Bare Demo of IEEEtran.cls for IEEE Journals

***Abstract*—The abstract goes here.**

***Index Terms*—IEEE, IEEEtran, journal, LATEX, paper, template.**

I. Introduction

THIS demo file is intended to serve as a "starter file" for IEEE journal papers produced under LATEX using IEEEtran.cls version 1.8b and later. I wish you the best of success.

mds
August 26, 2015

II. Tutorial Description

In this section it will be presented the experiences done in the class by following the open cv tutorials. It will be divided in 8 subsections and for each subsection it will be analyzed not only the images displayed, but also the the important parts of the code of the tutorial.

A. Load and Display an Image

Images in OpenCV are of the cv::Mat type, explored later in II-C, to read an image the cv::imread function should be used, cv::imread reads image given a name and a flag that specifies how the image should be read and what eventual conversions to make, the full list of accepted flags can be accessed here here.

cv::namedWindow is then used to display our image, it needs a windowName and a flag to specify if and how the user can resize the image, the windowName is used to refer to the image in other functions and the full flag list can be seen here, the main flags are:

- WINDOW_NORMAL or WINDOW_AUTOSIZE indicates whether we can adjust the window or not, WINDOW_AUTOSIZE adjusts automatically the window size.
- WINDOW_FREERATIO or WINDOW_KEEPRATIO indicates whether the image ratio is maintained through size adjustments.

these flags can be joined with the operator "|". cv::imshow is used to update the content of a window given it's window name and an image to update for the image to be displayed continuously we should use cv::waitKey, it specifies how many milliseconds it should wait for user input before disappearing. 0 means to wait forever. image 1 shows an example of a window

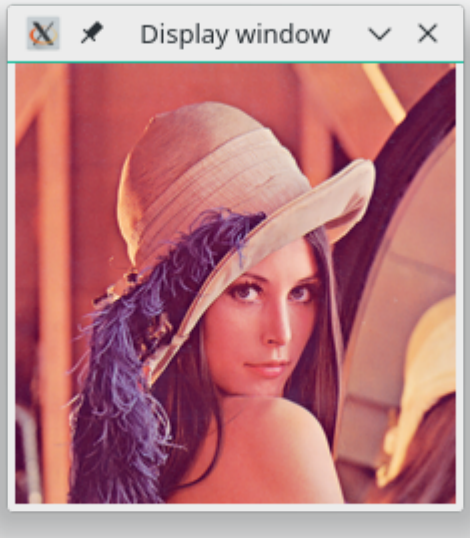

Fig. 1. OpenCV Window

B. Load, Modify and Save an Image

In this section the main goal was to learn how it is to modify and save an image. To modify an image the first thing that it is necessary to do is load the image. Using the code from the tutorial it is possible to choose the image by putting the path as argument when we run the program. Then, the next thing that it is necessary to do is load the image is use the cv::imread. This function will load the image to the object Mat. The next step is to create another object of type Mat to save there the image modify and then it is used the function cv::cvtColor to convert the image to grayscale. This function accepts 4 arguments. The first argument is the input image, the second one is output image, the third is the code to choose the modification that it will be performed and then the last one is the number of channels in the destination image. The last argument is not necessary to use because it has already a predefined value that is 0. By using the value 0 the number of the channels is derived automatically from src and code. After this steps if we want to save the image on disk to not lose the image at the end of the program. it is possible to use a function to do that. The function is cv::imwrite. To use this function we must put the in the first argument the path and the name of the file where we want to save the image, the second argument is the image object that we want to save and in this

case it is the object where we saved the version of the image in grayscale. This function also can accept another argument that is "Format-specific parameters encoded as pairs", however it has a predefined value so it is not necessary to pass as argument in this case. Finally, the only thing that is missing is create the windows to display there the original and the modified image. To do that it is used two functions. The first function is cv::namedWindow to create the window to display the images. This function takes two arguments. The first argument is to chose the name of the window and the other argument is to chose the size of it. The other function is cv::imshow and this function also takes two arguments. The first argument is the name of the window where we want to display the image and the second argument is the object that contains the image that we want do display.

