COSC342: OpenCV Introduction

Working with Images, Basic Structures, Filtering

Objectives

- Use CMake to create either XCode project file or Makefile
- Open OpenCV to read and display an image
- Access individual pixel values in OpenCV
- Apply image filters in OpenCV
- Save the resulting images to files in different formats.

1 Getting Started

The following tasks will give you a brief introduction to the OpenCV library. OpenCV is widely used for image processing and computer vision, and is freely available for a range of platforms from `blah`

Make sure that you copy the source files from the `pickup` directory into a directory within your home directory. Otherwise you may not be able to compile the code.

You can do this from Finder, or open a terminal window and enter the following commands

```
cd ~/Documents/cosc342
cp /home/cshome/coursework/342/pickup/lab02a-OpenCV * .
```

There is a CMake project in the directory you just copied, which you can build in the same way as the `Matrices` code from the last lab assignment. If you look inside `CMakeLists.txt` there are a few new things:

```
cmake_minimum_required(VERSION 2.8)
project(openCVFilters)

set (OpenCV_DIR /home/cshome/coursework/342/pickup/OpenCV3/build)

find_package( OpenCV REQUIRED )

add_executable( openCVFilters openCVfilters.cpp )
target_link_libraries( openCVFilters ${OpenCV_LIBS} )
```

Line 4 says where the OpenCV libraries are (note, you do *not* need to copy these or anything, just use them where they are). Line 6 searches for the OpenCV libraries and will throw an error if they can't be found. Finally, line 9 links the program against the OpenCV libraries.

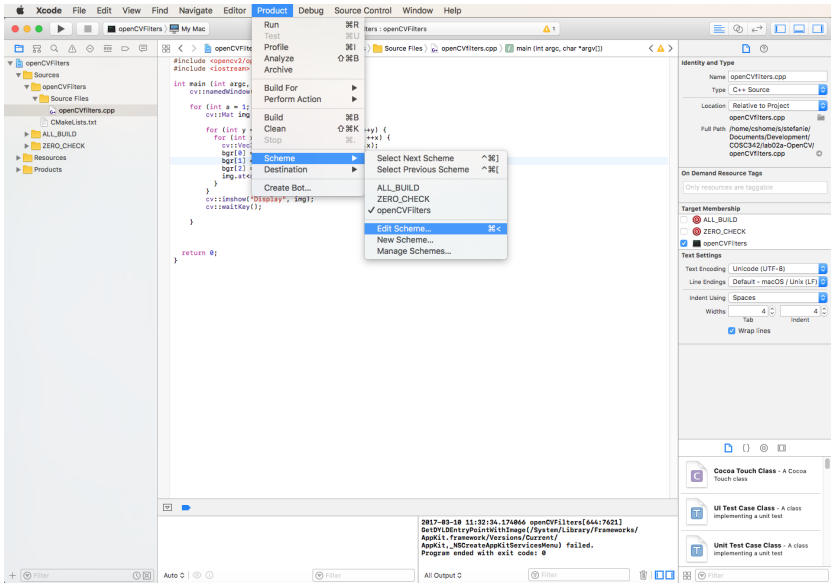
You should be able to build the program `openCVfilters`, but if you run it nothing will happen. To see what's going on (or not), lets take a look inside the source code:

```
1  #include <opencv2/opencv.hpp>
   #include <iostream>
3
   int main (int argc, char *argv[]) {
5      cv::namedWindow("Display");
7      for (int a = 1; a < argc; ++a) {
           cv::Mat img = cv::imread(argv[a]);
9
           for (int y = 0; y < img.size().height; ++y) {
11           for (int x = 0; x < img.size().width; ++x) {
               cv::Vec3b bgr = img.at<cv::Vec3b>(y,x);
13             bgr[0] = 0.5*bgr[0];
               bgr[1] = 0.5*bgr[1];
15             bgr[2] = 0.5*bgr[2];
               img.at<cv::Vec3b>(y,x) = bgr;
17           }
           }
19         cv::imshow("Display", img);
           cv::waitKey();
21
       }
23
     return 0;
25 }
```

