The OpenCV Tutorials

Release 2.4.6.0

July 05, 2013

CONTENTS

1 Introduction to OpenCV **5**
1.1 Installation in Linux . . . 9
1.2 Using OpenCV with gcc and CMake . . . 10
1.3 Using OpenCV with Eclipse (plugin CDT) . . . 12
1.4 Installation in Windows . . . 17
1.5 How to build applications with OpenCV inside the *Microsoft Visual Studio* . . . 27
1.6 Image Watch: viewing in-memory images in the Visual Studio debugger . . . 36
1.7 Introduction to Java Development . . . 41
1.8 Introduction into Android Development . . . 57
1.9 OpenCV4Android SDK . . . 85
1.10 Android Development with OpenCV . . . 95
1.11 Installation in iOS . . . 107
1.12 Cross compilation for ARM based Linux systems . . . 107
1.13 Load and Display an Image . . . 109
1.14 Load, Modify, and Save an Image . . . 112
1.15 How to write a tutorial for OpenCV . . . 114

2 *core* module. The Core Functionality **125**
2.1 Mat - The Basic Image Container . . . 127
2.2 How to scan images, lookup tables and time measurement with OpenCV . . . 133
2.3 Mask operations on matrices . . . 138
2.4 Adding (blending) two images using OpenCV . . . 140
2.5 Changing the contrast and brightness of an image! . . . 142
2.6 Basic Drawing . . . 146
2.7 Random generator and text with OpenCV . . . 151
2.8 Discrete Fourier Transform . . . 155
2.9 File Input and Output using XML and YAML files . . . 159
2.10 Interoperability with OpenCV 1 . . . 166

3 *imgproc* module. Image Processing **171**
3.1 Smoothing Images . . . 176
3.2 Eroding and Dilating . . . 181
3.3 More Morphology Transformations . . . 187
3.4 Image Pyramids . . . 193
3.5 Basic Thresholding Operations . . . 199
3.6 Making your own linear filters! . . . 207
3.7 Adding borders to your images . . . 211
3.8 Sobel Derivatives . . . 215
3.9 Laplace Operator . . . 221

3.10 Canny Edge Detector . . . 225
3.11 Hough Line Transform . . . 230
3.12 Hough Circle Transform . . . 236
3.13 Remapping . . . 240
3.14 Affine Transformations . . . 246
3.15 Histogram Equalization . . . 252
3.16 Histogram Calculation . . . 258
3.17 Histogram Comparison . . . 265
3.18 Back Projection . . . 270
3.19 Template Matching . . . 276
3.20 Finding contours in your image . . . 284
3.21 Convex Hull . . . 285
3.22 Creating Bounding boxes and circles for contours . . . 287
3.23 Creating Bounding rotated boxes and ellipses for contours . . . 290
3.24 Image Moments . . . 292
3.25 Point Polygon Test . . . 294

4 *highgui* module. High Level GUI and Media 297
4.1 Adding a Trackbar to our applications! . . . 298
4.2 Video Input with OpenCV and similarity measurement . . . 301
4.3 Creating a video with OpenCV . . . 309

5 *calib3d* module. Camera calibration and 3D reconstruction 315
5.1 Camera calibration with square chessboard . . . 316
5.2 Camera calibration With OpenCV . . . 316

6 *feature2d* module. 2D Features framework 327
6.1 Feature Description . . . 329
6.2 Harris corner detector . . . 331
6.3 Feature Matching with FLANN . . . 335
6.4 Features2D + Homography to find a known object . . . 338
6.5 Shi-Tomasi corner detector . . . 341
6.6 Creating yor own corner detector . . . 343
6.7 Detecting corners location in subpixeles . . . 347
6.8 Feature Detection . . . 350
6.9 Feature Matching with FLANN . . . 352
6.10 Features2D + Homography to find a known object . . . 355
6.11 Detection of planar objects . . . 358

7 *video* module. Video analysis 361

8 *objdetect* module. Object Detection 363
8.1 Cascade Classifier . . . 364

9 *ml* module. Machine Learning 369
9.1 Introduction to Support Vector Machines . . . 370
9.2 Support Vector Machines for Non-Linearly Separable Data . . . 375

10 *gpu* module. GPU-Accelerated Computer Vision 383
10.1 Similarity check (PNSR and SSIM) on the GPU . . . 384

11 *contrib* module. The additional contributions made available ! 393
11.1 Discovering the human retina and its use for image processing . . . 394

12 OpenCV iOS 407

12.1 OpenCV iOS Hello 408
12.2 OpenCV iOS - Image Processing 412
12.3 OpenCV iOS - Video Processing 414

13 General tutorials **421**

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 GIT 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*

Run OpenCV and your vision apps on an iDevice

- *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.

CHAPTER

ONE

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.

Title: *Image Watch: viewing in-memory images in the Visual Studio debugger*
Compatibility: >= OpenCV 2.4
Author: Wolf Kienzle
You will learn how to visualize OpenCV matrices and images within Visual Studio 2012.

- **Desktop Java**

Title: *Introduction to Java Development*
Compatibility: > OpenCV 2.4.4
Authors: Eric Christiansen and Andrey Pavlenko
Explains how to build and run a simple desktop Java application using Eclipse, Ant or the Simple Build Tool (SBT).

- **Android**

Title: *Introduction into Android Development*
Compatibility: > OpenCV 2.4.2
Author: Vsevolod Glumov
Not a tutorial, but a guide introducing Android development basics and environment setup

Title: *OpenCV4Android SDK*
Compatibility: > OpenCV 2.4.2
Author: Vsevolod Glumov
OpenCV4Android SDK: general info, installation, running samples

Title: *Android Development with OpenCV*
Compatibility: > OpenCV 2.4.3
Author: Vsevolod Glumov
Development with OpenCV4Android SDK

- **iOS**

Title: *Installation in iOS*
Compatibility: > OpenCV 2.4.2
Author: Artem Myagkov, Eduard Feicho
We will learn how to setup OpenCV for using it in iOS!

- **Embedded Linux**

Title: *Cross compilation for ARM based Linux systems*
Compatibility: > OpenCV 2.4.4
Author: Alexander Smorkalov
We will learn how to setup OpenCV cross compilation environment for ARM Linux.

- **Common**

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

- **Want to contribute, and see your own work between the OpenCV tutorials?**

Title: *How to write a tutorial for OpenCV*
Compatibility: > OpenCV 1.0
Author: Bernát Gábor
If you already have a good grasp on using OpenCV and have made some projects that would be perfect presenting an OpenCV feature not yet part of these tutorials, here it is what you need to know.

1.1 Installation in Linux

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

Required Packages

- GCC 4.4.x or later. This can be installed with:

  ```
  sudo apt-get install build-essential
  ```

- CMake 2.6 or higher;
- Git;
- GTK+2.x or higher, including headers (libgtk2.0-dev);
- pkgconfig;
- Python 2.6 or later and Numpy 1.5 or later with developer packages (python-dev, python-numpy);
- ffmpeg or libav development packages: libavcodec-dev, libavformat-dev, libswscale-dev;
- [optional] libdc1394 2.x;
- [optional] libjpeg-dev, libpng-dev, libtiff-dev, libjasper-dev.

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 our Git repository.

Getting the Latest Stable OpenCV Version

- Go to our page on Sourceforge;
- Download the source tarball and unpack it.

Getting the Cutting-edge OpenCV from the Git Repository

Launch Git client and clone OpenCV repository

In Linux it can be achieved with the following command in Terminal:

```
cd ~/<my_working _directory>
git clone https://github.com/Itseez/opencv.git
```

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
```

Note: If the size of the created library is a critical issue (like in case of an Android build) you can use the `install/strip` command to get the smallest size as possible. The *stripped* version appears to be twice as small. However, we do not recommend using this unless those extra megabytes do really matter.

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):
  1. No need to change anything when porting between Linux and Windows
  2. 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 <stdio.h>
#include <opencv2/opencv.hpp>

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:

```
cmake_minimum_required(VERSION 2.8)
project( DisplayImage )
find_package( OpenCV REQUIRED )
add_executable( DisplayImage DisplayImage.cpp )
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 a 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)

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 on Windows 7 SP1. Nevertheless, it should also work on any other relatively modern version of Windows OS. If you encounter errors after following the steps described below, feel free to contact us via our OpenCV Q&A forum. We'll do our best to help you out.

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 with the latest Microsoft Visual Studio IDE and doesn't take advantage of the most advanced technologies we integrate into our library.

Installation by Using the Pre-built Libraries

1. Launch a web browser of choice and go to our page on Sourceforge.

2. Choose a build you want to use and download it.
3. Make sure you have admin rights. Unpack the self-extracting archive.
4. You can check the installation at the chosen path as you can see below.

5. 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.

> **Warning:** These videos above are long-obsolete and contain inaccurate information. Be careful, since solutions described in those videos are no longer supported and may even break your install.

If you are building your own libraries you can take the source files from our Git repository.

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

- An IDE of choice (preferably), or just a CC++ compiler that will actually make the binary files. Here we will use the Microsoft Visual Studio. However, you can use any other IDE that has a valid CC++ compiler.
- CMake, which is a neat tool to make the project files (for your choosen 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.
- Git to acquire the OpenCV source files. A good tool for this is TortoiseGit. Alternatively, you can just download an archived version of the source files from our page on Sourceforge

OpenCV may come in multiple flavors. There is a "core" section that will work on its own. Nevertheless, there is a couple of tools, libraries made by 3rd parties that offer services of which the OpenCV may take advantage. These will improve its capabilities in many ways. 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 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, since 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 algorithms (e.g the HOG descriptor). Getting more and more of our algorithms to work 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, We will cover this in depth at the *How to Install Sphinx* section.

Now we 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 this section.

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 OK.
3. Download and install an up-to-date version of msysgit from its official site. There is also the portable version, which you need only to unpack to get access to the console version of Git. Supposing that for some of us it could be quite enough.
4. Install TortoiseGit. Choose the 32 or 64 bit version according to the type of OS you work in. While installing, locate your msysgit (if it doesn't do that automatically). Follow the wizard – the default options are OK for the most part.
5. Choose a directory in your file system, where you will download the OpenCV libraries to. I recommend creating a new one that has short path and no special charachters in it, for example `D:/OpenCV`. For this tutorial I'll suggest you do so. If you use your own path and know, what you're doing – it's OK.
   (a) Clone the repository to the selected directory. After clicking *Clone* button, a window will appear where you can select from what repository you want to download source files (https://github.com/Itseez/opencv.git) and to what directory (`D:/OpenCV`).
   (b) Push the OK button and be patient as the repository is quite a heavy download. It will take some time depending on your Internet connection.
6. 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 sourceforga 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 exctracting 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!!!):

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-ltcg
```

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 enviroment 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.

7. 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* -> It creates two projects for building the documentation of OpenCV (there will be a separate project for building the HTML and the PDF files). Note that these aren't built together with the solution. You need to make an explicit build project command on these to do so.
- *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_PACKAGE* -> Prior to version 2.3 with this you could build a project that will build an OpenCV installer. With this you can easily install your OpenCV flavor on other systems. For the latest source files of OpenCV it generates a new project that simply creates zip archive with OpenCV sources.

- *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.
- *BUILD_PERF_TESTS* -> There are also performance tests for many OpenCV functions. If you're concerned about performance, build them and run.
- *BUILD_opencv_python* -> Self-explanatory. Create the binaries to use OpenCV from the Python language.

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 explicitely 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.

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 at our Q&A forum. 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 enviroment variable and add it to the systems path

First we set an enviroment variable to make easier our work. This will hold the build 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\x86\vc10
```

Here the directory is where you have your OpenCV binaries (*extracted* or *built*). You can have different platform (e.g. x64 instead of x86) or compiler type, so substitute appropriate value. 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 unpleasent 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 build directories or want to try out your applicaton 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 **D**ynamic **L**inked **L**ibraries (*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 *Proporty 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. If you don't have *"C/C++"* group, you should add any .c/.cpp file to the project.

```
$(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 latest 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:

```
// Video Image PSNR and SSIM
#include <iostream> // for standard I/O
#include <string>   // for strings
#include <iomanip>  // for controlling float print precision
#include <sstream>  // string to number conversion

#include <opencv2/imgproc/imgproc.hpp>  // Gaussian Blur
#include <opencv2/core/core.hpp>        // Basic OpenCV structures (cv::Mat, Scalar)
#include <opencv2/highgui/highgui.hpp>  // OpenCV window I/O

using namespace std;
using namespace cv;

double getPSNR ( const Mat& I1, const Mat& I2);
Scalar getMSSIM( const Mat& I1, const Mat& I2);

static void help()
{
    cout
        << "\n--------------------------------------------------------------------------" << endl
        << "This program shows how to read a video file with OpenCV. In addition, it tests the"
        << " similarity of two input videos first with PSNR, and for the frames below a PSNR "  << endl
        << "trigger value, also with MSSIM."<< endl
        << "Usage:"                                                                  << endl
        << "./video-source referenceVideo useCaseTestVideo PSNR_Trigger_Value Wait_Between_Frames " << endl
        << "--------------------------------------------------------------------------"   << endl
        << endl;
}
int main(int argc, char *argv[])
{
    help();
    if (argc != 5)
    {
        cout << "Not enough parameters" << endl;
        return -1;
    }
    stringstream conv;

    const string sourceReference = argv[1],sourceCompareWith = argv[2];
    int psnrTriggerValue, delay;
    conv << argv[3] << argv[4];       // put in the strings
    conv >> psnrTriggerValue >> delay;// take out the numbers
```

```
    char c;
    int frameNum = -1;          // Frame counter

    VideoCapture captRefrnc(sourceReference),
                 captUndTst(sourceCompareWith);

    if ( !captRefrnc.isOpened())
    {
        cout  << "Could not open reference " << sourceReference << endl;
        return -1;
    }

    if( !captUndTst.isOpened())
    {
        cout  << "Could not open case test " << sourceCompareWith << endl;
        return -1;
    }

    Size refS = Size((int) captRefrnc.get(CV_CAP_PROP_FRAME_WIDTH),
                     (int) captRefrnc.get(CV_CAP_PROP_FRAME_HEIGHT)),
         uTSi = Size((int) captUndTst.get(CV_CAP_PROP_FRAME_WIDTH),
                     (int) captUndTst.get(CV_CAP_PROP_FRAME_HEIGHT));

    if (refS != uTSi)
    {
        cout << "Inputs have different size!!! Closing." << endl;
        return -1;
    }

    const char* WIN_UT = "Under Test";
    const char* WIN_RF = "Reference";

    // Windows
            namedWindow(WIN_RF, CV_WINDOW_AUTOSIZE );
            namedWindow(WIN_UT, CV_WINDOW_AUTOSIZE );
            cvMoveWindow(WIN_RF, 400       ,            0);      //750,  2 (bernat =0)
            cvMoveWindow(WIN_UT, refS.width,            0);      //1500, 2

    cout << "Frame resolution: Width=" << refS.width << "  Height=" << refS.height
         << " of nr#: " << captRefrnc.get(CV_CAP_PROP_FRAME_COUNT) << endl;

    cout << "PSNR trigger value " <<
          setiosflags(ios::fixed) << setprecision(3) << psnrTriggerValue << endl;

    Mat frameReference, frameUnderTest;
    double psnrV;
    Scalar mssimV;

    for(;;) //Show the image captured in the window and repeat
    {
        captRefrnc >> frameReference;
        captUndTst >> frameUnderTest;

        if( frameReference.empty()  || frameUnderTest.empty())
        {
            cout << " < < <  Game over!  > > > ";
            break;
```

```
        }

        ++frameNum;
        cout <<"Frame:" << frameNum;

        ////////////////////////////// PSNR ////////////////////////////////////////////////////
        psnrV = getPSNR(frameReference,frameUnderTest);                //get PSNR
        cout << setiosflags(ios::fixed) << setprecision(3) << psnrV << "dB";

        //////////////////////////////////// MSSIM ////////////////////////////////////////////////
        if (psnrV < psnrTriggerValue)
        {
            mssimV = getMSSIM(frameReference,frameUnderTest);

            cout << " MSSIM: "
                 << "R" << setiosflags(ios::fixed) << setprecision(3) << mssimV.val[2] * 100
                 << "G" << setiosflags(ios::fixed) << setprecision(3) << mssimV.val[1] * 100
                 << "B" << setiosflags(ios::fixed) << setprecision(3) << mssimV.val[0] * 100;
        }

       cout << endl;

        ////////////////////////////////// Show Image ////////////////////////////////////////////
        imshow( WIN_RF, frameReference);
        imshow( WIN_UT, frameUnderTest);

        c = (char)cvWaitKey(delay);
        if (c == 27) break;
    }

    return 0;
}

double getPSNR(const Mat& I1, const Mat& I2)
{
    Mat s1;
    absdiff(I1, I2, s1);       // |I1 - I2|
    s1.convertTo(s1, CV_32F);  // cannot make a square on 8 bits
    s1 = s1.mul(s1);           // |I1 - I2|^2

    Scalar s = sum(s1);         // sum elements per channel

    double sse = s.val[0] + s.val[1] + s.val[2]; // sum channels

    if( sse <= 1e-10) // for small values return zero
        return 0;
    else
    {
        double  mse =sse /(double)(I1.channels() * I1.total());
        double psnr = 10.0*log10((255*255)/mse);
        return psnr;
    }
}

Scalar getMSSIM( const Mat& i1, const Mat& i2)
{
    const double C1 = 6.5025, C2 = 58.5225;
    /***************************** INITS **********************************/
```

```
    int d     = CV_32F;

    Mat I1, I2;
    i1.convertTo(I1, d);           // cannot calculate on one byte large values
    i2.convertTo(I2, d);

    Mat I2_2   = I2.mul(I2);        // I2^2
    Mat I1_2   = I1.mul(I1);        // I1^2
    Mat I1_I2  = I1.mul(I2);        // I1 * I2

    /*************************** END INITS **********************************/

    Mat mu1, mu2;   // PRELIMINARY COMPUTING
    GaussianBlur(I1, mu1, Size(11, 11), 1.5);
    GaussianBlur(I2, mu2, Size(11, 11), 1.5);

    Mat mu1_2   =   mu1.mul(mu1);
    Mat mu2_2   =   mu2.mul(mu2);
    Mat mu1_mu2 =   mu1.mul(mu2);

    Mat sigma1_2, sigma2_2, sigma12;

    GaussianBlur(I1_2, sigma1_2, Size(11, 11), 1.5);
    sigma1_2 -= mu1_2;

    GaussianBlur(I2_2, sigma2_2, Size(11, 11), 1.5);
    sigma2_2 -= mu2_2;

    GaussianBlur(I1_I2, sigma12, Size(11, 11), 1.5);
    sigma12 -= mu1_mu2;

    ///////////////////////////////// FORMULA ////////////////////////////////
    Mat t1, t2, t3;

    t1 = 2 * mu1_mu2 + C1;
    t2 = 2 * sigma12 + C2;
    t3 = t1.mul(t2);              // t3 = ((2*mu1_mu2 + C1).*(2*sigma12 + C2))

    t1 = mu1_2 + mu2_2 + C1;
    t2 = sigma1_2 + sigma2_2 + C2;
    t1 = t1.mul(t2);               // t1 =((mu1_2 + mu2_2 + C1).*(sigma1_2 + sigma2_2 + C2))

    Mat ssim_map;
    divide(t3, t1, ssim_map);      // ssim_map =  t3./t1;

    Scalar mssim = mean( ssim_map ); // mssim = average of ssim map
    return mssim;
}
```

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 voilá:

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 commmand 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:

```
D:
CD OpenCV\MySolutionName\Release
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 enviroment you have automatic argument passing. In the next introductionary tutorial you'll see an in-depth explanation of the upper source code: *Load and Display an Image*.

1.6 Image Watch: viewing in-memory images in the Visual Studio debugger

Image Watch is a plug-in for Microsoft Visual Studio that lets you to visualize in-memory images (*cv::Mat* or *IplImage_* objects, for example) while debugging an application. This can be helpful for tracking down bugs, or for simply understanding what a given piece of code is doing.

Prerequisites

This tutorial assumes that you have the following available:

1. Visual Studio 2012 Professional (or better) with Update 1 installed. Update 1 can be downloaded here.
2. An OpenCV installation on your Windows machine (Tutorial: *Installation in Windows*).
3. Ability to create and build OpenCV projects in Visual Studio (Tutorial: *How to build applications with OpenCV inside the Microsoft Visual Studio*).

Installation

Download the Image Watch installer. The installer comes in a single file with extension .vsix (*Visual Studio Extension*). To launch it, simply double-click on the .vsix file in Windows Explorer. When the installer has finished, make sure to restart Visual Studio to complete the installation.

Example

Image Watch works with any existing project that uses OpenCV image objects (for example, *cv::Mat*). In this example, we use a minimal test program that loads an image from a file and runs an edge detector. To build the program, create a console application project in Visual Studio, name it "image-watch-demo", and insert the source code below.

```
// Test application for the Visual Studio Image Watch Debugger extension

#include <iostream>                        // std::cout
#include <opencv2/core/core.hpp>           // cv::Mat
#include <opencv2/highgui/highgui.hpp>     // cv::imread()
#include <opencv2/imgproc/imgproc.hpp>     // cv::Canny()

using namespace std;
using namespace cv;

void help()
{
    cout
        << "----------------------------------------------------" << endl
        << "This is a test program for the Image Watch Debugger " << endl
        << "plug-in for Visual Studio. The program loads an     " << endl
        << "image from a file and runs the Canny edge detector. " << endl
        << "No output is displayed or written to disk."
        << endl
        << "Usage:"                                               << endl
        << "image-watch-demo inputimage"                          << endl
        << "----------------------------------------------------" << endl
        << endl;
}

int main(int argc, char *argv[])
{
    help();

    if (argc != 2)
    {
        cout << "Wrong number of parameters" << endl;
        return -1;
    }
```

```
    cout << "Loading input image: " << argv[1] << endl;
    Mat input;
    input = imread(argv[1], CV_LOAD_IMAGE_COLOR);

    cout << "Detecting edges in input image" << endl;
    Mat edges;
    Canny(input, edges, 10, 100);

    return 0;
}
```

Make sure your active solution configuration (*Build → Configuration Manager*) is set to a debug build (usually called "Debug"). This should disable compiler optimizations so that viewing variables in the debugger can work reliably.

Build your solution (*Build → Build Solution*, or press *F7*).

Now set a breakpoint on the source line that says

```
Mat edges;
```

To set the breakpoint, right-click on the source line and select *Breakpoints → Insert Breakpoint* from the context menu.

Launch the program in the debugger (*Debug → Start Debugging*, or hit *F5*). When the breakpoint is hit, the program is paused and Visual Studio displays a yellow instruction pointer at the breakpoint:

Now you can inspect the state of you program. For example, you can bring up the *Locals* window (*Debug → Windows → Locals*), which will show the names and values of the variables in the current scope:

Note that the built-in *Locals* window will display text only. This is where the Image Watch plug-in comes in. Image Watch is like another *Locals* window, but with an image viewer built into it. To bring up Image Watch, select *View → Other Windows → Image Watch*. Like Visual Studio's *Locals* window, Image Watch can dock to the Visual Studio IDE. Also, Visual Studio will remember whether you had Image Watch open, and where it was located between debugging sessions. This means you only have to do this once–the next time you start debugging, Image Watch will be back where you left it. Here's what the docked Image Watch window looks like at our breakpoint:

The radio button at the top left (*Locals/Watch*) selects what is shown in the *Image List* below: *Locals* lists all OpenCV image objects in the current scope (this list is automatically populated). *Watch* shows image expressions that have been pinned for continuous inspection (not described here, see Image Watch documentation for details). The image list shows basic information such as width, height, number of channels, and, if available, a thumbnail. In our example, the image list contains our two local image variables, *input* and *edges*.

If an image has a thumbnail, left-clicking on that image will select it for detailed viewing in the *Image Viewer* on the right. The viewer lets you pan (drag mouse) and zoom (mouse wheel). It also displays the pixel coordinate and value at the current mouse position.

Note that the second image in the list, *edges*, is shown as "invalid". This indicates that some data members of this image object have corrupt or invalid values (for example, a negative image width). This is expected at this point in the program, since the C++ constructor for *edges* has not run yet, and so its members have undefined values (in debug

mode they are usually filled with "0xCD" bytes).

From here you can single-step through your code (*Debug->Step Over*, or press *F10*) and watch the pixels change: if you step once, over the *Mat edges;* statement, the *edges* image will change from "invalid" to "empty", which means that it is now in a valid state (default constructed), even though it has not been initialized yet (using *cv::Mat::create()*, for example). If you make one more step over the *cv::Canny()* call, you will see a thumbnail of the edge image appear in the image list.

Now assume you want to do a visual sanity check of the *cv::Canny()* implementation. Bring the *edges* image into the viewer by selecting it in the *Image List* and zoom into a region with a clearly defined edge:

Right-click on the *Image Viewer* to bring up the view context menu and enable *Link Views* (a check box next to the menu item indicates whether the option is enabled).

The *Link Views* feature keeps the view region fixed when flipping between images of the same size. To see how this works, select the input image from the image list–you should now see the corresponding zoomed-in region in the input image:

You may also switch back and forth between viewing input and edges with your up/down cursor keys. That way you can easily verify that the detected edges line up nicely with the data in the input image.

More ...

Image watch has a number of more advanced features, such as

1. pinning images to a *Watch* list for inspection across scopes or between debugging sessions
2. clamping, thresholding, or diff'ing images directly inside the Watch window
3. comparing an in-memory image against a reference image from a file

Please refer to the online Image Watch Documentation for details–you also can get to the documentation page by clicking on the *Help* link in the Image Watch window:

1.7 Introduction to Java Development

As of OpenCV 2.4.4, OpenCV supports desktop Java development using nearly the same interface as for Android development. This guide will help you to create your first Java (or Scala) application using OpenCV. We will use either Eclipse, Apache Ant or the Simple Build Tool (SBT) to build the application.

For further reading after this guide, look at the *Introduction into Android Development* tutorials.

What we'll do in this guide

In this guide, we will:

- Get OpenCV with desktop Java support
- Create an `Ant`, `Eclipse` or `SBT` project
- Write a simple OpenCV application in Java or Scala

The same process was used to create the samples in the `samples/java` folder of the OpenCV repository, so consult those files if you get lost.

Get proper OpenCV

Starting from version 2.4.4 OpenCV includes desktop Java bindings.

Download

The most simple way to get it is downloading the appropriate package of **version 2.4.4 or higher** from the OpenCV SourceForge repository.

Note: Windows users can find the prebuilt files needed for Java development in the `opencv/build/java/` folder inside the package. For other OSes it's required to build OpenCV from sources.

Another option to get OpenCV sources is to clone OpenCV git repository. In order to build OpenCV with Java bindings you need JDK (Java Development Kit) (we recommend Oracle/Sun JDK 6 or 7), Apache Ant and *Python* v2.6 or higher to be installed.

Build

Let's build OpenCV:

```
git clone git://github.com/Itseez/opencv.git
cd opencv
git checkout 2.4
mkdir build
cd build
```

Generate a Makefile or a MS Visual Studio* solution, or whatever you use for building executables in your system:

```
cmake -DBUILD_SHARED_LIBS=OFF ..
```

or

```
cmake -DBUILD_SHARED_LIBS=OFF -G "Visual Studio 10" ..
```

Note: When OpenCV is built as a set of **static** libraries (`-DBUILD_SHARED_LIBS=OFF` option) the Java bindings dynamic library is all-sufficient, i.e. doesn't depend on other OpenCV libs, but includes all the OpenCV code inside.

Examine the output of CMake and ensure `java` is one of the modules "To be built". If not, it's likely you're missing a dependency. You should troubleshoot by looking through the CMake output for any Java-related tools that aren't found and installing them.

Note: If `CMake` can't find Java in your system set the `JAVA_HOME` environment variable with the path to installed JDK before running it. E.g.:

```
export JAVA_HOME=/usr/lib/jvm/java-6-oracle
cmake -DBUILD_SHARED_LIBS=OFF ..
```

Now start the build:

```
make -j8
```

or

```
msbuild /m OpenCV.sln /t:Build /p:Configuration=Release /v:m
```

Besides all this will create a `jar` containing the Java interface (`bin/opencv-244.jar`) and a native dynamic library containing Java bindings and all the OpenCV stuff (`lib/libopencv_java244.so` or `bin/Release/opencv_java244.dll` respectively). We'll use these files later.

Java sample with Ant

Note: The described sample is provided with OpenCV library in the `opencv/samples/java/ant` folder.

- Create a folder where you'll develop this sample application.
- In this folder create the `build.xml` file with the following content using any text editor:

```xml
<project name="SimpleSample" basedir="." default="rebuild-run">

    <property name="src.dir"     value="src"/>

    <property name="lib.dir"     value="${ocvJarDir}"/>
    <path id="classpath">
        <fileset dir="${lib.dir}" includes="**/*.jar"/>
    </path>

    <property name="build.dir"   value="build"/>
    <property name="classes.dir" value="${build.dir}/classes"/>
    <property name="jar.dir"     value="${build.dir}/jar"/>

    <property name="main-class"  value="${ant.project.name}"/>


    <target name="clean">
        <delete dir="${build.dir}"/>
    </target>

    <target name="compile">
        <mkdir dir="${classes.dir}"/>
        <javac includeantruntime="false" srcdir="${src.dir}" destdir="${classes.dir}" classpathref="classpath
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="${jar.dir}"/>
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">
            <manifest>
                <attribute name="Main-Class" value="${main-class}"/>
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java fork="true" classname="${main-class}">
            <sysproperty key="java.library.path" path="${ocvLibDir}"/>
            <classpath>
```

```
39                      <path refid="classpath"/>
40                      <path location="${jar.dir}/${ant.project.name}.jar"/>
41                  </classpath>
42              </java>
43          </target>
44
45          <target name="rebuild" depends="clean,jar"/>
46
47          <target name="rebuild-run" depends="clean,run"/>
48
49      </project>
```

Note: This XML file can be reused for building other Java applications. It describes a common folder structure in the lines 3 - 12 and common targets for compiling and running the application.

When reusing this XML don't forget to modify the project name in the line 1, that is also the name of the *main* class (line 14). The paths to OpenCV *jar* and *jni lib* are expected as parameters (`"${ocvJarDir}"` in line 5 and `"${ocvLibDir}"` in line 37), but you can hardcode these paths for your convenience. See Ant documentation for detailed description of its build file format.

- Create an `src` folder next to the `build.xml` file and a `SimpleSample.java` file in it.
- **Put the following Java code into the `SimpleSample.java` file:**

```
import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.CvType;
import org.opencv.core.Scalar;

class SimpleSample {

  static{ System.loadLibrary(Core.NATIVE_LIBRARY_NAME); }

  public static void main(String[] args) {
    System.out.println("Welcome to OpenCV " + Core.VERSION);
    Mat m = new Mat(5, 10, CvType.CV_8UC1, new Scalar(0));
    System.out.println("OpenCV Mat: " + m);
    Mat mr1 = m.row(1);
    mr1.setTo(new Scalar(1));
    Mat mc5 = m.col(5);
    mc5.setTo(new Scalar(5));
    System.out.println("OpenCV Mat data:\n" + m.dump());
  }

}
```

- **Run the following command in console in the folder containing `build.xml`:**

```
ant -DocvJarDir=path/to/dir/containing/opencv-244.jar -DocvLibDir=path/to/dir/containing/opencv_java244/nati
```

For example:

```
ant -DocvJarDir=X:\opencv-2.4.4\bin -DocvLibDir=X:\opencv-2.4.4\bin\Release
```

The command should initiate [re]building and running the sample. You should see on the screen something like this:

Java project in Eclipse

Now let's look at the possiblity of using OpenCV in Java when developing in Eclipse IDE.

- Create a new Eclipse workspace
- Create a new Java project via *File –> New –> Java Project*

Call it say "HelloCV".

- Open *Java Build Path* tab on *Project Properties* dialog and configure additional library (OpenCV) reference (jar and native library location):

Properties for HelloCV
Java Build Path
Source
Projects
Order and Export
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel

Add Library
Add Library
Select the library type to add.
Android Classpath Container
JRE System Library
JUnit
Plug-in Dependencies
User Library
< Back
Next >
Finish
Cancel

Add Library
User Library
Select a library to add to the classpath.
User libraries:
User Libraries...
< Back
Next >
Finish
Cancel

Preferences (Filtered)
User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.
Defined user libraries:
New...
Edit...
Add JARs...
Add External JARs...
Remove
Up
Down
Import...
Export...
OK
Cancel

New User Library
User library name:
opencv-2.4.4
System library (added to the boot class path)
OK
Cancel

Preferences (Filtered)
type filter text
User Libraries
can be added to a Java Build path and bundle a number of external
em libraries will be added to the boot class path when launched.
ibraries:
cv-2.4.4
New...
Edit...
Add JARs...
Add External JARs...
Remove
Up
Down
Import...
Export...
OK
Cancel
Java
Build Path
User Libraries
User libraries
archives. Syste
Defined user l
open

Preferences (Filtered)
type filter text
User Libraries
User Libraries
User libraries can be added to a Java Build path and bundle a nur
archives. System libraries will be added to the boot class path wh
Defined user libraries:
opencv-2.4.4
opencv-2.4.4.jar - X:\opencv-2.4.4\bin
Source attachment: (None)
Javadoc location: (None)
Native library location: (None)
Access rules: (No restrictions)
New...
Edit...
Add JARs...
Remove
Up
Down
Import...
Export...
Cancel
OK

Add Library
User Library
Select a library to add to the classpath.
User libraries:
opencv-2.4.4
User Libraries...
< Back
Next >
Finish
Cancel

- Add a new Java class (say `Main`) containing the application entry:

- Put some simple OpenCV calls there, e.g.:

```
import org.opencv.core.Core;
import org.opencv.core.CvType;
import org.opencv.core.Mat;

public class Main {
    public static void main(String[] args) {
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
        Mat m  = Mat.eye(3, 3, CvType.CV_8UC1);
        System.out.println("m = " + m.dump());
    }
}
```

- Press *Run* button and find the identity matrix content in the Eclipse `Console` window.

SBT project for Java and Scala

Now we'll create a simple Java application using SBT. This serves as a brief introduction to those unfamiliar with this build tool. We're using SBT because it is particularly easy and powerful.

First, download and install SBT using the instructions on its web site.

Next, navigate to a new directory where you'd like the application source to live (outside `opencv` dir). Let's call it "JavaSample" and create a directory for it:

```
cd <somewhere outside opencv>
mkdir JavaSample
```

Now we will create the necessary folders and an SBT project:

```
cd JavaSample
mkdir -p src/main/java # This is where SBT expects to find Java sources
mkdir project # This is where the build definitions live
```

Now open `project/build.scala` in your favorite editor and paste the following. It defines your project:

```
import sbt._
import Keys._

object JavaSampleBuild extends Build {
```

```
  def scalaSettings = Seq(
    scalaVersion := "2.10.0",
    scalacOptions ++= Seq(
      "-optimize",
      "-unchecked",
      "-deprecation"
    )
  )

  def buildSettings =
    Project.defaultSettings ++
    scalaSettings

  lazy val root = {
    val settings = buildSettings ++ Seq(name := "JavaSample")
    Project(id = "JavaSample", base = file("."), settings = settings)
  }
}
```

Now edit `project/plugins.sbt` and paste the following. This will enable auto-generation of an Eclipse project:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.1.0")
```

Now run `sbt` from the `JavaSample` root and from within SBT run `eclipse` to generate an eclipse project:

```
sbt # Starts the sbt console
> eclipse # Running "eclipse" from within the sbt console
```

You should see something like this:

You can now import the SBT project to Eclipse using *Import ... -> Existing projects into workspace*. Whether you actually do this is optional for the guide; we'll be using SBT to build the project, so if you choose to use Eclipse it will just serve as a text editor.

To test that everything is working, create a simple "Hello OpenCV" application. Do this by creating a file `src/main/java/HelloOpenCV.java` with the following contents:

```
public class HelloOpenCV {
  public static void main(String[] args) {
    System.out.println("Hello, OpenCV");
  }
}
```

Now execute `run` from the sbt console, or more concisely, run `sbt run` from the command line:

```
sbt run
```

You should see something like this:

Running SBT samples

Now we'll create a simple face detection application using OpenCV.

First, create a `lib/` folder and copy the OpenCV jar into it. By default, SBT adds jars in the lib folder to the Java library search path. You can optionally rerun `sbt eclipse` to update your Eclipse project.

