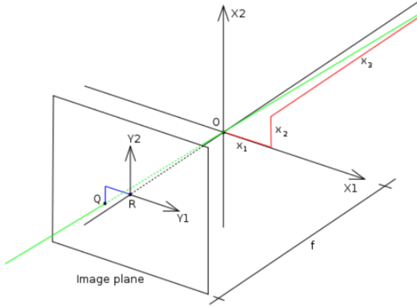
Computer Vision Libraries



Computer Vision Libraries
OpenCV Installation
Introduction
Basic Image Container
Image Storage
Space Domain Techniques
Point Processing Techniques
Kernels
Edges
Video

OpenCV

OpenCV Installation

1 .- Install the following required packages

```
$ sudo apt-get install build-essential
$ sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev
$ sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev
libjasper-dev libdc1394-22-dev
```

2 .- Getting OpenCV Source Code

```
$ mkdir opencv
$ cd opencv
$ git clone https://github.com/opencv/opencv.git
```

3 .- Building OpenCV from Source Using CMake

```
$ cd opencv
$ mkdir build
$ cd build
```

4 .- Configuring, Building and Install

```
$ cmake -D CMAKE_BUILD_TYPE=Release
    -D CMAKE_INSTALL_PREFIX=/usr/local ..
$ make
$ sudo make install
```

NOTE: You may need to copy by hand the commands (ctrl+c may not work)

Introduction

Load and Display an Image using OpenCV

```cpp
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <stdio.h>

using namespace cv;

int main(int argc, char** argv )
{
    if ( argc != 2 )
    {
        printf("usage: DisplayImage <Image_Path>\n");
        return -1;
    }
    Mat image;
    image = imread( argv[1], IMREAD_COLOR );
    if ( !image.data )
    {
        printf("No image data \n");
        return -1;
    }
    namedWindow("Display Image", WINDOW_AUTOSIZE );
    imshow("Display Image", image);
    waitKey(0);
    return 0;
}
```

Create a CMake file

```cmake
cmake_minimum_required(VERSION 2.8)
project( DisplayImage )
find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )
add_executable( DisplayImage DisplayImage.cpp )
target_link_libraries( DisplayImage ${OpenCV_LIBS} )
```

Generate the executable

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

Run it giving an image location as an argument

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

Introduction

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

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

OpenCV has its own namespace: cv

```
using namespace cv;
```

We start up assuring that we acquire a valid image name argument from the command line:

```
if ( argc != 2 )
{
    printf("usage: DisplayImage <Image_Path>\n");
    return -1;
}
```

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

```
Mat image;
```

Now we call the **cv::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:

```
image = imread( argv[1], IMREAD_COLOR );
```

After checking that the image data was loaded correctly, we want to display our image, so we create an OpenCV window using the **cv::namedWindow** function and finally, to update the content of the OpenCV window with a new image use the **cv::imshow** function

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

Introduction

Modify and Save an Image using OpenCV

```cpp
#include <opencv2/opencv.hpp>

using namespace cv;

int main( int argc, char** argv )
{
 char* imageName = argv[1];
 Mat image;
 image = imread( imageName, IMREAD_COLOR );
 if( argc != 2 || !image.data )
 {
  printf( " No image data \n " );
  return -1;
 }
 Mat gray_image;
 cvtColor( image, gray_image, COLOR_BGR2GRAY );
 imwrite( "Gray_Image.jpg", gray_image );
 namedWindow( imageName, WINDOW_AUTOSIZE );
 namedWindow( "Gray image", WINDOW_AUTOSIZE );
 imshow( imageName, image );
 imshow( "Gray image", gray_image );
 waitKey(0);
 return 0;
}
```

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:

```cpp
cvtColor( image, gray_image, COLOR_BGR2GRAY );
```

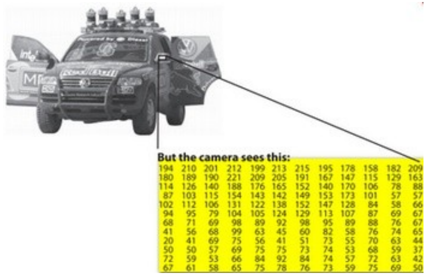
To save it, we will use a function analogous to cv::imread : **cv::imwrite**

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

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

```cpp
waitKey(0);
```

Basic Image Container



But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	74	65
20	41	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

The mirror of the car is nothing more than a matrix containing all the intensity values of the pixel points.

