OpenCV 3.0 Computer Vision with Java

OpenCV 3.0 Computer Vision with Java is a practical tutorial guide that explains fundamental tasks from computer vision while focusing on Java development. This book will teach you how to set up OpenCV for Java and handle matrices using the basic operations of image processing such as filtering and image transforms. It will also help you learn how to use Haar cascades for tracking faces and to detect foreground and background regions with the help of a Kinect device. It will even give you insights into server-side OpenCV. Each chapter is presented with several projects that are ready to use. The functionality of these projects is found in many classes that allow developers to understand computer vision principles and rapidly extend or customize the projects for their needs.

Who this book is written for
If you are a Java developer, student, researcher, or hobbyist wanting to create computer vision applications in Java then this book is for you. If you are an experienced C/C++ developer who is used to working with OpenCV, you will also find this book very useful for migrating your applications to Java. All you need is basic knowledge of Java, with no prior understanding of computer vision required, as this book will give you clear explanations and examples of the basics.

What you will learn from this book
- Create powerful GUIs for computer vision applications with panels, scroll panes, radio buttons, sliders, windows, and mouse interaction using the popular Swing GUI widget toolkit
- Stretch, shrink, warp, and rotate images, as well as apply image transforms to find edges, lines, and circles, and even use Discrete Fourier Transforms (DFT)
- Detect foreground or background regions and work with depth images with a Kinect device
- Learn how to add computer vision capabilities to rock solid Java web applications allowing you to upload photos and create astonishing effects
- Track faces and apply mixed reality effects such as adding virtual hats to uploaded photos
- Filter noisy images, work with morphological operators, use flood fill, and threshold the important regions of an image
- Open and process video streams from webcams or video files

Daniel Lélis Baggio


$ 39.99 US£ 25.99 UK

Prices do not include local sales tax or VAT

OpenCV 3.0 Computer Vision with Java

Create multiplatform computer vision desktop and web applications using the combination of OpenCV and Java
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Image Transforms'
- A synopsis of the book’s content
- More information on OpenCV 3.0 Computer Vision with Java
Daniel Lélis Baggio started his work in computer vision through medical image processing at Instituto do Coração (InCor), which is a heart institute in São Paulo, Brazil, where he worked with intravascular ultrasound (IVUS) image segmentation. After this, he focused on GPGPU and ported that algorithm to work with NVIDIA’s CUDA. He also dived into the topic of six degrees of freedom (6DoF), head tracking through a project called EHCI (http://code.google.com/p/ehci/) with the Natural User Interface group.

He is also the author of Mastering OpenCV with Practical Computer Vision Projects, Packt Publishing.
Preface

Living in times when self-driving vehicles are becoming a reality might trigger curious minds as to how could computers' incipient vision works. Having a face recognized for access control, getting our pictures automatically organized by a subject or person, and having characters automatically recognized from paper scans are tasks that have become common in our lives. All these aforementioned actions have been enlisted in the so-called study area of computer vision.

As a scientific discipline, the theory behind systems that can extract information from images can be described as computer vision, and it has been adopted to extract valuable measurements from medical images, as well as to help humans delineate the boundaries of important image areas in the so-called semi-automatic procedures.

In the context of providing a simple-to-use computer vision infrastructure to help people rapidly build sophisticated vision applications, an open source library was created: OpenCV. It was designed for real-time applications and is written in C++, containing several hundred computer vision algorithms.

Although OpenCV had its debut alpha release back in January 1999, it was only in February 2013 that it officially supported desktop Java through bindings. As this is one of the most popular introductory teaching languages adopted in computer science departments as well as K-12 computer-related courses, it is important to have a good reference for how to build vision apps in a Java environment.

This book covers the basic OpenCV computer vision algorithms and their integration with Java. As the Swing GUI widget toolkit is widely adopted to build GUIs in Java, in this book, you will benefit from the chapters that deal with this topic as well as come to know how to set up your development environment that deals with native code bindings. Besides, operations such as stretching, shrinking, warping, and rotating, as well as finding edges, lines, and circles are all covered through interesting and practical sample projects in this book.
As the Kinect device has become a great tool for background segmentation, we have covered it in this chapter as well.