```
mkdir lib
cp <opencv_dir>/build/bin/opencv_<version>.jar lib/
sbt eclipse
```

Next, create the directory `src/main/resources` and download this Lena image into it:

Make sure it's called `"lena.png"`. Items in the resources directory are available to the Java application at runtime.

Next, copy `lbpcascade_frontalface.xml` from `opencv/data/lbpcascades/` into the `resources` directory:

```
cp <opencv_dir>/data/lbpcascades/lbpcascade_frontalface.xml src/main/resources/
```

Now modify src/main/java/HelloOpenCV.java so it contains the following Java code:

```
import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.MatOfRect;
import org.opencv.core.Point;
import org.opencv.core.Rect;
import org.opencv.core.Scalar;
import org.opencv.highgui.Highgui;
import org.opencv.objdetect.CascadeClassifier;

//
// Detects faces in an image, draws boxes around them, and writes the results
// to "faceDetection.png".
//
class DetectFaceDemo {
  public void run() {
    System.out.println("\nRunning DetectFaceDemo");
```

```
    // Create a face detector from the cascade file in the resources
    // directory.
    CascadeClassifier faceDetector = new CascadeClassifier(getClass().getResource("/lbpcascade_frontalface.xml").getPa
    Mat image = Highgui.imread(getClass().getResource("/lena.png").getPath());

    // Detect faces in the image.
    // MatOfRect is a special container class for Rect.
    MatOfRect faceDetections = new MatOfRect();
    faceDetector.detectMultiScale(image, faceDetections);

    System.out.println(String.format("Detected %s faces", faceDetections.toArray().length));

    // Draw a bounding box around each face.
    for (Rect rect : faceDetections.toArray()) {
        Core.rectangle(image, new Point(rect.x, rect.y), new Point(rect.x + rect.width, rect.y + rect.height), new Sca
    }

    // Save the visualized detection.
    String filename = "faceDetection.png";
    System.out.println(String.format("Writing %s", filename));
    Highgui.imwrite(filename, image);
  }
}

public class HelloOpenCV {
  public static void main(String[] args) {
    System.out.println("Hello, OpenCV");

    // Load the native library.
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    new DetectFaceDemo().run();
  }
}
```

Note the call to `System.loadLibrary(Core.NATIVE_LIBRARY_NAME)`. This command must be executed exactly once per Java process prior to using any native OpenCV methods. If you don't call it, you will get `UnsatisfiedLinkErrors`. You will also get errors if you try to load OpenCV when it has already been loaded.

Now run the face detection app using `sbt run`:

```
sbt run
```

You should see something like this:

It should also write the following image to `faceDetection.png`:

You're done! Now you have a sample Java application working with OpenCV, so you can start the work on your own. We wish you good luck and many years of joyful life!

1.8 Introduction into Android Development

This guide was designed to help you in learning Android development basics and seting up your working environment quickly. It was written with Windows 7 in mind, though it would work with Linux (Ubuntu), Mac OS X and any other OS supported by Android SDK.

If you encounter any error after thoroughly following these steps, feel free to contact us via OpenCV4Android discussion group or OpenCV Q&A forum. We'll do our best to help you out.

Preface

Android is a Linux-based, open source mobile operating system developed by Open Handset Alliance led by Google. See the Android home site for general details.

Development for Android significantly differs from development for other platforms. So before starting programming for Android we recommend you make sure that you are familiar with the following key topis:

1. Java programming language that is the primary development technology for Android OS. Also, you can find Oracle docs on Java useful.
2. Java Native Interface (JNI) that is a technology of running native code in Java virtual machine. Also, you can find Oracle docs on JNI useful.
3. Android Activity and its lifecycle, that is an essential Android API class.
4. OpenCV development will certainly require some knowlege of the Android Camera specifics.

Quick environment setup for Android development

If you are making a clean environment install, then you can try Tegra Android Development Pack (**TADP**) released by **NVIDIA**.

> **Note:** Starting the *version 2.0* the TADP package includes *OpenCV for Tegra* SDK that is a regular *OpenCV4Android SDK* extended with Tegra-specific stuff.

When unpacked, TADP will cover all of the environment setup automatically and you can skip the rest of the guide.

If you are a beginner in Android development then we also recommend you to start with TADP.

> **Note:** *NVIDIA*'s Tegra Android Development Pack includes some special features for *NVIDIA*'s Tegra platform but its use is not limited to *Tegra* devices only.

- You need at least *1.6 Gb* free disk space for the install.
- TADP will download Android SDK platforms and Android NDK from Google's server, so Internet connection is required for the installation.
- TADP may ask you to flash your development kit at the end of installation process. Just skip this step if you have no Tegra Development Kit.
- (`UNIX`) TADP will ask you for *root* in the middle of installation, so you need to be a member of *sudo* group.

Manual environment setup for Android development

Development in Java

You need the following software to be installed in order to develop for Android in Java:

1. **Sun JDK 6** (Sun JDK 7 is also possible)

   Visit Java SE Downloads page and download an installer for your OS.

   Here is a detailed JDK installation guide for Ubuntu and Mac OS (only JDK sections are applicable for OpenCV)

Note: OpenJDK is not suitable for Android development, since Android SDK supports only Sun JDK. If you use Ubuntu, after installation of Sun JDK you should run the following command to set Sun java environment:

```
sudo update-java-alternatives --set java-6-sun
```

2. **Android SDK**

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

   Here is Google's install guide for the SDK.

Note: You can choose downloading **ADT Bundle package** that in addition to Android SDK Tools includes Eclipse + ADT + NDK/CDT plugins, Android Platform-tools, the latest Android platform and the latest Android system image for the emulator - this is the best choice for those who is setting up Android development environment the first time!

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

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

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

```
sudo yum install libXtst.i386
```

3. **Android SDK components**

   You need the following SDK components to be installed:

   - *Android SDK Tools, revision 20* or newer.

     Older revisions should also work, but they are not recommended.

   - *SDK Platform Android 3.0* (`API 11`).

     The minimal platform supported by OpenCV Java API is **Android 2.2** (`API 8`). This is also the minimum API Level required for the provided samples to run. See the `<uses-sdk android:minSdkVersion="8"/>` tag in their **AndroidManifest.xml** files. But for successful compilation the **target** platform should be set to Android 3.0 (API 11) or higher. It will not prevent them from running on Android 2.2.

See Adding Platforms and Packages 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 2.4.x we recommend **Eclipse 3.7 (Indigo)** or **Eclipse 4.2 (Juno)**. They work well for OpenCV under both Windows and Linux.

   If you have no Eclipse installed, you can get it from the official site.

5. **ADT plugin for Eclipse**

   These instructions are copied from Android Developers site, check it out in case of any ADT-related problem.

   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*.

Note: If you also plan to develop native C++ code with Android NDK don't forget to enable *NDK Plugins* installations as well.

(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.

Native development in C++

You need the following software to be installed in order to develop for Android in C++:

1. **Android NDK**

   To compile C++ code for Android platform you need `Android Native Development Kit` (*NDK*).

   You can get the latest version of NDK from the download page. To install Android NDK just extract the archive to some folder on your computer. Here are installation instructions.

   Note: Before start you can read official Android NDK documentation which is in the Android NDK archive, in the folder `docs/`. The main article about using Android NDK build system is in the `ANDROID-MK.html` file. Some additional information you can find in the `APPLICATION-MK.html`, `NDK-BUILD.html` files, and `CPU-ARM-NEON.html`, `CPLUSPLUS-SUPPORT.html`, `PREBUILTS.html`.

2. **CDT plugin for Eclipse**

   If you selected for installation the `NDK plugins` component of Eclipse ADT plugin (see the picture above) your Eclipse IDE should already have `CDT plugin` (that means `C/C++ Development Tooling`). There are several possible ways to integrate compilation of C++ code by Android NDK into Eclipse compilation process. We recommend the approach based on Eclipse CDT (C/C++ Development Tooling) Builder.

Android application structure

Usually source code of an Android application has the following structure:

- `root folder of the project/`
  - `jni/`
  - `libs/`
  - `res/`
  - `src/`
  - `AndroidManifest.xml`
  - `project.properties`
  - `... other files ...`

Where:

- the `src` folder contains Java code of the application,
- the `res` folder contains resources of the application (images, xml files describing UI layout, etc),
- the `libs` folder will contain native libraries after a successful build,
- and the `jni` folder contains C/C++ application source code and NDK's build scripts `Android.mk` and `Application.mk` producing the native libraries,
- `AndroidManifest.xml` file presents essential information about application to the Android system (name of the Application, name of main application's package, components of the application, required permissions, etc).

  It can be created using Eclipse wizard or **android** tool from Android SDK.
- `project.properties` is a text file containing information about target Android platform and other build details. This file is generated by Eclipse or can be created with **android** tool included in Android SDK.

Note: Both `AndroidManifest.xml` and `project.properties` files are required to compile the C++ part of the application, since Android NDK build system relies on them. If any of these files does not exist, compile the Java part of the project before the C++ part.

Android.mk and Application.mk scripts

The script Android.mk usually has the following structure:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE    := <module_name>
LOCAL_SRC_FILES := <list of .c and .cpp project files>
<some variable name> := <some variable value>
...
<some variable name> := <some variable value>

include $(BUILD_SHARED_LIBRARY)
```

This is the minimal file Android.mk, which builds C++ source code of an Android application. Note that the first two lines [illegible]

3. After executing this command the C++ part of the source code is compiled.

After that the Java part of the application can be (re)compiled (using either *Eclipse* or *Ant* build tool).

Note: Some parameters can be set for the **ndk-build**:

Example 1: Verbose compilation

```
<path_where_NDK_is_placed>/ndk-build V=1
```

Example 2: Rebuild all

```
<path_where_NDK_is_placed>/ndk-build -B
```

Building application native part from *Eclipse* (CDT Builder)

There are several possible ways to integrate compilation of native C++ code by Android NDK into Eclipse build process. We recommend the approach based on Eclipse CDT Builder.

Important: OpenCV for Android package since version 2.4.2 contains sample projects pre-configured CDT Builders. For your own projects follow the steps below.

---

1. Define the `NDKROOT` environment variable containing the path to Android NDK in your system (e.g. `"X:\\Apps\\android-ndk-r8"` or `"/opt/android-ndk-r8"`).

   On Windows an environment variable can be set via *My Computer -> Properties -> Advanced -> Environment variables*. On Windows 7 it's also possible to use setx command in a console session.

   On Linux and **MacOS** an environment variable can be set via appending a `"export VAR_NAME=VAR_VALUE"` line to the `"~/.bashrc"` file and logging off and then on.

   Note: It's also possible to define the `NDKROOT` environment variable within Eclipse IDE, but it should be done for every new workspace you create. If you prefer this option better than setting system environment variable, open Eclipse menu *Window -> Preferences -> C/C++ -> Build -> Environment*, press the *Add...* button and set variable name to `NDKROOT` and value to local Android NDK path.

   ---

2. After that you need to **restart Eclipse** to apply the changes.
3. Open Eclipse and load the Android app project to configure.
4. Add C/C++ Nature to the project via Eclipse menu *New -> Other -> C/C++ -> Convert to a C/C++ Project*.

Tutorial 2 (Basi
Tutorial 3 (Adv
Tutorial 4 (Adv
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Java Project
Android Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
Android XML File
JUnit Test Case
Example...
Other... Ctrl+N

And:

5. Select the project(s) to convert. Specify "Project type" = `Makefile project`, "Toolchains" = `Other Toolchain`.

6. Open *Project Properties -> C/C++ Build*, uncheck `Use default build command`, replace "Build command" text from `"make"` to

   `"${NDKROOT}/ndk-build.cmd"` on Windows,

   `"${NDKROOT}/ndk-build"` on Linux and MacOS.

7. Go to *Behaviour* tab and change "Workbench build type" section like shown below:

8. Press *OK* and make sure the `ndk-build` is successfully invoked when building the project.

9. If you open your C++ source file in Eclipse editor, you'll see syntax error notifications. They are not real errors, but additional CDT configuring is required.

10. Open *Project Properties -> C/C++ General -> Paths and Symbols* and add the following **Include** paths for **C++**:

```
# for NDK r8 and prior:
${NDKROOT}/platforms/android-9/arch-arm/usr/include
${NDKROOT}/sources/cxx-stl/gnu-libstdc++/include
${NDKROOT}/sources/cxx-stl/gnu-libstdc++/libs/armeabi-v7a/include
${ProjDirPath}/../../sdk/native/jni/include
```

```
# for NDK r8b and later:
${NDKROOT}/platforms/android-9/arch-arm/usr/include
${NDKROOT}/sources/cxx-stl/gnu-libstdc++/4.6/include
${NDKROOT}/sources/cxx-stl/gnu-libstdc++/4.6/libs/armeabi-v7a/include
${ProjDirPath}/../../sdk/native/jni/include
```

The last path should be changed to the correct absolute or relative path to OpenCV4Android SDK location.

This should clear the syntax error notifications in Eclipse C++ editor.

Debugging and Testing

In this section we will give you some easy-to-follow instructions on how to set up an emulator or hardware device for testing and debugging an Android project.

AVD

AVD (*Android Virtual Device*) is not probably the most convenient way to test an OpenCV-dependent application, but sure the most uncomplicated one to configure.

1. Assuming you already have *Android SDK* and *Eclipse IDE* installed, in Eclipse go *Window -> AVD Manager*.
2. Press the *New* button in *AVD Manager* window.
3. *Create new Android Virtual Device* window will let you select some properties for your new device, like target API level, size of SD-card and other.

4. When you click the *Create AVD* button, your new AVD will be availible in *AVD Manager*.
5. Press *Start* to launch the device. Be aware that any AVD (a.k.a. Emulator) is usually much slower than a hardware Android device, so it may take up to several minutes to start.
6. Go *Run -> Run/Debug* in Eclipse IDE to run your application in regular or debugging mode. *Device Chooser* will let you choose among the running devices or to start a new one.

Hardware Device

If you have an Android device, you can use it to test and debug your applications. This way is more authentic, though a little bit harder to set up. You need to make some actions for Windows and Linux operating systems to be able to work with Android devices. No extra actions are required for Mac OS. See detailed information on configuring hardware

devices in subsections below.

You may also consult the official Android Developers site instructions for more information.

Windows host computer

1. Enable USB debugging on the Android device (via *Settings* menu).
2. Attach the Android device to your PC with a USB cable.
3. Go to *Start Menu* and **right-click** on *Computer*. Select *Manage* in the context menu. You may be asked for Administrative permissions.
4. Select *Device Manager* in the left pane and find an unknown device in the list. You may try unplugging it and then plugging back in order to check whether it's your exact equipment appears in the list.

5. Try your luck installing *Google USB drivers* without any modifications: **right-click** on the unknown device, select *Properties* menu item –> *Details* tab –> *Update Driver* button.

6. Select *Browse computer for driver software*.

7. Specify the path to `<Android SDK folder>/extras/google/usb_driver/` folder.

8. If you get the prompt to install unverified drivers and report about success - you've finished with USB driver installation.

‘ ‘

9. Otherwise (getting the failure like shown below) follow the next steps.

10. Again **right-click** on the unknown device, select *Properties –> Details –> Hardware Ids* and copy the line like `USB\VID_XXXX&PID_XXXX&MI_XX`.

11. Now open file `<Android SDK folder>/extras/google/usb_driver/android_winusb.inf`. Select either `Google.NTx86` or `Google.NTamd64` section depending on your host system architecture.

```
[Version]
Signature           = "$Windows NT$"
Class               = AndroidUsbDeviceClass
ClassGuid           = {3F966BD9-FA04-4ec5-991C-D326973B5128}
Provider            = %ProviderName%
DriverVer           = 12/06/2010,4.0.0000.00000
CatalogFile.NTx86   = androidwinusb86.cat
CatalogFile.NTamd64 = androidwinusba64.cat

;
; This section seems to be required for WinUsb driver installation.
; If this section is removed the installer will report an error
; "Required section not found in INF file".
;
[ClassInstall32]
Addreg = AndroidWinUsbClassReg

[AndroidWinUsbClassReg]
HKR,,,0,%ClassName%
HKR,,Icon,,-1

[Manufacturer]
%ProviderName% = Google, NTx86, NTamd64

[Google.NTx86]
; HTC Dream
%SingleAdbInterface%        = USB_Install, USB\VID_0BB4&PID_0C01
%CompositeAdbInterface%     = USB_Install, USB\VID_0BB4&PID_0C02&MI_01
%SingleBootLoaderInterface% = USB_Install, USB\VID_0BB4&PID_0FFF
; HTC Magic
%CompositeAdbInterface%     = USB_Install, USB\VID_0BB4&PID_0C03&MI_01
;
;Moto Sholes
%SingleAdbInterface%        = USB_Install, USB\VID_22B8&PID_41DB
%CompositeAdbInterface%     = USB_Install, USB\VID_22B8&PID_41DB&MI_01
;
;Google NexusOne
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_0D02
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_0D02&MI_01
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_4E11
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E12&MI_01
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E22&MI_01

[Google.NTamd64]
; HTC Dream
%SingleAdbInterface%        = USB_Install, USB\VID_0BB4&PID_0C01
```

12. There should be a record like existing ones for your device and you need to add one manually.

```
CatalogFile.NTx86   = androidwinusb86.cat
CatalogFile.NTamd64 = androidwinusba64.cat

;
; This section seems to be required for WinUsb driver installation.
; If this section is removed the installer will report an error
; "Required section not found in INF file".
;
[ClassInstall32]
Addreg = AndroidWinUsbClassReg

[AndroidWinUsbClassReg]
HKR,,,0,%ClassName%
HKR,,Icon,,-1

[Manufacturer]
%ProviderName% = Google, NTx86, NTamd64

[Google.NTx86]

;my device (HTC Desire HD)
%CompositeAdbInterface%     = USB_Install, USB\VID_0BB4&PID_0CA2&MI_01

; HTC Dream
%SingleAdbInterface%        = USB_Install, USB\VID_0BB4&PID_0C01
%CompositeAdbInterface%     = USB_Install, USB\VID_0BB4&PID_0C02&MI_01
%SingleBootLoaderInterface% = USB_Install, USB\VID_0BB4&PID_0FFF
; HTC Magic
%CompositeAdbInterface%     = USB_Install, USB\VID_0BB4&PID_0C03&MI_01
;
;Moto Sholes
%SingleAdbInterface%        = USB_Install, USB\VID_22B8&PID_41DB
%CompositeAdbInterface%     = USB_Install, USB\VID_22B8&PID_41DB&MI_01
;
;Google NexusOne
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_0D02
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_0D02&MI_01
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_4E11
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E12&MI_01
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E22&MI_01

[Google.NTamd64]
; HTC Dream
%SingleAdbInterface%        = USB_Install, USB\VID_0BB4&PID_0C01
%CompositeAdbInterface%     = USB_Install, USB\VID_0BB4&PID_0C02&MI_01
%SingleBootLoaderInterface% = USB_Install, USB\VID_0BB4&PID_0FFF
```

13. Save the `android_winusb.inf` file and try to install the USB driver again.

‘ ‘

Update Driver Software - Android Phone
Update Driver Software - Android Phone
How do you want to search for driver software?
Search automatically for updated driver software
Windows will search your computer and the Internet for the latest driver software for your device, unless you've disabled this feature in your device installation settings.
Browse my computer for driver software
Locate and install driver software manually.
Cancel

‘ ‘

14. This time installation should go successfully.

‘ ‘

15. And an unknown device is now recognized as an Android phone.

16. Successful device USB connection can be verified in console via `adb devices` command.

17. Now, in Eclipse go *Run -> Run/Debug* to run your application in regular or debugging mode. *Device Chooser* will let you choose among the devices.

Linux host computer

By default Linux doesn't recognize Android devices, but it's easy to fix this issue. On Ubuntu Linux you have to create a new **/etc/udev/rules.d/51-android.rules** configuration file that contains information about your Android device. You may find some Vendor ID's here or execute **lsusb** command to view VendorID of plugged Android device. Here is an example of such file for LG device:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="1004",  MODE="0666", GROUP="plugdev"
```

Then restart your adb server (even better to restart the system), plug in your Android device and execute **adb devices** command. You will see the list of attached devices:

Mac OS host computer

No actions are required, just connect your device via USB and run `adb devices` to check connection.

What's next

Now, when you have your development environment set up and configured, you may want to proceed to installing OpenCV4Android SDK. You can learn how to do that in a separate *OpenCV4Android SDK* tutorial.

1.9 OpenCV4Android SDK

This tutorial was designed to help you with installation and configuration of OpenCV4Android SDK.

This guide was written with MS Windows 7 in mind, though it should work with GNU Linux and Apple Mac OS as well.

This tutorial assumes you have the following software installed and configured:

- JDK
- Android SDK and NDK
- Eclipse IDE
- ADT and CDT plugins for Eclipse

If you need help with anything of the above, you may refer to our *Introduction into Android Development* guide.

If you encounter any error after thoroughly following these steps, feel free to contact us via OpenCV4Android discussion group or OpenCV Q&A forum. We'll do our best to help you out.

Tegra Android Development Pack users

You may have used Tegra Android Development Pack (**TADP**) released by **NVIDIA** for Android development environment setup.

Beside Android development tools the TADP 2.0 includes OpenCV4Android SDK, so it can be already installed in your system and you can skip to *Running OpenCV Samples* section of this tutorial.

More details regarding TADP can be found in the *Introduction into Android Development* guide.

General info

OpenCV4Android SDK package enables development of Android applications with use of OpenCV library.

The structure of package contents looks as follows:

```
OpenCV-2.4.6-android-sdk
|_ apk
|   |_ OpenCV_2.4.6_binary_pack_armv7a.apk
|   |_ OpenCV_2.4.6_Manager_2.9_XXX.apk
|
|_ doc
|_ samples
|_ sdk
|    |_ etc
|    |_ java
|    |_ native
|          |_ 3rdparty
|          |_ jni
|          |_ libs
|               |_ armeabi
|               |_ armeabi-v7a
|               |_ x86
|
|_ license.txt
|_ README.android
```

- `sdk` folder contains OpenCV API and libraries for Android:
- `sdk/java` folder contains an Android library Eclipse project providing OpenCV Java API that can be imported into developer's workspace;
- `sdk/native` folder contains OpenCV C++ headers (for JNI code) and native Android libraries (*.so and *.a) for ARM-v5, ARM-v7a and x86 architectures;
- `sdk/etc` folder contains Haar and LBP cascades distributed with OpenCV.
- `apk` folder contains Android packages that should be installed on the target Android device to enable OpenCV library access via OpenCV Manager API (see details below).

  On production devices that have access to Google Play Market (and Internet) these packages will be installed from Market on the first start of an application using OpenCV Manager API. But devkits without Market or Internet connection require this packages to be installed manually. Install the *Manager.apk* and optional *binary_pack.apk* if it needed. See *manager_selection* for details.

  Note: Installation from Internet is the preferable way since OpenCV team may publish updated versions of this packages on the Market.

- `samples` folder contains sample applications projects and their prebuilt packages (APK). Import them into Eclipse workspace (like described below) and browse the code to learn possible ways of OpenCV use on Android.
- `doc` folder contains various OpenCV documentation in PDF format. It's also available online at http://docs.opencv.org.

> **Note:** The most recent docs (nightly build) are at http://docs.opencv.org/2.4. Generally, it's more up-to-date, but can refer to not-yet-released functionality.

Starting from version 2.4.3 *OpenCV4Android SDK* uses *OpenCV Manager* API for library initialization. *OpenCV Manager* is an Android service based solution providing the following benefits for OpenCV applications developers:

- Compact apk-size, since all applications use the same binaries from Manager and do not store native libs within themselves;
- Hardware specific optimizations are automatically enabled on all supported platforms;
- Automatic updates and bug fixes;
- Trusted OpenCV library source. All packages with OpenCV are published on Google Play;

For additional information on OpenCV Manager see the:

- Slides
- Reference Manual

Manual OpenCV4Android SDK setup

Get the OpenCV4Android SDK

1. Go to the OpenCV download page on SourceForge and download the latest available version. Currently it's `OpenCV-2.4.6-android-sdk.zip`.
2. Create a new folder for Android with OpenCV development. For this tutorial we have unpacked OpenCV SDK to the `C:\Work\OpenCV4Android\` directory.

   > **Note:** Better to use a path without spaces in it. Otherwise you may have problems with **ndk-build**.

3. Unpack the SDK archive into the chosen directory.

   You can unpack it using any popular archiver (e.g with 7-Zip):

On Unix you can use the following command:

```
unzip ~/Downloads/OpenCV-2.4.6-android-sdk.zip
```

Import OpenCV library and samples to the Eclipse

1. Start Eclipse and choose your workspace location.

   We recommend to start working with OpenCV for Android from a new clean workspace. A new Eclipse workspace can for example be created in the folder where you have unpacked OpenCV4Android SDK package:

2. Import OpenCV library and samples into workspace.

   OpenCV library is packed as a ready-for-use Android Library Project. You can simply reference it in your projects.

   Each sample included into the `OpenCV-2.4.6-android-sdk.zip` is a regular Android project that already references OpenCV library. Follow the steps below to import OpenCV and samples into the workspace:

   Note: OpenCV samples are indeed **dependent** on OpenCV library project so don't forget to import it to your workspace as well.

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

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

- In the *Select root directory* field locate your OpenCV package folder. 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 you have to wait some time while it is building OpenCV samples. Just give a minute to Eclipse to complete initialization.

Warning: After the initial import, on a non-Windows (Linux and Mac OS) operating system Eclipse will still show build errors for applications with native C++ code. To resolve the issues, please do the following: Open *Project Properties* -> *C/C++ Build*, and replace "Build command" text to `"${NDKROOT}/ndk-build"` (remove .cmd at the end).

Note: In some cases the build errors don't disappear, then try the following actions:

- right click on `OpenCV Library` project -> *Android Tools* -> *Fix Project Properties*, then menu *Project* -> *Clean...* -> *Clean all*
- right click on the project with errors -> *Properties* -> *Android*, make sure the `Target` is selected and is `Android 3.0` or higher
- check the build errors in the *Problems* view window and try to resolve them by yourselves

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

Running OpenCV Samples

At this point you should be able to build and run the samples. Keep in mind, that `face-detection` and `Tutorial 2 - Mixed Processing` include some native code and require Android NDK and NDK/CDT plugin for Eclipse to build working applications. If you haven't installed these tools, see the corresponding section of *Introduction into Android Development*.

> **Warning:** Please consider that some samples use Android Java Camera API, which is accessible with an AVD. But most of samples use OpenCV Native Camera which **may not work** with an emulator.

Note: Recent *Android SDK tools, revision 19+* can run ARM v7a OS images but they available not for all Android versions.

Well, running samples from Eclipse is very simple:

- Connect your device with **adb** tool from Android SDK or create an emulator with camera support.
  - See Managing Virtual Devices document for help with Android Emulator.
  - See Using Hardware Devices for help with real devices (not emulators).
- Select project you want to start in *Package Explorer* and just press `Ctrl + F11` or select option *Run* → *Run* from the main menu, or click *Run* button on the toolbar.

Note: Android Emulator can take several minutes to start. So, please, be patient.

- On the first run Eclipse will ask you about the running mode for your application:

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

  Chances are that on the first launch you will not have the OpenCV Manager package installed. In this case you will see the following message:

To get rid of the message you will need to install *OpenCV Manager* and the appropriate *OpenCV binary pack*. Simply tap *Yes* if you have *Google Play Market* installed on your device/emulator. It will redirect you to the corresponding page on *Google Play Market*.

If you have no access to the *Market*, which is often the case with emulators - you will need to install the packages from OpenCV4Android SDK folder manually. See *manager_selection* for details.

```
<Android SDK path>/platform-tools/adb install <OpenCV4Android SDK path>/apk/OpenCV_2.4.6_Manager_2.9_armv7a-neon
```

Note: `armeabi`, `armv7a-neon`, `arm7a-neon-android8`, `mips` and `x86` stand for platform targets:

- `armeabi` is for ARM v5 and ARM v6 architectures with Android API 8+,
- `armv7a-neon` is for NEON-optimized ARM v7 with Android API 9+,
- `arm7a-neon-android8` is for NEON-optimized ARM v7 with Android API 8,
- `mips` is for MIPS architecture with Android API 9+,
- `x86` is for Intel x86 CPUs with Android API 9+.

If using hardware device for testing/debugging, run the following command to learn its CPU architecture:

```
adb shell getprop ro.product.cpu.abi
```

If you're using an AVD emulator, go *Window* > *AVD Manager* to see the list of availible devices. Click *Edit* in the context menu of the selected device. In the window, which then pop-ups, find the CPU field.

You may also see section *manager_selection* for details.

When done, you will be able to run OpenCV samples on your device/emulator seamlessly.

- Here is `Sample - image-manipulations` sample, running on top of stock camera-preview of the emulator.

What's next

Now, when you have your instance of OpenCV4Adroid SDK set up and configured, you may want to proceed to using OpenCV in your own application. You can learn how to do that in a separate *Android Development with OpenCV* tutorial.

1.10 Android Development with OpenCV

This tutorial has been created to help you use OpenCV library within your Android project.

This guide was written with Windows 7 in mind, though it should work with any other OS supported by OpenCV4Android SDK.

This tutorial assumes you have the following installed and configured:

- JDK
- Android SDK and NDK

- Eclipse IDE
- ADT and CDT plugins for Eclipse

If you need help with anything of the above, you may refer to our *Introduction into Android Development* guide.

This tutorial also assumes you have OpenCV4Android SDK already installed on your development machine and OpenCV Manager on your testing device correspondingly. If you need help with any of these, you may consult our *OpenCV4Android SDK* tutorial.

If you encounter any error after thoroughly following these steps, feel free to contact us via OpenCV4Android discussion group or OpenCV Q&A forum . We'll do our best to help you out.

Using OpenCV Library Within Your Android Project

In this section we will explain how to make some existing project to use OpenCV. Starting with 2.4.2 release for Android, *OpenCV Manager* is used to provide apps with the best available version of OpenCV. You can get more information here: *Android_OpenCV_Manager* and in these slides.

Java

Application Development with Async Initialization

Using async initialization is a **recommended** way for application development. It uses the OpenCV Manager to access OpenCV libraries externally installed in the target system.

1. Add OpenCV library project to your workspace. Use menu *File -> Import -> Existing project in your workspace*.

   Press *Browse* button and locate OpenCV4Android SDK (`OpenCV-2.4.6-android-sdk/sdk`).

2. In application project add a reference to the OpenCV Java SDK in *Project -> Properties -> Android -> Library -> Add* select `OpenCV Library - 2.4.6`.

In most cases OpenCV Manager may be installed automatically from Google Play. For the case, when Google Play is not available, i.e. emulator, developer board, etc, you can install it manually using adb tool. See *manager_selection* for details.

There is a very base code snippet implementing the async initialization. It shows basic principles. See the "15-puzzle" OpenCV sample for details.

```
public class Sample1Java extends Activity implements CvCameraViewListener {

    private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {
        @Override
```

```
        public void onManagerConnected(int status) {
            switch (status) {
                case LoaderCallbackInterface.SUCCESS:
                {
                    Log.i(TAG, "OpenCV loaded successfully");
                    mOpenCvCameraView.enableView();
                } break;
                default:
                {
                    super.onManagerConnected(status);
                } break;
            }
        }
    };

    @Override
    public void onResume()
    {
        super.onResume();
        OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_2_4_6, this, mLoaderCallback);
    }

    ...
}
```

It this case application works with OpenCV Manager in asynchronous fashion. `OnManagerConnected` callback will be called in UI thread, when initialization finishes. Please note, that it is not allowed to use OpenCV calls or load OpenCV-dependent native libs before invoking this callback. Load your own native libraries that depend on OpenCV after the successful OpenCV initialization. Default `BaseLoaderCallback` implementation treat application context as Activity and calls `Activity.finish()` method to exit in case of initialization failure. To override this behavior you need to override `finish()` method of `BaseLoaderCallback` class and implement your own finalization method.

Application Development with Static Initialization

According to this approach all OpenCV binaries are included into your application package. It is designed mostly for development purposes. This approach is deprecated for the production code, release package is recommended to communicate with OpenCV Manager via the async initialization described above.

1. Add the OpenCV library project to your workspace the same way as for the async initialization above. Use menu *File -> Import -> Existing project in your workspace*, press *Browse* button and select OpenCV SDK path (`OpenCV-2.4.6-android-sdk/sdk`).

2. In the application project add a reference to the OpenCV4Android SDK in *Project -> Properties -> Android -> Library -> Add* select `OpenCV Library - 2.4.6`;

3. If your application project **doesn't have a JNI part**, just copy the corresponding OpenCV native libs from `<OpenCV-2.4.6-android-sdk>/sdk/native/libs/<target_arch>` to your project directory to folder `libs/<target_arch>`.

   In case of the application project **with a JNI part**, instead of manual libraries copying you need to modify your `Android.mk` file: add the following two code lines after the `"include $(CLEAR_VARS)"` and before `"include path_to_OpenCV-2.4.6-android-sdk/sdk/native/jni/OpenCV.mk"`

```
OPENCV_CAMERA_MODULES:=on
OPENCV_INSTALL_MODULES:=on
```

   The result should look like the following:

```
include $(CLEAR_VARS)

# OpenCV
OPENCV_CAMERA_MODULES:=on
OPENCV_INSTALL_MODULES:=on
include ../../sdk/native/jni/OpenCV.mk
```

After that the OpenCV libraries will be copied to your application `libs` folder during the JNI build.v

Eclipse will automatically include all the libraries from the `libs` folder to the application package (APK).

4. The last step of enabling OpenCV in your application is Java initialization code before calling OpenCV API. It can be done, for example, in the static section of the `Activity` class:

```
static {
    if (!OpenCVLoader.initDebug()) {
        // Handle initialization error
    }
}
```

If you application includes other OpenCV-dependent native libraries you should load them **after** OpenCV initialization:

```
static {
    if (!OpenCVLoader.initDebug()) {
        // Handle initialization error
    } else {
        System.loadLibrary("my_jni_lib1");
        System.loadLibrary("my_jni_lib2");
    }
}
```

Native/C++

To build your own Android application, using OpenCV as native part, the following steps should be taken:

1. You can use an environment variable to specify the location of OpenCV package or just hardcode absolute or relative path in the `jni/Android.mk` of your projects.
2. The file `jni/Android.mk` should be written for the current application using the common rules for this file.

   For detailed information see the Android NDK documentation from the Android NDK archive, in the file `<path_where_NDK_is_placed>/docs/ANDROID-MK.html`.
3. The following line:

   ```
   include C:\Work\OpenCV4Android\OpenCV-2.4.6-android-sdk\sdk\native\jni\OpenCV.mk
   ```

   Should be inserted into the `jni/Android.mk` file **after** this line:

   ```
   include $(CLEAR_VARS)
   ```
4. Several variables can be used to customize OpenCV stuff, but you **don't need** to use them when your application uses the *async initialization* via the *OpenCV Manager* API.

   Note: These variables should be set **before** the `"include .../OpenCV.mk"` line:

   ```
   OPENCV_INSTALL_MODULES:=on
   ```

Copies necessary OpenCV dynamic libs to the project `libs` folder in order to include them into the APK.

```
OPENCV_CAMERA_MODULES:=off
```

Skip native OpenCV camera related libs copying to the project `libs` folder.

```
OPENCV_LIB_TYPE:=STATIC
```

Perform static linking with OpenCV. By default dynamic link is used and the project JNI lib depends on `libopencv_java.so`.

5. The file `Application.mk` should exist and should contain lines:

```
APP_STL := gnustl_static
APP_CPPFLAGS := -frtti -fexceptions
```

Also, the line like this one:

```
APP_ABI := armeabi-v7a
```

Should specify the application target platforms.

In some cases a linkage error (like `"In function 'cv::toUtf16(std::basic_string<...>... undefined reference to 'mbstowcs'"`) happens when building an application JNI library, depending on OpenCV. The following line in the `Application.mk` usually fixes it:

```
APP_PLATFORM := android-9
```

6. Either use *manual* `ndk-build` invocation or *setup Eclipse CDT Builder* to build native JNI lib before (re)building the Java part and creating an APK.

Hello OpenCV Sample

Here are basic steps to guide you trough the process of creating a simple OpenCV-centric application. It will be capable of accessing camera output, processing it and displaying the result.

1. Open Eclipse IDE, create a new clean workspace, create a new Android project *File → New → Android Project*
2. Set name, target, package and `minSDKVersion` accordingly. The minimal SDK version for build with OpenCV4Android SDK is 11. Minimal device API Level (for application manifest) is 8.
3. Allow Eclipse to create default activity. Lets name the activity `HelloOpenCvActivity`.
4. Choose Blank Activity with full screen layout. Lets name the layout `HelloOpenCvLayout`.
5. Import OpenCV library project to your workspace.
6. Reference OpenCV library within your project properties.

7. Edit your layout file as xml file and pass the following layout there:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:opencv="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <org.opencv.android.JavaCameraView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:visibility="gone"
        android:id="@+id/HelloOpenCvView"
        opencv:show_fps="true"
        opencv:camera_id="any" />
```

```

</LinearLayout>
```

8. Add the following permissions to the `AndroidManifest.xml` file:

```
</application>

<uses-permission android:name="android.permission.CAMERA"/>

<uses-feature android:name="android.hardware.camera" android:required="false"/>
<uses-feature android:name="android.hardware.camera.autofocus" android:required="false"/>
<uses-feature android:name="android.hardware.camera.front" android:required="false"/>
<uses-feature android:name="android.hardware.camera.front.autofocus" android:required="false"/>
```

9. Set application theme in AndroidManifest.xml to hide title and system buttons.

```
<application
    android:icon="@drawable/icon"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen" >
```

10. Add OpenCV library initialization to your activity. Fix errors by adding requited imports.

```
private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {
    @Override
    public void onManagerConnected(int status) {
        switch (status) {
            case LoaderCallbackInterface.SUCCESS:
            {
                Log.i(TAG, "OpenCV loaded successfully");
                mOpenCvCameraView.enableView();
            } break;
            default:
            {
                super.onManagerConnected(status);
            } break;
        }
    }
};

@Override
public void onResume()
{
    super.onResume();
    OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_2_4_6, this, mLoaderCallback);
}
```

11. Defines that your activity implements `CvViewFrameListener2` interface and fix activity related errors by defining missed methods. For this activity define `onCreate`, `onDestroy` and `onPause` and implement them according code snippet bellow. Fix errors by adding requited imports.

```
private CameraBridgeViewBase mOpenCvCameraView;

@Override
public void onCreate(Bundle savedInstanceState) {
    Log.i(TAG, "called onCreate");
    super.onCreate(savedInstanceState);
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    setContentView(R.layout.HelloOpenCvLayout);
```

```
        mOpenCvCameraView = (CameraBridgeViewBase) findViewById(R.id.HelloOpenCvView);
        mOpenCvCameraView.setVisibility(SurfaceView.VISIBLE);
        mOpenCvCameraView.setCvCameraViewListener(this);
    }

    @Override
    public void onPause()
    {
        super.onPause();
        if (mOpenCvCameraView != null)
            mOpenCvCameraView.disableView();
    }

    public void onDestroy() {
        super.onDestroy();
        if (mOpenCvCameraView != null)
            mOpenCvCameraView.disableView();
    }

    public void onCameraViewStarted(int width, int height) {
    }

    public void onCameraViewStopped() {
    }

    public Mat onCameraFrame(CvCameraViewFrame inputFrame) {
        return inputFrame.rgba();
    }
```

12. Run your application on device or emulator.

Lets discuss some most important steps. Every Android application with UI must implement Activity and View. By the first steps we create blank activity and default view layout. The simplest OpenCV-centric application must implement OpenCV initialization, create its own view to show preview from camera and implements `CvViewFrameListener2` interface to get frames from camera and process it.

First of all we create our application view using xml layout. Our layout consists of the only one full screen component of class `org.opencv.android.JavaCameraView`. This class is implemented inside OpenCV library. It is inherited from `CameraBridgeViewBase`, that extends `SurfaceView` and uses standard Android camera API. Alternatively you can use `org.opencv.android.NativeCameraView` class, that implements the same interface, but uses `VideoCapture` class as camera access back-end. `opencv:show_fps="true"` and `opencv:camera_id="any"` options enable FPS message and allow to use any camera on device. Application tries to use back camera first.

