Learning Image Processing with OpenCV

OpenCV, arguably the most widely used computer vision library, includes hundreds of ready-to-use imaging and vision functions and is used in both academia and enterprises.

This book provides an example-based tour of OpenCV’s main image processing algorithms. Starting with an exploration of library installation, wherein the library structure and basics of image and video reading/writing are covered, you will dive into image filtering and the color manipulation features of OpenCV with LUTs. You’ll then be introduced to techniques such as inpainting and denoising to enhance images as well as the process of HDR imaging. Finally, you’ll master GPU-based accelerations. By the end of this book, you will be able to create smart and powerful image processing applications with ease! All the topics are described with short, easy-to-follow examples.

Who this book is written for
If you are a competent C++ programmer and want to learn the tricks of image processing with OpenCV, then this book is for you. A basic understanding of image processing is required.

What you will learn from this book
- Create OpenCV programs with rich user interfaces
- Grasp basic concepts and tasks in image processing such as image types, pixel access techniques, and arithmetic operations with images and histograms
- Explore useful image processing techniques such as filtering, smoothing, sharpening, denoising, morphology, and geometrical transformations
- Get to know handy algorithms such as inpainting and LUTs
- Leverage the color manipulation features of OpenCV to optimize image processing
- Discover how to process a video and the main techniques involved such as stabilization, stitching, and even superresolution
- Understand the new computational photography module that covers high-dynamic range imaging, seamless cloning, decolorization, and non-photorealistic rendering

Exploit the amazing features of OpenCV to create powerful image processing applications through easy-to-follow examples.
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 6 'Computational Photography'
- A synopsis of the book’s content
- More information on Learning Image Processing with OpenCV

About the Authors

Gloria Bueno García holds a PhD in machine vision from Coventry University, UK. She has experience working as the principal researcher in several research centers, such as UMR 7005 research unit CNRS/ Louis Pasteur Univ. Strasbourg (France), Gilbert Gilkes & Gordon Technology (UK), and CEIT San Sebastian (Spain). She is the author of two patents, one registered type of software, and more than 100 refereed papers. Her interests are in 2D/3D multimodality image processing and artificial intelligence. She leads the VISILAB research group at the University of Castilla-La Mancha. She has coauthored a book on OpenCV programming for mobile devices: OpenCV essentials, Packt Publishing.

Oscar Deniz Suarez's research interests are mainly focused on computer vision and pattern recognition. He is the author of more than 50 refereed papers in journals and conferences. He received the runner-up award for the best PhD work on computer vision and pattern recognition by AERFAI and the Image File and Reformatting Software Challenge Award by Innocentive Inc. He has been a national finalist for the 2009 Cor Baayen award. His work is used by cutting-edge companies, such as Existor, Gliif, Tapmedia, E-Twenty, and others, and has also been added to OpenCV. Currently, he works as an associate professor at the University of Castilla-La Mancha and contributes to VISILAB. He is a senior member of IEEE and is affiliated with AAAI, SIANI, CEA-IFAC, AEPIA, and AERFAI-IAPR. He serves as an academic editor of the PLoS ONE journal. He has been a visiting researcher at Carnegie Mellon University, Imperial College London, and Leica Biosystems. He has coauthored two books on OpenCV previously.
José Luis Espinosa Aranda holds a PhD in computer science from the University of Castilla-La Mancha. He has been a finalist for Certamen Universitario Arquímedes de Introducción a la Investigación científica in 2009 for his final degree project in Spain. His research interests involve computer vision, heuristic algorithms, and operational research. He is currently working at the VISILAB group as an assistant researcher and developer in computer vision topics.

Jesus Salido Tercero gained his electrical engineering degree and PhD (1996) from Universidad Politécnica de Madrid (Spain). He then spent 2 years (1997 and 1998) as a visiting scholar at the Robotics Institute (Carnegie Mellon University, Pittsburgh, USA), working on cooperative multirobot systems. Since his return to the Spanish University of Castilla-La Mancha, he spends his time teaching courses on robotics and industrial informatics, along with research on vision and intelligent systems. Over the last 3 years, his efforts have been directed to develop vision applications on mobile devices. He has coauthored a book on OpenCV programming for mobile devices.