Another hot topic that is commonly explored with computer vision is machine learning, and in this book, you will find useful information to create your own object tracker and to use OpenCV’s built-in face tracker as well.

Since Java has been widely used for web applications, we have covered computer vision applications on the server side as well, explaining the details of image uploading and integration with OpenCV.

By the end of this book, you will have a solid background in how to use Java with OpenCV from setup to server side; a brief explanation of the basic computer vision topics are covered in the book. Also, you'll get the source code of several complete projects from which you can extend and add your own functionality.

What this book covers

Chapter 1, Setting Up OpenCV for Java, covers the setting up of a library and development environment. This chapter covers Eclipse and NetBeans IDEs, as well as explaining the Ant and Maven build tools configuration.

Chapter 2, Handling Matrices, Files, Cameras, and GUIs, shows how to access matrices at the pixel level as well as how to load and display images from files and web cameras. It also covers the Swing widget toolkit support and how to work with OpenCV.

Chapter 3, Image Filters and Morphological Operators, deals with the process of removing noise from images as well as morphological operators. It also explains image pyramids and topics such as flood fill and image thresholding.

Chapter 4, Image Transforms, explains important transformations to find edges, such as the Gradient and Sobel filters. Additionally, it also explains line and circle Hough transforms, which are used to identify not only straight but also radial lines. The Discrete Fourier analysis and some distance transforms are also explained in this chapter.

Chapter 5, Object Detection Using Ada Boost and Haar Cascades, demonstrates how to create your own classifier to find some objects, as well as how to use the famous face detection classifier.
Chapter 6, Detecting Foreground and Background Regions and Depth with a Kinect Device, explores the important problem of extracting your background. Furthermore, it explains how to use a Kinect device to retrieve depth information.

Chapter 7, OpenCV on the Server Side, explains how to set up a web server application with OpenCV.
This chapter covers the methods to change an image into an alternate representation of data in order to cover important problems of computer vision and image processing. Some examples of these methods are artifacts that are used to find image edges as well as transforms that help us find lines and circles in an image. In this chapter, we have covered stretch, shrink, warp, and rotate operations. A very useful and famous transform is Fourier, which transforms signals between the time domain and frequency domain. In OpenCV, you can find the Discrete Fourier Transform (DFT) and Discrete Cosine Transform (DCT). Another transform that we’ve covered in this chapter is related to integral images that allow rapid summing of sub regions, which is a very useful step in tracking faces algorithm. Besides this, you will also get to see distance transform and histogram equalization in this chapter.

we will cover the following topics:

- Gradients and sobel derivatives
- The Laplace and canny transforms
- The line and circle Hough transforms
- Geometric transforms: stretch, shrink, warp, and rotate
- Discrete Fourier Transform (DFT) and Discrete Cosine Transform (DCT)
- Integral images
- Distance transforms
- Histogram equalization

By the end of this chapter, you will have learned a handful of transforms that will enable you to find edges, lines, and circles in images. Besides, you will be able to stretch, shrink, warp, and rotate images as well as you will be able to change the domain from the spatial domain to the frequency domain. Other important transforms used for face tracking will be covered in this chapter as well. Finally, distance transforms and histogram equalization will also be explored in detail.
The Gradient and Sobel derivatives

A key building block in computer vision is finding edges and this is closely related to finding an approximation to derivatives in an image. From basic calculus, it is known that a derivative shows the variation of a given function or an input signal with some dimension. When we find the local maximum of the derivative, this will yield regions where the signal varies the most, which for an image might mean an edge. Hopefully, there's an easy way to approximate a derivative for discrete signals through a kernel convolution. A convolution basically means applying some transforms to every part of the image. The most used transform for differentiation is the Sobel filter [1], which works for horizontal, vertical, and even mixed partial derivatives of any order.

