OPENCV

MANOEL NETO

WHAT IS OPENCV?

- Is an open source C++ library for image processing and computer vision
- Developed by Intel
- Supported by Willow Garage
- Free for both commercial and non-commercial use
- It is a library of many inbuilt functions mainly aimed at real time image processing
- it has several hundreds of image processing and computer vision algorithms which make developing advanced computer vision applications easy and efficient

KEY FEATURES

- Optimized for real time image processing & computer vision applications
- Primary interface of OpenCV is in C++
- There are also C, Python and JAVA full interfaces
- OpenCV applications run on Windows, Android, Linux, Mac and iOS
- Optimized for Intel processors

INSTALL

- **Instalar o OpenCV**
- **http://opencv.org/downloads.html**
- **Instalar o Eclipse / CDT**
- **http://www.eclipse.org/cdt/downloads.php**

CONFIGURE ECLIPSE

- http://docs.opencv.org/doc/tutorials/introduction/linux_eclipse/linux_eclipse.html

BASICS OF OPENCV API

- **The main modules of OpenCV are listed below**
  - **Core**: basic module of OpenCV. It includes basic data structures (e.g.- **Mat** data structure) and basic image processing functions
  - **Highgui:** provides simple user interface capabilities, several image and video codecs, image and video capturing capabilities, manipulating image windows, handling track bars and mouse events and etc.
  - **Imgproc:** includes basic image processing algorithms including image filtering, image transformations, color space conversions and etc.
  - **Video:** This is a video analysis module which includes object tracking algorithms, background subtraction algorithms and etc.
  - **Objdetect:** This includes object detection and recognition algorithms for standard objects.

BASICS OF OPENCV API

What is an IMAGE?

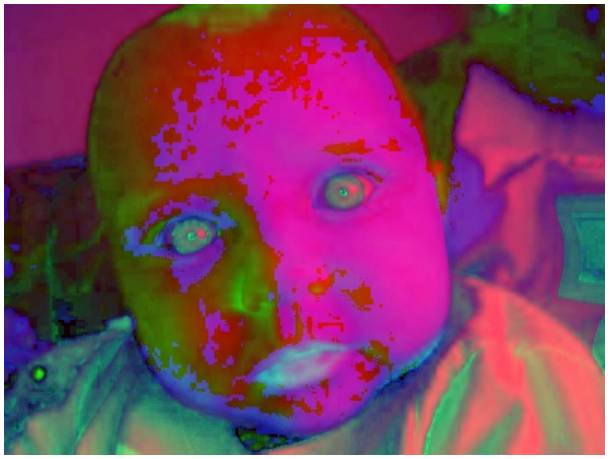
- **Any digital image consists of pixels (a matrix of pixels).**
- **Any pixel should have some value.**
- **The minimum value for a pixel is 0 and it represents black color.**
- **When the value of the pixel is increased, the intensity of that pixel is also increased.**
- **what is image-depth? image-depth** means the number of bits allocated for each pixel.
  - If it is 8, each pixel can have a value between 0 and 255.
  - If it is 4, each pixel can have a value between 0 to 15 (1111 in binary).

PIXELTYPES

PixelTypes shows how the image is represented in data

- BGR - The default color of imread(). Normal 3 channel color
- HSV - Hue is color, Saturation is amount, Value is lightness. 3 channels
- GRAYSCALE - Gray values, Single channel

- OpenCV requires that images be in BGR or Grayscale in order to be shown or saved. Otherwise, undesirable effects may appear.


BASICS OF OPENCV API

Data Types for Arrays

- **Data type of an array defines the number of bits allocated for each element of array (pixels in an image) and how the value is represented using those bits. Any array elements should have one of following data types:**

CV_8U (8 bit unsigned integer)

CV_8S (8 bit signed integer)

CV_16U (16 bit unsigned integer)

CV_16S (16 bit signed integer)

CV_32S (32 bit signed integer)

CV_32F (32 bit floating point number)

CV_64F (64 bit float floating point number)