Ismael Serrano Gracia received his degree in computer science in 2012 from the University of Castilla-La Mancha. He got the highest marks for his final degree project on person detection. This application uses depth cameras with OpenCV libraries. Currently, he is a PhD candidate at the same university, holding a research grant from the Spanish Ministry of Science and Research. He is also working at the VISILAB group as an assistant researcher and developer on different computer vision topics.

Noelia Vállez Enano has liked computers since her childhood, though she didn't have one before her mid-teens. In 2009, she finished her studies in computer science at the University of Castilla-La Mancha, where she graduated with top honors. She started working at the VISILAB group through a project on mammography CAD systems and electronic health records. Since then, she has obtained a master's degree in physics and mathematics and has enrolled for a PhD degree. Her work involves using image processing and pattern recognition methods. She also likes teaching and working in other areas of artificial intelligence.
Learning Image Processing with OpenCV

OpenCV, arguably the most widely used computer vision library, includes hundreds of ready-to-use imaging and vision functions and is used in both academia and industry. As cameras get cheaper and imaging features grow in demand, the range of applications using OpenCV increases significantly, both for desktop and mobile platforms.

This book provides an example-based tour of OpenCV's main image processing algorithms. While other OpenCV books try to explain the underlying theory or provide large examples of nearly complete applications, This book is aimed at people who want to have an easy-to-understand working example as soon as possible, and possibly develop additional features on top of that.

The book starts with an introductory chapter in which the library installation is explained, the structure of the library is described, and basic image and video reading and writing examples are given. From this, the following functionalities are covered: handling of images and videos, basic image processing tools, correcting and enhancing images, color, video processing, and computational photography. Last but not least, advanced features such as GPU-based accelerations are also considered in the final chapter. New functions and techniques in the latest major release, OpenCV 3, are explained throughout.

What This Book Covers

Chapter 1, Handling Image and Video Files, shows you how to read image and video files. It also shows basic user-interaction tools, which are very useful in image processing to change a parameter value, select regions of interest, and so on.

Chapter 2, Establishing Image Processing Tools, describes the main data structures and basic procedures needed in subsequent chapters.

Chapter 3, Correcting and Enhancing Images, deals with transformations typically used to correct image defects. This chapter covers filtering, point transformations using Look Up Tables, geometrical transformations, and algorithms for inpainting and denoising images.

Chapter 4, Processing Color, deals with color topics in image processing. This chapter explains how to use different color spaces and perform color transfers between two images.
Chapter 5, *Image Processing for Video*, covers techniques that use a video or a sequence of images. This chapter is focused on algorithms' implementation for video stabilization, superresolution, and stitching.

Chapter 6, *Computational Photography*, explains how to read HDR images and perform tone mapping on them.

Chapter 7, *Accelerating Image Processing*, covers an important topic in image processing: speed. Modern GPUs are the best available technology to accelerate time-consuming image processing tasks.
Computational photography refers to techniques that allow you to extend the typical capabilities of digital photography. This may include hardware add-ons or modifications, but it mostly refers to software-based techniques. These techniques may produce output images that cannot be obtained with a "traditional" digital camera. This chapter introduces some of the lesser-known techniques available in OpenCV for computational photography: high-dynamic-range imaging, seamless cloning, decolorization, and non-photorealistic rendering. These three are inside the photo module of the library. Note that other techniques inside this module (inpainting and denoising) have been already considered in previous chapters.