Mat

Mat is 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
- **Pointer to the matrix** containing the pixel values

Mat has the following properties:

- 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 **cv::Mat::clone()** and **cv::Mat::copyTo()** functions.

```
Mat matA, matC;                            // creates just the header parts
matA = imread(argv[1], IMREAD_COLOR); // allocate matrix
Mat matB(matA);                            // Use the copy constructor
matC = matA;                               // Assignment operator
Mat matD(matA, Rect(10, 10, 100, 100) ); // using a rectangle
Mat matE = matA(Range::all(), Range(1,3)); // using row and column
Mat matF = matA.clone();
Mat matG;
matA.copyTo(matG);
```

Mat

Create a Mat object in multiple ways

- **cv::Mat::Mat** Constructor

```
Mat matM(2,2, CV_8UC3, Scalar(0,0,255));
```

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

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

- **cv::Mat::create** function

```
Mat.create(4,4, CV_8UC(2));
```

- For small matrices you may use comma separated

```
Mat matC = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
```

Output formatting

- Default

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

- Comma separated values (CSV)

```
cout << "matR (csv)= " << endl << format(matR, Formatter::FMT_CSV );
```

- 2D Point

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

- 3D Point

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

Image Storage

The size of the matrix depends on 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 BGR 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

The efficient method

The classic C style operator[] (pointer) access:

```
Mat& ScanImageAndReduceC(Mat& matI, const uchar* const table)
{
    CV_Assert(matI.depth() == CV_8U);  //only char
    int channels = matI.channels();
    int nRows = matI.rows;
    int nCols = matI.cols * channels;
    if (matI.isContinuous())
    {
        nCols *= nRows;
        nRows = 1;
    }
    int idx,jdx;
    uchar* ptr;
    for( idx = 0; idx < nRows; ++idx)
    {
        ptr = matI.ptr<uchar>(idx);
        for ( jdx = 0; jdx < nCols; ++jdx)
        {
            ptr[jdx] = table[ptr[jdx]];
        }
    }
    return matI;
}
```

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

The iterator (safe) method

```
Mat& ScanImageAndReduceIterator(Mat& matI, const uchar* const table)
{
CV_Assert(matI.depth() == CV_8U);   //only char
const int channels = matI.channels();
switch(channels)
{
    case 1:
    {
        MatIterator_<uchar> it, end;
        for( it = matI.begin<uchar>(), end = matI.end<uchar>(); it != end; ++it)
            *it = table[*it];
        break;
    }
    case 3:
    {
        MatIterator_<Vec3b> it, end;
        for( it = matI.begin<Vec3b>(), end = matI.end<Vec3b>(); it != end; ++it)
        {
            (*it)[0] = table[(*it)[0]];
            (*it)[1] = table[(*it)[1]];
            (*it)[2] = table[(*it)[2]];
        }
    }
}
return matI;
}
```

- The iterator method is considered a safer way as it takes over these tasks from the user
- Ask the begin and the end of the image matrix and then just increase the begin iterator until you reach the end
- In case of color images we have three uchar items per column (Vec3b)

```
Vec3b intensity = img.at<Vec3b>(y, x);
uchar blue = intensity.val[0];
uchar green = intensity.val[1];
uchar red = intensity.val[2];
```

Performance Difference

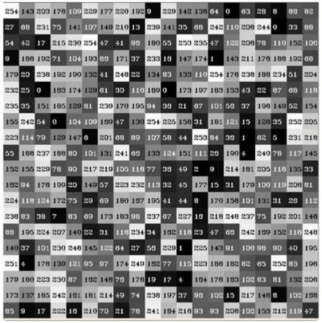
To make the differences more clear, a quite large (2560 X 1600) image is used. The performance presented here are for color images. For a more accurate value I've averaged the value I got from the call of the function for hundred times.

