

---

# The OpenCV Tutorials

*Release 2.3*

July 21, 2011



# CONTENTS

<b>1</b>	<b>Introduction to OpenCV</b>	<b>3</b>
1.1	Installation in Linux	5
1.2	Using OpenCV with gcc and CMake	6
1.3	Using OpenCV with Eclipse (plugin CDT)	8
1.4	Installation in Windows	13
1.5	How to build applications with OpenCV inside the <i>Microsoft Visual Studio</i>	24
1.6	Using Android binary package with Eclipse	30
1.7	Load and Display an Image	41
1.8	Load, Modify, and Save an Image	44
<b>2</b>	<b>core module. The Core Functionality</b>	<b>47</b>
2.1	Mat - The Basic Image Container	49
2.2	How to scan images, lookup tables and time measurement with OpenCV	54
2.3	Adding (blending) two images using OpenCV	57
2.4	Changing the contrast and brightness of an image!	59
2.5	Basic Drawing	63
2.6	Fancy Drawing!	67
2.7	Mask operations on matrixes	72
<b>3</b>	<b>imgproc module. Image Processing</b>	<b>73</b>
3.1	Smoothing Images	78
3.2	Eroding and Dilating	83
3.3	More Morphology Transformations	89
3.4	Image Pyramids	95
3.5	Basic Thresholding Operations	101
3.6	Making your own linear filters!	109
3.7	Adding borders to your images	113
3.8	Sobel Derivatives	117
3.9	Laplace Operator	123
3.10	Canny Edge Detector	127
3.11	Hough Line Transform	132
3.12	Hough Circle Transform	138
3.13	Remapping	142
3.14	Affine Transformations	148
3.15	Histogram Equalization	154
3.16	Histogram Calculation	160
3.17	Histogram Comparison	167
3.18	Back Projection	172
3.19	Template Matching	177

3.20	Finding contours in your image . . . . .	185
3.21	Convex Hull . . . . .	187
3.22	Creating Bounding boxes and circles for contours . . . . .	189
3.23	Creating Bounding rotated boxes and ellipses for contours . . . . .	191
3.24	Image Moments . . . . .	193
3.25	Point Polygon Test . . . . .	196
<b>4</b>	<b><i>highgui</i> module. High Level GUI and Media</b>	<b>199</b>
4.1	Adding a Trackbar to our applications! . . . . .	200
<b>5</b>	<b><i>calib3d</i> module. Camera calibration and 3D reconstruction</b>	<b>205</b>
<b>6</b>	<b><i>feature2d</i> module. 2D Features framework</b>	<b>207</b>
6.1	Harris corner detector . . . . .	208
6.2	Shi-Tomasi corner detector . . . . .	210
6.3	Creating yor own corner detector . . . . .	213
6.4	Detecting corners location in subpixeles . . . . .	217
<b>7</b>	<b><i>video</i> module. Video analysis</b>	<b>221</b>
<b>8</b>	<b><i>objdetect</i> module. Object Detection</b>	<b>223</b>
<b>9</b>	<b><i>ml</i> module. Machine Learning</b>	<b>225</b>
<b>10</b>	<b><i>gpu</i> module. GPU-Accelerated Computer Vision</b>	<b>227</b>
<b>11</b>	<b>General tutorials</b>	<b>229</b>

The following links describe a set of basic OpenCV tutorials. All the source code mentioned here is provide as part of the OpenCV regular releases, so check before you start copy & pasting the code. The list of tutorials below is automatically generated from reST files located in our SVN repository.

As always, we would be happy to hear your comments and receive your contributions on any tutorial.

- [\*Introduction to OpenCV\*](#)



You will learn how to setup OpenCV on your computer!

---

- [\*core module. The Core Functionality\*](#)



Here you will learn the about the basic building blocks of the library. A must read and know for understanding how to manipulate the images on a pixel level.

---

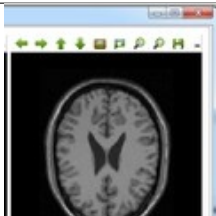
- [\*imgproc module. Image Processing\*](#)



In this section you will learn about the image processing (manipulation) functions inside OpenCV.

---

- [\*highgui module. High Level GUI and Media\*](#)



This section contains valuable tutorials about how to read/save your image/video files and how to use the built-in graphical user interface of the library.

---

- [\*calib3d module. Camera calibration and 3D reconstruction\*](#)



Although we got most of our images in a 2D format they do come from a 3D world. Here you will learn how to find out from the 2D images information about the 3D world.

---

- [\*feature2d module. 2D Features framework\*](#)



Learn about how to use the feature points detectors, descriptors and matching framework found inside OpenCV.

- *video module. Video analysis*



Look here in order to find use on your video stream algorithms like: motion extraction, feature tracking and foreground extractions.

- *objdetect module. Object Detection*



Ever wondered how your digital camera detects peoples and faces? Look here to find out!

- *ml module. Machine Learning*



Use the powerfull machine learning classes for statistical classification, regression and clustering of data.

- *gpu module. GPU-Accelerated Computer Vision*



Squeeze out every little computation power from your system by using the power of your video card to run the OpenCV algorithms.

- *General tutorials*



These tutorials are the bottom of the iceberg as they link together multiple of the modules presented above in order to solve complex problems.

# INTRODUCTION TO OPENCV

Here you can read tutorials about how to set up your computer to work with the OpenCV library. Additionally you can find a few very basic sample source code that will let introduce you to the world of the OpenCV.

- **Linux**



**Title:** *Installation in Linux*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
We will learn how to setup OpenCV in your computer!



**Title:** *Using OpenCV with gcc and CMake*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
We will learn how to compile your first project using gcc and CMake



**Title:** *Using OpenCV with Eclipse (plugin CDT)*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
We will learn how to compile your first project using the Eclipse environment

- **Windows**



**Title:** *Installation in Windows*  
**Compatibility:** > OpenCV 2.0  
**Author:** Bernát Gábor  
You will learn how to setup OpenCV in your Windows Operating System!

---



**Title:** *How to build applications with OpenCV inside the Microsoft Visual Studio*

**Compatibility:** > OpenCV 2.0

**Author:** Bernát Gábor

You will learn what steps you need to perform in order to use the OpenCV library inside a new Microsoft Visual Studio project.

---

- **Android**



**Title:** *Using Android binary package with Eclipse*

**Compatibility:** > OpenCV 2.3.1

**Author:** Andrey Kamaev

You will learn how to setup OpenCV for Android platform!

---

- **From where to start?**



**Title:** *Load and Display an Image*

**Compatibility:** > OpenCV 2.0

**Author:** Ana Huamán

We will learn how to display an image using OpenCV

---



**Title:** *Load, Modify, and Save an Image*

**Compatibility:** > OpenCV 2.0

**Author:** Ana Huamán

We will learn how to save an Image in OpenCV...plus a small conversion to grayscale

---



## 1.1 Installation in Linux

These steps have been tested for Ubuntu 10.04 but should work with other distros.

### Required packages

- GCC 4.x or later. This can be installed with  
`sudo apt-get install build-essential`
- CMake 2.6 or higher
- Subversion (SVN) client
- GTK+2.x or higher, including headers
- pkgconfig
- libpng, zlib, libjpeg, libtiff, libjasper with development files (e.g. libjpeg-dev)
- Python 2.3 or later with developer packages (e.g. python-dev)
- SWIG 1.3.30 or later
- libavcodec
- libdc1394 2.x

All the libraries above can be installed via Terminal or by using Synaptic Manager

### Getting OpenCV source code

You can use the latest stable OpenCV version available in *sourceforge* or you can grab the latest snapshot from the SVN repository:

#### Getting the latest stable OpenCV version

- Go to <http://sourceforge.net/projects/opencvlibrary>
- Download the source tarball and unpack it

#### Getting the cutting-edge OpenCV from SourceForge SVN repository

Launch SVN client and checkout either

1. the current OpenCV snapshot from here: <https://code.ros.org/svn/opencv/trunk>
2. or the latest tested OpenCV snapshot from here: [http://code.ros.org/svn/opencv/tags/latest\\_tested\\_snapshot](http://code.ros.org/svn/opencv/tags/latest_tested_snapshot)

In Ubuntu it can be done using the following command, e.g.:

```
cd ~/<my_working_directory>
svn co https://code.ros.org/svn/opencv/trunk
```

## Building OpenCV from source using CMake, using the command line

1. Create a temporary directory, which we denote as <cmake\_binary\_dir>, where you want to put the generated Makefiles, project files as well the object files and output binaries
2. Enter the <cmake\_binary\_dir> and type

```
cmake [<some optional parameters>] <path to the OpenCV source directory>
```

For example

```
cd ~/opencv
mkdir release
cd release
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX= /usr/local
```

3. Enter the created temporary directory (<cmake\_binary\_dir>) and proceed with:

```
make
sudo make install
```

## 1.2 Using OpenCV with gcc and CMake

---

**Note:** We assume that you have successfully installed OpenCV in your workstation.

---

The easiest way of using OpenCV in your code is to use [CMake](#). A few advantages (taken from the Wiki):

- No need to change anything when porting between Linux and Windows
- Can easily be combined with other tools by CMake( i.e. Qt, ITK and VTK )

If you are not familiar with CMake, checkout the [tutorial](#) on its website.

### Steps

#### Create a program using OpenCV

Let's use a simple program such as DisplayImage.cpp shown below.

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat image;
    image = imread( argv[1], 1 );

    if( argc != 2 || !image.data )
    {
        printf( "No image data \n" );
        return -1;
    }
}
```

```
namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
imshow( "Display Image", image );

waitKey(0);

return 0;
}
```

## Create a CMake file

Now you have to create your CMakeLists.txt file. It should look like this:

```
project( DisplayImage )
find_package( OpenCV REQUIRED )
add_executable( DisplayImage DisplayImage )
target_link_libraries( DisplayImage ${OpenCV_LIBS} )
```

## Generate the executable

This part is easy, just proceed as with any other project using CMake:

```
cd <DisplayImage_directory>
cmake .
make
```

## Result

By now you should have an executable (called DisplayImage in this case). You just have to run it giving an image location as an argument, i.e.:

```
./DisplayImage lena.jpg
```

You should get a nice window as the one shown below:



## 1.3 Using OpenCV with Eclipse (plugin CDT)

---

**Note:** Two ways, one by forming a project directly, and another by CMake

---

### Prerequisites

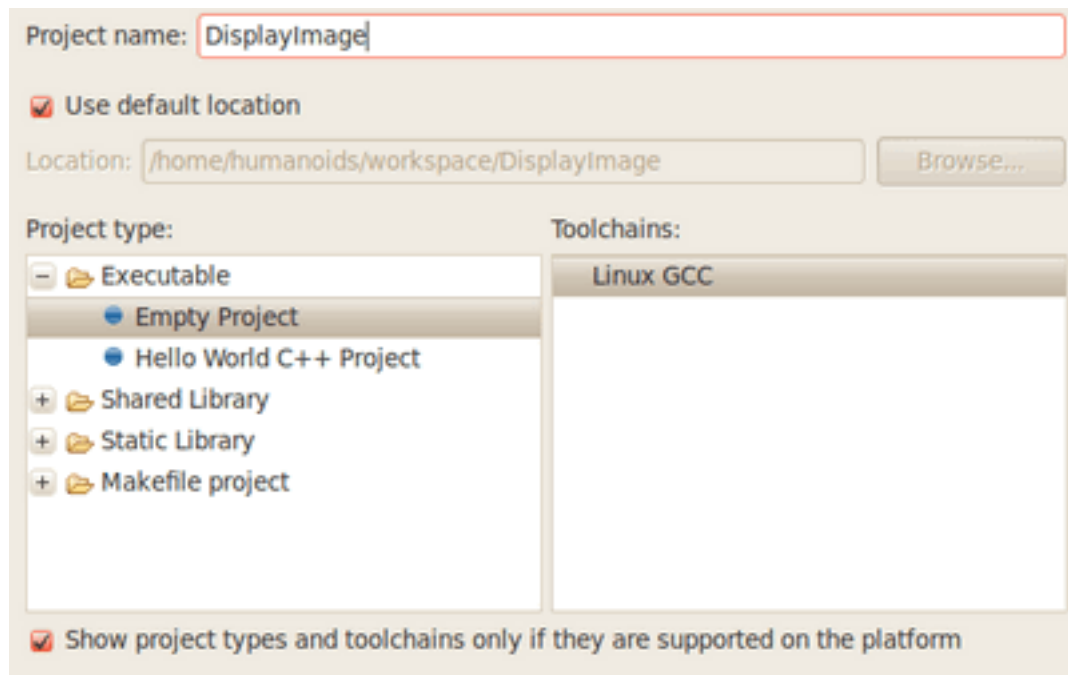
1. Having installed [Eclipse](#) in your workstation (only the CDT plugin for C/C++ is needed). You can follow the following steps:
  - Go to the [Eclipse site](#)
  - Download [Eclipse IDE for C/C++ Developers](#) . Choose the link according to your workstation.
2. Having installed OpenCV. If not yet, go [here](#).

### Making a project

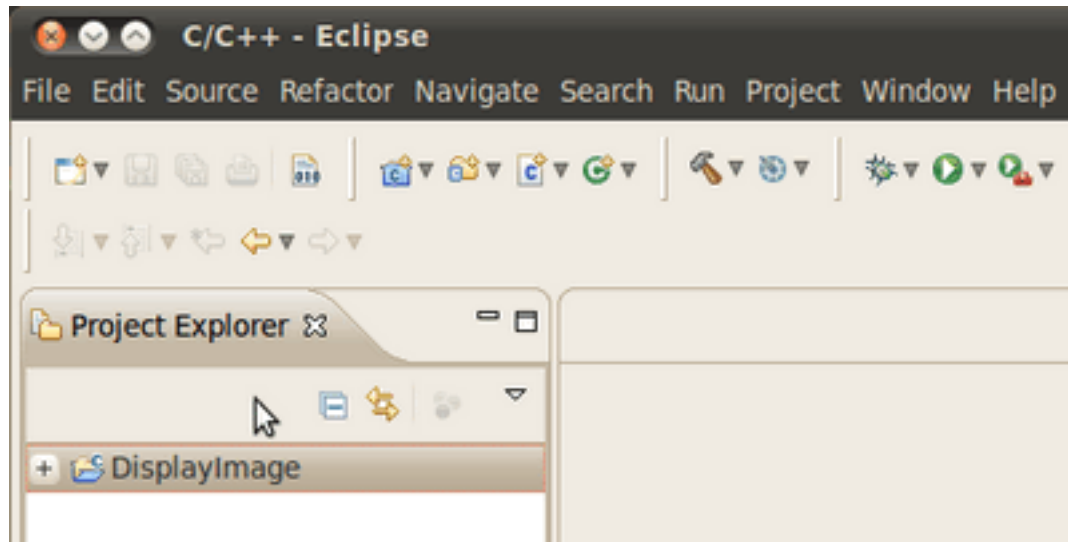
1. Start Eclipse. Just run the executable that comes in the folder.
2. Go to **File -> New -> C/C++ Project**



3. Choose a name for your project (i.e. DisplayImage). An **Empty Project** should be okay for this example.

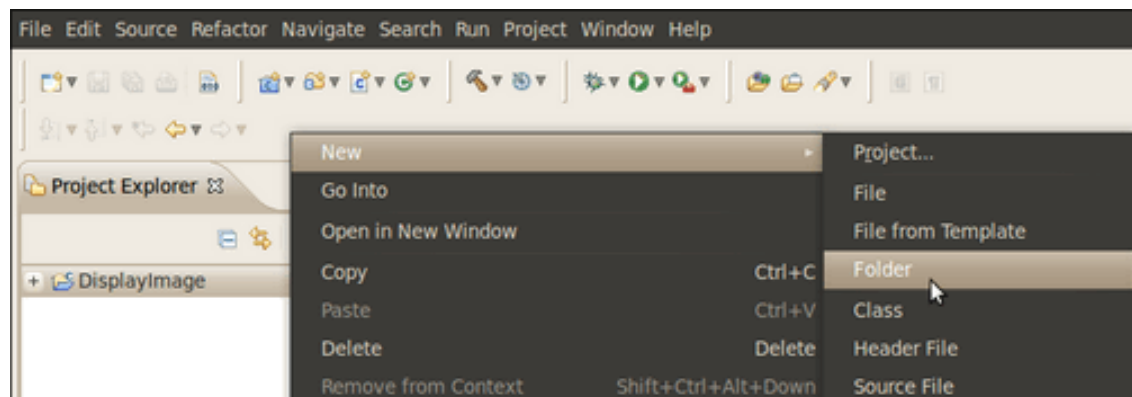


4. Leave everything else by default. Press **Finish**.
5. Your project (in this case DisplayImage) should appear in the **Project Navigator** (usually at the left side of your window).

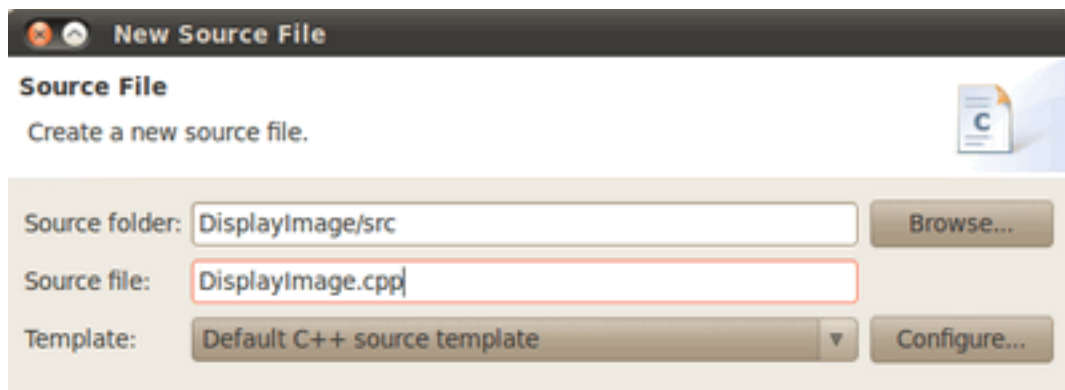


6. Now, let's add a source file using OpenCV:

- Right click on **DisplayImage** (in the Navigator). **New -> Folder** .



- Name your folder **src** and then hit **Finish**
- Right click on your newly created **src** folder. Choose **New source file**:
- Call it **DisplayImage.cpp**. Hit **Finish**



7. So, now you have a project with an empty .cpp file. Let's fill it with some sample code (in other words, copy and paste the snippet below):

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat image;
    image = imread( argv[1], 1 );

    if( argc != 2 || !image.data )
    {
        printf( "No image data \n" );
        return -1;
    }

    namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
    imshow( "Display Image", image );

    waitKey(0);

    return 0;
}
```

8. We are only missing one final step: To tell OpenCV where the OpenCV headers and libraries are. For this, do the following:

- Go to **Project**→**Properties**
- In **C/C++ Build**, click on **Settings**. At the right, choose the **Tool Settings** Tab. Here we will enter the headers and libraries info:
  - (a) In **GCC C++ Compiler**, go to **Includes**. In **Include paths(-I)** you should include the path of the folder where opencv was installed. In our example, this is `/usr/local/include/opencv`.



---

**Note:** If you do not know where your opencv files are, open the **Terminal** and type:

```
pkg-config --cflags opencv
```

For instance, that command gave me this output:

```
-I/usr/local/include/opencv -I/usr/local/include
```

---

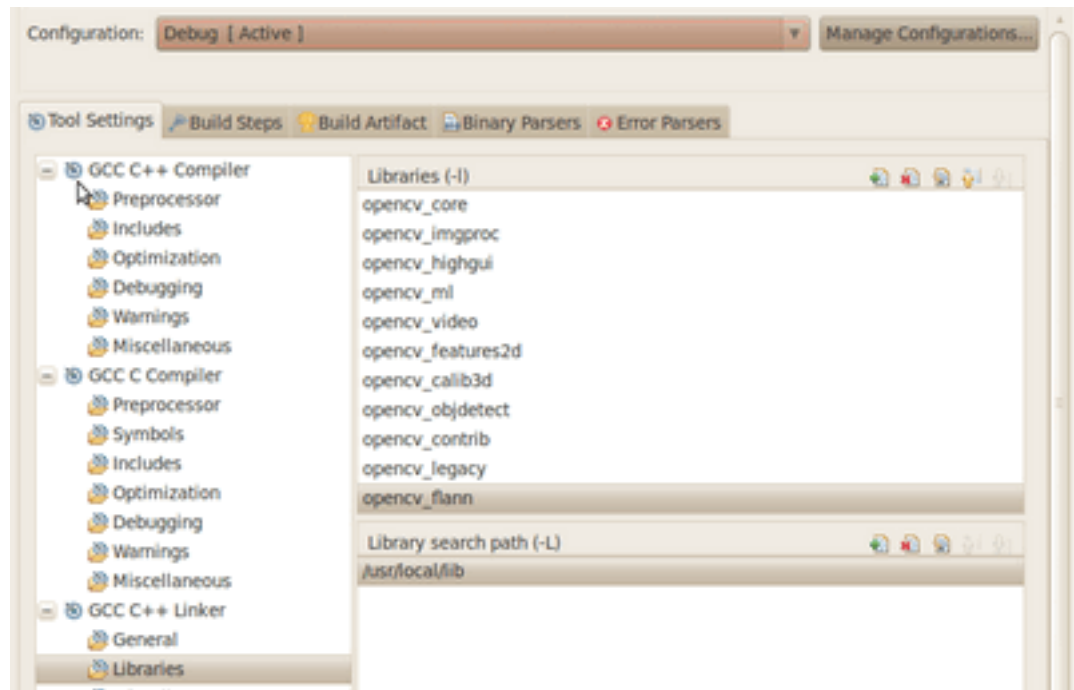
- (b) Now go to **GCC C++ Linker**, there you have to fill two spaces:

First in **Library search path (-L)** you have to write the path to where the opencv libraries reside, in my case the path is:

```
/usr/local/lib
```

Then in **Libraries(-l)** add the OpenCV libraries that you may need. Usually just the 3 first on the list below are enough (for simple applications) . In my case, I am putting all of them since I plan to use the whole bunch:

opencv\_core opencv\_imgproc opencv\_highgui opencv\_ml opencv\_video opencv\_features2d  
 opencv\_calib3d opencv\_objdetect opencv\_contrib opencv\_legacy opencv\_flann



If you don't know where your libraries are (or you are just psychotic and want to make sure the path is fine), type in **Terminal**:

```
pkg-config --libs opencv
```

My output (in case you want to check) was: .. code-block:: bash

```
-L/usr/local/lib -lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -  

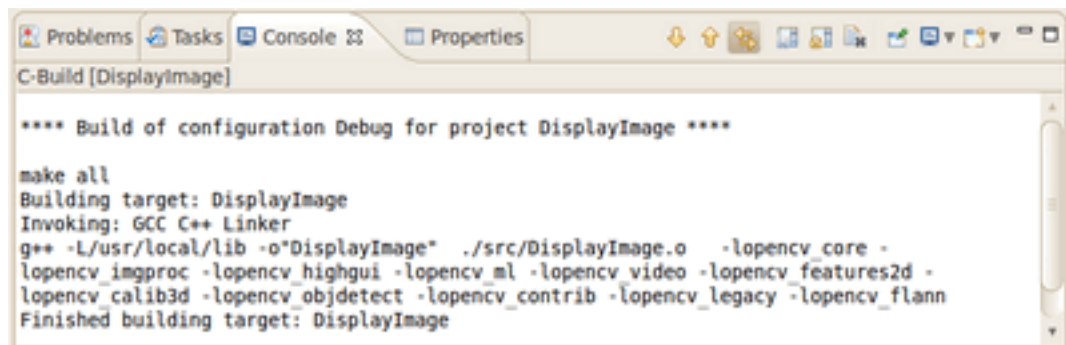
  -lopencv_video -lopencv_features2d -lopencv_calib3d -lopencv_objdetect -lopencv_contrib  

  -lopencv_legacy -lopencv_flann
```

Now you are done. Click **OK**

- Your project should be ready to be built. For this, go to **Project->Build all**

In the Console you should get something like



If you check in your folder, there should be an executable there.

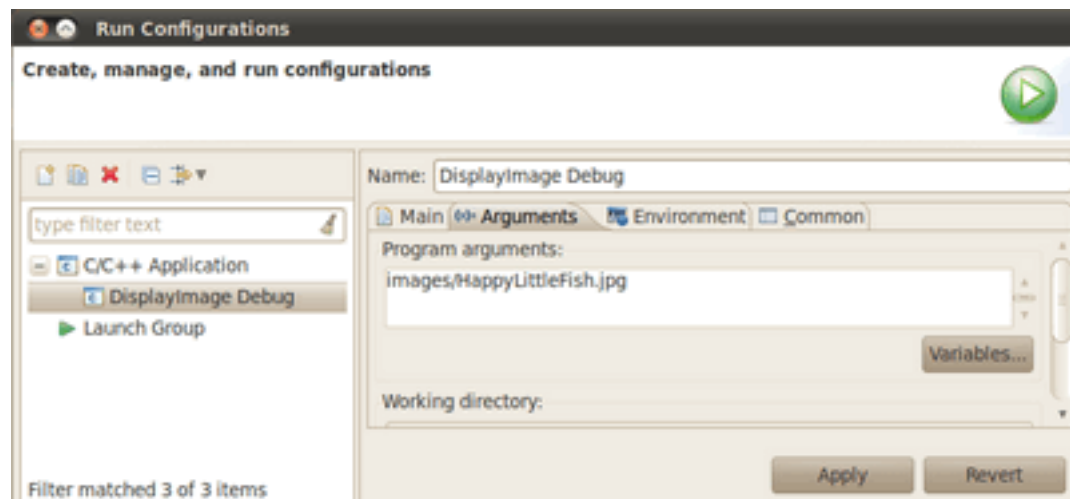
### Running the executable

So, now we have an executable ready to run. If we were to use the Terminal, we would probably do something like:

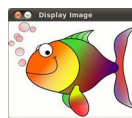
```
cd <DisplayImage_directory>
cd src
./DisplayImage ../images/HappyLittleFish.png
```

Assuming that the image to use as the argument would be located in <DisplayImage\_directory>/images/HappyLittleFish.png. We can still do this, but let's do it from Eclipse:

1. Go to **Run->Run Configurations**
2. Under C/C++ Application you will see the name of your executable + Debug (if not, click over C/C++ Application a couple of times). Select the name (in this case **DisplayImage Debug**).
3. Now, in the right side of the window, choose the **Arguments** Tab. Write the path of the image file we want to open (path relative to the workspace/DisplayImage folder). Let's use **HappyLittleFish.png**:



4. Click on the **Apply** button and then in Run. An OpenCV window should pop up with the fish image (or whatever you used).



5. Congratulations! You are ready to have fun with OpenCV using Eclipse.

### V2: Using CMake+OpenCV with Eclipse (plugin CDT)

(See the *getting started* <[http://opencv.willowgarage.com/wiki/Getting\\_started](http://opencv.willowgarage.com/wiki/Getting_started)> section of the OpenCV Wiki)

Say you have or create a new file, *helloworld.cpp* in a directory called *foo*:

```
#include <cv.h>
#include <highgui.h>
int main ( int argc, char **argv )
{
    cvNamedWindow( "My Window", 1 );
    IplImage *img = cvCreateImage( cvSize( 640, 480 ), IPL_DEPTH_8U, 1 );
    CvFont font;
```



```

double hScale = 1.0;
double vScale = 1.0;
int lineWidth = 1;
cvInitFont( &font, CV_FONT_HERSHEY_SIMPLEX | CV_FONT_ITALIC,
            hScale, vScale, 0, lineWidth );
cvPutText( img, "Hello World!", cvPoint( 200, 400 ), &font,
           cvScalar( 255, 255, 0 ) );
cvShowImage( "My Window", img );
cvWaitKey();
return 0;
}

```

1. Create a build directory, say, under *foo*: `mkdir /build`. Then `cd build`.
2. Put a *CmakeLists.txt* file in build:

```

PROJECT( helloworld_proj )
FIND_PACKAGE( OpenCV REQUIRED )
ADD_EXECUTABLE( helloworld helloworld.cxx )
TARGET_LINK_LIBRARIES( helloworld ${OpenCV_LIBS} )

```

1. Run: `cmake-gui ..` and make sure you fill in where opencv was built.
  2. Then click configure and then generate. If it's OK, **quit cmake-gui**
  3. Run `make -j4` (the `-j4` is optional, it just tells the compiler to build in 4 threads). Make sure it builds.
  4. Start eclipse. Put the workspace in some directory but **not** in `foo` or `foo\\build`
  5. Right click in the Project Explorer section. Select Import And then open the C/C++ filter. Choose **\*Existing Code as a Makefile Project**
  6. Name your project, say *helloworld*. Browse to the Existing Code location `foo\\build` (where you ran your `cmake-gui` from). Select *Linux GCC* in the *"Toolchain for Indexer Settings"* and press *Finish*.
  7. Right click in the Project Explorer section. Select Properties. Under C/C++ Build, set the *build directory*: from something like `${workspace_loc:/helloworld}` to `${workspace_loc:/helloworld}/build` since that's where you are building to.
1. You can also optionally modify the Build command: from `make` to something like `make VERBOSE=1 -j4` which tells the compiler to produce detailed symbol files for debugging and also to compile in 4 parallel threads.
  1. Done!

## 1.4 Installation in Windows

The description here was tested by the author using the Windows 7 SP1 operating system (OS). Nevertheless, it should also work on any other Windows OS too. If you encounter errors after following the steps described here feel free to contact us via our [user group](#) and we will try to fix your problem.

---

**Note:** To use the OpenCV library you have two options: *Installation by using the pre-built libraries* or *Installation by making your own libraries from the source files*. While the first one is easier to complete, it only works if you are coding inside the latest Microsoft Visual Studio integrated development environments (IDE) and doesn't take advantage of the most novel technologies we integrate into our library.

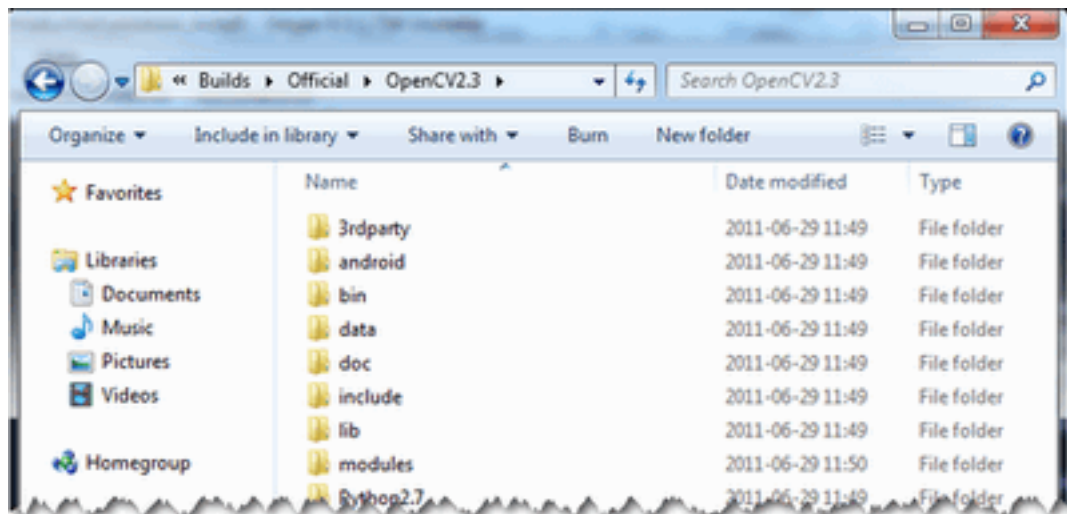
---

## Installation by using the pre-built libraries

1. Open up a web browser and go to: <http://sourceforge.net/projects/opencvlibrary/files/opencv-win/>
2. Open the folder for the latest version (currently this is 2.3).
3. Choose a build you want to use and download it. The naming conventions used will show what kind of support they offer. For example:
  - *vs2010* means the Visual Studio
  - *win32* means that it is for 32 bit applications in the OS
  - *gpu* means that it includes the support for using your GPU in order to further increase the performance of the library).

If you downloaded the source files present here see *Installation by making your own libraries from the source files*.

4. Make sure you have admin rights. Start the setup and follow the wizard. Agree to the " License Agreement " .
5. While adding the OpenCV library to the system path is a good decision for a better control of this we will do it manually. Therefore, make sure you do not set this option.
6. Most of the time it is a good idea to install the source files as this will allow for you to debug into the OpenCV library, if it is necessary. Therefore, just follow the default settings of the wizard and finish the installation.
7. You can check the installation at the chosen path as you can see below.



8. To finalize the installation go to the *Set the OpenCV enviroment variable and add it to the systems path* section.

## Installation by making your own libraries from the source files

You may find the content of this tutorial also inside the following videos: [Part 1](#) and [Part 2](#), hosted on YouTube.

If you are building your own libraries you can take either the source files from our latest:

- stable and tested build - <https://code.ros.org/svn/opencv/branches/>
- development build - <https://code.ros.org/svn/opencv/trunk/>

While the later one may contain a couple of new and experimental algorithms, performance increases and interface improvements, be aware, that it may also contain many-many bugs. Using the first one is recommended in most of the cases. That is unless you are extending the OpenCV library itself or really need to most up to date version of it.

Building the OpenCV library from scratch requires a couple of tools installed beforehand:

- An **Integrated Developer Environment (IDE)** preferably, or just a C/C++ compiler that will actually make the binary files. Here I will use the [Microsoft Visual Studio](#). Nevertheless, you can use any other *IDE* that has a valid C/C++ compiler.
- Then **CMake** is a neat tool that will make the project files (for your chosen *IDE*) from the OpenCV source files. It will also allow an easy configuration of the OpenCV build files, in order to make binary files that fits exactly to your needs.
- A **Subversion Control System (SVN)** to acquire the OpenCV source files. A good tool for this is [TortoiseSVN](#). Alternatively, you can just download an archived version of the source files from the [Sourceforge OpenCV page](#).

OpenCV may come in multiple flavors. There is a “core” section that will work on its own. Nevertheless, there are a couple of tools, libraries made by other organizations (so called 3rd parties) that offer services of which the OpenCV may take advantage. These will improve in many ways its capabilities. In order to use any of them, you need to download and install them on your system.

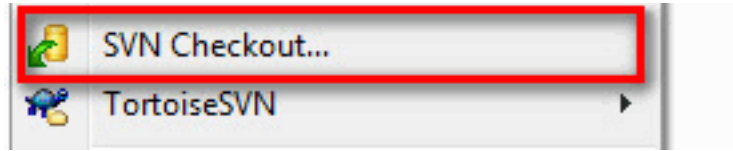
- The **Python libraries** are required to build the *Python interface* of OpenCV. For now use the version 2.7.x. This is also a must have if you want to build the *OpenCV documentation*.
- **Numpy** is a scientific computing package for Python. Required for the *Python interface*.
- **Intel © Threading Building Blocks (TBB)** is used inside OpenCV for parallel code snippets. Using this will make sure that the OpenCV library will take advantage of all the cores you have in your systems CPU.
- **Intel © Integrated Performance Primitives (IPP)** may be used to improve the performance of color conversion, Haar training and DFT functions of the OpenCV library. Watch out as this isn't a *free* service.
- OpenCV offers a somewhat fancier and more useful graphical user interface, than the default one by using the **Qt framework**. For a quick overview of what this has to offer look into the documentations *highgui* module, under the *Qt New Functions* section. Version 4.6 or later of the framework is required.
- **Eigen** is a C++ template library for linear algebra.
- The latest **CUDA Toolkit** will allow you to use the power lying inside your GPU. This will drastically improve performance for some of the algorithms, like the HOG descriptor. Getting to work more and more of our algorithms on the GPUs is a constant effort of the OpenCV team.
- **OpenEXR** source files are required for the library to work with this high dynamic range (HDR) image file format.
- The **OpenNI Framework** contains a set of open source APIs that provide support for natural interaction with devices via methods such as voice command recognition, hand gestures and body motion tracking.
- **Miktex** is the best **TEX** implementation on the Windows OS. It is required to build the *OpenCV documentation*.
- **Sphinx** is a python documentation generator and is the tool that will actually create the *OpenCV documentation*. This on its own requires a couple of tools installed, I will cover this in depth at the [How to Install Sphinx](#) section.

Now I will describe the steps to follow for a full build (using all the above frameworks, tools and libraries). If you do not need the support for some of these you can just freely skip those parts.

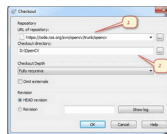
## Building the library

1. Make sure you have a working IDE with a valid compiler. In case of the Microsoft Visual Studio just install it and make sure it starts up.
2. Install **CMake**. Simply follow the wizard, no need to add it to the path. The default install options are great. No need to change them.

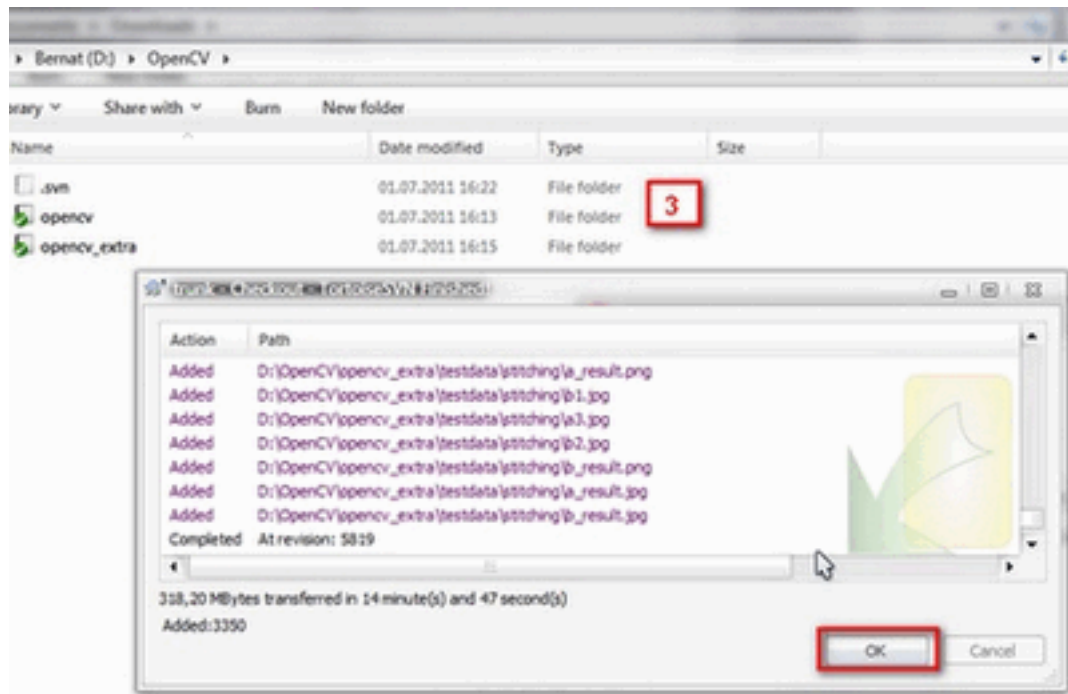
3. Install [TortoiseSVN](#). Choose the 32 or 64 bit version according to the type of OS you work in. Again follow the wizard, default options are good. Restart of your system is required.
4. Choose a directory in your file system where you will download the OpenCV libraries. I recommend creating a new one that has short path and no special characters in it, for example `D:/OpenCV`. During this tutorial I'll suppose you've done so. If you use a different directory just change this front part of the path in my future examples. Then, *Right Click* → *SVN Checkout...* in the directory.



A window will appear where you can select from what repository you want to download source files (1) and to which directory (2):



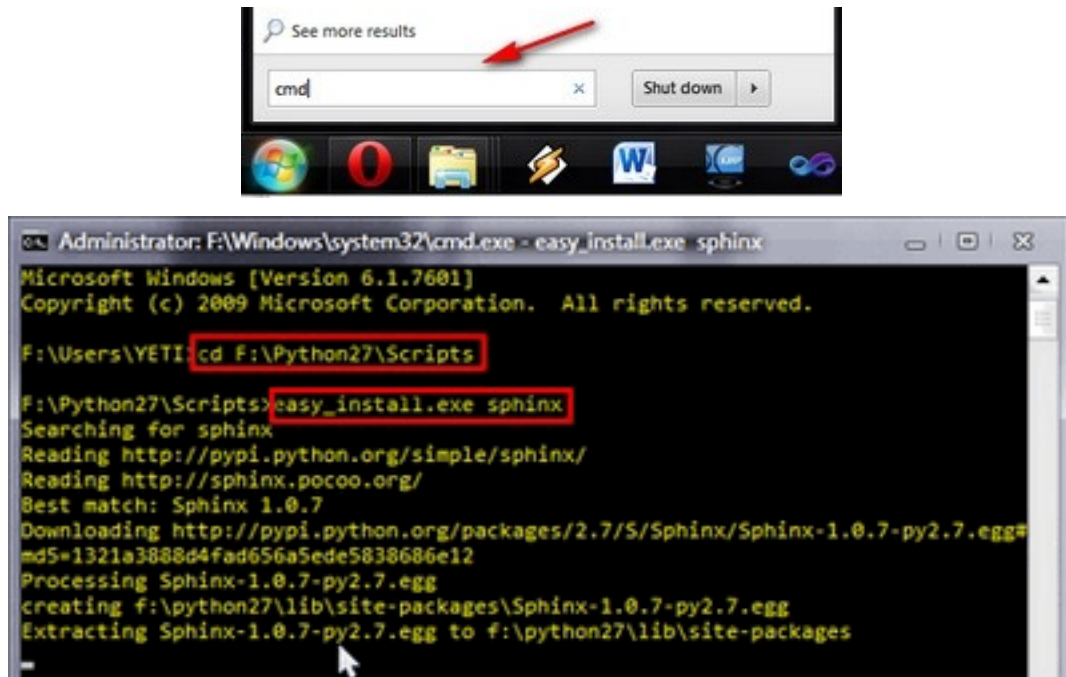
Add here either ones of the versions described above. Then push the OK button and be patient as the repository currently is over 330MB to download. It will take some time until it is finished depending on your Internet connection.



When you are done you should have a *opencv* and an *opencv\_extra* directory as seen at (3).

5. In this section I will cover installing the 3rd party libraries.
  - (a) Download the [Python libraries](#) and install it with the default options. You will need a couple other python extensions. Luckily installing all these may be automated by a nice tool called [Setuptools](#). Download and install again.
  - (b) Installing Sphinx is easy once you have installed *Setuptools*. This contains a little application that will automatically connect to the python databases and download the latest version of many python scripts.

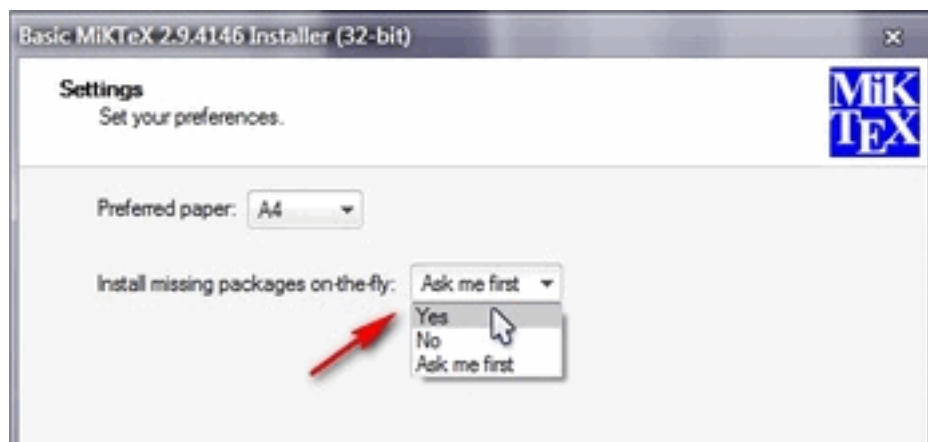
Start up a command window (enter `cmd` into the windows start menu and press enter) and use the `CD` command to navigate to your Python folders Script sub-folder. Here just pass to the `easy_install.exe` as argument the name of the program you want to install. Add the `sphinx` argument.



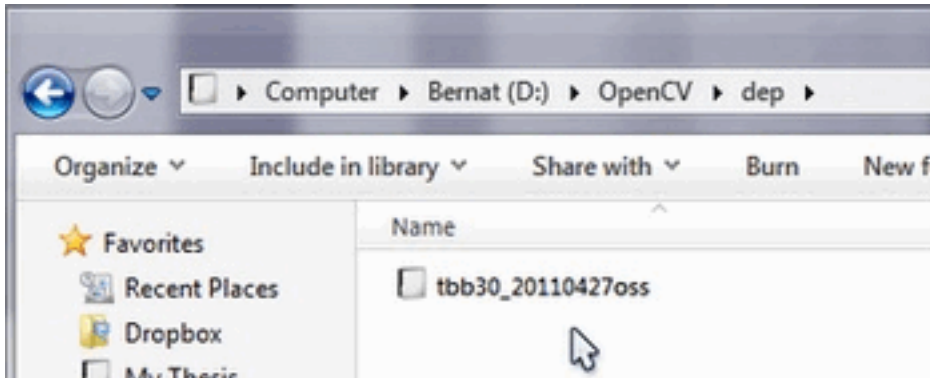
**Note:** The `CD` navigation command works only inside a drive. For example if you are somewhere in the `C:` drive you cannot use it this to go to another drive (like for example `D:`). To do so you first need to change drives letters. For this simply enter the command `D:`. Then you can use the `CD` to navigate to specific folder inside the drive. Bonus tip: you can clear the screen by using the `CLS` command.

This will also install its prerequisites [Jinja2](#) and [Pygments](#).

- (c) The easiest way to install [Numpy](#) is to just download its binaries from the [sourceforge](#) page. Make sure your download and install exactly the binary for your python version (so for version 2.7).
- (d) Download the [Miktex](#) and install it. Again just follow the wizard. At the fourth step make sure you select for the “*Install missing packages on-the-fly*” the *Yes* option, as you can see on the image below. Again this will take quite some time so be patient.



- (e) For the Intel © Threading Building Blocks (*TBB*) download the source files and extract it inside a directory on your system. For example let there be `D:/OpenCV/dep`. For installing the Intel © Integrated Performance Primitives (*IPP*) the story is the same. For extracting the archives I recommend using the 7-Zip application.

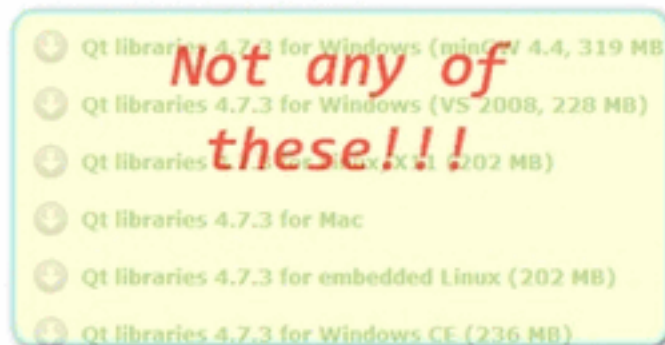


- (f) In case of the Eigen library it is again a case of download and extract to the `D:/OpenCV/dep` directory.
- (g) Same as above with OpenEXR.
- (h) For the OpenNI Framework you need to install both the development build and the PrimeSensor Module.
- (i) For the CUDA you need again two modules: the latest CUDA Toolkit and the *CUDA Tools SDK*. Download and install both of them with a *complete* option by using the 32 or 64 bit setups according to your OS.
- (j) In case of the Qt framework you need to build yourself the binary files (unless you use the Microsoft Visual Studio 2008 with 32 bit compiler). To do this go to the Qt Downloads page. Download the source files (not the installers!!!):



Here is the latest version of the Qt libraries. The edition you download here must match the OS you have your development system on.

The source code is available as a **zip (236 MB)** or a tar.gz (202 MB). Or visit the repository at [qt.gitorious.org/qt](http://qt.gitorious.org/qt).



Extract it into a nice and short named directory like `D:/openCV/dep/qt/`. Then you need to build it. Start up a *Visual Studio Command Prompt (2010)* by using the start menu search (or navigate through the start menu *All Programs* → *Microsoft Visual Studio 2010* → *Visual Studio Tools* → *Visual Studio Command Prompt (2010)*).





Now navigate to the extracted folder and enter inside it by using this console window. You should have a folder containing files like *Install*, *Make* and so on. Use the *dir* command to list files inside your current directory. Once arrived at this directory enter the following command:

```
configure.exe -release -no-webkit -no-phonon -no-phonon-backend -no-script -no-scripttools
              -no-qt3support -no-multimedia -no-ltcd
```

Completing this will take around 10-20 minutes. Then enter the next command that will take a lot longer (can easily take even more than a full hour):

```
nmake
```

After this set the Qt environment variables using the following command on Windows 7:

```
setx -m QTDIR D:/OpenCV/dep/qt/qt-everywhere-opensource-src-4.7.3
```

Also, add the built binary files path to the system path by using the [Path Editor](#). In our case this is `D:/OpenCV/dep/qt/qt-everywhere-opensource-src-4.7.3/bin`.

---

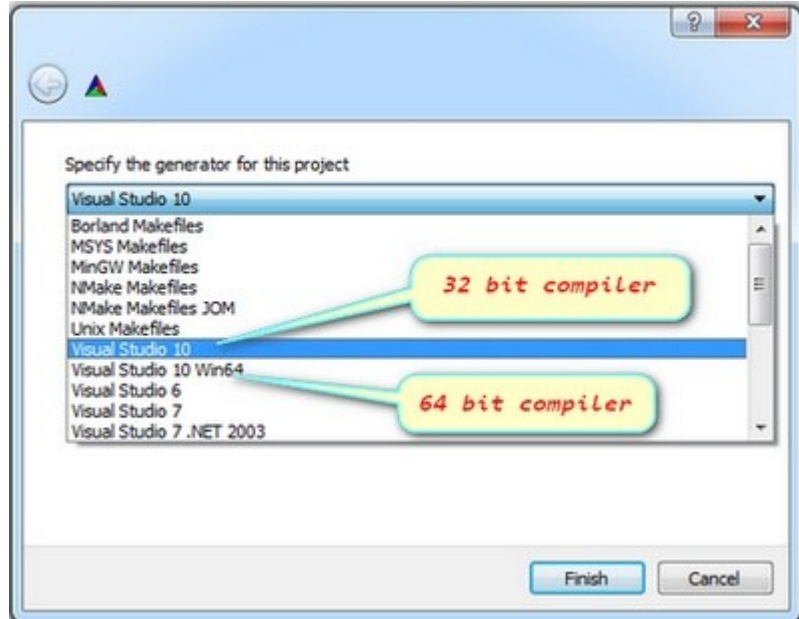
**Note:** If you plan on doing Qt application development you can also install at this point the *Qt Visual Studio Add-in*. After this you can make and build Qt applications without using the *Qt Creator*. Everything is nicely integrated into Visual Studio.

---

- Now start the *CMake (cmake-gui)*. You may again enter it in the start menu search or get it from the *All Programs* → *CMake 2.8* → *CMake (cmake-gui)*. First, select the directory for the source files of the OpenCV library (1). Then, specify a directory where you will build the binary files for OpenCV (2).



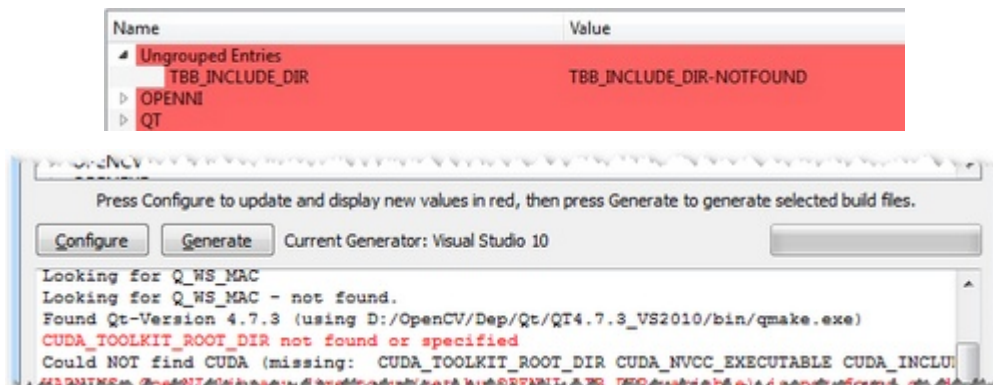
Press the *Configure* button to specify the compiler (and *IDE*) you want to use. Note that in case you can choose between different compilers for making either 64 bit or 32 bit libraries. Select the one you use in your application development.