**High-dynamic-range images**

The typical images we process have 8 bits per pixel (bpp). Color images also use 8 bits to represent the value of each channel, that is, red, green, and blue. This means that only 256 different intensity values are used. This 8 bpp limit has prevailed throughout the history of digital imaging. However, it is obvious that light in nature does not have only 256 different levels. We should, therefore, consider whether this discretization is desirable or even sufficient. The human eye, for example, is known to capture a much higher dynamic range (the number of light levels between the dimmest and brightest levels), estimated at between 1 and 100 million light levels. With only 256 light levels, there are cases where bright lights appear overexposed or saturated, while dark scenes are simply captured as black.
There are cameras that can capture more than 8 bpp. However, the most common way to create high-dynamic-range images is to use an 8 bpp camera and take images with different exposure values. When we do this, problems of a limited dynamic range are evident. Consider, for example, the following figure:

![A scene captured with six different exposure values](image)

The top-left image is mostly black, but window details are visible. Conversely, the bottom-right image shows details of the room, but the window details are barely visible.

We can take pictures with different exposure levels using modern smartphone cameras. With iPhone and iPads, for example, as of iOS 8, it is very easy to change the exposure with the native camera app. By touching the screen, a yellow box appears with a small sun on its side. Swiping up or down can then change the exposure (see the following screenshot).

The range of exposure levels is quite large, so we may have to repeat the swiping gesture a number of times.

If you use previous versions of iOS, you can download camera apps such as Camera+ that allow you to focus on a specific point and change exposure.
For Android, tons of camera apps are available on Google Play that can adjust the exposure. One example is *Camera FV-5*, which has both free and paid versions.

If you use a handheld device to capture the images, make sure the device is static. In fact, you may well use a tripod. Otherwise, images with different exposures will not be aligned. Also, moving subjects will inevitably produce ghost artifacts. Three images are sufficient for most cases, with low, medium, and high exposure levels.

Smartphones and tables are handy to capture a number of images with different exposures. To create HDR images, we need to know the exposure (or shutter) time for each captured image (see the following section for the reason). Not all apps allow you to control (or even see) this manually (the iOS 8 native app doesn't). At the time of writing this, at least two free apps allow this for iOS: *Manually* and *ManualShot!*

In Android, the free *Camera FV-5* allows you to control and see exposure times. Note that F/Stop and ISO are two other parameters that control the exposure.
Images that are captured can be transferred to the development computer and used to create the HDR image.

As of iOS 7, the native camera app has an HDR mode that automatically captures three images in a rapid sequence, each with different exposure. These images are also automatically combined into a single (sometimes better) image.

**Creating HDR images**

How do we combine multiple (three, for example) exposure images into an HDR image? If we consider only one of the channels and a given pixel, the three pixel values (one for each exposure level) must be mapped to a single value in the larger output range (say, 16 bpp). This mapping is not easy. First of all, we have to consider that pixel intensities are a (rough) measure of sensor irradiance (the amount of light incident on the camera sensor). Digital cameras measure irradiance but in a nonlinear way. Cameras have a nonlinear response function that translates irradiance to pixel intensity values in the range of 0 to 255. In order to map these values to a larger set of discrete values, we must estimate the camera response function (that is, the response within the 0 to 255 range).

How do we estimate the camera response function? We do that from the pixels themselves! The response function is an S-shaped curve for each color channel, and it can be estimated from the pixels (with three exposures of a pixel, we have three points on the curve for each color channel). As this is very time consuming, usually, a set of random pixels is chosen.

There's only one thing left. We previously talked about estimating the relationship between irradiance and pixel intensity. How do we know irradiance? Sensor irradiance is directly proportional to the exposure time (or equivalently, the shutter speed). This is the reason why we need exposure time!

Finally, the HDR image is computed as a weighted sum of the recovered irradiance values from the pixels of each exposure. Note that this image cannot be displayed on conventional screens, which also have a limited range.

A good book on high-dynamic-range imaging is *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting* by Reinhard et al, Morgan Kaufmann Pub. The book is accompanied by a DVD containing images in different HDR formats.
Example

OpenCV (as of 3.0 only) provides functions to create HDR images from a set of images taken with different exposures. There's even a tutorial example called `hdr_imaging`, which reads a list of image files and exposure times (from a text file) and creates the HDR image.

In order to run the `hdr_imaging` tutorial, you will need to download the required image files and text files with the list. You can download them from [https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/hdr](https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/hdr).