After creating layout we need to implement `Activity` class. OpenCV initialization process has been already discussed above. In this sample we use asynchronous initialization. Implementation of `CvCameraViewListener` interface allows you to add processing steps after frame grabbing from camera and before its rendering on screen. The most important function is `onCameraFrame`. It is callback function and it is called on retrieving frame from camera. The callback input is object of `CvCameraViewFrame` class that represents frame from camera.

Note: Do not save or use `CvCameraViewFrame` object out of `onCameraFrame` callback. This object does not have its own state and its behavior out of callback is unpredictable!

It has `rgba()` and `gray()` methods that allows to get frame as RGBA and one channel gray scale `Mat` respectively. It expects that `onCameraFrame` function returns RGBA frame that will be drawn on the screen.

1.11 Installation in iOS

Required Packages

- CMake 2.8.8 or higher
- Xcode 4.2 or higher

Getting the Cutting-edge OpenCV from Git Repository

Launch GIT client and clone OpenCV repository from here

In MacOS it can be done using the following command in Terminal:

```
cd ~/<my_working _directory>
git clone https://github.com/Itseez/opencv.git
```

Building OpenCV from Source, using CMake and Command Line

1. Make symbolic link for Xcode to let OpenCV build scripts find the compiler, header files etc.

   ```
   cd /
   sudo ln -s /Applications/Xcode.app/Contents/Developer Developer
   ```

2. Build OpenCV framework:

   ```
   cd ~/<my_working_directory>
   python opencv/platforms/ios/build_framework.py ios
   ```

If everything's fine, a few minutes later you will get ~/<my_working_directory>/ios/opencv2.framework. You can add this framework to your Xcode projects.

Further Reading

You can find several OpenCV+iOS tutorials here *OpenCV iOS*.

1.12 Cross compilation for ARM based Linux systems

This steps are tested on Ubuntu Linux 12.04, but should work for other Linux distributions. I case of other distributions package names and names of cross compilation tools may differ. There are several popular EABI versions that are used on ARM platform. This tutorial is written for *gnueabi* and *gnueabihf*, but other variants should work with minimal changes.

Prerequisites

- Host computer with Linux;
- Git;
- CMake 2.6 or higher;

- Cross compilation tools for ARM: gcc, libstc++, etc. Depending on target platform you need to choose *gnueabi* or *gnueabihf* tools. Install command for *gnueabi*:

  ```
  sudo apt-get install gcc-arm-linux-gnueabi
  ```

  Install command for *gnueabihf*:

  ```
  sudo apt-get install gcc-arm-linux-gnueabihf
  ```

- pkgconfig;
- Python 2.6 for host system;
- [optional] ffmpeg or libav development packages for armeabi(hf): libavcodec-dev, libavformat-dev, libswscale-dev;
- [optional] GTK+2.x or higher, including headers (libgtk2.0-dev) for armeabi(hf);
- [optional] libdc1394 2.x;
- [optional] libjpeg-dev, libpng-dev, libtiff-dev, libjasper-dev for armeabi(hf).

Getting OpenCV Source Code

You can use the latest stable OpenCV version available in *sourceforge* or you can grab the latest snapshot from our Git repository.

Getting the Latest Stable OpenCV Version

- Go to our page on Sourceforge;
- Download the source tarball and unpack it.

Getting the Cutting-edge OpenCV from the Git Repository

Launch Git client and clone OpenCV repository

In Linux it can be achieved with the following command in Terminal:

```
cd ~/<my_working _directory>
git clone https://github.com/Itseez/opencv.git
```

Building OpenCV

1. Create a build directory, make it current and run the following command:

   ```
   cmake [<some optional parameters>] -DCMAKE_TOOLCHAIN_FILE=<path to the OpenCV source directory>/platforms/linux/a
   ```

   Toolchain uses *gnueabihf* EABI convention by default. Add `-DSOFTFP=ON` cmake argument to switch on softfp compiler.

   ```
   cmake [<some optional parameters>] -DSOFTFP=ON -DCMAKE_TOOLCHAIN_FILE=<path to the OpenCV source directory>/plat
   ```

   For example:

```
cd ~/opencv/platforms/linux
mkdir -p build_hardfp
cd build_hardfp

cmake -DCMAKE_TOOLCHAIN_FILE=../arm-gnueabi.toolchain.cmake ../../..
```

2. Run make in build (<cmake_binary_dir>) directory:

```
make
```

Note: Optionally you can strip symbols info from the created library via install/strip make target. This option produces smaller binary (~ twice smaller) but makes further debugging harder.

Enable hardware optimizations

Depending on target platfrom architecture different instruction sets can be used. By default compiler generates code for armv5l without VFPv3 and NEON extensions. Add `-DUSE_VFPV3=ON` to cmake command line to enable code generation for VFPv3 and `-DUSE_NEON=ON` for using NEON SIMD extensions.

TBB is supported on multi core ARM SoCs also. Add `-DWITH_TBB=ON` and `-DBUILD_TBB=ON` to enable it. Cmake scripts download TBB sources from official project site http://threadingbuildingblocks.org/ and build it.

1.13 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.

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

using namespace cv;
using namespace std;

int main( int argc, char** argv )
{
    if( argc != 2)
    {
     cout <<" Usage: display_image ImageToLoadAndDisplay" << endl;
     return -1;
    }

```

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

    if(! image.data )                              // Check for invalid input
    {
        cout <<  "Could not open or find the image" << std::endl ;
        return -1;
    }

    namedWindow( "Display window", CV_WINDOW_AUTOSIZE );// Create a window for display.
    imshow( "Display window", image );                   // Show our image inside it.

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

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

```
// Video Image PSNR and SSIM
#include <iostream> // for standard I/O
#include <string>   // for strings
```

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:

1.14 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 BGR to Grayscale format by using cvtColor
- Save your transformed image in a file on disk (using imwrite)

Code

Here it is:

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

using namespace cv;

int main( int argc, char** argv )
{
 char* imageName = argv[1];

 Mat image;
 image = imread( imageName, 1 );

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

```

 Mat gray_image;
 cvtColor( image, gray_image, CV_BGR2GRAY );

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

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

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

 waitKey(0);

 return 0;
}
```

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*. Fort this example, assume you are loading a RGB image.
2. Now we are going to convert our image from BGR to Grayscale format. OpenCV has a really nice function to do this kind of transformations:

   ```
   cvtColor( image, gray_image, CV_BGR2GRAY );
   ```

   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_BGR2GRAY** (because of imread has BGR default channel order in case of color images).
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 analagous 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!

1.15 How to write a tutorial for OpenCV

Okay, so assume you have just finished a project of yours implementing something based on OpenCV and you want to present/share it with the community. Luckily, OpenCV is an *open source project*. This means that anyone has access to the full source code and may propose extensions. And a good tutorial is a valuable addition to the library! Please read instructions on contribution process here: http://opencv.org/contribute.html. You may also find this page helpful: How to contribute.

While making a robust and practical library (like OpenCV) is great, the success of a library also depends on how user friendly it is. To improve on this aspect, the OpenCV team has already been listening to user feedback at OpenCV

Q&A forum and by making samples you can find in the source directories `samples` folder. The addition of the tutorials (in both online and PDF format) is an extension of these efforts.

Goal

The tutorials are just as an important part of the library as the implementation of those crafty data structures and algorithms you can find in OpenCV. Therefore, the source codes for the tutorials are part of the library. And yes, I meant source codes. The reason for this formulation is that the tutorials are written by using the Sphinx documentation generation system. This is based on the popular Python documentation system called reStructuredText (reST). ReStructuredText is a really neat language that by using a few simple conventions (indentation, directives) and emulating old school email writing techniques (text only) tries to offer a simple way to create and edit documents. Sphinx extends this with some new features and creates the resulting document in both HTML (for web) and PDF (for offline usage) format.

Usually, an OpenCV tutorial has the following parts:

1. A source code demonstration of an OpenCV feature:
   1. One or more CPP, Python, Java or other type of files depending for what OpenCV offers support and for what language you make the tutorial.
   2. Occasionaly, input resource files required for running your tutorials application.
2. A table of content entry (so people may easily find the tutorial):
   1. Adding your stuff to the tutorials table of content (**reST** file).
   2. Add an image file near the TOC entry.
3. The content of the tutorial itself:
   1. The **reST** text of the tutorial
   2. Images following the idea that "*A picture is worth a thousand words*".
   3. For more complex demonstrations you may create a video.

As you can see you will need at least some basic knowledge of the *reST* system in order to complete the task at hand with success. However, don't worry *reST* (and *Sphinx*) was made with simplicity in mind. It is easy to grasp its basics. I found that the OpenAlea documentations introduction on this subject (or the Thomas Cokelaer one) should enough for this. If for some directive or feature you need a more in-depth description look it up in the official reStructuredText help files or at the Sphinx documentation.

In our world achieving some tasks is possible in multiple ways. However, some of the roads to take may have obvious or hidden advantages over others. Then again, in some other cases it may come down to just simple user preference. Here, I'll present how I decided to write the tutorials, based on my personal experience. If for some of them you know a better solution and you can back it up feel free to use that. I've nothing against it, as long as it gets the job done in an elegant fashion.

Now the best would be if you could make the integration yourself. For this you need first to have the source code. I recommend following the guides for your operating system on acquiring OpenCV sources. For Linux users look *here* and for *Windows here*. You must also install python and sphinx with its dependencies in order to be able to build the documentation.

Once you have downloaded the repository to your hard drive you can take a look in the OpenCV directory to make sure you have both the samples and doc folder present. Anyone may download the latest source files from `git://github.com/Itseez/opencv.git` . Nevertheless, not everyone has upload (commit/submit) rights. This is to protect the integrity of the library. If you plan doing more than one tutorial, and would like to have an account with commit user rights you should first register an account at http://code.opencv.org/ and then contact OpenCV administrator -delete-admin@-delete-opencv.org. Otherwise, you can just send the resulting files to us at -delete-admin@-delete-opencv.org and we'll add it.

Format the Source Code

Before I start this let it be clear: the main goal is to have a working sample code. However, for your tutorial to be of a top notch quality you should follow a few guide lines I am going to present here. In case you have an application by using the older interface (with *IplImage*, *cvMat*, *cvLoadImage* and such) consider migrating it to the new C++ interface. The tutorials are intended to be an up to date help for our users. And as of OpenCV 2 the OpenCV emphasis on using the less error prone and clearer C++ interface. Therefore, if possible please convert your code to the C++ interface. For this it may help to read the *Interoperability with OpenCV 1* tutorial. However, once you have an OpenCV 2 working code, then you should make your source code snippet as easy to read as possible. Here're a couple of advices for this:

- Add a standard output with the description of what your program does. Keep it short and yet, descriptive. This output is at the start of the program. In my example files this usually takes the form of a *help* function containing the output. This way both the source file viewer and application runner can see what all is about in your sample. Here's an instance of this:

  ```
  void help()
  {
  cout
  << "--------------------------------------------------------------------------"   << endl
  << "This program shows how to write video files. You can extract the R or G or B color channel "
  << " of the input video. You can choose to use the source codec (Y) or select a custom one. (N)"<< endl
  << "Usage:"                                                                        << endl
  << "./video-write inputvideoName [ R | G | B] [Y | N]"                             << endl
  << "--------------------------------------------------------------------------"   << endl
  << endl;
  }
  // ...
  int main(int argc, char *argv[], char *window_name)
  {
  help();
  // here comes the actual source code
  }
  ```

  Additionally, finalize the description with a short usage guide. This way the user will know how to call your programs, what leads us to the next point.

- Prefer command line argument controlling instead of hard coded one. If your program has some variables that may be changed use command line arguments for this. The tutorials, can be a simple try-out ground for the user. If you offer command line controlling for the input image (for example), then you offer the possibility for the user to try it out with his/her own images, without the need to mess in the source code. In the upper example you can see that the input image, channel and codec selection may all be changed from the command line. Just compile the program and run it with your own input arguments.

- Be as verbose as possible. There is no shame in filling the source code with comments. This way the more advanced user may figure out what's happening right from the sample code. This advice goes for the output console too. Specify to the user what's happening. Never leave the user hanging there and thinking on: "Is this program now crashing or just doing some computationally intensive task?." So, if you do a training task that may take some time, make sure you print out a message about this before starting and after finishing it.

- Throw out unnecessary stuff from your source code. This is a warning to not take the previous point too seriously. Balance is the key. If it's something that can be done in a fewer lines or simpler than that's the way you should do it. Nevertheless, if for some reason you have such sections notify the user why you have chosen to do so. Keep the amount of information as low as possible, while still getting the job done in an elegant way.

- Put your sample file into the `opencv/samples/cpp/tutorial_code/sectionName` folder. If you write a tutorial for other languages than cpp, then change that part of the path. Before completing this you need to decide that to what section (module) does your tutorial goes. Think about on what module relies most heavily

your code and that is the one to use. If the answer to this question is more than one modules then the *general* section is the one to use. For finding the *opencv* directory open up your file system and navigate where you downloaded our repository.

- If the input resources are hard to acquire for the end user consider adding a few of them to the `opencv/samples/cpp/tutorial_code/images`. Make sure that who reads your code can try it out!

Add the TOC entry

For this you will need to know some reStructuredText. There is no going around this. reStructuredText files have **rst** extensions. However, these are simple text files. Use any text editor you like. Finding a text editor that offers syntax highlighting for reStructuredText was quite a challenge at the time of writing this tutorial. In my experience, Intype is a solid option on Windows, although there is still place for improvement.

Adding your source code to a table of content is important for multiple reasons. First and foremost this will allow for the user base to find your tutorial from our websites tutorial table of content. Secondly, if you omit this *Sphinx* will throw a warning that your tutorial file isn't part of any TOC tree entry. And there is nothing more than the developer team hates than an ever increasing warning/error list for their builds. *Sphinx* also uses this to build up the previous-back-up buttons on the website. Finally, omitting this step will lead to that your tutorial will **not** be added to the PDF version of the tutorials.

Navigate to the `opencv/doc/tutorials/section/table_of_content_section` folder (where the section is the module to which you're adding the tutorial). Open the *table_of_content_section* file. Now this may have two forms. If no prior tutorials are present in this section that there is a template message about this and has the following form:

```
.. _Table-Of-Content-Section:

 Section title
 ------------------------------------------------------------
 Description about the section.
 .. include:: ../../definitions/noContent.rst
 .. raw:: latex
    \pagebreak
```

The first line is a reference to the section title in the reST system. The section title will be a link and you may refer to it via the `:ref:` directive. The *include* directive imports the template text from the definitions directories *noContent.rst* file. *Sphinx* does not creates the PDF from scratch. It does this by first creating a latex file. Then creates the PDF from the latex file. With the *raw* directive you can directly add to this output commands. Its unique argument is for what kind of output to add the content of the directive. For the PDFs it may happen that multiple sections will overlap on a single page. To avoid this at the end of the TOC we add a *pagebreak* latex command, that hints to the LATEX system that the next line should be on a new page.

If you have one of this, try to transform it to the following form:

```
.. _Table-Of-Content-Section:
Section title
------------------------------------------------------------
.. include:: ../../definitions/tocDefinitions.rst
+
  .. tabularcolumns:: m{100pt} m{300pt}
  .. cssclass:: toctableopencv
  =============== ======================================================
   |MatBasicIma|  **Title:** :ref:`matTheBasicImageContainer`
                  *Compatibility:* > OpenCV 2.0
                  *Author:* |Author_BernatG|
                  You will learn how to store images in the memory and how to print out their content to the console.
  =============== ======================================================
```

```
  .. |MatBasicIma| image:: images/matTheBasicImageStructure.jpg
                   :height: 90pt
                   :width:  90pt
.. raw:: latex

   \pagebreak

.. toctree::

   :hidden:
   ../mat - the basic image container/mat - the basic image container
```

If this is already present just add a new section of the content between the include and the raw directives (excluding those lines). Here you'll see a new include directive. This should be present only once in a TOC tree and the reST file contains the definitions of all the authors contributing to the OpenCV tutorials. We are a multicultural community and some of our name may contain some funky characters. However, reST **only supports** ANSI characters. Luckily we can specify Unicode characters with the *unicode* directive. Doing this for all of your tutorials is a troublesome procedure. Therefore, the tocDefinitions file contains the definition of your author name. Add it here once and afterwards just use the replace construction. For example here's the definition for my name:

```
.. |Author_BernatG| unicode:: Bern U+00E1 t U+0020 G U+00E1 bor
```

The `|Author_BernatG|` is the text definitions alias. I can use later this to add the definition, like I've done in the TOCs *Author* part. After the `::` and a space you start the definition. If you want to add an UNICODE character (non-ASCI) leave an empty space and specify it in the format U+(UNICODE code). To find the UNICODE code of a character I recommend using the FileFormat websites service. Spaces are trimmed from the definition, therefore we add a space by its UNICODE character (U+0020). Until the *raw* directive what you can see is a TOC tree entry. Here's how a TOC entry will look like:

```
.. tabularcolumns:: m{100pt} m{300pt}
.. cssclass:: toctableopencv
=============== ======================================================
 |MatBasicIma|  **Title:** :ref:'matTheBasicImageContainer'
                *Compatibility:* > OpenCV 2.0
                *Author:* |Author_BernatG|
                You will learn how to store images in the memory and how to print out their content to the console.
=============== ======================================================
.. |MatBasicIma| image:: images/matTheBasicImageStructure.jpg
                 :height: 90pt
                 :width:  90pt
```

As you can see we have an image to the left and a description box to the right. To create two boxes we use a table with two columns and a single row. In the left column is the image and in the right one the description. However, the image directive is way too long to fit in a column. Therefore, we need to use the substitution definition system. We add this definition after the TOC tree. All images for the TOC tree are to be put in the images folder near its reStructuredText file. We use the point measurement system because we are also creating PDFs. PDFs are printable documents, where there is no such thing that pixels (px), just points (pt). And while generally space is no problem for web pages (we have monitors with **huge** resolutions) the size of the paper (A4 or letter) is constant and will be for a long time in the future. Therefore, size constrains come in play more like for the PDF, than the generated HTML code.

Now your images should be as small as possible, while still offering the intended information for the user. Remember that the tutorial will become part of the OpenCV source code. If you add large images (that manifest in form of large image size) it will just increase the size of the repository pointlessly. If someone wants to download it later, its download time will be that much longer. Not to mention the larger PDF size for the tutorials and the longer load time for the web pages. In terms of pixels a TOC image should not be larger than 120 X 120 pixels. Resize your images if they are larger!

Note: If you add a larger image and specify a smaller image size, *Sphinx* will not resize that. At build time will add the full size image and the resize will be done by your browser after the image is loaded. A 120 X 120 image is somewhere below 10KB. If you add a 110KB image, you have just pointlessly added a 100KB extra data to transfer over the internet for every user!

Generally speaking you shouldn't need to specify your images size (excluding the TOC entries). If no such is found *Sphinx* will use the size of the image itself (so no resize occurs). Then again if for some reason you decide to specify a size that should be the **width** of the image rather than its height. The reason for this again goes back to the PDFs. On a PDF page the height is larger than the width. In the PDF the images will not be resized. If you specify a size that does not fit in the page, then what does not fits in **will be cut off**. When creating your images for your tutorial you should try to keep the image widths below 500 pixels, and calculate with around 400 point page width when specifying image widths.

The image format depends on the content of the image. If you have some complex scene (many random like colors) then use *jpg*. Otherwise, prefer using *png*. They are even some tools out there that optimize the size of *PNG* images, such as PNGGauntlet. Use them to make your images as small as possible in size. Now on the right side column of the table we add the information about the tutorial:

- In the first line it is the title of the tutorial. However, there is no need to specify it explicitly. We use the reference system. We'll start up our tutorial with a reference specification, just like in case of this TOC entry with its ‘‘ .. _Table-Of-Content-Section:‘‘ . If after this you have a title (pointed out by the following line of -), then Sphinx will replace the `:ref:‘Table-Of-Content-Section‘` directive with the tile of the section in reference form (creates a link in web page). Here's how the definition looks in my case:

  ```
  .. _matTheBasicImageContainer:
     Mat - The Basic Image Container
     ******************************
  ```

  Note, that according to the reStructuredText rules the * should be as long as your title.

- Compatibility. What version of OpenCV is required to run your sample code.
- Author. Use the substitution markup of reStructuredText.
- A short sentence describing the essence of your tutorial.

Now before each TOC entry you need to add the three lines of:

```
+
  .. tabularcolumns:: m{100pt} m{300pt}
  .. cssclass:: toctableopencv
```

The plus sign (+) is to enumerate tutorials by using bullet points. So for every TOC entry we have a corresponding bullet point represented by the +. Sphinx is highly indenting sensitive. Indentation is used to express from which point until to which point does a construction last. Un-indentation means end of that construction. So to keep all the bullet points to the same group the following TOC entries (until the next +) should be indented by two spaces.

Here, I should also mention that **always** prefer using spaces instead of tabs. Working with only spaces makes possible that if we both use monotype fonts we will see the same thing. Tab size is text editor dependent and as should be avoided. *Sphinx* translates all tabs into 8 spaces before interpreting it.

It turns out that the automatic formatting of both the HTML and PDF(LATEX) system messes up our tables. Therefore, we need to help them out a little. For the PDF generation we add the `.. tabularcolumns:: m{100pt} m{300pt}` directive. This means that the first column should be 100 points wide and middle aligned. For the HTML look we simply name the following table of a *toctableopencv* class type. Then, we can modify the look of the table by modifying the CSS of our web page. The CSS definitions go into the `opencv/doc/_themes/blue/static/default.css_t` file.

```
.toctableopencv
{
```

```
 width: 100% ;
 table-layout: fixed;
}
.toctableopencv colgroup col:first-child
{
 width: 100pt !important;
 max-width: 100pt !important;
 min-width: 100pt !important;
}
.toctableopencv colgroup col:nth-child(2)
{
 width: 100% !important;
}
```

However, you should not need to modify this. Just add these three lines (plus keep the two space indentation) for all TOC entries you add. At the end of the TOC file you'll find:

```
.. raw:: latex
   \pagebreak
.. toctree::
   :hidden:
   ../mat - the basic image container/mat - the basic image container
```

The page break entry comes for separating sections and should be only one in a TOC tree reStructuredText file. Finally, at the end of the TOC tree we need to add our tutorial to the *Sphinx* TOC tree system. *Sphinx* will generate from this the previous-next-up information for the HTML file and add items to the PDF according to the order here. By default this TOC tree directive generates a simple table of contents. However, we already created a fancy looking one so we no longer need this basic one. Therefore, we add the *hidden* option to do not show it.

The path is of a relative type. We step back in the file system and then go into the `mat - the basic image container` directory for the `mat - the basic image container.rst` file. Putting out the *rst* extension for the file is optional.

Write the tutorial

Create a folder with the name of your tutorial. Preferably, use small letters only. Then create a text file in this folder with *rst* extension and the same name. If you have images for the tutorial create an `images` folder and add your images there. When creating your images follow the guidelines described in the previous part!

Now here's our recommendation for the structure of the tutorial (although, remember that this is not carved in the stone; if you have a better idea, use it!):

- Create the reference point and the title.

  ```
  .. _matTheBasicImageContainer:
  Mat - The Basic Image Container
  *******************************
  ```

  You start the tutorial by specifying a reference point by the `.. _matTheBasicImageContainer:` and then its title. The name of the reference point should be a unique one over the whole documentation. Therefore, do not use general names like *tutorial1*. Use the * character to underline the title for its full width. The subtitles of the tutorial should be underlined with = charachter.

- Goals. You start your tutorial by specifying what you will present. You can also enumerate the sub jobs to be done. For this you can use a bullet point construction. There is a single configuration file for both the reference manual and the tutorial documentation. In the reference manuals at the argument enumeration we do not want any kind of bullet point style enumeration. Therefore, by default all the bullet points at this level are set to do not show the dot before the entries in the HTML. You can override this by putting the bullet point in a container.

I've defined a square type bullet point view under the name *enumeratevisibleitemswithsquare*. The CSS style definition for this is again in the `opencvdoc_themesbluestaticdefault.css_t` file. Here's a quick example of using it:

```
.. container:: enumeratevisibleitemswithsquare
   + Create the reference point and the title.
   + Second entry
   + Third entry
```

Note that you need the keep the indentation of the container directive. Directive indentations are always three (3) spaces. Here you may even give usage tips for your sample code.

- Source code. Present your samples code to the user. It's a good idea to offer a quick download link for the HTML page by using the *download* directive and pointing out where the user may find your source code in the file system by using the *file* directive:

```
Text :file:`samples/cpp/tutorial_code/highgui/video-write/` folder of the OpenCV source library
or :download:`text to appear in the webpage
<../../../../samples/cpp/tutorial_code/HighGUI/video-write/video-write.cpp>`.
```

For the download link the path is a relative one, hence the multiple back stepping operations (..). Then you can add the source code either by using the *code block* directive or the *literal include* one. In case of the code block you will need to actually add all the source code text into your reStructuredText text and also apply the required indentation:

```
.. code-block:: cpp
   int i = 0;
   l = ++j;
```

The only argument of the directive is the language used (here CPP). Then you add the source code into its content (meaning one empty line after the directive) by keeping the indentation of the directive (3 spaces). With the *literal include* directive you do not need to add the source code of the sample. You just specify the sample and *Sphinx* will load it for you, during build time. Here's an example usage:

```
.. literalinclude:: ../../../../samples/cpp/tutorial_code/HighGUI/video-write/video-write.cpp
   :language: cpp
   :linenos:
   :tab-width: 4
   :lines: 1-8, 21-22, 24-
```

After the directive you specify a relative path to the file from what to import. It has four options: the language to use, if you add the `:linenos:` the line numbers will be shown, you can specify the tab size with the `:tab-width:` and you do not need to load the whole file, you can show just the important lines. Use the *lines* option to do not show redundant information (such as the *help* function). Here basically you specify ranges, if the second range line number is missing than that means that until the end of the file. The ranges specified here do no need to be in an ascending order, need ascende

```
# ---- External links for tutorials -----------------
extlinks = {
    'hgvideo' : ('http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html#%s', None
    }
```

In short here we defined a new **hgvideo** directive that refers to an external webpage link. Its usage is:

```
A sample function of the highgui modules image write and read page is the :hgvideo:'imread() function <imread>'.
```

Which turns to: A sample function of the highgui modules image write and read page is the imread() function. The argument you give between the <> will be put in place of the `%s` in the upper definition, and as the link will anchor to the correct function. To find out the anchor of a given function just open up a web page, search for the function and click on it. In the address bar it should appear like: `http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html#imread` . Look here for the name of the directives for each page of the OpenCV reference manual. If none present for one of them feel free to add one for it. For formulas you can add LATEX code that will translate in the web pages into images. You do this by using the *math* directive. A usage tip:

```
.. math::
   MSE = \frac{1}{c*i*j} \sum{(I_1-I_2)^2}
```

That after build turns into:

$$\mathrm{MSE} = \frac{1}{c*i*j} \sum{(I_1-I_2)^2}$$

You can even use it inline as `:math:' MSE = \frac{1}{c*i*j} \sum{(I_1-I_2)^2}'` that turns into $\mathrm{MSE} = \frac{1}{c*i*j} \sum{(I_1-I_2)^2}$. If you use some crazy LATEX library extension you need to add those to the ones to use at build time. Look into the file:*opencv/doc/conf.py* configuration file for more information on this.

- Results. Well, here depending on your program show one of more of the following: - Console outputs by using the code block directive. - Output images. - Runtime videos, visualization. For this use your favorite screens capture software. Camtasia Studio certainly is one of the better choices, however their prices are out of this world. CamStudio is a free alternative, but less powerful. If you do a video you can upload it to YouTube and then use the raw directive with HTML option to embed it into the generated web page:

  ```
  You may observe a runtime instance of this on the 'YouTube here <https://www.youtube.com/watch?v=jpBwHxsl1_0

  .. raw:: html

     <div align="center">
     <iframe title="Creating a video with OpenCV" width="560" height="349" src="http://www.youtube.com/embed/j
     </div>
  ```

  This results in the text and video: You may observe a runtime instance of this on the YouTube here.

  When these aren't self-explanatory make sure to throw in a few guiding lines about what and why we can see.

- Build the documentation and check for errors or warnings. In the CMake make sure you check or pass the option for building documentation. Then simply build the **docs** project for the PDF file and the **docs_html** project for the web page. Read the output of the build and check for errors/warnings for what you have added. This is also the time to observe and correct any kind of *not so good looking* parts. Remember to keep clean our build logs.
- Read again your tutorial and check for both programming and spelling errors. If found any, please correct them.

Take home the pride and joy of a job well done!

Once you are done please make a GitHub pull request with the tutorial. Now, to see your work **live** you may need to wait some time. The PDFs are updated usually at the launch of a new OpenCV version. The web pages are a little

more diverse. They are automatically rebuilt nightly. Currently we use `2.4` and `master` branches for daily builds. So, if your pull request was merged to any of these branches, your material will be published at docs.opencv.org/2.4 or docs.opencv.org/master correspondingly. Everything that was added to `2.4` is merged to `master` branch every week. Although, we try to make a build every night, occasionally we might freeze any of the branches to fix upcoming issues. During this it may take a little longer to see your work online, however if you submitted it, be sure that eventually it will show up.

If you have any questions or advices relating to this tutorial you can contact us at -delete-admin@-delete-opencv.org (delete the -delete- parts of that email address).

CHAPTER

TWO

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: *Mask operations on matrices*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  You'll find out how to scan images with neighbor access and use the filter2D function to apply kernel filters on images.

-

  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: *Random generator and text with OpenCV*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  We will draw some *fancy-looking* stuff using OpenCV!

-

  Title: *Discrete Fourier Transform*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  You will see how and why use the Discrete Fourier transformation with OpenCV.

-

  Title: *File Input and Output using XML and YAML files*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  You will see how to use the FileStorage data structure of OpenCV to write and read data to XML or YAML file format.

-

  Title: *Interoperability with OpenCV 1*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  Did you used OpenCV before its 2.0 version? Do you wanna know what happened with your library with 2.0? Don't you know how to convert your old OpenCV programs to the new C++ interface? Look here to shed light on all this questions.

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, and magnetic resonance imaging to 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 car is nothing more than a matrix containing all the intensity values of the pixel points. How we get and store the pixels values may vary according to our needs, but in the end all images inside a computer world may be reduced to numerical matrices and other information describing the matrix itself. *OpenCV* is a computer vision library whose main focus is to process and manipulate this information. Therefore, the first thing you need to be familiar with is how OpenCV stores and handles images.

Mat

OpenCV has been around since 2001. In those days the library was built around a *C* interface and to store the image in the memory they used a C structure called *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 memory management. It builds on the assumption that the user is responsible for taking care of memory allocation and deallocation. While this is not a problem with smaller programs, once your code base grows it will be more of a struggle to handle all this rather than focusing on solving your development goal.

Luckily C++ came around and introduced the concept of classes making easier for the user through 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 2.0 introduced a new C++ interface which offered a new way of doing things which means you do not need to fiddle with memory management, making your code concise (less to write, to achieve more). The main downside of the C++ interface is that many embedded development systems at the moment support only C. Therefore, unless you are targeting embedded platforms, there's no point to 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 memory 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, which has already allocated the required space for the matrix, this will be reused. In other words we use at all times only as much memory as we need to perform the task.

Mat is basically a class with 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 (taking 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 orders of magnitude.

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. Because of this, passing 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 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 pointers point to the same address. Moreover, the copy operators **will only copy the headers** and the pointer to the large matrix, not the data itself.

```
Mat A, C;                                // creates just the header parts
A = imread(argv[1], CV_LOAD_IMAGE_COLOR); // here we'll know the method used (allocate matrix)

Mat B(A);                                // Use the copy constructor

C = A;                                   // Assignment operator
```

All the above objects, in the end, point to the same single data matrix. Their headers are different, however, and making a modification using any of them will affect all the other ones as well. 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 is that you can create headers which refer to only 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:

```
Mat D (A, Rect(10, 10, 100, 100) ); // using a rectangle
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 takes responsibility for cleaning it up when it's no longer needed. The short answer is: the last object that used it. This is handled by using a reference counting mechanism. 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. Sometimes you will want to copy the matrix itself too, so OpenCV provides the clone() and copyTo() functions.

```
Mat F = A.clone();
Mat G;
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).
- You do not need to think about memory management with OpenCVs C++ interface.
- The assignment operator and the copy constructor only copies the header.
- The underlying matrix of an image may be copied using the clone() and copyTo() functions.

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 where 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 methods to choose from. Each of them breaks it down to three or four basic components and we can use the combination of these to create the others. The most popular one is RGB, mainly because this is also how our eye builds up colors. Its base colors are red, green and blue. To code the transparency of a color sometimes a fourth element: alpha (A) is added.

There are, however, many other 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. You might, for example, dismiss the last component, making your algorithm less sensible to the 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.

Each of the building components has their own valid domains. This leads to the data type used. How we store a component defines the control we have over its domain. The smallest data type possible is *char*, which means one byte or 8 bits. This may be unsigned (so can store values from 0 to 255) or signed (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 a *Mat* object explicitly

In the *Load, Modify, and Save an Image* tutorial you have already learned 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 do this using the << operator of *Mat*. Be aware that this only works for two dimensional matrices.

Although *Mat* works really well as an 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

```
Mat M(2,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl << endl;
```

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 constructed 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 of these 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. If you need more you can create the type with the upper macro, setting the channel number in parenthesis as you can see below.

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

```
int sz[3] = {2,2,2};
Mat L(3,sz, CV_8UC(1), Scalar::all(0));
```

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.create(4,4, CV_8UC(2));
cout << "M = "<< endl << " "  << M << endl << endl;
```

```
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:

```
Mat E = Mat::eye(4, 4, CV_64F);
cout << "E = " << endl << " " << E << endl << endl;

Mat O = Mat::ones(2, 2, CV_32F);
cout << "O = " << endl << " " << O << endl << endl;

Mat Z = Mat::zeros(3,3, CV_8UC1);
cout << "Z = " << endl << " " << Z << endl << endl;
```

- For small matrices you may use comma separated initializers:

```
Mat C = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
cout << "C = " << endl << " " << C << endl << endl;
```

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

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

```
Mat RowClone = C.row(1).clone();
cout << "RowClone = " << endl << " " << RowClone << endl << endl;
```

Note: You can fill out a matrix with random values using the randu() function. You need to give the lower and upper value for the random values:

```
Mat R = Mat(3, 2, CV_8UC3);
randu(R, Scalar::all(0), Scalar::all(255));
```

Output formatting

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

- Default

```
cout << "R (default) = " << endl <<        R           << endl << endl;
```

- Python

```
cout << "R (python)  = " << endl << format(R,"python") << endl << endl;
```

- Comma separated values (CSV)

```
cout << "R (csv)     = " << endl << format(R,"csv"   ) << endl << endl;
```

- Numpy

```
cout << "R (numpy)   = " << endl << format(R,"numpy" ) << endl << endl;
```

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

- C

```
cout << "R (c)       = " << endl << format(R,"C"     ) << endl << endl;
```

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

Output of other common items

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

- 2D Point

```
Point2f P(5, 1);
cout << "Point (2D) = " << P << endl << endl;
```

```
Point (2D) = [5, 1]
```

- 3D Point

```
Point3f P3f(2, 6, 7);
cout << "Point (3D) = " << P3f << endl << endl;
```

```
Point (3D) = [2, 6, 7]
```

- std::vector via cv::Mat

```
vector<float> v;
v.push_back( (float)CV_PI);   v.push_back(2);    v.push_back(3.01f);

cout << "Vector of floats via Mat = " << Mat(v) << endl << endl;
```

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

- std::vector of points

```
vector<Point2f> vPoints(20);
for (size_t i = 0; i < vPoints.size(); ++i)
    vPoints[i] = Point2f((float)(i * 5), (float)(i % 7));

cout << "A vector of 2D Points = " << vPoints << endl << endl;
```

```
A vector of 2D Points = [0, 0; 5, 1; 10, 2; 15, 3; 20, 4; 25, 5; 30, 6; 35, 0; 4
0, 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 in a small console application. You can download it from here or in the core section of the cpp samples.

You can also find a quick video demonstration of this 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:

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

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.

```
int divideWith = 0; // convert our input string to number - C++ style
stringstream s;
s << argv[2];
s >> divideWith;
if (!s || !divideWith)
{
    cout << "Invalid number entered for dividing. " << endl;
    return -1;
}

uchar table[256];
for (int i = 0; i < 256; ++i)
   table[i] = (uchar)(divideWith * (i/divideWith));
```

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.

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).

```
Mat& ScanImageAndReduceIterator(Mat& I, const uchar* const table)
{
    // accept only char type matrices
    CV_Assert(I.depth() != sizeof(uchar));

    const int channels = I.channels();
    switch(channels)
    {
    case 1:
        {
            MatIterator_<uchar> it, end;
            for( it = I.begin<uchar>(), end = I.end<uchar>(); it != end; ++it)
                *it = table[*it];
            break;
        }
    case 3:
        {
            MatIterator_<Vec3b> it, end;
            for( it = I.begin<Vec3b>(), end = I.end<Vec3b>(); it != end; ++it)
            {
                (*it)[0] = table[(*it)[0]];
                (*it)[1] = table[(*it)[1]];
                (*it)[2] = table[(*it)[2]];
            }
        }
    }

    return I;
}
```

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.

On-the-fly address calculation with reference returning

The final method isn't recommended for scanning. It was made to acquire or modify somehow 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):

```
Mat& ScanImageAndReduceRandomAccess(Mat& I, const uchar* const table)
{
    // accept only char type matrices
    CV_Assert(I.depth() != sizeof(uchar));

    const int channels = I.channels();
    switch(channels)
    {
    case 1:
```

```
        {
            for( int i = 0; i < I.rows; ++i)
                for( int j = 0; j < I.cols; ++j )
                    I.at<uchar>(i,j) = table[I.at<uchar>(i,j)];
            break;
        }
    case 3:
        {
         Mat_<Vec3b> _I = I;

         for( int i = 0; i < I.rows; ++i)
            for( int j = 0; j < I.cols; ++j )
               {
                   _I(i,j)[0] = table[_I(i,j)[0]];
                   _I(i,j)[1] = table[_I(i,j)[1]];
                   _I(i,j)[2] = table[_I(i,j)[2]];
            }
         I = _I;
         break;
        }
    }

    return I;
}
```

The functions takes your input type and coordinates and calculates on the fly the address of the queried item. Then returns a reference to that. This may be a constant when you *get* the value and non-constant when you *set* the value. As a safety step in **debug mode only*** there is performed a check that your input coordinates are valid and does exist. If this isn't the case you'll get a nice output message of this on the standard error output stream. Compared to the efficient way in release mode the only difference in using this is that for every element of the image you'll get a new row pointer for what we use the C operator[] to acquire the column element.