BASICS OF OPENCV API

Example Single channel array with 8 bit unsigned integers

54	58	255	8	0
45	24	25	214	23
85	124	85	23	55
22	78	25	21	0
52	52	36	127	47

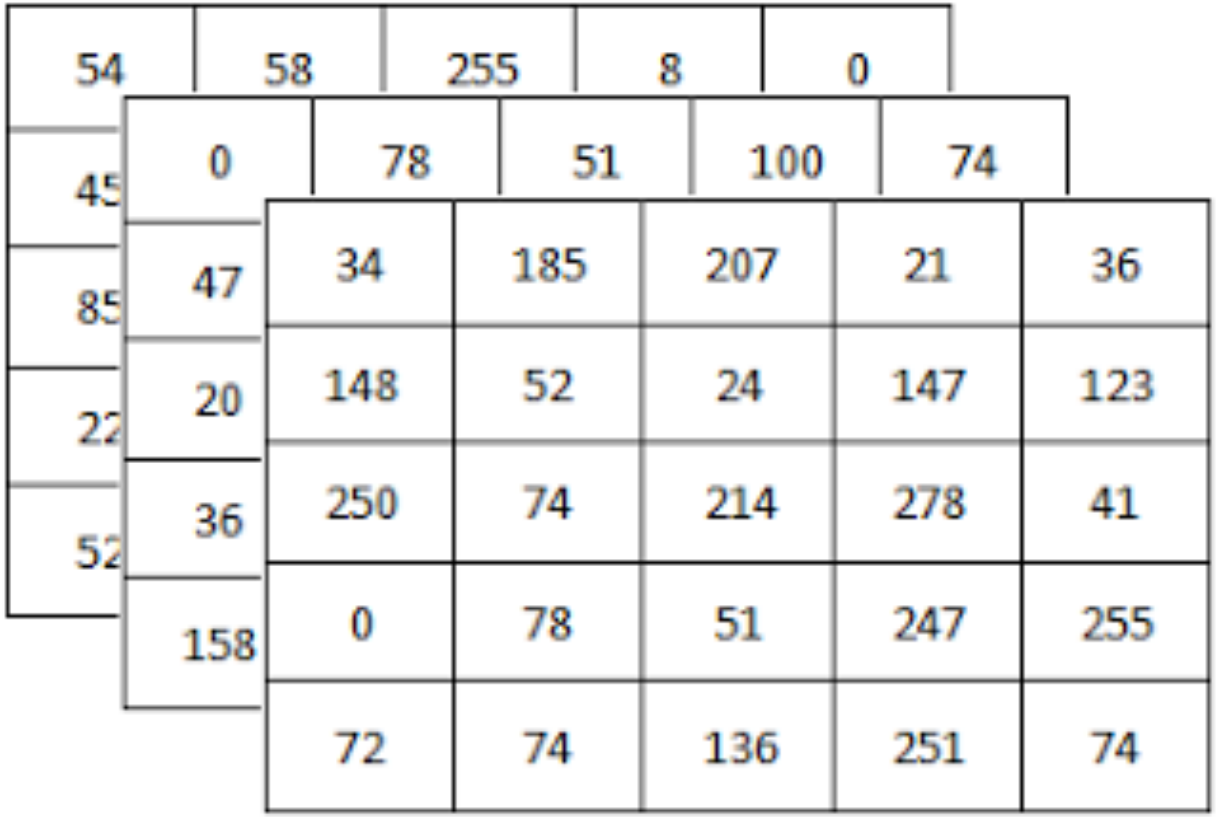
BASICS OF OPENCV API

For multi channel arrays :

- **CV_8UC1 (single channel array with 8 bit unsigned integers)**
- **CV_8UC2 (2 channel array with 8 bit unsigned integers)**
- **CV_8UC3 (3 channel array with 8 bit unsigned integers)**
- **CV_8UC4 (4 channel array with 8 bit unsigned integers)**
- **CV_8UC(n) (n channel array with 8 bit unsigned integers (n can be from 1 to 512))**

BASICS OF OPENCV API

Example 3 channel array with 8 bit unsigned integers:



BASICS OF OPENCV API

Example 2 channel array with 8 bit signed integers:

-85	25	120	127	-128	
48	-127	23	48	-54	100
-50	76	52	24	-47	23
0	64	74	-14	78	41
72	-3	-78	51	47	55
	23	74	127	51	74

EXAMPLE

Here is some properties of the following image:

23	23	34	255	0
78	245	129	25	251
23	12	89	90	37
84	26	47	127	199

Image-depth 8 bit

1 channel (So, this is a grayscale image, no color content)

The height is 4 pixel

The width is 5 pixels

The resolution of this image is 4x5.

EXAMPLE

- Image-depth 24 bit / 3 channels (this is a color image)
- The height is 4 pixel and The width is 5 pixels
- The resolution of this image is 4x5.
- Color image should consist of at least 3 planes; Red, Green and Blue. Any pixel is a combination of three 3 values. (255, 0, 0) represent pure red. (0, 255, 0) represent pure green. (255, 0, 255) represents pure violate.
- Why 24 bit of image-depth?

Red Plane				
23	23	34	255	0
78	245	129	25	251
23	12	89	90	37
84	26	47	127	199

Green Plane				
231	0	35	45	45
77	21	79	1	74
145	154	47	10	34
71	255	74	27	19

Blue Plane				
46	45	3	78	13
75	50	70	71	42
14	214	111	74	88
123	72	90	13	67

HELLO OPENCV

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


using namespace cv;
```

All OpenCV classes and functions are in cv namespace.