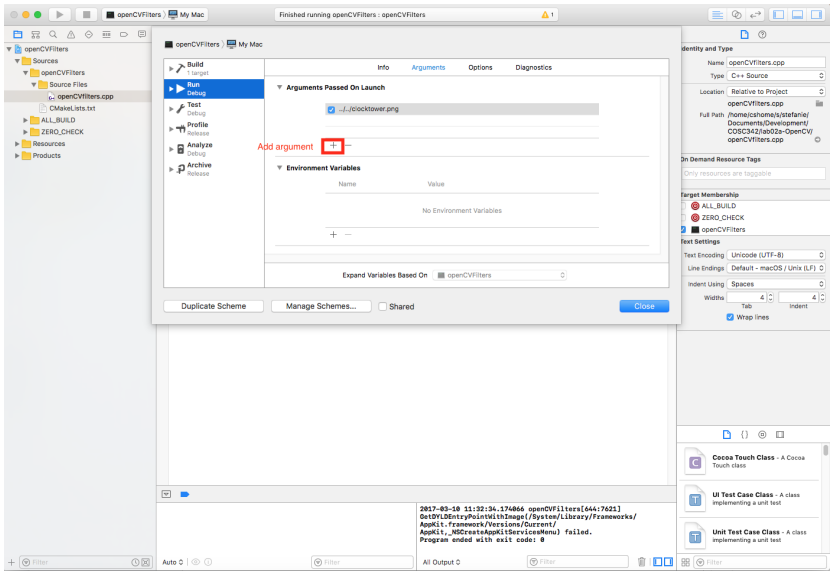
The loop inside `main` on line 7 iterates over the values passed to the command line, and the program is expecting a list of image files. A picture of the clocktower building is included in the directory you've copied from `pickup`, so if you are using Makefiles you can run

```
1 ./openCVfilters ../clocktower.png
```

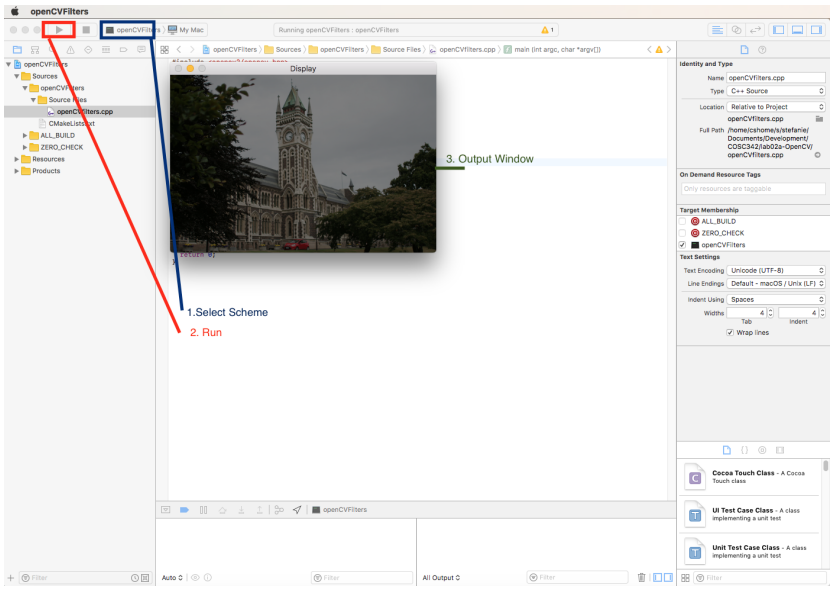
If you are using XCode you'll need to set the command line parameters by using "Edit Scheme" in the XCode "Product" menu.



From the popup window select the "Arguments" tab and use the + button to pass an argument to your application. Since the application will run from the debug folder in XCode if we do not set a special working directory, we have to move two directories up in order to access the clocktower.png image. Therefore, we will type *../../clocktower.png*.



Again, to run the project, select the scheme you want to use from the pull-down menu. In this case it is called "openCVFilters". Then, pess the run button to run the application.



Now when you run the program it should open up a window with a picture of the clock tower, although it will be a bit dim. This might be hidden under other windows, so clear them away from the top-left of the screen or press `command-tab` to switch to the `openCVfilters` program (it's icon is a grey box with `exec` in the top corner. If you make this the active window, and press any key, it should close.

2 OpenCV

Let's take a look through the code listing piece by piece.