In order to approximate the value for the horizontal derivative, the following sobel kernel matrix is convoluted with an input image:

\[
G_x = \begin{bmatrix} 
-1 & 0 & +1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 
\end{bmatrix} \ast T_x
\]

This means that, for each input pixel, the calculated value of its upper-right neighbor plus twice its right neighbor, plus its bottom-right neighbor, minus its upper-left neighbor, minus its left neighbor, minus its left-bottom neighbor will be calculated, yielding a resulting image. In order to use this operator in OpenCV, you can call Imgproc's Sobel function according to the following signature:

```java
public static void Sobel(Mat src, Mat dst, int ddepth, int dx, int dy)
```

The `src` parameter is the input image and `dst` is the output. `Ddepth` is the output image's depth and when this is assigned as -1, this has the same depth as the source. The `dx` and `dy` parameters will inform us about the order in each of these directions. When setting `dy` to 0 and `dx` to 1, the kernel that we've used is the one mentioned in the preceding matrix. The example project `kernels` from this chapter shows a customizable look of these operators, as shown in the following screenshot:
The Laplace and Canny transforms

Another quite useful operator to find edges is the Laplacian transformation. Instead of relying on the first order derivatives, OpenCV's Laplacian transformation implements the discrete operator for the following function:

\[
\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}
\]
The matrix can be approximated to the convolution with the following kernel when using finite difference methods and a 3x3 aperture:

\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]

The signature for the preceding function is as follows:

\[
\text{Laplacian(Mat source, Mat destination, int ddepth)}
\]

While source and destination matrices are simple parameters, \text{ddepth} is the depth of the destination matrix. When you set this parameter to -1, it will have the same depth as the source image, although you might want more depth when you apply this operator. Besides this, there are overloaded versions of this method that receive an aperture size, a scale factor, and an adding scalar.

Besides using the Laplacian method, you can also use the Canny algorithm, which is an excellent approach that was proposed by computer scientist John F. Canny, who optimized edge detection for low error rate, single identification, and correct localization. In order to fulfill it, the Canny algorithm applies a Gaussian to filter the noise, calculates intensity gradients through sobel, suppresses spurious responses, and applies double thresholds followed by a hysteresis that suppresses the weak and unconnected edges. For more information, check this paper [2]. The method's signature is as follows:

\[
\text{Canny(Mat image, Mat edges, double threshold1, double threshold2, int apertureSize, boolean L2gradient)}
\]

The \text{image} parameter is the input matrix, \text{edges} is the output image, \text{threshold1} is the first threshold for the hysteresis procedure (values smaller than this will be ignored), and \text{threshold2} is the high threshold for hysteresis (values higher than this will be considered as strong edges, while the smaller values and the ones higher than the low threshold will be checked for connection with strong edges). The aperture size is used for the Sobel operator when calculating the gradient and the \text{boolean} informs us which norm to use for the gradient. You can also check out the source code to use this operator in the kernel's project sample in this chapter.
The line and circle Hough transforms

In case you need to find straight lines or circles in an image, you can use Hough transforms, as they are very useful. In this section, we will cover OpenCV methods to extract them from your image.

The idea behind the original Hough line transform is that any point in a binary image could be part of a set of lines. Suppose each straight line could be parameterized by the \( y = mx + b \) line equation, where \( m \) is the line slope and \( b \) is the \( y \) axis intercept of this line. Now, we could iterate the whole binary image, storing each of the \( m \) and \( b \) parameters and checking their accumulation. The local maximum points of the \( m \) and \( b \) parameters would yield equations of straight lines that mostly appeared in the image. Actually, instead of using the slope and \( y \) axis interception point, we use the polar straight line representation.

Since OpenCV not only supports the standard Hough transform, but also the progressive probabilistic Hough transform for which the two functions are Imgproc.HoughLines and Imgproc.HoughLinesP, respectively. For detailed information, refer to [3]. These functions' signatures are explained as follows:

```java
HoughLines(Mat image, Mat lines, double rho, double theta, int threshold)
HoughLinesP(Mat image, Mat lines, double rho, double theta, int threshold)
```

The hough project from this chapter shows an example of the usage of them. The following is the code to retrieve lines from Imgproc.HoughLines:

```java
Mat canny = new Mat();
Imgproc.Canny(originalImage, canny, 10, 50, aperture, false);
image = originalImage.clone();
Mat lines = new Mat();
Imgproc.HoughLines(canny, lines, 1, Math.PI/180, lowThreshold);
```