```
using namespace std;
Mat img = imread("MyPic.JPG",
CV_LOAD_IMAGE_UNCHANGED);
```

//read the image data in the file "MyPic.JPG" and store it in 'img' Mat datastructure

HELLO OPENCV

- **Mat img = imread(**const string& **filename**, int **flags**=CV_LOAD_IMAGE_COLOR**)**
- **Mat** is a data structure to store images in a matrix.
- **Mat** data structure in the above program. It is declared in **"opencv2/core/core.hpp"** header file.
- Then why don't we include this? It's because **"opencv2/highgui/highgui.hpp"** header file include that header file inside it. So, we don't need to include it again in our program.
- **imread()** is a function declared in **"opencv2/highgui/highgui.hpp"** header file. **It loads an image from a file and stores it in Mat data structure.**

HELLO OPENCV

- **Arguments of imread() function**
  - **filename** - location of the file. If you just give the filename only, that image should be in the same folder as your C++ file. Otherwise you have to give the full path to your image.
  - **flags** - There are five possible inputs:
    - CV_LOAD_IMAGE_UNCHANGED - image-depth=8 bits per pixel in each channel, no. of channels=unchanged
    - CV_LOAD_IMAGE_GRAYSCALE - image depth=8 bits, no. of channels=1
    - CV_LOAD_IMAGE_COLOR - image-depth=?, no. of channels=3
    - CV_LOAD_IMAGE_ANYDEPTH - image-depth=unchanged , no. of channels=?
    - CV_LOAD_IMAGE_ANYCOLOR - image-depth=?, no. of channels=unchanged

- **if (img.empty()) //check whether the image is loaded or not**
- **bool Mat::empty()**
- **This function returns true, if Mat::data==NULL or Mat::total() == 0**
  - If imread() function fails to load the image, 'img' will not be loaded any data.
  - Therefore 'img.empty()' should return true. It's a good practice to check whether the image is loaded successfully and if not exit the program.
  - Otherwise your program will crash when executing imshow() function.

- **void namedWindow(const string& winname, int flags = WINDOW_AUTOSIZE);**
  - This function creates a window. Parameters :
    - **winname** - Title of the window. That name will display in the title bar of the newly created window.
    - **flags** - determine the size of the window. There are two options:
      - **WINDOW_AUTOSIZE -** User cannot resize the image. Image will be displayed in its original size
      - **CV_WINDOW_NORMAL -** Image will resized if you resize the the window

HELLO OPENCV

- **void imshow(const string& winname, InputArray mat);**
  - This function shows the image which is stored in the 'mat' in a window specified by winname.
  - If the window is created with **WINDOW_AUTOSIZE** flag, image will be displayed in its original size. Otherwise image may be scaled to the size of the window.
  - Parameters:
    - winname - Title of the window. This name is used to identify the window created by namedWindow() function.
    - mat - hold the image data

HELLO OPENCV

- **int waitKey(int delay = 0);**
  - waitKey() function wait for keypress for certain time, specified by **delay** (in milliseconds).
  - If **delay** is zero or negative, it will wait for infinite time.
  - If any key is pressed, this function returns the ASCII value of the key and your program will continue.
  - If there is no key press for the specified time, it will return -1 and program will continue.

HELLO OPENCV

- **void destroyWindow(const string& winname)**
  - This function closes the opened window, with the title of **winname** and deallocate any associated memory usage.
  - This function is not essential for this application because when the program exits, operating system usually close all the opened windows and deallocate any associated memory usage.

HELLO OPENCV

- **Mat (int rows, int cols, int type, const Scalar& s);**
  - This is the one of the many constructor available in Mat class. It initialize the Mat object with the value given by the Scalar object.
  - Parameters :
    - **rows** - Number of rows in the 2D array (height of the image in pixels)
    - **cols** - Number of columns in the 2D array (width of the image in pixels)
    - **type** - specify the bit depth, data type and number of channels of the image.
      - Here are some of possible inputs for this parameter:
        - CV_8UC1
        - CV_8UC3
        - CV_64FC1

HELLO OPENCV

- **Mat(int rows, int cols, int type, const Scalar& s);**
  - Scalar objetct Initialize each array element with the value given.
  - For example: Scalar(0,0,100). First channel (Blue plane) with 0, 2nd channel (Green plane) with 0 and 3rd channel (Red Plane) with 100.
  - In this case final image is red!!

CAPTURE VIDEO FROM FILE OR CAMERA

- **VideoCapture(const string& filename)**
  - This is one of few constructors available in VideoCapture class.
  - This constructor open the video file and initializes the VideoCapture object for reading the video stream from the specified file.
- **VideoCapture::VideoCapture(int device)**
  - This constructor open the camera indexed by the argument of this constructor and initializes the VideoCapture object for reading the video stream from the specified camera.
- **bool VideoCapture::IsOpened()**
  - If the previous call to VideoCapture constructor is successful, this method will return true. Otherwise it will return false.