CMake will start out and based on your system variables will try to automatically locate as many packages as possible. You can modify the packages to use for the build in the *WITH* → *WITH\_X* menu points (where X is the package abbreviation). Here are a list of current packages you can turn on or off:

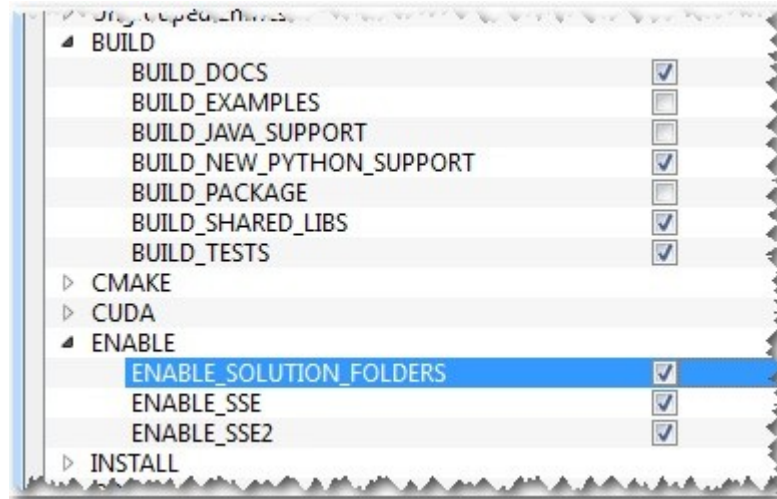


Select all the packages you want to use and press again the *Configure* button. For an easier overview of the build options make sure the *Grouped* option under the binary directory selection is turned on. For some of the packages CMake may not find all of the required files or directories. In case of these CMake will throw an error in its output window (located at the bottom of the GUI) and set its field values, to not found constants. For example:





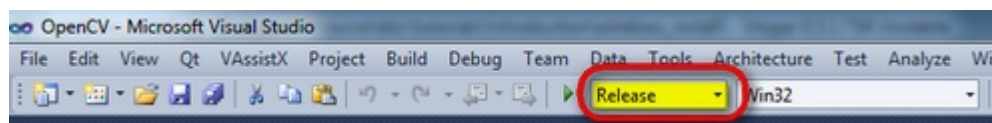
For these you need to manually set the queried directories or files path. After this press again the *Configure* button to see if the value entered by you was accepted or not. Do this until all entries are good and you cannot see errors in the field/value or the output part of the GUI. Now I want to emphasize an option that you will definitely love: *ENABLE* → *ENABLE\_SOLUTION\_FOLDERS*. OpenCV will create many-many projects and turning this option will make sure that they are categorized inside directories in the *Solution Explorer*. It is a must have feature, if you ask me.



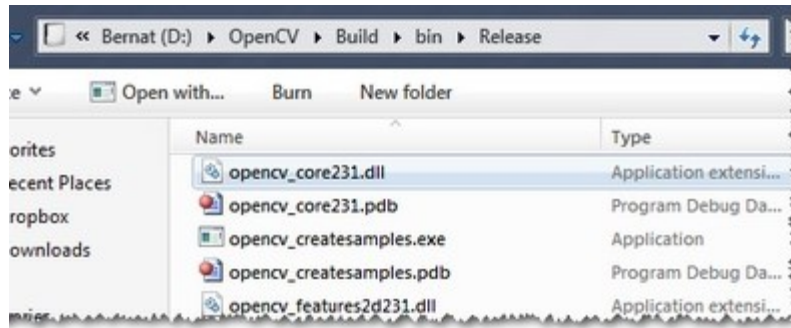
Furthermore, you need to select what part of OpenCV you want to build.

- *BUILD\_DOCS* -> Build the documentation of OpenCV (there will be a separate project for building the HTML and the PDF files).
- *BUILD\_EXAMPLES* -> OpenCV comes with many example applications from which you may learn most of the libraries capabilities. This will also come handy to easily try out if OpenCV is fully functional on your computer.
- *BUILD\_JAVA\_SUPPORT* -> This is a fresh addition to OpenCV, which slowly start to support the java language.
- *BUILD\_NEW\_PYTHON\_SUPPORT* -> Self-explanatory. Create the binaries to use OpenCV from the Python language.
- *BUILD\_PACKAGE* -> Build a project that will build an OpenCV installer. With this you can easily install your OpenCV flavor on other systems.
- *BUILD\_SHARED\_LIBS* -> With this you can control to build DLL files (when turned on) or static library files (\*.lib) otherwise.
- *BUILD\_TESTS* -> Each module of OpenCV has a test project assigned to it. Building these test projects is also a good way to try out, that the modules work just as expected on your system too.

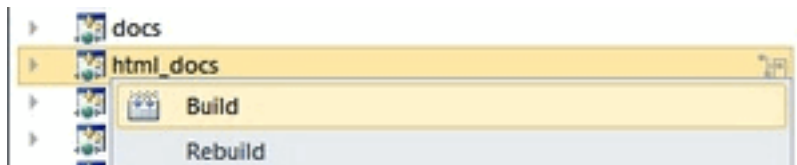
Press again the *Configure* button and ensure no errors are reported. If this is the case you can tell CMake to create the project files by pushing the *Generate* button. Go to the build directory and open the created **OpenCV** solution. Depending on just how much of the above options you have selected the solution may contain quite a lot of projects so be tolerant on the IDE at the startup. Now you need to build both the *Release* and the *Debug* binaries. Use the drop-down menu on your IDE to change to another of these after building for one of them.



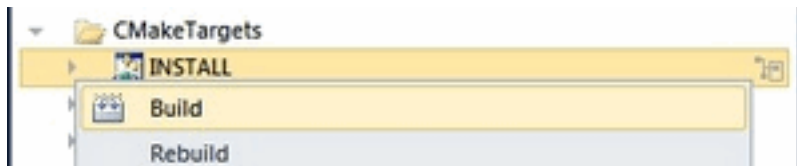
In the end you can observe the built binary files inside the bin directory:



For the documentation you need to explicitly issue the build commands on the *doc* project for the PDF files and on the *doc\_html* for the HTML ones. Each of these will call *Sphinx* to do all the hard work. You can find the generated documentation inside the *Build/Doc/\_html* for the HTML pages and within the *Build/Doc* the PDF manuals.

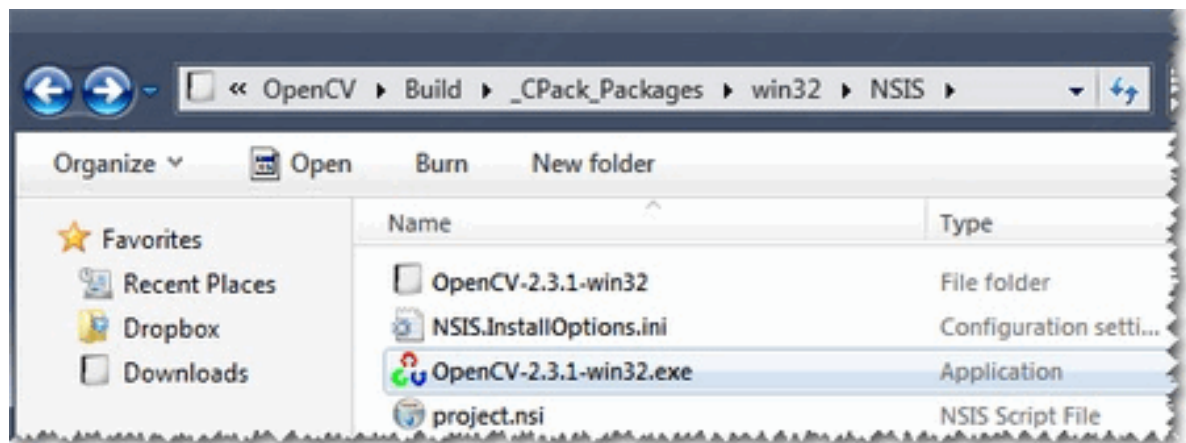


To collect the header and the binary files, that you will use during your own projects, into a separate directory (simillary to how the pre-built binaries ship) you need to explicitly build the *Install* project.



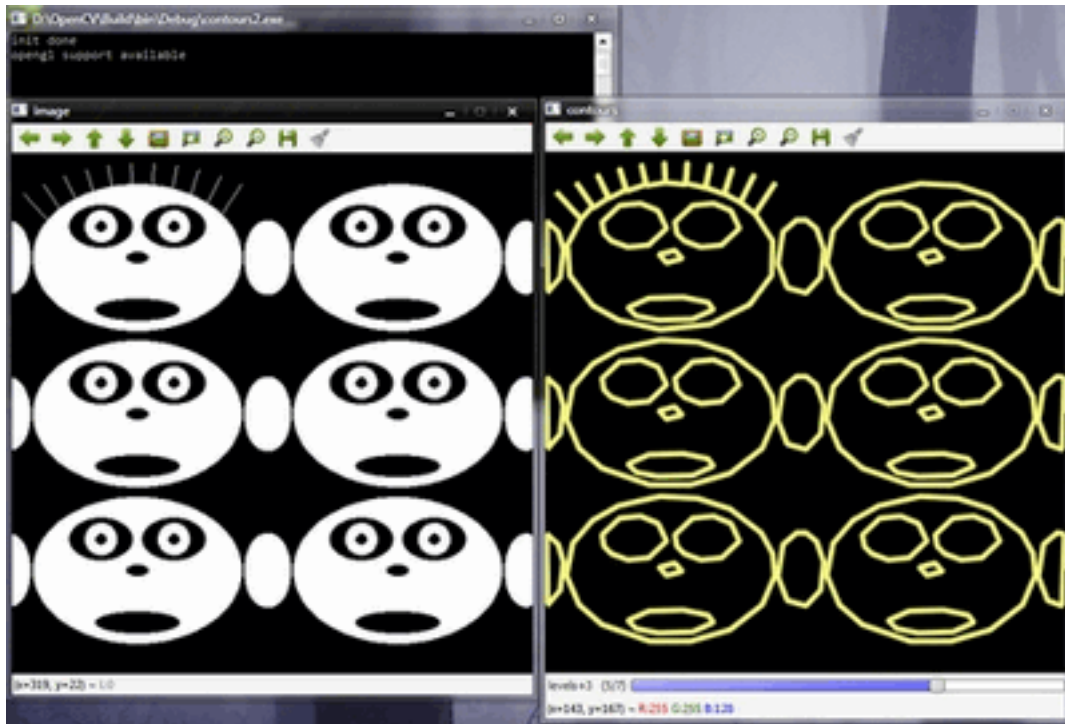
This will create an *install* directory inside the *Build* one collecting all the built binaries into a single place. Use this only after you built both the *Release* and *Debug* versions.

**Note:** To create an installer you need to install *NSIS*. Then just build the *Package* project to build the installer into the *Build/\_CPack\_Packages/win32/NSIS* folder. You can then use this to distribute OpenCV with your build settings on other systems.



To test your build just go into the *Build/bin/Debug* or *Build/bin/Release* directory and start a couple of applications like the *contours.exe*. If they run, you are done. Otherwise, something definitely went awfully

wrong. In this case you should contact us via our [user group](#). If everything is okay the `contours.exe` output should resemble the following image (if built with Qt support):



**Note:** If you use the GPU module (CUDA libraries) make sure you also upgrade to the latest drivers of your GPU. Error messages containing invalid entries in (or cannot find) the `nvcuda.dll` are caused mostly by old video card drivers. For testing the GPU (if built) run the `performance_gpu.exe` sample application.

## Set the OpenCV environment variable and add it to the systems path

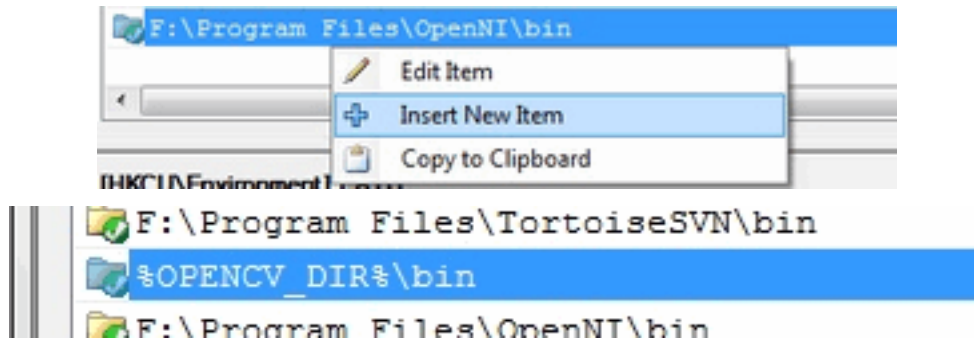
First we set an environment variable to make easier our work. This will hold the install directory of our OpenCV library that we use in our projects. Start up a command window and enter:

```
setx -m OPENCV_DIR D:\OpenCV\Build\Install
```

Here the directory is where you have your OpenCV binaries (*installed* or *built*). Inside this you should have folders like `bin` and `include`. The `-m` should be added if you wish to make the settings computer wise, instead of user wise.

If you built static libraries then you are done. Otherwise, you need to add the `bin` folders path to the systems path. This is cause you will use the OpenCV library in form of “*Dynamic-link libraries*” (also known as **DLL**). Inside these are stored all the algorithms and information the OpenCV library contains. The operating system will load them only on demand, during runtime. However, to do this he needs to know where they are. The systems **PATH** contains a list of folders where DLLs can be found. Add the OpenCV library path to this and the OS will know where to look if he ever needs the OpenCV binaries. Otherwise, you will need to copy the used DLLs right beside the applications executable file (`exe`) for the OS to find it, which is highly unpleasant if you work on many projects. To do this start up again the [Path Editor](#) and add the following new entry (right click in the application to bring up the menu):

```
%OPENCV_DIR%\bin
```

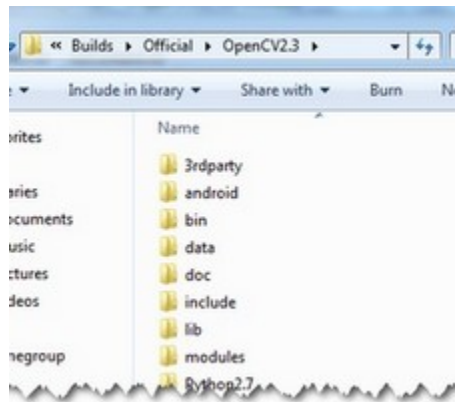


Save it to the registry and you are done. If you ever change the location of your install directories or want to try out your application with a different build all you will need to do is to update the `OPENCV_DIR` variable via the `setx` command inside a command window.

Now you can continue reading the tutorials with the *How to build applications with OpenCV inside the Microsoft Visual Studio* section. There you will find out how to use the OpenCV library in your own projects with the help of the Microsoft Visual Studio IDE.

## 1.5 How to build applications with OpenCV inside the *Microsoft Visual Studio*

Everything I describe here will apply to the C/C++ interface of OpenCV. I start out from the assumption that you have read and completed with success the *Installation in Windows* tutorial. Therefore, before you go any further make sure you have an OpenCV directory that contains the OpenCV header files plus binaries and you have set the environment variables as *described here*.



The OpenCV libraries, distributed by us, on the Microsoft Windows operating system are in a **Dynamic Linked Libraries (DLL)**. These have the advantage that all the content of the library are loaded only at runtime, on demand, and that countless programs may use the same library file. This means that if you have ten applications using the OpenCV library, no need to have around a version for each one of them. Of course you need to have the *dll* of the OpenCV on all systems where you want to run your application.

Another approach is to use static libraries that have *lib* extensions. You may build these by using our source files as described in the *Installation in Windows* tutorial. When you use this the library will be built-in inside your *exe* file. So there is no chance that the user deletes them, for some reason. As a drawback your application will be larger one and as, it will take more time to load it during its startup.

To build an application with OpenCV you need to do two things:

- *Tell* to the compiler how the OpenCV library *looks*. You do this by *showing* it the header files.

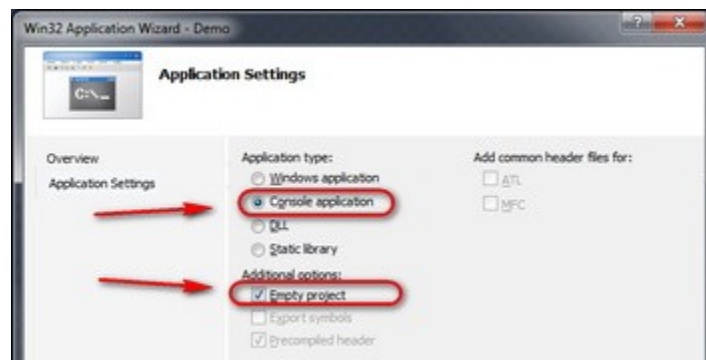
- *Tell* to the linker from where to get the functions or data structures of OpenCV, when they are needed.

If you use the *lib* system you must set the path where the library files are and specify in which one of them to look. During the build the linker will look into these libraries and add the definitions and implementation of all *used* functions and data structures to the executable file.

If you use the *DLL* system you must again specify all this, however now for a different reason. This is a Microsoft OS specific stuff. It seems that the linker needs to know that where in the DLL to search for the data structure or function at the runtime. This information is stored inside *lib* files. Nevertheless, they aren't static libraries. They are so called import libraries. This is why when you make some *DLLs* in Windows you will also end up with some *lib* extension libraries. The good part is that at runtime only the *DLL* is required.

To pass on all this information to the Visual Studio IDE you can either do it globally (so all your future projects will get these information) or locally (so only for you current project). The advantage of the global one is that you only need to do it once; however, it may be undesirable to clump all your projects all the time with all these information. In case of the global one how you do it depends on the Microsoft Visual Studio you use. There is a **2008 and previous versions** and a **2010 way** of doing it. Inside the global section of this tutorial I'll show what the main differences are.

The base item of a project in Visual Studio is a solution. A solution may contain multiple projects. Projects are the building blocks of an application. Every project will realize something and you will have a main project in which you can put together this project puzzle. In case of the many simple applications (like many of the tutorials will be) you do not need to break down the application into modules. In these cases your main project will be the only existing one. Now go create a new solution inside Visual studio by going through the *File* → *New* → *Project* menu selection. Choose *Win32 Console Application* as type. Enter its name and select the path where to create it. Then in the upcoming dialog make sure you create an empty project.



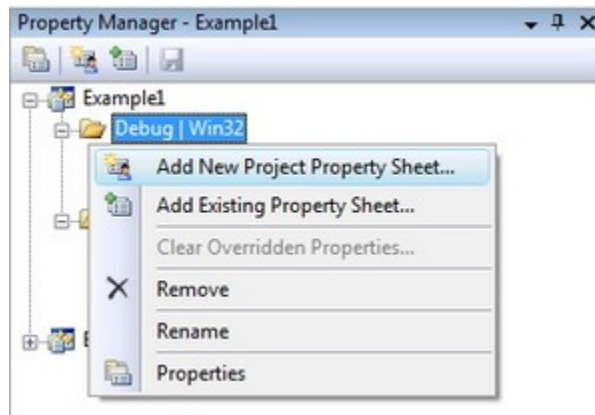
## The *local* method

Every project is built separately from the others. Due to this every project has its own rule package. Inside this rule packages are stored all the information the *IDE* needs to know to build your project. For any application there are at least two build modes: a *Release* and a *Debug* one. The *Debug* has many features that exist so you can find and resolve easier bugs inside your application. In contrast the *Release* is an optimized version, where the goal is to make the application run as fast as possible or to be as small as possible. You may figure that these modes also require different rules to use during build. Therefore, there exist different rule packages for each of your build modes. These rule packages are called inside the IDE as *project properties* and you can view and modify them by using the *Property Manger*. You can bring up this with *View* → *Property Pages*. Expand it and you can see the existing rule packages (called *Property Sheets*).



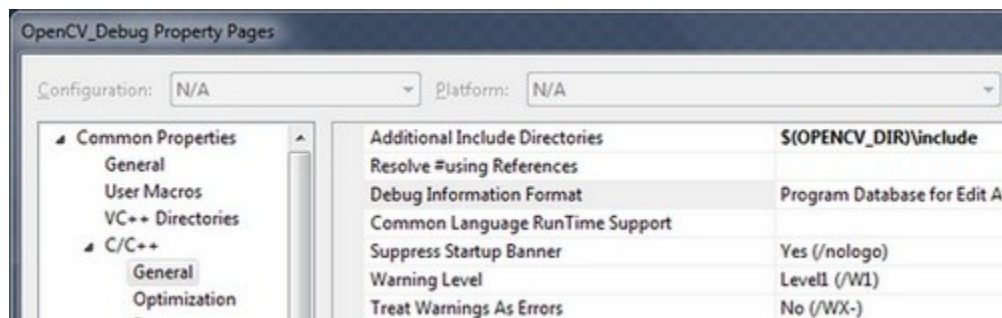


The really useful stuff of these is that you may create a rule package *once* and you can later just add it to your new projects. Create it once and reuse it later. We want to create a new *Property Sheet* that will contain all the rules that the compiler and linker needs to know. Of course we will need a separate one for the Debug and the Release Builds. Start up with the Debug one as shown in the image below:



Use for example the *OpenCV\_Debug* name. Then by selecting the sheet *Right Click* → *Properties*. In the following I will show to set the OpenCV rules locally, as I find unnecessary to pollute projects with custom rules that I do not use it. Go the C++ groups General entry and under the “*Additional Include Directories*” add the path to your OpenCV include.

```
$(OPENCV_DIR)\include
```

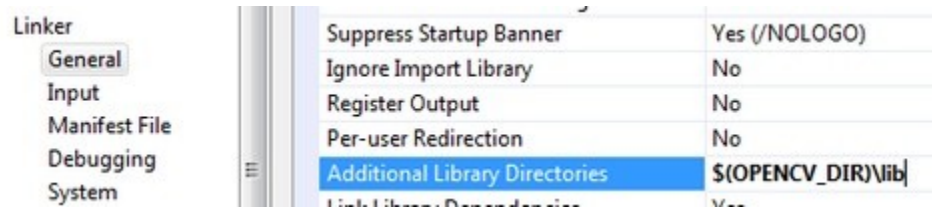


When adding third party libraries settings it is generally a good idea to use the power behind the environment variables. The full location of the OpenCV library may change on each system. Moreover, you may even end up yourself with moving the install directory for some reason. If you would give explicit paths inside your property sheet your project will end up not working when you pass it further to someone else who has a different OpenCV install path. Moreover, fixing this would require to manually modifying every explicit path. A more elegant solution is to use the environment variables. Anything that you put inside a parenthesis started with a dollar sign will be replaced at runtime with the current environment variables value. Here comes in play the environment variable setting we already made in our

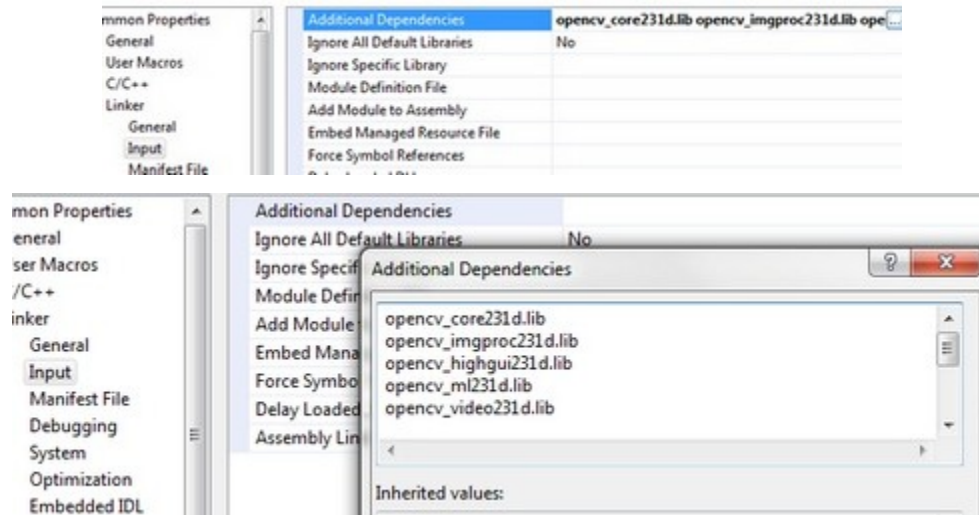
*previous tutorial.*

Next go to the *Linker* → *General* and under the “*Additional Library Directories*” add the `libs` directory:

```
$(OPENCV_DIR)\libs
```



Then you need to specify the libraries in which the linker should look into. To do this go to the *Linker* → *Input* and under the “*Additional Dependencies*” entry add the name of all modules which you want to use:



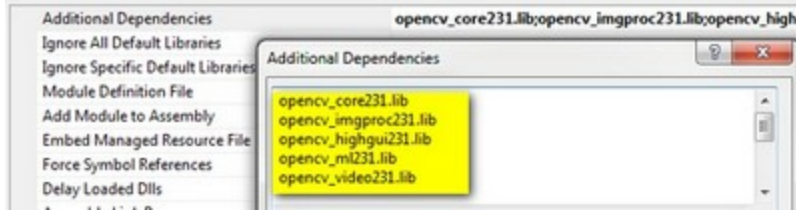
The names of the libraries are as follow:

```
opencv_(The Name of the module)(The version Number of the library you use)d.lib
```

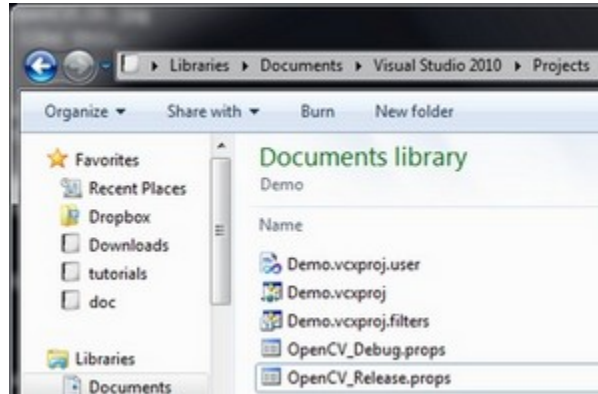
A full list, for the currently latest trunk version would contain:

```
opencv_core231d.lib
opencv_imgproc231d.lib
opencv_highgui231d.lib
opencv_ml231d.lib
opencv_video231d.lib
opencv_features2d231d.lib
opencv_calib3d231d.lib
opencv_objdetect231d.lib
opencv_contrib231d.lib
opencv_legacy231d.lib
opencv_flann231d.lib
```

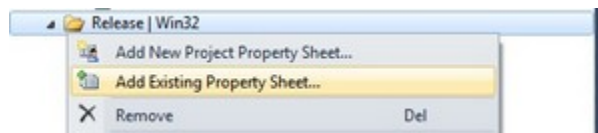
The letter *d* at the end just indicates that these are the libraries required for the debug. Now click ok to save and do the same with a new property inside the Release rule section. Make sure to omit the *d* letters from the library names and to save the property sheets with the save icon above them.



You can find your property sheets inside your projects directory. At this point it is a wise decision to back them up into some special directory, to always have them at hand in the future, whenever you create an OpenCV project. Note that for Visual Studio 2010 the file extension is *props*, while for 2008 this is *vsprops*.



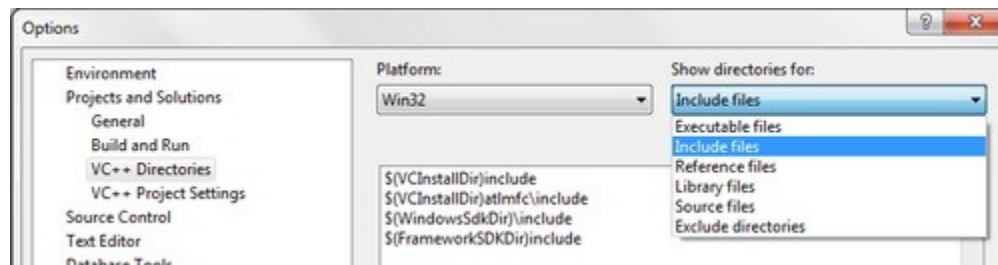
Next time when you make a new OpenCV project just use the “Add Existing Property Sheet...” menu entry inside the Property Manager to easily add the OpenCV build rules.



## The *global* method

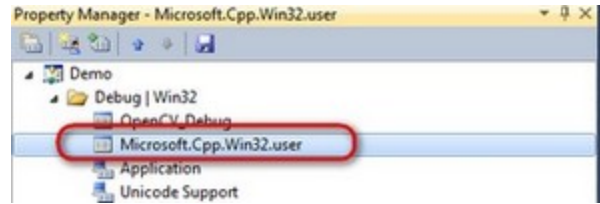
In case you find to troublesome to add the property pages to each and every one of your projects you can also add this rules to a “*global property page*”. However, this applies only to the additional include and library directories. The name of the libraries to use you still need to specify manually by using for instance: a Property page.

In Visual Studio 2008 you can find this under the: *Tools* → *Options* → *Projects and Solutions* → *VC++ Directories*.



In Visual Studio 2010 this has been moved to a global property sheet which is automatically added to every project you create:





The process is the same as described in case of the local approach. Just add the include directories by using the environment variable `OPENCV_DIR`.

## Test it!

Now to try this out download our little test source code or get it from the sample code folder of the OpenCV sources. Add this to your project and build it. Here's its content:

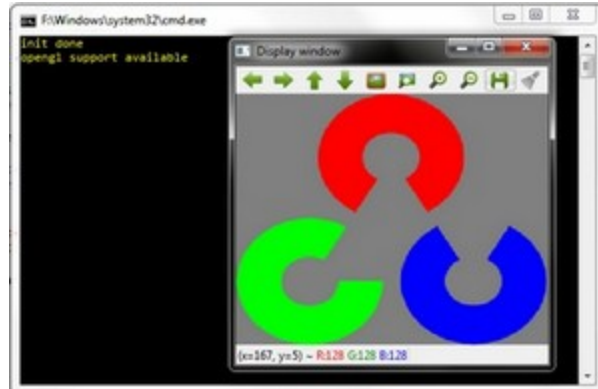
```

1  #include <opencv2/core/core.hpp>
2  #include <opencv2/highgui/highgui.hpp>
3  #include <iostream>
4
5  using namespace cv;
6
7  int main( int argc, char** argv )
8  {
9      Mat image;
10     image = imread("opencv-logo.png"); // Read the file
11
12     if(! image.data )                 // Check for invalid input
13     {
14         std::cout << "Could not open or find the image" << std::endl ;
15         return -1;
16     }
17
18     namedWindow( "Display window", CV_WINDOW_FREERATIO );// Create a window for display.
19     imshow( "Display window", image );           // Show our image inside it.
20
21     waitKey(0);                                 // Wait for a keystroke in the window
22
23     return 0;
24 }

```

You can start a Visual Studio build from two places. Either inside from the *IDE* (keyboard combination: `Control-F5`) or by navigating to your build directory and start the application with a double click. The catch is that these two **aren't** the same. When you start it from the *IDE* its current working directory is the projects directory, while otherwise it is the folder where the application file currently is (so usually your build directory). Moreover, in case of starting from the *IDE* the console window will not close once finished. It will wait for a keystroke of yours.

This is important to remember when you code inside the code open and save commands. You're resources will be saved ( and queried for at opening!!!) relatively to your working directory. This is unless you give a full, explicit path as parameter for the I/O functions. In the code above we open this `opencv logo`. Before starting up the application make sure you place the image file in your current working directory. Modify the image file name inside the code to try it out on other images too. Run it and voilà:

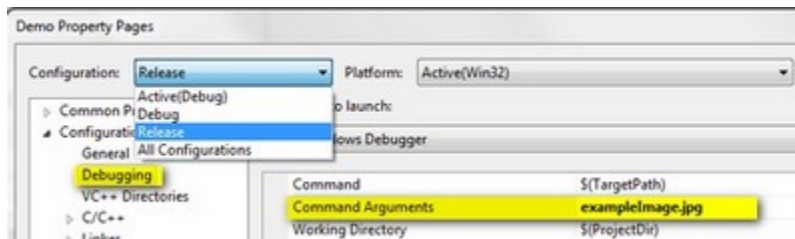


## Command line arguments with Visual Studio

Throughout some of our future tutorials you'll see that the programs main input method will be by giving a runtime argument. To do this you can just start up a command windows (cmd + Enter in the start menu), navigate to your executable file and start it with an argument. So for example in case of my upper project this would look like:

- 1 D:
- 2 CD OpenCV\MySolutionName\Release
- 3 MySolutionName.exe exampleImage.jpg

Here I first changed my drive (if your project isn't on the OS local drive), navigated to my project and start it with an example image argument. While under Linux system it is common to fiddle around with the console window on the Microsoft Windows many people come to use it almost never. Besides, adding the same argument again and again while you are testing your application is, somewhat, a cumbersome task. Luckily, in the Visual Studio there is a menu to automate all this:



Specify here the name of the inputs and while you start your application from the Visual Studio environment you have automatic argument passing. In the next introductory tutorial you'll see an in-depth explanation of the upper source code: *Load and Display an Image*.

## 1.6 Using Android binary package with Eclipse

This tutorial was tested using Ubuntu 10.04 and Windows 7 SP1 operating systems. Nevertheless, it should also work on any other OSes supported by Android SDK (including Mac OS X). If you encounter errors after following the steps described here feel free to contact us via *android-opencv* discussion group <https://groups.google.com/group/android-opencv/> and we will try to fix your problem.

### Setup environment to start Android Development

You need the following tools to be installed:

### 1. Sun JDK 6

Visit <http://www.oracle.com/technetwork/java/javase/downloads/index.html> and download installer for your OS.

Here is detailed JDK installation guide for Ubuntu and Mac OS: <http://source.android.com/source/initializing.html> (only JDK sections are applicable for OpenCV)

---

**Note:** OpenJDK is not usable for Android development because Android SDK supports only Sun JDK.

---

### 2. Android SDK

Get the latest Android SDK from <http://developer.android.com/sdk/index.html>

Here is Google's install guide for SDK <http://developer.android.com/sdk/installing.html>

---

**Note:** If you choose SDK packed into Windows installer then you should have installed 32-bit JRE. It does not needed for Android development but installer is x86 application and requires 32-bit Java runtime.

---

**Note:** If you are running x64 version of Ubuntu Linux then you need ia32 shared libraries for use on amd64 and ia64 systems installed. You can install them with following command:

```
sudo apt-get install ia32-libs
```

For Red Hat based systems the following command might be helpful:

```
yum install libXtst.i386
```

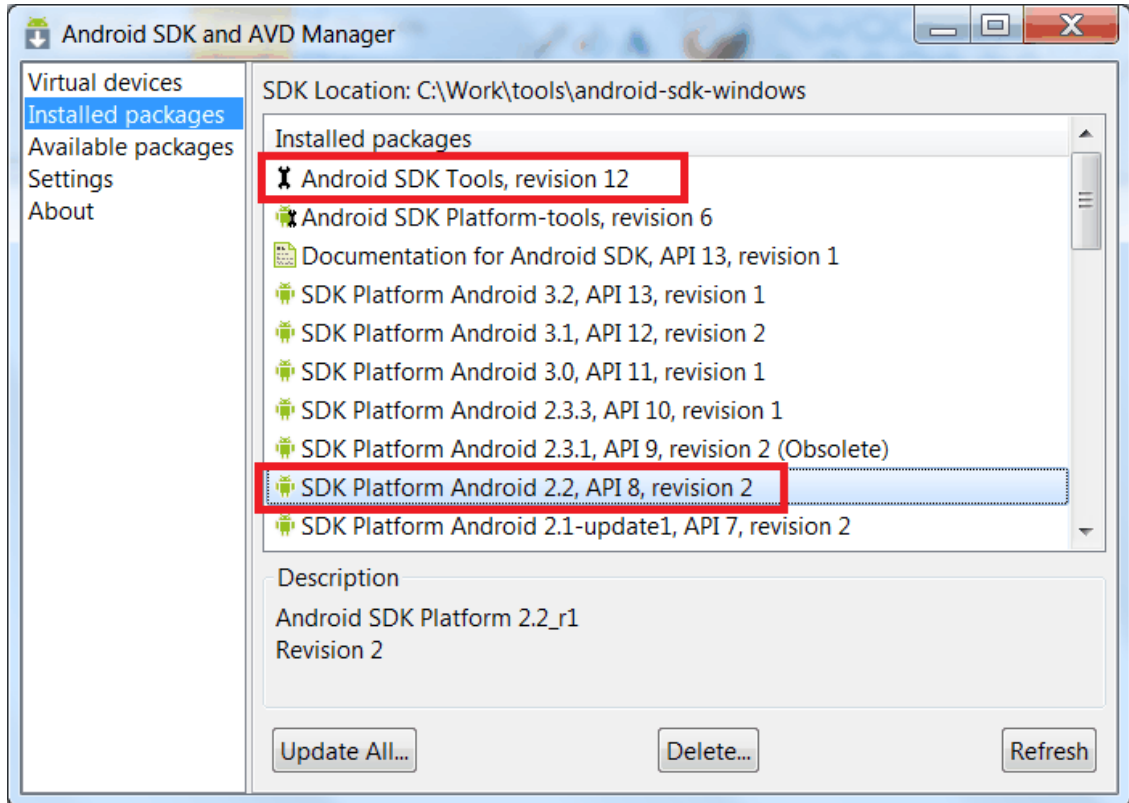
---

### 3. Android SDK components

You need the following SDK components to be installed:

- *Android SDK Tools, revision 12* or newer  
Older revisions should also work but they are not recommended.
- *SDK Platform Android 2.2, API 8, revision 2* (also known as *android-8*)

This is minimal platform supported by OpenCV Java API. And it is set as default for OpenCV distribution. It is possible to use platform having higher number with OpenCV package but it requires to edit OpenCV project settings.



See [Adding SDK Components](#) for help with installing/updating SDK components.

#### 4. Eclipse IDE

Check the [Android SDK System Requirements](#) document for a list of Eclipse versions that are compatible with the Android SDK. For OpenCV 2.3.1 we recommend Eclipse 3.7 (Indigo) or Eclipse 3.6 (Helios). They work well for OpenCV both under Windows and Linux.

If you have no Eclipse installed, you can download it from this location:

<http://www.eclipse.org/downloads/>

#### 5. ADT plugin for Eclipse

Assuming that you have Eclipse IDE installed, as described above, follow these steps to download and install the ADT plugin:

- (a) Start Eclipse, then select **Help > Install New Software...**
- (b) Click **Add**, in the top-right corner.
- (c) In the Add Repository dialog that appears, enter “ADT Plugin” for the Name and the following URL for the Location:

<https://dl-ssl.google.com/android/eclipse/>

- (d) Click **OK**

---

**Note:** If you have trouble acquiring the plugin, try using “http” in the Location URL, instead of “https” (https is preferred for security reasons).

---

- (e) In the Available Software dialog, select the checkbox next to Developer Tools and click **Next**.

- (f) In the next window, you'll see a list of the tools to be downloaded. Click **Next**.
- (g) Read and accept the license agreements, then click **Finish**.

---

**Note:** If you get a security warning saying that the authenticity or validity of the software can't be established, click **OK**.

---

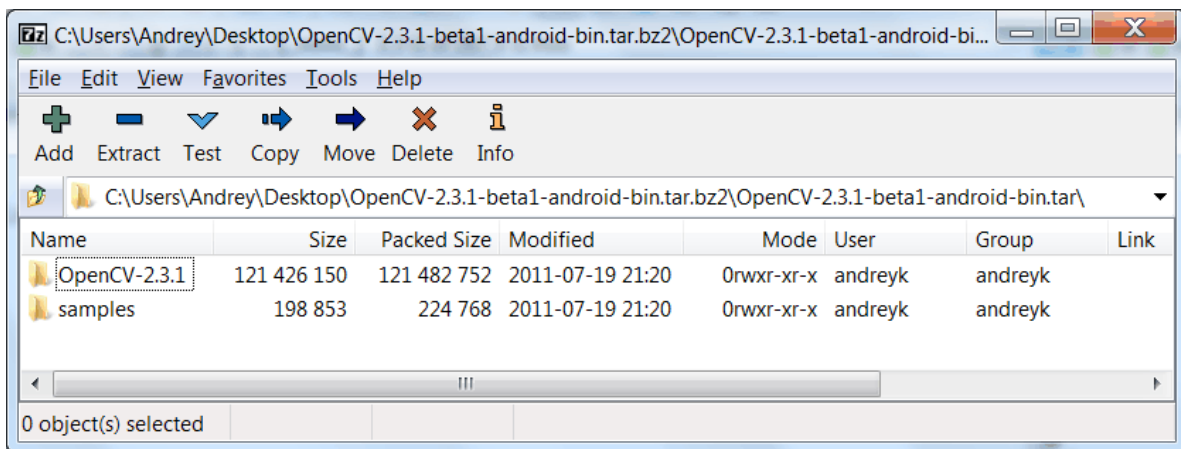
- (h) When the installation completes, restart Eclipse.

These steps are copied from <http://developer.android.com/sdk/eclipse-adt.html#downloading> . Please, visit that page if you have any troubles with ADT plugin installation.

## Get the OpenCV package for Android development

1. Go to the <http://sourceforge.net/projects/opencvlibrary/files/opencv-android/> and download the latest available version. Currently it is `OpenCV-2.3.1-beta1-android-bin.tar.bz2`
2. Create new folder for Android+OpenCV development
3. Unpack the OpenCV package into that dir.

You can unpack it using any popular archiver (for example with 7-Zip):



On Unix you also can unpack using the following command:

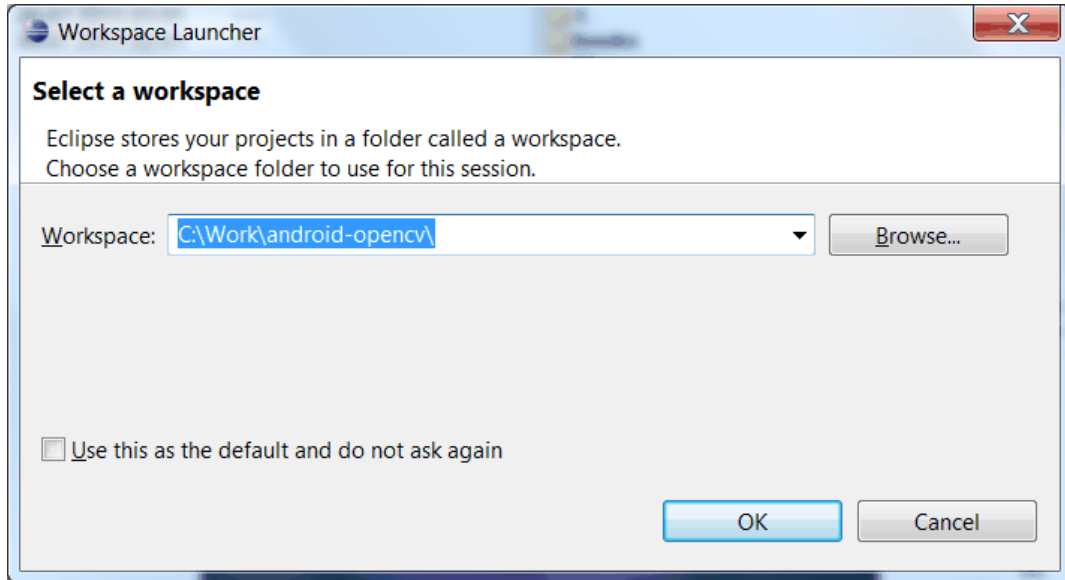
```
tar -jxvf ~/Downloads/0penCV-2.3.1-beta1-android-bin.tar.bz2
```

For this tutorial I have unpacked OpenCV into the `C:\Work\android-opencv\` directory.

## Open OpenCV library and samples in Eclipse

1. Start the *Eclipse* and choose your workspace location.

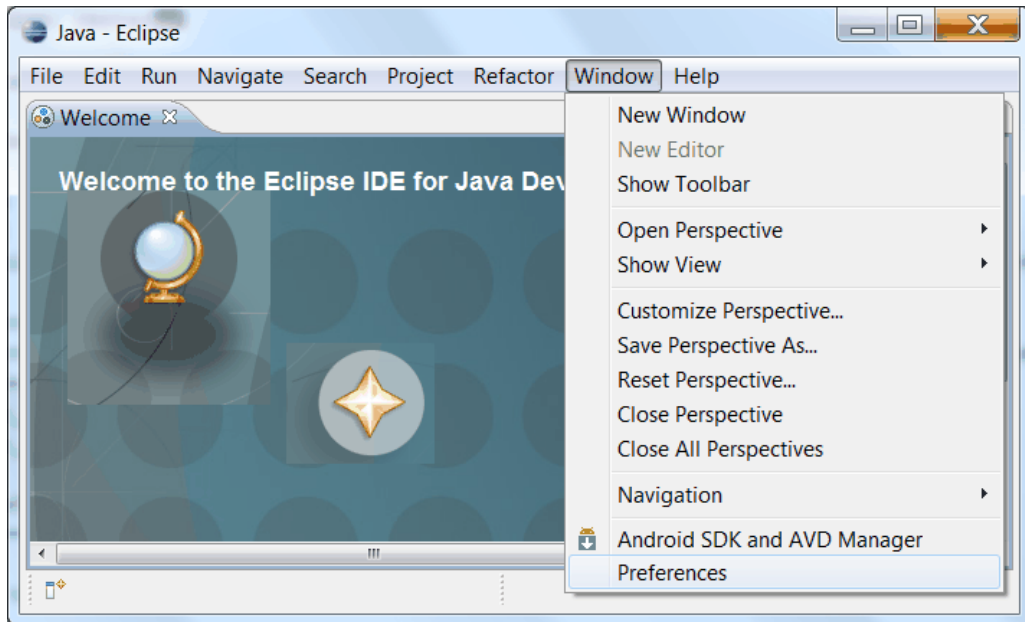
I recommend to start familiarising yourself with OpenCV for Android from new clean workspase. So I have choosen my OpenCV package directory for new workspace:



2. Configure your ADT plugin

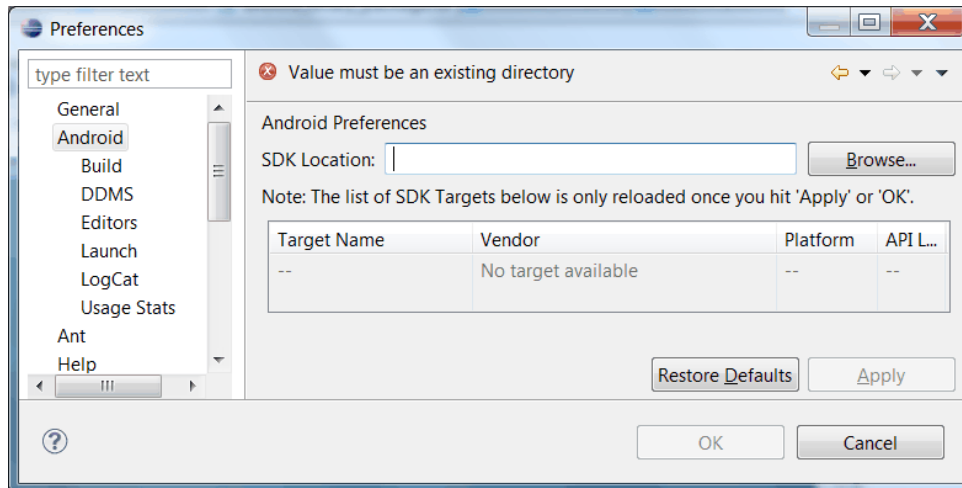
Once you have created new workspace you have to point the ADT plugin to the Android SDK directory. This setting is stored in workspace metadata so this step is required each time when you are creating new workspace for Android development. See [Configuring the ADT Plugin](#) document for the original instructions from *Google*.

- Select **Window > Preferences...** to open the Preferences panel (Mac OS X: **Eclipse > Preferences**):

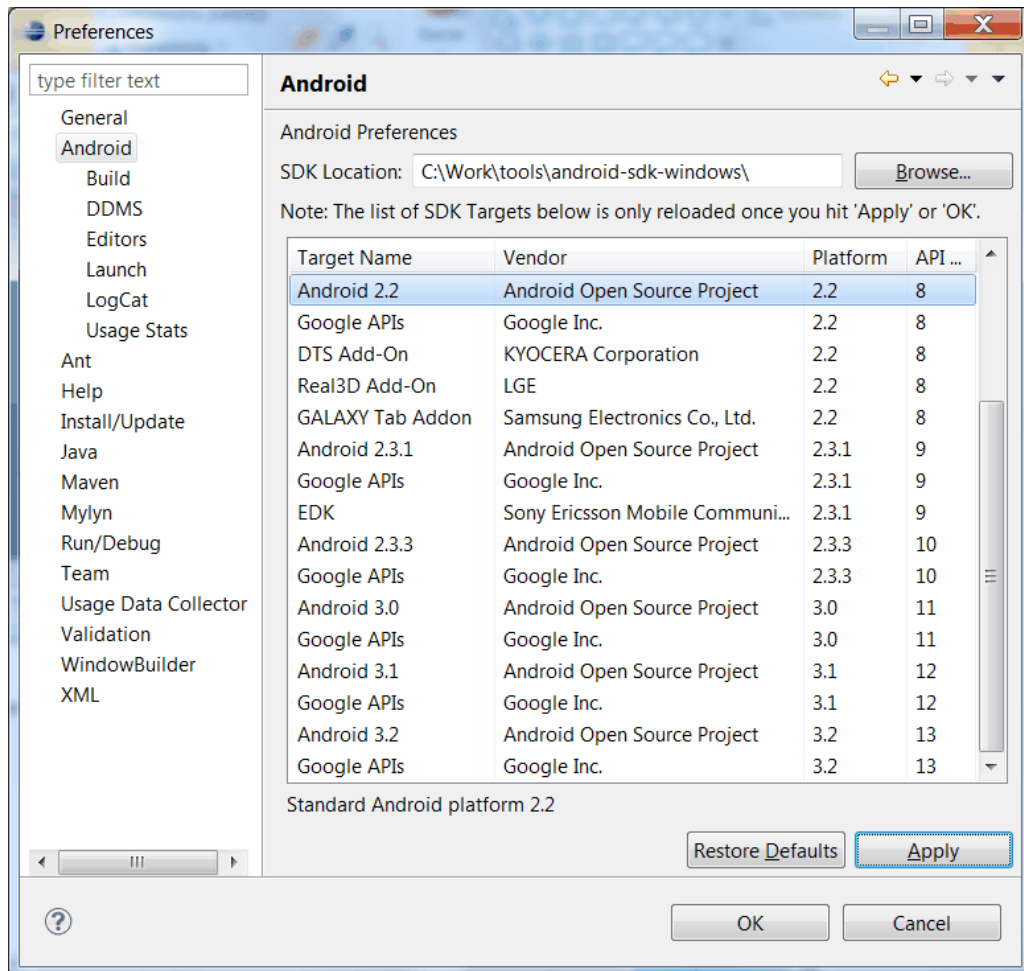


- Select **Android** from the left panel.

You may see a dialog asking whether you want to send usage statistics to Google. If so, make your choice and click **Proceed**. You cannot continue with this procedure until you click **Proceed**.



- For the SDK Location in the main panel, click **Browse...** and locate your Android SDK directory.
- Click **Apply** button at the bottom right corner of main panel:



- Click **OK** to close preferences dialog.

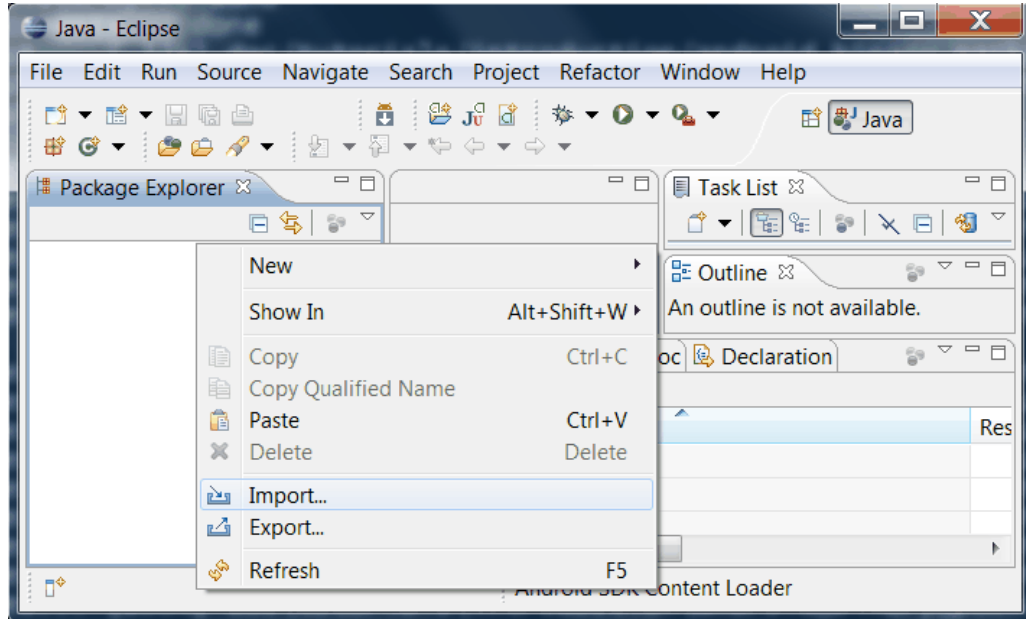
### 3. Import OpenCV and samples into workspace.