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:

```
Mat lookUpTable(1, 256, CV_8U);
uchar* p = lookUpTable.data;
for( int i = 0; i < 256; ++i)
    p[i] = table[i];
```

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

```
    LUT(I, lookUpTable, J);
```

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
On-The-Fly RA	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 on-the-fly reference access method for full image scan is the most costly in debug mode. In the release mode it may beat the iterator approach or not, however it surely sacrifices for this the safety trait of iterators.

Finally, you may watch a sample run of the program on the video posted on our YouTube channel.

2.3 Mask operations on matrices

Mask operations on matrices are quite simple. The idea is that we recalculate each pixels value in an image according to a mask matrix (also known as kernel). This mask holds values that will adjust how much influence neighboring pixels (and the current pixel) have on 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:

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

$$\iff I(i,j) * M, \text{where } M = \begin{array}{c|ccc} {}_{i}\backslash^{j} & -1 & 0 & +1 \\ \hline -1 & & & \\ 0 & & & \\ +1 & & & \end{array} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

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 latter notation is a lot easier to look over.

Now let us see how we can make this happen by using the basic pixel access method or by using the filter2D function.

The Basic Method

Here's a function that will do this:

```
void Sharpen(const Mat& myImage,Mat& Result)
{
    CV_Assert(myImage.depth() == CV_8U);  // accept only uchar images
```

```
    Result.create(myImage.size(),myImage.type());
    const int nChannels = myImage.channels();

    for(int j = 1 ; j < myImage.rows-1; ++j)
    {
        const uchar* previous = myImage.ptr<uchar>(j - 1);
        const uchar* current  = myImage.ptr<uchar>(j    );
        const uchar* next     = myImage.ptr<uchar>(j + 1);

        uchar* output = Result.ptr<uchar>(j);

        for(int i= nChannels;i < nChannels*(myImage.cols-1); ++i)
        {
            *output++ = saturate_cast<uchar>(5*current[i]
                         -current[i-nChannels] - current[i+nChannels] - previous[i] - next[i]);
        }
    }

    Result.row(0).setTo(Scalar(0));
    Result.row(Result.rows-1).setTo(Scalar(0));
    Result.col(0).setTo(Scalar(0));
    Result.col(Result.cols-1).setTo(Scalar(0));
}
```

At first we make sure that the input images data is in unsigned char format. For this we use the CV_Assert function that throws an error when the expression inside it is false.

```
CV_Assert(myImage.depth() == CV_8U);  // accept only uchar images
```

We create an output image with the same size and the same type as our input. As you can see in the *How the image matrix is stored in the memory?* section, depending on the number of channels we may have one or more subcolumns. We will iterate through them via pointers so the total number of elements depends from this number.

```
Result.create(myImage.size(),myImage.type());
const int nChannels = myImage.channels();
```

We'll use the plain C [] operator to access pixels. Because we need to access multiple rows at the same time we'll acquire the pointers for each of them (a previous, a current and a next line). We need another pointer to where we're going to save the calculation. Then simply access the right items with the [] operator. For moving the output pointer ahead we simply increase this (with one byte) after each operation:

```
for(int j = 1 ; j < myImage.rows-1; ++j)
{
    const uchar* previous = myImage.ptr<uchar>(j - 1);
    const uchar* current  = myImage.ptr<uchar>(j    );
    const uchar* next     = myImage.ptr<uchar>(j + 1);

    uchar* output = Result.ptr<uchar>(j);

    for(int i= nChannels;i < nChannels*(myImage.cols-1); ++i)
    {
        *output++ = saturate_cast<uchar>(5*current[i]
                     -current[i-nChannels] - current[i+nChannels] - previous[i] - next[i]);
    }
}
```

On the borders of the image the upper notation results inexistent pixel locations (like minus one - minus one). In these points our formula is undefined. A simple solution is to not apply the mask in these points and, for example, set the

pixels on the borders to zeros:

```
Result.row(0).setTo(Scalar(0));             // The top row
Result.row(Result.rows-1).setTo(Scalar(0)); // The bottom row
Result.col(0).setTo(Scalar(0));             // The left column
Result.col(Result.cols-1).setTo(Scalar(0)); // The right column
```

The filter2D function

Applying such filters are so common in image processing that in OpenCV there exist a function that will take care of applying the mask (also called a kernel in some places). For this you first need to define a *Mat* object that holds the mask:

```
Mat kern = (Mat_<char>(3,3) <<  0, -1,  0,
                               -1,  5, -1,
                                0, -1,  0);
```

Then call the filter2D function specifying the input, the output image and the kernell to use:

```
filter2D(I, K, I.depth(), kern );
```

The function even has a fifth optional argument to specify the center of the kernel, and a sixth one for determining what to do in the regions where the operation is undefined (borders). Using this function has the advantage that it's shorter, less verbose and because there are some optimization techniques implemented it is usually faster than the *hand-coded method*. For example in my test while the second one took only 13 milliseconds the first took around 31 milliseconds. Quite some difference.

For example:

You can download this source code from `here` or look in the OpenCV source code libraries sample directory at `samples/cpp/tutorial_code/core/mat_mask_operations/mat_mask_operations.cpp`.

Check out an instance of running the program on our YouTube channel .

2.4 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

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

Code

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

```
#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 if it is between 0 and 1
 if( input >= 0.0 && input <= 1.0 )
   { 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, γ 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

2.5 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

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

```
#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 α and β 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 α 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:

2.6 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 BGR
  - 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 BGR 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 BGR 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:

2.7 Random generator and text with OpenCV

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 (*Basic Drawing*) 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**:

```
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 radombly 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 impredictable, 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!

5. The explanation above applies for the other functions generating circles, ellipses, polygones, etc. The parameters such as *center* and *vertices* are also generated randomly.
6. 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 lineType = 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), lineType);
```

```
    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), lineType);
```

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 (analagously 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 lineType = 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 *foor* loop:

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

So, **image2** is the substraction of **image** and **Scalar::all(i)**. In fact, what happens here is that every pixel of **image2** will be the result of substracting 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 substraction 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):

2.8 Discrete Fourier Transform

Goal

We'll seek answers for the following questions:

- What is a Fourier transform and why use it?

- How to do it in OpenCV?
- Usage of functions such as: copyMakeBorder(), merge(), dft(), getOptimalDFTSize(), log() and normalize() .

Source code

You can download this from here or find it in the samples/cpp/tutorial_code/core/discrete_fourier_transform/discret
of the OpenCV source code library.

Here's a sample usage of dft() :

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
int main(int argc, char ** argv)
{
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";

    Mat I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
    if( I.empty())
        return -1;

    Mat padded;                            //expand input image to optimal size
    int m = getOptimalDFTSize( I.rows );
    int n = getOptimalDFTSize( I.cols ); // on the border add zero values
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));

    Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
    Mat complexI;
    merge(planes, 2, complexI);         // Add to the expanded another plane with zeros

    dft(complexI, complexI);            // this way the result may fit in the source matrix

    // compute the magnitude and switch to logarithmic scale
    // => log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
    split(complexI, planes);                   // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
    magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
    Mat magI = planes[0];

    magI += Scalar::all(1);                    // switch to logarithmic scale
    log(magI, magI);

    // crop the spectrum, if it has an odd number of rows or columns
    magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));

    // rearrange the quadrants of Fourier image  so that the origin is at the image center
    int cx = magI.cols/2;
    int cy = magI.rows/2;

    Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
    Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
    Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
    Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right

    Mat tmp;                           // swap quadrants (Top-Left with Bottom-Right)
    q0.copyTo(tmp);
    q3.copyTo(q0);
```

```
    tmp.copyTo(q3);

    q1.copyTo(tmp);                    // swap quadrant (Top-Right with Bottom-Left)
    q2.copyTo(q1);
    tmp.copyTo(q2);

    normalize(magI, magI, 0, 1, CV_MINMAX); // Transform the matrix with float values into a
                                             // viewable image form (float between values 0 and 1).

    imshow("Input Image"       , I   );    // Show the result
    imshow("spectrum magnitude", magI);
    waitKey();

    return 0;
}
```

Explanation

The Fourier Transform will decompose an image into its sinus and cosines components. In other words, it will transform an image from its spatial domain to its frequency domain. The idea is that any function may be approximated exactly with the sum of infinite sinus and cosines functions. The Fourier Transform is a way how to do this. Mathematically a two dimensional images Fourier transform is:

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

$$e^{ix} = \cos x + i \sin x$$

Here f is the image value in its spatial domain and F in its frequency domain. The result of the transformation is complex numbers. Displaying this is possible either via a *real* image and a *complex* image or via a *magnitude* and a *phase* image. However, throughout the image processing algorithms only the *magnitude* image is interesting as this contains all the information we need about the images geometric structure. Nevertheless, if you intend to make some modifications of the image in these forms and then you need to retransform it you'll need to preserve both of these.

In this sample I'll show how to calculate and show the *magnitude* image of a Fourier Transform. In case of digital images are discrete. This means they may take up a value from a given domain value. For example in a basic gray scale image values usually are between zero and 255. Therefore the Fourier Transform too needs to be of a discrete type resulting in a Discrete Fourier Transform (*DFT*). You'll want to use this whenever you need to determine the structure of an image from a geometrical point of view. Here are the steps to follow (in case of a gray scale input image *I*):

1. **Expand the image to an optimal size**. The performance of a DFT is dependent of the image size. It tends to be the fastest for image sizes that are multiple of the numbers two, three and five. Therefore, to achieve maximal performance it is generally a good idea to pad border values to the image to get a size with such traits. The getOptimalDFTSize() returns this optimal size and we can use the copyMakeBorder() function to expand the borders of an image:

   ```
   Mat padded;                            //expand input image to optimal size
   int m = getOptimalDFTSize( I.rows );
   int n = getOptimalDFTSize( I.cols ); // on the border add zero pixels
   copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));
   ```

   The appended pixels are initialized with zero.

2. **Make place for both the complex and the real values**. The result of a Fourier Transform is complex. This implies that for each image value the result is two image values (one per component). Moreover, the frequency

domains range is much larger than its spatial counterpart. Therefore, we store these usually at least in a *float* format. Therefore we'll convert our input image to this type and expand it with another channel to hold the complex values:

```
Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
Mat complexI;
merge(planes, 2, complexI);         // Add to the expanded another plane with zeros
```

3. **Make the Discrete Fourier Transform**. It's possible an in-place calculation (same input as output):

```
dft(complexI, complexI);            // this way the result may fit in the source matrix
```

4. **Transform the real and complex values to magnitude**. A complex number has a real (Re) and a complex (imaginary - Im) part. The results of a DFT are complex numbers. The magnitude of a DFT is:

$$M = \sqrt[2]{\mathrm{Re}(\mathrm{DFT}(I))^2 + \mathrm{Im}(\mathrm{DFT}(I))^2}$$

Translated to OpenCV code:

```
split(complexI, planes);                   // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
Mat magI = planes[0];
```

5. **Switch to a logarithmic scale**. It turns out that the dynamic range of the Fourier coefficients is too large to be displayed on the screen. We have some small and some high changing values that we can't observe like this. Therefore the high values will all turn out as white points, while the small ones as black. To use the gray scale values to for visualization we can transform our linear scale to a logarithmic one:

$$M_1 = \log(1 + M)$$

Translated to OpenCV code:

```
magI += Scalar::all(1);                    // switch to logarithmic scale
log(magI, magI);
```

6. **Crop and rearrange**. Remember, that at the first step, we expanded the image? Well, it's time to throw away the newly introduced values. For visualization purposes we may also rearrange the quadrants of the result, so that the origin (zero, zero) corresponds with the image center.

```
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));
int cx = magI.cols/2;
int cy = magI.rows/2;

Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right

Mat tmp;                            // swap quadrants (Top-Left with Bottom-Right)
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);

q1.copyTo(tmp);                     // swap quadrant (Top-Right with Bottom-Left)
q2.copyTo(q1);
tmp.copyTo(q2);
```

7. **Normalize**. This is done again for visualization purposes. We now have the magnitudes, however this are still out of our image display range of zero to one. We normalize our values to this range using the normalize() function.

```
normalize(magI, magI, 0, 1, CV_MINMAX); // Transform the matrix with float values into a
                                         // viewable image form (float between values 0 and 1).
```

Result

An application idea would be to determine the geometrical orientation present in the image. For example, let us find out if a text is horizontal or not? Looking at some text you'll notice that the text lines sort of form also horizontal lines and the letters form sort of vertical lines. These two main components of a text snippet may be also seen in case of the Fourier transform. Let us use `this horizontal` and `this rotated` image about a text.

In case of the horizontal text:

In case of a rotated text:

You can see that the most influential components of the frequency domain (brightest dots on the magnitude image) follow the geometric rotation of objects on the image. From this we may calculate the offset and perform an image rotation to correct eventual miss alignments.

2.9 File Input and Output using XML and YAML files

Goal

You'll find answers for the following questions:

- How to print and read text entries to a file and OpenCV using YAML or XML files?
- How to do the same for OpenCV data structures?
- How to do this for your data structures?
- Usage of OpenCV data structures such as FileStorage, FileNode or FileNodeIterator.

Source code

You can download this from here or find it in the samples/cpp/tutorial_code/core/file_input_output/file_input_outpu
of the OpenCV source code library.

Here's a sample code of how to achieve all the stuff enumerated at the goal list.

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

using namespace cv;
using namespace std;

class MyData
{
public:
    MyData() : A(0), X(0), id()
    {}
    explicit MyData(int) : A(97), X(CV_PI), id("mydata1234") // explicit to avoid implicit conversion
    {}
    void write(FileStorage& fs) const                        //Write serialization for this class
    {
        fs << "{" << "A" << A << "X" << X << "id" << id << "}";
    }
    void read(const FileNode& node)                          //Read serialization for this class
    {
        A = (int)node["A"];
        X = (double)node["X"];
        id = (string)node["id"];
    }
public:   // Data Members
    int A;
    double X;
    string id;
};

//These write and read functions must be defined for the serialization in FileStorage to work
static void write(FileStorage& fs, const std::string&, const MyData& x)
{
    x.write(fs);
}
static void read(const FileNode& node, MyData& x, const MyData& default_value = MyData()){
    if(node.empty())
        x = default_value;
    else
        x.read(node);
}

// This function will print our custom class to the console
static ostream& operator<<(ostream& out, const MyData& m)
{
    out << "{ id = " << m.id << ", ";
    out << "X = " << m.X << ", ";
    out << "A = " << m.A << "}";
    return out;
}

int main(int ac, char** av)
```

```
{
    if (ac != 2)
    {
        help(av);
        return 1;
    }

    string filename = av[1];
    { //write
        Mat R = Mat_<uchar>::eye(3, 3),
            T = Mat_<double>::zeros(3, 1);
        MyData m(1);

        FileStorage fs(filename, FileStorage::WRITE);

        fs << "iterationNr" << 100;
        fs << "strings" << "[";                              // text - string sequence
        fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
        fs << "]";                                           // close sequence

        fs << "Mapping";                              // text - mapping
        fs << "{" << "One" << 1;
        fs <<        "Two" << 2 << "}";

        fs << "R" << R;                                      // cv::Mat
        fs << "T" << T;

        fs << "MyData" << m;                                // your own data structures

        fs.release();                                       // explicit close
        cout << "Write Done." << endl;
    }

    {//read
        cout << endl << "Reading: " << endl;
        FileStorage fs;
        fs.open(filename, FileStorage::READ);

        int itNr;
        //fs["iterationNr"] >> itNr;
        itNr = (int) fs["iterationNr"];
        cout << itNr;
        if (!fs.isOpened())
        {
            cerr << "Failed to open " << filename << endl;
            help(av);
            return 1;
        }

        FileNode n = fs["strings"];                         // Read string sequence - Get node
        if (n.type() != FileNode::SEQ)
        {
            cerr << "strings is not a sequence! FAIL" << endl;
            return 1;
        }

        FileNodeIterator it = n.begin(), it_end = n.end(); // Go through the node
        for (; it != it_end; ++it)
```

```
            cout << (string)*it << endl;


        n = fs["Mapping"];                              // Read mappings from a sequence
        cout << "Two  " << (int)(n["Two"]) << "; ";
        cout << "One  " << (int)(n["One"]) << endl << endl;


        MyData m;
        Mat R, T;

        fs["R"] >> R;                                   // Read cv::Mat
        fs["T"] >> T;
        fs["MyData"] >> m;                              // Read your own structure_

        cout << endl
            << "R = " << R << endl;
        cout << "T = " << T << endl << endl;
        cout << "MyData = " << endl << m << endl << endl;

        //Show default behavior for non existing nodes
        cout << "Attempt to read NonExisting (should initialize the data structure with its default).";
        fs["NonExisting"] >> m;
        cout << endl << "NonExisting = " << endl << m << endl;
    }

    cout << endl
        << "Tip: Open up " << filename << " with a text editor to see the serialized data." << endl;

    return 0;
}
```

Explanation

Here we talk only about XML and YAML file inputs. Your output (and its respective input) file may have only one of these extensions and the structure coming from this. They are two kinds of data structures you may serialize: *mappings* (like the STL map) and *element sequence* (like the STL vector>. The difference between these is that in a map every element has a unique name through what you may access it. For sequences you need to go through them to query a specific item.

1. **XML\YAML File Open and Close.** Before you write any content to such file you need to open it and at the end to close it. The XMLYAML data structure in OpenCV is FileStorage. To specify that this structure to which file binds on your hard drive you can use either its constructor or the *open()* function of this:

   ```
   string filename = "I.xml";
   FileStorage fs(filename, FileStorage::WRITE);
   \\...
   fs.open(filename, FileStorage::READ);
   ```

   Either one of this you use the second argument is a constant specifying the type of operations you'll be able to on them: WRITE, READ or APPEND. The extension specified in the file name also determinates the output format that will be used. The output may be even compressed if you specify an extension such as *.xml.gz*.

   The file automatically closes when the FileStorage objects is destroyed. However, you may explicitly call for this by using the *release* function:

```
fs.release();                                       // explicit close
```

2. **Input and Output of text and numbers.** The data structure uses the same << output operator that the STL library. For outputting any type of data structure we need first to specify its name. We do this by just simply printing out the name of this. For basic types you may follow this with the print of the value :

```
fs << "iterationNr" << 100;
```

Reading in is a simple addressing (via the [] operator) and casting operation or a read via the >> operator :

```
int itNr;
fs["iterationNr"] >> itNr;
itNr = (int) fs["iterationNr"];
```

3. **Input\Output of OpenCV Data structures.** Well these behave exactly just as the basic C++ types:

```
Mat R = Mat_<uchar >::eye  (3, 3),
    T = Mat_<double>::zeros(3, 1);

fs << "R" << R;                                      // Write cv::Mat
fs << "T" << T;

fs["R"] >> R;                                      // Read cv::Mat
fs["T"] >> T;
```

4. **Input\Output of vectors (arrays) and associative maps.** As I mentioned beforehand we can output maps and sequences (array, vector) too. Again we first print the name of the variable and then we have to specify if our output is either a sequence or map.

For sequence before the first element print the "[" character and after the last one the "]" character:

```
fs << "strings" << "[";                              // text - string sequence
fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
fs << "]";                                            // close sequence
```

For maps the drill is the same however now we use the "{" and "}" delimiter characters:

```
fs << "Mapping";                              // text - mapping
fs << "{" << "One" << 1;
fs <<        "Two" << 2 << "}";
```

To read from these we use the FileNode and the FileNodeIterator data structures. The [] operator of the FileStorage class returns a FileNode data type. If the node is sequential we can use the FileNodeIterator to iterate through the items:

```
FileNode n = fs["strings"];                         // Read string sequence - Get node
if (n.type() != FileNode::SEQ)
{
    cerr << "strings is not a sequence! FAIL" << endl;
    return 1;
}

FileNodeIterator it = n.begin(), it_end = n.end(); // Go through the node
for (; it != it_end; ++it)
    cout << (string)*it << endl;
```

For maps you can use the [] operator again to acces the given item (or the >> operator too):

```
n = fs["Mapping"];                                // Read mappings from a sequence
cout << "Two  " << (int)(n["Two"]) << "; ";
cout << "One  " << (int)(n["One"]) << endl << endl;
```

5. **Read and write your own data structures.** Suppose you have a data structure such as:

```
class MyData
{
public:
      MyData() : A(0), X(0), id() {}
public:   // Data Members
   int A;
   double X;
   string id;
};
```

It's possible to serialize this through the OpenCV I/O XML/YAML interface (just as in case of the OpenCV data structures) by adding a read and a write function inside and outside of your class. For the inside part:

```
void write(FileStorage& fs) const                        //Write serialization for this class
{
  fs << "{" << "A" << A << "X" << X << "id" << id << "}";
}

void read(const FileNode& node)                          //Read serialization for this class
{
  A = (int)node["A"];
  X = (double)node["X"];
  id = (string)node["id"];
}
```

Then you need to add the following functions definitions outside the class:

```
void write(FileStorage& fs, const std::string&, const MyData& x)
{
x.write(fs);
}

void read(const FileNode& node, MyData& x, const MyData& default_value = MyData())
{
if(node.empty())
    x = default_value;
else
    x.read(node);
}
```

Here you can observe that in the read section we defined what happens if the user tries to read a non-existing node. In this case we just return the default initialization value, however a more verbose solution would be to return for instance a minus one value for an object ID.

Once you added these four functions use the >> operator for write and the << operator for read:

```
MyData m(1);
fs << "MyData" << m;                                // your own data structures
fs["MyData"] >> m;                                 // Read your own structure_
```

Or to try out reading a non-existing read:

```
fs["NonExisting"] >> m;   // Do not add a fs << "NonExisting" << m command for this to work
cout << endl << "NonExisting = " << endl << m << endl;
```

Result

Well mostly we just print out the defined numbers. On the screen of your console you could see:

```
Write Done.

Reading:
100image1.jpg
Awesomeness
baboon.jpg
Two  2; One  1


R = [1, 0, 0;
  0, 1, 0;
  0, 0, 1]
T = [0; 0; 0]

MyData =
{ id = mydata1234, X = 3.14159, A = 97}

Attempt to read NonExisting (should initialize the data structure with its default).
NonExisting =
{ id = , X = 0, A = 0}

Tip: Open up output.xml with a text editor to see the serialized data.
```

Nevertheless, it's much more interesting what you may see in the output xml file:

```
<?xml version="1.0"?>
<opencv_storage>
<iterationNr>100</iterationNr>
<strings>
  image1.jpg Awesomeness baboon.jpg</strings>
<Mapping>
  <One>1</One>
  <Two>2</Two></Mapping>
<R type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>u</dt>
  <data>
    1 0 0 0 1 0 0 0 1</data></R>
<T type_id="opencv-matrix">
  <rows>3</rows>
  <cols>1</cols>
  <dt>d</dt>
  <data>
    0. 0. 0.</data></T>
<MyData>
  <A>97</A>
  <X>3.1415926535897931e+000</X>
  <id>mydata1234</id></MyData>
</opencv_storage>
```

Or the YAML file:

```
%YAML:1.0
iterationNr: 100
```

```
strings:
   - "image1.jpg"
   - Awesomeness
   - "baboon.jpg"
Mapping:
   One: 1
   Two: 2
R: !!opencv-matrix
   rows: 3
   cols: 3
   dt: u
   data: [ 1, 0, 0, 0, 1, 0, 0, 0, 1 ]
T: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [ 0., 0., 0. ]
MyData:
   A: 97
   X: 3.1415926535897931e+000
   id: mydata1234
```

You may observe a runtime instance of this on the YouTube here .

2.10 Interoperability with OpenCV 1

Goal

For the OpenCV developer team it's important to constantly improve the library. We are constantly thinking about methods that will ease your work process, while still maintain the libraries flexibility. The new C++ interface is a development of us that serves this goal. Nevertheless, backward compatibility remains important. We do not want to break your code written for earlier version of the OpenCV library. Therefore, we made sure that we add some functions that deal with this. In the following you'll learn:

- What changed with the version 2 of OpenCV in the way you use the library compared to its first version
- How to add some Gaussian noise to an image
- What are lookup tables and why use them?

General

When making the switch you first need to learn some about the new data structure for images: *Mat - The Basic Image Container*, this replaces the old *CvMat* and *IplImage* ones. Switching to the new functions is easier. You just need to remember a couple of new things.

OpenCV 2 received reorganization. No longer are all the functions crammed into a single library. We have many modules, each of them containing data structures and functions relevant to certain tasks. This way you do not need to ship a large library if you use just a subset of OpenCV. This means that you should also include only those headers you will use. For example:

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

All the OpenCV related stuff is put into the *cv* namespace to avoid name conflicts with other libraries data structures and functions. Therefore, either you need to prepend the *cv::* keyword before everything that comes from OpenCV or after the includes, you just add a directive to use this:

```
using namespace cv;  // The new C++ interface API is inside this namespace. Import it.
```

Because the functions are already in a namespace there is no need for them to contain the *cv* prefix in their name. As such all the new C++ compatible functions don't have this and they follow the camel case naming rule. This means the first letter is small (unless it's a name, like Canny) and the subsequent words start with a capital letter (like *copyMakeBorder*).

Now, remember that you need to link to your application all the modules you use, and in case you are on Windows using the *DLL* system you will need to add, again, to the path all the binaries. For more in-depth information if you're on Windows read *How to build applications with OpenCV inside the Microsoft Visual Studio* and for Linux an example usage is explained in *Using OpenCV with Eclipse (plugin CDT)*.

Now for converting the *Mat* object you can use either the *IplImage* or the *CvMat* operators. While in the C interface you used to work with pointers here it's no longer the case. In the C++ interface we have mostly *Mat* objects. These objects may be freely converted to both *IplImage* and *CvMat* with simple assignment. For example:

```
Mat I;
IplImage pI = I;
CvMat    mI = I;
```

Now if you want pointers the conversion gets just a little more complicated. The compilers can no longer automatically determinate what you want and as you need to explicitly specify your goal. This is to call the *IplImage* and *CvMat* operators and then get their pointers. For getting the pointer we use the & sign:

```
Mat I;
IplImage* pI     = &I.operator IplImage();
CvMat* mI        =  &I.operator CvMat();
```

One of the biggest complaints of the C interface is that it leaves all the memory management to you. You need to figure out when it is safe to release your unused objects and make sure you do so before the program finishes or you could have troublesome memory leeks. To work around this issue in OpenCV there is introduced a sort of smart pointer. This will automatically release the object when it's no longer in use. To use this declare the pointers as a specialization of the *Ptr* :

```
Ptr<IplImage> piI = &I.operator IplImage();
```

Converting from the C data structures to the *Mat* is done by passing these inside its constructor. For example:

```
Mat K(piL), L;
L = Mat(pI);
```

A case study

Now that you have the basics done `here's` an example that mixes the usage of the C interface with the C++ one. You will also find it in the sample directory of the OpenCV source code library at the `samples/cpp/tutorial_code/core/interoperability_with_OpenCV_1/interoperability_with_OpenCV_1.cpp` . To further help on seeing the difference the programs supports two modes: one mixed C and C++ and one pure C++. If you define the *DEMO_MIXED_API_USE* you'll end up using the first. The program separates the color planes, does some modifications on them and in the end merge them back together.

```
#include <stdio.h>
#include <iostream>

```

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

using namespace cv;  // The new C++ interface API is inside this namespace. Import it.
using namespace std;
#define DEMO_MIXED_API_USE

int main( int argc, char** argv )
{
    const char* imagename = argc > 1 ? argv[1] : "lena.jpg";

#ifdef DEMO_MIXED_API_USE
    Ptr<IplImage> IplI = cvLoadImage(imagename);      // Ptr<T> is safe ref-counting pointer class
    if(IplI.empty())
    {
        cerr << "Can not load image " <<  imagename << endl;
        return -1;
    }
    Mat I(IplI); // Convert to the new style container. Only header created. Image not copied.
#else
    Mat I = imread(imagename);        // the newer cvLoadImage alternative, MATLAB-style function
    if( I.empty() )                   // same as if( !I.data )
    {
        cerr << "Can not load image " <<  imagename << endl;
        return -1;
    }
#endif
```

Here you can observe that with the new structure we have no pointer problems, although it is possible to use the old functions and in the end just transform the result to a *Mat* object.

```
    // convert image to YUV color space. The output image will be created automatically.
    Mat I_YUV;
    cvtColor(I, I_YUV, CV_BGR2YCrCb);

    vector<Mat> planes;    // Use the STL's vector structure to store multiple Mat objects
    split(I_YUV, planes);  // split the image into separate color planes (Y U V)
```

Because, we want to mess around with the images luma component we first convert from the default RGB to the YUV color space and then split the result up into separate planes. Here the program splits: in the first example it processes each plane using one of the three major image scanning algorithms in OpenCV (C [] operator, iterator, individual element access). In a second variant we add to the image some Gaussian noise and then mix together the channels according to some formula.

The scanning version looks like:

```
    // Method 1. process Y plane using an iterator
    MatIterator_<uchar> it = planes[0].begin<uchar>(), it_end = planes[0].end<uchar>();
    for(; it != it_end; ++it)
    {
        double v = *it * 1.7 + rand()%21 - 10;
        *it = saturate_cast<uchar>(v*v/255);
    }

    for( int y = 0; y < I_YUV.rows; y++ )
    {
        // Method 2. process the first chroma plane using pre-stored row pointer.
        uchar* Uptr = planes[1].ptr<uchar>(y);
```

```
        for( int x = 0; x < I_YUV.cols; x++ )
        {
            Uptr[x] = saturate_cast<uchar>((Uptr[x]-128)/2 + 128);

            // Method 3. process the second chroma plane using individual element access
            uchar& Vxy = planes[2].at<uchar>(y, x);
            Vxy =        saturate_cast<uchar>((Vxy-128)/2 + 128);
        }
    }
```

Here you can observe that we may go through all the pixels of an image in three fashions: an iterator, a C pointer and an individual element access style. You can read a more in-depth description of these in the *How to scan images, lookup tables and time measurement with OpenCV* tutorial. Converting from the old function names is easy. Just remove the cv prefix and use the new *Mat* data structure. Here's an example of this by using the weighted addition function:

```
    Mat noisyI(I.size(), CV_8U);           // Create a matrix of the specified size and type

    // Fills the matrix with normally distributed random values (around number with deviation off).
    // There is also randu() for uniformly distributed random number generation
    randn(noisyI, Scalar::all(128), Scalar::all(20));

    // blur the noisyI a bit, kernel size is 3x3 and both sigma's are set to 0.5
    GaussianBlur(noisyI, noisyI, Size(3, 3), 0.5, 0.5);

    const double brightness_gain = 0;
    const double contrast_gain = 1.7;

#ifdef DEMO_MIXED_API_USE
    // To pass the new matrices to the functions that only work with IplImage or CvMat do:
    // step 1) Convert the headers (tip: data will not be copied).
    // step 2) call the function   (tip: to pass a pointer do not forget unary "&" to form pointers)

    IplImage cv_planes_0 = planes[0], cv_noise = noisyI;
    cvAddWeighted(&cv_planes_0, contrast_gain, &cv_noise, 1, -128 + brightness_gain, &cv_planes_0);
#else
    addWeighted(planes[0], contrast_gain, noisyI, 1, -128 + brightness_gain, planes[0]);
#endif

    const double color_scale = 0.5;
    // Mat::convertTo() replaces cvConvertScale.
    // One must explicitly specify the output matrix type (we keep it intact - planes[1].type())
    planes[1].convertTo(planes[1], planes[1].type(), color_scale, 128*(1-color_scale));

    // alternative form of cv::convertScale if we know the datatype at compile time ("uchar" here).
    // This expression will not create any temporary arrays ( so should be almost as fast as above)
    planes[2] = Mat_<uchar>(planes[2]*color_scale + 128*(1-color_scale));

    // Mat::mul replaces cvMul(). Again, no temporary arrays are created in case of simple expressions.
    planes[0] = planes[0].mul(planes[0], 1./255);
```

As you may observe the *planes* variable is of type *Mat*. However, converting from *Mat* to *IplImage* is easy and made automatically with a simple assignment operator.

```
    merge(planes, I_YUV);                // now merge the results back
    cvtColor(I_YUV, I, CV_YCrCb2BGR);  // and produce the output RGB image
```

```
    namedWindow("image with grain", CV_WINDOW_AUTOSIZE);   // use this to create images

#ifdef DEMO_MIXED_API_USE
    // this is to demonstrate that I and IplI really share the data - the result of the above
    // processing is stored in I and thus in IplI too.
    cvShowImage("image with grain", IplI);
#else
    imshow("image with grain", I); // the new MATLAB style function show
```

The new *imshow* highgui function accepts both the *Mat* and *IplImage* data structures. Compile and run the program and if the first image below is your input you may get either the first or second as output:

You may observe a runtime instance of this on the YouTube here and you can `download the source code from here` or find it in the `samples/cpp/tutorial_code/core/interoperability_with_OpenCV_1/interoperability_with_OpenCV` of the OpenCV source code library.

CHAPTER

THREE

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

-

  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:

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

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 μ is the mean (the peak) and σ 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 σ_x and σ_y arguments.
   - σ_x: The standard deviation in x. Writing 0 implies that σ_x is calculated using kernel size.
   - σ_y: The standard deviation in y. Writing 0 implies that σ_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.
   - σ_{Color}: Standard deviation in the color space.
   - σ_{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.
- Analagously 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

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.

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

- 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 dissapear.

Closing

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

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

- Useful to remove small holes (dark regions).

Morphological Gradient

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

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

- 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.

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

Black Hat

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

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

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 Trackbar to select Morphology operation
createTrackbar("Operator:\n 0: Opening - 1: Closing \n 2: Gradient - 3: Top Hat \n 4: Black Hat", window_name, &morph

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

/// Create Trackbar to choose kernel size
createTrackbar( "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_

  /// 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 trackbar **"Operator"** returns the kind of morphology operation to use (**morph_operator**).

       ```
       createTrackbar("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 trackbar **"Element"** returns **morph_elem**, which indicates what kind of structure our kernel is:

       ```
       createTrackbar( "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,
```

- 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:
  1. *Upsize* the image (zoom in) or
  2. *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, wit 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 512×512, 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 $\mathrm{src}(x, y)$. The plot below depicts this. The horizontal blue line represents the threshold thresh (fixed).

Threshold Binary

- This thresholding operation can be expressed as:

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

- So, if the intensity of the pixel $\mathrm{src}(x, y)$ 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:

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

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

Truncate

- This thresholding operation can be expressed as:

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

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

Threshold to Zero

- This operation can be expressed as:

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

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

Threshold to Zero, Inverted

- This operation can be expressed as:

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

- If $\texttt{src}(x,y)$ is greater than $\texttt{thresh}$, 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 threhold 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 coefficeints along with an *anchor point* in that array, which is tipically located at the center.

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:

- 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:

- Use the OpenCV function Sobel to calculate the derivatives from an image.
- 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:

- 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:

$$\mathrm{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

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:

- 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:

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan(\frac{G_y}{G_x})$$

   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 functio
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

- 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:

- 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, θ)

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:

$$r = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta$$

Meaning that each pair (r, θ) 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 θ - 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 θ - 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 (θ, 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 $(\theta, r\ )$ 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 $(\theta, r\ )$
- 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 tracka

```
    Vec4i l = lines[i];
    line( cdst, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(0,0,255), 3, CV_AA);
  }
 #endif
 imshow("source", src);
 imshow("detected lines", cdst);

 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, θ) of the detected lines
   - *rho* : The resolution of the parameter r in pixels. We use $\mathbf{1}$ pixel.
   - *theta*: The resolution of the parameter θ in radians. We use $\mathbf{1}$ **degree** (CV_PI/180)
   - *threshold*: The minimum number of intersections to "*detect*" a line
   - *srn* and *stn*: 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 θ 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, θ). In the circle case, we need three parameters to define a circle:

$$C : (x_{center}, y_{center}, r)$$

  where (x_{center}, y_{center}) define the center position (gree 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 radio 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/// Load theJ0image
```

```
                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, TJ38a.1.71m[(_3252.347(;)-602({})
```

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 2×3 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 2×3 matrix.

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times 2} B = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}_{2\times 1}$$
$$M = \begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}_{2\times 3}$$

   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:

   $T = A \cdot \begin{bmatrix} x \\ y \end{bmatrix} + B$ 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 :math:'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 2×3 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;
```

8. We generate the rotation matrix with the OpenCV function getRotationMatrix2D, which returns a 2×3 matrix (in this case rot_mat)

```
rot_mat = getRotationMatrix2D( center, angle, scale );
```

9. We now apply the found rotation to the output of our previous Transformation.