CAPTURE VIDEO FROM FILE OR CAMERA

- **bool VideoCapture::set(int propId, double value)**
  - You can change some properties of VideoCapture object.
  - If it is successful, this method will return true. Otherwise it will return false.
  - Parameters :
    - **int propID -** This argument specify the property you are going to change. There are many options for this argument. Ex.:
      - **CV_CAP_PROP_POS_MSEC -** current position of the video in milliseconds
      - **CV_CAP_PROP_POS_FRAMES -** current position of the video in frames
      - **CV_CAP_PROP_FRAME_WIDTH -** width of the frame of the video stream
      - **CV_CAP_PROP_FRAME_HEIGHT -** height of the frame of the video stream
      - **CV_CAP_PROP_FPS -** frame rate (frames per second)
      - **CV_CAP_PROP_FOURCC** - four character code of codec
    - **double value -** This is the new value you are going to assign to the property, specified by the **propID**

CAPTURE VIDEO FROM FILE OR CAMERA

- **double VideoCapture::get(int propId)**
  - This function returns the value of the property which is specified by **propId.**
  - Parameters : it uses the same **propId list that we saw before.**

CAPTURE VIDEO FROM FILE OR CAMERA

- **bool VideoCapture::read(Mat& image);**
  - The function grabs the next frame from the video, decodes it and stores it in the **Mat 'image'** variable.

WRITE IMAGE TO FILE

- **imwrite("D:/TestImage.jpg", img, compression_params);**
  - The function saves the image in the variable '**img**' to a file, specified by **'filename'** . If this function fails to save the image, it will return false. On success of writing the file to the harddisk, it will return true.
- **vector<int> compression_params;**
  - This is a int vector to which you have to insert some int parameters specifying the format of the image. They are:
    - JPEG format - You have to puch_back **CV_IMWRITE_JPEG_QUALITY** first and then a number between 0 and 100 (higher is the better). If you want the best quality output, use 100. I have used 98 in the above sample program. But higher the value, it will take longer time to write the image
    - PNG format - You have to puch_back **CV_IMWRITE_PNG_COMPRESSION** first and then a number between 0 and 9 (higher is the better compression, but slower).

WRITE IMAGE TO FILE

- **The image format is chosen depending on the file name extension.**
- **Only images with 8 bit or 16 bit unsigned single channel or 3 channel (CV_8UC1, CV_8UC3, CV_8SC1, CV_8SC3, CV_16UC1, CV_16UC3) with 'BGR'** channel order, can be saved.
- If the depth or channel order of the image is different, use **'Mat::convertTo()'** or **'cvtColor'** functions to convert the image to supporting format before using **imwrite** functio**.**

WRITE VIDEO TO FILE

- **Size frameSize(static_cast<int>(dWidth), static_cast<int>(dHeight))**
  - Create a **Size** object with a given width and height.
  - Important: I have cast the width and height to integers because they are originally double values and the **Size** constructor does not accept double values as its parameters.
- **VideoWriter(const string& filename, int fourcc, double fps, Size frameSize, bool isColor=true)**
  - This is the constructor of the VideoWriter class. It initializes the object with following parameters:
    - **const string& filename** - Specify the name and the location of the output file. The video stream is written into this file

WRITE VIDEO TO FILE

- **This is the constructor of the VideoWriter class. It initializes the object with following parameters:**
  - **const string& filename** - Specify the name and the location of the output file. The video stream is written into this file
  - **int fourcc** - specify the 4 character code for the codec which is used to compress the video. Your computer may not be supported some codecs. So, if you fail to save the video, please try other codecs.
    - CV_FOURCC('D', 'I', 'V', '3') for DivX MPEG-4 codec
    - CV_FOURCC('M', 'P', '4', '2') for MPEG-4 codec
    - CV_FOURCC('D', 'I', 'V', 'X') for DivX codec
    - CV_FOURCC('P','I','M','1') for MPEG-1 codec
    - CV_FOURCC('I', '2', '6', '3') for ITU H.263 codec
    - CV_FOURCC('M', 'P', 'E', 'G') for MPEG-1 codec

WRITE VIDEO TO FILE

- **This is the constructor of the VideoWriter class. It initializes the object with following parameters:**
  - **const string& filename** - Specify the name and the location of the output file. The video stream is written into this file
  - **int fourcc** - specify the 4 character code for the codec which is used to compress the video. Your computer may not be supported some codecs. So, if you fail to save the video, please try other codecs (see **http://www.fourcc.org/codecs.php**).
    - CV_FOURCC('D', 'I', 'V', '3') for DivX MPEG-4 codec
    - CV_FOURCC('M', 'P', '4', '2') for MPEG-4 codec
    - CV_FOURCC('D', 'I', 'V', 'X') for DivX codec
    - CV_FOURCC('P','I','M','1') for MPEG-1 codec
    - CV_FOURCC('I', '2', '6', '3') for ITU H.263 codec
    - CV_FOURCC('M', 'P', 'E', 'G') for MPEG-1 codec

WRITE VIDEO TO FILE

- For Windows users, it is possible to use -1 instead of the above codecs in order to choose compression method and additional compression parameters from a dialog box. It is a best method for Microsoft Windows users.