Method	Time
Efficient	79.4717 milliseconds
Iterator	83.7201 milliseconds

Space Domain Techniques - Histogram

- Histograms are collected counts of data organized into a set of predefined bins
- The data collected can be whatever feature you find useful to describe your image
- Since we know that the range of information value, we can segment our range in subparts (**bins**)

Imagine that a Matrix contains information of an image (i.e. intensity in the range 0-255):



254	143	203	176	109	229	177	220	192	9	229	142	138	64	0	63	28	8	88	82
27	68	231	75	141	107	149	210	13	239	141	35	68	242	110	208	244	0	33	88
54	42	17	215	230	254	47	41	98	180	55	253	235	47	122	208	78	110	152	100
9	186	192	71	104	193	88	171	37	233	18	147	174	1	143	211	176	188	192	68
179	20	238	192	190	132	41	248	22	134	83	133	110	254	176	238	188	234	51	204
232	25	0	183	174	129	61	30	110	189	0	173	197	183	153	43	22	87	68	118
235	35	151	185	129	81	239	170	195	94	38	21	67	101	58	37	196	149	52	154
155	242	54	0	104	109	169	47	130	254	225	156	31	181	121	15	128	35	252	205
223	114	79	129	147	6	201	68	89	107	58	44	253	84	36	1	62	5	231	218
55	188	237	188	80	101	131	241	68	133	124	151	111	28	190	4	240	78	117	145
152	155	229	78	90	217	219	105	116	77	38	49	2	9	214	181	205	116	135	33
182	94	176	199	20	149	57	223	232	113	32	45	177	15	31	179	100	119	208	81
224	118	124	172	75	29	69	180	187	195	41	44	8	170	158	101	131	31	28	112
238	83	38	7	83	69	173	183	98	237	67	227	18	218	248	237	75	192	201	146
88	195	224	207	140	22	31	118	234	34	182	116	23	47	68	242	169	152	116	248
140	37	101	230	246	145	122	64	27	58	229	1	225	143	91	100	98	90	40	195
251	4	178	139	121	95	97	174	249	182	77	115	223	186	182	82	65	252	83	196
179	180	223	230	87	182	148	78	176	19	17	4	184	176	183	102	83	81	132	206
173	137	185	242	181	181	214	49	74	238	197	37	98	102	15	217	148	8	102	168
85	9	17	222	18	210	70	21	78	241	184	216	93	93	208	102	153	212	119	47

OpenCV implements the function **cv::calcHist** , which calculates the histogram of a set of arrays (usualy images or image planes). It can operate with up to 32 dimensions.

Space Domain Techniques - Histogram

```cpp
#include "opencv2/highgui.hpp"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>

int main(int argc, char** argv)
{
    Mat src; //... Image readed
    vector<Mat> bgr_planes;
    split( src, bgr_planes );
    int histSize = 256;
    float range[] = { 0, 256 }; //the upper boundary is exclusive
    const float* histRange = { range };
    bool uniform = true, accumulate = false;
    Mat b_hist, g_hist, r_hist;
    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 );
    int hist_w = 512, hist_h = 400;
    int bin_w = cvRound( (double) hist_w/histSize );
    Mat histImage( hist_h, hist_w, CV_8UC3, Scalar( 0,0,0) );
    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() );
```

```cpp
    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  );
    }
    imshow("Source image", src );
    imshow("calcHist Demo", histImage );
    waitKey();
    return 0;
}
```

Space Domain Techniques - Histogram

Calculate the histograms by using the OpenCV function **cv::calcHist**

```
Mat b_hist, g_hist, r_hist;
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 );
```

- **&bgr_planes[0]** The source array(s)
- **1** The number of source arrays (in this case we are using 1. We can enter here also a list of arrays)
- **0** The channel (dim) to be measured. In this case it is just the intensity (each array is single-channel) so we just write 0.
- **Mat()** A mask to be used on the source array (zeros indicating pixels to be ignored). If not defined it is not used
- **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.

Space Domain Techniques - Histogram

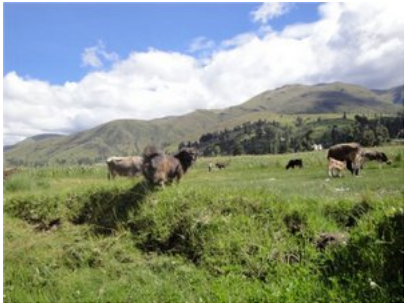
cv::normalize the histogram so its values fall in the range indicated by the parameters entered

```
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() );
```

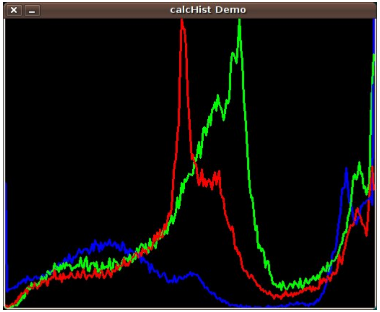
- **b_hist** Input array
- **b_hist** Output normalized array (can be the same)
- **0** and **histImage.rows** For this example, they are the lower and upper limits to normalize the values of **r_hist**
- **NORM_MINMAX** Argument that indicates the type of normalization (it adjusts the values between the two limits set before)
- **b_hist** The Mat object where the histogram will be stored
- **-1** Implies that the output normalized array will be the same type as the input
- **Mat()** Optional mask