Note that we need to apply the Hough transform over an edge image; therefore, the first two lines of the preceding code will take care of this. Then, the original image is cloned for display and a Mat object is created in the fourth line in order to keep the lines. In the last line, we can see the application of HoughLines.
The third parameter in `Imgproc.HoughLines` refers to the distance resolution of the accumulator in pixels, while the fourth parameter is the angle resolution of the accumulator in radians. The fifth parameter is the accumulator threshold, which means that only the lines with more than the specified amount of votes will be returned. The `lowThreshold` variable is tied to the scale slider in the example application for the user to experiment with it. It is important to observe that the lines are returned in the `lines` matrix, which has two columns in which each line returns the `rho` and `theta` parameters of the polar coordinates. These coordinates refer to the distance between the top-left corner of the image and the line rotation in radians, respectively. Following this example, you will find out how to draw the lines from the returned matrix. You can see the working of the Hough transform in the following screenshot:
Besides having the standard Hough transform, OpenCV also offers a probabilistic Hough line transform as well as a circular version. Both the implementations are explored in the same Hough sample project, and the following screenshot shows the working of the circular version:

Geometric transforms – stretch, shrink, warp, and rotate

While working with images and computer vision, it is very common that you will require the ability to preprocess an image using known geometric transforms, such as stretching, shrinking, rotation, and warping. The latter is the same as nonuniform resizing. These transforms can be realized through the multiplication of source points with a 2 x 3 matrix and they get the name of **affine transformations** while turning rectangles in parallelograms. Hence, they have the limitation of requiring the destination to have parallel sides. On the other hand, a 3 x 3 matrix multiplication represents perspective transforms. They offer more flexibility since they can map a 2D quadrilateral to another. The following screenshot shows a very useful application of this concept.
Here, we will find out which is the perspective transform that maps the side of a building in a perspective view to its frontal view:

Note that the input to this problem is the perspective photograph of the building, which is seen on the left-hand side of the preceding image, as well as the four corner points of the highlighted quadrilateral shape. The output is to the right and shows what a viewer would see if he/she looks at the side of the building.

Since affine transforms are a subset of perspective transformations, we will focus on the latter ones here. The code available for this example is in the warps project of this chapter. The main method used here is warpPerspective from Imgproc. It applies a perspective transformation to an input image. Here is the method signature for the warpPerspective method:

```java
public static void warpPerspective(Mat src, Mat dst, Mat M, Size dsize)
```

The `Mat src` parameter is, naturally, the input image, which is the left-hand side image in the preceding screenshot, while `dst Mat` is the image on the right-hand side; make sure you initialize this parameter before using the method. The not-so-straightforward parameter here is `Mat M`, which is the warping matrix. In order to calculate it, you can use the `getPerspectiveTransform` method from Imgproc as well. This method will calculate the perspective matrix from two sets of the four correlated 2D points, the source and destination points. In our example, the source points are the ones that are highlighted on the left-hand side of the screenshot, while the destination points are the four corner points of the image to the right. These points can be stored through the `MatOfPoint2f` class, which stores the `Point` objects. The `getPerspectiveTransform` method’s signature is as follows:

```java
public static Mat getPerspectiveTransform(Mat src, Mat dst)
```
Mat src and Mat dst are the same as the MatOfPoint2f class mentioned previously, which is a subclass of Mat.

In our example, we added a mouse listener to retrieve points clicked by the user. A detail to be kept in mind is that these points are stored in the order: top-left, top-right, bottom-left, and bottom-right. In the example application, the currently modified point can be chosen through four radio buttons above the images. The act of clicking and dragging listeners has been added to the code, so both approaches work.

**Discrete Fourier Transform and Discrete Cosine Transform**

When dealing with image analysis, it would be very useful if you could change an image from the spatial domain, which is the image in terms of its x and y coordinates, to the frequency domain—the image decomposed in its high and low frequency components—so that you would be able to see and manipulate frequency parameters. This could come in handy in image compression because it is known that human vision is not much sensitive to high frequency signals as it is to low frequency signals. In this way, you could transform an image from the spatial domain to the frequency domain and remove high frequency components, reducing the required memory to represent the image and hence compressing it. An image frequency can be pictured in a better way by the next image.

In order to change an image from the spatial domain to the frequency domain, the Discrete Fourier Transform can be used. As we might need to bring it back from the frequency domain to the spatial domain, another transform, which is the Inverse Discrete Fourier Transform, can be applied.