- **double fps -** frames per seconds of the video stream. I have used 20. You can try different values. But the codec should support the fps value. So, use an appropriate value.
- **Size frameSize** - Size object which specify the width and the height of each frame of the video stream.
- **bool isColor** - If you want to save a color video, pass the value as true. Otherwise false. Remember codec should support whatever value, you pass. In the above example, you have to give true as the 5th argument. Otherwise it will not work.

FILTERING IMAGES

- Image filtering is an important part of computer vision.
- For most of computer vision applications, filtering should be done before anything else.
- OpenCV supports lots of in-build filtering methods for images.

FILTERING IMAGES

- **Morphological Operations**
  - 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

FILTERING IMAGES

FILTERING IMAGES

- **Eroded**

FILTERING IMAGES

- **cvErode(img, img, 0, 2)**
  - The 1st parameter is the source image.
  - The 2nd parameter is the destination image which is to be the eroded image.
  - Here the 3rd parameter is the structuring element used for erosion. If it is 0, a 3×3 rectangular structuring element is used.
  - The 4th parameter is the number of times, erosion is applied.

FILTERING IMAGES

- **Dilated**

FILTERING IMAGES

- **cvDilate(img, img, 0, 2)**
  - The 1st parameter is the source image.
  - The 2nd parameter is the destination image which is to be the dilated image.
  - Here the 3rd parameter is the structuring element used for dilation. If it is 0, a 3×3 rectangular structuring element is used.
  - The 4th parameter is the number of times, dilatation is applied.

FILTERING IMAGES

- **Inverted**

FILTERING IMAGES

- **cvNot(img, img);**
  - Inverting an image is like taking the negative of an image.
  - This function inverts every bit in every element of the image in the 1st parameter and places the result in the image in the 2nd parameter.
  - This function can process images in place. That means same variable can be used for the 1st and 2nd parameters.
    - For a 8 bit image, the value 0 will be mapped to (255-0)=255

      the value 46 will be mapped to (255-46)=209
    - For a 16 bit image, the value 0 will be mapped to (65535-0)=65535 the value 46 will be mapped to (65535-46)=65489

CHANGE BRIGHTNESS OF IMAGE OR VIDEO

- **Changing brightness is a point operation on each pixel.**
- **If you want to increase the brightness, you have to add some constant value to each and every pixel.**
  - new_img (i, j) = img(i, j) + c
- **If you want to decrease the brightness, you have to subtract some constant value from each and every pixel.**
  - new_img (i, j) = img(i, j) - c

CHANGE BRIGHTNESS OF IMAGE OR VIDEO

12	23	84	122
123	34	92	200
23	45	29	73

CHANGE BRIGHTNESS OF IMAGE OR VIDEO

- **Say, you want to increase the brightness of the image by 20 units. Here is the output image of which the brightness is increased by 20 units.**

12 + 20	23 + 20	84 + 20	122 + 20
123 + 20	34 + 20	92 + 20	200 + 20
23 + 20	45 + 20	29 + 20	73 + 20

=

32	43	104	142
143	54	112	220
43	65	49	93

CHANGE BRIGHTNESS OF IMAGE OR VIDEO

- **Say, you want to decrease the brightness of the image by 20 units. Here is the output image of which the brightness is increased by 20 units.**

12 - 20	23 - 20	84 - 20	122 - 20
123 - 20	34 - 20	92 - 20	200 - 20
23 - 20	45 - 20	29 - 20	73 - 20

=

0	3	64	102
103	14	72	180
3	25	9	53

- **You may already notice that although the 1st pixel of the above image should have (12 - 20) = -8, I have put 0. It is because pixels never have negative values. Any pixel value is bounded below by 0 and bounded above by 2^(bit depth),.**

CHANGE BRIGHTNESS OF IMAGE OR VIDEO

- **Mat imgH = img + Scalar(75, 75, 75);**
  - This line of code adds 75 to each and every pixel in the 3 channels (B, G, R channels) of '**img**'.
  - Then it assigns this new image to 'imgH'.
  - Instead you can use this function also:
    - **img.convertTo(imgH, -1, 1, 75);**
  - **Mat imgL = img + Scalar(-75, -75, -75);**
    - This line of code subtracts 75 from each and every pixel in the 3 channels (B, G, R channels) of '**img**'.
    - Instead you can use this function also:
      - **img.convertTo(imgL, -1, 1, -75);**

CHANGE CONTRAST OF IMAGE OR VIDEO

- Changing the contrast is also a point operation on each pixel.
- The easiest way to increase the contrast of an image is, multiplying each pixel value by a number larger than 1.
  - new_img (i, j) = img(i, j) * c c > 1
- The easiest way to decrease the contrast is, multiplying each pixel value by a number smaller than 1.
  - new_img (i, j) = img(i, j) * c c < 1
- There are more advance methods to adjust contrast of an image such as histogram equalization.
- Such method adjust the contrast of an image such that color distribution is balanced equally.
- We will discuss the histogram equalization in the next lesson.

CHANGE CONTRAST OF IMAGE OR VIDEO

144 * 2	245 * 2	132 * 2	54 * 2
10 * 2	62 * 2	81 * 2	84 * 2
99 * 2	106 * 2	29 * 2	7 * 2

=

255	255	255	108
20	124	162	168
198	212	58	14

- By multiplying each pixel value by 2, you can effectively double the contrast of an image.
- Here is the image of which the contrast is increased.
- I have considered this image as a 8 bit unsigned image.
- So, any pixel value should be from 0 to 255.