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

We start by importing a couple of useful libraries. The important one here is OpenCV, and CMake has taken care of telling the build system (Make or XCode) where to find the relevant files.

```
int main (int argc, char *argv[]) {
    cv::namedWindow("Display");
```

Next we have the main function, which takes `argc` command line arguments, which are stored in the array `argv`. Note that `argc` is at least 1, since `argv[0]` is the name of the program being invoked. The first thing the program does is to create an OpenCV window called `Display`. This window is where we'll be showing the images.

```
for (int a = 1; a < argc; ++a) {
    cv::Mat img = cv::imread(argv[a]);
```

The main loop goes over the command line arguments, which should be image file names which are read in. the function `cv::imread()` reads an image and creates a `cv::Mat` object as the result. There are a few things to note here:

- OpenCV puts its functions etc. in the `cv` namespace. This avoids collisions with other libraries which might use the same names.
- The image is read in to a `cv::Mat`, which is used for matrices (which it gets its name from) as well as images.
- Because a `cv::Mat` could be an image or a matrix or whatever, it could contain floating point numbers, integers, RGB-triples, or a number of other things. We'll see how this works soon.

```
  for (int y = 0; y < img.size().height; ++y) {
2     for (int x = 0; x < img.size().width; ++x) {
```

Once we've loaded our image, we loop over its pixels. The `cv::Mat` object has a `size()` method that returns its width and height. These are fairly straightforward loops, but note that we loop over `y` then `x`. This doesn't matter too much, but images and matrices in OpenCV are stored in *row-major* order. This means that they are laid out in memory with the first row first, then the next row, and so on. Because of this it is better to loop over the rows (the `y` co-ordinate) first, and then over each column (`x` co-ordinate) within the rows. This means we access the memory sequentially rather than jumping around, which is easier to mange efficiently.

```
  cv::Vec3b bgr = img.at<cv::Vec3b>(y,x);
```

Now things get a little tricky. We want to access the pixel at co-ordinates (x, y), and the method for this is called `at`. Since a `cv::Mat` can be used for a matrix or an image, a choice has to be made between image (x then y) co-ordinates or matrix (row then column, or y then x) indexing. The choice in OpenCV is matrix indexing, which is why the line ends with `(y,x)`.

Next, we're accessing a colour image, so each pixel value is three bytes (by default). OpenCV represents this with a structure called a `Vec3b` (a *vec*tor of *3 b*ytes). This is what the `at` method, returns, but we also need to be able to call `at` on matrices or images different types, such as a double (for a normal Matrix) or an unsigned short (for a 16-bit greyscale image), or perhaps a vector of 3 floats (for a floating point colour image representation).

The way C++ deals with this is through the use of *templates*. The type of value that is being accessed is given in angle brackets, and this creates a version of `at` at compile time that is specialised for accessing 24-bit colour images. If we had a matrix, `M`, of doubles, we'd access its values as `M.at<double>(y,x)`.

Finally the result is a red, green, and blue value, but I have called it `bgr`. This is because OpenCV (like Windows bitmap files) stores the values in blue-green-red order.

```
1   bgr[0] = 0.5*bgr[0];
      bgr[1] = 0.5*bgr[1];
```

```
3     bgr[2] = 0.5*bgr[2];
```

Next we multiply the blue, green, and red values in the `Vec3b` by half. Note that this does not change the image. When we assigned the `Vec3b` to the variable `bgr` we took a deep copy of the object, unlike Java where references (i.e. pointers) are used for everything but basic types, so shallow copies are the norm.

```
1     img.at<cv::Vec3b>(y,x) = bgr;
```

To get the new pixel value back into the image, we use `at` again. Just like in the accessors in the `Matrix` class from last lecture, `at` returns a reference to the pixel value, so we can assign to it.

```
1   cv::imshow("Display", img);
      cv::waitKey();
```

Finally we show the image in the window we created earlier, and wait for a key. `waitKey()` doesn't just wait for a key to be pressed, it also triggers OpenCV's other event handling routines, including re-drawing the windows. If you don't call `waitKey()`, you won't see the image.

If you just want to refresh the image (or check for mouse or keyboard events), you can call `waitKey()` with an optional parameter which is the number of milliseconds to wait, so

```
    cv::waitKey(10);
```

will wait for 1/100th of a second.