The formal definition of DFT is as follows:

\[
F(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i,j) e^{-i2\pi \left(\frac{ki}{N}, \frac{lj}{N}\right)}
\]

The \( f(i,j) \) value is the image in the spatial domain and \( F(k,l) \) is the image in the frequency domain. Note that \( F(k,l) \) is a complex function, which means that it has a real and an imaginary part. This way, it will be represented by two OpenCV Mat objects or by Mat with two channels. The easiest way to analyze a DFT is by plotting its magnitude and taking its logarithm, since values for the DFT can be in different orders of magnitude.
For instance, this is a pulse pattern, which is a signal that can come from zero, represented as black, to the top, represented as white, on its left, and its Fourier transform magnitude with the applied logarithm to its right:

![Fourier Example](image)

Looking back at the preceding DFT transform, we can think of $F(k, l)$ as the value that would be yielded by multiplying each point of the spatial image with a base function, which is related to the frequency domain, and by summing the products. Remember that base functions are sinusoidal and they have increasing frequencies. This way, if some of the base functions oscillate at the same rate as the signal, it will be able to sum up to a big number, which will be seen as a white dot on the Fourier Transform image. On the other hand, if the given frequency is not present in the image, the oscillation and multiplication with the image will result in a small number, which won't be noticed in the Fourier Transform image.

Another thing to observe from the equation is that $F(0, 0)$ will yield a base function that is always 1. This way, $F(0, 0)$ will simply refer to the sum of all the pixels of the spatial image. We can also check whether $F(N-1, N-1)$ corresponds to the base function related to the highest frequency in the image. Note that the previous image basically has a DC component, which would be the image mean and it could be checked from the white dot in the middle of the Discrete Fourier transform image. Besides, the image to the left could be seen as a series of pulses and hence it would have a frequency in the $x$ axis, which can be noticed by the two dots near the central point in the Fourier Transform image to the right. Nonetheless, we will need to use multiple frequencies to approximate the pulse shape. In this way, more dots can be seen in the $x$-axis of the image to the right. The following screenshot gives more insight and helps you understand the Fourier analysis:
Now, we will again check the DC level at the center of the DFT image, to the right, as a bright central dot. Besides, we can also check multiple frequencies in a diagonal pattern. An important piece of information that can be retrieved is the direction of spatial variation, which is clearly seen as bright dots in the DFT image.

It is time to work on some code now. The following code shows you how to make room to apply the DFT. Remember, from the preceding screenshot, that the result of a DFT is complex. Besides, we need them stored as floating point values. This way, we first convert our 3-channel image to gray and then to a float. After this, we put the converted image and an empty Mat object into a list of mats, combining them into a single Mat object through the use of the Core.merge function, shown as follows:

```java
Mat gray = new Mat();
Imgproc.cvtColor(originalImage, gray, Imgproc.COLOR_RGB2GRAY);
Mat floatGray = new Mat();
gray.convertTo(floatGray, CvType.CV_32FC1);

List<Mat> matList = new ArrayList<Mat>();
matList.add(floatGray);
Mat zeroMat = Mat.zeros(floatGray.size(), CvType.CV_32F);
matList.add(zeroMat);
Mat complexImage = new Mat();
Core.merge(matList, complexImage);
```

Now, it's easy to apply an in-place Discrete Fourier Transform:

```java
Core.dft(complexImage, complexImage);
```
In order to get some meaningful information, we will print the image, but first, we have to obtain its magnitude. In order to get it, we will use the standard way that we learned in school, which is getting the square root of the sum of the squares of the real and complex parts of numbers.

Again, OpenCV has a function for this, which is `Core.magnitude`, whose signature is `magnitude(Mat x, Mat y, Mat magnitude)`, as shown in the following code:

```java
List<Mat> splitted = new ArrayList<Mat>();
Core.split(complexImage,splitted);
Mat magnitude = new Mat();
Core.magnitude(splitted.get(0), splitted.get(1), magnitude);
```

Before using `Core.magnitude`, just pay attention to the process of unpacking a DFT in the splitted mats using `Core.split`.