OpenCV library is packed as ready-for-use [Android Library Project](#). You can simply reference it in your

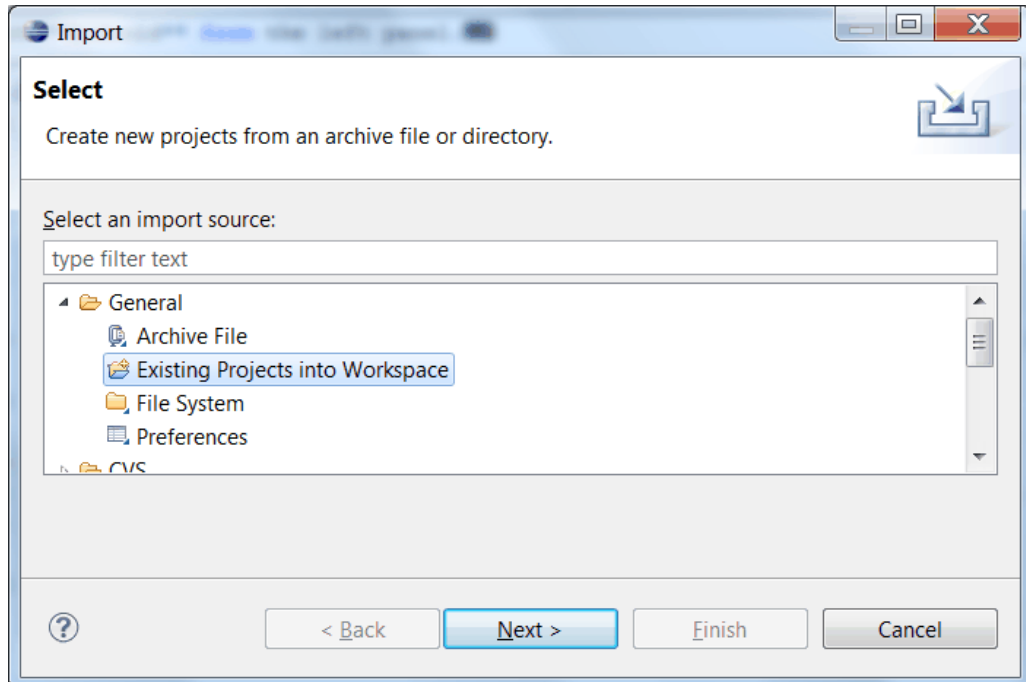
projects.

Each sample included into OpenCV-2.3.1-beta1-android-bin.tar.bz2 is a regular Android project that already reference OpenCV library. Follow next steps to import OpenCV and samples into workspace:

- Right click on the *Package Explorer* window and choose **Import...** option from context menu:

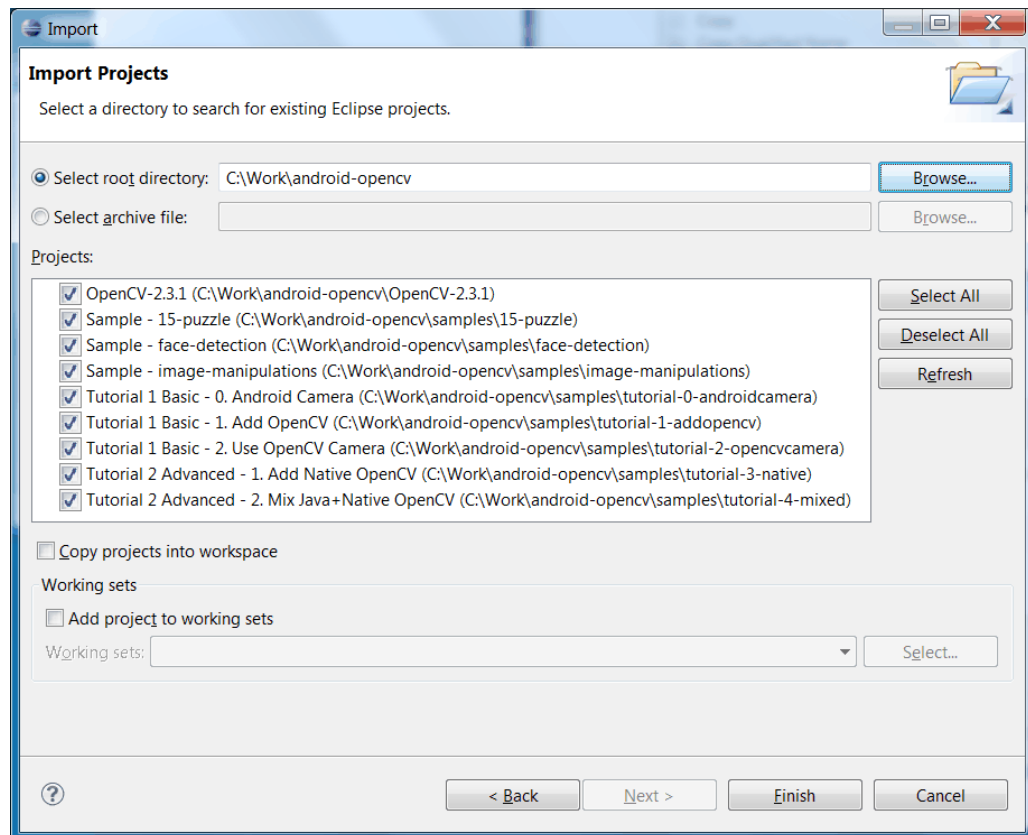


- In the main panel select **General > Existing Projects into Workspace** and press **Next** button:



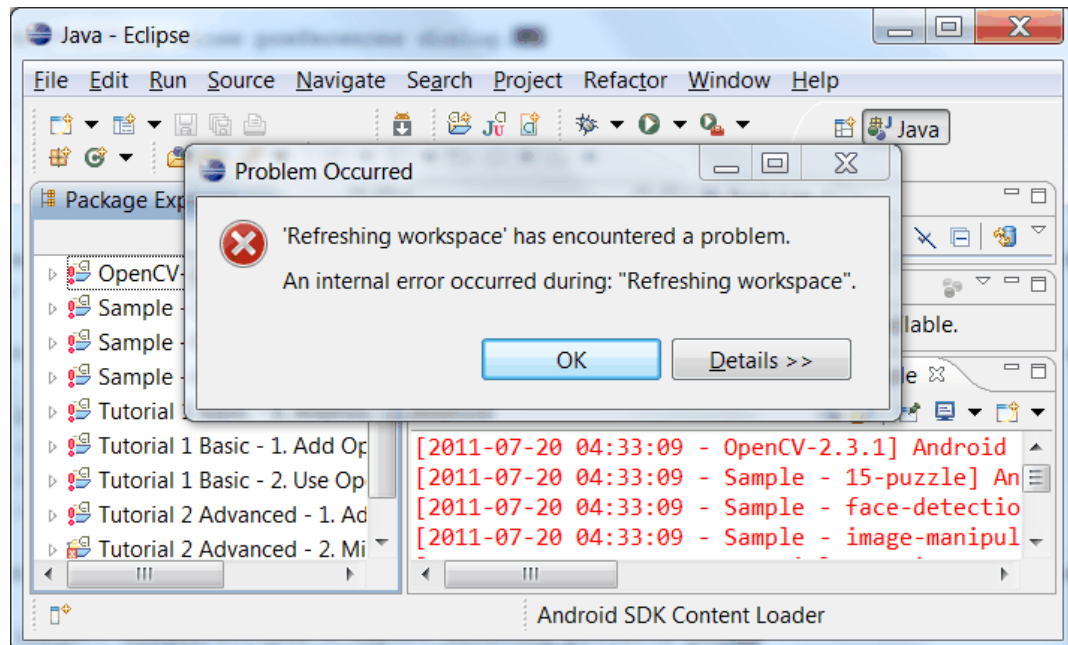
- For the *Select root directory* in the main panel locate your OpenCV package folder. (If you have created workspace in the package directory then just click **Browse...** button and instantly close directory choosing dialog with **OK** button!) Eclipse should automatically locate OpenCV library and samples:





- Click **Finish** button to complete the import operation.

After clicking **Finish** button Eclipse will load all selected projects into workspace. And... will indicate numerous errors:

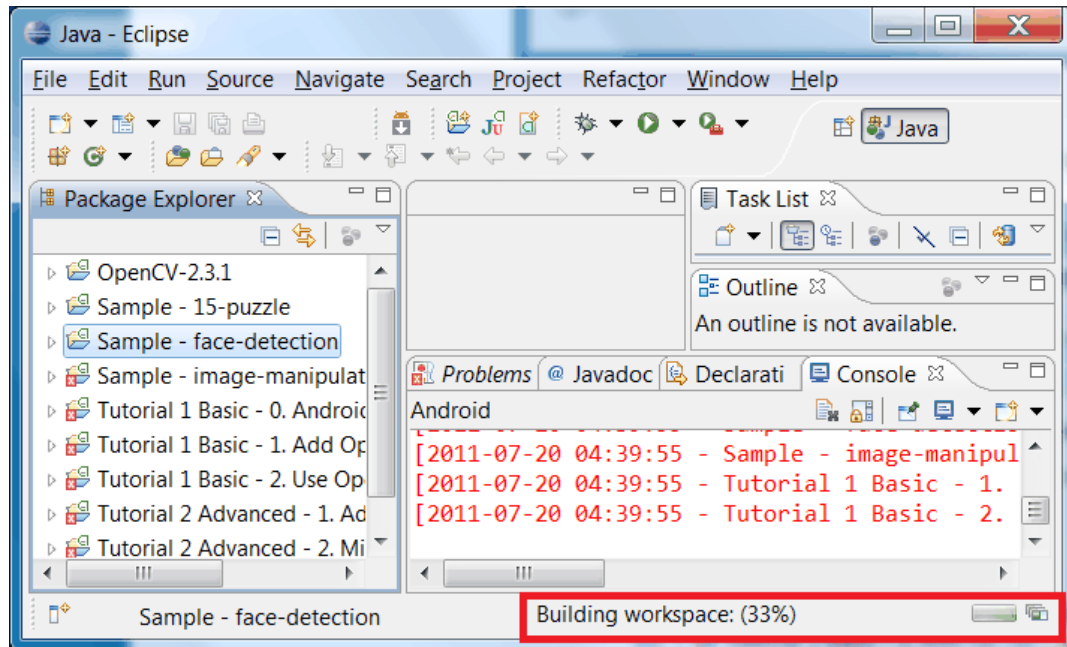


However **all these errors are only false-alarms!**

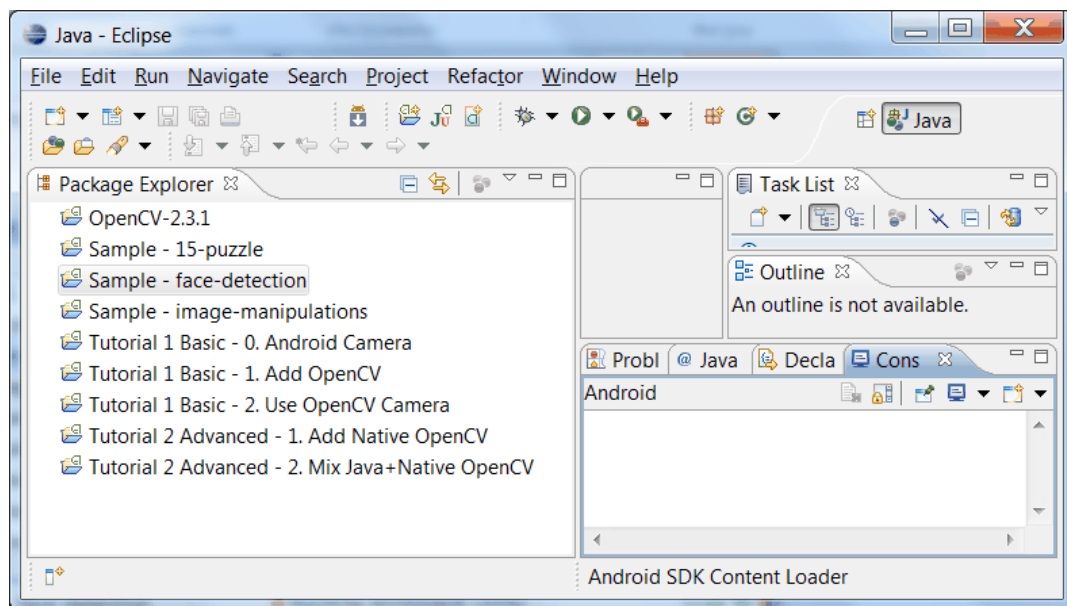
To help Eclipse to understand that there are no any errors choose OpenCV library in *Package Explorer* (left

mouse click) and press **F5** button on your keyboard. Then choose any sample (except first samples in *Tutorial Base* and *Tutorial Advanced*) and also press **F5**.

After this manipulation Eclipse will rebuild your workspace and error icons will disappear one by one:



Once Eclipse completes build you will have clean workspace without any build error:



**Note:** If you are importing only OpenCV library without samples then instead of second refresh command (**F5**) you might need to make **Android Tools > Fix Project Properties** from project context menu.

## Running OpenCV Samples

At this point you should be able to build and run all samples except two from Advanced tutorial (these samples require NDK to build working applications).

Also I want to note that only Tutorial 1 Basic - 0. Android Camera and Tutorial 1 Basic - 1. Add OpenCV samples are able to run on Emulator from Android SDK. Other samples are using OpenCV Native Camera which is supported only for ARM v7 CPUs.

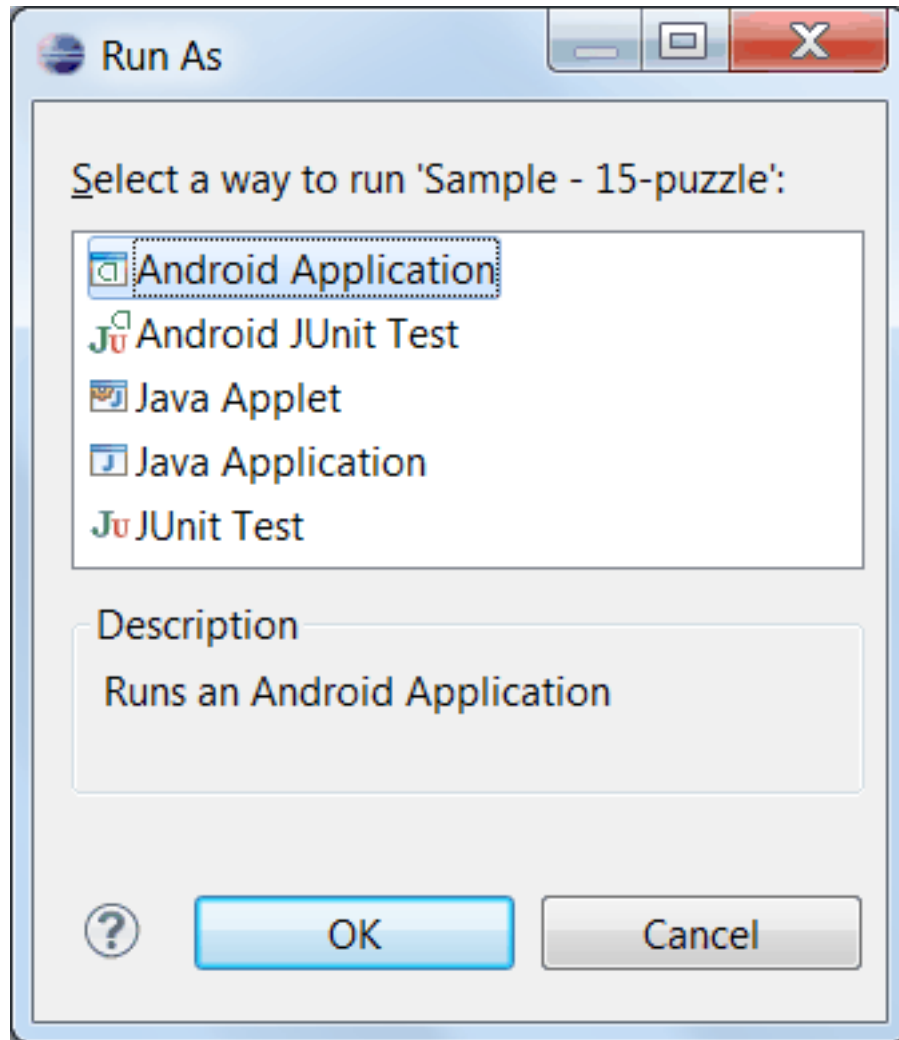
---

**Note:** Latest *Android SDK tools, revision 12* can run ARM v7 OS images but *Google* does not provide such images within SDK.

---

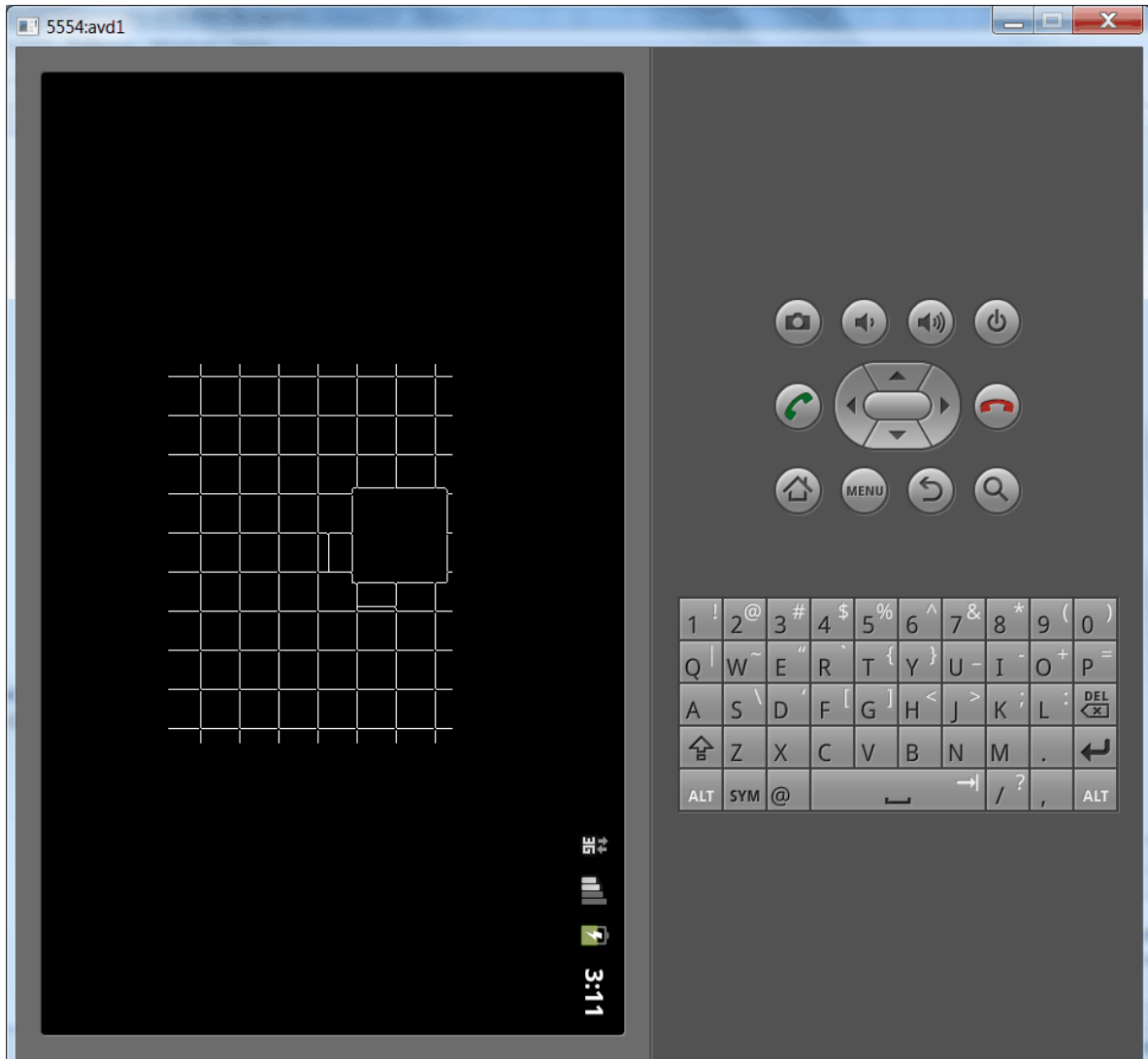
Well, running samples from Eclipse is very simple:

- Connect your device with adb tool from Android SDK or create Emulator with camera support.
  - See [Managing Virtual Devices](#) document for help with Android Emulator.
  - See [Using Hardware Devices](#) for help with physical devices.
- Select project you want to start in *Package Explorer* and just press **Ctrl + F11** or select option **Run > Run** from main menu, or click **Run** button on the toolbar.
- On first run Eclipse will ask you how to run your application:



- Select the *Android Application* option and click **OK** button. Eclipse will install and run the sample.

Here is sample Tutorial 1 Basic - 1. Add OpenCV detecting edges using Canny algorithm from OpenCV:



## 1.7 Load and Display an Image

### Goal

In this tutorial you will learn how to:

- Load an image (using `imread`)
- Create a named OpenCV window (using `namedWindow`)
- Display an image in an OpenCV window (using `imshow`)

### Source Code

Download the source code from [here](#) or look it up in our library at [samples/cpp/tutorial\\_code/introduction/display\\_image/display\\_image.cpp](#).

```
1  #include <opencv2/core/core.hpp>
2  #include <opencv2/highgui/highgui.hpp>
3  #include <iostream>
4
5  using namespace cv;
6  using namespace std;
7
8  int main( int argc, char** argv )
9  {
10     if( argc != 2)
11     {
12         cout <<" Usage: display_image ImageToLoadAndDisplay" << endl;
13         return -1;
14     }
15
16     Mat image;
17     image = imread(argv[1], CV_LOAD_IMAGE_COLOR); // Read the file
18
19     if(! image.data ) // Check for invalid input
20     {
21         cout << "Could not open or find the image" << std::endl ;
22         return -1;
23     }
24
25     namedWindow( "Display window", CV_WINDOW_AUTOSIZE );// Create a window for display.
26     imshow( "Display window", image ); // Show our image inside it.
27
28     waitKey(0); // Wait for a keystroke in the window
29     return 0;
30 }
```

## Explanation

In OpenCV 2 we have multiple modules. Each one takes care of a different area or approach towards image processing. You could already observe this in the structure of the user guide of these tutorials itself. Before you use any of them you first need to include the header files where the content of each individual module is declared.

You'll almost always end up using the:

- *core* section, as here are defined the basic building blocks of the library
- *highgui* module, as this contains the functions for input and output operations

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

We also include the *iostream* to facilitate console line output and input. To avoid data structure and function name conflicts with other libraries, OpenCV has its own namespace: *cv*. To avoid the need appending prior each of these the *cv::* keyword you can import the namespace in the whole file by using the lines:

```
using namespace cv;
using namespace std;
```

This is true for the STL library too (used for console I/O). Now, let's analyze the *main* function. We start up assuring that we acquire a valid image name argument from the command line.

```

if( argc != 2)
{
    cout <<" Usage: display_image ImageToLoadAndDisplay" << endl;
    return -1;
}

```

Then create a *Mat* object that will store the data of the loaded image.

```
Mat image;
```

Now we call the `imread` function which loads the image name specified by the first argument (`argv[1]`). The second argument specifies the format in what we want the image. This may be:

- `CV_LOAD_IMAGE_UNCHANGED` (<0) loads the image as is (including the alpha channel if present)
- `CV_LOAD_IMAGE_GRAYSCALE` ( 0) loads the image as an intensity one
- `CV_LOAD_IMAGE_COLOR` (>0) loads the image in the RGB format

```
image = imread(argv[1], CV_LOAD_IMAGE_COLOR); // Read the file
```

---

**Note:** OpenCV offers support for the image formats Windows bitmap (bmp), portable image formats (pbm, pgm, ppm) and Sun raster (sr, ras). With help of plugins (you need to specify to use them if you build yourself the library, nevertheless in the packages we ship present by default) you may also load image formats like JPEG (jpeg, jpg, jpe), JPEG 2000 (jp2 - codenamed in the CMake as Jasper), TIFF files (tiff, tif) and portable network graphics (png). Furthermore, OpenEXR is also a possibility.

---

After checking that the image data was loaded correctly, we want to display our image, so we create an OpenCV window using the `namedWindow` function. These are automatically managed by OpenCV once you create them. For this you need to specify its name and how it should handle the change of the image it contains from a size point of view. It may be:

- `CV_WINDOW_AUTOSIZE` is the only supported one if you do not use the Qt backend. In this case the window size will take up the size of the image it shows. No resize permitted!
- `CV_WINDOW_NORMAL` on Qt you may use this to allow window resize. The image will resize itself according to the current window size. By using the `|` operator you also need to specify if you would like the image to keep its aspect ratio (`CV_WINDOW_KEEPRATIO`) or not (`CV_WINDOW_FREERATIO`).

```
namedWindow( "Display window", CV_WINDOW_AUTOSIZE );// Create a window for display.
```

Finally, to update the content of the OpenCV window with a new image use the `imshow` function. Specify the OpenCV window name to update and the image to use during this operation:

```
imshow( "Display window", image ); // Show our image inside it.
```

Because we want our window to be displayed until the user presses a key (otherwise the program would end far too quickly), we use the `waitKey` function whose only parameter is just how long should it wait for a user input (measured in milliseconds). Zero means to wait forever.

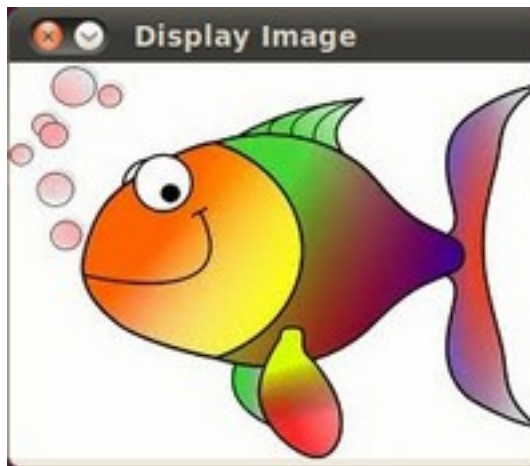
```
waitKey(0); // Wait for a keystroke in the window
```

## Result

- Compile your code and then run the executable giving an image path as argument. If you're on Windows the executable will of course contain an *exe* extension too. Of course assure the image file is near your program file.

```
./DisplayImage HappyFish.jpg
```

- You should get a nice window as the one shown below:



## 1.8 Load, Modify, and Save an Image

---

**Note:** We assume that by now you know how to load an image using `imread` and to display it in a window (using `imshow`). Read the *Load and Display an Image* tutorial otherwise.

---

### Goals

In this tutorial you will learn how to:

- Load an image using `imread`
- Transform an image from RGB to Grayscale format by using `cvtColor`
- Save your transformed image in a file on disk (using `imwrite`)

### Code

Here it is:

```
1  #include <cv.h>
2  #include <highgui.h>
3
4  using namespace cv;
5
6  int main( int argc, char** argv )
7  {
8      char* imageName = argv[1];
9
10     Mat image;
11     image = imread( imageName, 1 );
12
13     if( argc != 2 || !image.data )
```



```

14 {
15     printf( " No image data \n " );
16     return -1;
17 }
18
19 Mat gray_image;
20 cvtColor( image, gray_image, CV_RGB2GRAY );
21
22 imwrite( "../images/Gray_Image.jpg", gray_image );
23
24 namedWindow( imageName, CV_WINDOW_AUTOSIZE );
25 namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );
26
27 imshow( imageName, image );
28 imshow( "Gray image", gray_image );
29
30 waitKey(0);
31
32 return 0;
33 }

```

## Explanation

1. We begin by:

- Creating a Mat object to store the image information
- Load an image using `imread`, located in the path given by `imageName`. For this example, assume you are loading a RGB image.

2. Now we are going to convert our image from RGB to Grayscale format. OpenCV has a really nice function to do this kind of transformations:

```
cvtColor( image, gray_image, CV_RGB2GRAY );
```

As you can see, `cvtColor` takes as arguments:

- a source image (`image`)
- a destination image (`gray_image`), in which we will save the converted image.
- an additional parameter that indicates what kind of transformation will be performed. In this case we use **CV\_RGB2GRAY** (self-explanatory).

3. So now we have our new `gray_image` and want to save it on disk (otherwise it will get lost after the program ends). To save it, we will use a function analogous to `imread`: `imwrite`

```
imwrite( "../images/Gray_Image.jpg", gray_image );
```

Which will save our `gray_image` as `Gray_Image.jpg` in the folder `images` located two levels up of my current location.

4. Finally, let's check out the images. We create two windows and use them to show the original image as well as the new one:

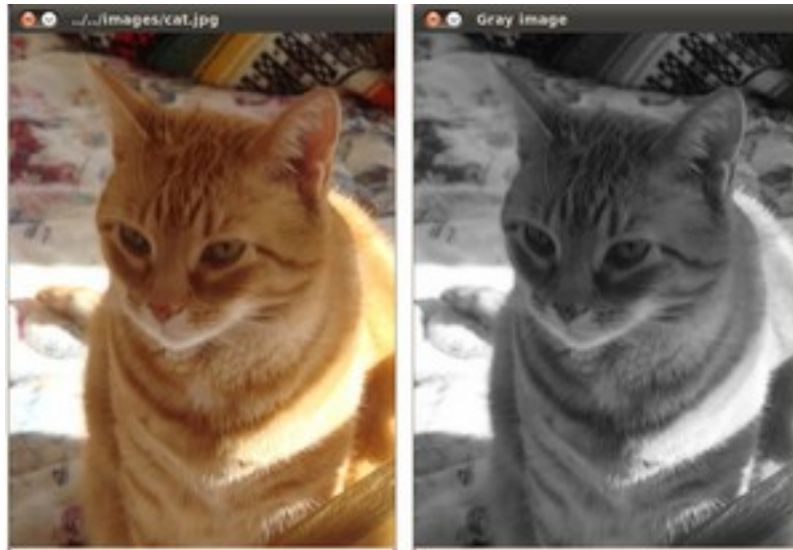
```
namedWindow( imageName, CV_WINDOW_AUTOSIZE );
namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );
```

```
imshow( imageName, image );
imshow( "Gray image", gray_image );
```

5. Add add the `waitKey(0)` function call for the program to wait forever for an user key press.

## Result

When you run your program you should get something like this:



And if you check in your folder (in my case `images`), you should have a newly .jpg file named `Gray_Image.jpg`:



Congratulations, you are done with this tutorial!

---

# CORE MODULE. THE CORE FUNCTIONALITY

Here you will learn the about the basic building blocks of the library. A must read and know for understanding how to manipulate the images on a pixel level.

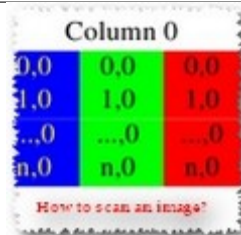


**Title:** *Mat - The Basic Image Container*

**Compatibility:** > OpenCV 2.0

**Author:** Bernát Gábor

You will learn how to store images in the memory and how to print out their content to the console.



**Title:** *How to scan images, lookup tables and time measurement with OpenCV*

**Compatibility:** > OpenCV 2.0

**Author:** Bernát Gábor

You'll find out how to scan images (go through each of the image pixels) with OpenCV. Bonus: time measurement with OpenCV.



**Title:** *Adding (blending) two images using OpenCV*

**Compatibility:** > OpenCV 2.0

**Author:** Ana Huamán

We will learn how to blend two images!

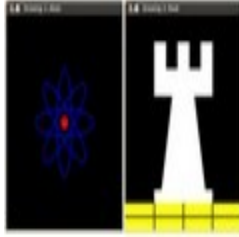


**Title:** *Changing the contrast and brightness of an image!*

**Compatibility:** > OpenCV 2.0

**Author:** Ana Huamán

We will learn how to change our image appearance!



**Title:** *Basic Drawing*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

We will learn how to draw simple geometry with OpenCV!



**Title:** \* *Fancy Drawing!*

*Compatibility:* > OpenCV 2.0

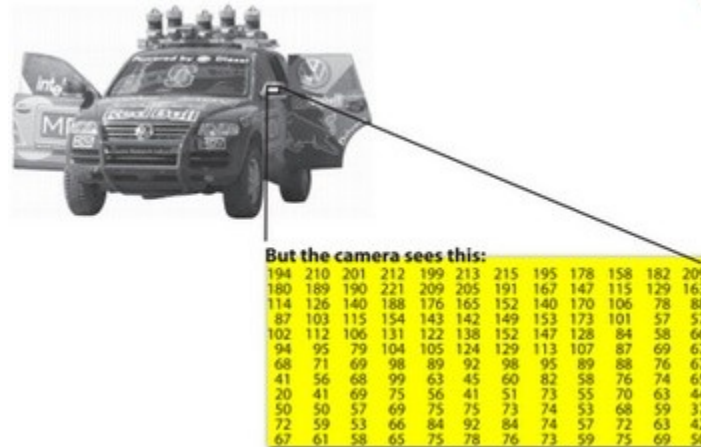
*Author:* Ana Huamán

We will draw some *fancy-looking* stuff using OpenCV!

## 2.1 Mat - The Basic Image Container

### Goal

We have multiple ways to acquire digital images from the real world: digital cameras, scanners, computed tomography or magnetic resonance imaging to just name a few. In every case what we (humans) see are images. However, when transforming this to our digital devices what we record are numerical values for each of the points of the image.



For example in the above image you can see that the mirror of the care is nothing more than a matrix containing all the intensity values of the pixel points. Now, how we get and store the pixels values may vary according to what fits best our need, in the end all images inside a computer world may be reduced to numerical matrices and some other information's describing the matrix itself. *OpenCV* is a computer vision library whose main focus is to process and manipulate these information to find out further ones. Therefore, the first thing you need to learn and get accommodated with is how *OpenCV* stores and handles images.

### Mat

*OpenCV* has been around ever since 2001. In those days the library was built around a *C* interface. In those days to store the image in the memory they used a *C* structure entitled *IplImage*. This is the one you'll see in most of the older tutorials and educational materials. The problem with this is that it brings to the table all the minuses of the *C* language. The biggest issue is the manual management. It builds on the assumption that the user is responsible for taking care of memory allocation and deallocation. While this is no issue in case of smaller programs once your code base start to grove larger and larger it will be more and more a struggle to handle all this rather than focusing on actually solving your development goal.

Luckily *C++* came around and introduced the concept of classes making possible to build another road for the user: automatic memory management (more or less). The good news is that *C++* is fully compatible with *C* so no compatibility issues can arise from making the change. Therefore, *OpenCV* with its 2.0 version introduced a new *C++* interface that by taking advantage of these offers a new way of doing things. A way, in which you do not need to fiddle with memory management; making your code concise (less to write, to achieve more). The only main downside of the *C++* interface is that many embedded development systems at the moment support only *C*. Therefore, unless you are targeting this platform, there's no point on using the *old* methods (unless you're a masochist programmer and you're asking for trouble).

The first thing you need to know about *Mat* is that you no longer need to manually allocate its size and release it as soon as you do not need it. While doing this is still a possibility, most of the *OpenCV* functions will allocate its output data manually. As a nice bonus if you pass on an already existing *Mat* object, what already has allocated the required space for the matrix, this will be reused. In other words we use at all times only as much memory as much we must to perform the task.

*Mat* is basically a class having two data parts: the matrix header (containing information such as the size of the matrix, the method used for storing, at which address is the matrix stored and so on) and a pointer to the matrix containing the pixel values (may take any dimensionality depending on the method chosen for storing). The matrix header size is constant. However, the size of the matrix itself may vary from image to image and usually is larger by order of magnitudes. Therefore, when you're passing on images in your program and at some point you need to create a copy of the image the big price you will need to build is for the matrix itself rather than its header. OpenCV is an image processing library. It contains a large collection of image processing functions. To solve a computational challenge most of the time you will end up using multiple functions of the library. Due to this passing on images to functions is a common practice. We should not forget that we are talking about image processing algorithms, which tend to be quite computational heavy. The last thing we want to do is to further decrease the speed of your program by making unnecessary copies of potentially *large* images.

To tackle this issue OpenCV uses a reference counting system. The idea is that each *Mat* object has its own header, however the matrix may be shared between two instance of them by having their matrix pointer point to the same address. Moreover, the copy operators **will only copy the headers**, and as also copy the pointer to the large matrix too, however not the matrix itself.

```
1 Mat A, C; // creates just the header parts
2 A = imread(argv[1], CV_LOAD_IMAGE_COLOR); // here we'll know the method used (allocate matrix)
3
4 Mat B(A); // Use the copy constructor
5
6 C = A; // Assignment operator
```

All the above objects, in the end point to the same single data matrix. Their headers are different, however making any modification using either one of them will affect all the other ones too. In practice the different objects just provide different access method to the same underlying data. Nevertheless, their header parts are different. The real interesting part comes that you can create headers that refer only to a subsection of the full data. For example, to create a region of interest (*ROI*) in an image you just create a new header with the new boundaries:

```
1 Mat D (A, Rect(10, 10, 100, 100) ); // using a rectangle
2 Mat E = A(Range::all(), Range(1,3)); // using row and column boundaries
```

Now you may ask if the matrix itself may belong to multiple *Mat* objects who will take responsibility for its cleaning when it's no longer needed. The short answer is: the last object that used it. For this a reference counting mechanism is used. Whenever somebody copies a header of a *Mat* object a counter is increased for the matrix. Whenever a header is cleaned this counter is decreased. When the counter reaches zero the matrix too is freed. Because, sometimes you will still want to copy the matrix itself too, there exists the `clone()` or the `copyTo()` function.

```
1 Mat F = A.clone();
2 Mat G;
3 A.copyTo(G);
```

Now modifying *F* or *G* will not affect the matrix pointed by the *Mat* header. What you need to remember from all this is that:

- Output image allocation for OpenCV functions is automatic (unless specified otherwise).
- No need to think about memory freeing with OpenCV's C++ interface.
- The assignment operator and the copy constructor (*ctor*) copies only the header.
- Use the `clone()` or the `copyTo()` function to copy the underlying matrix of an image.

## Storing methods

This is about how you store the pixel values. You can select the color space and the data type used. The color space refers to how we combine color components in order to code a given color. The simplest one is the gray scale. Here

the colors at our disposal are black and white. The combination of these allows us to create many shades of gray.

For *colorful* ways we have a lot more of methods to choose from. However, every one of them breaks it down to three or four basic components and the combination of this will give all others. The most popular one of this is RGB, mainly because this is also how our eye builds up colors in our eyes. Its base colors are red, green and blue. To code the transparency of a color sometimes a fourth element: alpha (A) is added.

However, they are many color systems each with their own advantages:

- RGB is the most common as our eyes use something similar, our display systems also compose colors using these.
- The HSV and HLS decompose colors into their hue, saturation and value/luminance components, which is a more natural way for us to describe colors. Using you may for example dismiss the last component, making your algorithm less sensible to light conditions of the input image.
- YCrCb is used by the popular JPEG image format.
- CIE L\*a\*b\* is a perceptually uniform color space, which comes handy if you need to measure the *distance* of a given color to another color.

Now each of the building components has their own valid domains. This leads to the data type used. How we store a component defines just how fine control we have over its domain. The smallest data type possible is *char*, which means one byte or 8 bits. This may be signed (so can store values from 0 to 255) or unsigned (values from -127 to +127). Although in case of three components this already gives 16 million possible colors to represent (like in case of RGB) we may acquire an even finer control by using the float (4 byte = 32 bit) or double (8 byte = 64 bit) data types for each component. Nevertheless, remember that increasing the size of a component also increases the size of the whole picture in the memory.

## Creating explicitly a *Mat* object

In the *Load, Modify, and Save an Image* tutorial you could already see how to write a matrix to an image file by using the `:readWriteImageVideo: imwrite() <imwrite>` function. However, for debugging purposes it's much more convenient to see the actual values. You can achieve this via the `<<` operator of *Mat*. However, be aware that this only works for two dimensional matrices.

Although *Mat* is a great class as image container it is also a general matrix class. Therefore, it is possible to create and manipulate multidimensional matrices. You can create a *Mat* object in multiple ways:

- `Mat()` Constructor

```
M =
[0, 0, 255, 0, 0, 255;
 0, 0, 255, 0, 0, 255]
```

For two dimensional and multichannel images we first define their size: row and column count wise.

Then we need to specify the data type to use for storing the elements and the number of channels per matrix point. To do this we have multiple definitions made according to the following convention:

```
CV_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The channel number]
```

For instance, `CV_8UC3` means we use unsigned char types that are 8 bit long and each pixel has three items of this to form the three channels. This are predefined for up to four channel numbers. The *Scalar* is four element short vector. Specify this and you can initialize all matrix points with a custom value. However if you need more you can create the type with the upper macro and putting the channel number in parenthesis as you can see below.



- Use C\C++ arrays and initialize via constructor

The upper example shows how to create a matrix with more than two dimensions. Specify its dimension, then pass a pointer containing the size for each dimension and the rest remains the same.

- Create a header for an already existing IplImage pointer:

```
IplImage* img = cvLoadImage("greatwave.png", 1);  
Mat mtx(img); // convert IplImage* -> Mat
```

- Create() function:

```
M =  
[205, 205, 205, 205, 205, 205, 205, 205;  
 205, 205, 205, 205, 205, 205, 205, 205;  
 205, 205, 205, 205, 205, 205, 205, 205;  
 205, 205, 205, 205, 205, 205, 205, 205]
```

You cannot initialize the matrix values with this construction. It will only reallocate its matrix data memory if the new size will not fit into the old one.

- MATLAB style initializer: `zeros()`, `ones()`, `:eyes()`. Specify size and data type to use:

```
E =  
[1, 0, 0, 0;  
 0, 1, 0, 0;  
 0, 0, 1, 0;  
 0, 0, 0, 1]  
Z =  
[0, 0, 0;  
 0, 0, 0;  
 0, 0, 0]  
O =  
[1, 1;  
 1, 1]
```

- For small matrices you may use comma separated initializers:

```
C =  
[0, -1, 0;  
 -1, 5, -1;  
 0, -1, 0]
```

- Create a new header for an existing *Mat* object and `clone()` or `copyTo()` it.

```
RowClone =  
[-1, 5, -1]
```

## Print out formatting

---

**Note:** You can fill out a matrix with random values using the `randu()` function. You need to give the lower and upper value between what you want the random values:

---

In the above examples you could see the default formatting option. Nevertheless, OpenCV allows you to format your matrix output format to fit the rules of:

- Default

```
R (default) =
[91, 2, 79, 179, 52, 205;
 236, 8, 181, 239, 26, 248;
 207, 218, 45, 183, 158, 101]
```

- Python

```
R (python) =
[[[91, 2, 79], [179, 52, 205]],
 [[236, 8, 181], [239, 26, 248]],
 [[207, 218, 45], [183, 158, 101]]]
```

- Comma separated values (CSV)

```
R (csv) =
91, 2, 79, 179, 52, 205
236, 8, 181, 239, 26, 248
207, 218, 45, 183, 158, 101
```

- Numpy

```
R (numpy) =
array([[91, 2, 79], [179, 52, 205]],
      [[236, 8, 181], [239, 26, 248]],
      [[207, 218, 45], [183, 158, 101]]], type='uint8')
```

- C

```
R (c) =
<91, 2, 79, 179, 52, 205,
 236, 8, 181, 239, 26, 248,
 207, 218, 45, 183, 158, 101>
```

## Print for other common items

OpenCV offers support for print of other common OpenCV data structures too via the << operator like:

- 2D Point

```
Point (2D) = [5, 11]
```

- 3D Point

```
Point (3D) = [2, 6, 71]
```

- std::vector via cv::Mat

```
Vector of floats via Mat = [3.1415927; 2; 3.01]
```

- std::vector of points

```
A vector of 2D Points = [0, 0; 5, 1; 10, 2; 15, 3; 20, 4; 25, 5; 30, 6; 35, 0; 40, 1; 45, 2; 50, 3; 55, 4; 60, 5; 65, 6; 70, 0; 75, 1; 80, 2; 85, 3; 90, 4; 95, 5]
```

Most of the samples here have been included into a small console application. You can download it from here or in the core section of the cpp samples.

A quick video demonstration of this you can find on [YouTube](#).

## 2.2 How to scan images, lookup tables and time measurement with OpenCV

### Goal

We'll seek answers for the following questions:

- How to go through each and every pixel of an image?
- How is OpenCV matrix values stored?
- How to measure the performance of our algorithm?
- What are lookup tables and why use them?

### Our test case

Let us consider a simple color reduction method. Using the unsigned char C and C++ type for matrix item storing a channel of pixel may have up to 256 different values. For a three channel image this can allow the formation of way too many colors (16 million to be exact). Working with so many color shades may give a heavy blow to our algorithm performance. However, sometimes it is enough to work with a lot less of them to get the same final result.

In this cases it's common that we make a *color space reduction*. This means that we divide the color space current value with a new input value to end up with fewer colors. For instance every value between zero and nine takes the new value zero, every value between ten and nineteen the value ten and so on.

When you divide an *uchar* (unsigned char - aka values between zero and 255) value with an *int* value the result will be also *char*. These values may only be char values. Therefore, any fraction will be rounded down. Taking advantage of this fact the upper operation in the *uchar* domain may be expressed as:

$$I_{new} = \left( \frac{I_{old}}{10} \right) * 10$$

A simple color space reduction algorithm would consist of just passing through every pixel of an image matrix and applying this formula. It's worth noting that we do a divide and a multiplication operation. These operations are bloody expensive for a system. If possible it's worth avoiding them by using cheaper operations such as a few subtractions, addition or in best case a simple assignment. Furthermore, note that we only have a limited number of input values for the upper operation. In case of the *uchar* system this is 256 to be exact.

Therefore, for larger images it would be wise to calculate all possible values beforehand and during the assignment just make the assignment, by using a lookup table. Lookup tables are simple arrays (having one or more dimensions) that for a given input value variation holds the final output value. Its strength lies that we do not need to make the calculation, we just need to read the result.

Our test case program (and the sample presented here) will do the following: read in a console line argument image (that may be either color or gray scale - console line argument too) and apply the reduction with the given console line argument integer value. In OpenCV, at the moment they are three major ways of going through an image pixel by

pixel. To make things a little more interesting will make the scanning for each image using all of these methods, and print out how long it took.

You can download the full source code here or look it up in the samples directory of OpenCV at the `cpp` tutorial code for the core section. Its basic usage is:

```
how_to_scan_images imageName.jpg intValueToReduce [G]
```

The final argument is optional. If given the image will be loaded in gray scale format, otherwise the RGB color way is used. The first thing is to calculate the lookup table.

Here we first use the C++ `stringstream` class to convert the third command line argument from text to an integer format. Then we use a simple look and the upper formula to calculate the lookup table. No OpenCV specific stuff here.

Another issue is how do we measure time? Well OpenCV offers two simple functions to achieve this `getTickCount()` and `getTickFrequency()`. The first returns the number of ticks of your systems CPU from a certain event (like since you booted your system). The second returns how many times your CPU emits a tick during a second. So to measure in seconds the number of time elapsed between two operations is easy as:

```
double t = (double)getTickCount();
// do something ...
t = ((double)getTickCount() - t)/getTickFrequency();
cout << "Times passed in seconds: " << t << endl;
```

## How the image matrix is stored in the memory?

As you could already read in my *Mat - The Basic Image Container* tutorial the size of the matrix depends of the color system used. More accurately, it depends from the number of channels used. In case of a gray scale image we have something like:

	Column 0	Column 1	Column ...	Column m
Row 0	0,0	0,1	...	0, m
Row 1	1,0	1,1	...	1, m
Row ...	...,0	...,1	...	..., m
Row n	n,0	n,1	n,...	n, m

For multichannel images the columns contain as many sub columns as the number of channels. For example in case of an RGB color system:

	Column 0			Column 1			Column ...			Column m		
Row 0	0,0	0,0	0,0	0,1	0,1	0,1	...	...	...	0, m	0, m	0, m
Row 1	1,0	1,0	1,0	1,1	1,1	1,1	...	...	...	1, m	1, m	1, m
Row ...	...,0	...,0	...,0	...,1	...,1	...,1	...	...	...	..., m	..., m	..., m
Row n	n,0	n,0	n,0	n,1	n,1	n,1	n,...	n,...	n,...	n, m	n, m	n, m

Note that the order of the channels is inverse: BGR instead of RGB. Because in many cases the memory is large enough to store the rows in a successive fashion the rows may follow one after another, creating a single long row. Because everything is in a single place following one after another this may help to speed up the scanning process. We can use the `isContinuous()` function to ask the matrix if this is the case. Continue on to the next section to find an example.

## The efficient way

When it comes to performance you cannot beat the classic C style operator[] (pointer) access. Therefore, the most efficient method we can recommend for making the assignment is:

Here we basically just acquire a pointer to the start of each row and go through it until it ends. In the special case that the matrix is stored in a continues manner we only need to request the pointer a single time and go all the way to the end. We need to look out for color images: we have three channels so we need to pass through three times more items in each row.

There's another way of this. The *data* data member of a *Mat* object returns the pointer to the first row, first column. If this pointer is null you have no valid input in that object. Checking this is the simplest method to check if your image loading was a success. In case the storage is continues we can use this to go through the whole data pointer. In case of a gray scale image this would look like:

```
uchar* p = I.data;

for( unsigned int i =0; i < ncol*nrows; ++i)
    *p++ = table[*p];
```

You would get the same result. However, this code is a lot harder to read later on. It gets even harder if you have some more advanced technique there. Moreover, in practice I've observed you'll get the same performance result (as most of the modern compilers will probably make this small optimization trick automatically for you).

### The iterator (safe) method

In case of the efficient way making sure that you pass through the right amount of *uchar* fields and to skip the gaps that may occur between the rows was your responsibility. The iterator method is considered a safer way as it takes over these tasks from the user. All you need to do is ask the begin and the end of the image matrix and then just increase the begin iterator until you reach the end. To acquire the value *pointed* by the iterator use the *\** operator (add it before it).

In case of color images we have three *uchar* items per column. This may be considered a short vector of *uchar* items, that has been baptized in OpenCV with the *Vec3b* name. To access the *n*-th sub column we use simple operator[] access. It's important to remember that OpenCV iterators go through the columns and automatically skip to the next row. Therefore in case of color images if you use a simple *uchar* iterator you'll be able to access only the blue channel values.

### Random Pixel Access

The final method isn't recommended for scanning. It was made to acquire or modify random elements in the image. Its basic usage is to specify the row and column number of the item you want to access. During our earlier scanning methods you could already observe that is important through what type we are looking at the image. It's no different here as you need manually to specify what type to use at the automatic lookup. You can observe this in case of the gray scale images for the following source code (the usage of the *+ at()* function):

If you need to multiple lookups using this method for an image it may be troublesome and time consuming to enter the type and the *at* keyword for each of the accesses. To solve this problem OpenCV has a *Mat\_* data type. It's the same as *Mat* with the extra need that *at* definition you need to specify the data type through what to look at the data matrix, however in return you can use the *operator()* for fast access of items. To make things even better this is easily convertible from and to the usual *Mat* data type. A sample usage of this you can see in case of the color images of the upper function. Nevertheless, it's important to note that the same operation (with the same runtime speed) could have been done with the *at()* function. It's just a less to write for the lazy programmer trick.

### The Core Function

This is a bonus method of achieving lookup table modification in an image. Because in image processing it's quite common that you want to replace all of a given image value to some other value OpenCV has a function that makes

the modification without the need from you to write the scanning of the image. We use the `LUT()` function of the core module. First we build a Mat type of the lookup table:

Finally call the function (I is our input image and J the output one):

## Performance Difference

For the best result compile the program and run it on your own speed. For showing off better the differences I've used a quite large (2560 X 1600) image. The performance presented here are for color images. For a more accurate value I've averaged the value I got from the call of the function for hundred times.

Efficient Way	79.4717 milliseconds
Iterator	83.7201 milliseconds
Random Access	93.7878 milliseconds
LUT function	32.5759 milliseconds

We can conclude a couple of things. If possible, use the already made functions of OpenCV (instead reinventing these). The fastest method turns out to be the LUT function. This is because the OpenCV library is multi-thread enabled via Intel Threaded Building Blocks. However, if you need to write a simple image scan prefer the pointer method. The iterator is a safer bet, however quite slower. Using the random access method for full image scan is the most costly. Your compiler might observe what you want to do and optimize it out somewhat, however in general case this approach is with a little slower than the iterator method.

Finally, you may watch a sample run of the program on the [video posted](#) on our YouTube channel.

## 2.3 Adding (blending) two images using OpenCV

### Goal

In this tutorial you will learn how to:

- What is *linear blending* and why it is useful.
- Add two images using `addWeighted`

### Cool Theory

**Note:** The explanation below belongs to the book [Computer Vision: Algorithms and Applications](#) by Richard Szeliski

From our previous tutorial, we know already a bit of *Pixel operators*. An interesting dyadic (two-input) operator is the *linear blend operator*:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

By varying  $\alpha$  from  $0 \rightarrow 1$  this operator can be used to perform a temporal *cross-dissolve* between two images or videos, as seen in slide shows and film production (cool, eh?)

### Code

As usual, after the not-so-lengthy explanation, let's go to the code. Here it is:

```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;

int main( int argc, char** argv )
{
    double alpha = 0.5; double beta; double input;

    Mat src1, src2, dst;

    /// Ask the user enter alpha
    std::cout<<" Simple Linear Blender "<<std::endl;
    std::cout<<"-----"<<std::endl;
    std::cout<<"* Enter alpha [0-1]: ";
    std::cin>>input;

    /// We use the alpha provided by the user iff it is between 0 and 1
    if( alpha >= 0 && alpha <= 1 )
        { alpha = input; }

    /// Read image ( same size, same type )
    src1 = imread("../images/LinuxLogo.jpg");
    src2 = imread("../images/WindowsLogo.jpg");

    if( !src1.data ) { printf("Error loading src1 \n"); return -1; }
    if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

    /// Create Windows
    namedWindow("Linear Blend", 1);

    beta = ( 1.0 - alpha );
    addWeighted( src1, alpha, src2, beta, 0.0, dst);

    imshow( "Linear Blend", dst );

    waitKey(0);
    return 0;
}
```

## Explanation

1. Since we are going to perform:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

We need two source images ( $f_0(x)$  and  $f_1(x)$ ). So, we load them in the usual way:

```
src1 = imread("../images/LinuxLogo.jpg");
src2 = imread("../images/WindowsLogo.jpg");
```

**Warning:** Since we are *adding* *src1* and *src2*, they both have to be of the same size (width and height) and type.

2. Now we need to generate the  $g(x)$  image. For this, the function `addWeighted` comes quite handy:



```
beta = ( 1.0 - alpha );  
addWeighted( src1, alpha, src2, beta, 0.0, dst);
```

since `addWeighted` produces:

$$dst = \alpha \cdot src1 + \beta \cdot src2 + \gamma$$

In this case,  $\gamma$  is the argument `0.0` in the code above.