```
warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size() );
```

10. 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 );
```

11. We just have to wait until the user exits the program

```
waitKey(0);
```

Result

1. 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:

$$equalized(x, y) = H^{'}(src(x, y))$$

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;
```

```
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:

$$[0, 255] = [0, 15] \cup [16, 31] \cup .... \cup [240, 255]$$
$$range = bin_1 \cup bin_2 \cup .... \cup bin_{n=15}$$

and we can keep count of the number of pixels that fall in the range of each 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 $\mathtt{bin}_x$ and $\mathtt{bin}_y$ for each feature and z would be the number of counts for each combination of $(\mathtt{bin}_x, \mathtt{bin}_y)$. 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 ( B, G and R )
  vector<Mat> bgr_planes;
  split( src, bgr_planes );

  /// Establish the number of bins
  int histSize = 256;

  /// Set the ranges ( for B,G,R) )
  float range[] = { 0, 256 } ;
  const float* histRange = { range };

  bool uniform = true; bool accumulate = false;

  Mat b_hist, g_hist, r_hist;

  /// Compute the histograms:
  calcHist( &bgr_planes[0], 1, 0, Mat(), b_hist, 1, &histSize, &histRange, uniform, accumulate );
  calcHist( &bgr_planes[1], 1, 0, Mat(), g_hist, 1, &histSize, &histRange, uniform, accumulate );
  calcHist( &bgr_planes[2], 1, 0, Mat(), r_hist, 1, &histSize, &histRange, uniform, accumulate );

  // Draw the histograms for B, G and R
  int hist_w = 512; int hist_h = 400;
  int bin_w = cvRound( (double) hist_w/histSize );

  Mat histImage( hist_h, hist_w, CV_8UC3, Scalar( 0,0,0) );

  /// Normalize the result to [ 0, histImage.rows ]
  normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
  normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
  normalize(r_hist, r_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(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  );
      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(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  );
  }
```

```
  /// 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> bgr_planes;
   split( src, bgr_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 B, G and R planes, we know that our values will range in the interval $[0, 255]$

   (a) Establish number of bins (5, 10...):

   ```
   int histSize = 256; //from 0 to 255
   ```

   (b) Set the range of values (as we said, between 0 and 255)

   ```
   /// Set the ranges ( for B,G,R) )
   float range[] = { 0, 256 } ; //the upper boundary is exclusive
   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)

- **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
- **b_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 = 512; int hist_h = 400;
int bin_w = cvRound( (double) hist_w/histSize );

Mat histImage( hist_h, hist_w, 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(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
```

this function receives these arguments:

- **b_hist:** Input array
- **b_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(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  );
    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(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  );
}
```

we use the expression:

```
b_hist.at<float>(i)
```

where `i` indicates the dimension. If it were a 2D-histogram we would use something like:

```
b_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)**

$$d(H_1, H_2) = \sum_I \frac{\left(H_1(I) - H_2(I)\right)^2}{H_1(I)}$$

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)**

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{H_1 H_2 N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

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 o-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_h
```

```
    }

  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() );
```

8. 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
  }
```

Results

1. 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.

2. 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:

3. Here the numeric results:

Method	Base - Base	Base - Half	Base - Test 1	Base - Test 2
Correlation	1.000000	0.930766	0.182073	0.120447
Chi-square	0.000000	4.940466	21.184536	49.273437
Intersection	24.391548	14.959809	3.889029	5.775088
Bhattacharyya	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

  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://docs.opencv.org/modules/core/doc/operations_on_arrays.html?highlight=mixchannels#mixc 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 ) ), Sca

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

$$\begin{aligned} T'(x',y') &= T(x',y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} T(x'',y'') \\ I'(x+x',y+y') &= I(x+x',y+y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} I(x+x'',y+y'') \end{aligned}$$

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 bette
  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 $\mathbf{I}$, the template $\mathbf{T}$, the result $\mathbf{R}$ 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 $\mathbf{R}$ 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 CCOEFF, 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( (Mat)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

1. 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 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(c
      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

1. 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:

CHAPTER

FOUR

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
  Author: Ana Huamán
  We will learn how to add a Trackbar to our applications

-

  Title: *Video Input with OpenCV and similarity measurement*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  You will learn how to read video streams, and how to calculate similarity values such as PSNR or SSIM.

-

  Title: *Creating a video with OpenCV*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  Whenever you work with video feeds you may eventually want to save your image processing result in a form of a new video file. Here's how to do it.

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 α and $\texttt{beta}$. 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 α 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*, *dist*, *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 α and another for β. The output might look like:

4.2 Video Input with OpenCV and similarity measurement

Goal

Today it is common to have a digital video recording system at your disposal. Therefore, you will eventually come to the situation that you no longer process a batch of images, but video streams. These may be of two kinds: real-time image feed (in the case of a webcam) or prerecorded and hard disk drive stored files. Luckily OpenCV threats these two in the same manner, with the same C++ class. So here's what you'll learn in this tutorial:

- How to open and read video streams
- Two ways for checking image similarity: PSNR and SSIM

The source code

As a test case where to show off these using OpenCV I've created a small program that reads in two video files and performs a similarity check between them. This is something you could use to check just how well a new video compressing algorithms works. Let there be a reference (original) video like `this small Megamind clip` and `a compressed version of it`. You may also find the source code and these video file in the `samples/cpp/tutorial_code/HighGUI/video-input-psnr-ssim/` folder of the OpenCV source library.

```
#include <iostream> // for standard I/O
#include <string>   // for strings
#include <iomanip>  // for controlling float print precision
#include <sstream>  // string to number conversion

#include <opencv2/core/core.hpp>        // Basic OpenCV structures (cv::Mat, Scalar)
#include <opencv2/imgproc/imgproc.hpp>  // Gaussian Blur
#include <opencv2/highgui/highgui.hpp>  // OpenCV window I/O

using namespace std;
```

```
using namespace cv;

double getPSNR ( const Mat& I1, const Mat& I2);
Scalar getMSSIM( const Mat& I1, const Mat& I2);

int main(int argc, char *argv[])
    help();

    if (argc != 5)
    {
        cout << "Not enough parameters" << endl;
        return -1;
    }

    stringstream conv;

    const string sourceReference = argv[1], sourceCompareWith = argv[2];
    int psnrTriggerValue, delay;
    conv << argv[3] << endl << argv[4];       // put in the strings
    conv >> psnrTriggerValue >> delay;        // take out the numbers

    char c;
    int frameNum = -1;          // Frame counter

    VideoCapture captRefrnc(sourceReference), captUndTst(sourceCompareWith);

    if (!captRefrnc.isOpened())
    {
        cout  << "Could not open reference " << sourceReference << endl;
        return -1;
    }

    if (!captUndTst.isOpened())
    {
        cout  << "Could not open case test " << sourceCompareWith << endl;
        return -1;
    }

    Size refS = Size((int) captRefrnc.get(CV_CAP_PROP_FRAME_WIDTH),
                     (int) captRefrnc.get(CV_CAP_PROP_FRAME_HEIGHT)),
         uTSi = Size((int) captUndTst.get(CV_CAP_PROP_FRAME_WIDTH),
                     (int) captUndTst.get(CV_CAP_PROP_FRAME_HEIGHT));

    if (refS != uTSi)
    {
        cout << "Inputs have different size!!! Closing." << endl;
        return -1;
    }

    const char* WIN_UT = "Under Test";
    const char* WIN_RF = "Reference";

    // Windows
    namedWindow(WIN_RF, CV_WINDOW_AUTOSIZE);
    namedWindow(WIN_UT, CV_WINDOW_AUTOSIZE);
    cvMoveWindow(WIN_RF, 400       , 0);         //750,  2 (bernat =0)
    cvMoveWindow(WIN_UT, refS.width, 0);         //1500, 2

```

```
    cout << "Reference frame resolution: Width=" << refS.width << "  Height=" << refS.height
         << " of nr#: " << captRefrnc.get(CV_CAP_PROP_FRAME_COUNT) << endl;

    cout << "PSNR trigger value " << setiosflags(ios::fixed) << setprecision(3)
         << psnrTriggerValue << endl;

    Mat frameReference, frameUnderTest;
    double psnrV;
    Scalar mssimV;

    for(;;) //Show the image captured in the window and repeat
    {
        captRefrnc >> frameReference;
        captUndTst >> frameUnderTest;

        if (frameReference.empty() || frameUnderTest.empty())
        {
            cout << " < < <  Game over!  > > > ";
            break;
        }

        ++frameNum;
        cout << "Frame: " << frameNum << "# ";

        ////////////////////////////// PSNR ////////////////////////////////////////////////////
        psnrV = getPSNR(frameReference,frameUnderTest);
        cout << setiosflags(ios::fixed) << setprecision(3) << psnrV << "dB";

        ////////////////////////////////// MSSIM ///////////////////////////////////////////////
        if (psnrV < psnrTriggerValue && psnrV)
        {
            mssimV = getMSSIM(frameReference, frameUnderTest);

            cout << " MSSIM: "
                << " R " << setiosflags(ios::fixed) << setprecision(2) << mssimV.val[2] * 100 << "%"
                << " G " << setiosflags(ios::fixed) << setprecision(2) << mssimV.val[1] * 100 << "%"
                << " B " << setiosflags(ios::fixed) << setprecision(2) << mssimV.val[0] * 100 << "%";
        }

        cout << endl;

        ////////////////////////////////// Show Image //////////////////////////////////////////
        imshow(WIN_RF, frameReference);
        imshow(WIN_UT, frameUnderTest);

        c = (char)cvWaitKey(delay);
        if (c == 27) break;
    }

    return 0;
}

double getPSNR(const Mat& I1, const Mat& I2)
{
    Mat s1;
    absdiff(I1, I2, s1);       // |I1 - I2|
    s1.convertTo(s1, CV_32F);  // cannot make a square on 8 bits
    s1 = s1.mul(s1);           // |I1 - I2|^2
```

```

    Scalar s = sum(s1);        // sum elements per channel

    double sse = s.val[0] + s.val[1] + s.val[2]; // sum channels

    if( sse <= 1e-10) // for small values return zero
        return 0;
    else
    {
        double mse  = sse / (double)(I1.channels() * I1.total());
        double psnr = 10.0 * log10((255 * 255) / mse);
        return psnr;
    }
}

Scalar getMSSIM( const Mat& i1, const Mat& i2)
{
    const double C1 = 6.5025, C2 = 58.5225;
    /***************************** INITS **********************************/
    int d = CV_32F;

    Mat I1, I2;
    i1.convertTo(I1, d);            // cannot calculate on one byte large values
    i2.convertTo(I2, d);

    Mat I2_2   = I2.mul(I2);        // I2^2
    Mat I1_2   = I1.mul(I1);        // I1^2
    Mat I1_I2  = I1.mul(I2);        // I1 * I2

    /*************************** END INITS **********************************/

    Mat mu1, mu2;                   // PRELIMINARY COMPUTING
    GaussianBlur(I1, mu1, Size(11, 11), 1.5);
    GaussianBlur(I2, mu2, Size(11, 11), 1.5);

    Mat mu1_2   =   mu1.mul(mu1);
    Mat mu2_2   =   mu2.mul(mu2);
    Mat mu1_mu2 =   mu1.mul(mu2);

    Mat sigma1_2, sigma2_2, sigma12;

    GaussianBlur(I1_2, sigma1_2, Size(11, 11), 1.5);
    sigma1_2 -= mu1_2;

    GaussianBlur(I2_2, sigma2_2, Size(11, 11), 1.5);
    sigma2_2 -= mu2_2;

    GaussianBlur(I1_I2, sigma12, Size(11, 11), 1.5);
    sigma12 -= mu1_mu2;

    ////////////////////////////////// FORMULA ////////////////////////////////
    Mat t1, t2, t3;

    t1 = 2 * mu1_mu2 + C1;
    t2 = 2 * sigma12 + C2;
    t3 = t1.mul(t2);                // t3 = ((2*mu1_mu2 + C1).*(2*sigma12 + C2))

    t1 = mu1_2 + mu2_2 + C1;
```

```
    t2 = sigma1_2 + sigma2_2 + C2;
    t1 = t1.mul(t2);                // t1 =((mu1_2 + mu2_2 + C1).*(sigma1_2 + sigma2_2 + C2))

    Mat ssim_map;
    divide(t3, t1, ssim_map);       // ssim_map =  t3./t1;

    Scalar mssim = mean(ssim_map);  // mssim = average of ssim map
```

How to read a video stream (online-camera or offline-file)?

Essentially, all the functionalities required for video manipulation is integrated in the VideoCapture C++ class. This on itself builds on the FFmpeg open source library. This is a basic dependency of OpenCV so you shouldn't need to worry about this. A video is composed of a succession of images, we refer to these in the literature as frames. In case of a video file there is a *frame rate* specifying just how long is between two frames. While for the video cameras usually there is a limit of just how many frames they can digitalize per second, this property is less important as at any time the camera sees the current snapshot of the world.

The first task you need to do is to assign to a VideoCapture class its source. You can do this either via the constructor or its open function. If this argument is an integer then you will bind the class to a camera, a device. The number passed here is the ID of the device, assigned by the operating system. If you have a single camera attached to your system its ID will probably be zero and further ones increasing from there. If the parameter passed to these is a string it will refer to a video file, and the string points to the location and name of the file. For example, to the upper source code a valid command line is:

```
video/Megamind.avi video/Megamind_bug.avi  35 10
```

We do a similarity check. This requires a reference and a test case video file. The first two arguments refer to this. Here we use a relative address. This means that the application will look into its current working directory and open the video folder and try to find inside this the *Megamind.avi* and the *Megamind_bug.avi*.

```
const string sourceReference = argv[1],sourceCompareWith = argv[2];

VideoCapture captRefrnc(sourceReference);
// or
VideoCapture captUndTst;
captUndTst.open(sourceCompareWith);
```

To check if the binding of the class to a video source was successful or not use the isOpened function:

```
if ( !captRefrnc.isOpened())
  {
  cout  << "Could not open reference " << sourceReference << endl;
  return -1;
  }
```

Closing the video is automatic when the objects destructor is called. However, if you want to close it before this you need to call its release function. The frames of the video are just simple images. Therefore, we just need to extract them from the VideoCapture object and put them inside a *Mat* one. The video streams are sequential. You may get the frames one after another by the read or the overloaded >> operator:

```
Mat frameReference, frameUnderTest;
captRefrnc >> frameReference;
captUndTst.open(frameUnderTest);
```

The upper read operations will leave empty the *Mat* objects if no frame could be acquired (either cause the video stream was closed or you got to the end of the video file). We can check this with a simple if:

```
if( frameReference.empty()  || frameUnderTest.empty())
{
 // exit the program
}
```

A read method is made of a frame grab and a decoding applied on that. You may call explicitly these two by using the grab and then the retrieve functions.

Videos have many-many information attached to them besides the content of the frames. These are usually numbers, however in some case it may be short character sequences (4 bytes or less). Due to this to acquire these information there is a general function named get that returns double values containing these properties. Use bitwise operations to decode the characters from a double type and conversions where valid values are only integers. Its single argument is the ID of the queried property. For example, here we get the size of the frames in the reference and test case video file; plus the number of frames inside the reference.

```
Size refS = Size((int) captRefrnc.get(CV_CAP_PROP_FRAME_WIDTH),
                 (int) captRefrnc.get(CV_CAP_PROP_FRAME_HEIGHT)),

cout << "Reference frame resolution: Width=" << refS.width << "  Height=" << refS.height
     << " of nr#: " << captRefrnc.get(CV_CAP_PROP_FRAME_COUNT) << endl;
```

When you are working with videos you may often want to control these values yourself. To do this there is a set function. Its first argument remains the name of the property you want to change and there is a second of double type containing the value to be set. It will return true if it succeeds and false otherwise. Good examples for this is seeking in a video file to a given time or frame:

```
captRefrnc.set(CV_CAP_PROP_POS_MSEC, 1.2);  // go to the 1.2 second in the video
captRefrnc.set(CV_CAP_PROP_POS_FRAMES, 10); // go to the 10th frame of the video
// now a read operation would read the frame at the set position
```

For properties you can read and change look into the documentation of the get and set functions.

Image similarity - PSNR and SSIM

We want to check just how imperceptible our video converting operation went, therefore we need a system to check frame by frame the similarity or differences. The most common algorithm used for this is the PSNR (aka **Peak signal-to-noise ratio**). The simplest definition of this starts out from the *mean squad error*. Let there be two images: I1 and I2; with a two dimensional size i and j, composed of c number of channels.

$$MSE = \frac{1}{c*i*j}\sum{(I_1 - I_2)^2}$$

Then the PSNR is expressed as:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

Here the MAX_I^2 is the maximum valid value for a pixel. In case of the simple single byte image per pixel per channel this is 255. When two images are the same the MSE will give zero, resulting in an invalid divide by zero operation in the PSNR formula. In this case the PSNR is undefined and as we'll need to handle this case separately. The transition to a logarithmic scale is made because the pixel values have a very wide dynamic range. All this translated to OpenCV and a C++ function looks like:

```
double getPSNR(const Mat& I1, const Mat& I2)
{
 Mat s1;
 absdiff(I1, I2, s1);       // |I1 - I2|
```

```
 s1.convertTo(s1, CV_32F);  // cannot make a square on 8 bits
 s1 = s1.mul(s1);           // |I1 - I2|^2

 Scalar s = sum(s1);         // sum elements per channel

 double sse = s.val[0] + s.val[1] + s.val[2]; // sum channels

 if( sse <= 1e-10) // for small values return zero
     return 0;
 else
 {
     double  mse =sse /(double)(I1.channels() * I1.total());
     double psnr = 10.0*log10((255*255)/mse);
     return psnr;
 }
}
```

Typically result values are anywhere between 30 and 50 for video compression, where higher is better. If the images significantly differ you'll get much lower ones like 15 and so. This similarity check is easy and fast to calculate, however in practice it may turn out somewhat inconsistent with human eye perception. The **structural similarity** algorithm aims to correct this.

Describing the methods goes well beyond the purpose of this tutorial. For that I invite you to read the article introducing it. Nevertheless, you can get a good image of it by looking at the OpenCV implementation below.

See Also:

SSIM is described more in-depth in the: "Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, Apr. 2004." article.

```
Scalar getMSSIM( const Mat& i1, const Mat& i2)
{
 const double C1 = 6.5025, C2 = 58.5225;
 /***************************** INITS **********************************/
 int d     = CV_32F;

 Mat I1, I2;
 i1.convertTo(I1, d);           // cannot calculate on one byte large values
 i2.convertTo(I2, d);

 Mat I2_2   = I2.mul(I2);        // I2^2
 Mat I1_2   = I1.mul(I1);        // I1^2
 Mat I1_I2  = I1.mul(I2);        // I1 * I2

 /***********************PRELIMINARY COMPUTING ******************************/

 Mat mu1, mu2;   //
 GaussianBlur(I1, mu1, Size(11, 11), 1.5);
 GaussianBlur(I2, mu2, Size(11, 11), 1.5);

 Mat mu1_2   =   mu1.mul(mu1);
 Mat mu2_2   =   mu2.mul(mu2);
 Mat mu1_mu2 =   mu1.mul(mu2);

 Mat sigma1_2, sigma2_2, sigma12;

 GaussianBlur(I1_2, sigma1_2, Size(11, 11), 1.5);
 sigma1_2 -= mu1_2;
```

```
 GaussianBlur(I2_2, sigma2_2, Size(11, 11), 1.5);
 sigma2_2 -= mu2_2;

 GaussianBlur(I1_I2, sigma12, Size(11, 11), 1.5);
 sigma12 -= mu1_mu2;

 ////////////////////////////// FORMULA ////////////////////////////////
 Mat t1, t2, t3;

 t1 = 2 * mu1_mu2 + C1;
 t2 = 2 * sigma12 + C2;
 t3 = t1.mul(t2);              // t3 = ((2*mu1_mu2 + C1).*(2*sigma12 + C2))

 t1 = mu1_2 + mu2_2 + C1;
 t2 = sigma1_2 + sigma2_2 + C2;
 t1 = t1.mul(t2);               // t1 =((mu1_2 + mu2_2 + C1).*(sigma1_2 + sigma2_2 + C2))

 Mat ssim_map;
 divide(t3, t1, ssim_map);      // ssim_map =  t3./t1;

 Scalar mssim = mean( ssim_map ); // mssim = average of ssim map
 return mssim;
}
```

This will return a similarity index for each channel of the image. This value is between zero and one, where one corresponds to perfect fit. Unfortunately, the many Gaussian blurring is quite costly, so while the PSNR may work in a real time like environment (24 frame per second) this will take significantly more than to accomplish similar performance results.

Therefore, the source code presented at the start of the tutorial will perform the PSNR measurement for each frame, and the SSIM only for the frames where the PSNR falls below an input value. For visualization purpose we show both images in an OpenCV window and print the PSNR and MSSIM values to the console. Expect to see something like:

You may observe a runtime instance of this on the YouTube here.

4.3 Creating a video with OpenCV

Goal

Whenever you work with video feeds you may eventually want to save your image processing result in a form of a new video file. For simple video outputs you can use the OpenCV built-in VideoWriter class, designed for this.

- How to create a video file with OpenCV
- What type of video files you can create with OpenCV
- How to extract a given color channel from a video

As a simple demonstration I'll just extract one of the RGB color channels of an input video file into a new video. You can control the flow of the application from its console line arguments:

- The first argument points to the video file to work on
- The second argument may be one of the characters: R G B. This will specify which of the channels to extract.
- The last argument is the character Y (Yes) or N (No). If this is no, the codec used for the input video file will be the same as for the output. Otherwise, a window will pop up and allow you to select yourself the codec to use.

For example, a valid command line would look like:

```
video-write.exe video/Megamind.avi R Y
```

The source code

You may also find the source code and these video file in the `samples/cpp/tutorial_code/highgui/video-write/` folder of the OpenCV source library or `download it from here`.

```
#include <iostream> // for standard I/O
#include <string>   // for strings

#include <opencv2/core/core.hpp>        // Basic OpenCV structures (cv::Mat)
#include <opencv2/highgui/highgui.hpp>  // Video write

using namespace std;
using namespace cv;

static void help()
{
    cout
        << "------------------------------------------------------------------------------" << endl
        << "This program shows how to write video files."                                  << endl
        << "You can extract the R or G or B color channel of the input video."             << endl
        << "Usage:"                                                                        << endl
        << "./video-write inputvideoName [ R | G | B] [Y | N]"                             << endl
        << "------------------------------------------------------------------------------" << endl
        << endl;
}

int main(int argc, char *argv[])
{
```

```
    help();

    if (argc != 4)
    {
        cout << "Not enough parameters" << endl;
        return -1;
    }

    const string source      = argv[1];           // the source file name
    const bool askOutputType = argv[3][0] =='Y';  // If false it will use the inputs codec type

    VideoCapture inputVideo(source);              // Open input
    if (!inputVideo.isOpened())
    {
        cout  << "Could not open the input video: " << source << endl;
        return -1;
    }

    string::size_type pAt = source.find_last_of('.');                  // Find extension point
    const string NAME = source.substr(0, pAt) + argv[2][0] + ".avi";   // Form the new name with container
    int ex = static_cast<int>(inputVideo.get(CV_CAP_PROP_FOURCC));     // Get Codec Type- Int form

    // Transform from int to char via Bitwise operators
    char EXT[] = {(char)(ex & 0XFF) , (char)((ex & 0XFF00) >> 8),(char)((ex & 0XFF0000) >> 16),(char)((ex & 0XFF000000

    Size S = Size((int) inputVideo.get(CV_CAP_PROP_FRAME_WIDTH),    // Acquire input size
                  (int) inputVideo.get(CV_CAP_PROP_FRAME_HEIGHT));

    VideoWriter outputVideo;                                        // Open the output
    if (askOutputType)
        outputVideo.open(NAME, ex=-1, inputVideo.get(CV_CAP_PROP_FPS), S, true);
    else
        outputVideo.open(NAME, ex, inputVideo.get(CV_CAP_PROP_FPS), S, true);

    if (!outputVideo.isOpened())
    {
        cout  << "Could not open the output video for write: " << source << endl;
        return -1;
    }

    cout << "Input frame resolution: Width=" << S.width << "  Height=" << S.height
         << " of nr#: " << inputVideo.get(CV_CAP_PROP_FRAME_COUNT) << endl;
    cout << "Input codec type: " << EXT << endl;

    int channel = 2; // Select the channel to save
    switch(argv[2][0])
    {
    case 'R' : channel = 2; break;
    case 'G' : channel = 1; break;
    case 'B' : channel = 0; break;
    }
    Mat src, res;
    vector<Mat> spl;

    for(;;) //Show the image captured in the window and repeat
    {
        inputVideo >> src;              // read
        if (src.empty()) break;         // check if at end
```

```
        split(src, spl);                // process - extract only the correct channel
        for (int i =0; i < 3; ++i)
            if (i != channel)
                spl[i] = Mat::zeros(S, spl[0].type());
       merge(spl, res);

       //outputVideo.write(res); //save or
       outputVideo << res;
    }

    cout << "Finished writing" << endl;
    return 0;
}
```

The structure of a video

For start, you should have an idea of just how a video file looks. Every video file in itself is a container. The type of the container is expressed in the files extension (for example *avi*, *mov* or *mkv*). This contains multiple elements like: video feeds, audio feeds or other tracks (like for example subtitles). How these feeds are stored is determined by the codec used for each one of them. In case of the audio tracks commonly used codecs are *mp3* or *aac*. For the video files the list is somehow longer and includes names such as *XVID*, *DIVX*, *H264* or *LAGS* (*Lagarith Lossless Codec*). The full list of codecs you may use on a system depends on just what one you have installed.

As you can see things can get really complicated with videos. However, OpenCV is mainly a computer vision library, not a video stream, codec and write one. Therefore, the developers tried to keep this part as simple as possible. Due to this OpenCV for video containers supports only the *avi* extension, its first version. A direct limitation of this is that you cannot save a video file larger than 2 GB. Furthermore you can only create and expand a single video track inside the container. No audio or other track editing support here. Nevertheless, any video codec present on your system might work. If you encounter some of these limitations you will need to look into more specialized video writing libraries such as *FFMpeg* or codecs as *HuffYUV*, *CorePNG* and *LCL*. As an alternative, create the video track with OpenCV and expand it with sound tracks or convert it to other formats by using video manipulation programs such as *VirtualDub* or *AviSynth*. The *VideoWriter* class ========== The content written here builds on the assumption you already read the *Video Input with OpenCV and similarity measurement* tutorial and you know how to read video files. To create a video file you just need to create an instance of the VideoWriter class. You can specify its properties either via parameters in the constructor or later on via the open function. Either way, the parameters are the same: 1. The name of the output that contains the container type in its extension. At the moment only *avi* is supported. We construct this from the input file, add to this the name of the channel to use, and finish it off with the container extension.

```
const string source      = argv[1];            // the source file name
string::size_type pAt = source.find_last_of('.');   // Find extension point
```

```
const string NAME = source.substr(0, pAt) + argv[2][0] + ".avi";   // Form the new name with container
```

1. The codec to use for the video track. Now all the video codecs have a unique short name of maximum four characters. Hence, the *XVID*, *DIVX* or *H264* names. This is called a four character code. You may also ask this from an input video by using its *get* function. Because the *get* function is a general function it always returns double values. A double value is stored on 64 bits. Four characters are four bytes, meaning 32 bits. These four characters are coded in the lower 32 bits of the *double*. A simple way to throw away the upper 32 bits would be to just convert this value to *int*:

   ```
   VideoCapture inputVideo(source);                                  // Open input
   int ex = static_cast<int>(inputVideo.get(CV_CAP_PROP_FOURCC));    // Get Codec Type- Int form
   ```

   OpenCV internally works with this integer type and expect this as its second parameter. Now to convert from the integer form to string we may use two methods: a bitwise operator and a union method. The first one extracting from an int the characters looks like (an "and" operation, some shifting and adding a 0 at the end to close the string):

   ```
   char EXT[] = {ex & 0XFF , (ex & 0XFF00) >> 8,(ex & 0XFF0000) >> 16,(ex & 0XFF000000) >> 24, 0};
   ```

   You can do the same thing with the *union* as:

   ```
   union { int v; char c[5];} uEx ;
   uEx.v = ex;                              // From Int to char via union
   uEx.c[4]='\0';
   ```

   The advantage of this is that the conversion is done automatically after assigning, while for the bitwise operator you need to do the operations whenever you change the codec type. In case you know the codecs four character code beforehand, you can use the *CV_FOURCC* macro to build the integer:

   If you pass for this argument minus one than a window will pop up at runtime that contains all the codec installed on your system and ask you to select the one to use:

2. The frame per second for the output video. Again, here I keep the input videos frame per second by using the *get* function.
3. The size of the frames for the output video. Here too I keep the input videos frame size per second by using the *get* function.
4. The final argument is an optional one. By default is true and says that the output will be a colorful one (so for write you will send three channel images). To create a gray scale video pass a false parameter here.

Here it is, how I use it in the sample:

```
VideoWriter outputVideo;
Size S = Size((int) inputVideo.get(CV_CAP_PROP_FRAME_WIDTH),    //Acquire input size
              (int) inputVideo.get(CV_CAP_PROP_FRAME_HEIGHT));
outputVideo.open(NAME , ex, inputVideo.get(CV_CAP_PROP_FPS),S, true);
```

Afterwards, you use the isOpened() function to find out if the open operation succeeded or not. The video file automatically closes when the *VideoWriter* object is destroyed. After you open the object with success you can send the frames of the video in a sequential order by using the write function of the class. Alternatively, you can use its overloaded operator << :

```
outputVideo.write(res);  //or
outputVideo << res;
```

Extracting a color channel from an RGB image means to set to zero the RGB values of the other channels. You can either do this with image scanning operations or by using the split and merge operations. You first split the channels up into different images, set the other channels to zero images of the same size and type and finally merge them back:

```
split(src, spl);                // process - extract only the correct channel
for( int i =0; i < 3; ++i)
   if (i != channel)
      spl[i] = Mat::zeros(S, spl[0].type());
merge(spl, res);
```

Put all this together and you'll get the upper source code, whose runtime result will show something around the idea:

You may observe a runtime instance of this on the YouTube here.

CHAPTER

FIVE

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.

-

  Title: *Camera calibration with square chessboard*
  Compatibility: > OpenCV 2.0
  Author: Victor Eruhimov
  You will use some chessboard images to calibrate your camera.

-

  Title: *Camera calibration With OpenCV*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  Camera calibration by using either the chessboard, circle or the asymmetrical circle pattern. Get the images either from a camera attached, a video file or from an image collection.

5.1 Camera calibration with square chessboard

The goal of this tutorial is to learn how to calibrate a camera given a set of chessboard images.

Test data: use images in your data/chess folder.

1. Compile opencv with samples by setting `BUILD_EXAMPLES` to `ON` in cmake configuration.
2. Go to `bin` folder and use `imagelist_creator` to create an `XML/YAML` list of your images.
3. Then, run `calibration` sample to get camera parameters. Use square size equal to 3cm.

Pose estimation

Now, let us write a code that detects a chessboard in a new image and finds its distance from the camera. You can apply the same method to any object with known 3D geometry that you can detect in an image.

Test data: use chess_test*.jpg images from your data folder.

1. Create an empty console project. Load a test image:

```
Mat img = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
```

2. Detect a chessboard in this image using findChessboard function.

```
bool found = findChessboardCorners( img, boardSize, ptvec, CV_CALIB_CB_ADAPTIVE_THRESH );
```

3. Now, write a function that generates a `vector<Point3f>` array of 3d coordinates of a chessboard in any coordinate system. For simplicity, let us choose a system such that one of the chessboard corners is in the origin and the board is in the plane $z = 0$.
4. Read camera parameters from XML/YAML file:

```
FileStorage fs(filename, FileStorage::READ);
Mat intrinsics, distortion;
fs["camera_matrix"] >> intrinsics;
fs["distortion_coefficients"] >> distortion;
```

5. Now we are ready to find chessboard pose by running `solvePnP`:

```
vector<Point3f> boardPoints;
// fill the array
...

solvePnP(Mat(boardPoints), Mat(foundBoardCorners), cameraMatrix,
                     distCoeffs, rvec, tvec, false);
```

6. Calculate reprojection error like it is done in `calibration` sample (see `opencv/samples/cpp/calibration.cpp`, function `computeReprojectionErrors`).

Question: how to calculate the distance from the camera origin to any of the corners?

5.2 Camera calibration With OpenCV

Cameras have been around for a long-long time. However, with the introduction of the cheap *pinhole* cameras in the late 20th century, they became a common occurrence in our everyday life. Unfortunately, this cheapness comes with its price: significant distortion. Luckily, these are constants and with a calibration and some remapping we can correct

this. Furthermore, with calibration you may also determinate the relation between the camera's natural units (pixels) and the real world units (for example millimeters).

Theory

For the distortion OpenCV takes into account the radial and tangential factors. For the radial one uses the following formula:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

So for an old pixel point at (x, y) coordinate in the input image, for a corrected output image its position will be $(x_{corrected} y_{corrected})$. The presence of the radial distortion manifests in form of the "barrel" or "fish-eye" effect.

Tangential distortion occurs because the image taking lenses are not perfectly parallel to the imaging plane. Correcting this is made via the formulas:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

So we have five distortion parameters, which in OpenCV are organized in a 5 column one row matrix:

$$Distortion_{coefficients} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Now for the unit conversion, we use the following formula:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Here the presence of the w is cause we use a homography coordinate system (and $w = Z$). The unknown parameters are f_x and f_y (camera focal lengths) and (c_x, c_y) what are the optical centers expressed in pixels coordinates. If for both axes a common focal length is used with a given a aspect ratio (usually 1), then $f_y = f_x * a$ and in the upper formula we will have a single f focal length. The matrix containing these four parameters is referred to as the *camera matrix*. While the distortion coefficients are the same regardless of the camera resolutions used, these should be scaled along with the current resolution from the calibrated resolution.

The process of determining these two matrices is the calibration. Calculating these parameters is done by some basic geometrical equations. The equations used depend on the calibrating objects used. Currently OpenCV supports three types of object for calibration:

- Classical black-white chessboard
- Symmetrical circle pattern
- Asymmetrical circle pattern

Basically, you need to take snapshots of these patterns with your camera and let OpenCV find them. Each found pattern equals in a new equation. To solve the equation you need at least a predetermined number of pattern snapshots to form a well-posed equation system. This number is higher for the chessboard pattern and less for the circle ones. For example, in theory the chessboard one requires at least two. However, in practice we have a good amount of noise present in our input images, so for good results you will probably want at least 10 good snapshots of the input pattern in different position.

Goal

The sample application will:

- Determinate the distortion matrix
- Determinate the camera matrix
- Input from Camera, Video and Image file list
- Configuration from XML/YAML file
- Save the results into XML/YAML file
- Calculate re-projection error

Source code

You may also find the source code in the `samples/cpp/tutorial_code/calib3d/camera_calibration/` folder of the OpenCV source library or `download it from here`. The program has a single argument. The name of its configuration file. If none given it will try to open the one named "default.xml". `Here's a sample configuration file` in XML format. In the configuration file you may choose to use as input a camera, a video file or an image list. If you opt for the later one, you need to create a configuration file where you enumerate the images to use. Here's `an example of this`. The important part to remember is that the images needs to be specified using the absolute path or the relative one from your applications working directory. You may find all this in the beforehand mentioned directory.

The application starts up with reading the settings from the configuration file. Although, this is an important part of it, it has nothing to do with the subject of this tutorial: *camera calibration*. Therefore, I've chosen to do not post here the code part for that. The technical background on how to do this you can find in the *File Input and Output using XML and YAML files* tutorial.

Explanation

1. **Read the settings.**

```
Settings s;
const string inputSettingsFile = argc > 1 ? argv[1] : "default.xml";
FileStorage fs(inputSettingsFile, FileStorage::READ); // Read the settings
if (!fs.isOpened())
{
      cout << "Could not open the configuration file: \"" << inputSettingsFile << "\"" << endl;
      return -1;
}
fs["Settings"] >> s;
fs.release();                                         // close Settings file

if (!s.goodInput)
{
      cout << "Invalid input detected. Application stopping. " << endl;
      return -1;
}
```

   For this I've used simple OpenCV class input operation. After reading the file I've an additional post-process function that checks for the validity of the input. Only if all of them are good will be the *goodInput* variable true.

2. **Get next input, if it fails or we have enough of them calibrate**. After this we have a big loop where we do the following operations: get the next image from the image list, camera or video file. If this fails or we have enough images we run the calibration process. In case of image we step out of the loop and otherwise the remaining frames will be undistorted (if the option is set) via changing from *DETECTION* mode to *CALIBRATED* one.

```
for(int i = 0;;++i)
{
  Mat view;
  bool blinkOutput = false;

  view = s.nextImage();

  //-----  If no more image, or got enough, then stop calibration and show result -------------
  if( mode == CAPTURING && imagePoints.size() >= (unsigned)s.nrFrames )
  {
        if( runCalibrationAndSave(s, imageSize,  cameraMatrix, distCoeffs, imagePoints))
              mode = CALIBRATED;
        else
              mode = DETECTION;
  }
  if(view.empty())          // If no more images then run calibration, save and stop loop.
  {
            if( imagePoints.size() > 0 )
                  runCalibrationAndSave(s, imageSize,  cameraMatrix, distCoeffs, imagePoints);
            break;
  imageSize = view.size();  // Format input image.
  s imageSize  cameraMatrix distCoeffl
```

purposes we will draw the found points on the input image with the findChessboardCorners function.

```
if ( found)                // If done with success,
  {
      // improve the found corners' coordinate accuracy for chessboard
        if( s.calibrationPattern == Settings::CHESSBOARD)
        {
            Mat viewGray;
            cvtColor(view, viewGray, CV_BGR2GRAY);
            cornerSubPix( viewGray, pointBuf, Size(11,11),
              Size(-1,-1), TermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 30, 0.1 ));
        }

        if( mode == CAPTURING &&  // For camera only take new samples after delay time
            (!s.inputCapture.isOpened() || clock() - prevTimestamp > s.delay*1e-3*CLOCKS_PER_SEC) )
        {
            imagePoints.push_back(pointBuf);
            prevTimestamp = clock();
            blinkOutput = s.inputCapture.isOpened();
        }

        // Draw the corners.
        drawChessboardCorners( view, s.boardSize, Mat(pointBuf), found );
  }
```

4. **Show state and result for the user, plus command line control of the application**. The showing part consists of a text output on the live feed, and for video or camera input to show the "capturing" frame we simply bitwise negate the input image.

```
//---------------------------- Output Text ------------------------------------------------
string msg = (mode == CAPTURING) ? "100/100" :
          mode == CALIBRATED ? "Calibrated" : "Press 'g' to start";
int baseLine = 0;
Size textSize = getTextSize(msg, 1, 1, 1, &baseLine);
Point textOrigin(view.cols - 2*textSize.width - 10, view.rows - 2*baseLine - 10);

if( mode == CAPTURING )
{
  if(s.showUndistorsed)
    msg = format( "%d/%d Undist", (int)imagePoints.size(), s.nrFrames );
  else
    msg = format( "%d/%d", (int)imagePoints.size(), s.nrFrames );
}

putText( view, msg, textOrigin, 1, 1, mode == CALIBRATED ?  GREEN : RED);

if( blinkOutput )
   bitwise_not(view, view);
```

If we only ran the calibration and got the camera matrix plus the distortion coefficients we may just as correct the image with the undistort function:

```
//------------------------- Video capture  output  undistorted ------------------------------
if( mode == CALIBRATED && s.showUndistorsed )
{
  Mat temp = view.clone();
  undistort(temp, view, cameraMatrix, distCoeffs);
}
//------------------------------ Show image and check for input commands -------------------
```

```
imshow("Image View", view);
```

Then we wait for an input key and if this is u we toggle the distortion removal, if it is g we start all over the detection process (or simply start it), and finally for the ESC key quit the application:

```
char key =  waitKey(s.inputCapture.isOpened() ? 50 : s.delay);
if( key  == ESC_KEY )
      break;

if( key == 'u' && mode == CALIBRATED )
   s.showUndistorsed = !s.showUndistorsed;

if( s.inputCapture.isOpened() && key == 'g' )
{
  mode = CAPTURING;
  imagePoints.clear();
}
```

5. **Show the distortion removal for the images too**. When you work with an image list it is not possible to remove the distortion inside the loop. Therefore, you must append this after the loop. Taking advantage of this now I'll expand the undistort function, which is in fact first a call of the initUndistortRectifyMap to find out the transformation matrices and then doing the transformation with the remap function. Because, after a successful calibration the map calculation needs to be done only once, by using this expanded form you may speed up your application:

```
if( s.inputType == Settings::IMAGE_LIST && s.showUndistorsed )
{
  Mat view, rview, map1, map2;
  initUndistortRectifyMap(cameraMatrix, distCoeffs, Mat(),
      getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, 1, imageSize, 0),
      imageSize, CV_16SC2, map1, map2);

  for(int i = 0; i < (int)s.imageList.size(); i++ )
  {
      view = imread(s.imageList[i], 1);
      if(view.empty())
          continue;
      remap(view, rview, map1, map2, INTER_LINEAR);
      imshow("Image View", rview);
      char c = waitKey();
      if( c  == ESC_KEY || c == 'q' || c == 'Q' )
          break;
  }
}
```

The calibration and save

Because the calibration needs to be only once per camera it makes sense to save them after a successful calibration. This way later on you can just load these values into your program. Due to this we first make the calibration, and if it succeeds we save the result into an OpenCV style XML or YAML file, depending on the extension you give in the configuration file.