In the image **??** it is displayed in the created window the original image that we used in this part of the tutorial and in the image **??** it is displayed the image of the created window to display the modified image.

C. Mat - The Basic Image Container

cv::Mat is a class with both the matrix data and the relevant headers, each Mat has it's own header but the matrix data can be shared between 2 instances, normal copies and assignments (Mat A;A = B; or Mat B(A);) only copy the header, if copying the matrix is really needed OpenCV provides cv::Mat::clone and cv::Mat::copyTo.

We can also easily access parts of an image like such: cv::Mat D (A, cv::Rect_<int>(10, 10, 100, 100)); to create a Mat object explicitly we need to provide a data type for storing elements, The naming convention is as follows: "CV_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The channel number], so CV_8UC3 means we use 8 bit long unsigned char types and each pixel has three of these (one for each channel). One possible declaration could be cv::Mat M(2,2, CV_8UC3 , cv::Scalar(0,0,255)); in which we specify the row size, the column size, the data type and the data (each 3 channel pixel is (0,0,255)) other ways of initializing matrices:

- Mat E = Mat::eye(4, 4, CV_64F), creates a diagonal matrix
- Mat R = Mat::ones(2, 2, CV_32F)
- Mat Z = Mat::zeros(3,3, CV_8UC1)
- Mat C = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0), allows us to specify each item individually
- Mat RowClone = C.row(1).clone(), we can also create a Mat from a subsection of a matrix
- randu(R, Scalar::all(0), Scalar::all(255)), can be used to create a random matrix

D. How to scan images

The goal of this section is to know how to scan an image and in this tutorial it is necessary to scan the image to do a simple color reduction method. This method consists in passing through every pixel of an image matrix and applying one specific formula. These specific formula has multiplications and divisions, therefore it is computationally heavy run this for each pixel. The solution for this problem is calculate all possible values beforehand and during the assignment just make the assignment, by using a lookup table. A lookup table is basically an array where the result of the operations are saved. OpenCV has three major methods to go by each pixel of an image, however we will test four and we will test each method to see how long it took. To measure the time it will be used the functions cv::getTickCount() and cv::getTickFrequency() and by getting the value of this functions it is possible to calculate the time. Also, it is important to know how a image matrix is stored in memory like we see in the previous section. The methods used are:

1) Efficient Way
2) The iterator (safe) method
3) On-the-fly address calculation with reference returning
4) The Core Function

To use the efficient way it is necessary to use pointers and acquire a pointer to the start of each row and go through it until it ends. One things that it must be considerate is that in color images have three channels so it is necessary to pass through three times more items in each row.

The iterator method is considered a safer way as it takes over some tasks from the user and the user only needs to ask the begin and the end of the image matrix and then just increase the begin iterator until you reach the end.

The method on-the-fly address calculation with reference returning consists in specify the row and column number of the item you want to access and for that reason this method isn't recommended for scanning.

The last method uses a function from OpenCV for modifying image values and by using this function it is not necessary to write the scanning logic of the image. The function that it is used is cv::LUT(). This function takes three arguments. The first argument is the input image, the second the lookup table and the third one is the output image.

The last part was to test all the methods to compare the performance time. The used images was the same that it was used in the section "Load, Modify and Save an Image". The results are presented in the next list.

1) Efficient Way - 2.51 ms
2) The iterator (safe) method - 9.65 ms
3) On-the-fly address calculation with reference returning - 10.24 ms
4) The Core Function - 0.23 ms

The results are the expected by comparing to the tutorial values. The most efficient way is using the core function and that happens because OpenCV library is multi-thread. The second most efficient method is the method that uses pointers. The third one is the iterator method and this is substantial worst comparing to the others two and lastly comes the method on the fly address calculation.

In conclusion, the core function is definitely the most efficient, however if it is only to process few images it is viable to choose any method, but if it is to process a lot of images is recommended to use the core function not only because it is much faster but also because it is the easiest one to use.