The CalibrateDebevec and MergeDebevec classes implementDebevec's method to estimate the camera response function and merge the exposures into an HDR image, respectively. The following `createHDR` example shows you how to use both classes:

```cpp
#include <opencv2/photo.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int, char** argv)
{
    vector<Mat> images;
    vector<float> times;

    // Load images and exposures...
    Mat img1 = imread("1div66.jpg");
    if (img1.empty())
    {
        cout << "Error! Input image cannot be read...\n";
        return -1;
    }
    Mat img2 = imread("1div32.jpg");
    Mat img3 = imread("1div12.jpg");
    images.push_back(img1);
    images.push_back(img2);
    images.push_back(img3);
    times.push_back((float)1/66);
```
times.push_back((float)1/32);
times.push_back((float)1/12);

// Estimate camera response...
Mat response;
Ptr<CalibrateDebevec> calibrate = createCalibrateDebevec();
calibrate->process(images, response, times);

// Show the estimated camera response function...
cout << response;

// Create and write the HDR image...
Mat hdr;
Ptr<MergeDebevec> merge_debevec = createMergeDebevec();
merge_debevec->process(images, hdr, times, response);
imwrite("hdr.hdr", hdr);

cout << "\nDone. Press any key to exit...\n";
waitKey(); // Wait for key press
return 0;
}

The example uses three images of a cup (the images are available along with the code accompanying this book). The images were taken with the ManualShot! app mentioned previously, using exposures of 1/66, 1/32, and 1/12 seconds; refer to the following figure:

The three images used in the example as inputs
Note that the `createCalibrateDebevec` method expects the images and exposure times in an STL vector (STL is a kind of library of useful common functions and data structures available in standard C++). The camera response function is given as a 256 real-valued vector. This represents the mapping between the pixel value and irradiance. Actually, it is a 256 x 3 matrix (one column per each of the three color channels). The following figure shows you the response given by the example:

![The estimated RGB camera response functions](image)

The `cout` part of code displays the matrix in the format used by MATLAB and Octave, two widely used packages for numerical computation. It is straightforward to copy the matrix in the output and paste it in MATLAB/Octave in order to display it.

The resulting HDR image is stored in the lossless RGBE format. This image format uses one byte per color channel plus one byte as a shared exponent. The format uses the same principle as the one used in the floating-point number representation: the shared exponent allows you to represent a much wider range of values. RGBE images use the `.hdr` extension. Note that as it is a lossless image format, `.hdr` files are relatively large. In this example, the RGB input images are 1224 x 1632 each (100 to 200 KB each), while the output `.hdr` file occupies 5.9 MB.

The example uses Debevec and Malik's method, but OpenCV also provides another calibration function based on Robertson's method. Both calibration and merge functions are available, that is, `createCalibrateRobertson` and `MergeRobertson`.

---

[171]
For more information on the other functions and the theory behind them, refer to [http://docs.opencv.org/trunk/modules/photo/doc/hdr_imaging.html](http://docs.opencv.org/trunk/modules/photo/doc/hdr_imaging.html).

Finally, note that the example does not display the resulting image. The HDR image cannot be displayed in conventional screens, so we need to perform another step called tone mapping.

**Tone mapping**

When high-dynamic-range images are to be displayed, information can be lost. This is due to the fact that computer screens also have a limited contrast ratio, and printed material is also typically limited to 256 tones. When we have a high-dynamic-range image, it is necessary to map the intensities to a limited set of values. This is called tone mapping.