3. Create windows, show the images and wait for the user to end the program.

## Result



## 2.4 Changing the contrast and brightness of an image!

### Goal

In this tutorial you will learn how to:

- Access pixel values
- Initialize a matrix with zeros
- Learn what `saturate_cast` does and why it is useful
- Get some cool info about pixel transformations

### Cool Theory

---

**Note:** The explanation below belongs to the book *Computer Vision: Algorithms and Applications* by Richard Szeliski

---

### Image Processing

- A general image processing operator is a function that takes one or more input images and produces an output image.

- Image transforms can be seen as:
  - Point operators (pixel transforms)
  - Neighborhood (area-based) operators

### Pixel Transforms

- In this kind of image processing transform, each output pixel's value depends on only the corresponding input pixel value (plus, potentially, some globally collected information or parameters).
- Examples of such operators include *brightness and contrast adjustments* as well as color correction and transformations.

### Brightness and contrast adjustments

- Two commonly used point processes are *multiplication* and *addition* with a constant:

$$g(x) = \alpha f(x) + \beta$$

- The parameters  $\alpha > 0$  and  $\beta$  are often called the *gain* and *bias* parameters; sometimes these parameters are said to control *contrast* and *brightness* respectively.
- You can think of  $f(x)$  as the source image pixels and  $g(x)$  as the output image pixels. Then, more conveniently we can write the expression as:

$$g(i, j) = \alpha \cdot f(i, j) + \beta$$

where  $i$  and  $j$  indicates that the pixel is located in the  $i$ -th row and  $j$ -th column.

### Code

- The following code performs the operation  $g(i, j) = \alpha \cdot f(i, j) + \beta$
- Here it is:

```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;

double alpha; /**< Simple contrast control */
int beta; /**< Simple brightness control */

int main( int argc, char** argv )
{
    /// Read image given by user
    Mat image = imread( argv[1] );
    Mat new_image = Mat::zeros( image.size(), image.type() );

    /// Initialize values
    std::cout<<" Basic Linear Transforms "<<std::endl;
    std::cout<<"-----"<<std::endl;
    std::cout<<"* Enter the alpha value [1.0-3.0]: ";std::cin>>alpha;
    std::cout<<"* Enter the beta value [0-100]: "; std::cin>>beta;
```

```

/// Do the operation new_image(i,j) = alpha*image(i,j) + beta
for( int y = 0; y < image.rows; y++ )
{ for( int x = 0; x < image.cols; x++ )
  { for( int c = 0; c < 3; c++ )
    {
      new_image.at<Vec3b>(y,x)[c] =
        saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
    }
  }
}

/// Create Windows
namedWindow("Original Image", 1);
namedWindow("New Image", 1);

/// Show stuff
imshow("Original Image", image);
imshow("New Image", new_image);

/// Wait until user press some key
waitKey();
return 0;
}

```

## Explanation

1. We begin by creating parameters to save  $\alpha$  and  $\beta$  to be entered by the user:

```

double alpha;
int beta;

```

2. We load an image using `imread` and save it in a Mat object:

```

Mat image = imread( argv[1] );

```

3. Now, since we will make some transformations to this image, we need a new Mat object to store it. Also, we want this to have the following features:

- Initial pixel values equal to zero
- Same size and type as the original image

```

Mat new_image = Mat::zeros( image.size(), image.type() );

```

We observe that `Mat::zeros` returns a Matlab-style zero initializer based on `image.size()` and `image.type()`

4. Now, to perform the operation  $g(i,j) = \alpha \cdot f(i,j) + \beta$  we will access to each pixel in image. Since we are operating with RGB images, we will have three values per pixel (R, G and B), so we will also access them separately. Here is the piece of code:

```

for( int y = 0; y < image.rows; y++ )
{ for( int x = 0; x < image.cols; x++ )
  { for( int c = 0; c < 3; c++ )
    { new_image.at<Vec3b>(y,x)[c] =
      saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta ); }
  }
}

```

Notice the following:

- To access each pixel in the images we are using this syntax: `image.at<Vec3b>(y,x)[c]` where  $y$  is the row,  $x$  is the column and  $c$  is R, G or B (0, 1 or 2).
- Since the operation  $\alpha \cdot p(i,j) + \beta$  can give values out of range or not integers (if  $\alpha$  is float), we use `saturate_cast` to make sure the values are valid.

5. Finally, we create windows and show the images, the usual way.

```
namedWindow("Original Image", 1);
namedWindow("New Image", 1);

imshow("Original Image", image);
imshow("New Image", new_image);

waitKey(0);
```

---

**Note:** Instead of using the `for` loops to access each pixel, we could have simply used this command:

```
image.convertTo(new_image, -1, alpha, beta);
```

where `convertTo` would effectively perform  $new\_image = a * image + beta$ . However, we wanted to show you how to access each pixel. In any case, both methods give the same result.

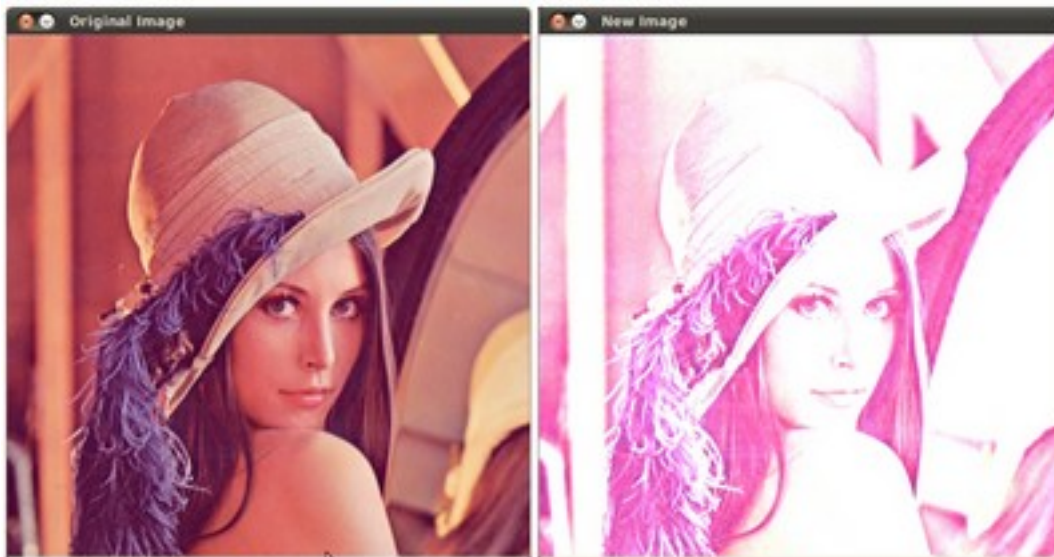
---

## Result

- Running our code and using  $\alpha = 2.2$  and  $\beta = 50$

```
$ ./BasicLinearTransforms lena.jpg
Basic Linear Transforms
-----
* Enter the alpha value [1.0-3.0]: 2.2
* Enter the beta value [0-100]: 50
```

- We get this:



## 2.5 Basic Drawing

### Goals

In this tutorial you will learn how to:

- Use `Point` to define 2D points in an image.
- Use `Scalar` and why it is useful
- Draw a **line** by using the OpenCV function `line`
- Draw an **ellipse** by using the OpenCV function `ellipse`
- Draw a **rectangle** by using the OpenCV function `rectangle`
- Draw a **circle** by using the OpenCV function `circle`
- Draw a **filled polygon** by using the OpenCV function `fillPoly`

### OpenCV Theory

For this tutorial, we will heavily use two structures: `Point` and `Scalar`:

#### Point

It represents a 2D point, specified by its image coordinates `x` and `y`. We can define it as:

```
Point pt;
pt.x = 10;
pt.y = 8;
```

or

```
Point pt = Point(10, 8);
```

#### Scalar

- Represents a 4-element vector. The type `Scalar` is widely used in OpenCV for passing pixel values.
- In this tutorial, we will use it extensively to represent RGB color values (3 parameters). It is not necessary to define the last argument if it is not going to be used.
- Let's see an example, if we are asked for a color argument and we give:

```
Scalar( a, b, c )
```

We would be defining a RGB color such as: *Red* = *c*, *Green* = *b* and *Blue* = *a*

#### Code

- This code is in your OpenCV sample folder. Otherwise you can grab it from [here](#)

## Explanation

1. Since we plan to draw two examples (an atom and a rook), we have to create 02 images and two windows to display them.

```
/// Windows names
char atom_window[] = "Drawing 1: Atom";
char rook_window[] = "Drawing 2: Rook";

/// Create black empty images
Mat atom_image = Mat::zeros( w, w, CV_8UC3 );
Mat rook_image = Mat::zeros( w, w, CV_8UC3 );
```

2. We created functions to draw different geometric shapes. For instance, to draw the atom we used *MyEllipse* and *MyFilledCircle*:

```
/// 1. Draw a simple atom:

/// 1.a. Creating ellipses
MyEllipse( atom_image, 90 );
MyEllipse( atom_image, 0 );
MyEllipse( atom_image, 45 );
MyEllipse( atom_image, -45 );

/// 1.b. Creating circles
MyFilledCircle( atom_image, Point( w/2.0, w/2.0 ) );
```

3. And to draw the rook we employed *MyLine*, *rectangle* and a *MyPolygon*:

```
/// 2. Draw a rook

/// 2.a. Create a convex polygon
MyPolygon( rook_image );

/// 2.b. Creating rectangles
rectangle( rook_image,
           Point( 0, 7*w/8.0 ),
           Point( w, w ),
           Scalar( 0, 255, 255 ),
           -1,
           8 );

/// 2.c. Create a few lines
MyLine( rook_image, Point( 0, 15*w/16 ), Point( w, 15*w/16 ) );
MyLine( rook_image, Point( w/4, 7*w/8 ), Point( w/4, w ) );
MyLine( rook_image, Point( w/2, 7*w/8 ), Point( w/2, w ) );
MyLine( rook_image, Point( 3*w/4, 7*w/8 ), Point( 3*w/4, w ) );
```

4. Let's check what is inside each of these functions:

- *MyLine*

```
void MyLine( Mat img, Point start, Point end )
{
    int thickness = 2;
    int lineType = 8;
    line( img,
          start,
          end,
          Scalar( 0, 0, 0 ),
```

```

        thickness,
        lineType );
}

```

As we can see, *MyLine* just call the function `line`, which does the following:

- Draw a line from Point **start** to Point **end**
- The line is displayed in the image **img**
- The line color is defined by **Scalar( 0, 0, 0)** which is the RGB value correspondent to **Black**
- The line thickness is set to **thickness** (in this case 2)
- The line is a 8-connected one (**lineType = 8**)

- *MyEllipse*

```

void MyEllipse( Mat img, double angle )
{
    int thickness = 2;
    int lineType = 8;

    ellipse( img,
            Point( w/2.0, w/2.0 ),
            Size( w/4.0, w/16.0 ),
            angle,
            0,
            360,
            Scalar( 255, 0, 0 ),
            thickness,
            lineType );
}

```

From the code above, we can observe that the function `ellipse` draws an ellipse such that:

- The ellipse is displayed in the image **img**
- The ellipse center is located in the point **(w/2.0, w/2.0)** and is enclosed in a box of size **(w/4.0, w/16.0)**
- The ellipse is rotated **angle** degrees
- The ellipse extends an arc between **0** and **360** degrees
- The color of the figure will be **Scalar( 255, 255, 0)** which means blue in RGB value.
- The ellipse's **thickness** is 2.

- *MyFilledCircle*

```

void MyFilledCircle( Mat img, Point center )
{
    int thickness = -1;
    int lineType = 8;

    circle( img,
            center,
            w/32.0,
            Scalar( 0, 0, 255 ),
            thickness,
            lineType );
}

```

Similar to the ellipse function, we can observe that `circle` receives as arguments:



- The image where the circle will be displayed (**img**)
- The center of the circle denoted as the Point **center**
- The radius of the circle: **w/32.0**
- The color of the circle: **Scalar(0, 0, 255)** which means *Red* in RGB
- Since **thickness** = -1, the circle will be drawn filled.

- *MyPolygon*

```

void MyPolygon( Mat img )
{
    int lineType = 8;

    /** Create some points */
    Point rook_points[1][20];
    rook_points[0][0] = Point( w/4.0, 7*w/8.0 );
    rook_points[0][1] = Point( 3*w/4.0, 7*w/8.0 );
    rook_points[0][2] = Point( 3*w/4.0, 13*w/16.0 );
    rook_points[0][3] = Point( 11*w/16.0, 13*w/16.0 );
    rook_points[0][4] = Point( 19*w/32.0, 3*w/8.0 );
    rook_points[0][5] = Point( 3*w/4.0, 3*w/8.0 );
    rook_points[0][6] = Point( 3*w/4.0, w/8.0 );
    rook_points[0][7] = Point( 26*w/40.0, w/8.0 );
    rook_points[0][8] = Point( 26*w/40.0, w/4.0 );
    rook_points[0][9] = Point( 22*w/40.0, w/4.0 );
    rook_points[0][10] = Point( 22*w/40.0, w/8.0 );
    rook_points[0][11] = Point( 18*w/40.0, w/8.0 );
    rook_points[0][12] = Point( 18*w/40.0, w/4.0 );
    rook_points[0][13] = Point( 14*w/40.0, w/4.0 );
    rook_points[0][14] = Point( 14*w/40.0, w/8.0 );
    rook_points[0][15] = Point( w/4.0, w/8.0 );
    rook_points[0][16] = Point( w/4.0, 3*w/8.0 );
    rook_points[0][17] = Point( 13*w/32.0, 3*w/8.0 );
    rook_points[0][18] = Point( 5*w/16.0, 13*w/16.0 );
    rook_points[0][19] = Point( w/4.0, 13*w/16.0 );

    const Point* ppt[1] = { rook_points[0] };
    int npt[] = { 20 };

    fillPoly( img,
              ppt,
              npt,
              1,
              Scalar( 255, 255, 255 ),
              lineType );
}

```

To draw a filled polygon we use the function `fillPoly`. We note that:

- The polygon will be drawn on **img**
- The vertices of the polygon are the set of points in **ppt**
- The total number of vertices to be drawn are **npt**
- The number of polygons to be drawn is only **1**
- The color of the polygon is defined by **Scalar( 255, 255, 255)**, which is the RGB value for *white*

- *rectangle*

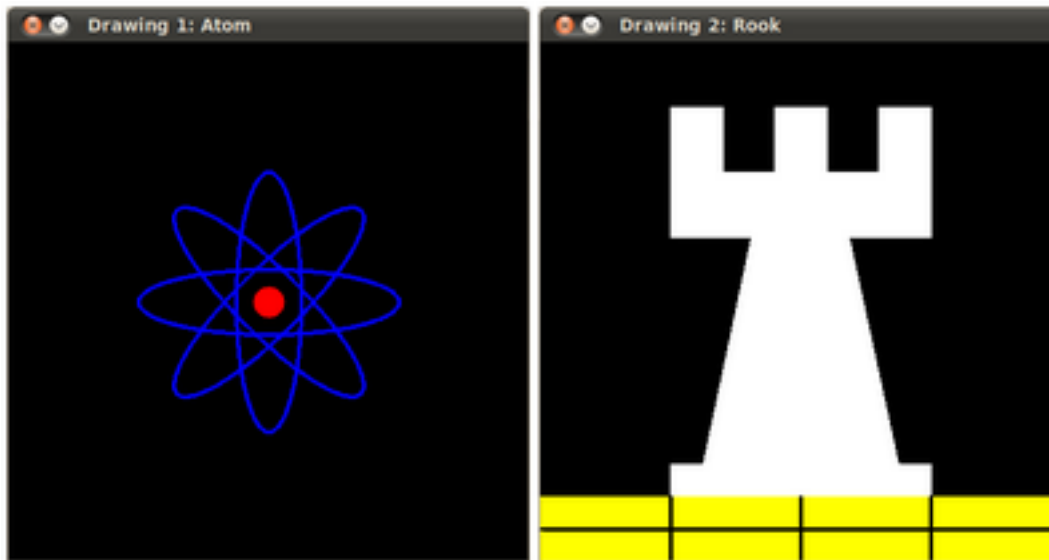
```
rectangle( rook_image,
           Point( 0, 7*w/8.0 ),
           Point( w, w ),
           Scalar( 0, 255, 255 ),
           -1,
           8 );
```

Finally we have the `rectangle` function (we did not create a special function for this guy). We note that:

- The rectangle will be drawn on **rook\_image**
- Two opposite vertices of the rectangle are defined by `** Point( 0, 7*w/8.0 )**` and `Point( w, w )`
- The color of the rectangle is given by `Scalar(0, 255, 255)` which is the RGB value for *yellow*
- Since the thickness value is given by `-1`, the rectangle will be filled.

## Result

Compiling and running your program should give you a result like this:



## 2.6 Fancy Drawing!

### Goals

In this tutorial you will learn how to:

- Use the *Random Number generator class (RNG)* and how to get a random number from a uniform distribution.
- Display Text on an OpenCV window by using the function `putText`

### Code

- In the previous tutorial we drew diverse geometric figures, giving as input parameters such as coordinates (in the form of `Points`), color, thickness, etc. You might have noticed that we gave specific values for these arguments.

- In this tutorial, we intend to use *random* values for the drawing parameters. Also, we intend to populate our image with a big number of geometric figures. Since we will be initializing them in a random fashion, this process will be automatic and made by using *loops*
- This code is in your OpenCV sample folder. Otherwise you can grab it from [here](#)

### Explanation

1. Let's start by checking out the *main* function. We observe that first thing we do is creating a *Random Number Generator* object (RNG):

```
RNG rng( 0xFFFFFFFF );
```

RNG implements a random number generator. In this example, *rng* is a RNG element initialized with the value *0xFFFFFFFF*

2. Then we create a matrix initialized to *zeros* (which means that it will appear as black), specifying its height, width and its type:

```
/// Initialize a matrix filled with zeros  
Mat image = Mat::zeros( window_height, window_width, CV_8UC3 );  
  
/// Show it in a window during DELAY ms  
imshow( window_name, image );
```

3. Then we proceed to draw crazy stuff. After taking a look at the code, you can see that it is mainly divided in 8 sections, defined as functions:

```
/// Now, let's draw some lines  
c = Drawing_Random_Lines(image, window_name, rng);  
if( c != 0 ) return 0;  
  
/// Go on drawing, this time nice rectangles  
c = Drawing_Random_Rectangles(image, window_name, rng);  
if( c != 0 ) return 0;  
  
/// Draw some ellipses  
c = Drawing_Random_Ellipses( image, window_name, rng );  
if( c != 0 ) return 0;  
  
/// Now some polylines  
c = Drawing_Random_Polylines( image, window_name, rng );  
if( c != 0 ) return 0;  
  
/// Draw filled polygons  
c = Drawing_Random_Filled_Polygons( image, window_name, rng );  
if( c != 0 ) return 0;  
  
/// Draw circles  
c = Drawing_Random_Circles( image, window_name, rng );  
if( c != 0 ) return 0;  
  
/// Display text in random positions  
c = Displaying_Random_Text( image, window_name, rng );  
if( c != 0 ) return 0;  
  
/// Displaying the big end!  
c = Displaying_Big_End( image, window_name, rng );
```

All of these functions follow the same pattern, so we will analyze only a couple of them, since the same explanation applies for all.

#### 4. Checking out the function `Drawing_Random_Lines`:

```
/**
 * @function Drawing_Random_Lines
 */
int Drawing_Random_Lines( Mat image, char* window_name, RNG rng )
{
    int lineType = 8;
    Point pt1, pt2;

    for( int i = 0; i < NUMBER; i++ )
    {
        pt1.x = rng.uniform( x_1, x_2 );
        pt1.y = rng.uniform( y_1, y_2 );
        pt2.x = rng.uniform( x_1, x_2 );
        pt2.y = rng.uniform( y_1, y_2 );

        line( image, pt1, pt2, randomColor(rng), rng.uniform(1, 10), 8 );
        imshow( window_name, image );
        if( waitKey( DELAY ) >= 0 )
            { return -1; }
    }

    return 0;
}
```

We can observe the following:

- The *for* loop will repeat **NUMBER** times. Since the function `line` is inside this loop, that means that **NUMBER** lines will be generated.
- The line extremes are given by *pt1* and *pt2*. For *pt1* we can see that:

```
pt1.x = rng.uniform( x_1, x_2 );
pt1.y = rng.uniform( y_1, y_2 );
```

- We know that **rng** is a *Random number generator* object. In the code above we are calling **rng.uniform(a,b)**. This generates a randomly uniformed distribution between the values **a** and **b** (inclusive in **a**, exclusive in **b**).
- From the explanation above, we deduce that the extremes *pt1* and *pt2* will be random values, so the lines positions will be quite unpredictable, giving a nice visual effect (check out the Result section below).
- As another observation, we notice that in the `line` arguments, for the *color* input we enter:

```
randomColor(rng)
```

Let's check the function implementation:

```
static Scalar randomColor( RNG& rng )
{
    int icolor = (unsigned) rng;
    return Scalar( icolor&255, (icolor>>8)&255, (icolor>>16)&255 );
}
```

As we can see, the return value is an *Scalar* with 3 randomly initialized values, which are used as the *R*, *G* and *B* parameters for the line color. Hence, the color of the lines will be random too!

- The explanation above applies for the other functions generating circles, ellipses, polygons, etc. The parameters such as *center* and *vertices* are also generated randomly.
- Before finishing, we also should take a look at the functions *Display\_Random\_Text* and *Displaying\_Big\_End*, since they both have a few interesting features:
- 7. Display\_Random\_Text:**

```
int Displaying_Random_Text( Mat image, char* window_name, RNG rng )
{
    int lineHeight = 8;

    for ( int i = 1; i < NUMBER; i++ )
    {
        Point org;
        org.x = rng.uniform(x_1, x_2);
        org.y = rng.uniform(y_1, y_2);

        putText( image, "Testing text rendering", org, rng.uniform(0,8),
                rng.uniform(0,100)*0.05+0.1, randomColor(rng), rng.uniform(1, 10), lineHeight);

        imshow( window_name, image );
        if( waitKey(DELAY) >= 0 )
            { return -1; }
    }

    return 0;
}
```

Everything looks familiar but the expression:

```
putText( image, "Testing text rendering", org, rng.uniform(0,8),
        rng.uniform(0,100)*0.05+0.1, randomColor(rng), rng.uniform(1, 10), lineHeight);
```

So, what does the function `putText` do? In our example:

- Draws the text “Testing text rendering” in **image**
- The bottom-left corner of the text will be located in the Point **org**
- The font type is a random integer value in the range:  $[0, 8 >$ .
- The scale of the font is denoted by the expression **`rng.uniform(0, 100)x0.05 + 0.1`** (meaning its range is:  $[0.1, 5.1 >$ )
- The text color is random (denoted by **`randomColor(rng)`**)
- The text thickness ranges between 1 and 10, as specified by **`rng.uniform(1,10)`**

As a result, we will get (analogously to the other drawing functions) **NUMBER** texts over our image, in random locations.

- 8. Displaying\_Big\_End**

```
int Displaying_Big_End( Mat image, char* window_name, RNG rng )
{
    Size textSize = getTextSize("OpenCV forever!", CV_FONT_HERSHEY_COMPLEX, 3, 5, 0);
    Point org((window_width - textSize.width)/2, (window_height - textSize.height)/2);
    int lineHeight = 8;
```

```

Mat image2;

for( int i = 0; i < 255; i += 2 )
{
    image2 = image - Scalar::all(i);
    putText( image2, "OpenCV forever!", org, CV_FONT_HERSHEY_COMPLEX, 3,
             Scalar(i, i, 255), 5, lineType );

    imshow( window_name, image2 );
    if( waitKey(DELAY) >= 0 )
    { return -1; }
}

return 0;
}

```

Besides the function `getTextSize` (which gets the size of the argument text), the new operation we can observe is inside the *for* loop:

```
image2 = image - Scalar::all(i)
```

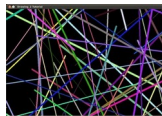
So, **image2** is the subtraction of **image** and **Scalar::all(i)**. In fact, what happens here is that every pixel of **image2** will be the result of subtracting every pixel of **image** minus the value of **i** (remember that for each pixel we are considering three values such as R, G and B, so each of them will be affected)

Also remember that the subtraction operation *always* performs internally a **saturate** operation, which means that the result obtained will always be inside the allowed range (no negative and between 0 and 255 for our example).

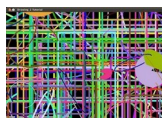
## Result

As you just saw in the Code section, the program will sequentially execute diverse drawing functions, which will produce:

1. First a random set of *NUMBER* lines will appear on screen such as it can be seen in this screenshot:



2. Then, a new set of figures, these time *rectangles* will follow.
3. Now some ellipses will appear, each of them with random position, size, thickness and arc length:



4. Now, *polylines* with 03 segments will appear on screen, again in random configurations.



5. Filled polygons (in this example triangles) will follow.
6. The last geometric figure to appear: circles!



7. Near the end, the text “*Testing Text Rendering*” will appear in a variety of fonts, sizes, colors and positions.
8. And the big end (which by the way expresses a big truth too):



## 2.7 Mask operations on matrixes

Mask operations on matrixes are quite simple. The idea is that we recalculate each pixels value in an image according to a matrix mask. This mask holds values that will just how much influence have neighbor pixel values (and the pixel value itself) to the new pixel value. From a mathematical point of view we make a weighted average, with our specified values.

### Our test case

Let us consider the issue of an image contrast enhancement method. Basically we want to apply for every pixel of the image the following formula:

$$I(i, j) = 5 * I(i, j) - [I(i - 1, j) + I(i + 1, j) + I(i, j - 1) + I(i, j + 1)]$$

$$\iff I(i, j) * M, \text{ where } M = \begin{matrix} & i \setminus j & -1 & 0 & -1 \\ -1 & \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} & & & \\ +1 & & & & \end{matrix}$$

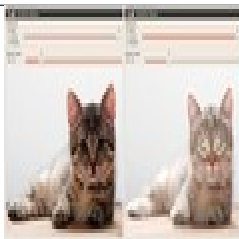
The first notation is by using a formula, while the second is a compacted version of the first by using a mask. You use the mask by putting the center of the mask matrix (in the upper case noted by the zero-zero index) on the pixel you want to calculate and sum up the pixel values multiplied with the overlapped matrix values. It's the same thing, however in case of large matrices the later notation is a lot easier to look over.

# IMGPROC MODULE. IMAGE PROCESSING

In this section you will learn about the image processing (manipulation) functions inside OpenCV.



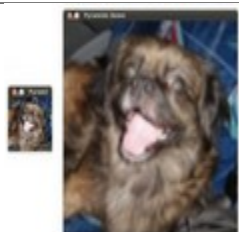
**Title:** *Smoothing Images*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Let's take a look at some basic linear filters!



**Title:** *Eroding and Dilating*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Let's *change* the shape of objects!



**Title:** *More Morphology Transformations*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Here we investigate different morphology operators



**Title:** *Image Pyramids*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
What if I need a bigger/smaller image?





**Title:** *Basic Thresholding Operations*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

After so much processing, it is time to decide which pixels stay!

---



**Title:** *Making your own linear filters!*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn to design our own filters by using OpenCV functions

---



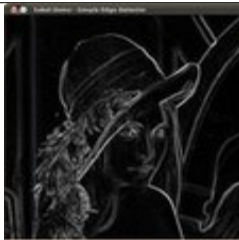
**Title:** *Adding borders to your images*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to pad our images!

---



**Title:** *Sobel Derivatives*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to calculate gradients and use them to detect edges!

---



**Title:** *Laplace Operator*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn about the *Laplace* operator and how to detect edges with it.

---



**Title:** *Canny Edge Detector*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn a sophisticated alternative to detect edges.

---



• **Title:** *Hough Line Transform*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we learn how to detect lines



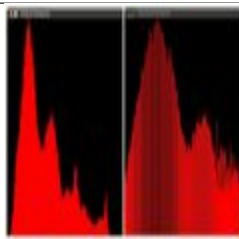
• **Title:** *Hough Circle Transform*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we learn how to detect circles



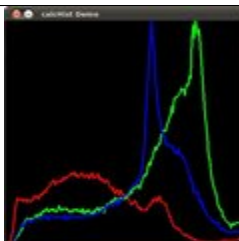
• **Title:** *Remapping*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we learn how to manipulate pixels locations



• **Title:** *Affine Transformations*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we learn how to rotate, translate and scale our images



• **Title:** *Histogram Equalization*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we learn how to improve the contrast in our images



• **Title:** *Histogram Calculation*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we learn how to create and generate histograms

•  
0.930766 0.182073 0.120447  
0.22609 0.646576 0.801869

**Title:** *Histogram Comparison*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn to calculate metrics between histograms

---



**Title:** *Back Projection*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to use histograms to find similar objects in images

---



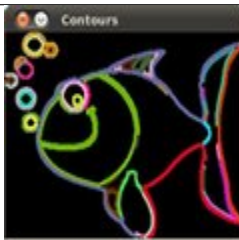
**Title:** *Template Matching*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to match templates in an image

---



**Title:** *Finding contours in your image*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to find contours of objects in our image

---



**Title:** *Convex Hull*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to get hull contours and draw them!

---



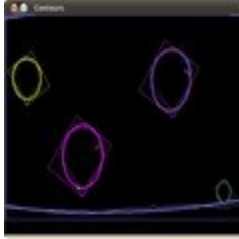
**Title:** *Creating Bounding boxes and circles for contours*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to obtain bounding boxes and circles for our contours.

---



**Title:** *Creating Bounding rotated boxes and ellipses for contours*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to obtain rotated bounding boxes and ellipses for our contours.

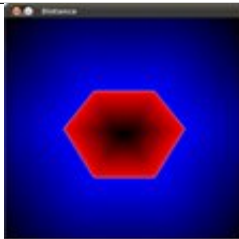


**Title:** *Image Moments*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn to calculate the moments of an image



**Title:** *Point Polygon Test*

*Compatibility:* > OpenCV 2.0

*Author:* Ana Huamán

Where we learn how to calculate distances from the image to contours

## 3.1 Smoothing Images

### Goal

In this tutorial you will learn how to apply diverse linear filters to smooth images using OpenCV functions such as:

- `blur`
- `GaussianBlur`
- `medianBlur`
- `bilateralFilter`

### Theory

---

**Note:** The explanation below belongs to the book *Computer Vision: Algorithms and Applications* by Richard Szeliski and to *LearningOpenCV*

---

- *Smoothing*, also called *blurring*, is a simple and frequently used image processing operation.
- There are many reasons for smoothing. In this tutorial we will focus on smoothing in order to reduce noise (other uses will be seen in the following tutorials).
- To perform a smoothing operation we will apply a *filter* to our image. The most common type of filters are *linear*, in which an output pixel's value (i.e.  $g(i, j)$ ) is determined as a weighted sum of input pixel values (i.e.  $f(i + k, j + l)$ ):

$$g(i, j) = \sum_{k,l} f(i + k, j + l)h(k, l)$$

$h(k, l)$  is called the *kernel*, which is nothing more than the coefficients of the filter.

It helps to visualize a *filter* as a window of coefficients sliding across the image.

- There are many kind of filters, here we will mention the most used:

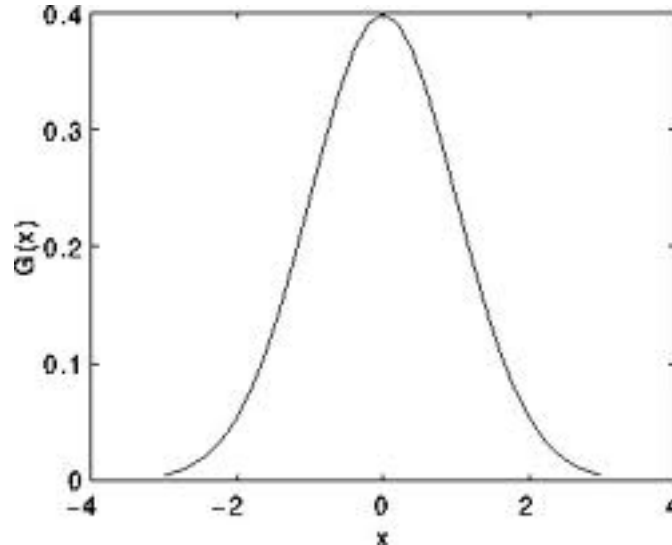
#### Normalized Box Filter

- This filter is the simplest of all! Each output pixel is the *mean* of its kernel neighbors ( all of them contribute with equal weights)
- The kernel is below:

$$K = \frac{1}{K_{\text{width}} \cdot K_{\text{height}}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ \cdot & \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

## Gaussian Filter

- Probably the most useful filter (although not the fastest). Gaussian filtering is done by convolving each point in the input array with a *Gaussian kernel* and then summing them all to produce the output array.
- Just to make the picture clearer, remember how a 1D Gaussian kernel look like?



Assuming that an image is 1D, you can notice that the pixel located in the middle would have the biggest weight. The weight of its neighbors decreases as the spatial distance between them and the center pixel increases.

**Note:** Remember that a 2D Gaussian can be represented as :

$$G_0(x, y) = Ae^{-\frac{(x - \mu_x)^2}{2\sigma_x^2} - \frac{(y - \mu_y)^2}{2\sigma_y^2}}$$

where  $\mu$  is the mean (the peak) and  $\sigma$  represents the variance (per each of the variables  $x$  and  $y$ )

## Median Filter

The median filter run through each element of the signal (in this case the image) and replace each pixel with the **median** of its neighboring pixels (located in a square neighborhood around the evaluated pixel).

## Bilateral Filter

- So far, we have explained some filters which main goal is to *smooth* an input image. However, sometimes the filters do not only dissolve the noise, but also smooth away the *edges*. To avoid this (at certain extent at least), we can use a bilateral filter.
- In an analogous way as the Gaussian filter, the bilateral filter also considers the neighboring pixels with weights assigned to each of them. These weights have two components, the first of which is the same weighting used by the Gaussian filter. The second component takes into account the difference in intensity between the neighboring pixels and the evaluated one.
- For a more detailed explanation you can check [this link](#)

## Code

- **What does this program do?**
  - Loads an image
  - Applies 4 different kinds of filters (explained in Theory) and show the filtered images sequentially
- **Downloadable code:** [Click here](#)
- **Code at glance:**

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace std;
using namespace cv;

/// Global Variables
int DELAY_CAPTION = 1500;
int DELAY_BLUR = 100;
int MAX_KERNEL_LENGTH = 31;

Mat src; Mat dst;
char window_name[] = "Filter Demo 1";

/// Function headers
int display_caption( char* caption );
int display_dst( int delay );

/**
 * function main
 */
int main( int argc, char** argv )
{
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );

    /// Load the source image
    src = imread( "../images/lena.jpg", 1 );

    if( display_caption( "Original Image" ) != 0 ) { return 0; }

    dst = src.clone();
    if( display_dst( DELAY_CAPTION ) != 0 ) { return 0; }

    /// Applying Homogeneous blur
    if( display_caption( "Homogeneous Blur" ) != 0 ) { return 0; }

    for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
        { blur( src, dst, Size( i, i ), Point(-1,-1) );
          if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

    /// Applying Gaussian blur
    if( display_caption( "Gaussian Blur" ) != 0 ) { return 0; }

    for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
        { GaussianBlur( src, dst, Size( i, i ), 0, 0 );
          if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

    /// Applying Median blur
```

```

if( display_caption( "Median Blur" ) != 0 ) { return 0; }

for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
    { medianBlur ( src, dst, i );
      if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

// Applying Bilateral Filter
if( display_caption( "Bilateral Blur" ) != 0 ) { return 0; }

for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
    { bilateralFilter ( src, dst, i, i*2, i/2 );
      if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

// Wait until user press a key
display_caption( "End: Press a key!" );

waitKey(0);
return 0;
}

int display_caption( char* caption )
{
    dst = Mat::zeros( src.size(), src.type() );
    putText( dst, caption,
             Point( src.cols/4, src.rows/2),
             CV_FONT_HERSHEY_COMPLEX, 1, Scalar(255, 255, 255) );

    imshow( window_name, dst );
    int c = waitKey( DELAY_CAPTION );
    if( c >= 0 ) { return -1; }
    return 0;
}

int display_dst( int delay )
{
    imshow( window_name, dst );
    int c = waitKey ( delay );
    if( c >= 0 ) { return -1; }
    return 0;
}

```

## Explanation

1. Let's check the OpenCV functions that involve only the smoothing procedure, since the rest is already known by now.
2. **Normalized Block Filter:**

OpenCV offers the function `blur` to perform smoothing with this filter.

```

for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
    { blur( src, dst, Size( i, i ), Point(-1,-1) );
      if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

```

We specify 4 arguments (more details, check the Reference):

- `src`: Source image



- *dst*: Destination image
- *Size( w, h )*: Defines the size of the kernel to be used ( of width *w* pixels and height *h* pixels)
- *Point(-1, -1)*: Indicates where the anchor point (the pixel evaluated) is located with respect to the neighborhood. If there is a negative value, then the center of the kernel is considered the anchor point.

### 3. Gaussian Filter:

It is performed by the function `GaussianBlur` :

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ GaussianBlur( src, dst, Size( i, i ), 0, 0 );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

Here we use 4 arguments (more details, check the OpenCV reference):

- *src*: Source image
- *dst*: Destination image
- *Size(w, h)*: The size of the kernel to be used (the neighbors to be considered). *w* and *h* have to be odd and positive numbers otherwise thi size will be calculated using the  $\sigma_x$  and  $\sigma_y$  arguments.
- $\sigma_x$ : The standard deviation in x. Writing 0 implies that  $\sigma_x$  is calculated using kernel size.
- $\sigma_y$ : The standard deviation in y. Writing 0 implies that  $\sigma_y$  is calculated using kernel size.

### 4. Median Filter:

This filter is provided by the `medianBlur` function:

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ medianBlur ( src, dst, i );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

We use three arguments:

- *src*: Source image
- *dst*: Destination image, must be the same type as *src*
- *i*: Size of the kernel (only one because we use a square window). Must be odd.

### 5. Bilateral Filter

Provided by OpenCV function `bilateralFilter`

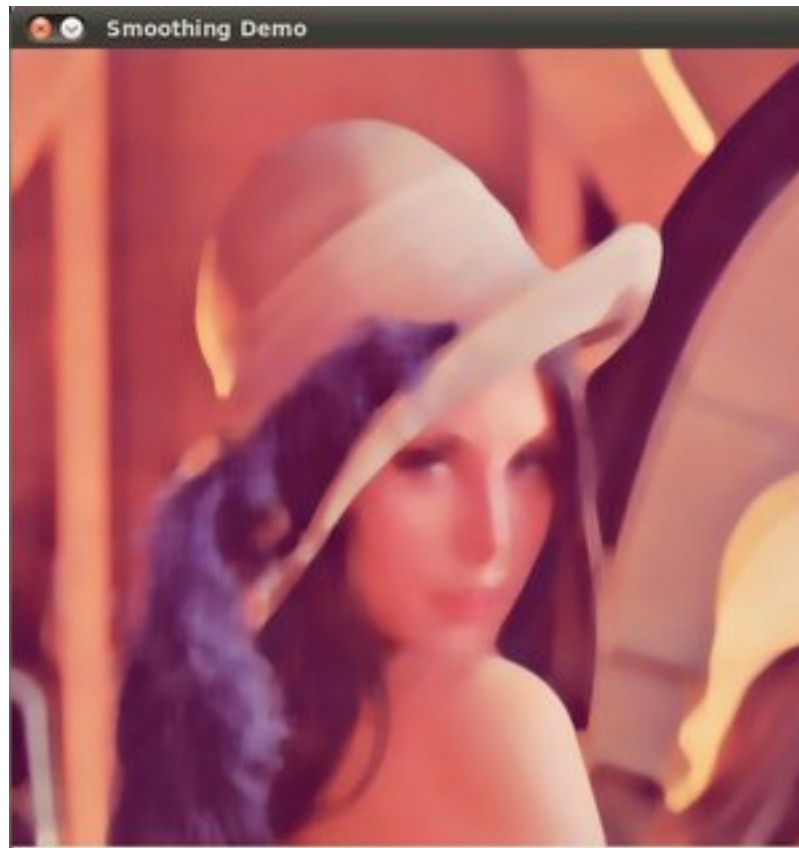
```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ bilateralFilter ( src, dst, i, i*2, i/2 );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

We use 5 arguments:

- *src*: Source image
- *dst*: Destination image
- *d*: The diameter of each pixel neighborhood.
- $\sigma_{\text{Color}}$ : Standard deviation in the color space.
- $\sigma_{\text{Space}}$ : Standard deviation in the coordinate space (in pixel terms)

## Results

- The code opens an image (in this case *lena.jpg*) and display it under the effects of the 4 filters explained.
- Here is a snapshot of the image smoothed using *medianBlur*:



## 3.2 Eroding and Dilating

### Goal

In this tutorial you will learn how to:

- Apply two very common morphology operators: Dilation and Erosion. For this purpose, you will use the following OpenCV functions:
  - erode
  - dilate

### Cool Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

---

## Morphological Operations

- In short: A set of operations that process images based on shapes. Morphological operations apply a *structuring element* to an input image and generate an output image.
- The most basic morphological operations are two: Erosion and Dilation. They have a wide array of uses, i.e. :
  - Removing noise
  - Isolation of individual elements and joining disparate elements in an image.
  - Finding of intensity bumps or holes in an image
- We will explain dilation and erosion briefly, using the following image as an example:



### Dilation

- This operations consists of convoluting an image  $A$  with some kernel ( $B$ ), which can have any shape or size, usually a square or circle.
- The kernel  $B$  has a defined *anchor point*, usually being the center of the kernel.
- As the kernel  $B$  is scanned over the image, we compute the maximal pixel value overlapped by  $B$  and replace the image pixel in the anchor point position with that maximal value. As you can deduce, this maximizing operation causes bright regions within an image to “grow” (therefore the name *dilation*). Take as an example the image above. Applying dilation we can get:



The background (bright) dilates around the black regions of the letter.

## Erosion

- This operation is the sister of dilation. What this does is to compute a local minimum over the area of the kernel.
- As the kernel B is scanned over the image, we compute the minimal pixel value overlapped by B and replace the image pixel under the anchor point with that minimal value.
- Analogously to the example for dilation, we can apply the erosion operator to the original image (shown above). You can see in the result below that the bright areas of the image (the background, apparently), get thinner, whereas the dark zones (the “writing”) gets bigger.



## Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "highgui.h"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/// Global variables
Mat src, erosion_dst, dilation_dst;

int erosion_elem = 0;
int erosion_size = 0;
int dilation_elem = 0;
int dilation_size = 0;
int const max_elem = 2;
int const max_kernel_size = 21;

/** Function Headers */
void Erosion( int, void* );
void Dilation( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load an image
    src = imread( argv[1] );
```

```

if( !src.data )
{ return -1; }

/// Create windows
namedWindow( "Erosion Demo", CV_WINDOW_AUTOSIZE );
namedWindow( "Dilation Demo", CV_WINDOW_AUTOSIZE );
cvMoveWindow( "Dilation Demo", src.cols, 0 );

/// Create Erosion Trackbar
createTrackbar( "Element:\n 0: Rect \n 1: Cross \n 2: Ellipse", "Erosion Demo",
               &erosion_elem, max_elem,
               Erosion );

createTrackbar( "Kernel size:\n 2n +1", "Erosion Demo",
               &erosion_size, max_kernel_size,
               Erosion );

/// Create Dilation Trackbar
createTrackbar( "Element:\n 0: Rect \n 1: Cross \n 2: Ellipse", "Dilation Demo",
               &dilation_elem, max_elem,
               Dilation );

createTrackbar( "Kernel size:\n 2n +1", "Dilation Demo",
               &dilation_size, max_kernel_size,
               Dilation );

/// Default start
Erosion( 0, 0 );
Dilation( 0, 0 );

waitKey(0);
return 0;
}

/** @function Erosion */
void Erosion( int, void* )
{
    int erosion_type;
    if( erosion_elem == 0 ){ erosion_type = MORPH_RECT; }
    else if( erosion_elem == 1 ){ erosion_type = MORPH_CROSS; }
    else if( erosion_elem == 2 ) { erosion_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( erosion_type,
                                       Size( 2*erosion_size + 1, 2*erosion_size+1 ),
                                       Point( erosion_size, erosion_size ) );

    /// Apply the erosion operation
    erode( src, erosion_dst, element );
    imshow( "Erosion Demo", erosion_dst );
}

/** @function Dilation */
void Dilation( int, void* )
{
    int dilation_type;
    if( dilation_elem == 0 ){ dilation_type = MORPH_RECT; }
    else if( dilation_elem == 1 ){ dilation_type = MORPH_CROSS; }
    else if( dilation_elem == 2 ) { dilation_type = MORPH_ELLIPSE; }

```

```

Mat element = getStructuringElement( dilation_type,
                                   Size( 2*dilation_size + 1, 2*dilation_size+1 ),
                                   Point( dilation_size, dilation_size ) );

/// Apply the dilation operation
dilate( src, dilation_dst, element );
imshow( "Dilation Demo", dilation_dst );
}

```

## Explanation

1. Most of the stuff shown is known by you (if you have any doubt, please refer to the tutorials in previous sections). Let's check the general structure of the program:

- Load an image (can be RGB or grayscale)
- Create two windows (one for dilation output, the other for erosion)
- Create a set of 02 Trackbars for each operation:
  - The first trackbar “Element” returns either **erosion\_elem** or **dilation\_elem**
  - The second trackbar “Kernel size” return **erosion\_size** or **dilation\_size** for the corresponding operation.
- Every time we move any slider, the user's function **Erosion** or **Dilation** will be called and it will update the output image based on the current trackbar values.

Let's analyze these two functions:

2. **erosion:**

```

/** @function Erosion */
void Erosion( int, void* )
{
    int erosion_type;
    if( erosion_elem == 0 ){ erosion_type = MORPH_RECT; }
    else if( erosion_elem == 1 ){ erosion_type = MORPH_CROSS; }
    else if( erosion_elem == 2) { erosion_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( erosion_type,
                                       Size( 2*erosion_size + 1, 2*erosion_size+1 ),
                                       Point( erosion_size, erosion_size ) );

    /// Apply the erosion operation
    erode( src, erosion_dst, element );
    imshow( "Erosion Demo", erosion_dst );
}

```

- The function that performs the *erosion* operation is `erode`. As we can see, it receives three arguments:
  - *src*: The source image
  - *erosion\_dst*: The output image
  - *element*: This is the kernel we will use to perform the operation. If we do not specify, the default is a simple 3x3 matrix. Otherwise, we can specify its shape. For this, we need to use the function `getStructuringElement`:

```

Mat element = getStructuringElement( erosion_type,
                                   Size( 2*erosion_size + 1, 2*erosion_size+1 ),
                                   Point( erosion_size, erosion_size ) );

```

We can choose any of three shapes for our kernel:

- \* Rectangular box: MORPH\_RECT
- \* Cross: MORPH\_CROSS
- \* Ellipse: MORPH\_ELLIPSE

Then, we just have to specify the size of our kernel and the *anchor point*. If not specified, it is assumed to be in the center.

- That is all. We are ready to perform the erosion of our image.

---

**Note:** Additionally, there is another parameter that allows you to perform multiple erosions (iterations) at once. We are not using it in this simple tutorial, though. You can check out the Reference for more details.

---

### 3. dilation:

The code is below. As you can see, it is completely similar to the snippet of code for **erosion**. Here we also have the option of defining our kernel, its anchor point and the size of the operator to be used.