Therefore in the first function we just split up these two processes. Because we want to save many of the calibration variables we'll create these variables here and pass on both of them to the calibration and saving function. Again, I'll not show the saving part as that has little in common with the calibration. Explore the source file in order to find out how and what:

```
bool runCalibrationAndSave(Settings& s, Size imageSize, Mat&  cameraMatrix, Mat& distCoeffs,vector<vector<Point2f> > i
{
 vector<Mat> rvecs, tvecs;
 vector<float> reprojErrs;
 double totalAvgErr = 0;

 bool ok = runCalibration(s,imageSize, cameraMatrix, distCoeffs, imagePoints, rvecs, tvecs,
                          reprojErrs, totalAvgErr);
 cout << (ok ? "Calibration succeeded" : "Calibration failed")
     << ". avg re projection error = "  << totalAvgErr ;

 if( ok )   // save only if the calibration was done with success
     saveCameraParams( s, imageSize, cameraMatrix, distCoeffs, rvecs ,tvecs, reprojErrs,
                        imagePoints, totalAvgErr);
 return ok;
}
```

We do the calibration with the help of the calibrateCamera function. This has the following parameters:

- The object points. This is a vector of *Point3f* vector that for each input image describes how should the pattern look. If we have a planar pattern (like a chessboard) then we can simply set all Z coordinates to zero. This is a collection of the points where these important points are present. Because, we use a single pattern for all the input images we can calculate this just once and multiply it for all the other input views. We calculate the corner points with the *calcBoardCornerPositions* function as:

  ```
  void calcBoardCornerPositions(Size boardSize, float squareSize, vector<Point3f>& corners,
                    Settings::Pattern patternType /*= Settings::CHESSBOARD*/)
  {
  corners.clear();

  switch(patternType)
  {
  case Settings::CHESSBOARD:
  case Settings::CIRCLES_GRID:
    for( int i = 0; i < boardSize.height; ++i )
      for( int j = 0; j < boardSize.width; ++j )
          corners.push_back(Point3f(float( j*squareSize ), float( i*squareSize ), 0));
    break;

  case Settings::ASYMMETRIC_CIRCLES_GRID:
    for( int i = 0; i < boardSize.height; i++ )
       for( int j = 0; j < boardSize.width; j++ )
          corners.push_back(Point3f(float((2*j + i % 2)*squareSize), float(i*squareSize), 0));
    break;
  }
  }
  ```

  And then multiply it as:

  ```
  vector<vector<Point3f> > objectPoints(1);
  calcBoardCornerPositions(s.boardSize, s.squareSize, objectPoints[0], s.calibrationPattern);
  objectPoints.resize(imagePoints.size(),objectPoints[0]);
  ```

- The image points. This is a vector of *Point2f* vector that for each input image contains where the important points (corners for chessboard, and center of circles for the circle patterns) were found. We already collected this from what the findChessboardCorners or the findCirclesGrid function returned. We just need to pass it on.
- The size of the image acquired from the camera, video file or the images.
- The camera matrix. If we used the fix aspect ratio option we need to set the f_x to zero:

```
cameraMatrix = Mat::eye(3, 3, CV_64F);
if( s.flag & CV_CALIB_FIX_ASPECT_RATIO )
     cameraMatrix.at<double>(0,0) = 1.0;
```

- The distortion coefficient matrix. Initialize with zero.

```
distCoeffs = Mat::zeros(8, 1, CV_64F);
```

- The function will calculate for all the views the rotation and translation vector that transform the object points (given in the model coordinate space) to the image points (given in the world coordinate space). The 7th and 8th parameters are an output vector of matrices containing in the ith position the rotation and translation vector for the ith object point to the ith image point.
- The final argument is a flag. You need to specify here options like fix the aspect ratio for the focal length, assume zero tangential distortion or to fix the principal point.

```
double rms = calibrateCamera(objectPoints, imagePoints, imageSize, cameraMatrix,
                             distCoeffs, rvecs, tvecs, s.flag|CV_CALIB_FIX_K4|CV_CALIB_FIX_K5);
```

- The function returns the average re-projection error. This number gives a good estimation of just how exact is the found parameters. This should be as close to zero as possible. Given the intrinsic, distortion, rotation and translation matrices we may calculate the error for one view by using the projectPoints to first transform the object point to image point. Then we calculate the absolute norm between what we got with our transformation and the corner/circle finding algorithm. To find the average error we calculate the arithmetical mean of the errors calculate for all the calibration images.

```
double computeReprojectionErrors( const vector<vector<Point3f> >& objectPoints,
                                  const vector<vector<Point2f> >& imagePoints,
                                  const vector<Mat>& rvecs, const vector<Mat>& tvecs,
                                  const Mat& cameraMatrix , const Mat& distCoeffs,
                                  vector<float>& perViewErrors)
{
vector<Point2f> imagePoints2;
int i, totalPoints = 0;
double totalErr = 0, err;
perViewErrors.resize(objectPoints.size());

for( i = 0; i < (int)objectPoints.size(); ++i )
{
  projectPoints( Mat(objectPoints[i]), rvecs[i], tvecs[i], cameraMatrix,  // project
                                       distCoeffs, imagePoints2);
  err = norm(Mat(imagePoints[i]), Mat(imagePoints2), CV_L2);              // difference

  int n = (int)objectPoints[i].size();
  perViewErrors[i] = (float) std::sqrt(err*err/n);                        // save for this view
  totalErr        += err*err;                                             // sum it up
  totalPoints     += n;
}

return std::sqrt(totalErr/totalPoints);              // calculate the arithmetical mean
}
```

Results

Let there be `this input chessboard pattern` that has a size of 9 X 6. I've used an AXIS IP camera to create a couple of snapshots of the board and saved it into a VID5 directory. I've put this inside the `images/CameraCalibraation` folder of my working directory and created the following `VID5.XML` file that describes which images to use:

```
<?xml version="1.0"?>
<opencv_storage>
<images>
images/CameraCalibraation/VID5/xx1.jpg
images/CameraCalibraation/VID5/xx2.jpg
images/CameraCalibraation/VID5/xx3.jpg
images/CameraCalibraation/VID5/xx4.jpg
images/CameraCalibraation/VID5/xx5.jpg
images/CameraCalibraation/VID5/xx6.jpg
images/CameraCalibraation/VID5/xx7.jpg
images/CameraCalibraation/VID5/xx8.jpg
</images>
</opencv_storage>
```

Then specified the `images/CameraCalibraation/VID5/VID5.XML` as input in the configuration file. Here's a chessboard pattern found during the runtime of the application:

After applying the distortion removal we get:

The same works for `this asymmetrical circle pattern` by setting the input width to 4 and height to 11. This time I've used a live camera feed by specifying its ID ("1") for the input. Here's, how a detected pattern should look:

In both cases in the specified output XML/YAML file you'll find the camera and distortion coefficients matrices:

```
<Camera_Matrix type_id="opencv-matrix">
<rows>3</rows>
<cols>3</cols>
<dt>d</dt>
```

CHAPTER

SIX

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 yor 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!

-

  Title: *Detecting corners location in subpixeles*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  Is pixel resolution enough? Here we learn a simple method to improve our accuracy.

-

  Title: *Feature Detection*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  In this tutorial, you will use *features2d* to detect interest points.

-

  Title: *Feature Description*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  In this tutorial, you will use *features2d* to calculate feature vectors.

-

  Title: *Feature Matching with FLANN*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  In this tutorial, you will use the FLANN library to make a fast matching.

-

  Title: *Features2D + Homography to find a known object*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  In this tutorial, you will use *features2d* and *calib3d* to detect an object in a scene.

-

  Title: *Detection of planar objects*
  Compatibility: > OpenCV 2.0
  Author: Victor Eruhimov
  You will use *features2d* and *calib3d* modules for detecting known planar objects in scenes.

6.1 Feature Description

Goal

In this tutorial you will learn how to:

- Use the DescriptorExtractor interface in order to find the feature vector correspondent to the keypoints. Specifically:
  - Use SurfDescriptorExtractor and its function compute to perform the required calculations.
  - Use a BFMatcher to match the features vector
  - Use the function drawMatches to draw the detected matches.

Theory

Code

This tutorial code's is shown lines below. You can also download it from here

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/nonfree/features2d.hpp"

using namespace cv;

void readme();

/** @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
   { return -1; }

  Mat img_1 = imread( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
  Mat img_2 = imread( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

  if( !img_1.data || !img_2.data )
   { return -1; }

  //-- Step 1: Detect the keypoints using SURF Detector
  int minHessian = 400;

  SurfFeatureDetector detector( minHessian );

  std::vector<KeyPoint> keypoints_1, keypoints_2;

  detector.detect( img_1, keypoints_1 );
  detector.detect( img_2, keypoints_2 );

  //-- Step 2: Calculate descriptors (feature vectors)
  SurfDescriptorExtractor extractor;
```

```
  Mat descriptors_1, descriptors_2;

  extractor.compute( img_1, keypoints_1, descriptors_1 );
  extractor.compute( img_2, keypoints_2, descriptors_2 );

  //-- Step 3: Matching descriptor vectors with a brute force matcher
  BFMatcher matcher(NORM_L2);
  std::vector< DMatch > matches;
  matcher.match( descriptors_1, descriptors_2, matches );

  //-- Draw matches
  Mat img_matches;
  drawMatches( img_1, keypoints_1, img_2, keypoints_2, matches, img_matches );

  //-- Show detected matches
  imshow("Matches", img_matches );

  waitKey(0);

  return 0;
  }

/** @function readme */
void readme()
{ std::cout << " Usage: ./SURF_descriptor <img1> <img2>" << std::endl; }
```

Explanation

Result

1. Here is the result after applying the BruteForce matcher between the two original images:

6.2 Harris corner detector

Goal

In this tutorial you will learn:

- What features are and why they are important
- Use the function cornerHarris to detect corners using the Harris-Stephens method.

Theory

What is a feature?

- In computer vision, usually we need to find matching points between different frames of an environment. Why? If we know how two images relate to each other, we can use *both* images to extract information of them.
- When we say **matching points** we are referring, in a general sense, to *characteristics* in the scene that we can recognize easily. We call these characteristics **features**.
- **So, what characteristics should a feature have?**
  - It must be *uniquely recognizable*

Types of Image Features

To mention a few:

- Edges
- **Corners** (also known as interest points)
- Blobs (also known as regions of interest)

In this tutorial we will study the *corner* features, specifically.

Why is a corner so special?

- Because, since it is the intersection of two edges, it represents a point in which the directions of these two edges *change*. Hence, the gradient of the image (in both directions) have a high variation, which can be used to detect it.

How does it work?

- Let's look for corners. Since corners represents a variation in the gradient in the image, we will look for this "variation".
- Consider a grayscale image I. We are going to sweep a window $w(x, y)$ (with displacements u in the x direction and v in the right direction) I and will calculate the variation of intensity.

$$E(u, v) = \sum_{x;y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$$

  where:

  - $w(x, y)$ is the window at position (x, y)

  - $I(x,y)$ is the intensity at (x,y)
  - $I(x+u,y+v)$ is the intensity at the moved window $(x+u,y+v)$
- Since we are looking for windows with corners, we are looking for windows with a large variation in intensity. Hence, we have to maximize the equation above, specifically the term:

$$\sum_{x;y}[I(x+u,y+v)-I(x,y)]^2$$

- Using *Taylor expansion*:

$$E(u,v) \approx \sum_{x;y}[I(x,y)+uI_x+vI_y-I(x,y)]^2$$

- Expanding the equation and cancelling properly:

$$E(u,v) \approx \sum_{x;y} u^2I_x^2+2uvI_xI_y+v^2I_y^2$$

- Which can be expressed in a matrix form as:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} \left(\sum_{x;y} w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

- Let's denote:

$$M = \sum_{x;y} w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

- So, our equation now is:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

- A score is calculated for each window, to determine if it can possibly contain a corner:

$$R = det(M) - k(trace(M))^2$$

  where:

  - det(M) = $\lambda_1\lambda_2$
  - trace(M) = $\lambda_1+\lambda_2$

  a window with a score R greater than a certain value is considered a "corner"

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

6.3 Feature Matching with FLANN

Goal

In this tutorial you will learn how to:

- Use the FlannBasedMatcher interface in order to perform a quick and efficient matching by using the FLANN (*Fast Approximate Nearest Neighbor Search Library*)

Theory

Code

This tutorial code's is shown lines below. You can also download it from here

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace cv;

void readme();

/** @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
  { readme(); return -1; }

  Mat img_1 = imread( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
```

```
Mat img_2 = imread( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

if( !img_1.data || !img_2.data )
{ std::cout<< " --(!) Error reading images " << std::endl; return -1; }

//-- Step 1: Detect the keypoints using SURF Detector
int minHessian = 400;

SurfFeatureDetector detector( minHessian );

std::vector<KeyPoint> keypoints_1, keypoints_2;

detector.detect( img_1, keypoints_1 );
detector.detect( img_2, keypoints_2 );

//-- Step 2: Calculate descriptors (feature vectors)
SurfDescriptorExtractor extractor;

Mat descriptors_1, descriptors_2;

extractor.compute( img_1, keypoints_1, descriptors_1 );
extractor.compute( img_2, keypoints_2, descriptors_2 );

//-- Step 3: Matching descriptor vectors using FLANN matcher
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match( descriptors_1, descriptors_2, matches );

double max_dist = 0; double min_dist = 100;

//-- Quick calculation of max and min distances between keypoints
for( int i = 0; i < descriptors_1.rows; i++ )
{ double dist = matches[i].distance;
  if( dist < min_dist ) min_dist = dist;
  if( dist > max_dist ) max_dist = dist;
}

printf("-- Max dist : %f \n", max_dist );
printf("-- Min dist : %f \n", min_dist );

//-- Draw only "good" matches (i.e. whose distance is less than 2*min_dist )
//-- PS.- radiusMatch can also be used here.
std::vector< DMatch > good_matches;

for( int i = 0; i < descriptors_1.rows; i++ )
{ if( matches[i].distance <= 2*min_dist )
  { good_matches.push_back( matches[i]); }
}

//-- Draw only "good" matches
Mat img_matches;
drawMatches( img_1, keypoints_1, img_2, keypoints_2,
             good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
             vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

//-- Show detected matches
imshow( "Good Matches", img_matches );
```

```
  for( int i = 0; i < good_matches.size(); i++ )
  { printf( "-- Good Match [%d] Keypoint 1: %d  -- Keypoint 2: %d  \n", i, good_matches[i].queryIdx, good_matches[i].t

  waitKey(0);

  return 0;
}

/** @function readme */
void readme()
{ std::cout << " Usage: ./SURF_FlannMatcher <img1> <img2>" << std::endl; }
```

Explanation

Result

1. Here is the result of the feature detection applied to the first image:

2. Additionally, we get as console output the keypoints filtered:

6.4 Features2D + Homography to find a known object

Goal

In this tutorial you will learn how to:

- Use the function findHomography to find the transform between matched keypoints.
- Use the function perspectiveTransform to map the points.

Theory

Code

This tutorial code's is shown lines below. You can also download it from here

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"

using namespace cv;

void readme();

/** @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
```

```
{ readme(); return -1; }

Mat img_object = imread( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
Mat img_scene = imread( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

if( !img_object.data || !img_scene.data )
{ std::cout<< " --(!) Error reading images " << std::endl; return -1; }

//-- Step 1: Detect the keypoints using SURF Detector
int minHessian = 400;

SurfFeatureDetector detector( minHessian );

std::vector<KeyPoint> keypoints_object, keypoints_scene;

detector.detect( img_object, keypoints_object );
detector.detect( img_scene, keypoints_scene );

//-- Step 2: Calculate descriptors (feature vectors)
SurfDescriptorExtractor extractor;

Mat descriptors_object, descriptors_scene;

extractor.compute( img_object, keypoints_object, descriptors_object );
extractor.compute( img_scene, keypoints_scene, descriptors_scene );

//-- Step 3: Matching descriptor vectors using FLANN matcher
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match( descriptors_object, descriptors_scene, matches );

double max_dist = 0; double min_dist = 100;

//-- Quick calculation of max and min distances between keypoints
for( int i = 0; i < descriptors_object.rows; i++ )
{ double dist = matches[i].distance;
  if( dist < min_dist ) min_dist = dist;
  if( dist > max_dist ) max_dist = dist;
}

printf("-- Max dist : %f \n", max_dist );
printf("-- Min dist : %f \n", min_dist );

//-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
std::vector< DMatch > good_matches;

for( int i = 0; i < descriptors_object.rows; i++ )
{ if( matches[i].distance < 3*min_dist )
   { good_matches.push_back( matches[i]); }
}

Mat img_matches;
drawMatches( img_object, keypoints_object, img_scene, keypoints_scene,
             good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
             vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

//-- Localize the object
std::vector<Point2f> obj;
```

```
std::vector<Point2f> scene;

for( int i = 0; i < good_matches.size(); i++ )
{
  //-- Get the keypoints from the good matches
  obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
  scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
}

Mat H = findHomography( obj, scene, CV_RANSAC );

//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] = cvPoint( 0, img_object.rows );
std::vector<Point2f> scene_corners(4);

perspectiveTransform( obj_corners, scene_corners, H);

//-- Draw lines between the corners (the mapped object in the scene - image_2 )
line( img_matches, scene_corners[0] + Point2f( img_object.cols, 0), scene_corners[1] + Point2f( img_object.cols, 0),
line( img_matches, scene_corners[1] + Point2f( img_object.cols, 0), scene_corners[2] + Point2f( img_object.cols, 0),
line( img_matches, scene_corners[2] + Point2f( img_object.cols, 0), scene_corners[3] + Point2f( img_object.cols, 0),
line( img_matches, scene_corners[3] + Point2f( img_object.cols, 0), scene_corners[0] + Point2f( img_object.cols, 0),

//-- Show detected matches
imshow( "Good Matches & Object detection", img_matches );

waitKey(0);
return 0;
}

/** @function readme */
void readme()
{ std::cout << " Usage: ./SURF_descriptor <img1> <img2>" << std::endl; }
```

Explanation

Result

1. And here is the result for the detected object (highlighted in green)

6.5 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

6.6 Creating yor 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

```
/**
 * @function cornerDetector_Demo.cpp
 * @brief Demo code for detecting corners using OpenCV built-in functions
 * @author OpenCV team
 */
#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);

const char* myHarris_window = "My Harris corner detector";
const 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, 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<Vec6f>(j, i)[0];
            float lambda_2 = myHarris_dst.at<Vec6f>(j, i)[1];
            Mc.at<float>(j,i) = lambda_1*lambda_2 - 0.04f*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_func
  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 )*mySh
              { circle( myShiTomasi_copy, Point(i,j), 4, Scalar( rng.uniform(0,255), rng.uniform(0,255), rng.uniform(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_
              { circle( myHarris_copy, Point(i,j), 4, Scalar( rng.uniform(0,255), rng.uniform(0,255), rng.uniform(0,25
          }
     }
  imshow( myHarris_window, myHarris_copy );
}
```

Explanation

Result

6.7 Detecting corners location in subpixeles

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 neeed 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:

6.8 Feature Detection

Goal

In this tutorial you will learn how to:

- Use the FeatureDetector interface in order to find interest points. Specifically:
  - Use the SurfFeatureDetector and its function detect to perform the detection process
  - Use the function drawKeypoints to draw the detected keypoints

Theory

Code

This tutorial code's is shown lines below. You can also download it from here

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace cv;

void readme();

/** @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
  { readme(); return -1; }

  Mat img_1 = imread( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
  Mat img_2 = imread( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

  if( !img_1.data || !img_2.data )
  { std::cout<< " --(!) Error reading images " << std::endl; return -1; }

  //-- Step 1: Detect the keypoints using SURF Detector
  int minHessian = 400;
```

```
SurfFeatureDetector detector( minHessian );

std::vector<KeyPoint> keypoints_1, keypoints_2;

detector.detect( img_1, keypoints_1 );
detector.detect( img_2, keypoints_2 );

//-- Draw keypoints
Mat img_keypoints_1; Mat img_keypoints_2;

drawKeypoints( img_1, keypoints_1, img_keypoints_1, Scalar::all(-1), DrawMatchesFlags::DEFAULT );
drawKeypoints( img_2, keypoints_2, img_keypoints_2, Scalar::all(-1), DrawMatchesFlags::DEFAULT );

//-- Show detected (drawn) keypoints
imshow("Keypoints 1", img_keypoints_1 );
imshow("Keypoints 2", img_keypoints_2 );

waitKey(0);

return 0;
}

/** @function readme */
void readme()
{ std::cout << " Usage: ./SURF_detector <img1> <img2>" << std::endl; }
```

Explanation

Result

1. Here is the result of the feature detection applied to the first image:

2. And here is the result for the second image:

6.9 Feature Matching with FLANN

Goal

In this tutorial you will learn how to:

- Use the FlannBasedMatcher interface in order to perform a quick and efficient matching by using the FLANN (*Fast Approximate Nearest Neighbor Search Library*)

Theory

Code

This tutorial code's is shown lines below. You can also download it from here

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace cv;

void readme();

/** @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
  { readme(); return -1; }

  Mat img_1 = imread( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
  Mat img_2 = imread( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

  if( !img_1.data || !img_2.data )
```

```
{ std::cout<< " --(!) Error reading images " << std::endl; return -1; }

//-- Step 1: Detect the keypoints using SURF Detector
int minHessian = 400;

SurfFeatureDetector detector( minHessian );

std::vector<KeyPoint> keypoints_1, keypoints_2;

detector.detect( img_1, keypoints_1 );
detector.detect( img_2, keypoints_2 );

//-- Step 2: Calculate descriptors (feature vectors)
SurfDescriptorExtractor extractor;

Mat descriptors_1, descriptors_2;

extractor.compute( img_1, keypoints_1, descriptors_1 );
extractor.compute( img_2, keypoints_2, descriptors_2 );

//-- Step 3: Matching descriptor vectors using FLANN matcher
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match( descriptors_1, descriptors_2, matches );

double max_dist = 0; double min_dist = 100;

//-- Quick calculation of max and min distances between keypoints
for( int i = 0; i < descriptors_1.rows; i++ )
{ double dist = matches[i].distance;
  if( dist < min_dist ) min_dist = dist;
  if( dist > max_dist ) max_dist = dist;
}

printf("-- Max dist : %f \n", max_dist );
printf("-- Min dist : %f \n", min_dist );

//-- Draw only "good" matches (i.e. whose distance is less than 2*min_dist )
//-- PS.- radiusMatch can also be used here.
std::vector< DMatch > good_matches;

for( int i = 0; i < descriptors_1.rows; i++ )
{ if( matches[i].distance <= 2*min_dist )
  { good_matches.push_back( matches[i]); }
}

//-- Draw only "good" matches
Mat img_matches;
drawMatches( img_1, keypoints_1, img_2, keypoints_2,
             good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
             vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

//-- Show detected matches
imshow( "Good Matches", img_matches );

for( int i = 0; i < good_matches.size(); i++ )
{ printf( "-- Good Match [%d] Keypoint 1: %d  -- Keypoint 2: %d  \n", i, good_matches[i].queryIdx, good_matches[i].t
```

```
 waitKey(0);

 return 0;
}

/** @function readme */
void readme()
{ std::cout << " Usage: ./SURF_FlannMatcher <img1> <img2>" << std::endl; }
```

Explanation

Result

1. Here is the result of the feature detection applied to the first image:

2. Additionally, we get as console output the keypoints filtered:

6.10 Features2D + Homography to find a known object

Goal

In this tutorial you will learn how to:

- Use the function findHomography to find the transform between matched keypoints.
- Use the function perspectiveTransform to map the points.

Theory

Code

This tutorial code's is shown lines below. You can also download it from here

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"

using namespace cv;

void readme();

/** @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
```

```
{ readme(); return -1; }

Mat img_object = imread( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
Mat img_scene = imread( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

if( !img_object.data || !img_scene.data )
{ std::cout<< " --(!) Error reading images " << std::endl; return -1; }

//-- Step 1: Detect the keypoints using SURF Detector
int minHessian = 400;

SurfFeatureDetector detector( minHessian );

std::vector<KeyPoint> keypoints_object, keypoints_scene;

detector.detect( img_object, keypoints_object );
detector.detect( img_scene, keypoints_scene );

//-- Step 2: Calculate descriptors (feature vectors)
SurfDescriptorExtractor extractor;

Mat descriptors_object, descriptors_scene;

extractor.compute( img_object, keypoints_object, descriptors_object );
extractor.compute( img_scene, keypoints_scene, descriptors_scene );

//-- Step 3: Matching descriptor vectors using FLANN matcher
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match( descriptors_object, descriptors_scene, matches );

double max_dist = 0; double min_dist = 100;

//-- Quick calculation of max and min distances between keypoints
for( int i = 0; i < descriptors_object.rows; i++ )
{ double dist = matches[i].distance;
  if( dist < min_dist ) min_dist = dist;
  if( dist > max_dist ) max_dist = dist;
}

printf("-- Max dist : %f \n", max_dist );
printf("-- Min dist : %f \n", min_dist );

//-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
std::vector< DMatch > good_matches;

for( int i = 0; i < descriptors_object.rows; i++ )
{ if( matches[i].distance < 3*min_dist )
   { good_matches.push_back( matches[i]); }
}

Mat img_matches;
drawMatches( img_object, keypoints_object, img_scene, keypoints_scene,
             good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
             vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

//-- Localize the object
std::vector<Point2f> obj;
```

```
std::vector<Point2f> scene;

for( int i = 0; i < good_matches.size(); i++ )
{
  //-- Get the keypoints from the good matches
  obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
  scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
}

Mat H = findHomography( obj, scene, CV_RANSAC );

//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] = cvPoint( 0, img_object.rows );
std::vector<Point2f> scene_corners(4);

perspectiveTransform( obj_corners, scene_corners, H);

//-- Draw lines between the corners (the mapped object in the scene - image_2 )
line( img_matches, scene_corners[0] + Point2f( img_object.cols, 0), scene_corners[1] + Point2f( img_object.cols, 0),
line( img_matches, scene_corners[1] + Point2f( img_object.cols, 0), scene_corners[2] + Point2f( img_object.cols, 0),
line( img_matches, scene_corners[2] + Point2f( img_object.cols, 0), scene_corners[3] + Point2f( img_object.cols, 0),
line( img_matches, scene_corners[3] + Point2f( img_object.cols, 0), scene_corners[0] + Point2f( img_object.cols, 0),

//-- Show detected matches
imshow( "Good Matches & Object detection", img_matches );

waitKey(0);
return 0;
}

/** @function readme */
void readme()
{ std::cout << " Usage: ./SURF_descriptor <img1> <img2>" << std::endl; }
```

Explanation

Result

1. And here is the result for the detected object (highlighted in green)

6.11 Detection of planar objects

The goal of this tutorial is to learn how to use *features2d* and *calib3d* modules for detecting known planar objects in scenes.

Test data: use images in your data folder, for instance, `box.png` and `box_in_scene.png`.

1. Create a new console project. Read two input images.

```
Mat img1 = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
Mat img2 = imread(argv[2], CV_LOAD_IMAGE_GRAYSCALE);
```

2. Detect keypoints in both images.

```
// detecting keypoints
FastFeatureDetector detector(15);
vector<KeyPoint> keypoints1;
detector.detect(img1, keypoints1);

... // do the same for the second image
```

3. Compute descriptors for each of the keypoints.

```
// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1;
extractor.compute(img1, keypoints1, descriptors1);

... // process keypoints from the second image as well
```

4. Now, find the closest matches between descriptors from the first image to the second:

```
// matching descriptors
BruteForceMatcher<L2<float> > matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
```

5. Visualize the results:

```
// drawing the results
namedWindow("matches", 1);
Mat img_matches;
drawMatches(img1, keypoints1, img2, keypoints2, matches, img_matches);
imshow("matches", img_matches);
waitKey(0);
```

6. Find the homography transformation between two sets of points:

```
vector<Point2f> points1, points2;
// fill the arrays with the points
....
Mat H = findHomography(Mat(points1), Mat(points2), CV_RANSAC, ransacReprojThreshold);
```

7. Create a set of inlier matches and draw them. Use perspectiveTransform function to map points with homography:

   Mat points1Projected; perspectiveTransform(Mat(points1), points1Projected, H);

8. Use `drawMatches` for drawing inliers.

CHAPTER

SEVEN

VIDEO MODULE. VIDEO ANALYSIS

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

Note: Unfortunetly we have no tutorials into this section. And you can help us with that, since OpenCV is a community effort. If you have a tutorial suggestion or you have written a tutorial yourself (or coded a sample code) that you would like to see here, please contact follow these instructions: *How to write a tutorial for OpenCV* and How to contribute.

CHAPTER

EIGHT

OBJDETECT MODULE. OBJECT DETECTION

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

-

  Title: *Cascade Classifier*
  Compatibility: > OpenCV 2.0
  Author: Ana Huamán
  Here we learn how to use *objdetect* to find objects in our images or videos

8.1 Cascade Classifier

Goal

In this tutorial you will learn how to:

- Use the CascadeClassifier class to detect objects in a video stream. Particularly, we will use the functions:
  - load to load a .xml classifier file. It can be either a Haar or a LBP classifer
  - detectMultiScale to perform the detection.

Theory

Code

This tutorial code's is shown lines below. You can also download it from here . The second version (using LBP for face detection) can be found here

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

#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;

/** Function Headers */
void detectAndDisplay( Mat frame );

/** Global variables */
String face_cascade_name = "haarcascade_frontalface_alt.xml";
String eyes_cascade_name = "haarcascade_eye_tree_eyeglasses.xml";
CascadeClassifier face_cascade;
CascadeClassifier eyes_cascade;
string window_name = "Capture - Face detection";
RNG rng(12345);

/** @function main */
int main( int argc, const char** argv )
{
  CvCapture* capture;
  Mat frame;

  //-- 1. Load the cascades
  if( !face_cascade.load( face_cascade_name ) ){ printf("--(!)Error loading\n"); return -1; };
  if( !eyes_cascade.load( eyes_cascade_name ) ){ printf("--(!)Error loading\n"); return -1; };

  //-- 2. Read the video stream
  capture = cvCaptureFromCAM( -1 );
  if( capture )
  {
    while( true )
    {
  frame = cvQueryFrame( capture );
```

```
   //-- 3. Apply the classifier to the frame
       if( !frame.empty() )
       { detectAndDisplay( frame ); }
       else
       { printf(" --(!) No captured frame -- Break!"); break; }

       int c = waitKey(10);
       if( (char)c == 'c' ) { break; }
      }
   }
   return 0;
 }

/** @function detectAndDisplay */
void detectAndDisplay( Mat frame )
{
  std::vector<Rect> faces;
  Mat frame_gray;

  cvtColor( frame, frame_gray, CV_BGR2GRAY );
  equalizeHist( frame_gray, frame_gray );

  //-- Detect faces
  face_cascade.detectMultiScale( frame_gray, faces, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE, Size(30, 30) );

  for( int i = 0; i < faces.size(); i++ )
  {
    Point center( faces[i].x + faces[i].width*0.5, faces[i].y + faces[i].height*0.5 );
    ellipse( frame, center, Size( faces[i].width*0.5, faces[i].height*0.5), 0, 0, 360, Scalar( 255, 0, 255 ), 4, 8, 0

    Mat faceROI = frame_gray( faces[i] );
    std::vector<Rect> eyes;

    //-- In each face, detect eyes
    eyes_cascade.detectMultiScale( faceROI, eyes, 1.1, 2, 0 |CV_HAAR_SCALE_IMAGE, Size(30, 30) );

    for( int j = 0; j < eyes.size(); j++ )
     {
       Point center( faces[i].x + eyes[j].x + eyes[j].width*0.5, faces[i].y + eyes[j].y + eyes[j].height*0.5 );
       int radius = cvRound( (eyes[j].width + eyes[j].height)*0.25 );
       circle( frame, center, radius, Scalar( 255, 0, 0 ), 4, 8, 0 );
     }
  }
  //-- Show what you got
  imshow( window_name, frame );
 }
```

Explanation

Result

1. Here is the result of running the code above and using as input the video stream of a build-in webcam:

Remember to copy the files *haarcascade_frontalface_alt.xml* and *haarcascade_eye_tree_eyeglasses.xml* in your current directory. They are located in *opencv/data/haarcascades*

2. This is the result of using the file *lbpcascade_frontalface.xml* (LBP trained) for the face detection. For the eyes we keep using the file used in the tutorial.

Capture - Face detection

CHAPTER

NINE

ML MODULE. MACHINE LEARNING

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

-

  Title: *Introduction to Support Vector Machines*
  Compatibility: > OpenCV 2.0
  Author: Fernando Iglesias García
  Learn what a Suport Vector Machine is.

-

  Title: *Support Vector Machines for Non-Linearly Separable Data*
  Compatibility: > OpenCV 2.0
  Author: Fernando Iglesias García
  Here you will learn how to define the optimization problem for SVMs when it is not possible to separate linearly the training data.

9.1 Introduction to Support Vector Machines

Goal

In this tutorial you will learn how to:

- Use the OpenCV functions CvSVM::train to build a classifier based on SVMs and CvSVM::predict to test its performance.

What is a SVM?

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (*supervised learning*), the algorithm outputs an optimal hyperplane which categorizes new examples.

In which sense is the hyperplane obtained optimal? Let's consider the following simple problem:

> For a linearly separable set of 2D-points which belong to one of two classes, find a separating straight line.

Note: In this example we deal with lines and points in the Cartesian plane instead of hyperplanes and vectors in a high dimensional space. This is a simplification of the problem.It is important to understand that this is done only because our intuition is better built from examples that are easy to imagine. However, the same concepts apply to tasks where the examples to classify lie in a space whose dimension is higher than two.

In the above picture you can see that there exists multiple lines that offer a solution to the problem. Is any of them better than the others? We can intuitively define a criterion to estimate the worth of the lines:

> A line is bad if it passes too close to the points because it will be noise sensitive and it will not generalize correctly. Therefore, our goal should be to find the line passing as far as possible from all points.

Then, the operation of the SVM algorithm is based on finding the hyperplane that gives the largest minimum distance to the training examples. Twice, this distance receives the important name of **margin** within SVM's theory. Therefore, the optimal separating hyperplane *maximizes* the margin of the training data.

How is the optimal hyperplane computed?

Let's introduce the notation used to define formally a hyperplane:

$$f(x) = \beta_0 + \beta^T x,$$

where β is known as the *weight vector* and β_0 as the *bias*.

See Also:

A more in depth description of this and hyperplanes you can find in the section 4.5 (*Seperating Hyperplanes*) of the book: *Elements of Statistical Learning* by T. Hastie, R. Tibshirani and J. H. Friedman.

The optimal hyperplane can be represented in an infinite number of different ways by scaling of β and β_0. As a matter of convention, among all the possible representations of the hyperplane, the one chosen is

$$|\beta_0 + \beta^T x| = 1$$

where x symbolizes the training examples closest to the hyperplane. In general, the training examples that are closest to the hyperplane are called **support vectors**. This representation is known as the **canonical hyperplane**.

Now, we use the result of geometry that gives the distance between a point x and a hyperplane (β, β_0):

$$\text{distance} = \frac{|\beta_0 + \beta^T x|}{||\beta||}.$$

In particular, for the canonical hyperplane, the numerator is equal to one and the distance to the support vectors is

$$\text{distance}_{\text{support vectors}} = \frac{|\beta_0 + \beta^T x|}{||\beta||} = \frac{1}{||\beta||}.$$

Recall that the margin introduced in the previous section, here denoted as M, is twice the distance to the closest examples:

$$M = \frac{2}{||\beta||}$$

Finally, the problem of maximizing M is equivalent to the problem of minimizing a function $L(\beta)$ subject to some constraints. The constraints model the requirement for the hyperplane to classify correctly all the training examples x_i. Formally,

$$\min_{\beta, \beta_0} L(\beta) = \frac{1}{2}||\beta||^2 \text{ subject to } y_i(\beta^T x_i + \beta_0) \geq 1 \ \forall i,$$

where y_i represents each of the labels of the training examples.

This is a problem of Lagrangian optimization that can be solved using Lagrange multipliers to obtain the weight vector β and the bias β_0 of the optimal hyperplane.

Source Code

```cpp
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/ml/ml.hpp>

using namespace cv;

int main()
{
    // Data for visual representation
    int width = 512, height = 512;
    Mat image = Mat::zeros(height, width, CV_8UC3);

    // Set up training data
    float labels[4] = {1.0, -1.0, -1.0, -1.0};
    Mat labelsMat(4, 1, CV_32FC1, labels);

    float trainingData[4][2] = { {501, 10}, {255, 10}, {501, 255}, {10, 501} };
    Mat trainingDataMat(4, 2, CV_32FC1, trainingData);

    // Set up SVM's parameters
    CvSVMParams params;
    params.svm_type    = CvSVM::C_SVC;
    params.kernel_type = CvSVM::LINEAR;
    params.term_crit   = cvTermCriteria(CV_TERMCRIT_ITER, 100, 1e-6);

    // Train the SVM
    CvSVM SVM;
    SVM.train(trainingDataMat, labelsMat, Mat(), Mat(), params);

    Vec3b green(0,255,0), blue (255,0,0);
    // Show the decision regions given by the SVM
    for (int i = 0; i < image.rows; ++i)
        for (int j = 0; j < image.cols; ++j)
        {
            Mat sampleMat = (Mat_<float>(1,2) << i,j);
            float response = SVM.predict(sampleMat);

            if (response == 1)
                image.at<Vec3b>(j, i)  = green;
            else if (response == -1)
                 image.at<Vec3b>(j, i)  = blue;
        }

    // Show the training data
    int thickness = -1;
    int lineType = 8;
    circle( image, Point(501,  10), 5, Scalar(  0,   0,   0), thickness, lineType);
    circle( image, Point(255,  10), 5, Scalar(255, 255, 255), thickness, lineType);
    circle( image, Point(501, 255), 5, Scalar(255, 255, 255), thickness, lineType);
    circle( image, Point( 10, 501), 5, Scalar(255, 255, 255), thickness, lineType);

    // Show support vectors
    thickness = 2;
    lineType  = 8;
    int c     = SVM.get_support_vector_count();

    for (int i = 0; i < c; ++i)
    {
```

```
        const float* v = SVM.get_support_vector(i);
        circle( image,  Point( (int) v[0], (int) v[1]),   6,  Scalar(128, 128, 128), thickness, lineType);
    }

    imwrite("result.png", image);        // save the image

    imshow("SVM Simple Example", image); // show it to the user
    waitKey(0);

}
```

Explanation

1. **Set up the training data**

   The training data of this exercise is formed by a set of labeled 2D-points that belong to one of two different classes; one of the classes consists of one point and the other of three points.

   ```
   float labels[4] = {1.0, -1.0, -1.0, -1.0};
   float trainingData[4][2] = {{501, 10}, {255, 10}, {501, 255}, {10, 501}};
   ```

   The function CvSVM::train that will be used afterwards requires the training data to be stored as Mat objects of floats. Therefore, we create these objects from the arrays defined above:

   ```
   Mat trainingDataMat(3, 2, CV_32FC1, trainingData);
   Mat labelsMat      (3, 1, CV_32FC1, labels);
   ```

2. **Set up SVM's parameters**

   In this tutorial we have introduced the theory of SVMs in the most simple case, when the training examples are spread into two classes that are linearly separable. However, SVMs can be used in a wide variety of problems (e.g. problems with non-linearly separable data, a SVM using a kernel function to raise the dimensionality of the examples, etc). As a consequence of this, we have to define some parameters before training the SVM. These parameters are stored in an object of the class CvSVMParams .

   ```
   CvSVMParams params;
   params.svm_type    = CvSVM::C_SVC;
   params.kernel_type = CvSVM::LINEAR;
   params.term_crit   = cvTermCriteria(CV_TERMCRIT_ITER, 100, 1e-6);
   ```

   - *Type of SVM*. We choose here the type **CvSVM::C_SVC** that can be used for n-class classification (n $\geq$ 2). This parameter is defined in the attribute *CvSVMParams.svm_type*.

     Note: The important feature of the type of SVM **CvSVM::C_SVC** deals with imperfect separation of classes (i.e. when the training data is non-linearly separable). This feature is not important here since the data is linearly separable and we chose this SVM type only for being the most commonly used.

   - *Type of SVM kernel*. We have not talked about kernel functions since they are not interesting for the training data we are dealing with. Nevertheless, let's explain briefly now the main idea behind a kernel function. It is a mapping done to the training data to improve its resemblance to a linearly separable set of data. This mapping consists of increasing the dimensionality of the data and is done efficiently using a kernel function. We choose here the type **CvSVM::LINEAR** which means that no mapping is done. This parameter is defined in the attribute *CvSVMParams.kernel_type*.

   - *Termination criteria of the algorithm*. The SVM training procedure is implemented solving a constrained quadratic optimization problem in an **iterative** fashion. Here we specify a maximum number of iterations

and a tolerance error so we allow the algorithm to finish in less number of steps even if the optimal hyperplane has not been computed yet. This parameter is defined in a structure cvTermCriteria.

3. **Train the SVM**

   We call the method CvSVM::train to build the SVM model.