Since the values can be in different orders of magnitude, it is important to get the values in a logarithmic scale. Before doing this, it is important to add 1 to all the values in the matrix just to make sure we won't get negative values when applying the `log` function. Besides this, there's already an OpenCV function to deal with logarithms, which is `Core.log`:

```java
Core.add(Mat.ones(magnitude.size(), CvType.CV_32F), magnitude, magnitude);
Core.log(magnitude, magnitude);
```

Now, it is time to shift the image to the center, so that it's easier to analyze its spectrum. The code to do this is simple and goes like this:

```java
int cx = magnitude.cols()/2;
int cy = magnitude.rows()/2;
Mat q0 = new Mat(magnitude,new Rect(0, 0, cx, cy));
Mat q1 = new Mat(magnitude,new Rect(cx, 0, cx, cy));
Mat q2 = new Mat(magnitude,new Rect(0, cy, cx, cy));
Mat q3 = new Mat(magnitude ,new Rect(cx, cy, cx, cy));
Mat tmp = new Mat();
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);
q1.copyTo(tmp);
q2.copyTo(q1);
tmp.copyTo(q2);
```
As a last step, it's important to normalize the image, so that it can be seen in a better way. Before we normalize it, it should be converted to CV_8UC1:

```java
magnitude.convertTo(magnitude, CvType.CV_8UC1);
Core.normalize(magnitude, magnitude, 0, 255, Core.NORM_MINMAX, CvType.CV_8UC1);
```

When using the DFT, it's often enough to calculate only half of the DFT when you deal with real-valued data, as is the case with images. This way, an analog concept called the Discrete Cosine Transform can be used. In case you want it, it can be invoked through Core.dct.

### Integral images

Some face recognition algorithms, such as OpenCV's face detection algorithm make heavy use of features like the ones shown in the following image:

1. Edge features
   - (a) (b) (c) (d)

2. Line features
   - (a) (b) (c) (d) (e) (f) (g) (h)

3. Center-surround features
   - (a) (b)
These are the so-called Haar-like features and they are calculated as the sum of pixels in the white area minus the sum of pixels in the black area. You might find this type of a feature kind of odd, but when training it for face detection, it can be built to be an extremely powerful classifier using only two of these features, as depicted in the following image:

In fact, a classifier that uses only the two preceding features can be adjusted to detect 100 percent of a given face training database with only 40 percent of false positives. Taking out the sum of all pixels in an image as well as calculating the sum of each area can be a long process. However, this process must be tested for each frame in a given input image, hence calculating these features fast is a requirement that we need to fulfill.

First, let's define an integral image sum as the following expression:

\[
\text{sum}(X, Y) = \sum_{x < X} \sum_{y < Y} \text{image}(x, y)
\]

For instance, if the following matrix represents our image:

\[
A = \begin{bmatrix}
0 & 2 & 4 \\
6 & 8 & 10 \\
12 & 14 & 16
\end{bmatrix}
\]
Chapter 4

An integral image would be like the following:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 2 & 6 \\
0 & 6 & 16 & 30 \\
0 & 18 & 42 & 72 \\
\end{bmatrix}
\]

The trick here follows from the following property:

\[
\sum_{x1y1\leq x\leq x2, y\leq y2} image(x,y) = \sum(x2, y2) - \sum(x1-1, y2) - \sum(x2, y1-1) + \sum(x1-1, y1-1)
\]

This means that in order to find the sum of a given rectangle bounded by the points \((x1, y1), (x2, y1), (x2, y2),\) and \((x1, y2),\) you just need to use the integral image at the point \((x2, y2),\) but you also need to subtract the points \((x1-1, y2)\) from \((x2, y1-1).\) Also, since the integral image at \((x1-1, y1-1)\) has been subtracted twice, we just need to add it once.

The following code will generate the preceding matrix and make use of Imgproc.integral to create the integral images:

```java
Mat image = new Mat(3,3 ,CvType.CV_8UC1);
Mat sum = new Mat();
byte[] buffer = {0,2,4,6,8,10,12,14,16};
image.put(0,0,buffer);
System.out.println(image.dump());
Imgproc.integral(image, sum);
System.out.println(sum.dump());
```

The output of this program is like the one shown in the preceding matrices for A and Sum A.