- If the resulting image has values more than 255, it should be rounded off to 255.

CHANGE CONTRAST OF IMAGE OR VIDEO

- By multiplying each pixel value by 0.5, you can effectively halve the contrast of an image.
- Here is the image of which contrast is decreased.

144 * 0.5	245 * 0.5	132 * 0.5	54 * 0.5
10 * 0.5	62 * 0.5	81 * 0.5	84 * 0.5
99 * 0.5	106 * 0.5	29 * 0.5	7 * 0.5

=

72	123	66	27
5	31	41	42
50	53	15	4

CHANGE CONTRAST OF IMAGE OR VIDEO

- **void convertTo(OutputArray m, int rtype, double alpha=1, double beta=0)**
  - This OpenCV function converts image into another format with scaling.
  - Scaling is done according to the following formula:
    - m[i,j] = alfa * img[i,j] + beta

- Parameters:
  - **OutputArray m -** Store the converted image
  - **int rtype -** Depth of the output image. If the value of **rtype** is negative, output type is same as the input image. Possible inputs to this parameter:
    - CV_8U
    - CV_32S
    - CV_64F
  - **double alpha -** Multiplication factor; Every pixel will be multiplied by this value
  - **double beta -** This value will be added to very pixels after multiplying with the above value.

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- Histogram is the intensity distribution of an image.
- For example consider the following image of 2 bits (depth)

2	3	2	1	2
1	2	0	2	3
0	1	1	3	1
2	3	0	1	2
0	1	2	2	0

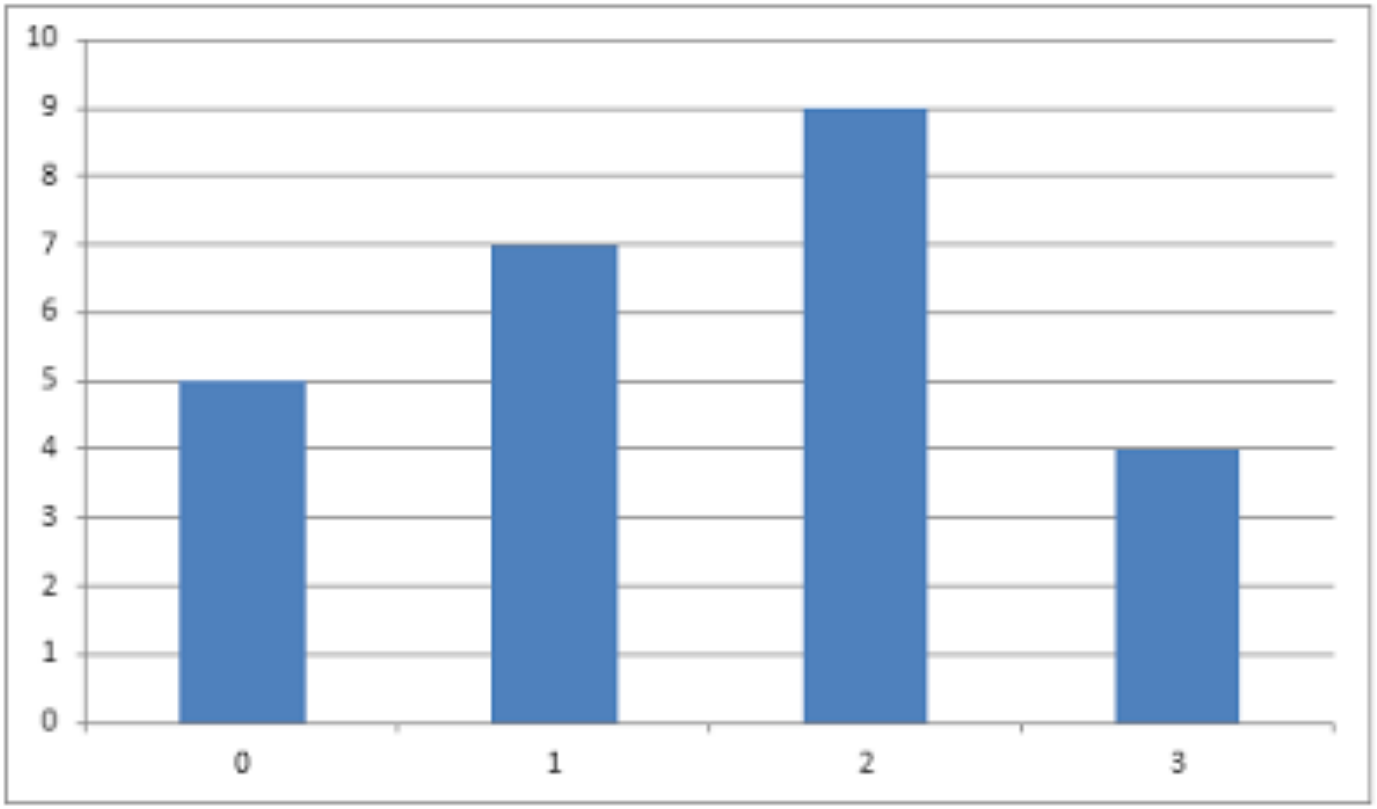
HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- Histogram of the a image shows how the pixel values are distributed.
- As you can see in the above image there are 5 pixels with value 0,
- 7 pixels with value 1,
- 9 pixels with value 2
- and 4 pixels with value 3.

Pixel Value	Number of Pixels
0	5
1	7
2	9
3	4

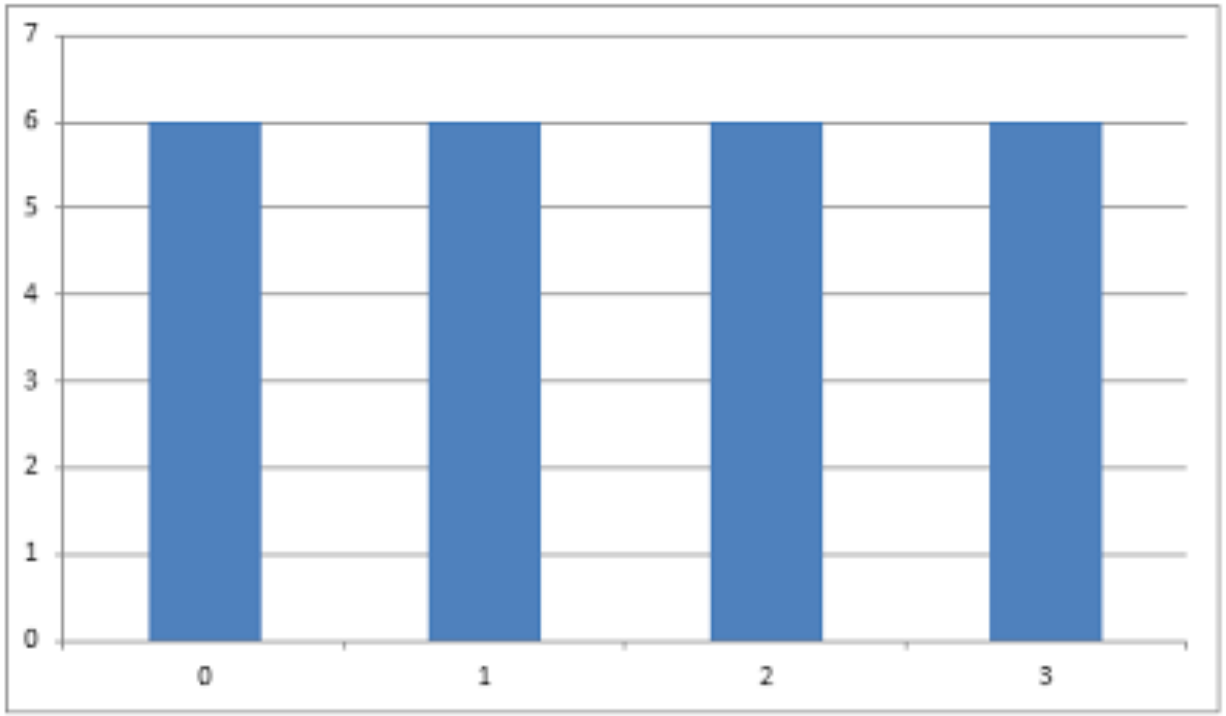
HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- Histogram of a image usually presented as a graph.
- The following graph represents the histogram of the previous image.



HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- Histogram Equalization is defined as equalizing the intensity distribution of an image or flattening (nivelar) the intensity distribution curve.
- Histogram equalization is used to improve the contrast of an image.
- The equalized histogram of the previous image should be ideally like the following graph.



HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- But practically, you cannot achieve this kind of perfect histogram equalization.
- But there are various techniques to achieve histogram equalization close to the perfect one.

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **void cvtColor(InputArray src, OutputArray dst, int code, int dstCn=0).**
  - This function converts image from one color space to another color space.
  - OpenCV usually loads an image in BGR color space.
  - In the code example, I want to change the image to grayscale color space.
  - So, I use the **CV_BGR2GRAY** as the 3rd parameter.
  - If you want to convert to HSV color space, you should use **CV_BGR2HSV.**

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **void cvtColor(InputArray src, OutputArray dst, int code, int dstCn=0).**
  - **InputArray src**- Input image (it should be 8 bit unsigned or 16 bit unsigned or 32 bit floating point image)
  - **OutputArray dst** - Output image (It should have a same size and depth as the source image)
  - **int code**- Should specify the color space conversion. There are many codes available. Here are some of them:
    - CV_BGR2HSV
    - CV_HSV2BGR
    - CV_RGB2HLS
    - CV_HLS2RGB
    - CV_BGR2GRAY
    - CV_GRAY2BGR
  - **int dstCn -** Number of channels in the destination image. If it is 0, number of channels of the destination image is automatically derived from source image and color space conversion code. For a beginner, it is recommended to use 0 for this parameter.

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **void equalizeHist(InputArray src, OutputArray dst)**
  - This function equalizes the histogram of a single channel image (Grayscale image is a single channel image)
  - By equalizing the histogram, the brightness is normalized.
  - As a result, the contrast is improved.
    - **InputArray src** - 8 bit single channel image
    - **OutputArray dst -** Destination image of which histogram is equalized (It should have the same size and depth as the source image.)

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **cvtColor(img, img_hist_equalized, CV_BGR2YCrCb)**
  - This line converts the color space of BGR in '**img**' to YCrCb color space and stores the resulting image in '**img_hist_equalized**'.
  - I am going to equalize the histogram of color images.
  - In this scenario, I have to equalize the histogram of the intensity component only, not the color components.
  - So, BGR format cannot be used because its all three planes represent color components blue, green and red.
  - So, I have to convert the original BGR color space to YCrCb color space because its 1st plane represents the intensity of the image where as other planes represent the color components.

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **void split(const Mat& m, vector<Mat>& mv)**
  - This function splits each channel of the '**m**' multi-channel array into separate channels and stores them in a vector, referenced by '**mv**'.