```
CvSVM SVM;
SVM.train(trainingDataMat, labelsMat, Mat(), Mat(), params);
```

4. **Regions classified by the SVM**

   The method CvSVM::predict is used to classify an input sample using a trained SVM. In this example we have used this method in order to color the space depending on the prediction done by the SVM. In other words, an image is traversed interpreting its pixels as points of the Cartesian plane. Each of the points is colored depending on the class predicted by the SVM; in green if it is the class with label 1 and in blue if it is the class with label -1.

```
Vec3b green(0,255,0), blue (255,0,0);

for (int i = 0; i < image.rows; ++i)
    for (int j = 0; j < image.cols; ++j)
    {
    Mat sampleMat = (Mat_<float>(1,2) << i,j);
    float response = SVM.predict(sampleMat);

    if (response == 1)
       image.at<Vec3b>(j, i)  = green;
    else
    if (response == -1)
       image.at<Vec3b>(j, i)  = blue;
    }
```

5. **Support vectors**

   We use here a couple of methods to obtain information about the support vectors. The method CvSVM::get_support_vector_count outputs the total number of support vectors used in the problem and with the method CvSVM::get_support_vector we obtain each of the support vectors using an index. We have used this methods here to find the training examples that are support vectors and highlight them.

```
int c     = SVM.get_support_vector_count();

for (int i = 0; i < c; ++i)
{
const float* v = SVM.get_support_vector(i); // get and then highlight with grayscale
circle(   image,  Point( (int) v[0], (int) v[1]),   6,  Scalar(128, 128, 128), thickness, lineType);
}
```

Results

- The code opens an image and shows the training examples of both classes. The points of one class are represented with white circles and black ones are used for the other class.
- The SVM is trained and used to classify all the pixels of the image. This results in a division of the image in a blue region and a green region. The boundary between both regions is the optimal separating hyperplane.
- Finally the support vectors are shown using gray rings around the training examples.

9.2 Support Vector Machines for Non-Linearly Separable Data

Goal

In this tutorial you will learn how to:

- Define the optimization problem for SVMs when it is not possible to separate linearly the training data.
- How to configure the parameters in CvSVMParams to adapt your SVM for this class of problems.

Motivation

Why is it interesting to extend the SVM optimation problem in order to handle non-linearly separable training data? Most of the applications in which SVMs are used in computer vision require a more powerful tool than a simple linear classifier. This stems from the fact that in these tasks **the training data can be rarely separated using an hyperplane**. Consider one of these tasks, for example, face detection. The training data in this case is composed by a set of images that are faces and another set of images that are non-faces (*every other thing in the world except from faces*). This training data is too complex so as to find a representation of each sample (*feature vector*) that could make the whole set of faces linearly separable from the whole set of non-faces.

Extension of the Optimization Problem

Remember that using SVMs we obtain a separating hyperplane. Therefore, since the training data is now non-linearly separable, we must admit that the hyperplane found will misclassify some of the samples. This *misclassification* is a new variable in the optimization that must be taken into account. The new model has to include both the old requirement of finding the hyperplane that gives the biggest margin and the new one of generalizing the training data correctly by not allowing too many classification errors. We start here from the formulation of the optimization problem of finding the hyperplane which maximizes the **margin** (this is explained in the *previous tutorial*):

$$\min_{\beta, \beta_0} L(\beta) = \frac{1}{2}||\beta||^2 \text{ subject to } y_i(\beta^T x_i + \beta_0) \geq 1 \ \forall i$$

There are multiple ways in which this model can be modified so it takes into account the misclassification errors. For example, one could think of minimizing the same quantity plus a constant times the number of misclassification errors in the training data, i.e.:

$$\min ||\beta||^2 + C(\text{\# misclassication errors})$$

However, this one is not a very good solution since, among some other reasons, we do not distinguish between samples that are misclassified with a small distance to their appropriate decision region or samples that are not. Therefore, a better solution will take into account the *distance of the misclassified samples to their correct decision regions*, i.e.:

$$\min ||\beta||^2 + C(\text{distance of misclassified samples to their correct regions})$$

For each sample of the training data a new parameter ξ_i is defined. Each one of these parameters contains the distance from its corresponding training sample to their correct decision region. The following picture shows non-linearly

separable training data from two classes, a separating hyperplane and the distances to their correct regions of the samples that are misclassified.

Note: Only the distances of the samples that are misclassified are shown in the picture. The distances of the rest of the samples are zero since they lay already in their correct decision region. The red and blue lines that appear on the picture are the margins to each one of the decision regions. It is very **important** to realize that each of the ξ_i goes from a misclassified training sample to the margin of its appropriate region. Finally, the new formulation for the optimization problem is:

$$\min_{\beta, \beta_0} L(\beta) = \|\beta\|^2 + C \sum_i \xi_i \text{ subject to } y_i(\beta^T x_i + \beta_0) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \; \forall i$$

How should the parameter C be chosen? It is obvious that the answer to this question depends on how the training data is distributed. Although there is no general answer, it is useful to take into account these rules:

- Large values of C give solutions with *less misclassification errors* but a *smaller margin*. Consider that in this case it is expensive to make misclassification errors. Since the aim of the optimization is to minimize the argument, few misclassifications errors are allowed.
- Small values of C give solutions with *bigger margin* and *more classification errors*. In this case the minimization does not consider that much the term of the sum so it focuses more on finding a hyperplane with big margin.

Source Code

You may also find the source code and these video file in the `samples/cpp/tutorial_code/gpu/non_linear_svms/non_linear_svm` folder of the OpenCV source library or `download it from here`.

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

#define NTRAINING_SAMPLES   100         // Number of training samples per class
```

```
#define FRAC_LINEAR_SEP     0.9f        // Fraction of samples which compose the linear separable part

using namespace cv;
using namespace std;

int main()
{
    // Data for visual representation
    const int WIDTH = 512, HEIGHT = 512;
    Mat I = Mat::zeros(HEIGHT, WIDTH, CV_8UC3);

    //--------------------- 1. Set up training data randomly --------------------------------------
    Mat trainData(2*NTRAINING_SAMPLES, 2, CV_32FC1);
    Mat labels   (2*NTRAINING_SAMPLES, 1, CV_32FC1);

    RNG rng(100); // Random value generation class

    // Set up the linearly separable part of the training data
    int nLinearSamples = (int) (FRAC_LINEAR_SEP * NTRAINING_SAMPLES);

    // Generate random points for the class 1
    Mat trainClass = trainData.rowRange(0, nLinearSamples);
    // The x coordinate of the points is in [0, 0.4)
    Mat c = trainClass.colRange(0, 1);
    rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(0.4 * WIDTH));
    // The y coordinate of the points is in [0, 1)
    c = trainClass.colRange(1,2);
    rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(HEIGHT));

    // Generate random points for the class 2
    trainClass = trainData.rowRange(2*NTRAINING_SAMPLES-nLinearSamples, 2*NTRAINING_SAMPLES);
    // The x coordinate of the points is in [0.6, 1]
    c = trainClass.colRange(0 , 1);
    rng.fill(c, RNG::UNIFORM, Scalar(0.6*WIDTH), Scalar(WIDTH));
    // The y coordinate of the points is in [0, 1)
    c = trainClass.colRange(1,2);
    rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(HEIGHT));

    //------------------ Set up the non-linearly separable part of the training data ---------------

    // Generate random points for the classes 1 and 2
    trainClass = trainData.rowRange(  nLinearSamples, 2*NTRAINING_SAMPLES-nLinearSamples);
    // The x coordinate of the points is in [0.4, 0.6)
    c = trainClass.colRange(0,1);
    rng.fill(c, RNG::UNIFORM, Scalar(0.4*WIDTH), Scalar(0.6*WIDTH));
    // The y coordinate of the points is in [0, 1)
    c = trainClass.colRange(1,2);
    rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(HEIGHT));

    //------------------------- Set up the labels for the classes ---------------------------------
    labels.rowRange(                0,   NTRAINING_SAMPLES).setTo(1);  // Class 1
    labels.rowRange(NTRAINING_SAMPLES, 2*NTRAINING_SAMPLES).setTo(2);  // Class 2

    //------------------------ 2. Set up the support vector machines parameters --------------------
    CvSVMParams params;
    params.svm_type    = SVM::C_SVC;
    params.C           = 0.1;
    params.kernel_type = SVM::LINEAR;
```

```
    params.term_crit   = TermCriteria(CV_TERMCRIT_ITER, (int)1e7, 1e-6);

    //------------------------ 3. Train the svm ----------------------------------------------------
    cout << "Starting training process" << endl;
    CvSVM svm;
    svm.train(trainData, labels, Mat(), Mat(), params);
    cout << "Finished training process" << endl;

    //------------------------ 4. Show the decision regions ----------------------------------------
    Vec3b green(0,100,0), blue (100,0,0);
    for (int i = 0; i < I.rows; ++i)
        for (int j = 0; j < I.cols; ++j)
        {
            Mat sampleMat = (Mat_<float>(1,2) << i, j);
            float response = svm.predict(sampleMat);

            if      (response == 1)    I.at<Vec3b>(j, i)  = green;
            else if (response == 2)    I.at<Vec3b>(j, i)  = blue;
        }

    //----------------------- 5. Show the training data -------------------------------------------
    int thick = -1;
    int lineType = 8;
    float px, py;
    // Class 1
    for (int i = 0; i < NTRAINING_SAMPLES; ++i)
    {
        px = trainData.at<float>(i,0);
        py = trainData.at<float>(i,1);
        circle(I, Point( (int) px,  (int) py ), 3, Scalar(0, 255, 0), thick, lineType);
    }
    // Class 2
    for (int i = NTRAINING_SAMPLES; i <2*NTRAINING_SAMPLES; ++i)
    {
        px = trainData.at<float>(i,0);
        py = trainData.at<float>(i,1);
        circle(I, Point( (int) px, (int) py ), 3, Scalar(255, 0, 0), thick, lineType);
    }

    //------------------------- 6. Show support vectors ---------------------------------------------
    thick = 2;
    lineType  = 8;
    int x     = svm.get_support_vector_count();

    for (int i = 0; i < x; ++i)
    {
        const float* v = svm.get_support_vector(i);
        circle( I,  Point( (int) v[0], (int) v[1]), 6, Scalar(128, 128, 128), thick, lineType);
    }

    imwrite("result.png", I);                      // save the Image
    imshow("SVM for Non-Linear Training Data", I); // show it to the user
    waitKey(0);
}
```

Explanation

1. **Set up the training data**

   The training data of this exercise is formed by a set of labeled 2D-points that belong to one of two different classes. To make the exercise more appealing, the training data is generated randomly using a uniform probability density functions (PDFs). We have divided the generation of the training data into two main parts. In the first part we generate data for both classes that is linearly separable.

```
// Generate random points for the class 1
Mat trainClass = trainData.rowRange(0, nLinearSamples);
// The x coordinate of the points is in [0, 0.4)
Mat c = trainClass.colRange(0, 1);
rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(0.4 * WIDTH));
// The y coordinate of the points is in [0, 1)
c = trainClass.colRange(1,2);
rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(HEIGHT));
// Generate random points for the class 2
trainClass = trainData.rowRange(2*NTRAINING_SAMPLES-nLinearSamples, 2*NTRAINING_SAMPLES);
// The x coordinate of the points is in [0.6, 1]
c = trainClass.colRange(0 , 1);
rng.fill(c, RNG::UNIFORM, Scalar(0.6*WIDTH), Scalar(WIDTH));
// The y coordinate of the points is in [0, 1)
c = trainClass.colRange(1,2);
rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(HEIGHT));
```

   In the second part we create data for both classes that is non-linearly separable, data that overlaps.

```
// Generate random points for the classes 1 and 2
trainClass = trainData.rowRange(  nLinearSamples, 2*NTRAINING_SAMPLES-nLinearSamples);
// The x coordinate of the points is in [0.4, 0.6)
c = trainClass.colRange(0,1);
rng.fill(c, RNG::UNIFORM, Scalar(0.4*WIDTH), Scalar(0.6*WIDTH));
// The y coordinate of the points is in [0, 1)
c = trainClass.colRange(1,2);
rng.fill(c, RNG::UNIFORM, Scalar(1), Scalar(HEIGHT));
```

2. **Set up SVM's parameters**

   See Also:

   In the previous tutorial *Introduction to Support Vector Machines* there is an explanation of the atributes of the class CvSVMParams that we configure here before training the SVM.

```
CvSVMParams params;
params.svm_type    = SVM::C_SVC;
params.C              = 0.1;
params.kernel_type = SVM::LINEAR;
params.term_crit   = TermCriteria(CV_TERMCRIT_ITER, (int)1e7, 1e-6);
```

   There are just two differences between the configuration we do here and the one that was done in the *previous tutorial* that we use as reference.

   - ***CvSVM::C_SVC*. We chose here a small value of this parameter in order not to punish too much the misclassification**

   > **Note:** Here there are just very few points in the overlapping region between classes, giving a smaller value to **FRAC_LINEAR_SEP** the density of points can be incremented and the impact of the parameter **CvSVM::C_SVC** explored deeply.

- *Termination Criteria of the algorithm.* The maximum number of iterations has to be increased considerably in order to solve correctly a problem with non-linearly separable training data. In particular, we have increased in five orders of magnitude this value.

3. **Train the SVM**

   We call the method CvSVM::train to build the SVM model. Watch out that the training process may take a quite long time. Have patiance when your run the program.

```
CvSVM svm;
svm.train(trainData, labels, Mat(), Mat(), params);
```

4. **Show the Decision Regions**

   The method CvSVM::predict is used to classify an input sample using a trained SVM. In this example we have used this method in order to color the space depending on the prediction done by the SVM. In other words, an image is traversed interpreting its pixels as points of the Cartesian plane. Each of the points is colored depending on the class predicted by the SVM; in dark green if it is the class with label 1 and in dark blue if it is the class with label 2.

```
Vec3b green(0,100,0), blue (100,0,0);
for (int i = 0; i < I.rows; ++i)
     for (int j = 0; j < I.cols; ++j)
     {
          Mat sampleMat = (Mat_<float>(1,2) << i, j);
          float response = svm.predict(sampleMat);
          if      (response == 1)    I.at<Vec3b>(j, i)  = green;
          else if (response == 2)    I.at<Vec3b>(j, i)  = blue;
     }
```

5. **Show the training data**

   The method circle is used to show the samples that compose the training data. The samples of the class labeled with 1 are shown in light green and in light blue the samples of the class labeled with 2.

```
int thick = -1;
int lineType = 8;
float px, py;
// Class 1
for (int i = 0; i < NTRAINING_SAMPLES; ++i)
{
     px = trainData.at<float>(i,0);
     py = trainData.at<float>(i,1);
     circle(I, Point( (int) px,  (int) py ), 3, Scalar(0, 255, 0), thick, lineType);
}
// Class 2
for (int i = NTRAINING_SAMPLES; i <2*NTRAINING_SAMPLES; ++i)
{
     px = trainData.at<float>(i,0);
     py = trainData.at<float>(i,1);
     circle(I, Point( (int) px, (int) py ), 3, Scalar(255, 0, 0), thick, lineType);
}
```

6. **Support vectors**

   We use here a couple of methods to obtain information about the support vectors. The method CvSVM::get_support_vector_count outputs the total number of support vectors used in the problem and with the method CvSVM::get_support_vector we obtain each of the support vectors using an index. We have used this methods here to find the training examples that are support vectors and highlight them.

```
thick = 2;
lineType  = 8;
int x     = svm.get_support_vector_count();
for (int i = 0; i < x; ++i)
{
     const float* v = svm.get_support_vector(i);
     circle(     I,  Point( (int) v[0], (int) v[1]), 6, Scalar(128, 128, 128), thick, lineType);
}
```

Results

- The code opens an image and shows the training examples of both classes. The points of one class are represented with light green and light blue ones are used for the other class.
- The SVM is trained and used to classify all the pixels of the image. This results in a division of the image in a blue region and a green region. The boundary between both regions is the separating hyperplane. Since the training data is non-linearly separable, it can be seen that some of the examples of both classes are misclassified; some green points lay on the blue region and some blue points lay on the green one.
- Finally the support vectors are shown using gray rings around the training examples.

You may observe a runtime instance of this on the YouTube here.

CHAPTER

TEN

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.

-

  Title: *Similarity check (PNSR and SSIM) on the GPU*
  Compatibility: > OpenCV 2.0
  Author: Bernát Gábor
  This will give a good grasp on how to approach coding on the GPU module, once you already know how to handle the other modules. As a test case it will port the similarity methods from the tutorial *Video Input with OpenCV and similarity measurement* to the GPU.

10.1 Similarity check (PNSR and SSIM) on the GPU

Goal

In the *Video Input with OpenCV and similarity measurement* tutorial I already presented the PSNR and SSIM methods for checking the similarity between the two images. And as you could see there performing these takes quite some time, especially in the case of the SSIM. However, if the performance numbers of an OpenCV implementation for the CPU do not satisfy you and you happen to have an NVidia CUDA GPU device in your system all is not lost. You may try to port or write your algorithm for the video card.

This tutorial will give a good grasp on how to approach coding by using the GPU module of OpenCV. As a prerequisite you should already know how to handle the core, highgui and imgproc modules. So, our goals are:

- What's different compared to the CPU?
- Create the GPU code for the PSNR and SSIM
- Optimize the code for maximal performance

The source code

You may also find the source code and these video file in the `samples/cpp/tutorial_code/gpu/gpu-basics-similarity/gpu-basi` folder of the OpenCV source library or `download it from here`. The full source code is quite long (due to the controlling of the application via the command line arguments and performance measurement). Therefore, to avoid cluttering up these sections with those you'll find here only the functions itself.

The PSNR returns a float number, that if the two inputs are similar between 30 and 50 (higher is better).

```
double getPSNR(const Mat& I1, const Mat& I2)
{
    Mat s1;
    absdiff(I1, I2, s1);       // |I1 - I2|
    s1.convertTo(s1, CV_32F);  // cannot make a square on 8 bits
    s1 = s1.mul(s1);           // |I1 - I2|^2

    Scalar s = sum(s1);         // sum elements per channel

    double sse = s.val[0] + s.val[1] + s.val[2]; // sum channels

    if( sse <= 1e-10) // for small values return zero
        return 0;
    else
    {
        double  mse =sse /(double)(I1.channels() * I1.total());
        double psnr = 10.0*log10((255*255)/mse);
        return psnr;
    }
}



double getPSNR_GPU_optimized(const Mat& I1, const Mat& I2, BufferPSNR& b)
{
    b.gI1.upload(I1);
    b.gI2.upload(I2);

    b.gI1.convertTo(b.t1, CV_32F);
```

```
    b.gI2.convertTo(b.t2, CV_32F);

    gpu::absdiff(b.t1.reshape(1), b.t2.reshape(1), b.gs);
    gpu::multiply(b.gs, b.gs, b.gs);

    double sse = gpu::sum(b.gs, b.buf)[0];

    if( sse <= 1e-10) // for small values return zero
        return 0;
    else
    {
        double mse = sse /(double)(I1.channels() * I1.total());
        double psnr = 10.0*log10((255*255)/mse);
        return psnr;
    }
}

struct BufferPSNR                                     // Optimized GPU versions
{   // Data allocations are very expensive on GPU. Use a buffer to solve: allocate once reuse later.
    gpu::GpuMat gI1, gI2, gs, t1,t2;

    gpu::GpuMat buf;
};

double getPSNR_GPU(const Mat& I1, const Mat& I2)
{
    gpu::GpuMat gI1, gI2, gs, t1,t2;

    gI1.upload(I1);
    gI2.upload(I2);

    gI1.convertTo(t1, CV_32F);
    gI2.convertTo(t2, CV_32F);

    gpu::absdiff(t1.reshape(1), t2.reshape(1), gs);
    gpu::multiply(gs, gs, gs);

    Scalar s = gpu::sum(gs);
    double sse = s.val[0] + s.val[1] + s.val[2];

    if( sse <= 1e-10) // for small values return zero
        return 0;
    else
    {
        double  mse =sse /(double)(gI1.channels() * I1.total());
        double psnr = 10.0*log10((255*255)/mse);
        return psnr;
    }
}
```

The SSIM returns the MSSIM of the images. This is too a float number between zero and one (higher is better), however we have one for each channel. Therefore, we return a *Scalar* OpenCV data structure:

```
}

Scalar getMSSIM( const Mat& i1, const Mat& i2)
{
    const double C1 = 6.5025, C2 = 58.5225;
```

```
    /***************************** INITS **********************************/
    int d     = CV_32F;

    Mat I1, I2;
    i1.convertTo(I1, d);           // cannot calculate on one byte large values
    i2.convertTo(I2, d);

    Mat I2_2   = I2.mul(I2);        // I2^2
    Mat I1_2   = I1.mul(I1);        // I1^2
    Mat I1_I2  = I1.mul(I2);        // I1 * I2

    /*************************** END INITS **********************************/

    Mat mu1, mu2;   // PRELIMINARY COMPUTING
    GaussianBlur(I1, mu1, Size(11, 11), 1.5);
    GaussianBlur(I2, mu2, Size(11, 11), 1.5);

    Mat mu1_2   =   mu1.mul(mu1);
    Mat mu2_2   =   mu2.mul(mu2);
    Mat mu1_mu2 =   mu1.mul(mu2);

    Mat sigma1_2, sigma2_2, sigma12;

    GaussianBlur(I1_2, sigma1_2, Size(11, 11), 1.5);
    sigma1_2 -= mu1_2;

    GaussianBlur(I2_2, sigma2_2, Size(11, 11), 1.5);
    sigma2_2 -= mu2_2;

    GaussianBlur(I1_I2, sigma12, Size(11, 11), 1.5);
    sigma12 -= mu1_mu2;

    ////////////////////////////////// FORMULA ////////////////////////////////
    Mat t1, t2, t3;

    t1 = 2 * mu1_mu2 + C1;
    t2 = 2 * sigma12 + C2;
    t3 = t1.mul(t2);              // t3 = ((2*mu1_mu2 + C1).*(2*sigma12 + C2))

    t1 = mu1_2 + mu2_2 + C1;
    t2 = sigma1_2 + sigma2_2 + C2;
    t1 = t1.mul(t2);               // t1 =((mu1_2 + mu2_2 + C1).*(sigma1_2 + sigma2_2 + C2))

    Mat ssim_map;
    divide(t3, t1, ssim_map);      // ssim_map =  t3./t1;

    Scalar mssim = mean( ssim_map ); // mssim = average of ssim map
    return mssim;
}

Scalar getMSSIM_GPU( const Mat& i1, const Mat& i2)
{
    const float C1 = 6.5025f, C2 = 58.5225f;
    /***************************** INITS **********************************/
    gpu::GpuMat gI1, gI2, gs1, tmp1,tmp2;

    gI1.upload(i1);
    gI2.upload(i2);
```

```
    gI1.convertTo(tmp1, CV_MAKE_TYPE(CV_32F, gI1.channels()));
    gI2.convertTo(tmp2, CV_MAKE_TYPE(CV_32F, gI2.channels()));

    vector<gpu::GpuMat> vI1, vI2;
    gpu::split(tmp1, vI1);
    gpu::split(tmp2, vI2);
    Scalar mssim;

    for( int i = 0; i < gI1.channels(); ++i )
    {
        gpu::GpuMat I2_2, I1_2, I1_I2;

        gpu::multiply(vI2[i], vI2[i], I2_2);        // I2^2
        gpu::multiply(vI1[i], vI1[i], I1_2);        // I1^2
        gpu::multiply(vI1[i], vI2[i], I1_I2);       // I1 * I2

        /*************************** END INITS **********************************/
        gpu::GpuMat mu1, mu2;   // PRELIMINARY COMPUTING
        gpu::GaussianBlur(vI1[i], mu1, Size(11, 11), 1.5);
        gpu::GaussianBlur(vI2[i], mu2, Size(11, 11), 1.5);

        gpu::GpuMat mu1_2, mu2_2, mu1_mu2;
        gpu::multiply(mu1, mu1, mu1_2);
        gpu::multiply(mu2, mu2, mu2_2);
        gpu::multiply(mu1, mu2, mu1_mu2);

        gpu::GpuMat sigma1_2, sigma2_2, sigma12;

        gpu::GaussianBlur(I1_2, sigma1_2, Size(11, 11), 1.5);
        gpu::subtract(sigma1_2, mu1_2, sigma1_2); // sigma1_2 -= mu1_2;

        gpu::GaussianBlur(I2_2, sigma2_2, Size(11, 11), 1.5);
        gpu::subtract(sigma2_2, mu2_2, sigma2_2); // sigma2_2 -= mu2_2;

        gpu::GaussianBlur(I1_I2, sigma12, Size(11, 11), 1.5);
        gpu::subtract(sigma12, mu1_mu2, sigma12); // sigma12 -= mu1_mu2;

        ////////////////////////////////// FORMULA ////////////////////////////////
        gpu::GpuMat t1, t2, t3;

        mu1_mu2.convertTo(t1, -1, 2, C1); // t1 = 2 * mu1_mu2 + C1;
        sigma12.convertTo(t2, -1, 2, C2); // t2 = 2 * sigma12 + C2;
        gpu::multiply(t1, t2, t3);        // t3 = ((2*mu1_mu2 + C1).*(2*sigma12 + C2))

        gpu::addWeighted(mu1_2, 1.0, mu2_2, 1.0, C1, t1);       // t1 = mu1_2 + mu2_2 + C1;
        gpu::addWeighted(sigma1_2, 1.0, sigma2_2, 1.0, C2, t2); // t2 = sigma1_2 + sigma2_2 + C2;
        gpu::multiply(t1, t2, t1);                              // t1 =((mu1_2 + mu2_2 + C1).*(sigma1_2 + sigma2_2 + C

        gpu::GpuMat ssim_map;
        gpu::divide(t3, t1, ssim_map);      // ssim_map =  t3./t1;

        Scalar s = gpu::sum(ssim_map);
        mssim.val[i] = s.val[0] / (ssim_map.rows * ssim_map.cols);

    }
    return mssim;
}
```

```
struct BufferMSSIM                                     // Optimized GPU versions
{   // Data allocations are very expensive on GPU. Use a buffer to solve: allocate once reuse later.
    gpu::GpuMat gI1, gI2, gs, t1,t2;

    gpu::GpuMat I1_2, I2_2, I1_I2;
    vector<gpu::GpuMat> vI1, vI2;

    gpu::GpuMat mu1, mu2;
    gpu::GpuMat mu1_2, mu2_2, mu1_mu2;

    gpu::GpuMat sigma1_2, sigma2_2, sigma12;
    gpu::GpuMat t3;

    gpu::GpuMat ssim_map;

    gpu::GpuMat buf;
};
Scalar getMSSIM_GPU_optimized( const Mat& i1, const Mat& i2, BufferMSSIM& b)
{
    const float C1 = 6.5025f, C2 = 58.5225f;
    /***************************** INITS *********************************/

    b.gI1.upload(i1);
    b.gI2.upload(i2);

    gpu::Stream stream;

    stream.enqueueConvert(b.gI1, b.t1, CV_32F);
    stream.enqueueConvert(b.gI2, b.t2, CV_32F);

    gpu::split(b.t1, b.vI1, stream);
    gpu::split(b.t2, b.vI2, stream);
    Scalar mssim;

    gpu::GpuMat buf;

    for( int i = 0; i < b.gI1.channels(); ++i )
    {
        gpu::multiply(b.vI2[i], b.vI2[i], b.I2_2, stream);        // I2^2
        gpu::multiply(b.vI1[i], b.vI1[i], b.I1_2, stream);        // I1^2
        gpu::multiply(b.vI1[i], b.vI2[i], b.I1_I2, stream);       // I1 * I2

        gpu::GaussianBlur(b.vI1[i], b.mu1, Size(11, 11), buf, 1.5, 0, BORDER_DEFAULT, -1, stream);
        gpu::GaussianBlur(b.vI2[i], b.mu2, Size(11, 11), buf, 1.5, 0, BORDER_DEFAULT, -1, stream);

        gpu::multiply(b.mu1, b.mu1, b.mu1_2, stream);
        gpu::multiply(b.mu2, b.mu2, b.mu2_2, stream);
        gpu::multiply(b.mu1, b.mu2, b.mu1_mu2, stream);

        gpu::GaussianBlur(b.I1_2, b.sigma1_2, Size(11, 11), buf, 1.5, 0, BORDER_DEFAULT, -1, stream);
        gpu::subtract(b.sigma1_2, b.mu1_2, b.sigma1_2, gpu::GpuMat(), -1, stream);
        //b.sigma1_2 -= b.mu1_2;  - This would result in an extra data transfer operation

        gpu::GaussianBlur(b.I2_2, b.sigma2_2, Size(11, 11), buf, 1.5, 0, BORDER_DEFAULT, -1, stream);
        gpu::subtract(b.sigma2_2, b.mu2_2, b.sigma2_2, gpu::GpuMat(), -1, stream);
        //b.sigma2_2 -= b.mu2_2;

        gpu::GaussianBlur(b.I1_I2, b.sigma12, Size(11, 11), buf, 1.5, 0, BORDER_DEFAULT, -1, stream);
```

```
        gpu::subtract(b.sigma12, b.mu1_mu2, b.sigma12, gpu::GpuMat(), -1, stream);
        //b.sigma12 -= b.mu1_mu2;

        //here too it would be an extra data transfer due to call of operator*(Scalar, Mat)
        gpu::multiply(b.mu1_mu2, 2, b.t1, 1, -1, stream); //b.t1 = 2 * b.mu1_mu2 + C1;
        gpu::add(b.t1, C1, b.t1, gpu::GpuMat(), -1, stream);
        gpu::multiply(b.sigma12, 2, b.t2, 1, -1, stream); //b.t2 = 2 * b.sigma12 + C2;
        gpu::add(b.t2, C2, b.t2, gpu::GpuMat(), -12, stream);

        gpu::multiply(b.t1, b.t2, b.t3, 1, -1, stream);     // t3 = ((2*mu1_mu2 + C1).*(2*sigma12 + C2))

        gpu::add(b.mu1_2, b.mu2_2, b.t1, gpu::GpuMat(), -1, stream);
        gpu::add(b.t1, C1, b.t1, gpu::GpuMat(), -1, stream);

        gpu::add(b.sigma1_2, b.sigma2_2, b.t2, gpu::GpuMat(), -1, stream);
        gpu::add(b.t2, C2, b.t2, gpu::GpuMat(), -1, stream);


        gpu::multiply(b.t1, b.t2, b.t1, 1, -1, stream);     // t1 =((mu1_2 + mu2_2 + C1).*(sigma1_2 + sigma2_2 + C2))
        gpu::divide(b.t3, b.t1, b.ssim_map, 1, -1, stream);      // ssim_map =  t3./t1;

        stream.waitForCompletion();

        Scalar s = gpu::sum(b.ssim_map, b.buf);
        mssim.val[i] = s.val[0] / (b.ssim_map.rows * b.ssim_map.cols);

    }
    return mssim;
}
```

How to do it? - The GPU

Now as you can see we have three types of functions for each operation. One for the CPU and two for the GPU. The reason I made two for the GPU is too illustrate that often simple porting your CPU to GPU will actually make it slower. If you want some performance gain you will need to remember a few rules, whose I'm going to detail later on.

The development of the GPU module was made so that it resembles as much as possible its CPU counterpart. This is to make porting easy. The first thing you need to do before writing any code is to link the GPU module to your project, and include the header file for the module. All the functions and data structures of the GPU are in a *gpu* sub namespace of the *cv* namespace. You may add this to the default one via the *use namespace* keyword, or mark it everywhere explicitly via the cv:: to avoid confusion. I'll do the later.

```
#include <opencv2/gpu/gpu.hpp>        // GPU structures and methods
```

GPU stands for **g**raphics **p**rocessing **u**nit. It was originally build to render graphical scenes. These scenes somehow build on a lot of data. Nevertheless, these aren't all dependent one from another in a sequential way and as it is possible a parallel processing of them. Due to this a GPU will contain multiple smaller processing units. These aren't the state of the art processors and on a one on one test with a CPU it will fall behind. However, its strength lies in its numbers. In the last years there has been an increasing trend to harvest these massive parallel powers of the GPU in non-graphical scene rendering too. This gave birth to the general-purpose computation on graphics processing units (GPGPU).

The GPU has its own memory. When you read data from the hard drive with OpenCV into a *Mat* object that takes place in your systems memory. The CPU works somehow directly on this (via its cache), however the GPU cannot. He has too transferred the information he will use for calculations from the system memory to its own. This is done via an upload process and takes time. In the end the result will have to be downloaded back to your system memory

for your CPU to see it and use it. Porting small functions to GPU is not recommended as the upload/download time will be larger than the amount you gain by a parallel execution.

Mat objects are stored **only** in the system memory (or the CPU cache). For getting an OpenCV matrix to the GPU you'll need to use its GPU counterpart GpuMat. It works similar to the *Mat* with a 2D only limitation and no reference returning for its functions (cannot mix GPU references with CPU ones). To upload a *Mat* object to the *GPU* you need to call the *upload* function after creating an instance of the class. To download you may use simple assignment to a *Mat* object or use the *download* function.

```
Mat I1;         // Main memory item - read image into with imread for example
gpu::GpuMat gI; // GPU matrix - for now empty
gI1.upload(I1); // Upload a data from the system memory to the GPU memory

I1 = gI1;       // Download, gI1.download(I1) will work too
```

Once you have your data up in the GPU memory you may call GPU enabled functions of OpenCV. Most of the functions keep the same name just as on the CPU, with the difference that they only accept *GpuMat* inputs. A full list of these you will find in the documentation: online here or the OpenCV reference manual that comes with the source code.