```
/** @function Dilation */
void Dilation( int, void* )
{
    int dilation_type;
    if( dilation_elem == 0 ){ dilation_type = MORPH_RECT; }
    else if( dilation_elem == 1 ){ dilation_type = MORPH_CROSS; }
    else if( dilation_elem == 2) { dilation_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( dilation_type,
                                       Size( 2*dilation_size + 1, 2*dilation_size+1 ),
                                       Point( dilation_size, dilation_size ) );

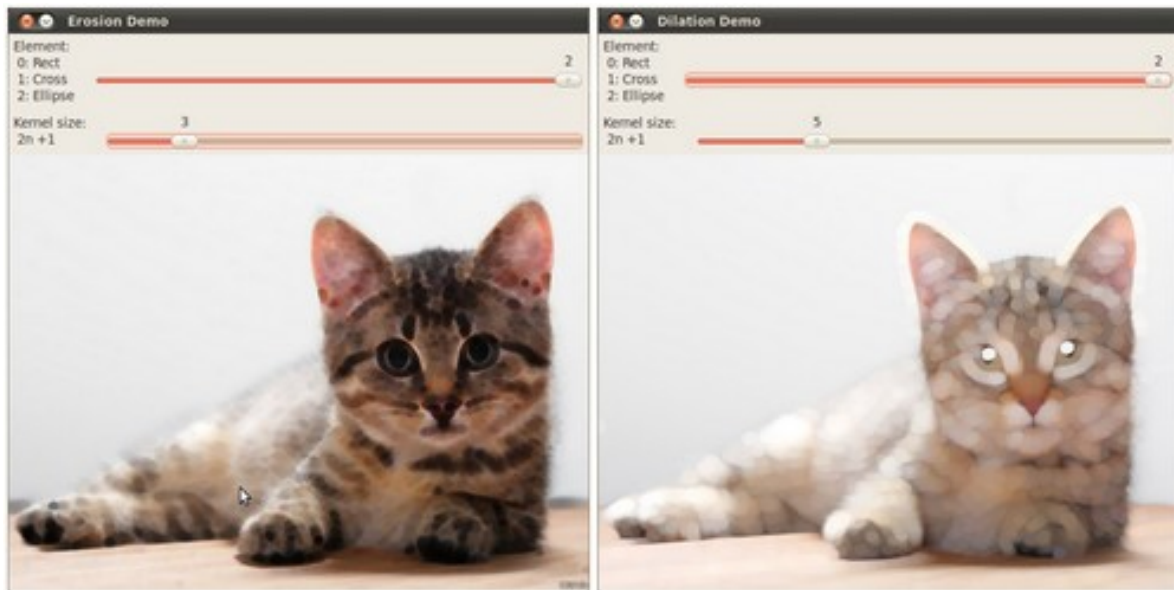
    /// Apply the dilation operation
    dilate( src, dilation_dst, element );
    imshow( "Dilation Demo", dilation_dst );
}
```

## Results

- Compile the code above and execute it with an image as argument. For instance, using this image:



We get the results below. Varying the indices in the Trackbars give different output images, naturally. Try them out! You can even try to add a third Trackbar to control the number of iterations.



### 3.3 More Morphology Transformations

#### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `morphologyEx` to apply Morphological Transformation such as:
  - Opening
  - Closing



- Morphological Gradient
- Top Hat
- Black Hat

### Cool Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

---

In the previous tutorial we covered two basic Morphology operations:

- Erosion
- Dilation.

Based on these two we can effectuate more sophisticated transformations to our images. Here we discuss briefly 05 operations offered by OpenCV:

#### Opening

- It is obtained by the erosion of an image followed by a dilation.

$$\text{dst} = \text{open}(\text{src}, \text{element}) = \text{dilate}(\text{erode}(\text{src}, \text{element}))$$

- Useful for removing small objects (it is assumed that the objects are bright on a dark foreground)
- For instance, check out the example below. The image at the left is the original and the image at the right is the result after applying the opening transformation. We can observe that the small spaces in the corners of the letter tend to disappear.



#### Closing

- It is obtained by the dilation of an image followed by an erosion.

$$\text{dst} = \text{close}(\text{src}, \text{element}) = \text{erode}(\text{dilate}(\text{src}, \text{element}))$$

- Useful to remove small holes (dark regions).



### Morphological Gradient

- It is the difference between the dilation and the erosion of an image.

$$\text{dst} = \text{morph}_{\text{grad}}(\text{src}, \text{element}) = \text{dilate}(\text{src}, \text{element}) - \text{erode}(\text{src}, \text{element})$$

- It is useful for finding the outline of an object as can be seen below:



### Top Hat

- It is the difference between an input image and its opening.

$$\text{dst} = \text{tophat}(\text{src}, \text{element}) = \text{src} - \text{open}(\text{src}, \text{element})$$



### Black Hat

- It is the difference between the closing and its input image

$$\text{dst} = \text{blackhat}(\text{src}, \text{element}) = \text{close}(\text{src}, \text{element}) - \text{src}$$



### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/// Global variables
Mat src, dst;

int morph_elem = 0;
int morph_size = 0;
int morph_operator = 0;
int const max_operator = 4;
int const max_elem = 2;
int const max_kernel_size = 21;

char* window_name = "Morphology Transformations Demo";

/** Function Headers */
void Morphology_Operations( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load an image
    src = imread( argv[1] );

    if( !src.data )
    { return -1; }

    /// Create window
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );
```

```

/// Create Tracker to select Morphology operation
createTracker("Operator:\n 0: Opening - 1: Closing \n 2: Gradient - 3: Top Hat \n 4: Black Hat", window_name, &morph_
operator, max_operator, Morphology_Operations );

/// Create Tracker to select kernel type
createTracker( "Element:\n 0: Rect - 1: Cross - 2: Ellipse", window_name,
&morph_elem, max_elem,
Morphology_Operations );

/// Create Tracker to choose kernel size
createTracker( "Kernel size:\n 2n +1", window_name,
&morph_size, max_kernel_size,
Morphology_Operations );

/// Default start
Morphology_Operations( 0, 0 );

waitKey(0);
return 0;
}

/**
 * @function Morphology_Operations
 */
void Morphology_Operations( int, void* )
{
    // Since MORPH_X : 2,3,4,5 and 6
    int operation = morph_operator + 2;

    Mat element = getStructuringElement( morph_elem, Size( 2*morph_size + 1, 2*morph_size+1 ), Point( morph_size, morph_size ));

    /// Apply the specified morphology operation
    morphologyEx( src, dst, operation, element );
    imshow( window_name, dst );
}

```

## Explanation

1. Let's check the general structure of the program:

- Load an image
- Create a window to display results of the Morphological operations
- Create 03 Trackbars for the user to enter parameters:
  - The first tracker “**Operator**” returns the kind of morphology operation to use (**morph\_operator**).

```

createTracker("Operator:\n 0: Opening - 1: Closing \n 2: Gradient - 3: Top Hat \n 4: Black Hat",
window_name, &morph_operator, max_operator,
Morphology_Operations );

```

- The second tracker “**Element**” returns **morph\_elem**, which indicates what kind of structure our kernel is:

```

createTracker( "Element:\n 0: Rect - 1: Cross - 2: Ellipse", window_name,
&morph_elem, max_elem,
Morphology_Operations );

```

- The final trackbar “**Kernel Size**” returns the size of the kernel to be used (**morph\_size**)

```
createTrackbar( "Kernel size:\n 2n +1", window_name,
               &morph_size, max_kernel_size,
               Morphology_Operations );
```

- Every time we move any slider, the user’s function **Morphology\_Operations** will be called to effectuate a new morphology operation and it will update the output image based on the current trackbar values.

```
/**
 * @function Morphology_Operations
 */
void Morphology_Operations( int, void* )
{
    // Since MORPH_X : 2,3,4,5 and 6
    int operation = morph_operator + 2;

    Mat element = getStructuringElement( morph_elem, Size( 2*morph_size + 1, 2*morph_size+1 ), Point( morph_s

    /// Apply the specified morphology operation
    morphologyEx( src, dst, operation, element );
    imshow( window_name, dst );
}
```

We can observe that the key function to perform the morphology transformations is `morphologyEx`. In this example we use four arguments (leaving the rest as defaults):

- **src** : Source (input) image
- **dst**: Output image
- **operation**: The kind of morphology transformation to be performed. Note that we have 5 alternatives:
  - \* *Opening*: MORPH\_OPEN : 2
  - \* *Closing*: MORPH\_CLOSE: 3
  - \* *Gradient*: MORPH\_GRADIENT: 4
  - \* *Top Hat*: MORPH\_TOPHAT: 5
  - \* *Black Hat*: MORPH\_BLACKHAT: 6

As you can see the values range from <2-6>, that is why we add (+2) to the values entered by the Trackbar:

```
int operation = morph_operator + 2;
```

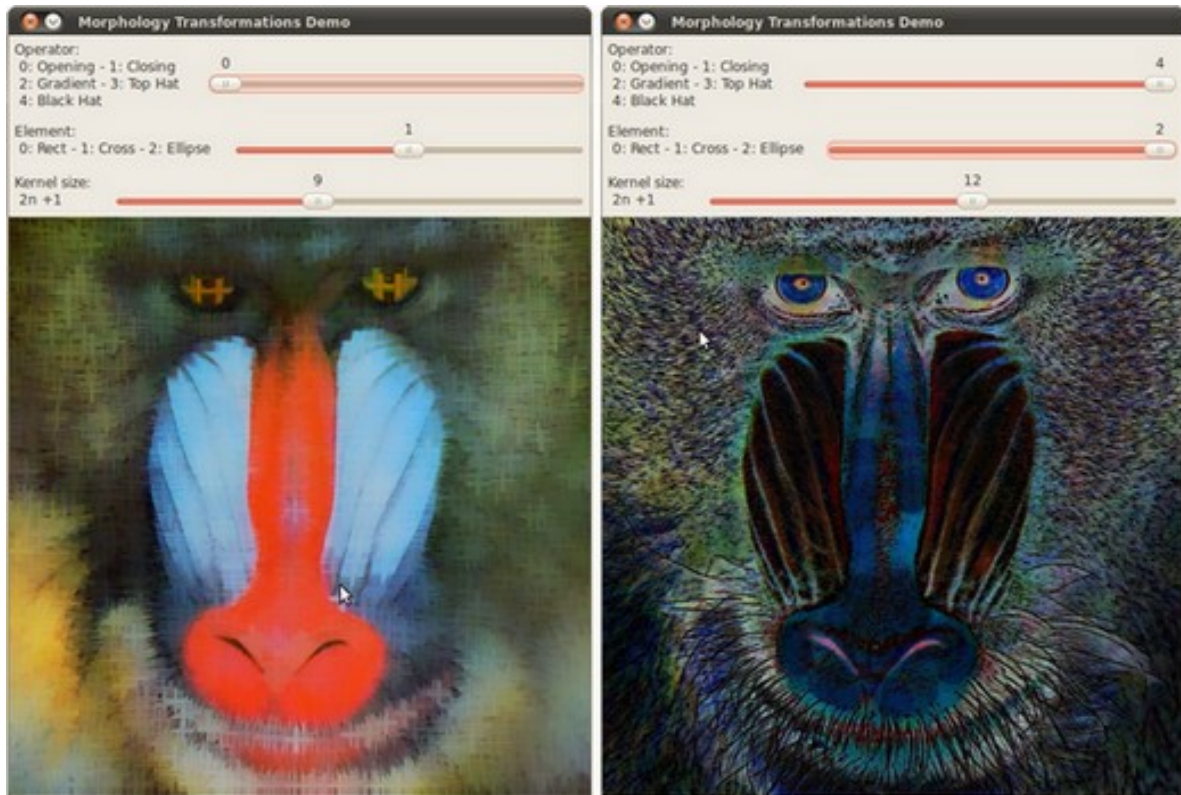
- **element**: The kernel to be used. We use the function `getStructuringElement` to define our own structure.

## Results

- After compiling the code above we can execute it giving an image path as an argument. For this tutorial we use as input the image: **baboon.png**:



- And here are two snapshots of the display window. The first picture shows the output after using the operator **Opening** with a cross kernel. The second picture (right side, shows the result of using a **Blackhat** operator with an ellipse kernel.



## 3.4 Image Pyramids

### Goal

In this tutorial you will learn how to:

- Use the OpenCV functions `pyrUp` and `pyrDown` to downsample or upsample a given image.

### Theory

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

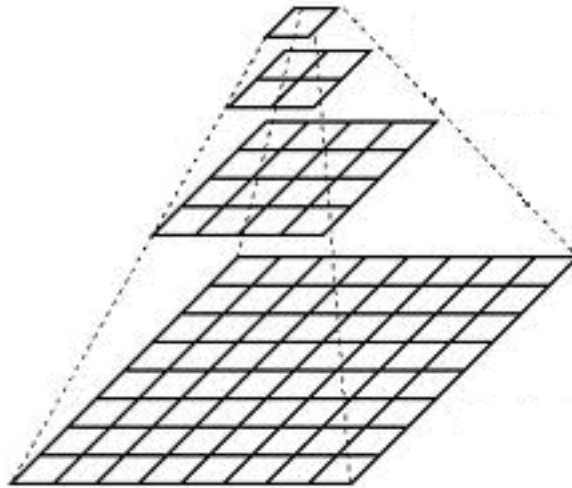
- Usually we need to convert an image to a size different than its original. For this, there are two possible options:
  - *Upsize* the image (zoom in) or
  - *Downsize* it (zoom out).
- Although there is a *geometric transformation* function in OpenCV that -literally- resize an image (`resize`, which we will show in a future tutorial), in this section we analyze first the use of **Image Pyramids**, which are widely applied in a huge range of vision applications.

## Image Pyramid

- An image pyramid is a collection of images - all arising from a single original image - that are successively downsampled until some desired stopping point is reached.
- There are two common kinds of image pyramids:
  - **Gaussian pyramid:** Used to downsample images
  - **Laplacian pyramid:** Used to reconstruct an upsampled image from an image lower in the pyramid (with less resolution)
- In this tutorial we'll use the *Gaussian pyramid*.

## Gaussian Pyramid

- Imagine the pyramid as a set of layers in which the higher the layer, the smaller the size.



- Every layer is numbered from bottom to top, so layer  $(i + 1)$  (denoted as  $G_{i+1}$ ) is smaller than layer  $i$  ( $G_i$ ).
- To produce layer  $(i + 1)$  in the Gaussian pyramid, we do the following:
  - Convolve  $G_i$  with a Gaussian kernel:

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

- Remove every even-numbered row and column.
- You can easily notice that the resulting image will be exactly one-quarter the area of its predecessor. Iterating this process on the input image  $G_0$  (original image) produces the entire pyramid.
- The procedure above was useful to downsample an image. What if we want to make it bigger?:
  - First, upsize the image to twice the original in each dimension, with the new even rows and columns filled with zeros (0)
  - Perform a convolution with the same kernel shown above (multiplied by 4) to approximate the values of the “missing pixels”

- These two procedures (downsampling and upsampling as explained above) are implemented by the OpenCV functions `pyrUp` and `pyrDown`, as we will see in an example with the code below:

---

**Note:** When we reduce the size of an image, we are actually *losing* information of the image.

---

## Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/// Global variables
Mat src, dst, tmp;
char* window_name = "Pyramids Demo";

/**
 * @function main
 */
int main( int argc, char** argv )
{
    /// General instructions
    printf( "\n Zoom In-Out demo \n " );
    printf( "----- \n" );
    printf( " * [u] -> Zoom in \n" );
    printf( " * [d] -> Zoom out \n" );
    printf( " * [ESC] -> Close program \n \n" );

    /// Test image - Make sure it s divisible by 2^{n}
    src = imread( "../images/chicky_512.jpg" );
    if( !src.data )
        { printf(" No data! -- Exiting the program \n");
          return -1; }

    tmp = src;
    dst = tmp;

    /// Create window
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );
    imshow( window_name, dst );

    /// Loop
    while( true )
    {
        int c;
        c = waitKey(10);

        if( (char)c == 27 )
            { break; }
        if( (char)c == 'u' )
```



```

        { pyrUp( tmp, dst, Size( tmp.cols*2, tmp.rows*2 ) );
          printf( "** Zoom In: Image x 2 \n" );
        }
    else if( (char)c == 'd' )
    { pyrDown( tmp, dst, Size( tmp.cols/2, tmp.rows/2 ) );
      printf( "** Zoom Out: Image / 2 \n" );
    }

    imshow( window_name, dst );
    tmp = dst;
}
return 0;
}

```

## Explanation

1. Let's check the general structure of the program:

- Load an image (in this case it is defined in the program, the user does not have to enter it as an argument)

```

// Test image - Make sure it s divisible by 2^{n}
src = imread( "../images/chicky_512.jpg" );
if( !src.data )
    { printf( " No data! -- Exiting the program \n" );
      return -1; }

```

- Create a Mat object to store the result of the operations (*dst*) and one to save temporal results (*tmp*).

```

Mat src, dst, tmp;
/* ... */
tmp = src;
dst = tmp;

```

- Create a window to display the result

```

namedWindow( window_name, CV_WINDOW_AUTOSIZE );
imshow( window_name, dst );

```

- Perform an infinite loop waiting for user input.

```

while( true )
{
    int c;
    c = waitKey(10);

    if( (char)c == 27 )
    { break; }
    if( (char)c == 'u' )
    { pyrUp( tmp, dst, Size( tmp.cols*2, tmp.rows*2 ) );
      printf( "** Zoom In: Image x 2 \n" );
    }
    else if( (char)c == 'd' )
    { pyrDown( tmp, dst, Size( tmp.cols/2, tmp.rows/2 ) );
      printf( "** Zoom Out: Image / 2 \n" );
    }

    imshow( window_name, dst );
}

```

```
tmp = dst;
}
```

Our program exits if the user presses *ESC*. Besides, it has two options:

- **Perform upsampling (after pressing ‘u’)**

```
pyrUp( tmp, dst, Size( tmp.cols*2, tmp.rows*2 )
```

We use the function `pyrUp` with 03 arguments:

- \* *tmp*: The current image, it is initialized with the *src* original image.
- \* *dst*: The destination image (to be shown on screen, supposedly the double of the input image)
- \* *Size( tmp.cols\*2, tmp.rows\*2 )*: The destination size. Since we are upsampling, `pyrUp` expects a size double than the input image (in this case *tmp*).

- **Perform downsampling (after pressing ‘d’)**

```
pyrDown( tmp, dst, Size( tmp.cols/2, tmp.rows/2 )
```

Similarly as with `pyrUp`, we use the function `pyrDown` with 03 arguments:

- \* *tmp*: The current image, it is initialized with the *src* original image.
- \* *dst*: The destination image (to be shown on screen, supposedly half the input image)
- \* *Size( tmp.cols/2, tmp.rows/2 )*: The destination size. Since we are upsampling, `pyrDown` expects half the size the input image (in this case *tmp*).

- Notice that it is important that the input image can be divided by a factor of two (in both dimensions). Otherwise, an error will be shown.
- Finally, we update the input image **tmp** with the current image displayed, so the subsequent operations are performed on it.

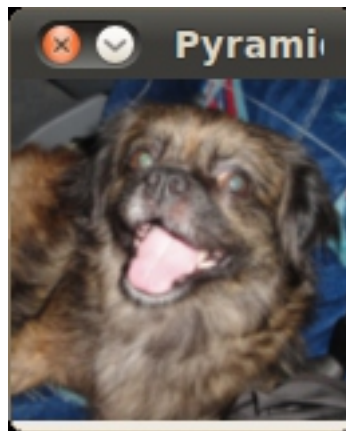
```
tmp = dst;
```

## Results

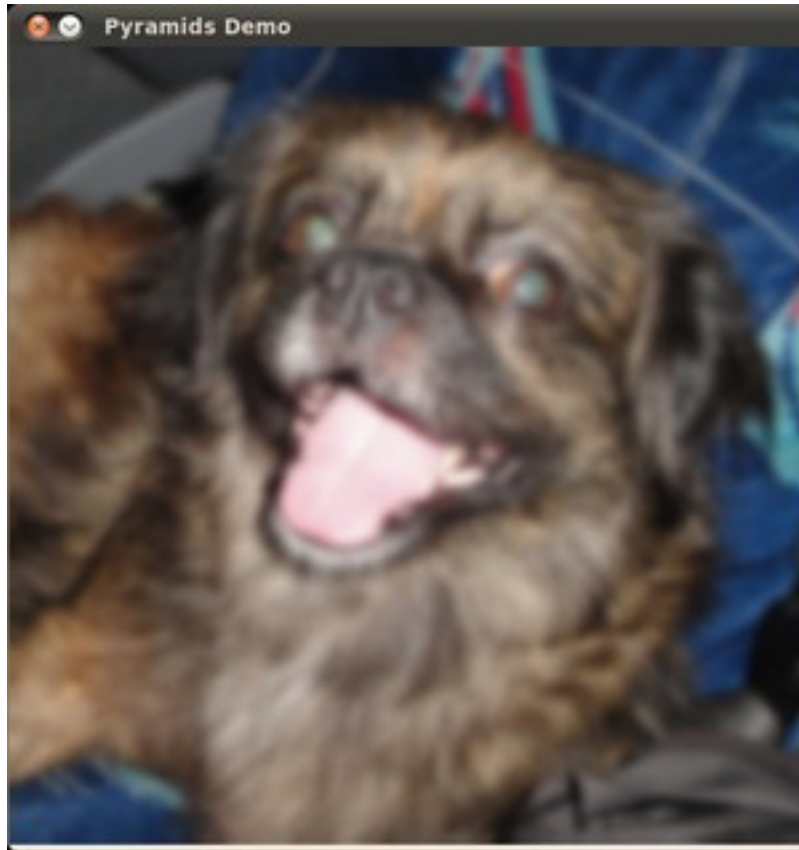
- After compiling the code above we can test it. The program calls an image **chicky\_512.jpg** that comes in the *tutorial\_code/image* folder. Notice that this image is  $512 \times 512$ , hence a downsample won't generate any error ( $512 = 2^9$ ). The original image is shown below:



- First we apply two successive `pyrDown` operations by pressing 'd'. Our output is:



- Note that we should have lost some resolution due to the fact that we are diminishing the size of the image. This is evident after we apply `pyrUp` twice (by pressing 'u'). Our output is now:



## 3.5 Basic Thresholding Operations

### Goal

In this tutorial you will learn how to:

- Perform basic thresholding operations using OpenCV function `threshold`

### Cool Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

---

#### What is Thresholding?

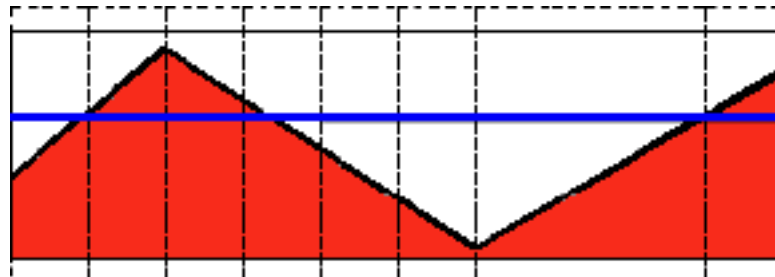
- The simplest segmentation method
- Application example: Separate out regions of an image corresponding to objects which we want to analyze. This separation is based on the variation of intensity between the object pixels and the background pixels.
- To differentiate the pixels we are interested in from the rest (which will eventually be rejected), we perform a comparison of each pixel intensity value with respect to a *threshold* (determined according to the problem to solve).

- Once we have separated properly the important pixels, we can set them with a determined value to identify them (i.e. we can assign them a value of 0 (black), 255 (white) or any value that suits your needs).



### Types of Thresholding

- OpenCV offers the function `threshold` to perform thresholding operations.
- We can effectuate 5 types of Thresholding operations with this function. We will explain them in the following subsections.
- To illustrate how these thresholding processes work, let's consider that we have a source image with pixels with intensity values  $src(x, y)$ . The plot below depicts this. The horizontal blue line represents the threshold `thresh` (fixed).

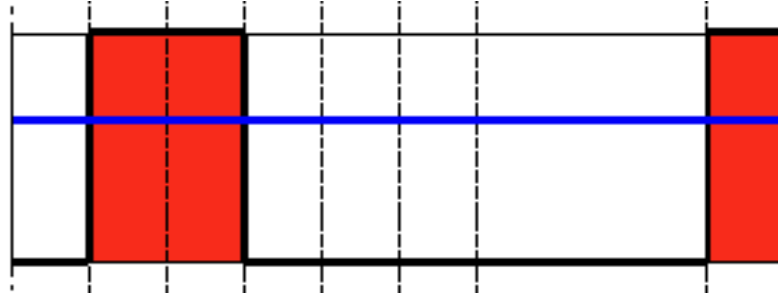


### Threshold Binary

- This thresholding operation can be expressed as:

$$dst(x, y) = \begin{cases} \text{maxVal} & \text{if } src(x, y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- So, if the intensity of the pixel  $src(x, y)$  is higher than `thresh`, then the new pixel intensity is set to a `MaxVal`. Otherwise, the pixels are set to 0.

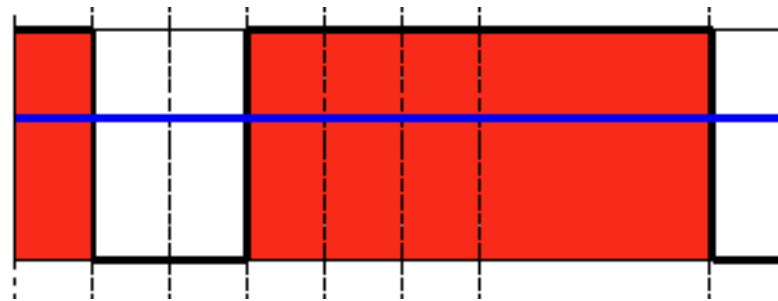


### Threshold Binary, Inverted

- This thresholding operation can be expressed as:

$$\text{dst}(x, y) = \begin{cases} 0 & \text{if } \text{src}(x, y) > \text{thresh} \\ \text{maxVal} & \text{otherwise} \end{cases}$$

- If the intensity of the pixel  $\text{src}(x, y)$  is higher than  $\text{thresh}$ , then the new pixel intensity is set to a 0. Otherwise, it is set to  $\text{MaxVal}$ .

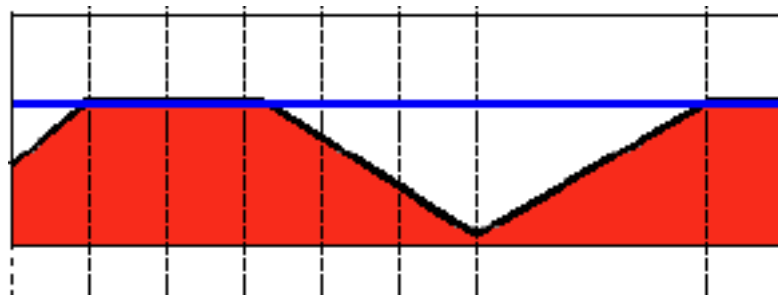


### Truncate

- This thresholding operation can be expressed as:

$$\text{dst}(x, y) = \begin{cases} \text{threshold} & \text{if } \text{src}(x, y) > \text{thresh} \\ \text{src}(x, y) & \text{otherwise} \end{cases}$$

- The maximum intensity value for the pixels is  $\text{thresh}$ , if  $\text{src}(x, y)$  is greater, then its value is *truncated*. See figure below:

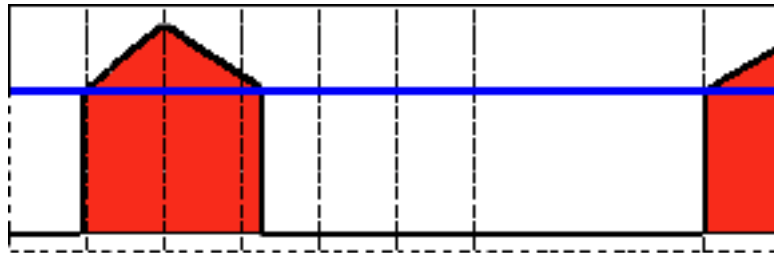


### Threshold to Zero

- This operation can be expressed as:

$$\text{dst}(x, y) = \begin{cases} \text{src}(x, y) & \text{if } \text{src}(x, y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- If  $\text{src}(x, y)$  is lower than  $\text{thresh}$ , the new pixel value will be set to 0.

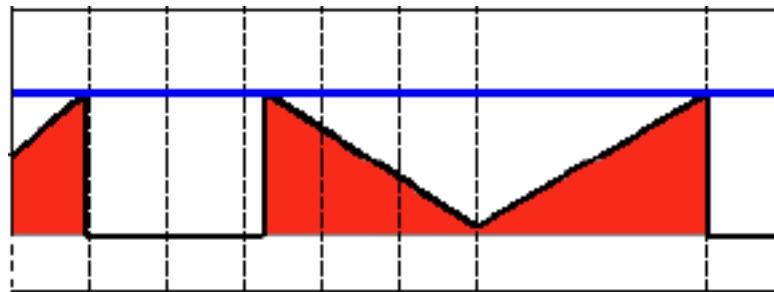


### Threshold to Zero, Inverted

- This operation can be expressed as:

$$\text{dst}(x, y) = \begin{cases} 0 & \text{if } \text{src}(x, y) > \text{thresh} \\ \text{src}(x, y) & \text{otherwise} \end{cases}$$

- If  $\text{src}(x, y)$  is greater than  $\text{thresh}$ , the new pixel value will be set to 0.



### Code

The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/// Global variables
```

```

int threshold_value = 0;
int threshold_type = 3;;
int const max_value = 255;
int const max_type = 4;
int const max_BINARY_value = 255;

Mat src, src_gray, dst;
char* window_name = "Threshold Demo";

char* trackbar_type = "Type: \n 0: Binary \n 1: Binary Inverted \n 2: Truncate \n 3: To Zero \n 4: To Zero Inverted";
char* trackbar_value = "Value";

/// Function headers
void Threshold_Demo( int, void* );

/**
 * @function main
 */
int main( int argc, char** argv )
{
    /// Load an image
    src = imread( argv[1], 1 );

    /// Convert the image to Gray
    cvtColor( src, src_gray, CV_RGB2GRAY );

    /// Create a window to display results
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );

    /// Create Trackbar to choose type of Threshold
    createTrackbar( trackbar_type,
                  window_name, &threshold_type,
                  max_type, Threshold_Demo );

    createTrackbar( trackbar_value,
                  window_name, &threshold_value,
                  max_value, Threshold_Demo );

    /// Call the function to initialize
    Threshold_Demo( 0, 0 );

    /// Wait until user finishes program
    while(true)
    {
        int c;
        c = waitKey( 20 );
        if( (char)c == 27 )
            { break; }
    }
}

/**
 * @function Threshold_Demo
 */
void Threshold_Demo( int, void* )
{

```



```
/* 0: Binary
   1: Binary Inverted
   2: Threshold Truncated
   3: Threshold to Zero
   4: Threshold to Zero Inverted
*/

threshold( src_gray, dst, threshold_value, max_BINARY_value,threshold_type );

imshow( window_name, dst );
}
```

### Explanation

1. Let's check the general structure of the program:

- Load an image. If it is RGB we convert it to Grayscale. For this, remember that we can use the function `cvtColor`:

```
src = imread( argv[1], 1 );

/// Convert the image to Gray
cvtColor( src, src_gray, CV_RGB2GRAY );
```

- Create a window to display the result

```
namedWindow( window_name, CV_WINDOW_AUTOSIZE );
```

- Create 2 trackbars for the user to enter user input:

- **Type of thresholding:** Binary, To Zero, etc...
- **Threshold value**

```
createTrackbar( trackbar_type,
               window_name, &threshold_type,
               max_type, Threshold_Demo );

createTrackbar( trackbar_value,
               window_name, &threshold_value,
               max_value, Threshold_Demo );
```

- Wait until the user enters the threshold value, the type of thresholding (or until the program exits)
- Whenever the user changes the value of any of the Trackbars, the function `Threshold_Demo` is called:

```
/**
 * @function Threshold_Demo
 */
void Threshold_Demo( int, void* )
{
    /* 0: Binary
       1: Binary Inverted
       2: Threshold Truncated
       3: Threshold to Zero
       4: Threshold to Zero Inverted
    */

    threshold( src_gray, dst, threshold_value, max_BINARY_value,threshold_type );
```

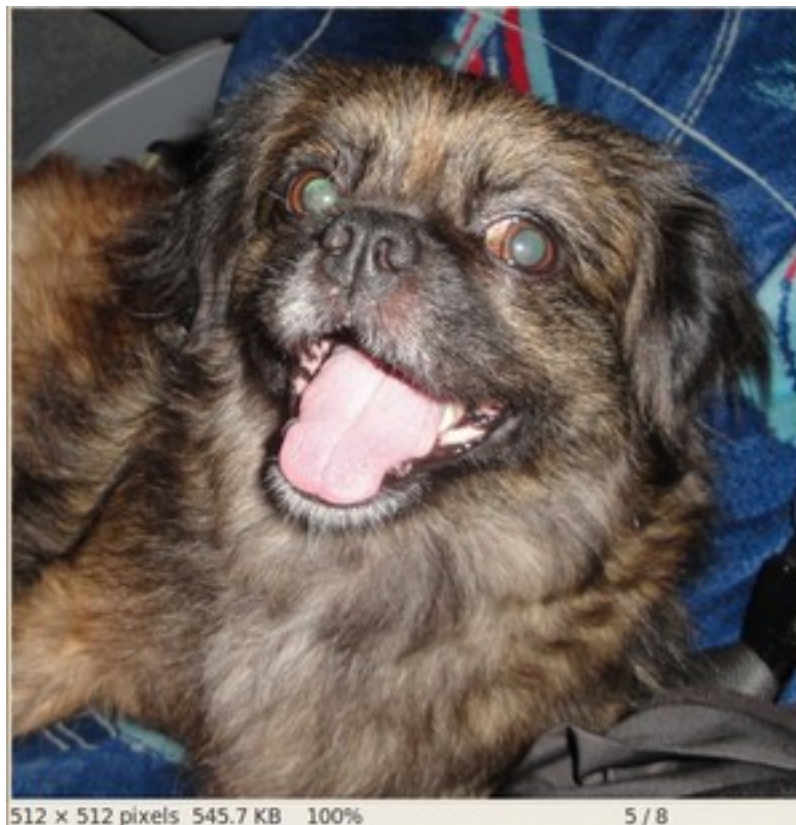
```
imshow( window_name, dst );  
}
```

As you can see, the function `threshold` is invoked. We give 5 parameters:

- `src_gray`: Our input image
- `dst`: Destination (output) image
- `threshold_value`: The thresh value with respect to which the thresholding operation is made
- `max_BINARY_value`: The value used with the Binary thresholding operations (to set the chosen pixels)
- `threshold_type`: One of the 5 thresholding operations. They are listed in the comment section of the function above.

## Results

1. After compiling this program, run it giving a path to an image as argument. For instance, for an input image as:



2. First, we try to threshold our image with a *binary threshold inverted*. We expect that the pixels brighter than the thresh will turn dark, which is what actually happens, as we can see in the snapshot below (notice from the original image, that the doggie's tongue and eyes are particularly bright in comparison with the image, this is reflected in the output image).



3. Now we try with the *threshold to zero*. With this, we expect that the darkest pixels (below the threshold) will become completely black, whereas the pixels with value greater than the threshold will keep its original value. This is verified by the following snapshot of the output image:



## 3.6 Making your own linear filters!

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `filter2D` to create your own linear filters.

### Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

---

### Convolution

In a very general sense, convolution is an operation between every part of an image and an operator (kernel).

#### What is a kernel?

A kernel is essentially a fixed size array of numerical coefficients along with an *anchor point* in that array, which is typically located at the center.

1	-2	1
2		2
1	-2	1

### How does convolution with a kernel work?

Assume you want to know the resulting value of a particular location in the image. The value of the convolution is calculated in the following way:

1. Place the kernel anchor on top of a determined pixel, with the rest of the kernel overlaying the corresponding local pixels in the image.
2. Multiply the kernel coefficients by the corresponding image pixel values and sum the result.
3. Place the result to the location of the *anchor* in the input image.
4. Repeat the process for all pixels by scanning the kernel over the entire image.

Expressing the procedure above in the form of an equation we would have:

$$H(x, y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j)K(i, j)$$

Fortunately, OpenCV provides you with the function `filter2D` so you do not have to code all these operations.

## Code

### 1. What does this program do?

- Loads an image
- Performs a *normalized box filter*. For instance, for a kernel of size `size = 3`, the kernel would be:

$$K = \frac{1}{3 \cdot 3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The program will perform the filter operation with kernels of sizes 3, 5, 7, 9 and 11.

- The filter output (with each kernel) will be shown during 500 milliseconds

### 2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>
```

```

using namespace cv;

/** @function main */
int main ( int argc, char** argv )
{
    /// Declare variables
    Mat src, dst;

    Mat kernel;
    Point anchor;
    double delta;
    int ddepth;
    int kernel_size;
    char* window_name = "filter2D Demo";

    int c;

    /// Load an image
    src = imread( argv[1] );

    if( !src.data )
    { return -1; }

    /// Create window
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );

    /// Initialize arguments for the filter
    anchor = Point( -1, -1 );
    delta = 0;
    ddepth = -1;

    /// Loop - Will filter the image with different kernel sizes each 0.5 seconds
    int ind = 0;
    while( true )
    {
        c = waitKey(500);
        /// Press 'ESC' to exit the program
        if( (char)c == 27 )
            { break; }

        /// Update kernel size for a normalized box filter
        kernel_size = 3 + 2*( ind%5 );
        kernel = Mat::ones( kernel_size, kernel_size, CV_32F )/ (float)(kernel_size*kernel_size);

        /// Apply filter
        filter2D(src, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT );
        imshow( window_name, dst );
        ind++;
    }

    return 0;
}

```

## Explanation

1. Load an image

```
src = imread( argv[1] );
```

```
if( !src.data )  
    { return -1; }
```

2. Create a window to display the result

```
namedWindow( window_name, CV_WINDOW_AUTOSIZE );
```

3. Initialize the arguments for the linear filter

```
anchor = Point( -1, -1 );  
delta = 0;  
ddepth = -1;
```

4. Perform an infinite loop updating the kernel size and applying our linear filter to the input image. Let's analyze that more in detail:

5. First we define the kernel our filter is going to use. Here it is:

```
kernel_size = 3 + 2*( ind%5 );  
kernel = Mat::ones( kernel_size, kernel_size, CV_32F ) / (float)(kernel_size*kernel_size);
```

The first line is to update the *kernel\_size* to odd values in the range: [3, 11]. The second line actually builds the kernel by setting its value to a matrix filled with 1's and normalizing it by dividing it between the number of elements.

6. After setting the kernel, we can generate the filter by using the function `filter2D`:

```
filter2D(src, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT );
```

The arguments denote:

- (a) *src*: Source image
  - (b) *dst*: Destination image
  - (c) *ddepth*: The depth of *dst*. A negative value (such as  $-1$ ) indicates that the depth is the same as the source.
  - (d) *kernel*: The kernel to be scanned through the image
  - (e) *anchor*: The position of the anchor relative to its kernel. The location *Point(-1, -1)* indicates the center by default.
  - (f) *delta*: A value to be added to each pixel during the convolution. By default it is 0
  - (g) *BORDER\_DEFAULT*: We let this value by default (more details in the following tutorial)
7. Our program will effectuate a *while* loop, each 500 ms the kernel size of our filter will be updated in the range indicated.

## Results

1. After compiling the code above, you can execute it giving as argument the path of an image. The result should be a window that shows an image blurred by a normalized filter. Each 0.5 seconds the kernel size should change, as can be seen in the series of snapshots below:



## 3.7 Adding borders to your images

### Goal

In this tutorial you will learn how to:

1. Use the OpenCV function `copyMakeBorder` to set the borders (extra padding to your image).

### Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

---

1. In our previous tutorial we learned to use convolution to operate on images. One problem that naturally arises is how to handle the boundaries. How can we convolve them if the evaluated points are at the edge of the image?
2. What most of OpenCV functions do is to copy a given image onto another slightly larger image and then automatically pads the boundary (by any of the methods explained in the sample code just below). This way, the convolution can be performed over the needed pixels without problems (the extra padding is cut after the operation is done).
3. In this tutorial, we will briefly explore two ways of defining the extra padding (border) for an image:
  - (a) **BORDER\_CONSTANT**: Pad the image with a constant value (i.e. black or 0)
  - (b) **BORDER\_REPLICATE**: The row or column at the very edge of the original is replicated to the extra border.

This will be seen more clearly in the Code section.

### Code

#### 1. What does this program do?

- Load an image
- Let the user choose what kind of padding use in the input image. There are two options:
  - (a) *Constant value border*: Applies a padding of a constant value for the whole border. This value will be updated randomly each 0.5 seconds.
  - (b) *Replicated border*: The border will be replicated from the pixel values at the edges of the original image.



The user chooses either option by pressing 'c' (constant) or 'r' (replicate)

- The program finishes when the user presses 'ESC'

2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

// Global Variables
Mat src, dst;
int top, bottom, left, right;
int borderType;
Scalar value;
char* window_name = "copyMakeBorder Demo";
RNG rng(12345);

/** @function main */
int main( int argc, char** argv )
{
    int c;

    // Load an image
    src = imread( argv[1] );

    if( !src.data )
    { return -1;
      printf(" No data entered, please enter the path to an image file \n");
    }

    // Brief how-to for this program
    printf( "\n \t copyMakeBorder Demo: \n" );
    printf( "\t ----- \n" );
    printf( " ** Press 'c' to set the border to a random constant value \n");
    printf( " ** Press 'r' to set the border to be replicated \n");
    printf( " ** Press 'ESC' to exit the program \n");

    // Create window
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );

    // Initialize arguments for the filter
    top = (int) (0.05*src.rows); bottom = (int) (0.05*src.rows);
    left = (int) (0.05*src.cols); right = (int) (0.05*src.cols);
    dst = src;

    imshow( window_name, dst );

    while( true )
    {
        c = waitKey(500);

        if( (char)c == 27 )
            { break; }
        else if( (char)c == 'c' )
            { borderType = BORDER_CONSTANT; }
```

```

else if( (char)c == 'r' )
    { borderType = BORDER_REPLICATE; }

value = Scalar( rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255) );
copyMakeBorder( src, dst, top, bottom, left, right, borderType, value );

imshow( window_name, dst );
}

return 0;
}

```

## Explanation

1. First we declare the variables we are going to use:

```

Mat src, dst;
int top, bottom, left, right;
int borderType;
Scalar value;
char* window_name = "copyMakeBorder Demo";
RNG rng(12345);

```

Especial attention deserves the variable *rng* which is a random number generator. We use it to generate the random border color, as we will see soon.

2. As usual we load our source image *src*:

```

src = imread( argv[1] );

if( !src.data )
{ return -1;
  printf(" No data entered, please enter the path to an image file \n");
}

```

3. After giving a short intro of how to use the program, we create a window:

```

namedWindow( window_name, CV_WINDOW_AUTOSIZE );

```

4. Now we initialize the argument that defines the size of the borders (*top*, *bottom*, *left* and *right*). We give them a value of 5% the size of *src*.

```

top = (int) (0.05*src.rows); bottom = (int) (0.05*src.rows);
left = (int) (0.05*src.cols); right = (int) (0.05*src.cols);

```

5. The program begins a *while* loop. If the user presses 'c' or 'r', the *borderType* variable takes the value of *BORDER\_CONSTANT* or *BORDER\_REPLICATE* respectively:

```

while( true )
{
    c = waitKey(500);

    if( (char)c == 27 )
        { break; }
    else if( (char)c == 'c' )
        { borderType = BORDER_CONSTANT; }
    else if( (char)c == 'r' )
        { borderType = BORDER_REPLICATE; }
}

```

6. In each iteration (after 0.5 seconds), the variable *value* is updated...

```
value = Scalar( rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255) );
```

with a random value generated by the **RNG** variable *rng*. This value is a number picked randomly in the range [0, 255]

7. Finally, we call the function `copyMakeBorder` to apply the respective padding:

```
copyMakeBorder( src, dst, top, bottom, left, right, borderType, value );
```

The arguments are:

- (a) *src*: Source image
  - (b) *dst*: Destination image
  - (c) *top, bottom, left, right*: Length in pixels of the borders at each side of the image. We define them as being 5% of the original size of the image.
  - (d) *borderType*: Define what type of border is applied. It can be constant or replicate for this example.
  - (e) *value*: If *borderType* is `BORDER_CONSTANT`, this is the value used to fill the border pixels.
8. We display our output image in the image created previously

```
imshow( window_name, dst );
```

## Results

1. After compiling the code above, you can execute it giving as argument the path of an image. The result should be:
  - By default, it begins with the border set to `BORDER_CONSTANT`. Hence, a succession of random colored borders will be shown.
  - If you press 'r', the border will become a replica of the edge pixels.
  - If you press 'c', the random colored borders will appear again
  - If you press 'ESC' the program will exit.

Below some screenshot showing how the border changes color and how the `BORDER_REPLICATE` option looks:



## 3.8 Sobel Derivatives

### Goal

In this tutorial you will learn how to:

1. Use the OpenCV function `Sobel` to calculate the derivatives from an image.
2. Use the OpenCV function `Scharr` to calculate a more accurate derivative for a kernel of size  $3 \cdot 3$

### Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

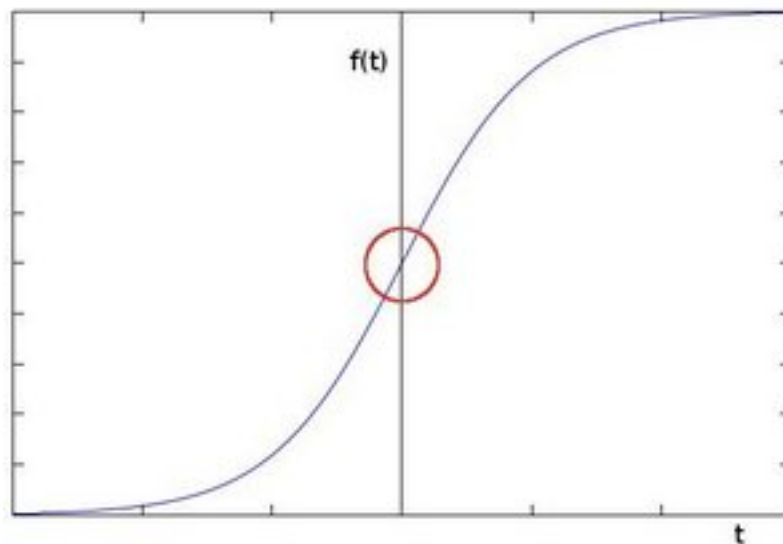
---

1. In the last two tutorials we have seen applicative examples of convolutions. One of the most important convolutions is the computation of derivatives in an image (or an approximation to them).
2. Why may be important the calculus of the derivatives in an image? Let's imagine we want to detect the *edges* present in the image. For instance:

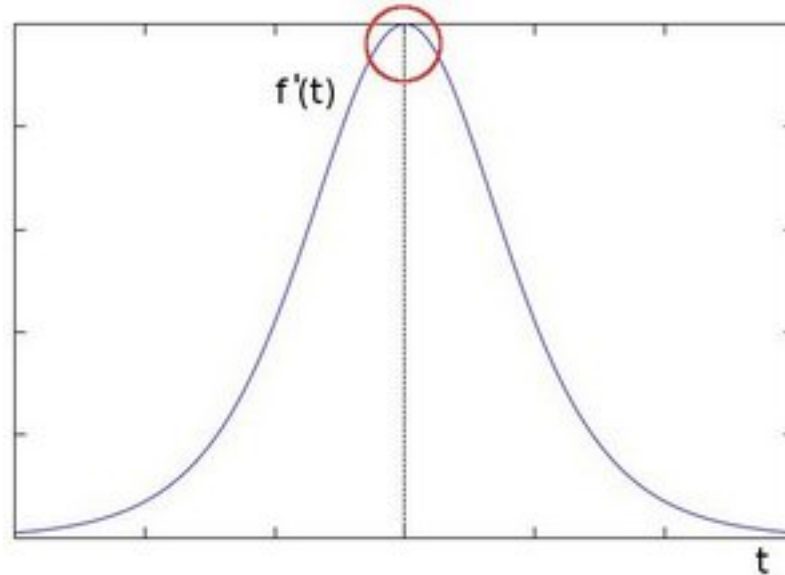


You can easily notice that in an *edge*, the pixel intensity *changes* in a notorious way. A good way to express *changes* is by using *derivatives*. A high change in gradient indicates a major change in the image.

3. To be more graphical, let's assume we have a 1D-image. An edge is shown by the "jump" in intensity in the plot below:



4. The edge "jump" can be seen more easily if we take the first derivative (actually, here appears as a maximum)



5. So, from the explanation above, we can deduce that a method to detect edges in an image can be performed by locating pixel locations where the gradient is higher than its neighbors (or to generalize, higher than a threshold).
6. More detailed explanation, please refer to **Learning OpenCV** by Bradski and Kaehler

### Sobel Operator

1. The Sobel Operator is a discrete differentiation operator. It computes an approximation of the gradient of an image intensity function.
2. The Sobel Operator combines Gaussian smoothing and differentiation.

### Formulation

Assuming that the image to be operated is  $I$ :

1. We calculate two derivatives:
  - (a) **Horizontal changes:** This is computed by convolving  $I$  with a kernel  $G_x$  with odd size. For example for a kernel size of 3,  $G_x$  would be computed as:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

- (b) **Vertical changes:** This is computed by convolving  $I$  with a kernel  $G_y$  with odd size. For example for a kernel size of 3,  $G_y$  would be computed as:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

2. At each point of the image we calculate an approximation of the *gradient* in that point by combining both results above:

$$G = \sqrt{G_x^2 + G_y^2}$$

Although sometimes the following simpler equation is used:

$$G = |G_x| + |G_y|$$

---

### Note:

When the size of the kernel is 3, the Sobel kernel shown above may produce noticeable inaccuracies (after all, Sobel is only an approximation of the derivative). OpenCV addresses this inaccuracy for kernels of size 3 by using the [Scharr](#) function. This is as fast but more accurate than the standar Sobel function. It implements the following kernels:

$$G_x = \begin{bmatrix} -3 & 0 & +3 \\ -10 & 0 & +10 \\ -3 & 0 & +3 \end{bmatrix}$$
$$G_y = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ +3 & +10 & +3 \end{bmatrix}$$

You can check out more information of this function in the OpenCV reference ([Scharr](#)). Also, in the sample code below, you will notice that above the code for [Sobel](#) function there is also code for the [Scharr](#) function commented. Uncommenting it (and obviously commenting the Sobel stuff) should give you an idea of how this function works.

---

## Code

### 1. What does this program do?

- Applies the *Sobel Operator* and generates as output an image with the detected *edges* bright on a darker background.

2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/** @function main */
int main( int argc, char** argv )
{

    Mat src, src_gray;
    Mat grad;
    char* window_name = "Sobel Demo - Simple Edge Detector";
    int scale = 1;
    int delta = 0;
    int ddepth = CV_16S;
```

```

int c;

/// Load an image
src = imread( argv[1] );

if( !src.data )
{ return -1; }

GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );

/// Convert it to gray
cvtColor( src, src_gray, CV_RGB2GRAY );

/// Create window
namedWindow( window_name, CV_WINDOW_AUTOSIZE );

/// Generate grad_x and grad_y
Mat grad_x, grad_y;
Mat abs_grad_x, abs_grad_y;

/// Gradient X
//Scharr( src_gray, grad_x, ddepth, 1, 0, scale, delta, BORDER_DEFAULT );
Sobel( src_gray, grad_x, ddepth, 1, 0, 3, scale, delta, BORDER_DEFAULT );
convertScaleAbs( grad_x, abs_grad_x );

/// Gradient Y
//Scharr( src_gray, grad_y, ddepth, 0, 1, scale, delta, BORDER_DEFAULT );
Sobel( src_gray, grad_y, ddepth, 0, 1, 3, scale, delta, BORDER_DEFAULT );
convertScaleAbs( grad_y, abs_grad_y );

/// Total Gradient (approximate)
addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );

imshow( window_name, grad );

waitKey(0);

return 0;
}

```

## Explanation

1. First we declare the variables we are going to use:

```

Mat src, src_gray;
Mat grad;
char* window_name = "Sobel Demo - Simple Edge Detector";
int scale = 1;
int delta = 0;
int ddepth = CV_16S;

```

2. As usual we load our source image *src*:

```

src = imread( argv[1] );

if( !src.data )
{ return -1; }

```



3. First, we apply a `GaussianBlur` to our image to reduce the noise ( kernel size = 3 )

```
GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );
```

4. Now we convert our filtered image to grayscale:

```
cvtColor( src, src_gray, CV_RGB2GRAY );
```

5. Second, we calculate the “*derivatives*” in *x* and *y* directions. For this, we use the function `Sobel` as shown below:

```
Mat grad_x, grad_y;
Mat abs_grad_x, abs_grad_y;

/// Gradient X
Sobel( src_gray, grad_x, ddepth, 1, 0, 3, scale, delta, BORDER_DEFAULT );
/// Gradient Y
Sobel( src_gray, grad_y, ddepth, 0, 1, 3, scale, delta, BORDER_DEFAULT );
```

The function takes the following arguments:

- *src\_gray*: In our example, the input image. Here it is *CV\_8U*
- *grad\_x/grad\_y*: The output image.
- *ddepth*: The depth of the output image. We set it to *CV\_16S* to avoid overflow.
- *x\_order*: The order of the derivative in *x* direction.
- *y\_order*: The order of the derivative in *y* direction.
- *scale, delta* and *BORDER\_DEFAULT*: We use default values.

Notice that to calculate the gradient in *x* direction we use:  $x_{order} = 1$  and  $y_{order} = 0$ . We do analogously for the *y* direction.

6. We convert our partial results back to *CV\_8U*:

```
convertScaleAbs( grad_x, abs_grad_x );
convertScaleAbs( grad_y, abs_grad_y );
```

7. Finally, we try to approximate the *gradient* by adding both directional gradients (note that this is not an exact calculation at all! but it is good for our purposes).

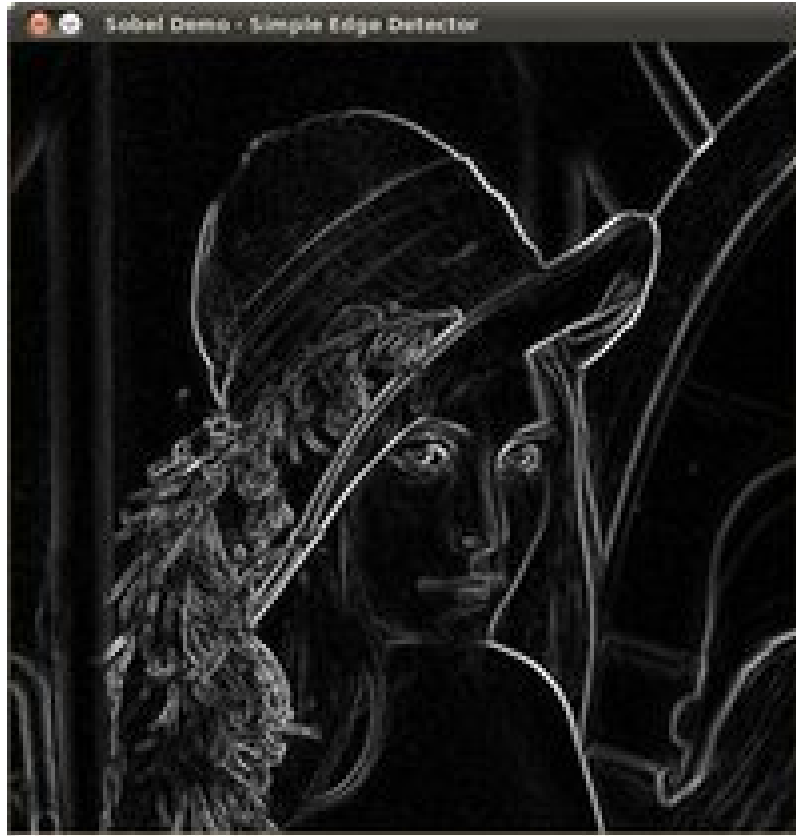
```
addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );
```

8. Finally, we show our result:

```
imshow( window_name, grad );
```

## Results

1. Here is the output of applying our basic detector to *lena.jpg*:



## 3.9 Laplace Operator

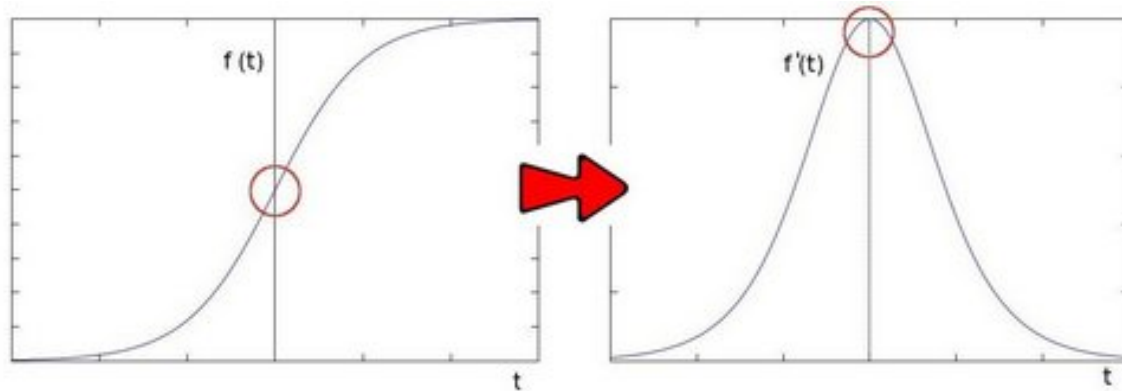
### Goal

In this tutorial you will learn how to:

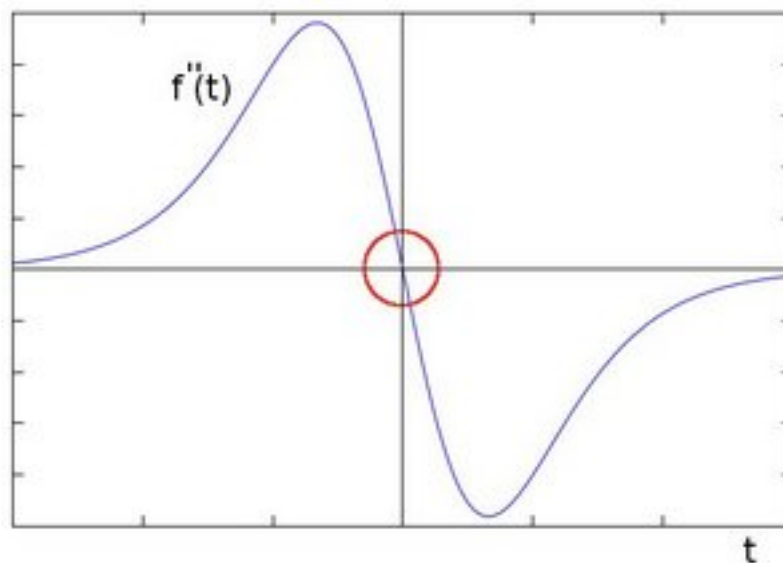
1. Use the OpenCV function `Laplacian` to implement a discrete analog of the *Laplacian operator*.

### Theory

1. In the previous tutorial we learned how to use the *Sobel Operator*. It was based on the fact that in the edge area, the pixel intensity shows a “jump” or a high variation of intensity. Getting the first derivative of the intensity, we observed that an edge is characterized by a maximum, as it can be seen in the figure:



2. And...what happens if we take the second derivative?



You can observe that the second derivative is zero! So, we can also use this criterion to attempt to detect edges in an image. However, note that zeros will not only appear in edges (they can actually appear in other meaningless locations); this can be solved by applying filtering where needed.

## Laplacian Operator

1. From the explanation above, we deduce that the second derivative can be used to *detect edges*. Since images are “2D”, we would need to take the derivative in both dimensions. Here, the Laplacian operator comes handy.
2. The *Laplacian operator* is defined by:

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

1. The Laplacian operator is implemented in OpenCV by the function `Laplacian`. In fact, since the Laplacian uses the gradient of images, it calls internally the *Sobel* operator to perform its computation.

## Code

### 1. What does this program do?

- Loads an image
- Remove noise by applying a Gaussian blur and then convert the original image to grayscale
- Applies a Laplacian operator to the grayscale image and stores the output image
- Display the result in a window

2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/** @function main */
int main( int argc, char** argv )
{
    Mat src, src_gray, dst;
    int kernel_size = 3;
    int scale = 1;
    int delta = 0;
    int ddepth = CV_16S;
    char* window_name = "Laplace Demo";

    int c;

    // Load an image
    src = imread( argv[1] );

    if( !src.data )
        { return -1; }

    // Remove noise by blurring with a Gaussian filter
    GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );

    // Convert the image to grayscale
    cvtColor( src, src_gray, CV_RGB2GRAY );

    // Create window
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );

    // Apply Laplace function
    Mat abs_dst;

    Laplacian( src_gray, dst, ddepth, kernel_size, scale, delta, BORDER_DEFAULT );
    convertScaleAbs( dst, abs_dst );

    // Show what you got
    imshow( window_name, abs_dst );

    waitKey(0);

    return 0;
}
```

```
}
```

### Explanation

1. Create some needed variables:

```
Mat src, src_gray, dst;
int kernel_size = 3;
int scale = 1;
int delta = 0;
int ddepth = CV_16S;
char* window_name = "Laplace Demo";
```

2. Loads the source image:

```
src = imread( argv[1] );

if( !src.data )
{ return -1; }
```

3. Apply a Gaussian blur to reduce noise:

```
GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );
```

4. Convert the image to grayscale using `cvtColor`

```
cvtColor( src, src_gray, CV_RGB2GRAY );
```

5. Apply the Laplacian operator to the grayscale image:

```
Laplacian( src_gray, dst, ddepth, kernel_size, scale, delta, BORDER_DEFAULT );
```

where the arguments are:

- *src\_gray*: The input image.
- *dst*: Destination (output) image
- *ddepth*: Depth of the destination image. Since our input is *CV\_8U* we define *ddepth = CV\_16S* to avoid overflow
- *kernel\_size*: The kernel size of the Sobel operator to be applied internally. We use 3 in this example.
- *scale*, *delta* and *BORDER\_DEFAULT*: We leave them as default values.

