateralFilter()` function from OpenCV-Python was used to smooth the textures while keeping the edges fairly sharp:

- The higher the value of the `sigmaColor` parameter, the more the pixel colors in the neighborhood will be mixed together. This will produce larger areas of semi-equal color in the output image.
- The higher the value of the `sigmaSpace` parameter, the more the pixels with similar colors will influence each other.

The image was downsampled to create an image pyramid (you will see more of this in the next chapter).

Next, repeated application of small bilateral filters was used to remove unimportant details. A subsequent upsampling was used to resize the image to its original size.

Finally, `medianBlur` was applied (to flatten the texture) followed by masking the original image with the binary image obtained by adaptive thresholding.

If you run the preceding code, you will get an output cartoonish image, as shown here:

There's more...

Play with the parameter values of the OpenCV functions to see the impact on the output image produced. Also, as mentioned earlier, there is more than one way to achieve the same effect. Try using anisotropic diffusion to obtain flat texture images. You should get an image like the following one (use the `anisotropic_diffusion()` function from the `medpy` library):