Exercise

Since images and matrices are the same sort of thing on OpenCV, we can often treat them interchangeably. In particular, we can multiply an image by a scalar. This means that instead of getting each pixel, halving each of its colour channels, and then putting it back into the image storage we can just write

```
1   img = 0.5*img;
```

Try this approach and confirm that you get the same result.

3 Filtering in OpenCV

OpenCV implements many algorithms and methods, including the image filters that we discussed in lectures.

A 5×5 mean filter, for example, is done with

```
cv::Mat blurred;
cv::blur(img, blurred, cv::Size(5,5));
```

, while a 9×9 Gaussian filter with $\sigma = 2$ is

```
cv::GaussianBlur(img, blurred, cv::Size(9,9), 2);
```

,

Exercise

Try out the mean and Gaussian blur filters, and see what effect changes to their parameters has.

The mean and Gaussian filters are fairly straightforward, because they return values in the normal pixel ranges. The Sobel filters we looked at in lectures for edge detection are not so simple. Recall that the kernels for these filters look like this:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & -2 \\ -1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

These filters could give negative values, or values much greater than 255. Either way, they won't fit into a normal 8-bit greyscale or 24-bit colour image.

The OpenCV function for Sobel filters takes this into account, by asking what sort of output you want. The result of the Sobel filter on an image with values in the range $[0 \ldots 255]$ is going to be an integer, but could be as small as -1020, or as large as +1020 (imagine 255 under the negative/positive values and zeroes everywhere else).

These values won't all fit into a byte, but they will fit into a 16-bit signed integer. OpenCV uses a shorthand for these types which is `CV_` followed the number of bits then one of:

- `U` for unsigned integer values,
- `S` for signed integer values, or
- `F` for floating point values.

For example, 8-bit values are denoted by `CV_8U`, while doubles are typically 64-bit, so `CV_64F`. Sometimes we'll have a number of channels after the type, introduced by `C`, so `CV_8UC3` is a 3-channel image with 8 bits for each channel. This is a typical 24-bit colour image (8 bits for Red, Green, and Blue).

Back to the Sobel filters, the OpenCV function takes the following parameters:

- The input image
- The destination image
- The type of the destination (e.g.: `CV_16S` or `CV_32F`)

- The order of the derivative to compute in x – 1 for horizontal edges, 0 otherwise
- The order of the derivative to compute in y – 1 for vertical edges, 0 otherwise

Exercise

Try computing the vertical and horizontal Sobel filters, and displaying the results.

You should see very faint edges, but nothing really clear. The problem is that the result has values in the range [-1020,1020], which is too big to display as a normal 8-bit values. We need to convert the result to an image where the values are bytes, or `CV-8U`. Suppose we have an image, `edges`, with the result of the Sobel filter stored as `CV_16S` values. We can convert this to an 8-bit image for display using

```
cv::Mat display;
edges.convertTo(display, CV_8U, a, b);
```

where `a` and `b` allow you to scale and shift the input values, v, to give the output values v' by the linear equation

$$v' = av + b.$$

The values we want are roughly $a = \frac{1}{8}$ and $b = 127$. This creates a new `cv::Mat` image, called `display`, which we can then show in the window.

Exercise

Convert the results of the Sobel filters to a format better for display and show them on screen. The result of the horizontal filter should look something like this:

Question: Why are the values of a and b suggested above useful ones to try? How could you determine these if you didn't know what filter had been applied?

Finally (for now at least) OpenCV has a general filtering function, `cv::filter2D` where you can provide your own filter kernel. It takes as parameters

- The input image,
- The destination image,
- The type of the destination image (`CV_32F` etc.),
- The kernel that you wish to convolve with.

The kernel itself is a matrix of numbers, so you need to create a new `cv::Mat` with the appropriate size and type. Suppose you wanted to make a 3×3 mean filter, you could do so with the following code:

```
cv::Mat kernel(cv::Size(3,3), CV_32F);
for (int row = 0; row < 3; ++row) {
  for (int col = 0; col < 3; ++col) {
    kernel.at<float>(row, col) = 1.0/9.0;
  }
}
```

Exercise

Implement a sharpen filter using `cv::filter2D()`. The sharpen filter's kernel is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$