6. Convert the output from the Laplacian operator to a *CV\_8U* image:

```
convertScaleAbs( dst, abs_dst );
```

7. Display the result in a window:

```
imshow( window_name, abs_dst );
```

### Results

1. After compiling the code above, we can run it giving as argument the path to an image. For example, using as an input:



2. We obtain the following result. Notice how the trees and the silhouette of the cow are approximately well defined (except in areas in which the intensity are very similar, i.e. around the cow's head). Also, note that the roof of the house behind the trees (right side) is notoriously marked. This is due to the fact that the contrast is higher in that region.



## 3.10 Canny Edge Detector

### Goal

In this tutorial you will learn how to:

1. Use the OpenCV function `Canny` to implement the Canny Edge Detector.

### Theory

1. The *Canny Edge detector* was developed by John F. Canny in 1986. Also known to many as the *optimal detector*, Canny algorithm aims to satisfy three main criteria:

- **Low error rate:** Meaning a good detection of only existent edges.
- **Good localization:** The distance between edge pixels detected and real edge pixels have to be minimized.
- **Minimal response:** Only one detector response per edge.

## Steps

1. Filter out any noise. The Gaussian filter is used for this purpose. An example of a Gaussian kernel of size = 5 that might be used is shown below:

$$K = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

2. Find the intensity gradient of the image. For this, we follow a procedure analogous to Sobel:
  - (a) Apply a pair of convolution masks (in x and y directions):

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

- (b) Find the gradient strength and direction with:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

The direction is rounded to one of four possible angles (namely 0, 45, 90 or 135)

3. *Non-maximum* suppression is applied. This removes pixels that are not considered to be part of an edge. Hence, only thin lines (candidate edges) will remain.
4. *Hysteresis*: The final step. Canny does use two thresholds (upper and lower):
  - (a) If a pixel gradient is higher than the *upper* threshold, the pixel is accepted as an edge
  - (b) If a pixel gradient value is below the *lower* threshold, then it is rejected.
  - (c) If the pixel gradient is between the two thresholds, then it will be accepted only if it is connected to a pixel that is above the *upper* threshold.

Canny recommended a *upper:lower* ratio between 2:1 and 3:1.
5. For more details, you can always consult your favorite Computer Vision book.

## Code

1. **What does this program do?**

- Asks the user to enter a numerical value to set the lower threshold for our *Canny Edge Detector* (by means of a *Trackbar*)
- Applies the *Canny Detector* and generates a **mask** (bright lines representing the edges on a black background).
- Applies the mask obtained on the original image and display it in a window.

2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

/// Global variables

Mat src, src_gray;
Mat dst, detected_edges;

int edgeThresh = 1;
int lowThreshold;
int const max_lowThreshold = 100;
int ratio = 3;
int kernel_size = 3;
char* window_name = "Edge Map";

/**
 * @function CannyThreshold
 * @brief Trackbar callback - Canny thresholds input with a ratio 1:3
 */
void CannyThreshold(int, void*)
{
    /// Reduce noise with a kernel 3x3
    blur( src_gray, detected_edges, Size(3,3) );

    /// Canny detector
    Canny( detected_edges, detected_edges, lowThreshold, lowThreshold*ratio, kernel_size );

    /// Using Canny's output as a mask, we display our result
    dst = Scalar::all(0);

    src.copyTo( dst, detected_edges);
    imshow( window_name, dst );
}

/** @function main */
int main( int argc, char** argv )
{
    /// Load an image
    src = imread( argv[1] );

    if( !src.data )
    { return -1; }

    /// Create a matrix of the same type and size as src (for dst)
    dst.create( src.size(), src.type() );
}
```



```
/// Convert the image to grayscale
cvtColor( src, src_gray, CV_BGR2GRAY );

/// Create a window
namedWindow( window_name, CV_WINDOW_AUTOSIZE );

/// Create a Trackbar for user to enter threshold
createTrackbar( "Min Threshold:", window_name, &lowThreshold, max_lowThreshold, CannyThreshold );

/// Show the image
CannyThreshold(0, 0);

/// Wait until user exit program by pressing a key
waitKey(0);

return 0;
}
```

## Explanation

1. Create some needed variables:

```
Mat src, src_gray;
Mat dst, detected_edges;

int edgeThresh = 1;
int lowThreshold;
int const max_lowThreshold = 100;
int ratio = 3;
int kernel_size = 3;
char* window_name = "Edge Map";
```

Note the following:

- a. We establish a ratio of lower:upper threshold of 3:1 (with the variable *\*ratio\**)
- b. We set the kernel size of *:math:'3'* (for the Sobel operations to be performed internally by the Canny function)
- c. We set a maximum value for the lower Threshold of *:math:'100'*.

2. Loads the source image:

```
/// Load an image
src = imread( argv[1] );

if( !src.data )
{ return -1; }
```

3. Create a matrix of the same type and size of *src* (to be *dst*)

```
dst.create( src.size(), src.type() );
```

4. Convert the image to grayscale (using the function `cvtColor`:

```
cvtColor( src, src_gray, CV_BGR2GRAY );
```

5. Create a window to display the results

```
namedWindow( window_name, CV_WINDOW_AUTOSIZE );
```

6. Create a `Trackbar` for the user to enter the lower threshold for our Canny detector:

```
createTrackbar( "Min Threshold:", window_name, &lowThreshold, max_lowThreshold, CannyThreshold );
```

Observe the following:

- (a) The variable to be controlled by the `Trackbar` is `lowThreshold` with a limit of `max_lowThreshold` (which we set to 100 previously)
  - (b) Each time the `Trackbar` registers an action, the callback function `CannyThreshold` will be invoked.
7. Let's check the `CannyThreshold` function, step by step:

- (a) First, we blur the image with a filter of kernel size 3:

```
blur( src_gray, detected_edges, Size(3,3) );
```

- (b) Second, we apply the OpenCV function `Canny`:

```
Canny( detected_edges, detected_edges, lowThreshold, lowThreshold*ratio, kernel_size );
```

where the arguments are:

- `detected_edges`: Source image, grayscale
- `detected_edges`: Output of the detector (can be the same as the input)
- `lowThreshold`: The value entered by the user moving the `Trackbar`
- `highThreshold`: Set in the program as three times the lower threshold (following Canny's recommendation)
- `kernel_size`: We defined it to be 3 (the size of the Sobel kernel to be used internally)

8. We fill a `dst` image with zeros (meaning the image is completely black).

```
dst = Scalar::all(0);
```

9. Finally, we will use the function `copyTo` to map only the areas of the image that are identified as edges (on a black background).

```
src.copyTo( dst, detected_edges);
```

`copyTo` copy the `src` image onto `dst`. However, it will only copy the pixels in the locations where they have non-zero values. Since the output of the Canny detector is the edge contours on a black background, the resulting `dst` will be black in all the area but the detected edges.

10. We display our result:

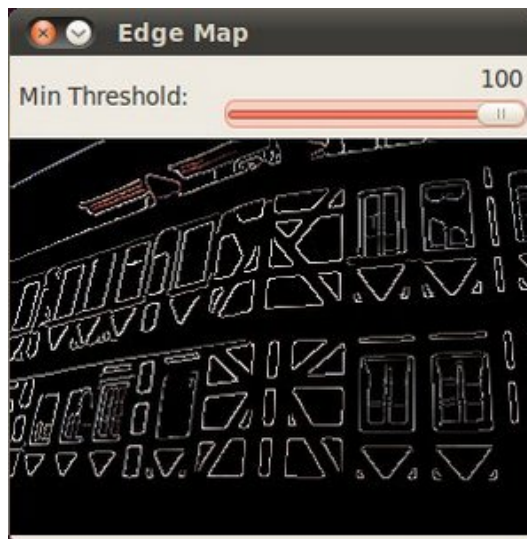
```
imshow( window_name, dst );
```

## Result

1. After compiling the code above, we can run it giving as argument the path to an image. For example, using as an input the following image:



and moving the slider, trying different threshold, we obtain the following result:



Notice how the image is superposed to the black background on the edge regions.

## 3.11 Hough Line Transform

### Goal

In this tutorial you will learn how to:

- Use the OpenCV functions [HoughLines](#) and [HoughLinesP](#) to detect lines in an image.

### Theory

---

**Note:** The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

---

#### Hough Line Transform

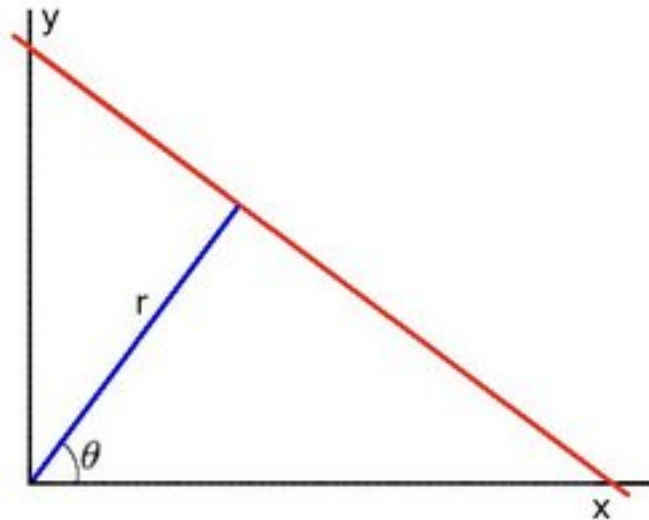
1. The Hough Line Transform is a transform used to detect straight lines.
2. To apply the Transform, first an edge detection pre-processing is desirable.

## How does it work?

1. As you know, a line in the image space can be expressed with two variables. For example:

(a) In the **Cartesian coordinate system**: Parameters:  $(m, b)$ .

(b) In the **Polar coordinate system**: Parameters:  $(r, \theta)$



For Hough Transforms, we will express lines in the *Polar system*. Hence, a line equation can be written as:

$$y = \left( -\frac{\cos \theta}{\sin \theta} \right) x + \left( \frac{r}{\sin \theta} \right)$$

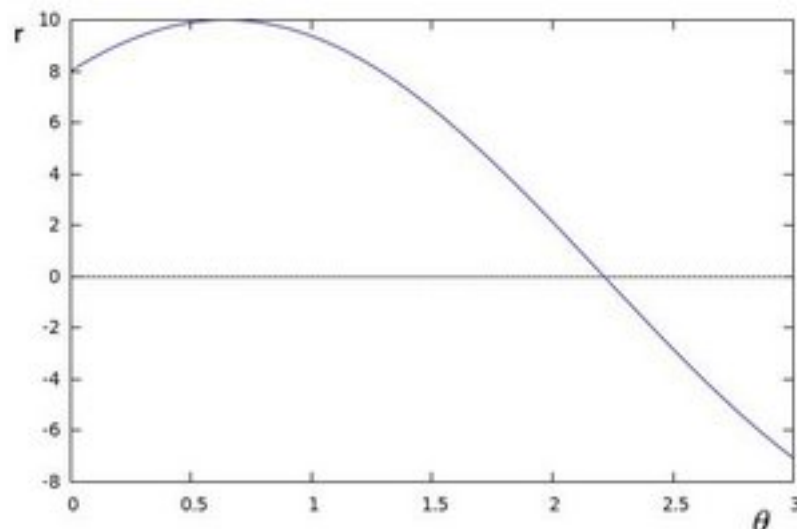
Arranging the terms:  $r = x \cos \theta + y \sin \theta$

1. In general for each point  $(x_0, y_0)$ , we can define the family of lines that goes through that point as:

$$r_\theta = x_0 \cdot \cos \theta + y_0 \cdot \sin \theta$$

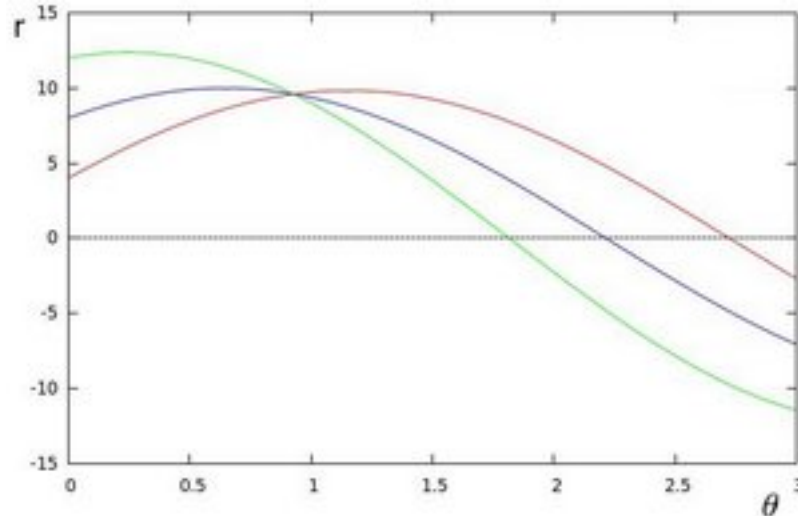
Meaning that each pair  $(r_\theta, \theta)$  represents each line that passes by  $(x_0, y_0)$ .

2. If for a given  $(x_0, y_0)$  we plot the family of lines that goes through it, we get a sinusoid. For instance, for  $x_0 = 8$  and  $y_0 = 6$  we get the following plot (in a plane  $\theta - r$ ):



We consider only points such that  $r > 0$  and  $0 < \theta < 2\pi$ .

3. We can do the same operation above for all the points in an image. If the curves of two different points intersect in the plane  $\theta - r$ , that means that both points belong to a same line. For instance, following with the example above and drawing the plot for two more points:  $x_1 = 9, y_1 = 4$  and  $x_2 = 12, y_2 = 3$ , we get:



The three plots intersect in one single point  $(0.925, 9.6)$ , these coordinates are the parameters  $(\theta, r)$  or the line in which  $(x_0, y_0)$ ,  $(x_1, y_1)$  and  $(x_2, y_2)$  lay.

4. What does all the stuff above mean? It means that in general, a line can be *detected* by finding the number of intersections between curves. The more curves intersecting means that the line represented by that intersection have more points. In general, we can define a *threshold* of the minimum number of intersections needed to *detect* a line.
5. This is what the Hough Line Transform does. It keeps track of the intersection between curves of every point in the image. If the number of intersections is above some *threshold*, then it declares it as a line with the parameters  $(\theta, r_\theta)$  of the intersection point.

### Standard and Probabilistic Hough Line Transform

OpenCV implements two kind of Hough Line Transforms:

1. **The Standard Hough Transform**

- It consists in pretty much what we just explained in the previous section. It gives you as result a vector of couples  $(\theta, r_\theta)$
- In OpenCV it is implemented with the function [HoughLines](#)

2. **The Probabilistic Hough Line Transform**

- A more efficient implementation of the Hough Line Transform. It gives as output the extremes of the detected lines  $(x_0, y_0, x_1, y_1)$
- In OpenCV it is implemented with the function [HoughLinesP](#)

### Code

1. **What does this program do?**

- Loads an image

- Applies either a *Standard Hough Line Transform* or a *Probabilistic Line Transform*.
- Display the original image and the detected line in two windows.

2. The sample code that we will explain can be downloaded from [here](#). A slightly fancier version (which shows both Hough standard and probabilistic with trackbars for changing the threshold values) can be found [here](#).

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>

using namespace cv;
using namespace std;

void help()
{
    cout << "\nThis program demonstrates line finding with the Hough transform.\n"
         << "Usage:\n"
         << ".\nhoughlines <image_name>, Default is pic1.jpg\n" << endl;
}

int main(int argc, char** argv)
{
    const char* filename = argc >= 2 ? argv[1] : "pic1.jpg";

    Mat src = imread(filename, 0);
    if(src.empty())
    {
        help();
        cout << "can not open " << filename << endl;
        return -1;
    }

    Mat dst, cdst;
    Canny(src, dst, 50, 200, 3);
    cvtColor(dst, cdst, CV_GRAY2BGR);

    #if 0
        vector<Vec2f> lines;
        HoughLines(dst, lines, 1, CV_PI/180, 100, 0, 0 );

        for( size_t i = 0; i < lines.size(); i++ )
        {
            float rho = lines[i][0], theta = lines[i][1];
            Point pt1, pt2;
            double a = cos(theta), b = sin(theta);
            double x0 = a*rho, y0 = b*rho;
            pt1.x = cvRound(x0 + 1000*(-b));
            pt1.y = cvRound(y0 + 1000*(a));
            pt2.x = cvRound(x0 - 1000*(-b));
            pt2.y = cvRound(y0 - 1000*(a));
            line( cdst, pt1, pt2, Scalar(0,0,255), 3, CV_AA);
        }
    #else
        vector<Vec4i> lines;
        HoughLinesP(dst, lines, 1, CV_PI/180, 50, 50, 10 );
        for( size_t i = 0; i < lines.size(); i++ )
        {
```

```
    Vec4i l = lines[i];
    line( dst, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(0,0,255), 3, CV_AA);
}
#endif
imshow("source", src);
imshow("detected lines", dst);

waitKey();

return 0;
}
```

## Explanation

### 1. Load an image

```
Mat src = imread(filename, 0);
if(src.empty())
{
    help();
    cout << "can not open " << filename << endl;
    return -1;
}
```

### 2. Detect the edges of the image by using a Canny detector

```
Canny(src, dst, 50, 200, 3);
```

Now we will apply the Hough Line Transform. We will explain how to use both OpenCV functions available for this purpose:

### 3. Standard Hough Line Transform

(a) First, you apply the Transform:

```
vector<Vec2f> lines;
HoughLines(dst, lines, 1, CV_PI/180, 100, 0, 0 );
```

with the following arguments:

- *dst*: Output of the edge detector. It should be a grayscale image (although in fact it is a binary one)
- *lines*: A vector that will store the parameters  $(r, \theta)$  of the detected lines
- *rho* : The resolution of the parameter  $r$  in pixels. We use **1** pixel.
- *theta*: The resolution of the parameter  $\theta$  in radians. We use **1 degree** (CV\_PI/180)
- *threshold*: The minimum number of intersections to “detect” a line
- *srn* and *stm*: Default parameters to zero. Check OpenCV reference for more info.

(b) And then you display the result by drawing the lines.

```
for( size_t i = 0; i < lines.size(); i++ )
{
    float rho = lines[i][0], theta = lines[i][1];
    Point pt1, pt2;
    double a = cos(theta), b = sin(theta);
    double x0 = a*rho, y0 = b*rho;
    pt1.x = cvRound(x0 + 1000*(-b));
```

```

    pt1.y = cvRound(y0 + 1000*(a));
    pt2.x = cvRound(x0 - 1000*(-b));
    pt2.y = cvRound(y0 - 1000*(a));
    line( cdst, pt1, pt2, Scalar(0,0,255), 3, CV_AA);
}

```

#### 4. Probabilistic Hough Line Transform

(a) First you apply the transform:

```

vector<Vec4i> lines;
HoughLinesP(dst, lines, 1, CV_PI/180, 50, 50, 10 );

```

with the arguments:

- *dst*: Output of the edge detector. It should be a grayscale image (although in fact it is a binary one)
- *lines*: A vector that will store the parameters  $(x_{start}, y_{start}, x_{end}, y_{end})$  of the detected lines
- *rho*: The resolution of the parameter  $r$  in pixels. We use **1** pixel.
- *theta*: The resolution of the parameter  $\theta$  in radians. We use **1 degree** ( $CV\_PI/180$ )
- *threshold*: The minimum number of intersections to “detect” a line
- *minLinLength*: The minimum number of points that can form a line. Lines with less than this number of points are disregarded.
- *maxLineGap*: The maximum gap between two points to be considered in the same line.

(b) And then you display the result by drawing the lines.

```

for( size_t i = 0; i < lines.size(); i++ )
{
    Vec4i l = lines[i];
    line( cdst, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(0,0,255), 3, CV_AA);
}

```

5. Display the original image and the detected lines:

```

imshow("source", src);
imshow("detected lines", cdst);

```

6. Wait until the user exits the program

```

waitKey();

```

## Result

---

**Note:** The results below are obtained using the slightly fancier version we mentioned in the *Code* section. It still implements the same stuff as above, only adding the *Trackbar* for the *Threshold*.

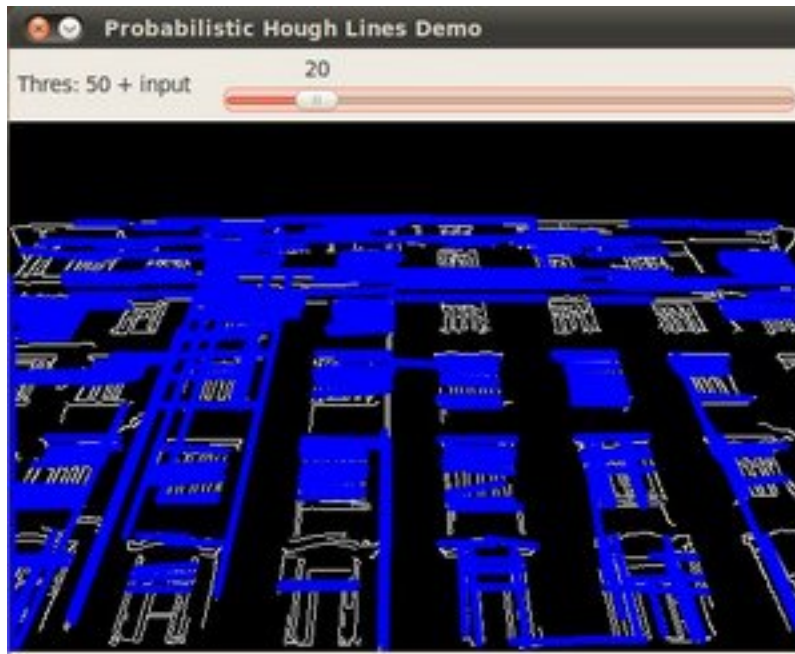
---

Using an input image such as:





We get the following result by using the Probabilistic Hough Line Transform:



You may observe that the number of lines detected vary while you change the *threshold*. The explanation is sort of evident: If you establish a higher threshold, fewer lines will be detected (since you will need more points to declare a line detected).

## 3.12 Hough Circle Transform

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `HoughCircles` to detect circles in an image.

## Theory

### Hough Circle Transform

- The Hough Circle Transform works in a *roughly* analogous way to the Hough Line Transform explained in the previous tutorial.
- In the line detection case, a line was defined by two parameters  $(r, \theta)$ . In the circle case, we need three parameters to define a circle:

$$C : (x_{\text{center}}, y_{\text{center}}, r)$$

where  $(x_{\text{center}}, y_{\text{center}})$  define the center position (green point) and  $r$  is the radius, which allows us to completely define a circle, as it can be seen below:



- For sake of efficiency, OpenCV implements a detection method slightly trickier than the standard Hough Transform: *The Hough gradient method*. For more details, please check the book *Learning OpenCV* or your favorite Computer Vision bibliography

## Code

### 1. What does this program do?

- Loads an image and blur it to reduce the noise
- Applies the *Hough Circle Transform* to the blurred image .
- Display the detected circle in a window.

2. The sample code that we will explain can be downloaded from [here](#). A slightly fancier version (which shows both Hough standard and probabilistic with trackbars for changing the threshold values) can be found [here](#).

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
```

```
using namespace cv;
```

```
/** @function main */
```

```
int main(int argc, char** argv)
{
    Mat src, src_gray;

    /// Read the image
    src = imread( argv[1], 1 );

    if( !src.data )
        { return -1; }

    /// Convert it to gray
    cvtColor( src, src_gray, CV_BGR2GRAY );

    /// Reduce the noise so we avoid false circle detection
    GaussianBlur( src_gray, src_gray, Size(9, 9), 2, 2 );

    vector<Vec3f> circles;

    /// Apply the Hough Transform to find the circles
    HoughCircles( src_gray, circles, CV_HOUGH_GRADIENT, 1, src_gray.rows/8, 200, 100, 0, 0 );

    /// Draw the circles detected
    for( size_t i = 0; i < circles.size(); i++ )
    {
        Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
        int radius = cvRound(circles[i][2]);
        // circle center
        circle( src, center, 3, Scalar(0,255,0), -1, 8, 0 );
        // circle outline
        circle( src, center, radius, Scalar(0,0,255), 3, 8, 0 );
    }

    /// Show your results
    namedWindow( "Hough Circle Transform Demo", CV_WINDOW_AUTOSIZE );
    imshow( "Hough Circle Transform Demo", src );

    waitKey(0);
    return 0;
}
```

## Explanation

1. Load an image

```
src = imread( argv[1], 1 );
```

```
if( !src.data )
    { return -1; }
```

2. Convert it to grayscale:

```
cvtColor( src, src_gray, CV_BGR2GRAY );
```

3. Apply a Gaussian blur to reduce noise and avoid false circle detection:

```
GaussianBlur( src_gray, src_gray, Size(9, 9), 2, 2 );
```

4. Proceed to apply Hough Circle Transform:

```
vector<Vec3f> circles;

HoughCircles( src_gray, circles, CV_HOUGH_GRADIENT, 1, src_gray.rows/8, 200, 100, 0, 0 );
```

with the arguments:

- *src\_gray*: Input image (grayscale)
- *circles*: A vector that stores sets of 3 values:  $x_c, y_c, r$  for each detected circle.
- *CV\_HOUGH\_GRADIENT*: Define the detection method. Currently this is the only one available in OpenCV
- *dp = 1*: The inverse ratio of resolution
- *min\_dist = src\_gray.rows/8*: Minimum distance between detected centers
- *param\_1 = 200*: Upper threshold for the internal Canny edge detector
- *param\_2 = 100\**: Threshold for center detection.
- *min\_radius = 0*: Minimum radius to be detected. If unknown, put zero as default.
- *max\_radius = 0*: Maximum radius to be detected. If unknown, put zero as default

5. Draw the detected circles:

```
for( size_t i = 0; i < circles.size(); i++ )
{
    Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
    int radius = cvRound(circles[i][2]);
    // circle center
    circle( src, center, 3, Scalar(0,255,0), -1, 8, 0 );
    // circle outline
    circle( src, center, radius, Scalar(0,0,255), 3, 8, 0 );
}
```

You can see that we will draw the circle(s) on red and the center(s) with a small green dot

6. Display the detected circle(s):

```
namedWindow( "Hough Circle Transform Demo", CV_WINDOW_AUTOSIZE );
imshow( "Hough Circle Transform Demo", src );
```

7. Wait for the user to exit the program

```
waitKey(0);
```

## Result

The result of running the code above with a test image is shown below:



## 3.13 Remapping

### Goal

In this tutorial you will learn how to:

1. Use the OpenCV function `remap` to implement simple remapping routines.

### Theory

#### What is remapping?

- It is the process of taking pixels from one place in the image and locating them in another position in a new image.
- To accomplish the mapping process, it might be necessary to do some interpolation for non-integer pixel locations, since there will not always be a one-to-one-pixel correspondence between source and destination images.
- We can express the remap for every pixel location  $(x, y)$  as:

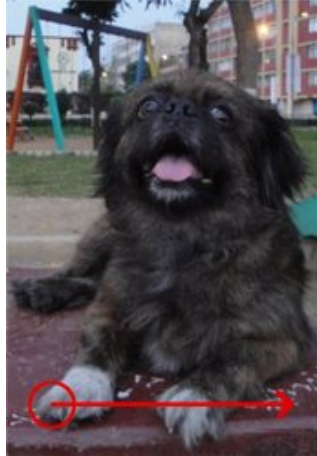
$$g(x, y) = f(h(x, y))$$

where  $g()$  is the remapped image,  $f()$  the source image and  $h(x, y)$  is the mapping function that operates on  $(x, y)$ .

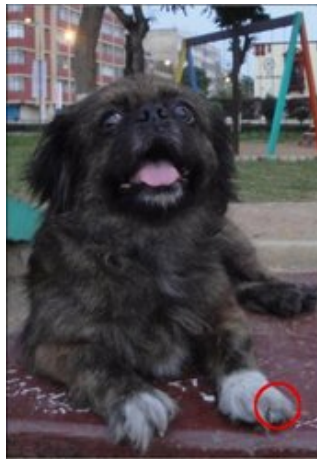
- Let's think in a quick example. Imagine that we have an image  $I$  and, say, we want to do a remap such that:

$$h(x, y) = (I.cols - x, y)$$

What would happen? It is easily seen that the image would flip in the  $x$  direction. For instance, consider the input image:



observe how the red circle changes positions with respect to  $x$  (considering  $x$  the horizontal direction):



- In OpenCV, the function `remap` offers a simple remapping implementation.

## Code

### 1. What does this program do?

- Loads an image
- Each second, apply 1 of 4 different remapping processes to the image and display them indefinitely in a window.
- Wait for the user to exit the program

2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;

/// Global variables
Mat src, dst;
```

```
Mat map_x, map_y;
char* remap_window = "Remap demo";
int ind = 0;

// Function Headers
void update_map( void );

/**
 * @function main
 */
int main( int argc, char** argv )
{
    // Load the image
    src = imread( argv[1], 1 );

    // Create dst, map_x and map_y with the same size as src:
    dst.create( src.size(), src.type() );
    map_x.create( src.size(), CV_32FC1 );
    map_y.create( src.size(), CV_32FC1 );

    // Create window
    namedWindow( remap_window, CV_WINDOW_AUTOSIZE );

    // Loop
    while( true )
    {
        // Each 1 sec. Press ESC to exit the program
        int c = waitKey( 1000 );

        if( (char)c == 27 )
            { break; }

        // Update map_x & map_y. Then apply remap
        update_map();
        remap( src, dst, map_x, map_y, CV_INTER_LINEAR, BORDER_CONSTANT, Scalar(0,0, 0) );

        // Display results
        imshow( remap_window, dst );
    }
    return 0;
}

/**
 * @function update_map
 * @brief Fill the map_x and map_y matrices with 4 types of mappings
 */
void update_map( void )
{
    ind = ind%4;

    for( int j = 0; j < src.rows; j++ )
    { for( int i = 0; i < src.cols; i++ )
        {
            switch( ind )
            {
                case 0:
                    if( i > src.cols*0.25 && i < src.cols*0.75 && j > src.rows*0.25 && j < src.rows*0.75 )
                    {
```

```

        map_x.at<float>(j,i) = 2*( i - src.cols*0.25 ) + 0.5 ;
        map_y.at<float>(j,i) = 2*( j - src.rows*0.25 ) + 0.5 ;
    }
    else
    { map_x.at<float>(j,i) = 0 ;
      map_y.at<float>(j,i) = 0 ;
    }
    break;
case 1:
    map_x.at<float>(j,i) = i ;
    map_y.at<float>(j,i) = src.rows - j ;
    break;
case 2:
    map_x.at<float>(j,i) = src.cols - i ;
    map_y.at<float>(j,i) = j ;
    break;
case 3:
    map_x.at<float>(j,i) = src.cols - i ;
    map_y.at<float>(j,i) = src.rows - j ;
    break;
} // end of switch
}
}
ind++;
}

```

## Explanation

1. Create some variables we will use:

```

Mat src, dst;
Mat map_x, map_y;
char* remap_window = "Remap demo";
int ind = 0;

```

2. Load an image:

```
src = imread( argv[1], 1 );
```

3. Create the destination image and the two mapping matrices (for x and y)

```

dst.create( src.size(), src.type() );
map_x.create( src.size(), CV_32FC1 );
map_y.create( src.size(), CV_32FC1 );

```

4. Create a window to display results

```
namedWindow( remap_window, CV_WINDOW_AUTOSIZE );
```

5. Establish a loop. Each 1000 ms we update our mapping matrices (*mat\_x* and *mat\_y*) and apply them to our source image:

```

while( true )
{
    // Each 1 sec. Press ESC to exit the program
    int c = waitKey( 1000 );

    if( (char)c == 27 )

```



```

    { break; }

    /// Update map_x & map_y. Then apply remap
    update_map();
    remap( src, dst, map_x, map_y, CV_INTER_LINEAR, BORDER_CONSTANT, Scalar(0,0, 0) );

    /// Display results
    imshow( remap_window, dst );
}

```

The function that applies the remapping is `remap`. We give the following arguments:

- **src**: Source image
- **dst**: Destination image of same size as *src*
- **map\_x**: The mapping function in the x direction. It is equivalent to the first component of  $h(i, j)$
- **map\_y**: Same as above, but in y direction. Note that *map\_y* and *map\_x* are both of the same size as *src*
- **CV\_INTER\_LINEAR**: The type of interpolation to use for non-integer pixels. This is by default.
- **BORDER\_CONSTANT**: Default

How do we update our mapping matrices *mat\_x* and *mat\_y*? Go on reading:

#### 6. Updating the mapping matrices: We are going to perform 4 different mappings:

- (a) Reduce the picture to half its size and will display it in the middle:

$$h(i, j) = (2 * i - \text{src.cols}/2 + 0.5, 2 * j - \text{src.rows}/2 + 0.5)$$

for all pairs (i, j) such that:  $\frac{\text{src.cols}}{4} < i < \frac{3 \cdot \text{src.cols}}{4}$  and  $\frac{\text{src.rows}}{4} < j < \frac{3 \cdot \text{src.rows}}{4}$

- (b) Turn the image upside down:  $h(i, j) = (i, \text{src.rows} - j)$   
(c) Reflect the image from left to right:  $h(i, j) = (\text{src.cols} - i, j)$   
(d) Combination of b and c:  $h(i, j) = (\text{src.cols} - i, \text{src.rows} - j)$

This is expressed in the following snippet. Here, *map\_x* represents the first coordinate of  $h(i, j)$  and *map\_y* the second coordinate.

```

for( int j = 0; j < src.rows; j++ )
{ for( int i = 0; i < src.cols; i++ )
  {
    switch( ind )
    {
      case 0:
        if( i > src.cols*0.25 && i < src.cols*0.75 && j > src.rows*0.25 && j < src.rows*0.75 )
        {
          map_x.at<float>(j,i) = 2*( i - src.cols*0.25 ) + 0.5 ;
          map_y.at<float>(j,i) = 2*( j - src.rows*0.25 ) + 0.5 ;
        }
        else
        { map_x.at<float>(j,i) = 0 ;
          map_y.at<float>(j,i) = 0 ;
        }
        break;
      case 1:
        map_x.at<float>(j,i) = i ;
        map_y.at<float>(j,i) = src.rows - j ;
    }
  }
}

```

```
        break;
    case 2:
        map_x.at<float>(j,i) = src.cols - i ;
        map_y.at<float>(j,i) = j ;
        break;
    case 3:
        map_x.at<float>(j,i) = src.cols - i ;
        map_y.at<float>(j,i) = src.rows - j ;
        break;
} // end of switch
}
}
ind++;
}
```

## Result

1. After compiling the code above, you can execute it giving as argument an image path. For instance, by using the following image:



2. This is the result of reducing it to half the size and centering it:



3. Turning it upside down:



4. Reflecting it in the x direction:



5. Reflecting it in both directions:



## 3.14 Affine Transformations

### Goal

In this tutorial you will learn how to:

1. Use the OpenCV function `warpAffine` to implement simple remapping routines.

2. Use the OpenCV function `getRotationMatrix2D` to obtain a  $2 \times 3$  rotation matrix

## Theory

### What is an Affine Transformation?

1. It is any transformation that can be expressed in the form of a *matrix multiplication* (linear transformation) followed by a *vector addition* (translation).
2. From the above, We can use an Affine Transformation to express:
  - (a) Rotations (linear transformation)
  - (b) Translations (vector addition)
  - (c) Scale operations (linear transformation)
 you can see that, in essence, an Affine Transformation represents a **relation** between two images.
3. The usual way to represent an Affine Transform is by using a  $2 \times 3$  matrix.

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2 \times 2} \quad B = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}_{2 \times 1}$$

$$M = [A \quad B] = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}_{2 \times 3}$$

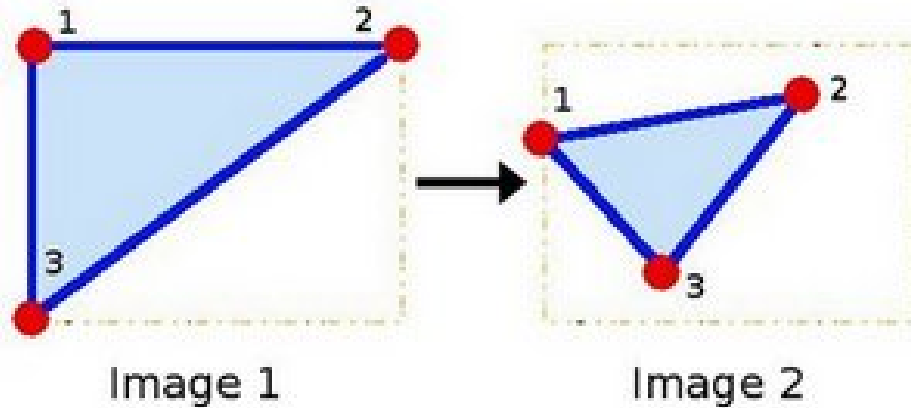
Considering that we want to transform a 2D vector  $X = \begin{bmatrix} x \\ y \end{bmatrix}$  by using  $A$  and  $B$ , we can do it equivalently with:

$$T = A \cdot \begin{bmatrix} x \\ y \end{bmatrix} + B \text{ or } T = M \cdot [x, y, 1]^T$$

$$T = \begin{bmatrix} a_{00}x + a_{01}y + b_{00} \\ a_{10}x + a_{11}y + b_{10} \end{bmatrix}$$

### How do we get an Affine Transformation?

1. Excellent question. We mentioned that an Affine Transformation is basically a **relation** between two images. The information about this relation can come, roughly, in two ways:
  - (a) We know both  $X$  and  $T$  and we also know that they are related. Then our job is to find  $M$
  - (b) We know  $M$  and  $X$ . To obtain  $T$  we only need to apply  $T = M \cdot X$ . Our information for  $M$  may be explicit (i.e. have the 2-by-3 matrix) or it can come as a geometric relation between points.
2. Let's explain a little bit better (b). Since  $M$  relates 02 images, we can analyze the simplest case in which it relates three points in both images. Look at the figure below:



the points 1, 2 and 3 (forming a triangle in image 1) are mapped into image 2, still forming a triangle, but now they have changed notoriously. If we find the Affine Transformation with these 3 points (you can choose them as you like), then we can apply this found relation to the whole pixels in the image.

## Code

### 1. What does this program do?

- Loads an image
- Applies an Affine Transform to the image. This Transform is obtained from the relation between three points. We use the function `warpAffine` for that purpose.
- Applies a Rotation to the image after being transformed. This rotation is with respect to the image center
- Waits until the user exits the program

2. The tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

/// Global variables
char* source_window = "Source image";
char* warp_window = "Warp";
char* warp_rotate_window = "Warp + Rotate";

/** @function main */
int main( int argc, char** argv )
{
    Point2f srcTri[3];
    Point2f dstTri[3];

    Mat rot_mat( 2, 3, CV_32FC1 );
    Mat warp_mat( 2, 3, CV_32FC1 );
    Mat src, warp_dst, warp_rotate_dst;

    /// Load the image
    src = imread( argv[1], 1 );
```

```

/// Set the dst image the same type and size as src
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );

/// Set your 3 points to calculate the Affine Transform
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );

/// Get the Affine Transform
warp_mat = getAffineTransform( srcTri, dstTri );

/// Apply the Affine Transform just found to the src image
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );

/** Rotating the image after Warp */

/// Compute a rotation matrix with respect to the center of the image
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;

/// Get the rotation matrix with the specifications above
rot_mat = getRotationMatrix2D( center, angle, scale );

/// Rotate the warped image
warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size() );

/// Show what you got
namedWindow( source_window, CV_WINDOW_AUTOSIZE );
imshow( source_window, src );

namedWindow( warp_window, CV_WINDOW_AUTOSIZE );
imshow( warp_window, warp_dst );

namedWindow( warp_rotate_window, CV_WINDOW_AUTOSIZE );
imshow( warp_rotate_window, warp_rotate_dst );

/// Wait until user exits the program
waitKey(0);

return 0;
}

```

## Explanation

1. Declare some variables we will use, such as the matrices to store our results and 2 arrays of points to store the 2D points that define our Affine Transform.

```

Point2f srcTri[3];
Point2f dstTri[3];

Mat rot_mat( 2, 3, CV_32FC1 );

```

```
Mat warp_mat( 2, 3, CV_32FC1 );
Mat src, warp_dst, warp_rotate_dst;
```

2. Load an image:

```
src = imread( argv[1], 1 );
```

3. Initialize the destination image as having the same size and type as the source:

```
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );
```

4. **Affine Transform:** As we explained lines above, we need two sets of 3 points to derive the affine transform relation. Take a look:

```
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );
```

You may want to draw the points to make a better idea of how they change. Their locations are approximately the same as the ones depicted in the example figure (in the Theory section). You may note that the size and orientation of the triangle defined by the 3 points change.

5. Armed with both sets of points, we calculate the Affine Transform by using OpenCV function `getAffineTransform`:

```
warp_mat = getAffineTransform( srcTri, dstTri );
```

We get as an output a  $2 \times 3$  matrix (in this case **warp\_mat**)

6. We apply the Affine Transform just found to the src image

```
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );
```

with the following arguments:

- **src:** Input image
- **warp\_dst:** Output image
- **warp\_mat:** Affine transform
- **warp\_dst.size():** The desired size of the output image

We just got our first transformed image! We will display it in one bit. Before that, we also want to rotate it...

7. **Rotate:** To rotate an image, we need to know two things:

- (a) The center with respect to which the image will rotate
- (b) The angle to be rotated. In OpenCV a positive angle is counter-clockwise
- (c) *Optional:* A scale factor

We define these parameters with the following snippet:

```
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;
```



- We generate the rotation matrix with the OpenCV function `getRotationMatrix2D`, which returns a  $2 \times 3$  matrix (in this case `rot_mat`)

```
rot_mat = getRotationMatrix2D( center, angle, scale );
```

- We now apply the found rotation to the output of our previous Transformation.

```
warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size() );
```

- Finally, we display our results in two windows plus the original image for good measure:

```
namedWindow( source_window, CV_WINDOW_AUTOSIZE );  
imshow( source_window, src );
```

```
namedWindow( warp_window, CV_WINDOW_AUTOSIZE );  
imshow( warp_window, warp_dst );
```

```
namedWindow( warp_rotate_window, CV_WINDOW_AUTOSIZE );  
imshow( warp_rotate_window, warp_rotate_dst );
```

- We just have to wait until the user exits the program

```
waitKey(0);
```

## Result

- After compiling the code above, we can give it the path of an image as argument. For instance, for a picture like:

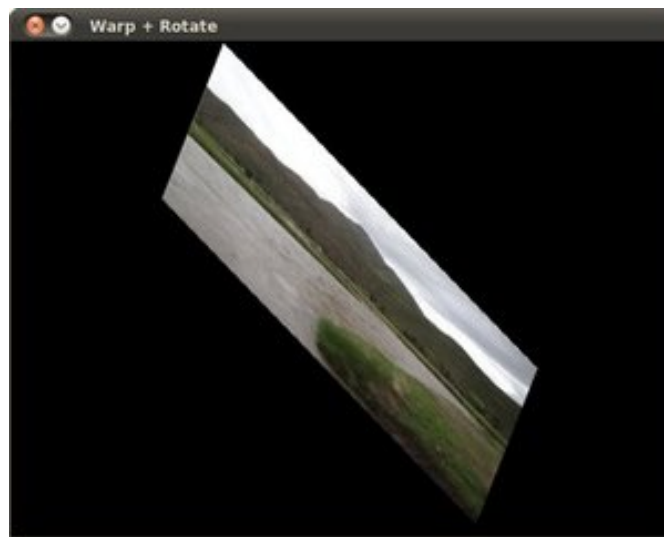


after applying the first Affine Transform we obtain:





and finally, after applying a negative rotation (remember negative means clockwise) and a scale factor, we get:



## 3.15 Histogram Equalization

### Goal

In this tutorial you will learn:

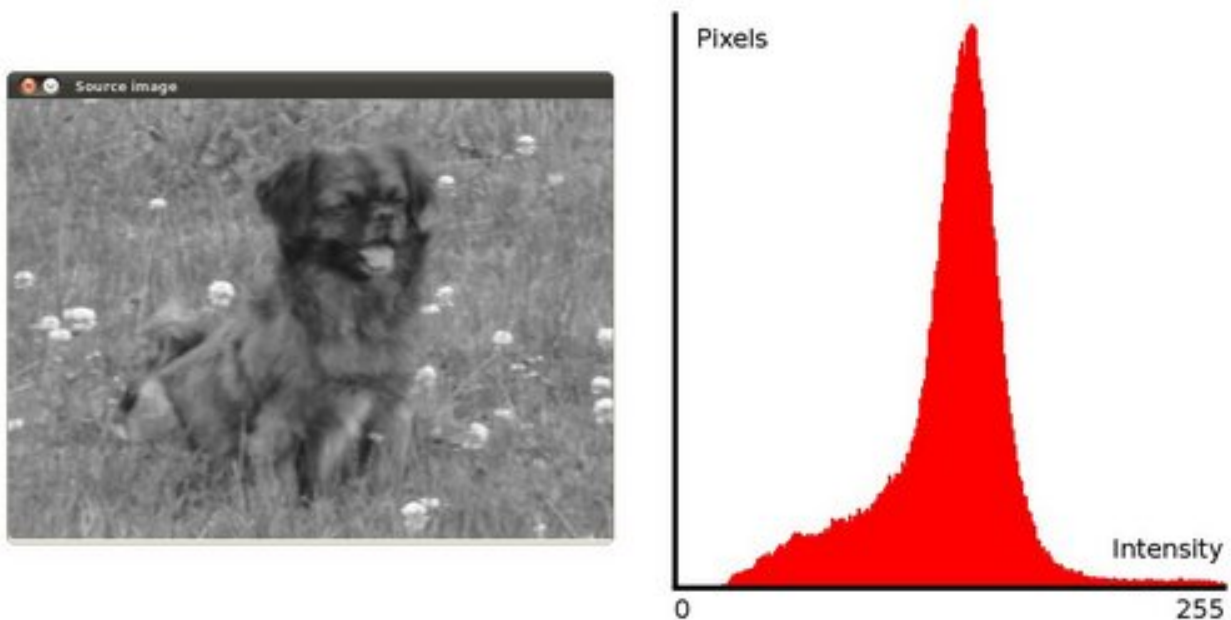
- What an image histogram is and why it is useful
- To equalize histograms of images by using the OpenCV function:`equalize_hist`:`equalizeHist` <>

### Theory

#### What is an Image Histogram?

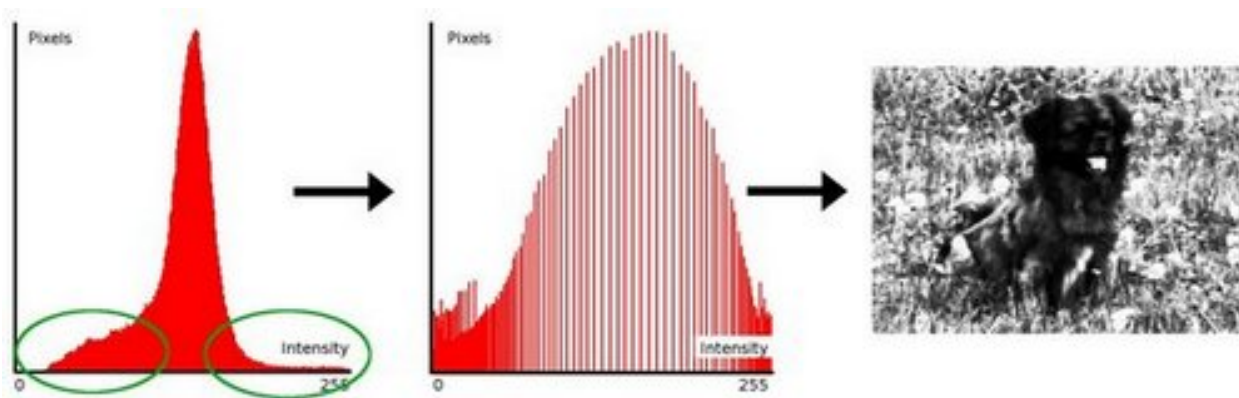
- It is a graphical representation of the intensity distribution of an image.

- It quantifies the number of pixels for each intensity value considered.



### What is Histogram Equalization?

- It is a method that improves the contrast in an image, in order to stretch out the intensity range.
- To make it clearer, from the image above, you can see that the pixels seem clustered around the middle of the available range of intensities. What Histogram Equalization does is to *stretch out* this range. Take a look at the figure below: The green circles indicate the *underpopulated* intensities. After applying the equalization, we get an histogram like the figure in the center. The resulting image is shown in the picture at right.



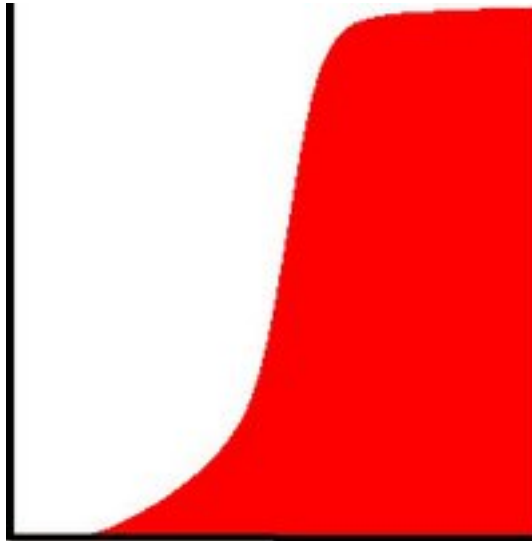
### How does it work?

- Equalization implies *mapping* one distribution (the given histogram) to another distribution (a wider and more uniform distribution of intensity values) so the intensity values are spreaded over the whole range.
- To accomplish the equalization effect, the remapping should be the *cumulative distribution function (cdf)* (more

details, refer to *Learning OpenCV*). For the histogram  $H(i)$ , its *cumulative distribution*  $H'(i)$  is:

$$H'(i) = \sum_{0 \leq j < i} H(j)$$

To use this as a remapping function, we have to normalize  $H'(i)$  such that the maximum value is 255 ( or the maximum value for the intensity of the image ). From the example above, the cumulative function is:



- Finally, we use a simple remapping procedure to obtain the intensity values of the equalized image:

$$\text{equalized}(x, y) = H'(\text{src}(x, y))$$

## Code

- **What does this program do?**
  - Loads an image
  - Convert the original image to grayscale
  - Equalize the Histogram by using the OpenCV function `EqualizeHist`
  - Display the source and equalized images in a window.

- **Downloadable code:** Click [here](#)

- **Code at glance:**

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

/** @function main */
int main( int argc, char** argv )
{
    Mat src, dst;
```

```

char* source_window = "Source image";
char* equalized_window = "Equalized Image";

// Load image
src = imread( argv[1], 1 );

if( !src.data )
{ cout<<"Usage: ./Histogram_Demo <path_to_image>"<<endl;
  return -1;}

// Convert to grayscale
cvtColor( src, src, CV_BGR2GRAY );

// Apply Histogram Equalization
equalizeHist( src, dst );

// Display results
namedWindow( source_window, CV_WINDOW_AUTOSIZE );
namedWindow( equalized_window, CV_WINDOW_AUTOSIZE );

imshow( source_window, src );
imshow( equalized_window, dst );

// Wait until user exits the program
waitKey(0);

return 0;
}

```

## Explanation

1. Declare the source and destination images as well as the windows names:

```
Mat src, dst;
```

```
char* source_window = "Source image";
char* equalized_window = "Equalized Image";
```

2. Load the source image:

```
src = imread( argv[1], 1 );
```

```
if( !src.data )
{ cout<<"Usage: ./Histogram_Demo <path_to_image>"<<endl;
  return -1;}

```

3. Convert it to grayscale:

```
cvtColor( src, src, CV_BGR2GRAY );
```

4. Apply histogram equalization with the function `equalizeHist` :

```
equalizeHist( src, dst );
```

As it can be easily seen, the only arguments are the original image and the output (equalized) image.