Simply scaling the HDR image values to the reduced range of the display device is not sufficient in order to provide a realistic output. Scaling typically produces images that appear as lacking detail (contrast), eliminating the original scene content. Ultimately, tone-mapping algorithms aim at providing outputs that appear visually similar to the original scene (that is, similar to what a human would see when viewing the scene). Various tone-mapping algorithms have been proposed and it is still a matter of extensive research. The following lines of code can apply tone mapping to the HDR image obtained in the previous example:

```cpp
Mat ldr;
Ptr<TonemapDurand> tonemap = createTonemapDurand(2.2f);
tonemap->process(hdr, ldr); // ldr is a floating point image with values in interval [0..1]
ldr=ldr*255; // ldr is a floating point image with values in interval [0..1]
imshow("LDR", ldr);
```

The method was proposed by Durand and Dorsey in 2002. The constructor actually accepts a number of parameters that affect the output. The following figure shows you the output. Note how this image is not necessarily better than any of the three original images:
Three other tone-mapping algorithms are available in OpenCV: 
createTonemapDrago, createTonemapReinhard, and createTonemapMantiuk.

An HDR image (the RGBE format, that is, files with the .hdr extension) can be 
displayed using MATLAB. All it takes is three lines of code:

```
hdr=hdrread('hdr.hdr');
rgb=tonemap(hdr);
imshow(rgb);
```

pfstools is an open source suite of command-line tools to read, write, 
and render HDR images. The suite, which can read .hdr and other 
formats, includes a number of camera calibration and tone-mapping 
algorithms. Luminance HDR is free GUI software based on pfstools.
Alignment

The scene that will be captured with multiple exposure images must be static. The camera must also be static. Even if the two conditions met, it is advisable to perform an alignment procedure.

OpenCV provides an algorithm for image alignment proposed by G. Ward in 2003. The main function, `createAlignMTB`, takes an input parameter that defines the maximum shift (actually, a logarithm the base two of the maximum shift in each dimension). The following lines should be inserted right before estimating the camera response function in the previous example:

```cpp
vector<Mat> images_(images);
Ptr<AlignMTB> align=createAlignMTB(4);// 4=max 16 pixel shift
align->process(images_, images);
```

Exposure fusion

We can also combine images with multiple exposures with neither camera response calibration (that is, exposure times) nor intermediate HDR image. This is called exposure fusion. The method was proposed by Mertens et al in 2007. The following lines perform exposure fusion (`images` is the STL vector of input images; refer to the previous example):

```cpp
Mat fusion;
Ptr<MergeMertens> merge_mertens = createMergeMertens();
merge_mertens->process(images, fusion); // fusion is a
fusion=fusion*255; // float. point image w. values in [0..1]
imwrite("fusion.png", fusion);
```

The following figure shows you the result:
Seamless cloning

In photomontages, we typically want to cut an object/person in a source image and insert it into a target image. Of course, this can be done in a straightforward way by simply pasting the object. However, this would not produce a realistic effect. See, for example, the following figure, in which we wanted to insert the boat in the top half of the image into the sea at the bottom half of the image:
As of OpenCV 3, there are seamless cloning functions available in which the result is more realistic. This function is called `seamlessClone` and it uses a method proposed by Perez and Gangnet in 2003. The following `seamlessCloning` example shows you how it can be used:

```cpp
#include <opencv2/photo.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int, char** argv)
{
    // Load and show images...
    Mat source = imread("source1.png", IMREAD_COLOR);
    Mat destination = imread("destination1.png", IMREAD_COLOR);
    Mat mask = imread("mask.png", IMREAD_COLOR);
    imshow("source", source);
    imshow("mask", mask);
    imshow("destination", destination);

    Mat result;
    Point p;    // p will be near top right corner
    p.x = (float)2*destination.size().width/3;
    p.y = (float)destination.size().height/4;
    seamlessClone(source, destination, mask, p, result, NORMAL_CLONE);
    imshow("result", result);

    cout << "Done. Press any key to exit...\n";
    waitKey(); // Wait for key press
    return 0;
}
```
The example is straightforward. The `seamlessClone` function takes the source, destination, and mask images and a point in the destination image in which the cropped object will be inserted (these three images can be downloaded from https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/cloning/Normal_Cloning). See the result in the following figure:

Seamless cloning

The last parameter of `seamlessClone` represents the exact method to be used (there are three methods available that produce a different final effect). On the other hand, the library provides the following related functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>colorChange</code></td>
<td>Multiplies each of the three color channels of the source image by a factor, applying the multiplication only in the region given by the mask</td>
</tr>
<tr>
<td><code>illuminationChange</code></td>
<td>Changes illumination of the source image, only in the region given by the mask</td>
</tr>
<tr>
<td><code>textureFlattening</code></td>
<td>Washes out textures in the source image, only in the region given by the mask</td>
</tr>
</tbody>
</table>

As opposed to `seamlessClone`, these three functions only accept source and mask images.
Decolorization

Decolorization is the process of converting a color image to grayscale. Given this definition, the reader may well ask, don't we already have grayscale conversion? Yes, grayscale conversion is a basic routine in OpenCV and any image-processing library. The standard conversion is based on a linear combination of the R, G, and B channels. The problem is that such a conversion may produce images in which contrast in the original image is lost. The reason is that two different colors (which are perceived as contrasts in the original image) may end up being mapped to the same grayscale value. Consider the conversion of two colors, A and B, to grayscale. Let's suppose that B is a variation of A in the R and G channels:

\[
A = (R, G, B) \quad \Rightarrow \quad G = (R + G + B)/3
\]

\[
B = (R-x, G+x, B) \quad \Rightarrow \quad G = (R-x + G+x + B)/3 = (R + G + B)/3
\]

Even though they are perceived as distinct, the two colors A and B are mapped to the same grayscale value! The images from the following `decolorization` example show this:

```cpp
#include <opencv2/photo.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int, char** argv)
{
    // Load and show images...
    Mat source = imread("color_image_3.png", IMREAD_COLOR);
    imshow("source", source);

    // first compute and show standard grayscale conversion...
    Mat grayscale = Mat(source.size(),CV_8UC1);
    cvtColor(source, grayscale, COLOR_BGR2GRAY);
    imshow("grayscale",grayscale);

    // now compute and show decolorization...
    Mat decolorized = Mat(source.size(),CV_8UC1);
```

---

[178]
Mat dummy = Mat(source.size(), CV_8UC3);
decolor(source, decolorized, dummy);
imshow("decolorized", decolorized);

cout << "\nDone. Press any key to exit...\n";
waitKey(); // Wait for key press
return 0;
}

Decolorization example output

The example is straightforward. After reading the image and showing the result of a standard grayscale conversion, it uses the `decolor` function to perform the decolorization. The image used (the `color_image_3.png` file) is included in the `opencv_extra` repository at https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/decolor.

The image used in the example is actually an extreme case. Its colors have been chosen so that the standard grayscale output is fairly homogeneous.
Non-photorealistic rendering

As part of the photo module, four functions are available that transform an input image in a way that produces a non-realistic but still artistic output. The functions are very easy to use and a nice example is included with OpenCV (npr_demo). For illustrative purposes, here we show you a table that allows you to grasp the effect of each function. Take a look at the following fruits.jpg input image, included with OpenCV:

![The input reference image](image)

The effects are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>edgePreservingFilter</td>
<td>Smoothing is a handy and frequently used filter. This function performs smoothing while preserving object edge details.</td>
</tr>
<tr>
<td>Function</td>
<td>Effect</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>detailEnhance</td>
<td>Enhances details in the image</td>
</tr>
<tr>
<td>pencilSketch</td>
<td>A pencil-like line drawing version of the input image</td>
</tr>
</tbody>
</table>

![Detail Enhanced](image1.jpg)

![Color Pencil Sketch](image2.jpg)
Summary

In this chapter, you learned what computational photography is and the related functions available in OpenCV 3. We explained the most important functions within the photo module, but note that other functions of this module (inpainting and noise reduction) were also considered in previous chapters. Computational photography is a rapidly expanding field, with strong ties to computer graphics. Therefore, this module of OpenCV is expected to grow in future versions.

The next chapter will be devoted to an important aspect that we have not yet considered: time. Many of the functions explained take a significant time to compute the results. The next chapter will show you how to deal with that using modern hardware.
Where to buy this book

You can buy Learning Image Processing with OpenCV 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.