Space Domain Techniques - Histogram

Using as input argument an image like this one

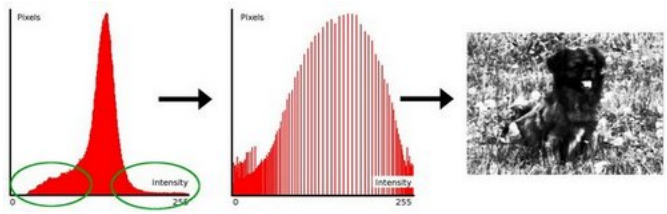


Produces the following histogram



Space Domain Techniques - Histogram Equalization

- It is a method that improves the contrast in an image, in order to stretch out the intensity range
- 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. 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.



Space Domain Techniques - Histogram Equalization

```cpp
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;
int main( int argc, char** argv )
{
    Mat src; //... Image readed
    cvtColor( src, src, COLOR_BGR2GRAY );
    Mat dst;
    equalizeHist( src, dst );
    imshow( "Source image", src );
    imshow( "Equalized Image", dst );
    waitKey();
    return 0;
}
```

- Convert it to grayscale

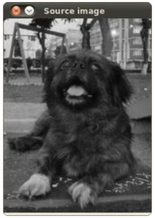
```cpp
cvtColor( src, src, COLOR_BGR2GRAY );
```

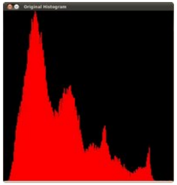
- Apply histogram equalization with the function **cv::equalizeHist**

```cpp
   Mat dst;
   equalizeHist( src, dst );
```

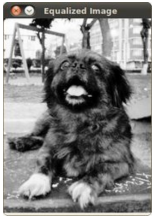
Space Domain Techniques - Histogram Equalization

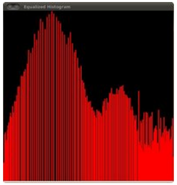
Source Image





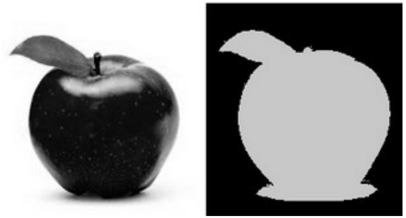
Equalized Image





Point Processing Techniques - Threshold

- The simplest segmentation method
- Separate out regions of an image corresponding to objects which we want to analyze
- To differentiate the pixels we are interested in from the rest
- Once we have separated properly the important pixels, we can set them with a determined value to identify them



Point Processing Techniques - Threshold

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

// 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* );

int main( int argc, char** argv )
{
  /// Load an image
  src = imread( argv[1], 1 );
  cvtColor( src, src_gray, CV_BGR2GRAY );
  namedWindow( window_name, CV_WINDOW_AUTOSIZE );
  createTrackbar( trackbar_type,
                  window_name, &threshold_type,
                  max_type, Threshold_Demo );
  createTrackbar( trackbar_value,
                  window_name, &threshold_value,
                  max_value, Threshold_Demo );
  Threshold_Demo( 0, 0 );
  while(true)
  {
    int c;
    c = waitKey( 20 );
    if( (char)c == 27 )
      { break; }
  }
}
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 );
}
```

Point Processing Techniques - Threshold

- Create 2 trackbars for the user to enter user input

```
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:

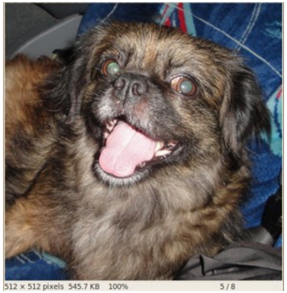
```
void Threshold_Demo( int, void* )
```

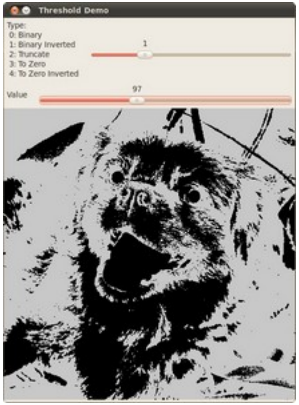
The function **threshold**:

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

- **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

Point Processing Techniques - Threshold

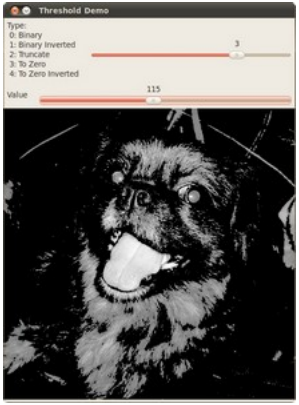




Type:
0: Binary
1: Binary Inverted
2: Truncate
3: To Zero
4: To Zero Inverted

1

Value 97



Type:
0: Binary
1: Binary Inverted
2: Truncate
3: To Zero
4: To Zero Inverted

3

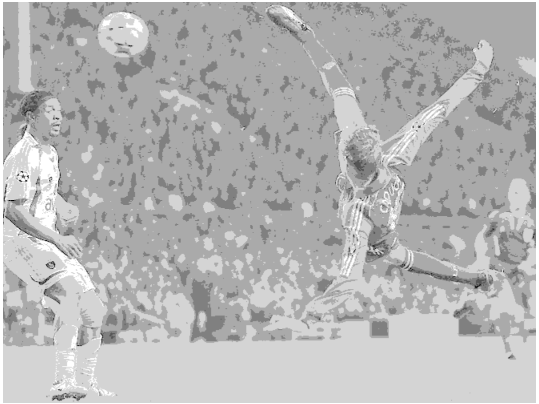
Value 115

Point Processing Techniques - Log

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

int main ()
{
    cv::Mat binary = cv::imread("path",0);

    cv::Mat fg;
    binary.convertTo(fg,CV_32F);
    fg = fg + 1;
    cv::log(fg,fg);
    cv::convertScaleAbs(fg,fg);
    cv::normalize(fg,fg,0,255,cv::NORM_MINMAX);
    cv::imshow("a",fg);
    cv::waitKey(0);
}
```



Kernels

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.

Consider, for example, the issue of an image contrast enhancement method. Basically we want to apply for every pixel of the image the following formula:

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

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

Smoothing Images

Smoothing, also called blurring, is a simple and frequently used image processing operation.
There are many reasons for smoothing, for example, to reduce noise.
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. $f(i+k, j+l)$) :

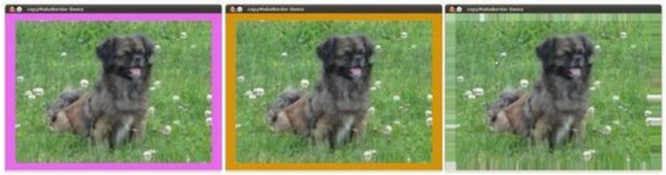
$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$$

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

- Normalized Box Filter
- Gaussian Filter
- Median Filter
- Bilateral Filter

Borders

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?
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).

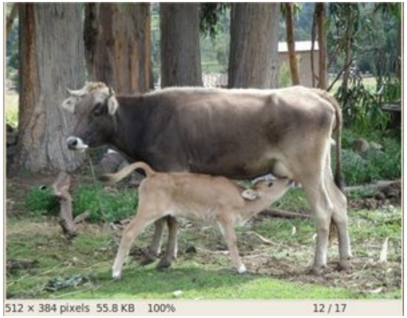


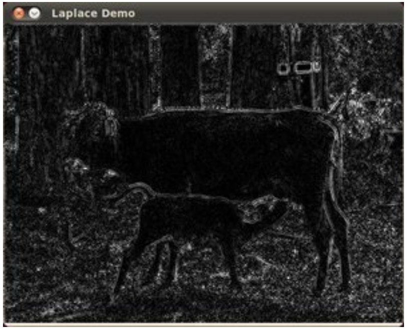
Laplacian

The Laplacian operator is defined by:

$$\mathfrak{Laplace}(f) = \frac{\partial f^2}{\partial x^2} + \frac{\partial f^2}{\partial y^2}$$

The Laplacian operator is implemented in OpenCV by the function Laplacian.





Video

Open a camera with

```
cv::VideoCapture input_video;
input_video.open(camera_id);
```

Check if it is open

```
if(!input_video.isOpened())
```

Read a frame with

```
input_video.read(frame_input);
```

At the end, release the camera

```
input_video.release();
```