Another thing to keep in mind is that not for all channel numbers you can make efficient algorithms on the GPU. Generally, I found that the input images for the GPU images need to be either one or four channel ones and one of the char or float type for the item sizes. No double support on the GPU, sorry. Passing other types of objects for some functions will result in an exception thrown, and an error message on the error output. The documentation details in most of the places the types accepted for the inputs. If you have three channel images as an input you can do two things: either adds a new channel (and use char elements) or split up the image and call the function for each image. The first one isn't really recommended as you waste memory.

For some functions, where the position of the elements (neighbor items) doesn't matter quick solution is to just reshape it into a single channel image. This is the case for the PSNR implementation where for the *absdiff* method the value of the neighbors is not important. However, for the *GaussianBlur* this isn't an option and such need to use the split method for the SSIM. With this knowledge you can already make a GPU viable code (like mine GPU one) and run it. You'll be surprised to see that it might turn out slower than your CPU implementation.

Optimization

The reason for this is that you're throwing out on the window the price for memory allocation and data transfer. And on the GPU this is damn high. Another possibility for optimization is to introduce asynchronous OpenCV GPU calls too with the help of the gpu::Stream.

1. Memory allocation on the GPU is considerable. Therefore, if it's possible allocate new memory as few times as possible. If you create a function what you intend to call multiple times it is a good idea to allocate any local parameters for the function only once, during the first call. To do this you create a data structure containing all the local variables you will use. For instance in case of the PSNR these are:

   ```
   struct BufferPSNR                                     // Optimized GPU versions
     {   // Data allocations are very expensive on GPU. Use a buffer to solve: allocate once reuse later.
     gpu::GpuMat gI1, gI2, gs, t1,t2;

     gpu::GpuMat buf;
   };
   ```

   Then create an instance of this in the main program:

   ```
   BufferPSNR bufferPSNR;
   ```

   And finally pass this to the function each time you call it:

```
double getPSNR_GPU_optimized(const Mat& I1, const Mat& I2, BufferPSNR& b)
```

Now you access these local parameters as: $b.gI1$, $b.buf$ and so on. The GpuMat will only reallocate itself on a new call if the new matrix size is different from the previous one.

2. Avoid unnecessary function data transfers. Any small data transfer will be significant one once you go to the GPU. Therefore, if possible make all calculations in-place (in other words do not create new memory objects - for reasons explained at the previous point). For example, although expressing arithmetical operations may be easier to express in one line formulas, it will be slower. In case of the SSIM at one point I need to calculate:

```
b.t1 = 2 * b.mu1_mu2 + C1;
```

Although the upper call will succeed observe that there is a hidden data transfer present. Before it makes the addition it needs to store somewhere the multiplication. Therefore, it will create a local matrix in the background, add to that the $C1$ value and finally assign that to $t1$. To avoid this we use the gpu functions, instead of the arithmetic operators:

```
gpu::multiply(b.mu1_mu2, 2, b.t1); //b.t1 = 2 * b.mu1_mu2 + C1;
gpu::add(b.t1, C1, b.t1);
```

3. Use asynchronous calls (the gpu::Stream). By default whenever you call a gpu function it will wait for the call to finish and return with the result afterwards. However, it is possible to make asynchronous calls, meaning it will call for the operation execution, make the costly data allocations for the algorithm and return back right away. Now you can call another function if you wish to do so. For the MSSIM this is a small optimization point. In our default implementation we split up the image into channels and call then for each channel the gpu functions. A small degree of parallelization is possible with the stream. By using a stream we can make the data allocation, upload operations while the GPU is already executing a given method. For example we need to upload two images. We queue these one after another and call already the function that processes it. The functions will wait for the upload to finish, however while that happens makes the output buffer allocations for the function to be executed next.

```
gpu::Stream stream;

stream.enqueueConvert(b.gI1, b.t1, CV_32F);    // Upload

gpu::split(b.t1, b.vI1, stream);              // Methods (pass the stream as final parameter).
gpu::multiply(b.vI1[i], b.vI1[i], b.I1_2, stream);        // I1^2
```

Result and conclusion

On an Intel P8700 laptop CPU paired with a low end NVidia GT220M here are the performance numbers:

```
Time of PSNR CPU (averaged for 10 runs): 41.4122 milliseconds. With result of: 19.2506
Time of PSNR GPU (averaged for 10 runs): 158.977 milliseconds. With result of: 19.2506
Initial call GPU optimized:              31.3418 milliseconds. With result of: 19.2506
Time of PSNR GPU OPTIMIZED ( / 10 runs): 24.8171 milliseconds. With result of: 19.2506

Time of MSSIM CPU (averaged for 10 runs): 484.343 milliseconds. With result of B0.890964 G0.903845 R0.936934
Time of MSSIM GPU (averaged for 10 runs): 745.105 milliseconds. With result of B0.89922 G0.909051 R0.968223
Time of MSSIM GPU Initial Call            357.746 milliseconds. With result of B0.890964 G0.903845 R0.936934
Time of MSSIM GPU OPTIMIZED ( / 10 runs): 203.091 milliseconds. With result of B0.890964 G0.903845 R0.936934
```

In both cases we managed a performance increase of almost 100% compared to the CPU implementation. It may be just the improvement needed for your application to work. You may observe a runtime instance of this on the YouTube here.

CHAPTER

ELEVEN

CONTRIB MODULE. THE ADDITIONAL CONTRIBUTIONS MADE AVAILABLE !

Here you will learn how to use additional modules of OpenCV defined in the "contrib" module.

-

  Title: *Discovering the human retina and its use for image processing*
  Compatibility: > OpenCV 2.4
  Author: Alexandre Benoit
  You will learn how to process images and video streams with a model of retina filter for details enhancement, spatio-temporal noise removal, luminance correction and spatio-temporal events detection.

11.1 Discovering the human retina and its use for image processing

Goal

I present here a model of human retina that shows some interesting properties for image preprocessing and enhancement. In this tutorial you will learn how to:

- discover the main two channels outing from your retina
- see the basics to use the retina model
- discover some parameters tweaks

General overview

The proposed model originates from Jeanny Herault's research at Gipsa. It is involved in image processing applications with Listic (code maintainer) lab. This is not a complete model but it already present interesting properties that can be involved for enhanced image processing experience. The model allows the following human retina properties to be used :

- spectral whitening that has 3 important effects: high spatio-temporal frequency signals canceling (noise), mid-frequencies details enhancement and low frequencies luminance energy reduction. This *all in one* property directly allows visual signals cleaning of classical undesired distortions introduced by image sensors and input luminance range.
- local logarithmic luminance compression allows details to be enhanced even in low light conditions.
- decorrelation of the details information (Parvocellular output channel) and transient information (events, motion made available at the Magnocellular output channel).

The first two points are illustrated below :

In the figure below, the OpenEXR image sample *CrissyField.exr*, a High Dynamic Range image is shown. In order to make it visible on this web-page, the original input image is linearly rescaled to the classical image luminance range [0-255] and is converted to 8bit/channel format. Such strong conversion hides many details because of too strong local contrasts. Furthermore, noise energy is also strong and pollutes visual information.

In the following image, as your retina does, local luminance adaptation, spatial noise removal and spectral whitening work together and transmit accurate information on lower range 8bit data channels. On this picture, noise in significantly removed, local details hidden by strong luminance contrasts are enhanced. Output image keeps its naturalness and visual content is enhanced.

Note : image sample can be downloaded from the OpenEXR website. Regarding this demonstration, before retina processing, input image has been linearly rescaled within 0-255 keeping its channels float format. 5% of its histogram ends has been cut (mostly removes wrong HDR pixels). Check out the sample *opencv/samples/cpp/OpenEXRimages_HighDynamicRange_Retina_toneMapping.cpp* for similar processing. The following demonstration will only consider classical 8bit/channel images.

The retina model output channels

The retina model presents two outputs that benefit from the above cited behaviors.

- The first one is called the Parvocellular channel. It is mainly active in the foveal retina area (high resolution central vision with color sensitive photo-receptors), its aim is to provide accurate color vision for visual details remaining static on the retina. On the other hand objects moving on the retina projection are blurred.
- The second well known channel is the Magnocellular channel. It is mainly active in the retina peripheral vision and send signals related to change events (motion, transient events, etc.). These outing signals also help visual system to focus/center retina on 'transient'/moving areas for more detailed analysis thus improving visual scene context and object classification.

NOTE : regarding the proposed model, contrary to the real retina, we apply these two channels on the entire input images using the same resolution. This allows enhanced visual details and motion information to be extracted on all the considered images... but remember, that these two channels are complementary. For example, if Magnocellular channel gives strong energy in an area, then, the Parvocellular channel is certainly blurred there since there is a transient event.

As an illustration, we apply in the following the retina model on a webcam video stream of a dark visual scene. In this visual scene, captured in an amphitheater of the university, some students are moving while talking to the teacher.

In this video sequence, because of the dark ambiance, signal to noise ratio is low and color artifacts are present on visual features edges because of the low quality image capture tool-chain.

Below is shown the retina foveal vision applied on the entire image. In the used retina configuration, global luminance is preserved and local contrasts are enhanced. Also, signal to noise ratio is improved : since high frequency spatio-temporal noise is reduced, enhanced details are not corrupted by any enhanced noise.

Below is the output of the Magnocellular output of the retina model. Its signals are strong where transient events occur. Here, a student is moving at the bottom of the image thus generating high energy. The remaining of the image is static however, it is corrupted by a strong noise. Here, the retina filters out most of the noise thus generating low false motion area 'alarms'. This channel can be used as a transient/moving areas detector : it would provide relevant information for a low cost segmentation tool that would highlight areas in which an event is occurring.

Retina use case

This model can be used basically for spatio-temporal video effects but also in the aim of :

- performing texture analysis with enhanced signal to noise ratio and enhanced details robust against input images luminance ranges (check out the Parvocellular retina channel output)
- performing motion analysis also taking benefit of the previously cited properties.

For more information, refer to the following papers :

- Benoit A., Caplier A., Durette B., Herault, J., "Using Human Visual System Modeling For Bio-Inspired Low Level Image Processing", Elsevier, Computer Vision and Image Understanding 114 (2010), pp. 758-773. DOI <http://dx.doi.org/10.1016/j.cviu.2010.01.011>
- Please have a look at the reference work of Jeanny Herault that you can read in his book :

Vision: Images, Signals and Neural Networks: Models of Neural Processing in Visual Perception (Progress in Neural Processing),By: Jeanny Herault, ISBN: 9814273686. WAPI (Tower ID): 113266891.

This retina filter code includes the research contributions of phd/research collegues from which code has been redrawn by the author :

- take a look at the *retinacolor.hpp* module to discover Brice Chaix de Lavarene phD color mosaicing/demosaicing and his reference paper: B. Chaix de Lavarene, D. Alleysson, B. Durette, J. Herault (2007). "Efficient demosaicing through recursive filtering", IEEE International Conference on Image Processing ICIP 2007

- take a look at *imagelogpolprojection.hpp* to discover retina spatial log sampling which originates from Barthelemy Durette phd with Jeanny Herault. A Retina / V1 cortex projection is also proposed and originates from Jeanny's discussions. ====> more information in the above cited Jeanny Heraults's book.

Code tutorial

Please refer to the original tutorial source code in file *opencv_folder/samples/cpp/tutorial_code/contrib/retina_tutorial.cpp*.

To compile it, assuming OpenCV is correctly installed, use the following command. It requires the opencv_core *(cv::Mat and friends objects management)*, opencv_highgui *(display and image/video read)* and opencv_contrib *(Retina description)* libraries to compile.

```
// compile
gcc retina_tutorial.cpp -o Retina_tuto -lopencv_core -lopencv_highgui -lopencv_contrib

// Run commands : add 'log' as a last parameter to apply a spatial log sampling (simulates retina sampling)
// run on webcam
./Retina_tuto -video
// run on video file
./Retina_tuto -video myVideo.avi
// run on an image
./Retina_tuto -image myPicture.jpg
// run on an image with log sampling
./Retina_tuto -image myPicture.jpg log
```

Here is a code explanation :

Retina definition is present in the contrib package and a simple include allows to use it

```
#include "opencv2/opencv.hpp"
```

Provide user some hints to run the program with a help function

```
// the help procedure
static void help(std::string errorMessage)
{
 std::cout<<"Program init error : "<<errorMessage<<std::endl;
 std::cout<<"\nProgram call procedure : retinaDemo [processing mode] [Optional : media target] [Optional LAST paramete
 std::cout<<"\t[processing mode] :"<<std::endl;
 std::cout<<"\t -image : for still image processing"<<std::endl;
 std::cout<<"\t -video : for video stream processing"<<std::endl;
 std::cout<<"\t[Optional : media target] :"<<std::endl;
 std::cout<<"\t if processing an image or video file, then, specify the path and filename of the target to process"<<s
 std::cout<<"\t leave empty if processing video stream coming from a connected video device"<<std::endl;
 std::cout<<"\t[Optional : activate retina log sampling] : an optional last parameter can be specified for retina spat
 std::cout<<"\t set \"log\" without quotes to activate this sampling, output frame size will be divided by 4"<<std::en
 std::cout<<"\nExamples:"<<std::endl;
 std::cout<<"\t-Image processing : ./retinaDemo -image lena.jpg"<<std::endl;
 std::cout<<"\t-Image processing with log sampling : ./retinaDemo -image lena.jpg log"<<std::endl;
 std::cout<<"\t-Video processing : ./retinaDemo -video myMovie.mp4"<<std::endl;
 std::cout<<"\t-Live video processing : ./retinaDemo -video"<<std::endl;
 std::cout<<"\nPlease start again with new parameters"<<std::endl;
 std::cout<<"****************************************************"<<std::endl;
 std::cout<<" NOTE : this program generates the default retina parameters file 'RetinaDefaultParameters.xml'"<<std::en
 std::cout<<" => you can use this to fine tune parameters and load them if you save to file 'RetinaSpecificParameters.
}
```

Then, start the main program and first declare a *cv::Mat* matrix in which input images will be loaded. Also allocate a *cv::VideoCapture* object ready to load video streams (if necessary)

```
int main(int argc, char* argv[]) {
  // declare the retina input buffer... that will be fed differently in regard of the input media
  cv::Mat inputFrame;
  cv::VideoCapture videoCapture; // in case a video media is used, its manager is declared here
```

In the main program, before processing, first check input command parameters. Here it loads a first input image coming from a single loaded image (if user chose command *-image*) or from a video stream (if user chose command *-video*). Also, if the user added *log* command at the end of its program call, the spatial logarithmic image sampling performed by the retina is taken into account by the Boolean flag *useLogSampling*.

```
// welcome message
  std::cout<<"*****************************************************"<<std::endl;
  std::cout<<"* Retina demonstration : demonstrates the use of is a wrapper class of the Gipsa/Listic Labs retina mode
  std::cout<<"* This demo will try to load the file 'RetinaSpecificParameters.xml' (if exists).\nTo create it, copy th
  // basic input arguments checking
  if (argc<2)
  {
      help("bad number of parameter");
      return -1;
  }

  bool useLogSampling = !strcmp(argv[argc-1], "log"); // check if user wants retina log sampling processing

  std::string inputMediaType=argv[1];

  ////////////////////////////////////////////////////////////////////////////
  // checking input media type (still image, video file, live video acquisition)
  if (!strcmp(inputMediaType.c_str(), "-image") && argc >= 3)
  {
      std::cout<<"RetinaDemo: processing image "<<argv[2]<<std::endl;
      // image processing case
      inputFrame = cv::imread(std::string(argv[2]), 1); // load image in RGB mode
  }else
      if (!strcmp(inputMediaType.c_str(), "-video"))
      {
          if (argc == 2 || (argc == 3 && useLogSampling)) // attempt to grab images from a video capture device
          {
              videoCapture.open(0);
          }else// attempt to grab images from a video filestream
          {
              std::cout<<"RetinaDemo: processing video stream "<<argv[2]<<std::endl;
              videoCapture.open(argv[2]);
          }

          // grab a first frame to check if everything is ok
          videoCapture>>inputFrame;
      }else
      {
          // bad command parameter
          help("bad command parameter");
          return -1;
      }
```

Once all input parameters are processed, a first image should have been loaded, if not, display error and stop program :

```
if (inputFrame.empty())
{
```

```
    help("Input media could not be loaded, aborting");
    return -1;
}
```

Now, everything is ready to run the retina model. I propose here to allocate a retina instance and to manage the eventual log sampling option. The Retina constructor expects at least a cv::Size object that shows the input data size that will have to be managed. One can activate other options such as color and its related color multiplexing strategy (here Bayer multiplexing is chosen using enum cv::RETINA_COLOR_BAYER). If using log sampling, the image reduction factor (smaller output images) and log sampling strengh can be adjusted.

```
// pointer to a retina object
cv::Ptr<cv::Retina> myRetina;

// if the last parameter is 'log', then activate log sampling (favour foveal vision and subsamples peripheral vision)
if (useLogSampling)
{
    myRetina = new cv::Retina(inputFrame.size(), true, cv::RETINA_COLOR_BAYER, true, 2.0, 10.0);
}
else// -> else allocate "classical" retina :
    myRetina = new cv::Retina(inputFrame.size());
```

Once done, the proposed code writes a default xml file that contains the default parameters of the retina. This is useful to make your own config using this template. Here generated template xml file is called *RetinaDefaultParameters.xml*.

```
// save default retina parameters file in order to let you see this and maybe modify it and reload using method "setup
myRetina->write("RetinaDefaultParameters.xml");
```

In the following line, the retina attempts to load another xml file called *RetinaSpecificParameters.xml*. If you created it and introduced your own setup, it will be loaded, in the other case, default retina parameters are used.

```
// load parameters if file exists
myRetina->setup("RetinaSpecificParameters.xml");
```

It is not required here but just to show it is possible, you can reset the retina buffers to zero to force it to forget past events.

```
// reset all retina buffers (imagine you close your eyes for a long time)
myRetina->clearBuffers();
```

Now, it is time to run the retina ! First create some output buffers ready to receive the two retina channels outputs

```
// declare retina output buffers
cv::Mat retinaOutput_parvo;
cv::Mat retinaOutput_magno;
```

Then, run retina in a loop, load new frames from video sequence if necessary and get retina outputs back to dedicated buffers.

```
// processing loop with no stop condition
while(true)
{
    // if using video stream, then, grabbing a new frame, else, input remains the same
    if (videoCapture.isOpened())
        videoCapture>>inputFrame;

    // run retina filter on the loaded input frame
    myRetina->run(inputFrame);
    // Retrieve and display retina output
    myRetina->getParvo(retinaOutput_parvo);
```

```
    myRetina->getMagno(retinaOutput_magno);
    cv::imshow("retina input", inputFrame);
    cv::imshow("Retina Parvo", retinaOutput_parvo);
    cv::imshow("Retina Magno", retinaOutput_magno);
    cv::waitKey(10);
}
```

That's done ! But if you want to secure the system, take care and manage Exceptions. The retina can throw some when it sees irrelevant data (no input frame, wrong setup, etc.). Then, i recommend to surround all the retina code by a try/catch system like this :

```
try{
     // pointer to a retina object
     cv::Ptr<cv::Retina> myRetina;
     [---]
     // processing loop with no stop condition
     while(true)
     {
         [---]
     }

}catch(cv::Exception e)
{
    std::cerr<<"Error using Retina : "<<e.what()<<std::endl;
}
```

Retina parameters, what to do ?

First, it is recommended to read the reference paper :

- Benoit A., Caplier A., Durette B., Herault, J., *"Using Human Visual System Modeling For Bio-Inspired Low Level Image Processing"*, Elsevier, Computer Vision and Image Understanding 114 (2010), pp. 758-773. DOI <http://dx.doi.org/10.1016/j.cviu.2010.01.011>

Once done open the configuration file *RetinaDefaultParameters.xml* generated by the demo and let's have a look at it.

```
<?xml version="1.0"?>
<opencv_storage>
<OPLandIPLparvo>
  <colorMode>1</colorMode>
  <normaliseOutput>1</normaliseOutput>
  <photoreceptorsLocalAdaptationSensitivity>7.0e-01</photoreceptorsLocalAdaptationSensitivity>
  <photoreceptorsTemporalConstant>5.0e-01</photoreceptorsTemporalConstant>
  <photoreceptorsSpatialConstant>5.3e-01</photoreceptorsSpatialConstant>
  <horizontalCellsGain>0.</horizontalCellsGain>
  <hcellsTemporalConstant>1.</hcellsTemporalConstant>
  <hcellsSpatialConstant>7.</hcellsSpatialConstant>
  <ganglionCellsSensitivity>7.0e-01</ganglionCellsSensitivity></OPLandIPLparvo>
<IPLmagno>
  <normaliseOutput>1</normaliseOutput>
  <parasolCells_beta>0.</parasolCells_beta>
  <parasolCells_tau>0.</parasolCells_tau>
  <parasolCells_k>7.</parasolCells_k>
  <amacrinCellsTemporalCutFrequency>1.2e+00</amacrinCellsTemporalCutFrequency>
  <V0CompressionParameter>9.5e-01</V0CompressionParameter>
  <localAdaptintegration_tau>0.</localAdaptintegration_tau>
  <localAdaptintegration_k>7.</localAdaptintegration_k></IPLmagno>
</opencv_storage>
```

Here are some hints but actually, the best parameter setup depends more on what you want to do with the retina rather than the images input that you give to retina. Apart from the more specific case of High Dynamic Range images (HDR) that require more specific setup for specific luminance compression objective, the retina behaviors should be rather stable from content to content. Note that OpenCV is able to manage such HDR format thanks to the OpenEXR images compatibility.

Then, if the application target requires details enhancement prior to specific image processing, you need to know if mean luminance information is required or not. If not, the the retina can cancel or significantly reduce its energy thus giving more visibility to higher spatial frequency details.

Basic parameters

The most simple parameters are the following :

- **colorMode** : let the retina process color information (if 1) or gray scale images (if 0). In this last case, only the first channel of the input will be processed.
- **normaliseOutput** : each channel has this parameter, if value is 1, then the considered channel output is rescaled between 0 and 255. Take care in this case at the Magnocellular output level (motion/transient channel detection). Residual noise will also be rescaled !

Note : using color requires color channels multiplexing/demultipexing which requires more processing. You can expect much faster processing using gray levels : it would require around 30 product per pixel for all the retina processes and it has recently been parallelized for multicore architectures.

Photo-receptors parameters

The following parameters act on the entry point of the retina - photo-receptors - and impact all the following processes. These sensors are low pass spatio-temporal filters that smooth temporal and spatial data and also adjust there sensitivity to local luminance thus improving details extraction and high frequency noise canceling.

- **photoreceptorsLocalAdaptationSensitivity** between 0 and 1. Values close to 1 allow high luminance log compression effect at the photo-receptors level. Values closer to 0 give a more linear sensitivity. Increased alone, it can burn the *Parvo (details channel)* output image. If adjusted in collaboration with **ganglionCellsSensitivity** images can be very contrasted whatever the local luminance there is... at the price of a naturalness decrease.
- **photoreceptorsTemporalConstant** this setups the temporal constant of the low pass filter effect at the entry of the retina. High value lead to strong temporal smoothing effect : moving objects are blurred and can disappear while static object are favored. But when starting the retina processing, stable state is reached lately.
- **photoreceptorsSpatialConstant** specifies the spatial constant related to photo-receptors low pass filter effect. This parameters specify the minimum allowed spatial signal period allowed in the following. Typically, this filter should cut high frequency noise. Then a 0 value doesn't cut anything noise while higher values start to cut high spatial frequencies and more and more lower frequencies... Then, do not go to high if you wanna see some details of the input images ! A good compromise for color images is 0.53 since this won't affect too much the color spectrum. Higher values would lead to gray and blurred output images.

Horizontal cells parameters

This parameter set tunes the neural network connected to the photo-receptors, the horizontal cells. It modulates photo-receptors sensitivity and completes the processing for final spectral whitening (part of the spatial band pass effect thus favoring visual details enhancement).

- **horizontalCellsGain** here is a critical parameter ! If you are not interested by the mean luminance and focus on details enhancement, then, set to zero. But if you want to keep some environment luminance data, let some low spatial frequencies pass into the system and set a higher value (<1).
- **hcellsTemporalConstant** similar to photo-receptors, this acts on the temporal constant of a low pass temporal filter that smooths input data. Here, a high value generates a high retina after effect while a lower value makes the retina more reactive.
- **hcellsSpatialConstant** is the spatial constant of the low pass filter of these cells filter. It specifies the lowest spatial frequency allowed in the following. Visually, a high value leads to very low spatial frequencies processing and leads to salient halo effects. Lower values reduce this effect but the limit is : do not go lower than the value of **photoreceptorsSpatialConstant**. Those 2 parameters actually specify the spatial band-pass of the retina.

NOTE after the processing managed by the previous parameters, input data is cleaned from noise and luminance in already partly enhanced. The following parameters act on the last processing stages of the two outing retina signals.

Parvo (details channel) dedicated parameter

- **ganglionCellsSensitivity** specifies the strength of the final local adaptation occurring at the output of this details dedicated channel. Parameter values remain between 0 and 1. Low value tend to give a linear response while higher values enforces the remaining low contrasted areas.

Note : this parameter can correct eventual burned images by favoring low energetic details of the visual scene, even in bright areas.

IPL Magno (motion/transient channel) parameters

Once image information is cleaned, this channel acts as a high pass temporal filter that only selects signals related to transient signals (events, motion, etc.). A low pass spatial filter smooths extracted transient data and a final logarithmic compression enhances low transient events thus enhancing event sensitivity.

- **parasolCells_beta** generally set to zero, can be considered as an amplifier gain at the entry point of this processing stage. Generally set to 0.
- **parasolCells_tau** the temporal smoothing effect that can be added
- **parasolCells_k** the spatial constant of the spatial filtering effect, set it at a high value to favor low spatial frequency signals that are lower subject to residual noise.
- **amacrinCellsTemporalCutFrequency** specifies the temporal constant of the high pass filter. High values let slow transient events to be selected.
- **V0CompressionParameter** specifies the strength of the log compression. Similar behaviors to previous description but here it enforces sensitivity of transient events.
- **localAdaptintegration_tau** generally set to 0, no real use here actually
- **localAdaptintegration_k** specifies the size of the area on which local adaptation is performed. Low values lead to short range local adaptation (higher sensitivity to noise), high values secure log compression.

CHAPTER

TWELVE

OPENCV IOS

-

  Title: *OpenCV iOS Hello*
  Compatibility: > OpenCV 2.4.3
  Author: Charu Hans
  You will learn how to link OpenCV with iOS and write a basic application.

-

  Title: *OpenCV iOS - Image Processing*
  Compatibility: > OpenCV 2.4.3
  Author: Charu Hans
  You will learn how to do simple image manipulation using OpenCV in iOS.

-

  Title: *OpenCV iOS - Video Processing*
  Compatibility: > OpenCV 2.4.3
  Author: Eduard Feicho
  You will learn how to capture and process video from camera using OpenCV in iOS.

12.1 OpenCV iOS Hello

Goal

In this tutorial we will learn how to:

- Link OpenCV framework with Xcode
- How to write simple Hello World application using OpenCV and Xcode.

Linking OpenCV iOS

Follow this step by step guide to link OpenCV to iOS.

1. Create a new XCode project.
2. Now we need to link *opencv2.framework* with Xcode. Select the project Navigator in the left hand panel and click on project name.
3. Under the TARGETS click on Build Phases. Expand Link Binary With Libraries option.
4. Click on Add others and go to directory where *opencv2.framework* is located and click open
5. Now you can start writing your application.

Hello OpenCV iOS Application

Now we will learn how to write a simple Hello World Application in Xcode using OpenCV.

- Link your project with OpenCV as shown in previous section.
- Open the file named *NameOfProject-Prefix.pch* (replace NameOfProject with name of your project) and add the following lines of code.

```
#ifdef __cplusplus
#import <opencv2/opencv.hpp>
#endif
```

- Add the following lines of code to viewDidLoad method in ViewController.m.

```
UIAlertView * alert = [[UIAlertView alloc] initWithTitle:@"Hello!" message:@"Welcome to OpenCV" delegate:self cancelBu
[alert show];
```

- You are good to run the project.

Output

12.2 OpenCV iOS - Image Processing

Goal

In this tutorial we will learn how to do basic image processing using OpenCV in iOS.

Introduction

In *OpenCV* all the image processing operations are done on *Mat*. iOS uses UIImage object to display image. One of the thing is to convert UIImage object to Mat object. Below is the code to convert UIImage to Mat.

```
- (cv::Mat)cvMatFromUIImage:(UIImage *)image
{
  CGColorSpaceRef colorSpace = CGImageGetColorSpace(image.CGImage);
  CGFloat cols = image.size.width;
  CGFloat rows = image.size.height;

  cv::Mat cvMat(rows, cols, CV_8UC4); // 8 bits per component, 4 channels

  CGContextRef contextRef = CGBitmapContextCreate(cvMat.data,                 // Pointer to  data
                                                  cols,                       // Width of bitmap
                                                  rows,                       // Height of bitmap
                                                  8,                          // Bits per component
                                                  cvMat.step[0],              // Bytes per row
                                                  colorSpace,                 // Colorspace
                                                  kCGImageAlphaNoneSkipLast |
                                                  kCGBitmapByteOrderDefault); // Bitmap info flags

  CGContextDrawImage(contextRef, CGRectMake(0, 0, cols, rows), image.CGImage);
  CGContextRelease(contextRef);
  CGColorSpaceRelease(colorSpace);

  return cvMat;
}

- (cv::Mat)cvMatGrayFromUIImage:(UIImage *)image
{
  CGColorSpaceRef colorSpace = CGImageGetColorSpace(image.CGImage);
  CGFloat cols = image.size.width;
  CGFloat rows = image.size.height;

  cv::Mat cvMat(rows, cols, CV_8UC1); // 8 bits per component, 1 channels

  CGContextRef contextRef = CGBitmapContextCreate(cvMat.data,                 // Pointer to data
                                                  cols,                       // Width of bitmap
                                                  rows,                       // Height of bitmap
                                                  8,                          // Bits per component
                                                  cvMat.step[0],              // Bytes per row
                                                  colorSpace,                 // Colorspace
                                                  kCGImageAlphaNoneSkipLast |
                                                  kCGBitmapByteOrderDefault); // Bitmap info flags

  CGContextDrawImage(contextRef, CGRectMake(0, 0, cols, rows), image.CGImage);
  CGContextRelease(contextRef);
  CGColorSpaceRelease(colorSpace);
```

```
  return cvMat;
 }
```

Once we obtain the Mat Object. We can do all our processing on Mat object, similar to cpp. For example if we want to convert image to gray, we can do it via below code.

```
cv::Mat greyMat;
cv::cvtColor(inputMat, greyMat, CV_BGR2GRAY);
```

After the processing we need to convert it back to UIImage.

```
-(UIImage *)UIImageFromCVMat:(cv::Mat)cvMat
{
  NSData *data = [NSData dataWithBytes:cvMat.data length:cvMat.elemSize()*cvMat.total()];
  CGColorSpaceRef colorSpace;

  if (cvMat.elemSize() == 1) {
      colorSpace = CGColorSpaceCreateDeviceGray();
  } else {
      colorSpace = CGColorSpaceCreateDeviceRGB();
  }

  CGDataProviderRef provider = CGDataProviderCreateWithCFData((__bridge CFDataRef)data);

  // Creating CGImage from cv::Mat
  CGImageRef imageRef = CGImageCreate(cvMat.cols,                                 //width
                                      cvMat.rows,                                 //height
                                      8,                                          //bits per component
                                      8 * cvMat.elemSize(),                       //bits per pixel
                                      cvMat.step[0],                            //bytesPerRow
                                      colorSpace,                                 //colorspace
                                      kCGImageAlphaNone|kCGBitmapByteOrderDefault,// bitmap info
                                      provider,                                   //CGDataProviderRef
                                      NULL,                                       //decode
                                      false,                                      //should interpolate
                                      kCGRenderingIntentDefault                   //intent
                                      );


  // Getting UIImage from CGImage
  UIImage *finalImage = [UIImage imageWithCGImage:imageRef];
  CGImageRelease(imageRef);
  CGDataProviderRelease(provider);
  CGColorSpaceRelease(colorSpace);

  return finalImage;
 }
```

Output

Check out an instance of running code with more Image Effects on YouTube .

12.3 OpenCV iOS - Video Processing

This tutorial explains how to process video frames using the iPhone's camera and OpenCV.

Prerequisites:

- Xcode 4.3 or higher
- Basic knowledge of iOS programming (Objective-C, Interface Builder)

Including OpenCV library in your iOS project

The OpenCV library comes as a so-called framework, which you can directly drag-and-drop into your XCode project. Download the latest binary from <http://sourceforge.net/projects/opencvlibrary/files/opencv-ios/>. Alternatively follow this guide *Installation in iOS* to compile the framework manually. Once you have the framework, just drag-and-drop into XCode:

Also you have to locate the prefix header that is used for all header files in the project. The file is typically located at "ProjectName/Supporting Files/ProjectName-Prefix.pch". There, you have add an include statement to import the opencv library. However, make sure you include opencv before you include UIKit and Foundation, because else you will get some weird compile errors that some macros like min and max are defined multiple times. For example the prefix header could look like the following:

```
//
// Prefix header for all source files of the 'VideoFilters' target in the 'VideoFilters' project
//

#import <Availability.h>

#ifndef __IPHONE_4_0
#warning "This project uses features only available in iOS SDK 4.0 and later."
#endif

#ifdef __cplusplus
#import <opencv2/opencv.hpp>
#endif

#ifdef __OBJC__
        #import <UIKit/UIKit.h>
        #import <Foundation/Foundation.h>
#endif
```

Example video frame processing project

User Interface

First, we create a simple iOS project, for example Single View Application. Then, we create and add an UIImageView and UIButton to start the camera and display the video frames. The storyboard could look like that:

Make sure to add and connect the IBOutlets and IBActions to the corresponding ViewController:

```
@interface ViewController : UIViewController
{
        IBOutlet UIImageView* imageView;
        IBOutlet UIButton* button;
}

- (IBAction)actionStart:(id)sender;

@end
```

Adding the Camera

We add a camera controller to the view controller and initialize it when the view has loaded:

```
#import <opencv2/highgui/cap_ios.h>
using namespace cv;


@interface ViewController : UIViewController
{
        ...
        CvVideoCamera* videoCamera;
}
...
@property (nonatomic, retain) CvVideoCamera* videoCamera;

@end
```

```
- (void)viewDidLoad
{
        [super viewDidLoad];
        // Do any additional setup after loading the view, typically from a nib.

        self.videoCamera = [[CvVideoCamera alloc] initWithParentView:imageView];
        self.videoCamera.defaultAVCaptureDevicePosition = AVCaptureDevicePositionFront;
        self.videoCamera.defaultAVCaptureSessionPreset = AVCaptureSessionPreset352x288;
        self.videoCamera.defaultAVCaptureVideoOrientation = AVCaptureVideoOrientationPortrait;
        self.videoCamera.defaultFPS = 30;
        self.videoCamera.grayscale = NO;
}
```

In this case, we initialize the camera and provide the imageView as a target for rendering each frame. CvVideoCamera is basically a wrapper around AVFoundation, so we provie as properties some of the AVFoundation camera options. For example we want to use the front camera, set the video size to 352x288 and a video orientation (the video camera normally outputs in landscape mode, which results in transposed data when you design a portrait application).

The property defaultFPS sets the FPS of the camera. If the processing is less fast than the desired FPS, frames are automatically dropped.

The property grayscale=YES results in a different colorspace, namely "YUV (YpCbCr 4:2:0)", while grayscale=NO will output 32 bit BGRA.

Additionally, we have to manually add framework dependencies of the opencv framework. Finally, you should have at least the following frameworks in your project:

- opencv2
- Accelerate
- AssetsLibrary
- AVFoundation
- CoreGraphics
- CoreImage
- CoreMedia
- CoreVideo
- QuartzCore
- UIKit
- Foundation

Processing frames

We follow the delegation pattern, which is very common in iOS, to provide access to each camera frame. Basically, the View Controller has to implement the CvVideoCameraDelegate protocol and has to be set as delegate to the video camera:

```
@interface ViewController : UIViewController<CvVideoCameraDelegate>
```

```
- (void)viewDidLoad
{
        ...
        self.videoCamera = [[CvVideoCamera alloc] initWithParentView:imageView];
        self.videoCamera.delegate = self;
        ...
}
```

```
#pragma mark - Protocol CvVideoCameraDelegate

#ifdef __cplusplus
- (void)processImage:(Mat&)image;
{
        // Do some OpenCV stuff with the image
}
#endif
```

Note that we are using C++ here (cv::Mat). Important: You have to rename the view controller's extension .m into .mm, so that the compiler compiles it under the assumption of Objective-C++ (Objective-C and C++ mixed). Then, __cplusplus is defined when the compiler is processing the file for C++ code. Therefore, we put our code within a block where __cplusplus is defined.

Basic video processing

From here you can start processing video frames. For example the following snippet color-inverts the image:

```
- (void)processImage:(Mat&)image;
{
        // Do some OpenCV stuff with the image
        Mat image_copy;
        cvtColor(image, image_copy, CV_BGRA2BGR);

        // invert image
        bitwise_not(image_copy, image_copy);
        cvtColor(image_copy, image, CV_BGR2BGRA);
}
```

Start!

Finally, we have to tell the camera to actually start/stop working. The following code will start the camera when you press the button, assuming you connected the UI properly:

```
#pragma mark - UI Actions

- (IBAction)actionStart:(id)sender;
{
```

```
        [self.videoCamera start];
}
```

Hints

Try to avoid costly matrix copy operations as much as you can, especially if you are aiming for real-time. As the image data is passed as reference, work in-place, if possible.

When you are working on grayscale data, turn set grayscale = YES as the YUV colorspace gives you directly access the luminance plane.

The Accelerate framework provides some CPU-accelerated DSP filters, which come handy in your case.

CHAPTER

THIRTEEN

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: Unfortunetly we have no tutorials into this section. And you can help us with that, since OpenCV is a community effort. If you have a tutorial suggestion or you have written a tutorial yourself (or coded a sample code) that you would like to see here, please contact follow these instructions: *How to write a tutorial for OpenCV* and How to contribute.