    - **const Mat& m -** Input multi-channel array
    - **vector<Mat>& mv -** vector that stores the each channel of the input array

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **equalizeHist(channels[0], channels[0]);**
  - Here we are only interested in the 1st channel (Y) because it represents the intensity information whereas other two channels (Cr and Cb) represent color components.
  - So, we equalize the histogram of the 1st channel using OpenCV in-built function, '**equalizeHist**(..)' and other two channels remain unchanged.

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **void merge(const vector<Mat>& mv, OutputArray dst)**
  - This function does the reverse operation of the split function.
  - It takes the vector of channels and create a single multi-channel array.
    - **const vector<Mat>& mv** - vector that holds several channels. All channels should have same size and same depths
    - **OutputArray dst** - stores the destination multi-channel array

HISTOGRAM EQUALIZATION OF GRAYSCALE OR COLOR IMAGE

- **cvtColor(img_hist_equalized, img_hist_equalized, CV_YCrCb2BGR)**
  - This line converts the image from YCrCb color space to BGR color space.
  - It is essential to convert to BGR color space because '**imshow**(..)' OpenCV function can only show images with that color space.

SMOOTH / BLUR IMAGES

- Sometimes it is also called blurring
- The main objective of smoothing is to reduce noise
- Such noise reduction is a typical image pre-processing method which will improve the final result.
- There are various ways to smooth or blur an image.
- Smoothing is done by sliding a window (kernel or filter) across the whole image and calculating each pixel a value based on the value of the kernel and the value of overlapping pixels of original image.
- This process is mathematically called as convolving an image with some kernel.
- The kernel is the only difference in all of the types of smoothing (Blurring) methods.

SMOOTH / BLUR IMAGES

- **For Example, 5 x 5 kernel used in homogeneous smoothing (blurring) is as below.**
- **This kernel is known as "Normalized box filter".**

$$\frac{1}{25}$$

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

SMOOTH / BLUR IMAGES

- **For Example, 5 x 5 kernel used in Gaussian smoothing (blurring) is as below.**
- **This kernel is known as "Gaussian kernel".**

$$\frac{1}{273}$$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

SMOOTH / BLUR IMAGES

- **Some important facts about smoothing kernels (filters):**
  - Number of rows and number of columns of a kernel should be odd (e.g. - 3x3, 11x5, 7x7, etc).
  - When the size of the kernel is getting larger, processing time also becomes larger.

HOMOGENEOUS SMOOTHING

- Is also called as "Homogeneous Blurring", "Homogeneous Filtering" or "Box Blurring".
- This is the most simplest method of smoothing an image.
- It takes simply the average of the neighborhood of a pixel and assign that value to itself.
- You have to choose right size of the kernel.
- If it is too large, small features of the image may be disappeared and image will look blurred.
- If it is too small, you cannot eliminate noises of the image.

GAUSSIAN SMOOTHING

- is also called as "Gaussian Blurring" or "Gaussian Filtering".
- http://en.wikipedia.org/wiki/Gaussian_blur

MEDIAN SMOOTHING

- is also called as "Median Blurring" or "Median Filtering".
- The input image is convolved with a Median kernel
- Detection algorithms because under certain conditions, it preserves edges while removing noise.
- http://en.wikipedia.org/wiki/Median_filter

BILATERAL SMOOTHING

- is also called as "Bilateral Blurring" or "Bilateral Filtering".
- This is the most advanced filter to smooth an image and reduce noise
- All of the above filters will smooth away the edges while removing noises.
- But this filter is able to reduce noise of the image while preserving the edges.
- The drawback of this type of filter is that it takes longer time to process.
- http://en.wikipedia.org/wiki/Bilateral_filter

HOW TO ADD TRACKBAR

- Trackbars are very useful in lots of occasions.
- It enables users to change various parameters while the OpenCV application is running.
- Whenever you change the position of a trackbar, the value of an integer variable is changed.
- Using that value, we can change a property of an image or a video.
- The following example will show you how to do it with OpenCV.

HOW TO ADD TRACKBAR

- **OpenCV Example of How to Change Brightness and Contrast of an Image with Trackbars:**
  - In the following example, I have added two trackbars to change the brightness and contrast of an image.
  - It is iterating in a infinite while loop and applying the brightness and contrast to the image periodically because I want to apply the changes to the image whenever the user changes the position of the trackbar.

HOW TO ADD TRACKBAR

- **int createTrackbar(const string& trackbarname, const string& winname, int* value, int count, TrackbarCallback onChange = 0, void* userdata = 0)**
  - **trackbarname -** The name of the trackbar
  - **winname** - The name of the window to which the trackbar is attached
  - **value** - This integer, pointed by this pointer, holds the value associated with the position of the trackbar