It is important to verify that the output is a 4 x 4 matrix because of the initial row and column of zeroes, which are used to make the computation efficient.
Distance transforms

Simply put, a distance transform applied to an image will generate an output image whose pixel values will be the closest distance to a zero-valued pixel in the input image. Basically, they will have the closest distance to the background, given a specified distance measure. The following screenshot gives you an idea of what happens to the silhouette of a human body:

![Human silhouette by J E Theriot](image)

This transform can be very useful in the process of getting the topological skeleton of a given segmented image as well as to produce blurring effects. Another interesting application of this transform is in the segmentation of overlapping objects, along with a watershed.

Generally, the distance transform is applied to an edge image, which results from a Canny filter. We are going to make use of Imgproc's distanceTransform method, which can be seen in action in the distance project, which you can find in this chapter's source code. Here are the most important lines of this example program:

```java
protected void processOperation() {
    Imgproc.Canny(originalImage, image, 220, 255, 3, false);
}
Firstly, a Canny edge detector filter is applied to the input image. Then, a threshold with THRESH_BINARY_INV converts the edges to black and beans to white. Only then, the distance transform is applied. The first argument is the input image, the second one is the output matrix, and the third argument specifies how distances are calculated. In our example, CV_DIST_L2 means Euclidean, while other distances, such as CV_DIST_L1 or CV_DIST_L12, among others exist. Since the output of distanceTransform is a single channel 32 bit Float image, a conversion is required. Finally, we apply Core.multiply to increase the contrast.

The following screenshot gives you a good idea of the whole process:

![Distance Transform Example](image)

**Histogram equalization**

The human visual system is very sensitive to contrast in images, which is the difference in the color and brightness of different objects. Besides, the human eye is a miraculous system that can feel intensities at the $10^{16}$ light levels [4]. No wonder some sensors could mess up the image data.
When analyzing images, it is very useful to draw their histograms. They simply show you the lightness distribution of a digital image. In order to do that, you need to count the number of pixels with the exact lightness and plot that as a distribution graph. This gives us a great insight into the dynamic range of an image.

When a camera picture has been captured with a very narrow light range, it gets difficult to see the details in the shadowed areas or other areas with poor local contrast. Fortunately, there's a technique to spread frequencies for uniform intensity distribution, which is called **histogram equalization**. The following image shows the same picture with their respective histograms before and after the histogram equalization technique is applied:
Note that the light values, located at the rightmost part of the upper histogram, are rarely used, while the middle range values are too tied. Spreading the values along the full range yields better contrast and details can be more easily perceived by this. The histogram equalized image makes better use of intensities that generate better contrast. In order to accomplish this task, a cumulative distribution can be used to remap the histogram to something that resembles a uniform distribution. Then, it's just a matter of checking where the points from the original histogram would be mapped to the uniform distribution through the use of a cumulative Gaussian distribution, for instance.

Now, the good part is that all these details have been wrapped in a simple call to OpenCV's `equalizeHist` function. Here is the sample from the histogram project in this chapter:

```java
protected void processOperation() {
    Imgproc.cvtColor(originalImage, grayImage, Imgproc.COLOR_RGB2GRAY);
    Imgproc.equalizeHist(grayImage, image);
    updateView();
}
```

This piece of code simply converts the image to a single channel image; however, you can use `equalizeHist` on a color image as long as you treat each channel separately. The `Imgproc.equalizeHist` method outputs the corrected image following the previously mentioned concept.

**References**


Summary
This chapter covered the key aspects of computer vision's daily use. We started with the important edge detectors, where you gained the experience of how to find them through the Sobel, Laplacian, and Canny edge detectors. Then, we saw how to use the Hough transforms to find straight lines and circles. After that, the geometric transforms stretch, shrink, warp, and rotate were explored with an interactive sample. We then explored how to transform images from the spatial domain to the frequency domain using the Discrete Fourier analysis. After that, we showed you a trick to calculate Haar-like features fast in an image through the use of integral images. We then explored the important distance transforms and finished the chapter by explaining histogram equalization to you.

Now, be ready to dive into machine learning algorithms, as we will cover how to detect faces in the next chapter. Also, you will learn how to create your own object detector and understand how supervised learning works in order to better train your classification trees.
Where to buy this book

You can buy OpenCV 3.0 Computer Vision with Java from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. 

Click here for ordering and shipping details.