5. Display both images (original and equalized) :

```
namedWindow( source_window, CV_WINDOW_AUTOSIZE );  
namedWindow( equalized_window, CV_WINDOW_AUTOSIZE );
```

```
imshow( source_window, src );  
imshow( equalized_window, dst );
```

6. Wait until user exists the program

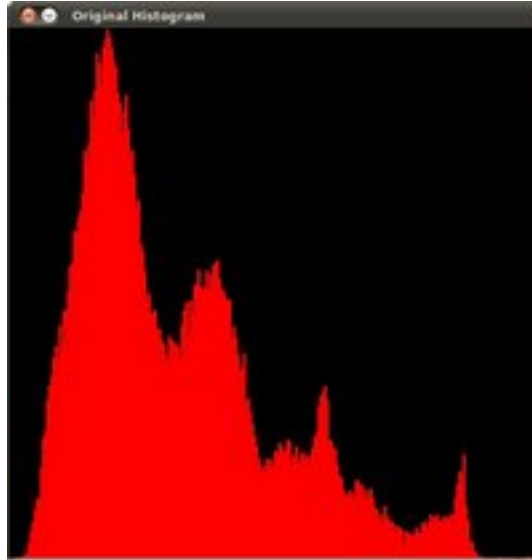
```
waitKey(0);  
return 0;
```

## Results

1. To appreciate better the results of equalization, let's introduce an image with not much contrast, such as:



which, by the way, has this histogram:

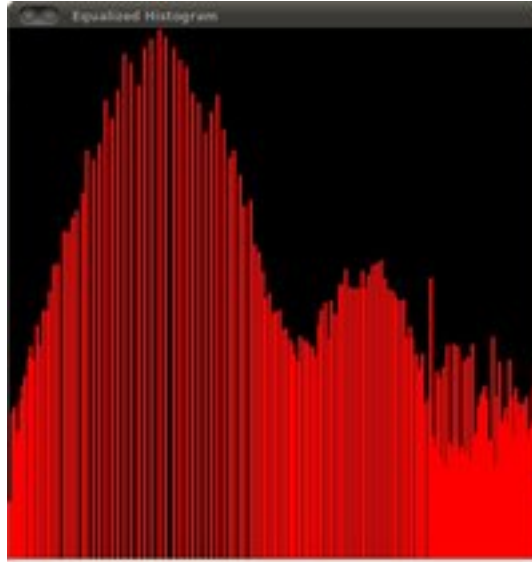


notice that the pixels are clustered around the center of the histogram.

2. After applying the equalization with our program, we get this result:



this image has certainly more contrast. Check out its new histogram like this:



Notice how the number of pixels is more distributed through the intensity range.

---

**Note:** Are you wondering how did we draw the Histogram figures shown above? Check out the following tutorial!

---

## 3.16 Histogram Calculation

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `split` to divide an image into its correspondent planes.
- To calculate histograms of arrays of images by using the OpenCV function `calcHist`
- To normalize an array by using the function `normalize`

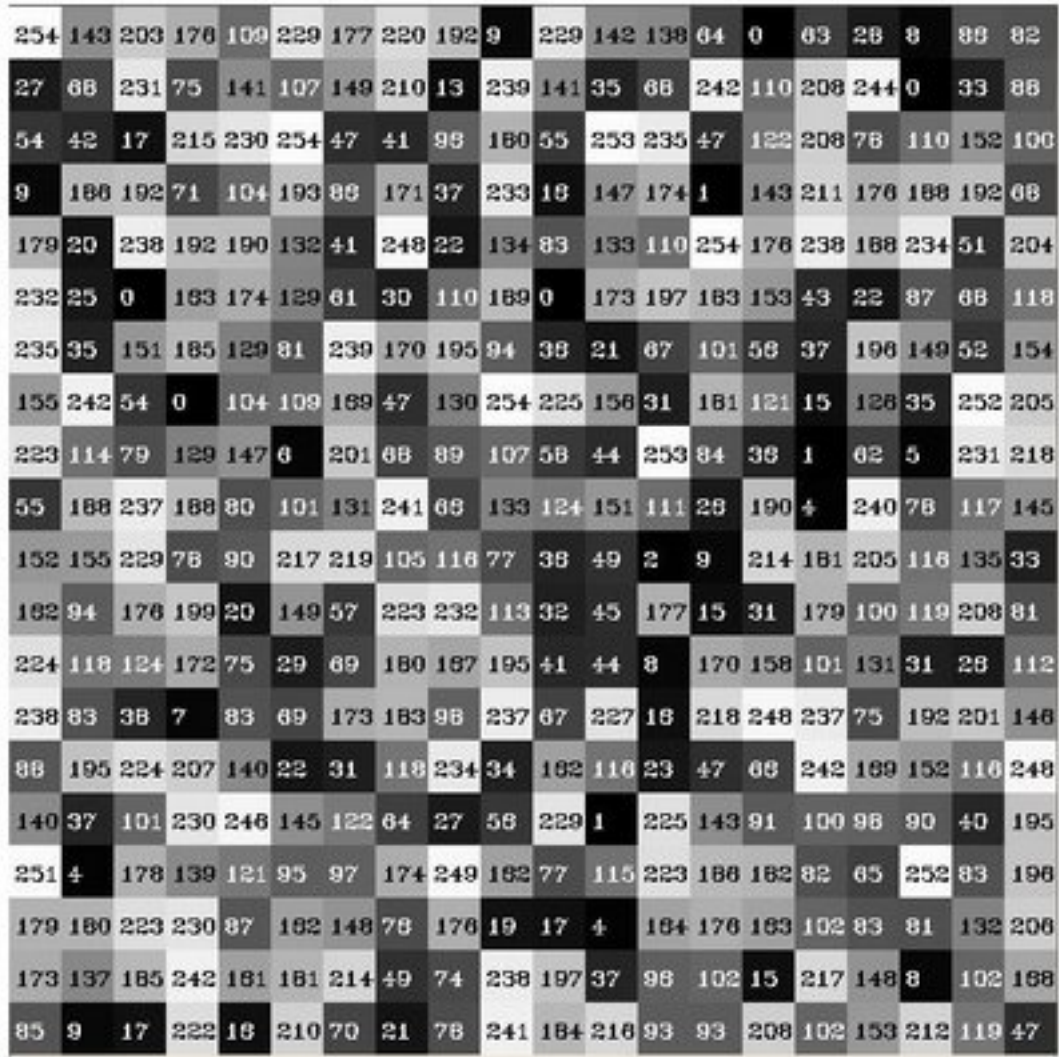
---

**Note:** In the last tutorial (*Histogram Equalization*) we talked about a particular kind of histogram called *Image histogram*. Now we will considerate it in its more general concept. Read on!

---

### What are histograms?

- Histograms are collected *counts* of data organized into a set of predefined *bins*
- When we say *data* we are not restricting it to be intensity values (as we saw in the previous Tutorial). The data collected can be whatever feature you find useful to describe your image.
- Let's see an example. Imagine that a Matrix contains information of an image (i.e. intensity in the range 0–255):



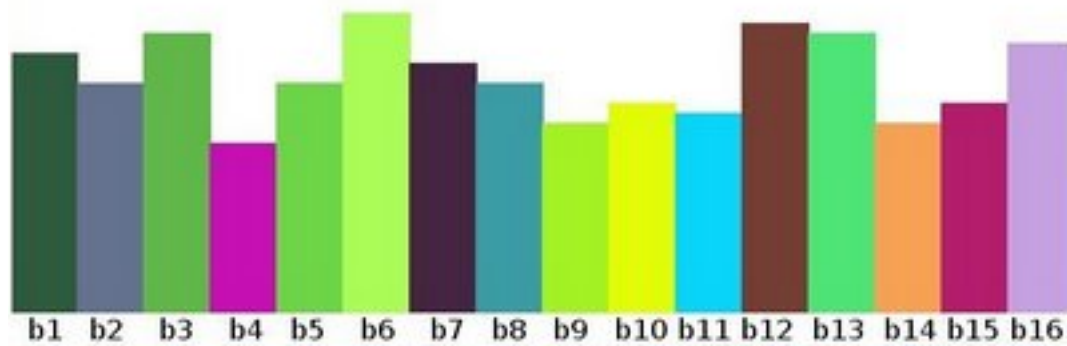
- What happens if we want to *count* this data in an organized way? Since we know that the *range* of information value for this case is 256 values, we can segment our range in subparts (called **bins**) like:

$$[0, 255] = [0, 15] \cup [16, 31] \cup \dots \cup [240, 255]$$

$$\text{range} = \text{bin}_1 \cup \text{bin}_2 \cup \dots \cup \text{bin}_{n=15}$$

and we can keep count of the number of pixels that fall in the range of each  $\text{bin}_i$ . Applying this to the example above we get the image below ( axis x represents the bins and axis y the number of pixels in each of them).





- This was just a simple example of how an histogram works and why it is useful. An histogram can keep count not only of color intensities, but of whatever image features that we want to measure (i.e. gradients, directions, etc).
- Let's identify some parts of the histogram:
  1. **dims**: The number of parameters you want to collect data of. In our example, **dims = 1** because we are only counting the intensity values of each pixel (in a greyscale image).
  2. **bins**: It is the number of **subdivisions** in each dim. In our example, **bins = 16**
  3. **range**: The limits for the values to be measured. In this case: **range = [0,255]**
- What if you want to count two features? In this case your resulting histogram would be a 3D plot (in which x and y would be  $bin_x$  and  $bin_y$  for each feature and z would be the number of counts for each combination of  $(bin_x, bin_y)$ ). The same would apply for more features (of course it gets trickier).

### What OpenCV offers you

For simple purposes, OpenCV implements the function `calcHist`, which calculates the histogram of a set of arrays (usually images or image planes). It can operate with up to 32 dimensions. We will see it in the code below!

### Code

- **What does this program do?**
  - Loads an image
  - Splits the image into its R, G and B planes using the function `split`
  - Calculate the Histogram of each 1-channel plane by calling the function `calcHist`
  - Plot the three histograms in a window
- **Downloadable code:** Click [here](#)
- **Code at glance:**

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;
```

```

/** @function main */
int main( int argc, char** argv )
{
    Mat src, dst;

    /// Load image
    src = imread( argv[1], 1 );

    if( !src.data )
        { return -1; }

    /// Separate the image in 3 places ( R, G and B )
    vector<Mat> rgb_planes;
    split( src, rgb_planes );

    /// Establish the number of bins
    int histSize = 255;

    /// Set the ranges ( for R,G,B )
    float range[] = { 0, 255 } ;
    const float* histRange = { range };

    bool uniform = true; bool accumulate = false;

    Mat r_hist, g_hist, b_hist;

    /// Compute the histograms:
    calcHist( &rgb_planes[0], 1, 0, Mat(), r_hist, 1, &histSize, &histRange, uniform, accumulate );
    calcHist( &rgb_planes[1], 1, 0, Mat(), g_hist, 1, &histSize, &histRange, uniform, accumulate );
    calcHist( &rgb_planes[2], 1, 0, Mat(), b_hist, 1, &histSize, &histRange, uniform, accumulate );

    /// Draw the histograms for R, G and B
    int hist_w = 400; int hist_h = 400;
    int bin_w = cvRound( (double) hist_w/histSize );

    Mat histImage( hist_w, hist_h, CV_8UC3, Scalar( 0,0,0) );

    /// Normalize the result to [ 0, histImage.rows ]
    normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
    normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
    normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );

    /// Draw for each channel
    for( int i = 1; i < histSize; i++ )
    {
        line( histImage, Point( bin_w*(i-1), hist_h - cvRound(r_hist.at<float>(i-1)) ) ,
            Point( bin_w*(i), hist_h - cvRound(r_hist.at<float>(i)) ) ,
            Scalar( 0, 0, 255), 2, 8, 0 );
        line( histImage, Point( bin_w*(i-1), hist_h - cvRound(g_hist.at<float>(i-1)) ) ,
            Point( bin_w*(i), hist_h - cvRound(g_hist.at<float>(i)) ) ,
            Scalar( 0, 255, 0), 2, 8, 0 );
        line( histImage, Point( bin_w*(i-1), hist_h - cvRound(b_hist.at<float>(i-1)) ) ,
            Point( bin_w*(i), hist_h - cvRound(b_hist.at<float>(i)) ) ,
            Scalar( 255, 0, 0), 2, 8, 0 );
    }

    /// Display
    namedWindow("calcHist Demo", CV_WINDOW_AUTOSIZE );

```

```
imshow("calcHist Demo", histImage );

waitKey(0);

return 0;

}
```

### Explanation

1. Create the necessary matrices:

```
Mat src, dst;
```

2. Load the source image

```
src = imread( argv[1], 1 );

if( !src.data )
{ return -1; }
```

3. Separate the source image in its three R,G and B planes. For this we use the OpenCV function `split`:

```
vector<Mat> rgb_planes;
split( src, rgb_planes );
```

our input is the image to be divided (this case with three channels) and the output is a vector of Mat )

4. Now we are ready to start configuring the **histograms** for each plane. Since we are working with the R, G and B planes, we know that our values will range in the interval [0,255]

- (a) Establish number of bins (5, 10...):

```
int histSize = 255;
```

- (b) Set the range of values (as we said, between 0 and 255 )

```
/// Set the ranges ( for R,G,B )
float range[] = { 0, 255 } ;
const float* histRange = { range };
```

- (c) We want our bins to have the same size (uniform) and to clear the histograms in the beginning, so:

```
bool uniform = true; bool accumulate = false;
```

- (d) Finally, we create the Mat objects to save our histograms. Creating 3 (one for each plane):

```
Mat r_hist, g_hist, b_hist;
```

- (e) We proceed to calculate the histograms by using the OpenCV function `calcHist`:

```
/// Compute the histograms:
calcHist( &rgb_planes[0], 1, 0, Mat(), r_hist, 1, &histSize, &histRange, uniform, accumulate );
calcHist( &rgb_planes[1], 1, 0, Mat(), g_hist, 1, &histSize, &histRange, uniform, accumulate );
calcHist( &rgb_planes[2], 1, 0, Mat(), b_hist, 1, &histSize, &histRange, uniform, accumulate );
```

where the arguments are:

- **&rgb\_planes[0]**: The source array(s)
- **1**: The number of source arrays (in this case we are using 1. We can enter here also a list of arrays )

- **0**: The channel (*dim*) to be measured. In this case it is just the intensity (each array is single-channel) so we just write 0.
- **Mat()**: A mask to be used on the source array ( zeros indicating pixels to be ignored ). If not defined it is not used
- **r\_hist**: The Mat object where the histogram will be stored
- **1**: The histogram dimensionality.
- **histSize**: The number of bins per each used dimension
- **histRange**: The range of values to be measured per each dimension
- **uniform** and **accumulate**: The bin sizes are the same and the histogram is cleared at the beginning.

5. Create an image to display the histograms:

```
// Draw the histograms for R, G and B
int hist_w = 400; int hist_h = 400;
int bin_w = cvRound( (double) hist_w/histSize );

Mat histImage( hist_w, hist_h, CV_8UC3, Scalar( 0,0,0) );
```

6. Notice that before drawing, we first *normalize* the histogram so its values fall in the range indicated by the parameters entered:

```
// Normalize the result to [ 0, histImage.rows ]
normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
```

this function receives these arguments:

- **r\_hist**: Input array
- **r\_hist**: Output normalized array (can be the same)
- **0** and **histImage.rows**: For this example, they are the lower and upper limits to normalize the values of **r\_hist**
- **NORM\_MINMAX**: Argument that indicates the type of normalization (as described above, it adjusts the values between the two limits set before)
- **-1**: Implies that the output normalized array will be the same type as the input
- **Mat()**: Optional mask

7. Finally, observe that to access the bin (in this case in this 1D-Histogram):

```
/// Draw for each channel
for( int i = 1; i < histSize; i++ )
{
    line( histImage, Point( bin_w*(i-1), hist_h - cvRound(r_hist.at<float>(i-1)) ) ,
          Point( bin_w*i, hist_h - cvRound(r_hist.at<float>(i)) ) ,
          Scalar( 0, 0, 255), 2, 8, 0 );
    line( histImage, Point( bin_w*(i-1), hist_h - cvRound(g_hist.at<float>(i-1)) ) ,
          Point( bin_w*i, hist_h - cvRound(g_hist.at<float>(i)) ) ,
          Scalar( 0, 255, 0), 2, 8, 0 );
    line( histImage, Point( bin_w*(i-1), hist_h - cvRound(b_hist.at<float>(i-1)) ) ,
          Point( bin_w*i, hist_h - cvRound(b_hist.at<float>(i)) ) ,
          Scalar( 255, 0, 0), 2, 8, 0 );
}
```

we use the expression:

```
.. code-block:: cpp  
  
    r_hist.at<float>(i)
```

where  $i$  indicates the dimension. If it were a 2D-histogram we would use something like:

```
.. code-block:: cpp  
  
    r_hist.at<float>( i, j )
```

8. Finally we display our histograms and wait for the user to exit:

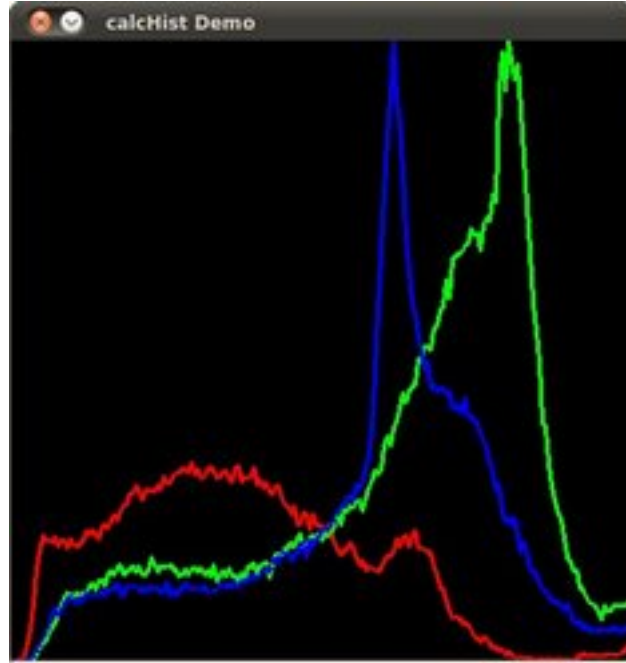
```
namedWindow("calcHist Demo", CV_WINDOW_AUTOSIZE );  
imshow("calcHist Demo", histImage );  
  
waitKey(0);  
  
return 0;
```

## Result

1. Using as input argument an image like the shown below:



2. Produces the following histogram:



## 3.17 Histogram Comparison

### Goal

In this tutorial you will learn how to:

- Use the function `compareHist` to get a numerical parameter that express how well two histograms match with each other.
- Use different metrics to compare histograms

### Theory

- To compare two histograms (  $H_1$  and  $H_2$  ), first we have to choose a *metric* ( $d(H_1, H_2)$ ) to express how well both histograms match.
- OpenCV implements the function `compareHist` to perform a comparison. It also offers 4 different metrics to compute the matching:

#### 1. Correlation ( `CV_COMP_CORREL` )

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H}_1)(H_2(I) - \bar{H}_2)}{\sqrt{\sum_I (H_1(I) - \bar{H}_1)^2 \sum_I (H_2(I) - \bar{H}_2)^2}}$$

where

$$\bar{H}_k = \frac{1}{N} \sum_J H_k(J)$$

and N is the total number of histogram bins.

## 2. Chi-Square ( CV\_COMP\_CHISQR )

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

## 3. Intersection ( method=CV\_COMP\_INTERSECT )

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

## 4. Bhattacharyya distance ( CV\_COMP\_BHATTACHARYYA )

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H}_1 \bar{H}_2 N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

## Code

- **What does this program do?**

- Loads a *base image* and 2 *test images* to be compared with it.
- Generate 1 image that is the lower half of the *base image*
- Convert the images to HSV format
- Calculate the H-S histogram for all the images and normalize them in order to compare them.
- Compare the histogram of the *base image* with respect to the 2 test histograms, the histogram of the lower half base image and with the same base image histogram.
- Display the numerical matching parameters obtained.

- **Downloadable code:** [Click here](#)

- **Code at glance:**

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;

/** @function main */
int main( int argc, char** argv )
{
    Mat src_base, hsv_base;
    Mat src_test1, hsv_test1;
    Mat src_test2, hsv_test2;
    Mat hsv_half_down;

    /// Load three images with different environment settings
    if( argc < 4 )
    { printf("** Error. Usage: ./compareHist_Demo <image_settings0> <image_setting1> <image_settings2>\n");
      return -1;
    }
}
```

```

src_base = imread( argv[1], 1 );
src_test1 = imread( argv[2], 1 );
src_test2 = imread( argv[3], 1 );

/// Convert to HSV
cvtColor( src_base, hsv_base, CV_BGR2HSV );
cvtColor( src_test1, hsv_test1, CV_BGR2HSV );
cvtColor( src_test2, hsv_test2, CV_BGR2HSV );

hsv_half_down = hsv_base( Range( hsv_base.rows/2, hsv_base.rows - 1 ), Range( 0, hsv_base.cols - 1 ) );

/// Using 30 bins for hue and 32 for saturation
int h_bins = 50; int s_bins = 60;
int histSize[] = { h_bins, s_bins };

// hue varies from 0 to 256, saturation from 0 to 180
float h_ranges[] = { 0, 256 };
float s_ranges[] = { 0, 180 };

const float* ranges[] = { h_ranges, s_ranges };

// Use the 0-th and 1-st channels
int channels[] = { 0, 1 };

/// Histograms
MatND hist_base;
MatND hist_half_down;
MatND hist_test1;
MatND hist_test2;

/// Calculate the histograms for the HSV images
calcHist( &hsv_base, 1, channels, Mat(), hist_base, 2, histSize, ranges, true, false );
normalize( hist_base, hist_base, 0, 1, NORM_MINMAX, -1, Mat() );

calcHist( &hsv_half_down, 1, channels, Mat(), hist_half_down, 2, histSize, ranges, true, false );
normalize( hist_half_down, hist_half_down, 0, 1, NORM_MINMAX, -1, Mat() );

calcHist( &hsv_test1, 1, channels, Mat(), hist_test1, 2, histSize, ranges, true, false );
normalize( hist_test1, hist_test1, 0, 1, NORM_MINMAX, -1, Mat() );

calcHist( &hsv_test2, 1, channels, Mat(), hist_test2, 2, histSize, ranges, true, false );
normalize( hist_test2, hist_test2, 0, 1, NORM_MINMAX, -1, Mat() );

/// Apply the histogram comparison methods
for( int i = 0; i < 4; i++ )
{
    int compare_method = i;
    double base_base = compareHist( hist_base, hist_base, compare_method );
    double base_half = compareHist( hist_base, hist_half_down, compare_method );
    double base_test1 = compareHist( hist_base, hist_test1, compare_method );
    double base_test2 = compareHist( hist_base, hist_test2, compare_method );

    printf( " Method [%d] Perfect, Base-Half, Base-Test(1), Base-Test(2) : %f, %f, %f, %f \n", i, base_base, base_half, base_test1, base_test2 );
}

printf( "Done \n" );

return 0;
}

```



## Explanation

1. Declare variables such as the matrices to store the base image and the two other images to compare ( RGB and HSV )

```
Mat src_base, hsv_base;
Mat src_test1, hsv_test1;
Mat src_test2, hsv_test2;
Mat hsv_half_down;
```

2. Load the base image (src\_base) and the other two test images:

```
if( argc < 4 )
{ printf("** Error. Usage: ./compareHist_Demo <image_settings0> <image_setting1> <image_settings2>\n");
  return -1;
}
```

```
src_base = imread( argv[1], 1 );
src_test1 = imread( argv[2], 1 );
src_test2 = imread( argv[3], 1 );
```

3. Convert them to HSV format:

```
cvtColor( src_base, hsv_base, CV_BGR2HSV );
cvtColor( src_test1, hsv_test1, CV_BGR2HSV );
cvtColor( src_test2, hsv_test2, CV_BGR2HSV );
```

4. Also, create an image of half the base image (in HSV format):

```
hsv_half_down = hsv_base( Range( hsv_base.rows/2, hsv_base.rows - 1 ), Range( 0, hsv_base.cols - 1 ) );
```

5. Initialize the arguments to calculate the histograms (bins, ranges and channels H and S ).

```
int h_bins = 50; int s_bins = 32;
int histSize[] = { h_bins, s_bins };

float h_ranges[] = { 0, 256 };
float s_ranges[] = { 0, 180 };

const float* ranges[] = { h_ranges, s_ranges };

int channels[] = { 0, 1 };
```

6. Create the MatND objects to store the histograms:

```
MatND hist_base;
MatND hist_half_down;
MatND hist_test1;
MatND hist_test2;
```

7. Calculate the Histograms for the base image, the 2 test images and the half-down base image:

```
calcHist( &hsv_base, 1, channels, Mat(), hist_base, 2, histSize, ranges, true, false );
normalize( hist_base, hist_base, 0, 1, NORM_MINMAX, -1, Mat() );

calcHist( &hsv_half_down, 1, channels, Mat(), hist_half_down, 2, histSize, ranges, true, false );
normalize( hist_half_down, hist_half_down, 0, 1, NORM_MINMAX, -1, Mat() );

calcHist( &hsv_test1, 1, channels, Mat(), hist_test1, 2, histSize, ranges, true, false );
normalize( hist_test1, hist_test1, 0, 1, NORM_MINMAX, -1, Mat() );
```

```
calcHist( &hsv_test2, 1, channels, Mat(), hist_test2, 2, histSize, ranges, true, false );
normalize( hist_test2, hist_test2, 0, 1, NORM_MINMAX, -1, Mat() );
```

- Apply sequentially the 4 comparison methods between the histogram of the base image (hist\_base) and the other histograms:

```
for( int i = 0; i < 4; i++ )
{
    int compare_method = i;
    double base_base = compareHist( hist_base, hist_base, compare_method );
    double base_half = compareHist( hist_base, hist_half_down, compare_method );
    double base_test1 = compareHist( hist_base, hist_test1, compare_method );
    double base_test2 = compareHist( hist_base, hist_test2, compare_method );

    printf( " Method [%d] Perfect, Base-Half, Base-Test(1), Base-Test(2) : %f, %f, %f, %f \n", i, base_base, base_half, base_test1, base_test2 );
}
```

## Results

- We use as input the following images:



where the first one is the base (to be compared to the others), the other 2 are the test images. We will also compare the first image with respect to itself and with respect of half the base image.

- We should expect a perfect match when we compare the base image histogram with itself. Also, compared with the histogram of half the base image, it should present a high match since both are from the same source. For the other two test images, we can observe that they have very different lighting conditions, so the matching should not be very good:
- Here the numeric results:

<i>Method</i>	<b>Base - Base</b>	<b>Base - Half</b>	<b>Base - Test 1</b>	<b>Base - Test 2</b>
<i>Correlation</i>	1.000000	0.930766	0.182073	0.120447
<i>Chi-square</i>	0.000000	4.940466	21.184536	49.273437
<i>Intersection</i>	24.391548	14.959809	3.889029	5.775088
<i>Bhattacharyya</i>	0.000000	0.222609	0.646576	0.801869

For the *Correlation* and *Intersection* methods, the higher the metric, the more accurate the match. As we can see, the match *base-base* is the highest of all as expected. Also we can observe that the match *base-half* is the second best match (as we predicted). For the other two metrics, the less the result, the better the match. We can observe that the matches between the test 1 and test 2 with respect to the base are worse, which again, was expected.

## 3.18 Back Projection

### Goal

In this tutorial you will learn:

- What is Back Projection and why it is useful
- How to use the OpenCV function `calcBackProject` to calculate Back Projection
- How to mix different channels of an image by using the OpenCV function `mixChannels`

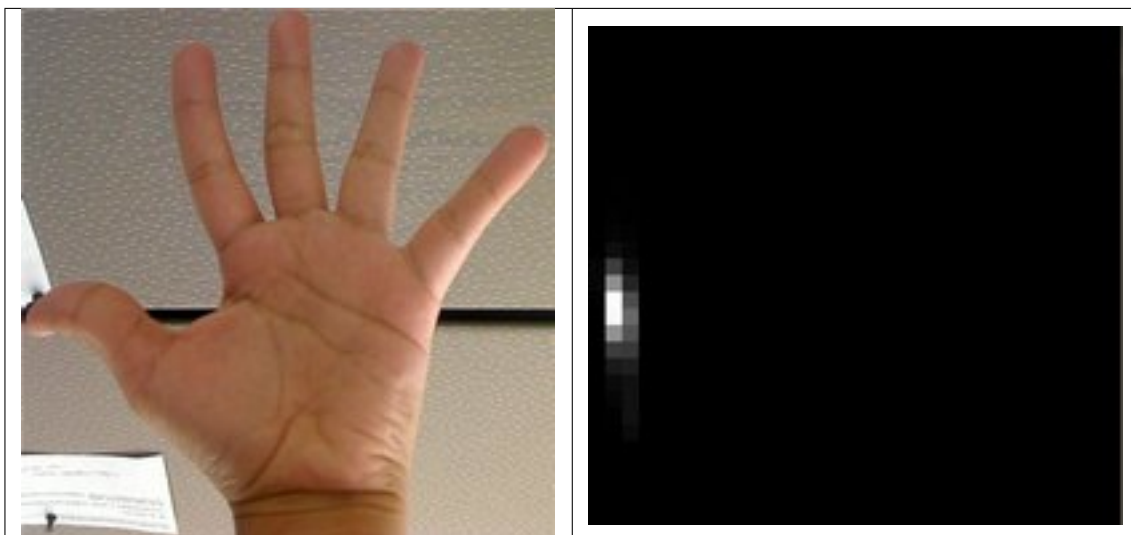
### Theory

#### What is Back Projection?

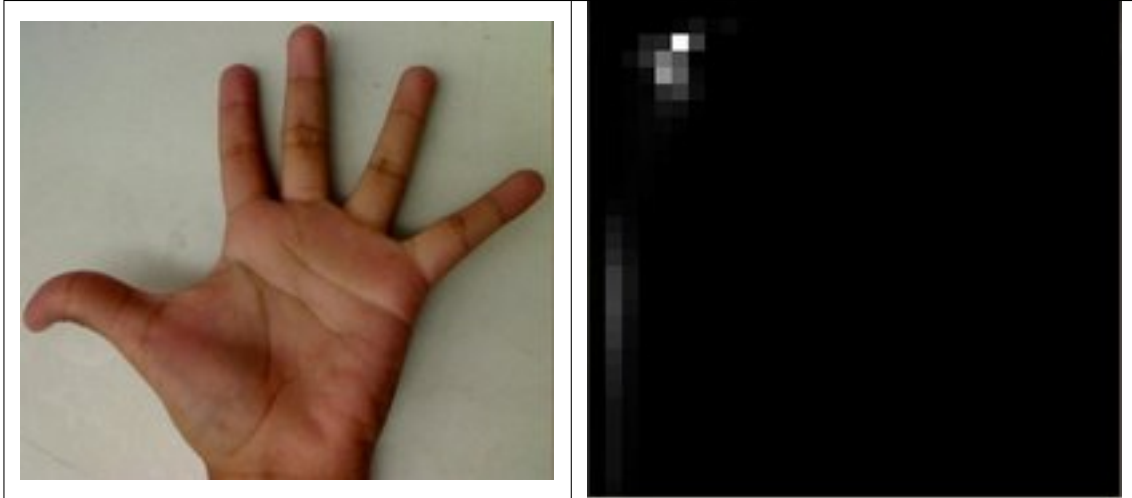
- Back Projection is a way of recording how well the pixels of a given image fit the distribution of pixels in a histogram model.
- To make it simpler: For Back Projection, you calculate the histogram model of a feature and then use it to find this feature in an image.
- Application example: If you have a histogram of flesh color (say, a Hue-Saturation histogram ), then you can use it to find flesh color areas in an image:

#### How does it work?

- We explain this by using the skin example:
- Let's say you have gotten a skin histogram (Hue-Saturation) based on the image below. The histogram besides is going to be our *model histogram* (which we know represents a sample of skin tonality). You applied some mask to capture only the histogram of the skin area:



- Now, let's imagine that you get another hand image (Test Image) like the one below: (with its respective histogram):



- What we want to do is to use our *model histogram* (that we know represents a skin tonality) to detect skin areas in our Test Image. Here are the steps
  1. In each pixel of our Test Image (i.e.  $p(i, j)$ ), collect the data and find the correspondent bin location for that pixel (i.e.  $(h_{i,j}, s_{i,j})$ ).
  2. Lookup the *model histogram* in the correspondent bin -  $(h_{i,j}, s_{i,j})$  - and read the bin value.
  3. Store this bin value in a new image (*BackProjection*). Also, you may consider to normalize the *model histogram* first, so the output for the Test Image can be visible for you.
  4. Applying the steps above, we get the following BackProjection image for our Test Image:



5. In terms of statistics, the values stored in *BackProjection* represent the *probability* that a pixel in *Test Image* belongs to a skin area, based on the *model histogram* that we use. For instance in our Test image, the brighter areas are more probable to be skin area (as they actually are), whereas the darker areas have less probability (notice that these “dark” areas belong to surfaces that have some shadow on it, which in turns affects the detection).

## Code

- **What does this program do?**
  - Loads an image

- Convert the original to HSV format and separate only *Hue* channel to be used for the Histogram (using the OpenCV function `mixChannels`)
- Let the user to enter the number of bins to be used in the calculation of the histogram.
- Calculate the histogram (and update it if the bins change) and the backprojection of the same image.
- Display the backprojection and the histogram in windows.

- **Downloadable code:**

1. Click [here](#) for the basic version (explained in this tutorial).
2. For stuff slightly fancier (using H-S histograms and floodFill to define a mask for the skin area) you can check the [improved demo](#)
3. ...or you can always check out the classical [camshiftdemo](#) in samples.

- **Code at glance:**

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

#include <iostream>

using namespace cv;
using namespace std;

/// Global Variables
Mat src; Mat hsv; Mat hue;
int bins = 25;

/// Function Headers
void Hist_and_Backproj(int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Read the image
    src = imread( argv[1], 1 );
    /// Transform it to HSV
    cvtColor( src, hsv, CV_BGR2HSV );

    /// Use only the Hue value
    hue.create( hsv.size(), hsv.depth() );
    int ch[] = { 0, 0 };
    mixChannels( &hsv, 1, &hue, 1, ch, 1 );

    /// Create Trackbar to enter the number of bins
    char* window_image = "Source image";
    namedWindow( window_image, CV_WINDOW_AUTOSIZE );
    createTrackbar(" Hue bins: ", window_image, &bins, 180, Hist_and_Backproj );
    Hist_and_Backproj(0, 0);

    /// Show the image
    imshow( window_image, src );

    /// Wait until user exits the program
    waitKey(0);
    return 0;
}
```

```

/**
 * @function Hist_and_Backproj
 * @brief Callback to Trackbar
 */
void Hist_and_Backproj(int, void* )
{
    MatND hist;
    int histSize = MAX( bins, 2 );
    float hue_range[] = { 0, 180 };
    const float* ranges = { hue_range };

    // Get the Histogram and normalize it
    calcHist( &hue, 1, 0, Mat(), hist, 1, &histSize, &ranges, true, false );
    normalize( hist, hist, 0, 255, NORM_MINMAX, -1, Mat() );

    // Get Backprojection
    MatND backproj;
    calcBackProject( &hue, 1, 0, hist, backproj, &ranges, 1, true );

    // Draw the backproj
    imshow( "BackProj", backproj );

    // Draw the histogram
    int w = 400; int h = 400;
    int bin_w = cvRound( (double) w / histSize );
    Mat histImg = Mat::zeros( w, h, CV_8UC3 );

    for( int i = 0; i < bins; i ++ )
        { rectangle( histImg, Point( i*bin_w, h ), Point( (i+1)*bin_w, h - cvRound( hist.at<float>(i)*h/255.0 ) ), Scalar( i, i, i ) ); }

    imshow( "Histogram", histImg );
}

```

## Explanation

1. Declare the matrices to store our images and initialize the number of bins to be used by our histogram:

```

Mat src; Mat hsv; Mat hue;
int bins = 25;

```

2. Read the input image and transform it to HSV format:

```

src = imread( argv[1], 1 );
cvtColor( src, hsv, CV_BGR2HSV );

```

3. For this tutorial, we will use only the Hue value for our 1-D histogram (check out the fancier code in the links above if you want to use the more standard H-S histogram, which yields better results):

```

hue.create( hsv.size(), hsv.depth() );
int ch[] = { 0, 0 };
mixChannels( &hsv, 1, &hue, 1, ch, 1 );

```

as you see, we use the function [http://opencv.willowgarage.com/documentation/cpp/core\\_operations\\_on\\_arrays.html?#mixChannels](http://opencv.willowgarage.com/documentation/cpp/core_operations_on_arrays.html?#mixChannels) to get only the channel 0 (Hue) from the hsv image. It gets the following parameters:

- **&hsv:** The source array from which the channels will be copied
- **1:** The number of source arrays

- **&hue**: The destination array of the copied channels
- **1**: The number of destination arrays
- **ch[] = {0,0}**: The array of index pairs indicating how the channels are copied. In this case, the Hue(0) channel of &hsv is being copied to the 0 channel of &hue (1-channel)
- **1**: Number of index pairs

4. Create a **Trackbar** for the user to enter the bin values. Any change on the **Trackbar** means a call to the **Hist\_and\_Backproj** callback function.

```
char* window_image = "Source image";
namedWindow( window_image, CV_WINDOW_AUTOSIZE );
createTrackbar(" Hue bins: ", window_image, &bins, 180, Hist_and_Backproj );
Hist_and_Backproj(0, 0);
```

5. Show the image and wait for the user to exit the program:

```
imshow( window_image, src );

waitKey(0);
return 0;
```

6. **Hist\_and\_Backproj** function: Initialize the arguments needed for `calcHist`. The number of bins comes from the **Trackbar**:

```
void Hist_and_Backproj(int, void* )
{
    MatND hist;
    int histSize = MAX( bins, 2 );
    float hue_range[] = { 0, 180 };
    const float* ranges = { hue_range };
```

7. Calculate the Histogram and normalize it to the range [0, 255]

```
calcHist( &hue, 1, 0, Mat(), hist, 1, &histSize, &ranges, true, false );
normalize( hist, hist, 0, 255, NORM_MINMAX, -1, Mat() );
```

8. Get the Backprojection of the same image by calling the function `calcBackProject`

```
MatND backproj;
calcBackProject( &hue, 1, 0, hist, backproj, &ranges, 1, true );
```

all the arguments are known (the same as used to calculate the histogram), only we add the backproj matrix, which will store the backprojection of the source image (&hue)

9. Display backproj:

```
imshow( "BackProj", backproj );
```

10. Draw the 1-D Hue histogram of the image:

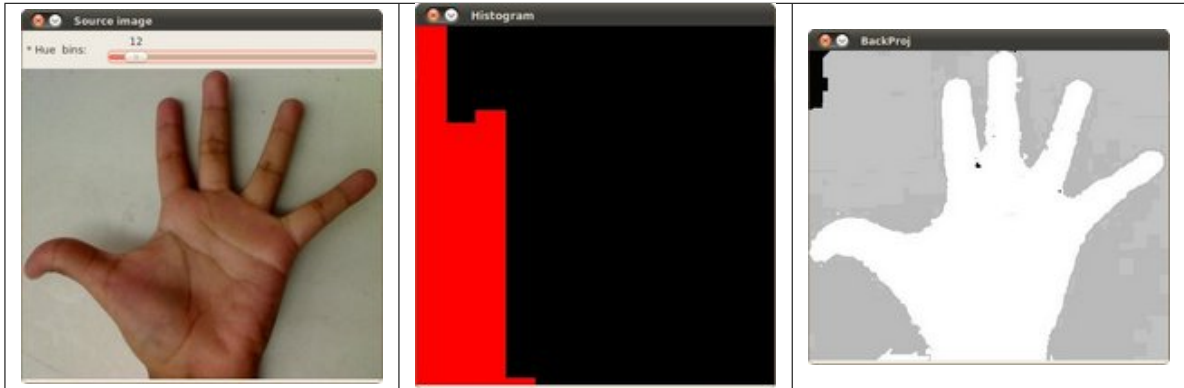
```
int w = 400; int h = 400;
int bin_w = cvRound( (double) w / histSize );
Mat histImg = Mat::zeros( w, h, CV_8UC3 );

for( int i = 0; i < bins; i ++ )
    { rectangle( histImg, Point( i*bin_w, h ), Point( (i+1)*bin_w, h - cvRound( hist.at<float>(i)*h/255.0 ) ), Scalar(0,0,0) ); }

imshow( "Histogram", histImg );
```

## Results

1. Here are the output by using a sample image ( guess what? Another hand ). You can play with the bin values and you will observe how it affects the results:



## 3.19 Template Matching

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `matchTemplate` to search for matches between an image patch and an input image
- Use the OpenCV function `minMaxLoc` to find the maximum and minimum values (as well as their positions) in a given array.

### Theory

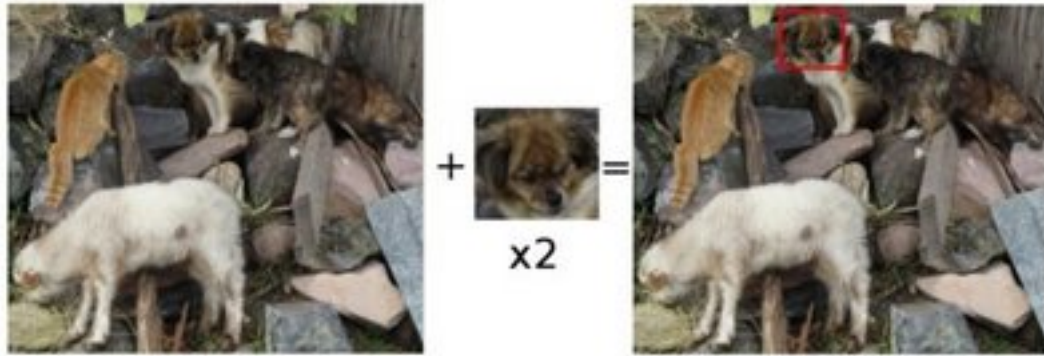
#### What is template matching?

Template matching is a technique for finding areas of an image that match (are similar) to a template image (patch).

#### How does it work?

- We need two primary components:
  1. **Source image (I)**: The image in which we expect to find a match to the template image
  2. **Template image (T)**: The patch image which will be compared to the template image
 our goal is to detect the highest matching area:





- To identify the matching area, we have to *compare* the template image against the source image by sliding it:



- By **sliding**, we mean moving the patch one pixel at a time (left to right, up to down). At each location, a metric is calculated so it represents how “good” or “bad” the match at that location is (or how similar the patch is to that particular area of the source image).
- For each location of **T** over **I**, you *store* the metric in the *result matrix* (**R**). Each location  $(x, y)$  in **R** contains the match metric:



the image above is the result **R** of sliding the patch with a metric **TM\_CCORR\_NORMED**. The brightest locations indicate the highest matches. As you can see, the location marked by the red circle is probably the one with the highest value, so that location (the rectangle formed by that point as a corner and width and height equal to the patch image) is considered the match.

- In practice, we use the function `minMaxLoc` to locate the highest value (or lower, depending of the type of matching method) in the *R* matrix.

### Which are the matching methods available in OpenCV?

Good question. OpenCV implements Template matching in the function `matchTemplate`. The available methods are 6:

1. `method=CV_TM_SQDIFF`

$$R(x, y) = \sum_{x', y'} (T(x', y') - I(x + x', y + y'))^2$$

2. `method=CV_TM_SQDIFF_NORMED`

$$R(x, y) = \frac{\sum_{x', y'} (T(x', y') - I(x + x', y + y'))^2}{\sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x + x', y + y')^2}}$$

3. `method=CV_TM_CCORR`

$$R(x, y) = \sum_{x', y'} (T(x', y') \cdot I(x + x', y + y'))$$

4. `method=CV_TM_CCORR_NORMED`

$$R(x, y) = \frac{\sum_{x', y'} (T(x', y') \cdot I(x + x', y + y'))}{\sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x + x', y + y')^2}}$$

**5. method=CV\_TM\_CCOEFF**

$$R(x, y) = \sum_{x', y'} (T'(x', y') \cdot I(x + x', y + y'))$$

where

$$T'(x', y') = T(x', y') - 1/(w \cdot h) \cdot \sum_{x'', y''} T(x'', y'')$$
$$I'(x + x', y + y') = I(x + x', y + y') - 1/(w \cdot h) \cdot \sum_{x'', y''} I(x + x'', y + y'')$$

**6. method=CV\_TM\_CCOEFF\_NORMED**

$$R(x, y) = \frac{\sum_{x', y'} (T'(x', y') \cdot I'(x + x', y + y'))}{\sqrt{\sum_{x', y'} T'(x', y')^2 \cdot \sum_{x', y'} I'(x + x', y + y')^2}}$$

## Code

- **What does this program do?**

- Loads an input image and a image patch (*template*)
- Perform a template matching procedure by using the OpenCV function `matchTemplate` with any of the 6 matching methods described before. The user can choose the method by entering its selection in the Trackbar.
- Normalize the output of the matching procedure
- Localize the location with higher matching probability
- Draw a rectangle around the area corresponding to the highest match

- **Downloadable code:** Click [here](#)

- **Code at glance:**

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;

/// Global Variables
Mat img; Mat templ; Mat result;
char* image_window = "Source Image";
char* result_window = "Result window";

int match_method;
int max_Trackbar = 5;

/// Function Headers
void MatchingMethod( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load image and template
    img = imread( argv[1], 1 );
```

```

templ = imread( argv[2], 1 );

/// Create windows
namedWindow( image_window, CV_WINDOW_AUTOSIZE );
namedWindow( result_window, CV_WINDOW_AUTOSIZE );

/// Create Trackbar
char* trackbar_label = "Method: \n 0: SQDIFF \n 1: SQDIFF NORMED \n 2: TM CCORR \n 3: TM CCORR NORMED \n 4: TM COEFF";
createTrackbar( trackbar_label, image_window, &match_method, max_Trackbar, MatchingMethod );

MatchingMethod( 0, 0 );

waitKey(0);
return 0;
}

/**
 * @function MatchingMethod
 * @brief Trackbar callback
 */
void MatchingMethod( int, void* )
{
    /// Source image to display
    Mat img_display;
    img.copyTo( img_display );

    /// Create the result matrix
    int result_cols = img.cols - templ.cols + 1;
    int result_rows = img.rows - templ.rows + 1;

    result.create( result_cols, result_rows, CV_32FC1 );

    /// Do the Matching and Normalize
    matchTemplate( img, templ, result, match_method );
    normalize( result, result, 0, 1, NORM_MINMAX, -1, Mat() );

    /// Localizing the best match with minMaxLoc
    double minVal; double maxVal; Point minLoc; Point maxLoc;
    Point matchLoc;

    minMaxLoc( result, &minVal, &maxVal, &minLoc, &maxLoc, Mat() );

    /// For SQDIFF and SQDIFF_NORMED, the best matches are lower values. For all the other methods, the higher the better
    if( match_method == CV_TM_SQDIFF || match_method == CV_TM_SQDIFF_NORMED )
        { matchLoc = minLoc; }
    else
        { matchLoc = maxLoc; }

    /// Show me what you got
    rectangle( img_display, matchLoc, Point( matchLoc.x + templ.cols, matchLoc.y + templ.rows ), Scalar::all(0), 2, 8,
    rectangle( result, matchLoc, Point( matchLoc.x + templ.cols, matchLoc.y + templ.rows ), Scalar::all(0), 2, 8, 0 );

    imshow( image_window, img_display );
    imshow( result_window, result );

    return;
}

```

## Explanation

1. Declare some global variables, such as the image, template and result matrices, as well as the match method and the window names:

```
Mat img; Mat templ; Mat result;
char* image_window = "Source Image";
char* result_window = "Result window";

int match_method;
int max_Trackbar = 5;
```

2. Load the source image and template:

```
img = imread( argv[1], 1 );
templ = imread( argv[2], 1 );
```

3. Create the windows to show the results:

```
namedWindow( image_window, CV_WINDOW_AUTOSIZE );
namedWindow( result_window, CV_WINDOW_AUTOSIZE );
```

4. Create the Trackbar to enter the kind of matching method to be used. When a change is detected the callback function **MatchingMethod** is called.

```
char* trackbar_label = "Method: \n 0: SQDIFF \n 1: SQDIFF NORMED \n 2: TM CCORR \n 3: TM CCORR NORMED \n 4: TM C";
createTrackbar( trackbar_label, image_window, &match_method, max_Trackbar, MatchingMethod );
```

5. Wait until user exits the program.

```
waitKey(0);
return 0;
```

6. Let's check out the callback function. First, it makes a copy of the source image:

```
Mat img_display;
img.copyTo( img_display );
```

7. Next, it creates the result matrix that will store the matching results for each template location. Observe in detail the size of the result matrix (which matches all possible locations for it)

```
int result_cols = img.cols - templ.cols + 1;
int result_rows = img.rows - templ.rows + 1;

result.create( result_cols, result_rows, CV_32FC1 );
```

8. Perform the template matching operation:

```
matchTemplate( img, templ, result, match_method );
```

the arguments are naturally the input image **I**, the template **T**, the result **R** and the match\_method (given by the Trackbar)

9. We normalize the results:

```
normalize( result, result, 0, 1, NORM_MINMAX, -1, Mat() );
```

10. We localize the minimum and maximum values in the result matrix **R** by using [minMaxLoc](#).

```
double minVal; double maxVal; Point minLoc; Point maxLoc;
Point matchLoc;
```

```
minMaxLoc( result, &minVal, &maxVal, &minLoc, &maxLoc, Mat() );
```

the function calls as arguments:

- **result**: The source array
  - **&minVal** and **&maxVal**: Variables to save the minimum and maximum values in **result**
  - **&minLoc** and **&maxLoc**: The Point locations of the minimum and maximum values in the array.
  - **Mat()**: Optional mask
11. For the first two methods ( `CV_SQDIFF` and `CV_SQDIFF_NORMED` ) the best match are the lowest values. For all the others, higher values represent better matches. So, we save the corresponding value in the **matchLoc** variable:

```
if( match_method == CV_TM_SQDIFF || match_method == CV_TM_SQDIFF_NORMED )
    { matchLoc = minLoc; }
else
    { matchLoc = maxLoc; }
```

12. Display the source image and the result matrix. Draw a rectangle around the highest possible matching area:

```
rectangle( img_display, matchLoc, Point( matchLoc.x + templ.cols , matchLoc.y + templ.rows ), Scalar::all(0), 2,
rectangle( result, matchLoc, Point( matchLoc.x + templ.cols , matchLoc.y + templ.rows ), Scalar::all(0), 2, 8, 0

imshow( image_window, img_display );
imshow( result_window, result );
```

## Results

1. Testing our program with an input image such as:



and a template image:



2. Generate the following result matrices (first row are the standard methods SQDIFF, CCORR and CCDEFF, second row are the same methods in its normalized version). In the first column, the darkest is the better match, for the other two columns, the brighter a location, the higher the match.



3. The right match is shown below (black rectangle around the face of the guy at the right). Notice that CCORR and CCDEFF gave erroneous best matches, however their normalized version did it right, this may be due to the fact that we are only considering the “highest match” and not the other possible high matches.