E. Mask operations

Masks, or Kernels are most efficiently applied with the cv::filter2D function, it needs an input image and a kernel to calculate the output image, we can specify the kernel using one of the methods discussed in II-C, for example : Mat kern = (Mat_<char>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0), and is applied as such (I being the input image and K the output): cv::filter2D(I, K, I.depth(), kern). The above kernel has a sharpening effect, it's not very noticible but can be seen in 3

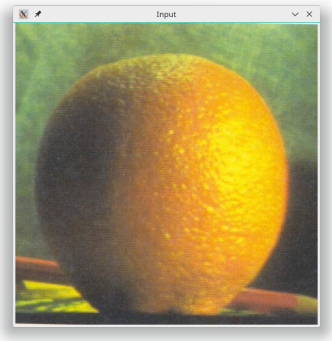

Fig. 2. Input image

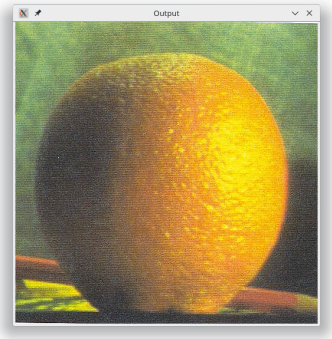

Fig. 3. Output Image

F. Operations with images

This section is to learn how to do operations in images. To do an operation in an image the first thing that it is necessary to do is load the image. Like it was said in one of the previous sections, to load an image it is used the function cv::imread. It can take two arguments. The first argument is the path for the file and the second it is used for example if you want to change the number of channels that you want to load.

The next step was to learn how it is possible to access the intensity values of each pixel. By using the function "at" from the image object it is possible to access. If the image is in grayscale it will be received a scalar, however if the image has 3 channel image with BGR color ordering, it will be received a vector and then it is possible to access the value of each channel.

In the third part of this section it is explained due to the fact that Mat is a structure that keeps matrix/image characteristics and a pointer to data, nothing prevents us from having several instances of Mat corresponding to the same data. If the goal is copy the matrix it is necessary to use the functions cv::Mat::copyTo or cv::Mat::clone.

In the last part of this section it is explained how it is possible to see intermediate results of your algorithm during development process. This is already explained previously how it can be made by using cd::imshow function. Another thing that we learn was how the function cv::waitKey can be used. This function is used to wait for an input from the keyboard to close the window.

G. Adding (blending) two images

The blending of two images is described by the following formula ($f_n(x)$ describes the pixels across channels, alpha is how much one image contributes to each pixel)

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

$$\alpha \in [0, 1]$$

and is preformed by cv::addWeighted as such: cv::addWeighted(src1, alpha, src2, 1-alpha, gamma, dst) The effect can be seen in 6, where an alpha of 0.5 is used

Fig. 4. Source image 1

Fig. 5. Source image 2

H. Changing the contrast and brightness

The last part of the tutorial the goal was to learn how it is possible to change the contrast and the brightness of an image. Like it was already done in previous sections the first thing that it is necessary to process or manipulate an image is load the image and then save it in a Mat object. The next thing that it is needed it is a new Mat object to store some transformations of the image. To accomplish that it is created an matrix of zeros of the same size of the original image. The last step is run all the pixel of the images and each color

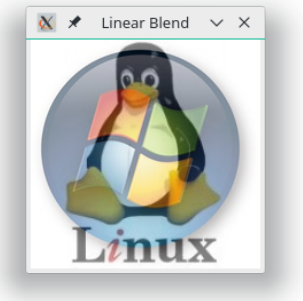

Fig. 6. Output image

channel and perform the function to change the brightness and the contrast using the values defined by the user and save the result in the matrix created with zeros. After this, it is showed the original image and the image that was created. To test the program it was used the same image from the university. The value that was chosen for alha was 2.2 and the beta value was 50. The result can be seen in **??** and be compared with the original in **??**.