  - **count** - The maximum value of the trackbar. The minimum value is always zero.
  - **onChange** - This function will be called everytime the position of the trackbar is changed. The prototype of this function should be "**FunctionName(int, void*)**". The "**int**" value is the value associate with the position of the trackbar. And "**void***" is any pointer value which you pass as the "**userdata**" (See the next parameter).
  - **userdata** - This pointer variable will be passed as the second parameter of the above function

COLOR DETECTION & OBJECT TRACKING

- Object detection and segmentation is the most important and challenging fundamental task of computer vision.
- It is a critical part in many applications such as image search, image auto-annotation and scene understanding.
- However it is still an open problem due to the complexity of object classes and images.
- The easiest way to detect and segment an object from an image is the color based methods .
- The colors in the object and the background should have a significant color difference in order to segment objects successfully using color based methods.

SIMPLE EXAMPLE OF DETECTING RED OBJECTS

- OpenCV usually captures images and videos in 8-bit, unsigned integer, BGR format.
- In other words, captured images can be considered as 3 matrices, BLUE,RED and GREEN with integer values ranges from 0 to 255.

Blue matrix

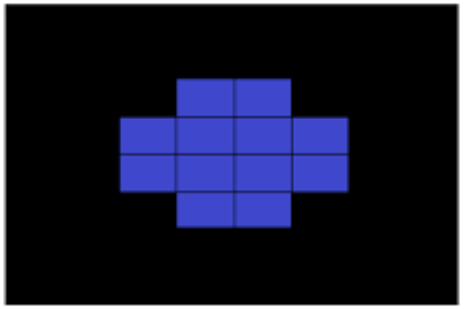
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	255	255	0	0	0
0	0	255	255	255	255	0	0
0	0	255	255	255	255	0	0
0	0	0	255	255	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

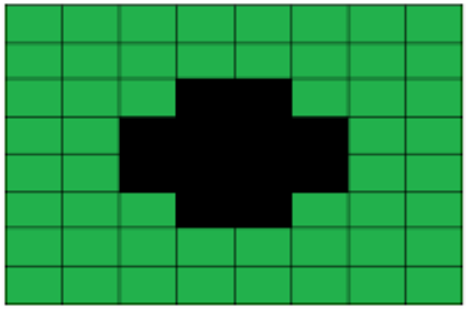
Green matrix

255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	0	0	255	255	255
255	255	0	0	0	0	255	255
255	255	0	0	0	0	255	255
255	255	255	0	0	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

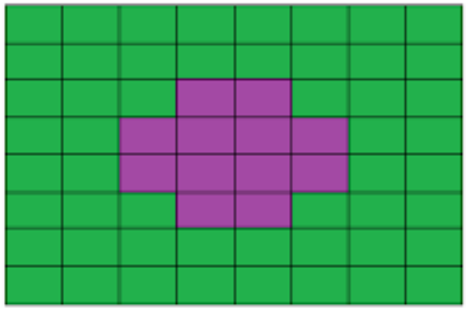
Red matrix

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	255	255	0	0	0
0	0	255	255	255	255	0	0
0	0	255	255	255	255	0	0
0	0	0	255	255	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0





Resultant Image



SIMPLE EXAMPLE OF DETECTING RED OBJECTS

- In the previous application, I have considered that the red object has 'hue', 'saturation' and 'value' in between 170-180, 160-255, 60-255 respectively.
- Here the 'hue' is unique for that specific color distribution of that object.
- But 'saturation' and 'value' may be vary according to the lighting condition of that environment.

SIMPLE EXAMPLE OF DETECTING RED OBJECTS

- **Hue values of basic colors:**
  - Orange 0-22
  - Yellow 22- 38
  - Green 38-75
  - Blue 75-130
  - Violet 130-160
  - Red 160-179

  - There are a lot of programs that can calculate HSV values of images.
  - These are approximate values!!!

SIMPLE EXAMPLE OF DETECTING RED OBJECTS

- You have to find the exact range of 'hue' values according to the color of the object.
- I found that the range of 170-179 is perfect for the range of hue values of my object.
- The 'saturation' and 'value' is depend on the lighting condition of the environment as well as the surface of the object.
- How to find the exact range of 'hue', 'saturation' and 'value' for a object is discussed later

SIMPLE EXAMPLE OF DETECTING RED OBJECTS

- **cvInRangeS(const CvArr* *src*, CvScalar *lower*, CvScalar *upper*, CvArr* *dst*)**
  - Checks that each array element of 'src' lies between 'lower' and 'upper'.
  - If so, array element in the relevant location of 'dst' is assigned '255' , otherwise '0'.
    - const CvArr* *src - source array which is the image*
    - CvScalar *lower - inclusive lower bound (In the above application, it is cvScalar(170,160,60) because my lower bound for 'hue','saturation' and 'value' is 170,160 and 60 respectively)*
    - CvScalar *upper - exclusive upper bound (In the above application, it is cvScalar(180,256,256) because my upper bound for 'hue','saturation' and 'value' is 179,255 and 255 respectively)*
    - CvArr* *dst - destination array which is the binary image* (must have 8-bit unsigned integer or 8-bit signed integer type)

SHAPE DETECTION

- In this lesson, let's see how to identify a shape and position of an object using contours with OpenCV.
- Using contours with OpenCV, you can get a sequence of points of vertices of each white patch (White patches are considered as polygons).
- As example, you will get 3 points (vertices) for a triangle, and 4 points for quadrilaterals.
- So, you can identify any polygon by the number of vertices of that polygon.
- You can even identify features of polygons such as convexity, concavity, equilateral and etc by calculating and comparing distances between vertices.

SHAPE DETECTION

- In this lesson, let's see how to identify a shape and position of an object using contours with OpenCV.
- Using contours with OpenCV, you can get a sequence of points of vertices of each white patch (White patches are considered as polygons).
- As example, you will get 3 points (vertices) for a triangle, and 4 points for quadrilaterals.
- So, you can identify any polygon by the number of vertices of that polygon.
- You can even identify features of polygons such as convexity, concavity, equilateral and etc by calculating and comparing distances between vertices.