## 3.20 Finding contours in your image

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `findContours`
- Use the OpenCV function `drawContours`

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;
```



```
Mat src; Mat src_gray;
int thresh = 100;
int max_thresh = 255;
RNG rng(12345);

// Function header
void thresh_callback(int, void* );

/** @function main */
int main( int argc, char** argv )
{
    // Load source image and convert it to gray
    src = imread( argv[1], 1 );

    // Convert image to gray and blur it
    cvtColor( src, src_gray, CV_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );

    // Create Window
    char* source_window = "Source";
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, src );

    createTrackbar( " Canny thresh:", "Source", &thresh, max_thresh, thresh_callback );
    thresh_callback( 0, 0 );

    waitKey(0);
    return(0);
}

/** @function thresh_callback */
void thresh_callback(int, void* )
{
    Mat canny_output;
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    // Detect edges using canny
    Canny( src_gray, canny_output, thresh, thresh*2, 3 );
    // Find contours
    findContours( canny_output, contours, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );

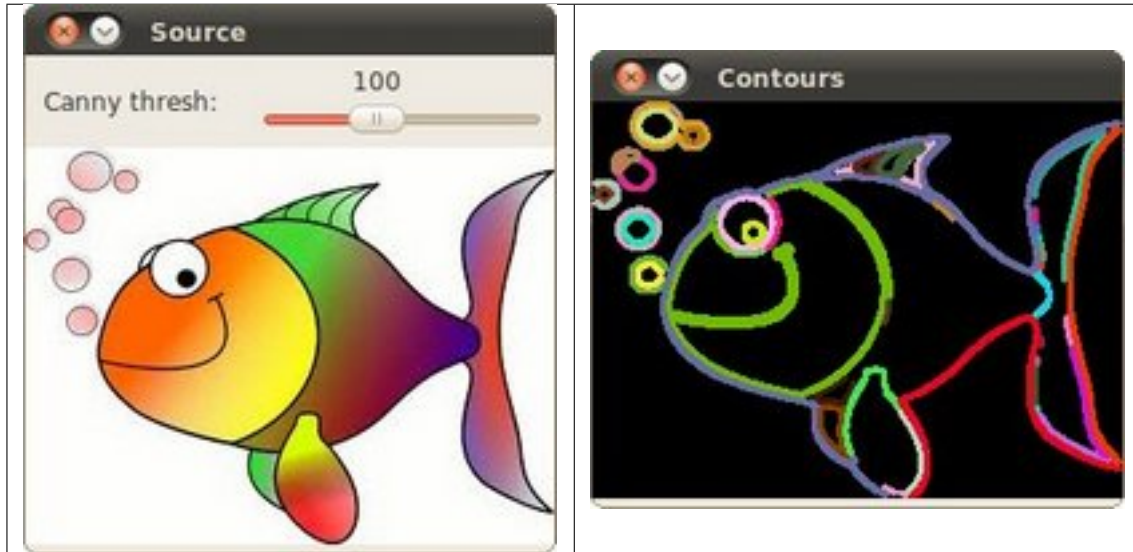
    // Draw contours
    Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
    for( int i = 0; i< contours.size(); i++ )
    {
        Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
        drawContours( drawing, contours, i, color, 2, 8, hierarchy, 0, Point() );
    }

    // Show in a window
    namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
    imshow( "Contours", drawing );
}
```

## Explanation

## Result

1. Here it is:



## 3.21 Convex Hull

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `convexHull`

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
```

```
using namespace cv;
using namespace std;
```

```
Mat src; Mat src_gray;
int thresh = 100;
int max_thresh = 255;
RNG rng(12345);
```

```
/// Function header
```

```

void thresh_callback(int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load source image and convert it to gray
    src = imread( argv[1], 1 );

    /// Convert image to gray and blur it
    cvtColor( src, src_gray, CV_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );

    /// Create Window
    char* source_window = "Source";
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, src );

    createTrackbar( " Threshold:", "Source", &thresh, max_thresh, thresh_callback );
    thresh_callback( 0, 0 );

    waitKey(0);
    return(0);
}

/** @function thresh_callback */
void thresh_callback(int, void* )
{
    Mat src_copy = src.clone();
    Mat threshold_output;
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    /// Detect edges using Threshold
    threshold( src_gray, threshold_output, thresh, 255, THRESH_BINARY );

    /// Find contours
    findContours( threshold_output, contours, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );

    /// Find the convex hull object for each contour
    vector<vector<Point> > hull( contours.size() );
    for( int i = 0; i < contours.size(); i++ )
        { convexHull( Mat(contours[i]), hull[i], false ); }

    /// Draw contours + hull results
    Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
    for( int i = 0; i < contours.size(); i++ )
        {
            Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
            drawContours( drawing, contours, i, color, 1, 8, vector<Vec4i>(), 0, Point() );
            drawContours( drawing, hull, i, color, 1, 8, vector<Vec4i>(), 0, Point() );
        }

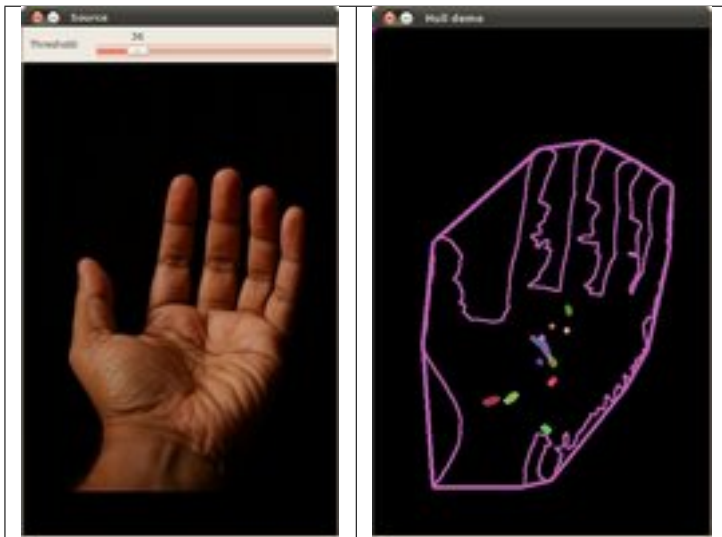
    /// Show in a window
    namedWindow( "Hull demo", CV_WINDOW_AUTOSIZE );
    imshow( "Hull demo", drawing );
}

```

## Explanation

## Result

1. Here it is:



## 3.22 Creating Bounding boxes and circles for contours

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `boundingRect`
- Use the OpenCV function `minEnclosingCircle`

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
```

```
using namespace cv;
using namespace std;
```

```
Mat src; Mat src_gray;
int thresh = 100;
int max_thresh = 255;
RNG rng(12345);
```

```
/// Function header
void thresh_callback(int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load source image and convert it to gray
    src = imread( argv[1], 1 );

    /// Convert image to gray and blur it
    cvtColor( src, src_gray, CV_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );

    /// Create Window
    char* source_window = "Source";
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, src );

    createTrackbar( " Threshold:", "Source", &thresh, max_thresh, thresh_callback );
    thresh_callback( 0, 0 );

    waitKey(0);
    return(0);
}

/** @function thresh_callback */
void thresh_callback(int, void* )
{
    Mat threshold_output;
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    /// Detect edges using Threshold
    threshold( src_gray, threshold_output, thresh, 255, THRESH_BINARY );
    /// Find contours
    findContours( threshold_output, contours, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );

    /// Approximate contours to polygons + get bounding rects and circles
    vector<vector<Point> > contours_poly( contours.size() );
    vector<Rect> boundRect( contours.size() );
    vector<Point2f>center( contours.size() );
    vector<float>radius( contours.size() );

    for( int i = 0; i < contours.size(); i++ )
        { approxPolyDP( Mat(contours[i]), contours_poly[i], 3, true );
          boundRect[i] = boundingRect( Mat(contours_poly[i]) );
          minEnclosingCircle( contours_poly[i], center[i], radius[i] );
        }

    /// Draw polygonal contour + bonding rects + circles
    Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
    for( int i = 0; i < contours.size(); i++ )
        {
            Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
            drawContours( drawing, contours_poly, i, color, 1, 8, vector<Vec4i>(), 0, Point() );
            rectangle( drawing, boundRect[i].tl(), boundRect[i].br(), color, 2, 8, 0 );
            circle( drawing, center[i], (int)radius[i], color, 2, 8, 0 );
        }
}
```

```

    }

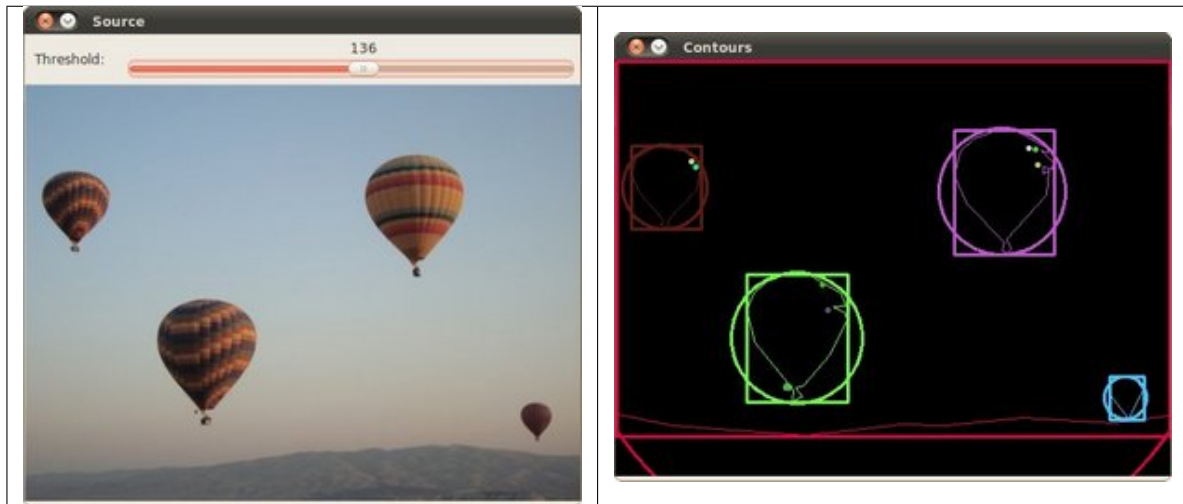
    /// Show in a window
    namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
    imshow( "Contours", drawing );
}

```

## Explanation

## Result

1. Here it is:



## 3.23 Creating Bounding rotated boxes and ellipses for contours

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `minAreaRect`
- Use the OpenCV function `fitEllipse`

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

```

```
using namespace cv;
using namespace std;

Mat src; Mat src_gray;
int thresh = 100;
int max_thresh = 255;
RNG rng(12345);

// Function header
void thresh_callback(int, void* );

/** @function main */
int main( int argc, char** argv )
{
    // Load source image and convert it to gray
    src = imread( argv[1], 1 );

    // Convert image to gray and blur it
    cvtColor( src, src_gray, CV_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );

    // Create Window
    char* source_window = "Source";
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, src );

    createTrackbar( " Threshold:", "Source", &thresh, max_thresh, thresh_callback );
    thresh_callback( 0, 0 );

    waitKey(0);
    return(0);
}

/** @function thresh_callback */
void thresh_callback(int, void* )
{
    Mat threshold_output;
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    // Detect edges using Threshold
    threshold( src_gray, threshold_output, thresh, 255, THRESH_BINARY );
    // Find contours
    findContours( threshold_output, contours, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );

    // Find the rotated rectangles and ellipses for each contour
    vector<RotatedRect> minRect( contours.size() );
    vector<RotatedRect> minEllipse( contours.size() );

    for( int i = 0; i < contours.size(); i++ )
        { minRect[i] = minAreaRect( Mat(contours[i]) );
          if( contours[i].size() > 5 )
              { minEllipse[i] = fitEllipse( Mat(contours[i]) ); }
        }

    // Draw contours + rotated rects + ellipses
    Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
    for( int i = 0; i < contours.size(); i++ )
```

```

{
  Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
  // contour
  drawContours( drawing, contours, i, color, 1, 8, vector<Vec4i>(), 0, Point() );
  // ellipse
  ellipse( drawing, minEllipse[i], color, 2, 8 );
  // rotated rectangle
  Point2f rect_points[4]; minRect[i].points( rect_points );
  for( int j = 0; j < 4; j++ )
    line( drawing, rect_points[j], rect_points[(j+1)%4], color, 1, 8 );
}

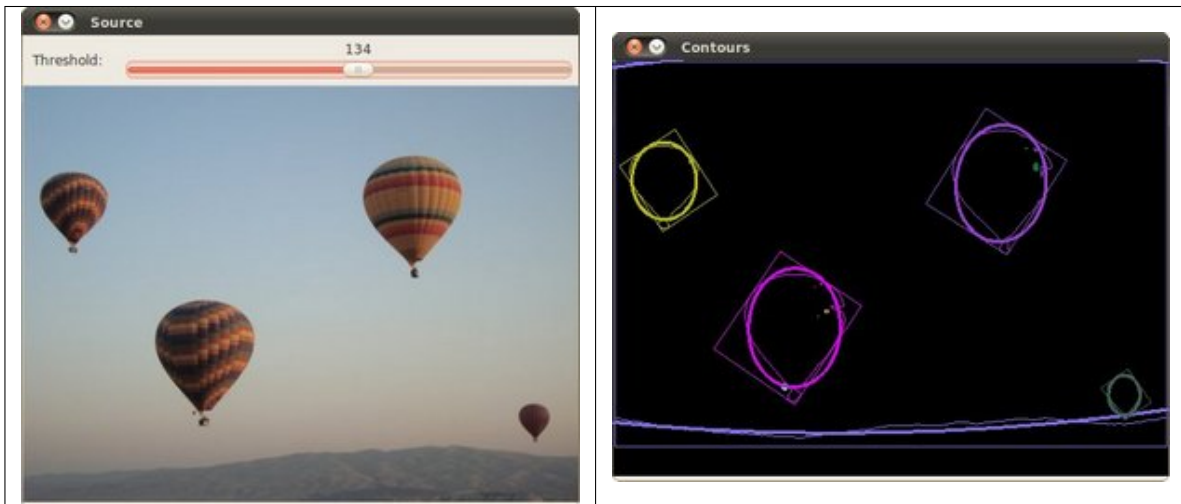
/// Show in a window
namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
imshow( "Contours", drawing );
}

```

## Explanation

## Result

- Here it is:



## 3.24 Image Moments

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `moments`
- Use the OpenCV function `contourArea`
- Use the OpenCV function `arcLength`



## Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

Mat src; Mat src_gray;
int thresh = 100;
int max_thresh = 255;
RNG rng(12345);

/// Function header
void thresh_callback(int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load source image and convert it to gray
    src = imread( argv[1], 1 );

    /// Convert image to gray and blur it
    cvtColor( src, src_gray, CV_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );

    /// Create Window
    char* source_window = "Source";
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, src );

    createTrackbar( " Canny thresh:", "Source", &thresh, max_thresh, thresh_callback );
    thresh_callback( 0, 0 );

    waitKey(0);
    return(0);
}

/** @function thresh_callback */
void thresh_callback(int, void* )
{
    Mat canny_output;
    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;

    /// Detect edges using canny
    Canny( src_gray, canny_output, thresh, thresh*2, 3 );
    /// Find contours
    findContours( canny_output, contours, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );

    /// Get the moments
```

```

vector<Moments> mu(contours.size() );
for( int i = 0; i < contours.size(); i++ )
    { mu[i] = moments( contours[i], false ); }

/// Get the mass centers:
vector<Point2f> mc( contours.size() );
for( int i = 0; i < contours.size(); i++ )
    { mc[i] = Point2f( mu[i].m10/mu[i].m00 , mu[i].m01/mu[i].m00 ); }

/// Draw contours
Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
for( int i = 0; i < contours.size(); i++ )
    {
    Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
    drawContours( drawing, contours, i, color, 2, 8, hierarchy, 0, Point() );
    circle( drawing, mc[i], 4, color, -1, 8, 0 );
    }

/// Show in a window
namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
imshow( "Contours", drawing );

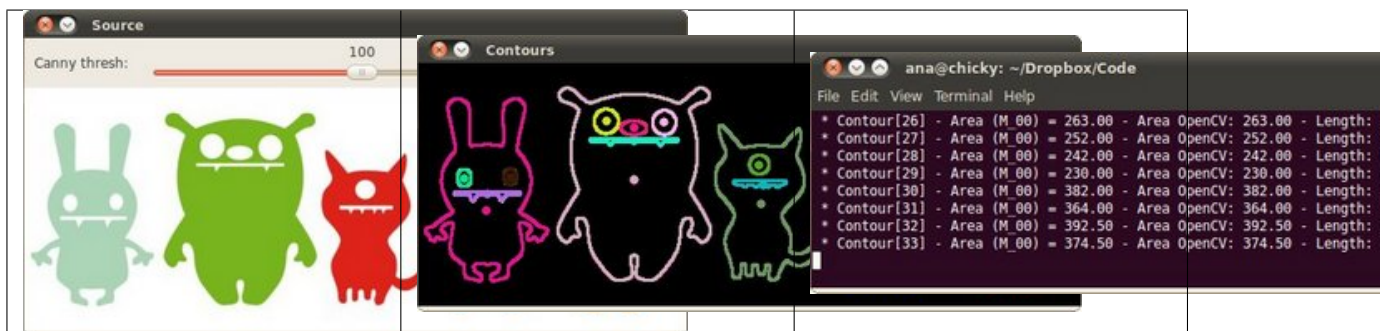
/// Calculate the area with the moments  $M_{00}$  and compare with the result of the OpenCV function
printf("\t Info: Area and Contour Length \n");
for( int i = 0; i < contours.size(); i++ )
    {
    printf(" * Contour[%d] - Area (M_00) = %.2f - Area OpenCV: %.2f - Length: %.2f \n", i, mu[i].m00, contourArea(i),
    Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
    drawContours( drawing, contours, i, color, 2, 8, hierarchy, 0, Point() );
    circle( drawing, mc[i], 4, color, -1, 8, 0 );
    }
}

```

## Explanation

## Result

- Here it is:



## 3.25 Point Polygon Test

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `pointPolygonTest`

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

/** @function main */
int main( int argc, char** argv )
{
    /// Create an image
    const int r = 100;
    Mat src = Mat::zeros( Size( 4*r, 4*r ), CV_8UC1 );

    /// Create a sequence of points to make a contour:
    vector<Point2f> vert(6);

    vert[0] = Point( 1.5*r, 1.34*r );
    vert[1] = Point( 1*r, 2*r );
    vert[2] = Point( 1.5*r, 2.866*r );
    vert[3] = Point( 2.5*r, 2.866*r );
    vert[4] = Point( 3*r, 2*r );
    vert[5] = Point( 2.5*r, 1.34*r );

    /// Draw it in src
    for( int j = 0; j < 6; j++ )
        { line( src, vert[j], vert[(j+1)%6], Scalar( 255 ), 3, 8 ); }

    /// Get the contours
    vector<vector<Point> > contours; vector<Vec4i> hierarchy;
    Mat src_copy = src.clone();

    findContours( src_copy, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE);

    /// Calculate the distances to the contour
    Mat raw_dist( src.size(), CV_32FC1 );

    for( int j = 0; j < src.rows; j++ )
        { for( int i = 0; i < src.cols; i++ )
            { raw_dist.at<float>(j,i) = pointPolygonTest( contours[0], Point2f(i,j), true ); }
        }
```

```

    }

    double minVal; double maxVal;
    minMaxLoc( raw_dist, &minVal, &maxVal, 0, 0, Mat() );
    minVal = abs(minVal); maxVal = abs(maxVal);

    /// Depicting the distances graphically
    Mat drawing = Mat::zeros( src.size(), CV_8UC3 );

    for( int j = 0; j < src.rows; j++ )
        { for( int i = 0; i < src.cols; i++ )
            {
                if( raw_dist.at<float>(j,i) < 0 )
                    { drawing.at<Vec3b>(j,i)[0] = 255 - (int) abs(raw_dist.at<float>(j,i))*255/minVal; }
                else if( raw_dist.at<float>(j,i) > 0 )
                    { drawing.at<Vec3b>(j,i)[2] = 255 - (int) raw_dist.at<float>(j,i)*255/maxVal; }
                else
                    { drawing.at<Vec3b>(j,i)[0] = 255; drawing.at<Vec3b>(j,i)[1] = 255; drawing.at<Vec3b>(j,i)[2] = 255; }
            }
        }

    /// Create Window and show your results
    char* source_window = "Source";
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, src );
    namedWindow( "Distance", CV_WINDOW_AUTOSIZE );
    imshow( "Distance", drawing );

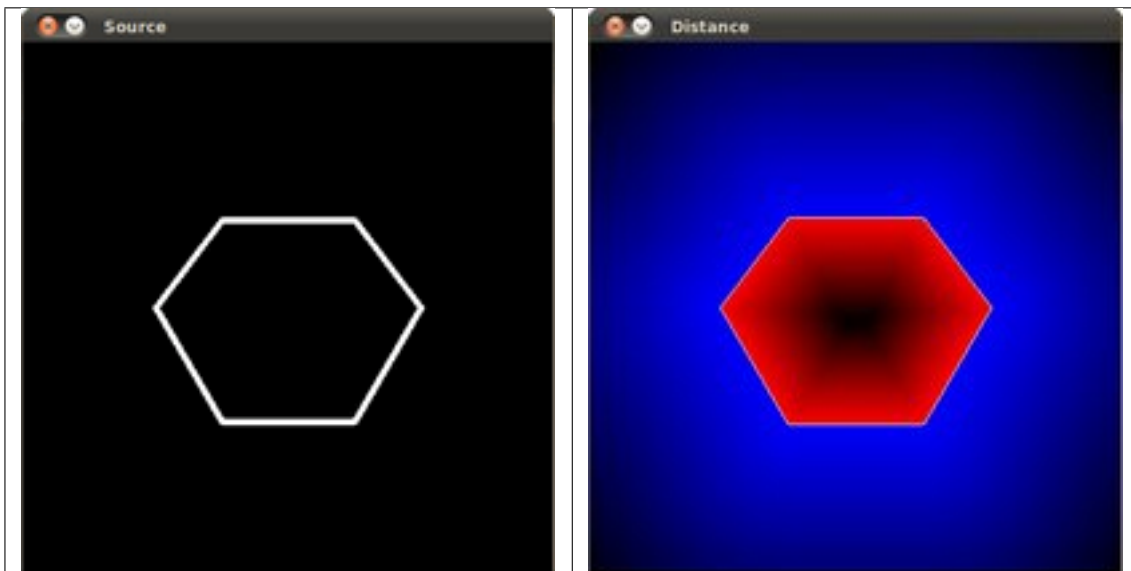
    waitKey(0);
    return(0);
}

```

## Explanation

## Result

1. Here it is:





# ***HIGHGUI* MODULE. HIGH LEVEL GUI AND MEDIA**

This section contains valuable tutorials about how to read/save your image/video files and how to use the built-in graphical user interface of the library.



*Title: Adding a Trackbar to our applications!*

*Compatibility: > OpenCV 2.0*

We will learn how to add a Trackbar to our applications

---

## 4.1 Adding a Trackbar to our applications!

- In the previous tutorials (about *linear blending* and the *brightness and contrast adjustments*) you might have noted that we needed to give some **input** to our programs, such as  $\alpha$  and  $\beta$ . We accomplished that by entering this data using the Terminal
- Well, it is time to use some fancy GUI tools. OpenCV provides some GUI utilities (*highgui.h*) for you. An example of this is a **Trackbar**



- In this tutorial we will just modify our two previous programs so that they get the input information from the trackbar.

### Goals

In this tutorial you will learn how to:

- Add a Trackbar in an OpenCV window by using `createTrackbar`

### Code

Let's modify the program made in the tutorial *Adding (blending) two images using OpenCV*. We will let the user enter the  $\alpha$  value by using the Trackbar.

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

/// Global Variables
const int alpha_slider_max = 100;
int alpha_slider;
double alpha;
double beta;

/// Matrices to store images
Mat src1;
Mat src2;
Mat dst;

/**
 * @function on_trackbar
 * @brief Callback for trackbar
 */
void on_trackbar( int, void* )
{
    alpha = (double) alpha_slider/alpha_slider_max ;
    beta = ( 1.0 - alpha );

    addWeighted( src1, alpha, src2, beta, 0.0, dst);

    imshow( "Linear Blend", dst );
}
```

```

}

int main( int argc, char** argv )
{
    // Read image ( same size, same type )
    src1 = imread("../images/LinuxLogo.jpg");
    src2 = imread("../images/WindowsLogo.jpg");

    if( !src1.data ) { printf("Error loading src1 \n"); return -1; }
    if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

    // Initialize values
    alpha_slider = 0;

    // Create Windows
    namedWindow("Linear Blend", 1);

    // Create Trackbars
    char TrackbarName[50];
    sprintf( TrackbarName, "Alpha x %d", alpha_slider_max );

    createTrackbar( TrackbarName, "Linear Blend", &alpha_slider, alpha_slider_max, on_trackbar );

    // Show some stuff
    on_trackbar( alpha_slider, 0 );

    // Wait until user press some key
    waitKey(0);
    return 0;
}

```

## Explanation

We only analyze the code that is related to Trackbar:

1. First, we load 02 images, which are going to be blended.

```

src1 = imread("../images/LinuxLogo.jpg");
src2 = imread("../images/WindowsLogo.jpg");

```

2. To create a trackbar, first we have to create the window in which it is going to be located. So:

```

namedWindow("Linear Blend", 1);

```

3. Now we can create the Trackbar:

```

createTrackbar( TrackbarName, "Linear Blend", &alpha_slider, alpha_slider_max, on_trackbar );

```

Note the following:

- Our Trackbar has a label **TrackbarName**
- The Trackbar is located in the window named **“Linear Blend”**
- The Trackbar values will be in the range from 0 to **alpha\_slider\_max** (the minimum limit is always **zero**).
- The numerical value of Trackbar is stored in **alpha\_slider**
- Whenever the user moves the Trackbar, the callback function **on\_trackbar** is called



#### 4. Finally, we have to define the callback function `on_trackbar`

```
void on_trackbar( int, void* )
{
    alpha = (double) alpha_slider/alpha_slider_max ;
    beta = ( 1.0 - alpha );

    addWeighted( src1, alpha, src2, beta, 0.0, dst);

    imshow( "Linear Blend", dst );
}
```

Note that:

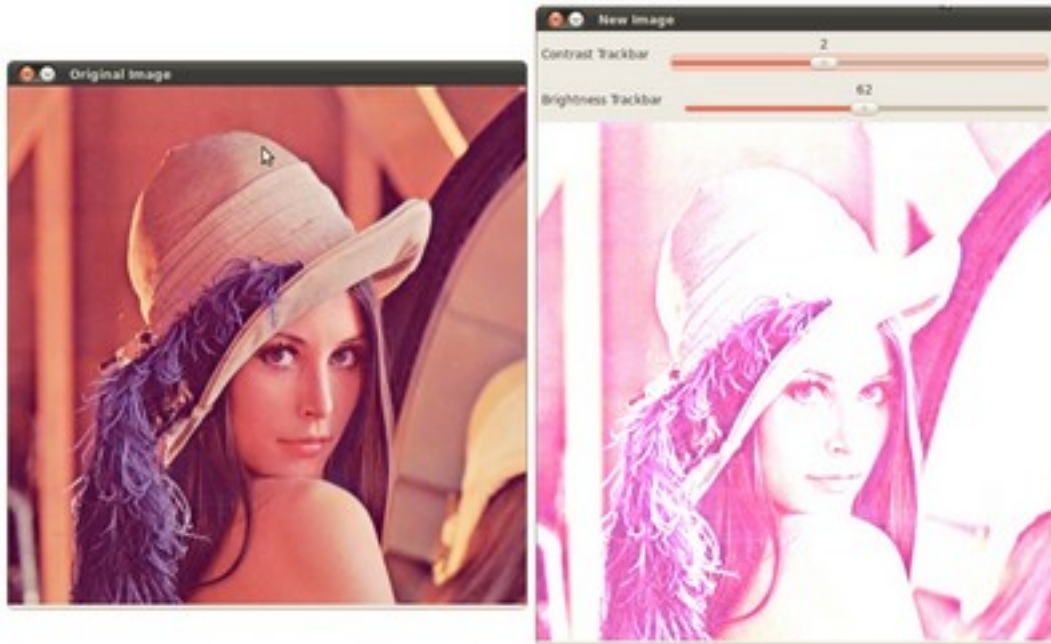
- We use the value of `alpha_slider` (integer) to get a double value for `alpha`.
- `alpha_slider` is updated each time the trackbar is displaced by the user.
- We define `src1`, `src2`, `dst`, `alpha`, `alpha_slider` and `beta` as global variables, so they can be used everywhere.

## Result

- Our program produces the following output:



- As a manner of practice, you can also add 02 trackbars for the program made in *Changing the contrast and brightness of an image!*. One trackbar to set  $\alpha$  and another for  $\beta$ . The output might look like:





# ***CALIB3D* MODULE. CAMERA CALIBRATION AND 3D RECONSTRUCTION**

Although we got most of our images in a 2D format they do come from a 3D world. Here you will learn how to find out from the 2D images information about the 3D world.

---

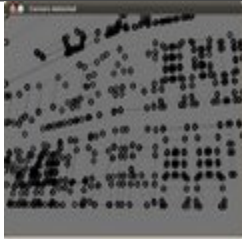
**Note:** Unfortunately we have no tutorials into this section. Nevertheless, our tutorial writing team is working on it. If you have a tutorial suggestion or you have written yourself a tutorial (or coded a sample code) that you would like to see here please contact us via our [user group](#).


---




# FEATURE2D MODULE. 2D FEATURES FRAMEWORK

Learn about how to use the feature points detectors, descriptors and matching framework found inside OpenCV.

- 

**Title:** *Harris corner detector*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Why is it a good idea to track corners? We learn to use the Harris method to detect corners
  - 

**Title:** *Shi-Tomasi corner detector*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Where we use an improved method to detect corners more accurately!
  - 

**Title:** *Creating your own corner detector*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Here you will learn how to use OpenCV functions to make your personalized corner detector!
- 
- ```
** Number of corners detected: 4
-- Refined Corner(0) : (108.906, 59.0857)
-- Refined Corner(1) : (107, 73)
-- Refined Corner(2) : (70.7671, 72.7351)
-- Refined Corner(3) : (27.4633, 60.7455)
```
- Title:** *Detecting corners location in subpixels*  
**Compatibility:** > OpenCV 2.0  
**Author:** Ana Huamán  
Is pixel resolution enough? Here we learn a simple method to improve our accuracy.
-

## 6.1 Harris corner detector

### Goal

In this tutorial you will learn how to:

- Use the function `cornerHarris` to detect corners using the Harris-Stephens method.

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

// Global variables
Mat src, src_gray;
int thresh = 200;
int max_thresh = 255;

char* source_window = "Source image";
char* corners_window = "Corners detected";

// Function header
void cornerHarris_demo( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    // Load source image and convert it to gray
    src = imread( argv[1], 1 );
    cvtColor( src, src_gray, CV_BGR2GRAY );

    // Create a window and a trackbar
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    createTrackbar( "Threshold: ", source_window, &thresh, max_thresh, cornerHarris_demo );
    imshow( source_window, src );

    cornerHarris_demo( 0, 0 );

    waitKey(0);
    return(0);
}

/** @function cornerHarris_demo */
void cornerHarris_demo( int, void* )
{
```

```
Mat dst, dst_norm, dst_norm_scaled;
dst = Mat::zeros( src.size(), CV_32FC1 );

/// Detector parameters
int blockSize = 2;
int apertureSize = 3;
double k = 0.04;

/// Detecting corners
cornerHarris( src_gray, dst, blockSize, apertureSize, k, BORDER_DEFAULT );

/// Normalizing
normalize( dst, dst_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat() );
convertScaleAbs( dst_norm, dst_norm_scaled );

/// Drawing a circle around corners
for( int j = 0; j < dst_norm.rows ; j++ )
    { for( int i = 0; i < dst_norm.cols; i++ )
        {
            if( (int) dst_norm.at<float>(j,i) > thresh )
                {
                    circle( dst_norm_scaled, Point( i, j ), 5, Scalar(0), 2, 8, 0 );
                }
        }
    }

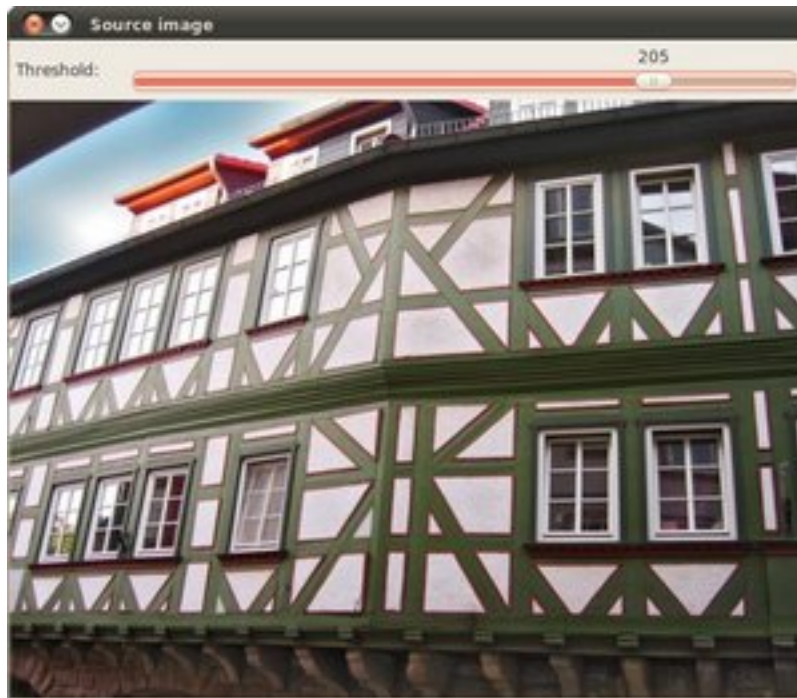
/// Showing the result
namedWindow( corners_window, CV_WINDOW_AUTOSIZE );
imshow( corners_window, dst_norm_scaled );
}
```

## Explanation

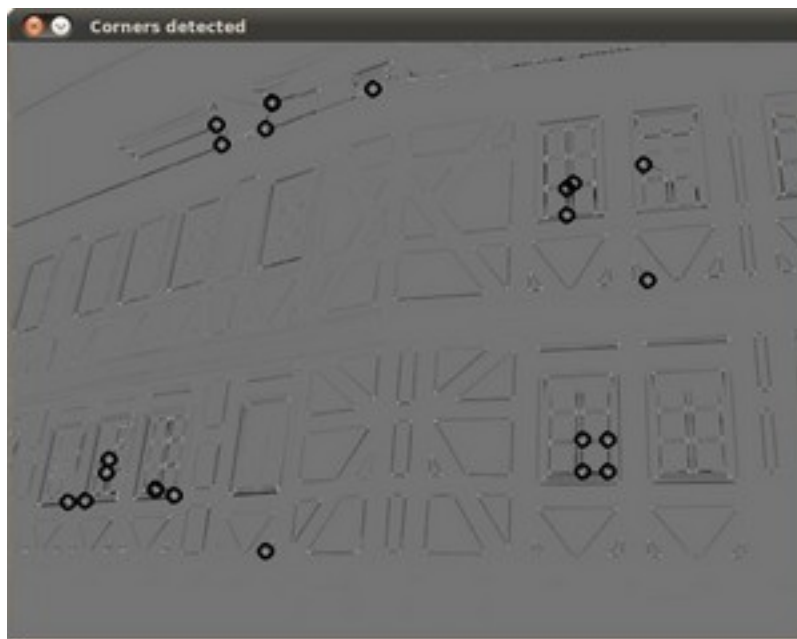
## Result

The original image:





The detected corners are surrounded by a small black circle



## 6.2 Shi-Tomasi corner detector

### Goal

In this tutorial you will learn how to:

- Use the function `goodFeaturesToTrack` to detect corners using the Shi-Tomasi method.

## Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

/// Global variables
Mat src, src_gray;

int maxCorners = 23;
int maxTrackbar = 100;

RNG rng(12345);
char* source_window = "Image";

/// Function header
void goodFeaturesToTrack_Demo( int, void* );

/**
 * @function main
 */
int main( int argc, char** argv )
{
    /// Load source image and convert it to gray
    src = imread( argv[1], 1 );
    cvtColor( src, src_gray, CV_BGR2GRAY );

    /// Create Window
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );

    /// Create Trackbar to set the number of corners
    createTrackbar( "Max corners:", source_window, &maxCorners, maxTrackbar, goodFeaturesToTrack_Demo );

    imshow( source_window, src );

    goodFeaturesToTrack_Demo( 0, 0 );

    waitKey(0);
    return(0);
}

/**
 * @function goodFeaturesToTrack_Demo.cpp
 * @brief Apply Shi-Tomasi corner detector
 */
void goodFeaturesToTrack_Demo( int, void* )
{
    if( maxCorners < 1 ) { maxCorners = 1; }
```

```
/// Parameters for Shi-Tomasi algorithm
vector<Point2f> corners;
double qualityLevel = 0.01;
double minDistance = 10;
int blockSize = 3;
bool useHarrisDetector = false;
double k = 0.04;

/// Copy the source image
Mat copy;
copy = src.clone();

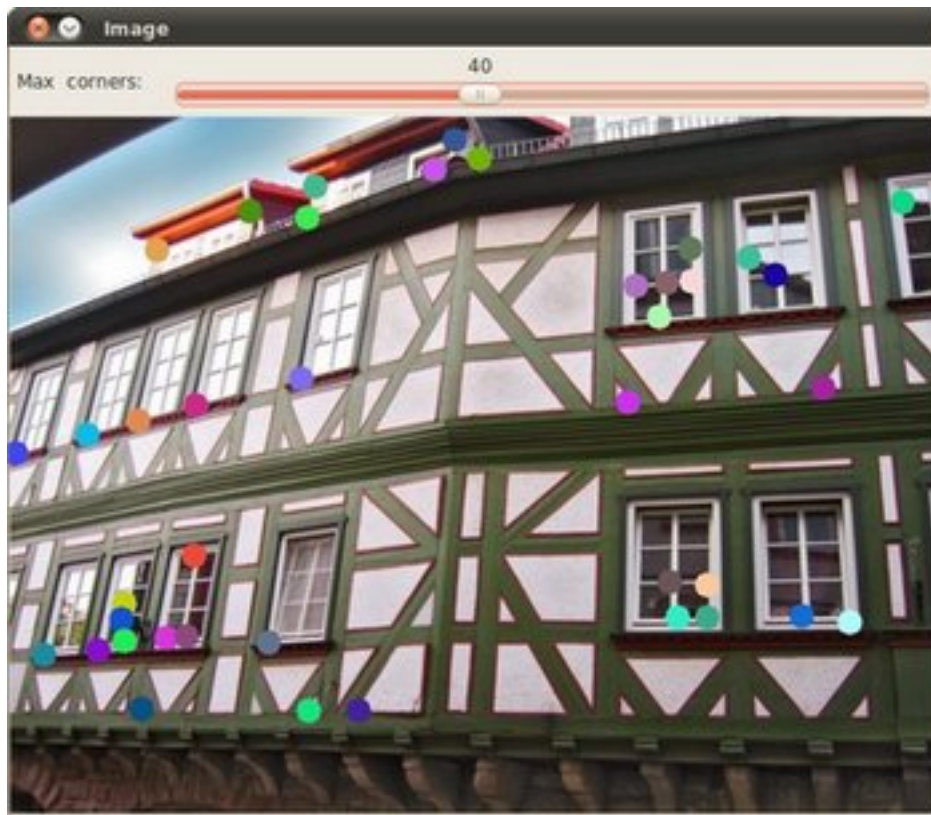
/// Apply corner detection
goodFeaturesToTrack( src_gray,
                    corners,
                    maxCorners,
                    qualityLevel,
                    minDistance,
                    Mat(),
                    blockSize,
                    useHarrisDetector,
                    k );

/// Draw corners detected
cout<<"** Number of corners detected: "<<corners.size()<<endl;
int r = 4;
for( int i = 0; i < corners.size(); i++ )
    { circle( copy, corners[i], r, Scalar(rng.uniform(0,255), rng.uniform(0,255),
      rng.uniform(0,255)), -1, 8, 0 ); }

/// Show what you got
namedWindow( source_window, CV_WINDOW_AUTOSIZE );
imshow( source_window, copy );
}
```

## Explanation

## Result



## 6.3 Creating your own corner detector

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `cornerEigenValsAndVecs` to find the eigenvalues and eigenvectors to determine if a pixel is a corner.
- Use the OpenCV function `cornerMinEigenVal` to find the minimum eigenvalues for corner detection.
- To implement our own version of the Harris detector as well as the Shi-Tomasi detector, by using the two functions above.

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

/// Global variables
Mat src, src_gray;
Mat myHarris_dst; Mat myHarris_copy; Mat Mc;
Mat myShiTomasi_dst; Mat myShiTomasi_copy;

int myShiTomasi_qualityLevel = 50;
int myHarris_qualityLevel = 50;
int max_qualityLevel = 100;

double myHarris_minVal; double myHarris_maxVal;
double myShiTomasi_minVal; double myShiTomasi_maxVal;

RNG rng(12345);

char* myHarris_window = "My Harris corner detector";
char* myShiTomasi_window = "My Shi Tomasi corner detector";

/// Function headers
void myShiTomasi_function( int, void* );
void myHarris_function( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load source image and convert it to gray
    src = imread( argv[1], 1 );
    cvtColor( src, src_gray, CV_BGR2GRAY );

    /// Set some parameters
    int blockSize = 3; int apertureSize = 3;

    /// My Harris matrix -- Using cornerEigenValsAndVecs
    myHarris_dst = Mat::zeros( src_gray.size(), CV_32FC(6) );
    Mc = Mat::zeros( src_gray.size(), CV_32FC1 );

    cornerEigenValsAndVecs( src_gray, myHarris_dst, blockSize, apertureSize, BORDER_DEFAULT );

    /* calculate Mc */
    for( int j = 0; j < src_gray.rows; j++ )
        { for( int i = 0; i < src_gray.cols; i++ )
            {
                float lambda_1 = myHarris_dst.at<float>( j, i, 0 );
                float lambda_2 = myHarris_dst.at<float>( j, i, 1 );
                Mc.at<float>(j,i) = lambda_1*lambda_2 - 0.04*pow( ( lambda_1 + lambda_2 ), 2 );
            }
        }

    minMaxLoc( Mc, &myHarris_minVal, &myHarris_maxVal, 0, 0, Mat() );

```

```

/* Create Window and Trackbar */
namedWindow( myHarris_window, CV_WINDOW_AUTOSIZE );
createTrackbar( " Quality Level:", myHarris_window, &myHarris_qualityLevel, max_qualityLevel,
               myHarris_function );
myHarris_function( 0, 0 );

/// My Shi-Tomasi -- Using cornerMinEigenVal
myShiTomasi_dst = Mat::zeros( src_gray.size(), CV_32FC1 );
cornerMinEigenVal( src_gray, myShiTomasi_dst, blockSize, apertureSize, BORDER_DEFAULT );

minMaxLoc( myShiTomasi_dst, &myShiTomasi_minVal, &myShiTomasi_maxVal, 0, 0, Mat() );

/* Create Window and Trackbar */
namedWindow( myShiTomasi_window, CV_WINDOW_AUTOSIZE );
createTrackbar( " Quality Level:", myShiTomasi_window, &myShiTomasi_qualityLevel, max_qualityLevel,
               myShiTomasi_function );
myShiTomasi_function( 0, 0 );

waitKey(0);
return(0);
}

/** @function myShiTomasi_function */
void myShiTomasi_function( int, void* )
{
    myShiTomasi_copy = src.clone();

    if( myShiTomasi_qualityLevel < 1 ) { myShiTomasi_qualityLevel = 1; }

    for( int j = 0; j < src_gray.rows; j++ )
        { for( int i = 0; i < src_gray.cols; i++ )
            {
                if( myShiTomasi_dst.at<float>(j,i) > myShiTomasi_minVal + ( myShiTomasi_maxVal -
                    myShiTomasi_minVal ) * myShiTomasi_qualityLevel / max_qualityLevel )
                    { circle( myShiTomasi_copy, Point(i,j), 4, Scalar( rng.uniform(0,255),
                        rng.uniform(0,255), rng.uniform(0,255) ), -1, 8, 0 ); }
            }
        }
    imshow( myShiTomasi_window, myShiTomasi_copy );
}

/** @function myHarris_function */
void myHarris_function( int, void* )
{
    myHarris_copy = src.clone();

    if( myHarris_qualityLevel < 1 ) { myHarris_qualityLevel = 1; }

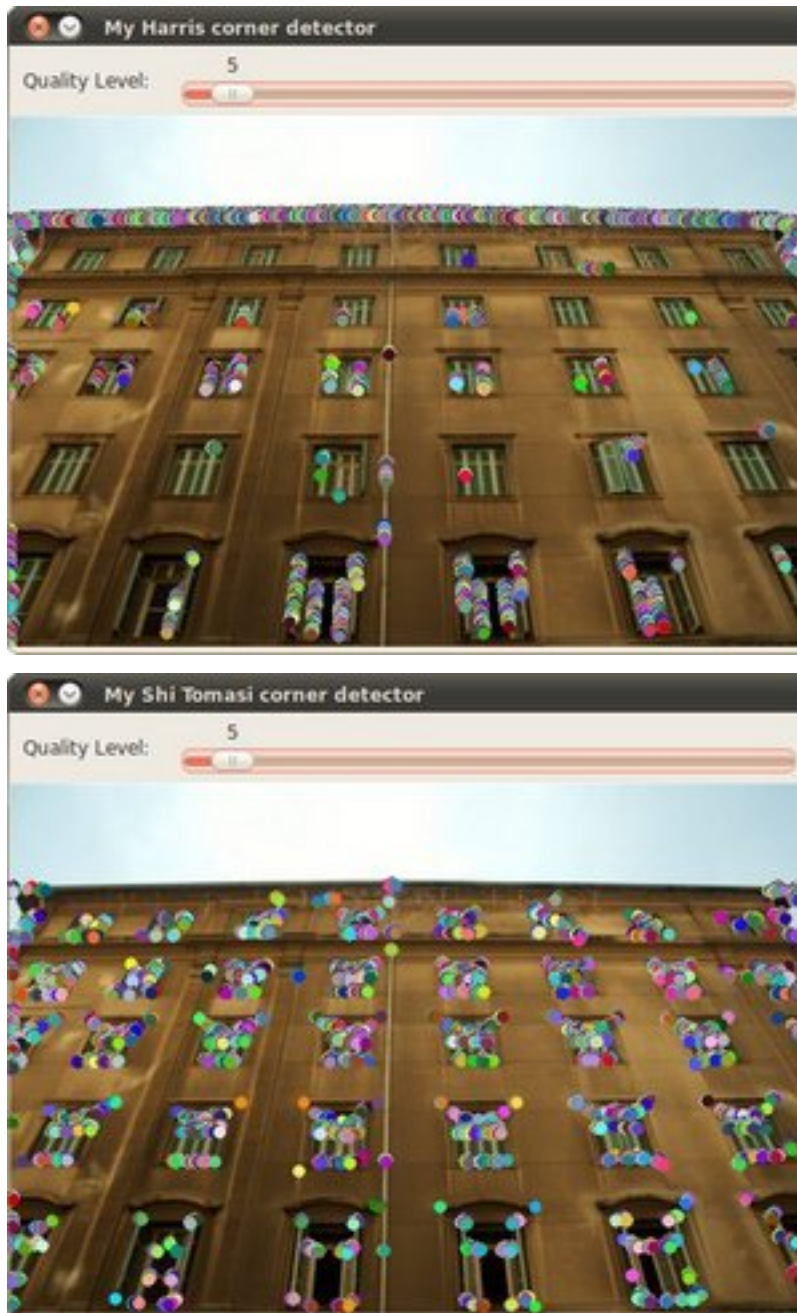
    for( int j = 0; j < src_gray.rows; j++ )
        { for( int i = 0; i < src_gray.cols; i++ )
            {
                if( Mc.at<float>(j,i) > myHarris_minVal + ( myHarris_maxVal - myHarris_minVal )
                    * myHarris_qualityLevel / max_qualityLevel )
                    { circle( myHarris_copy, Point(i,j), 4, Scalar( rng.uniform(0,255), rng.uniform(0,255),
                        rng.uniform(0,255) ), -1, 8, 0 ); }
            }
        }
    imshow( myHarris_window, myHarris_copy );
}

```

}

## Explanation

## Result





## 6.4 Detecting corners location in subpixels

### Goal

In this tutorial you will learn how to:

- Use the OpenCV function `cornerSubPix` to find more exact corner positions (more exact than integer pixels).

### Theory

### Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

/// Global variables
Mat src, src_gray;

int maxCorners = 10;
int maxTrackbar = 25;

RNG rng(12345);
char* source_window = "Image";

/// Function header
void goodFeaturesToTrack_Demo( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load source image and convert it to gray
    src = imread( argv[1], 1 );
    cvtColor( src, src_gray, CV_BGR2GRAY );

    /// Create Window
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );

    /// Create Trackbar to set the number of corners
    createTrackbar( "Max corners:", source_window, &maxCorners, maxTrackbar, goodFeaturesToTrack_Demo);

    imshow( source_window, src );

    goodFeaturesToTrack_Demo( 0, 0 );

    waitKey(0);
    return(0);
}
```



```
/**
 * @function goodFeaturesToTrack_Demo.cpp
 * @brief Apply Shi-Tomasi corner detector
 */
void goodFeaturesToTrack_Demo( int, void* )
{
    if( maxCorners < 1 ) { maxCorners = 1; }

    /// Parameters for Shi-Tomasi algorithm
    vector<Point2f> corners;
    double qualityLevel = 0.01;
    double minDistance = 10;
    int blockSize = 3;
    bool useHarrisDetector = false;
    double k = 0.04;

    /// Copy the source image
    Mat copy;
    copy = src.clone();

    /// Apply corner detection
    goodFeaturesToTrack( src_gray,
                        corners,
                        maxCorners,
                        qualityLevel,
                        minDistance,
                        Mat(),
                        blockSize,
                        useHarrisDetector,
                        k );

    /// Draw corners detected
    cout<<"** Number of corners detected: "<<corners.size()<<endl;
    int r = 4;
    for( int i = 0; i < corners.size(); i++ )
        { circle( copy, corners[i], r, Scalar(rng.uniform(0,255), rng.uniform(0,255),
   rng.uniform(0,255)), -1, 8, 0 ); }

    /// Show what you got
    namedWindow( source_window, CV_WINDOW_AUTOSIZE );
    imshow( source_window, copy );

    /// Set the needed parameters to find the refined corners
    Size winSize = Size( 5, 5 );
    Size zeroZone = Size( -1, -1 );
    TermCriteria criteria = TermCriteria( CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 40, 0.001 );

    /// Calculate the refined corner locations
    cornerSubPix( src_gray, corners, winSize, zeroZone, criteria );

    /// Write them down
    for( int i = 0; i < corners.size(); i++ )
        { cout<<" -- Refined Corner ["<<i<<" ("<<corners[i].x<<","<<corners[i].y<<)"<<endl; }
}
```

## Explanation

## Result



Here is the result:

```
ana@chicky: ~/Dropbox/Code
File Edit View Terminal Help
** Number of corners detected: 4
-- Refined Corner [0] (108.906,59.0857)
-- Refined Corner [1] (107,73)
-- Refined Corner [2] (70.7671,72.7351)
-- Refined Corner [3] (27.4633,60.7455)
ana@chicky:~/Dropbox/Code$
```



## ***VIDEO* MODULE. VIDEO ANALYSIS**

Look here in order to find use on your video stream algorithms like: motion extraction, feature tracking and foreground extractions.

---

**Note:** Unfortunately we have no tutorials into this section. Nevertheless, our tutorial writing team is working on it. If you have a tutorial suggestion or you have written yourself a tutorial (or coded a sample code) that you would like to see here please contact us via our [user group](#).

---



# *OBJDETECT* MODULE. OBJECT DETECTION

Ever wondered how your digital camera detects peoples and faces? Look here to find out!

---

**Note:** Unfortunately we have no tutorials into this section. Nevertheless, our tutorial writing team is working on it. If you have a tutorial suggestion or you have written yourself a tutorial (or coded a sample code) that you would like to see here please contact us via our [user group](#).

---



# *ML* MODULE. MACHINE LEARNING

Use the powerfull machine learning classes for statistical classification, regression and clustering of data.

---

**Note:** Unfortunately we have no tutorials into this section. Nevertheless, our tutorial writing team is working on it. If you have a tutorial suggestion or you have writen yourself a tutorial (or coded a sample code) that you would like to see here please contact us via our [user group](#).

---





# *GPU* MODULE. GPU-ACCELERATED COMPUTER VISION

Squeeze out every little computation power from your system by using the power of your video card to run the OpenCV algorithms.

---

**Note:** Unfortunately we have no tutorials into this section. Nevertheless, our tutorial writing team is working on it. If you have a tutorial suggestion or you have written yourself a tutorial (or coded a sample code) that you would like to see here please contact us via our [user group](#).

---



# GENERAL TUTORIALS

These tutorials are the bottom of the iceberg as they link together multiple of the modules presented above in order to solve complex problems.

---

**Note:** Unfortunately we have no tutorials into this section. Nevertheless, our tutorial writing team is working on it. If you have a tutorial suggestion or you have written yourself a tutorial (or coded a sample code) that you would like to see here please contact us via our [user group](#).

---