# Laboratory 8: Edge detection / Basic GUI / Displaying several images in the same window

The aims of today's lab are:

- Add sliders to our windows;
- Displaying several images next to each other.

In short, you are going to write parts of the demo programs seen during the lecture.

To achieve these goals, we will create several programs:

1. `edgeDetection1.cxx`: A simple program using OpenCV to detect edges;
2. `edgeDetection2.cxx`: We will improve the functionalities of the previous program to make it interactive;

You are provided with the skeletons and a `CMakeLists.txt` file in the `src` directory.

Everything we did last week is relevant to today's session. You are expected to have completed Lab 7 already. You are also expected to look for information on OpenCV's website if needed. The lab script provides a good starting point. Additional information can be found at `http://docs.opencv.org/master/d4/d86/group__imgproc__filter.html`.

# 1 Edge Detection using Scharr and Thresholding

We will write the code in `edgeDetection1.cxx`. The program takes two arguments from the command line:

- The input file (`argv[1]`);
- The output file (`argv[2]`).

## 1.1 Main Steps

The overall flow chart corresponding to the program is given in Figure 1. The skeleton of the program is provided, you need to complete it with your own code.

There are several main steps:

1. Declare some local variables;
2. Read the input;
3. Convert the RGB data to greyscale:
   (a) Convert the RGB data to greyscale (see `cv::cvtColor`). We did it last week and remember that OpenCV4 is not fully backward compatible with OpenCV3. In OpenCV3, the colour space conversion code is `CV_RGB2GRAY`; in OpenCV4, it is `cv::COLOR_RGB2GRAY`.
   (b) Convert the image from unsigned char to float (see `cv::Mat::convertTo`). Again, we did it last week and remember that OpenCV4 is not fully backward compatible with OpenCV3.
   (c) Normalise the image (see `cv::normalize`) so that the pixel values are in the range between 0 and 1. Last week we used
   ```
   cv::normalize(log_image, normalised_image, 0, 255,
       cv::NORM_MINMAX, CV_8UC1)
   ```
   `CV_8UC1` means that pixel values in the produced output (`normalised_image`) are stored using unsigned integers with 8 bits (`8U` in `CV_8UC1`) (i.e. `unsigned char` in C/C++), they correspond to luminance values (`C1` in `CV_8UC1`), and the dynamic range of `normalised_image` is [0, 255]. This week we will use
   ```
   cv::normalize(grey_image, normalised_image, 0.0, 1.0,
       cv::NORM_MINMAX, CV_32FC1)
   ```
   `32F` in `CV_32FC1` means `float` in C/C++.
4. Apply a $3 \times 3$ Gaussian filter with $\sigma$ equal to 0.5 to reduce noise (see `cv::GaussianBlur`).
5. Get the gradient image:
   (a) Apply the Scharr filter on the blurred image along the X-axis (see `cv::Scharr`);

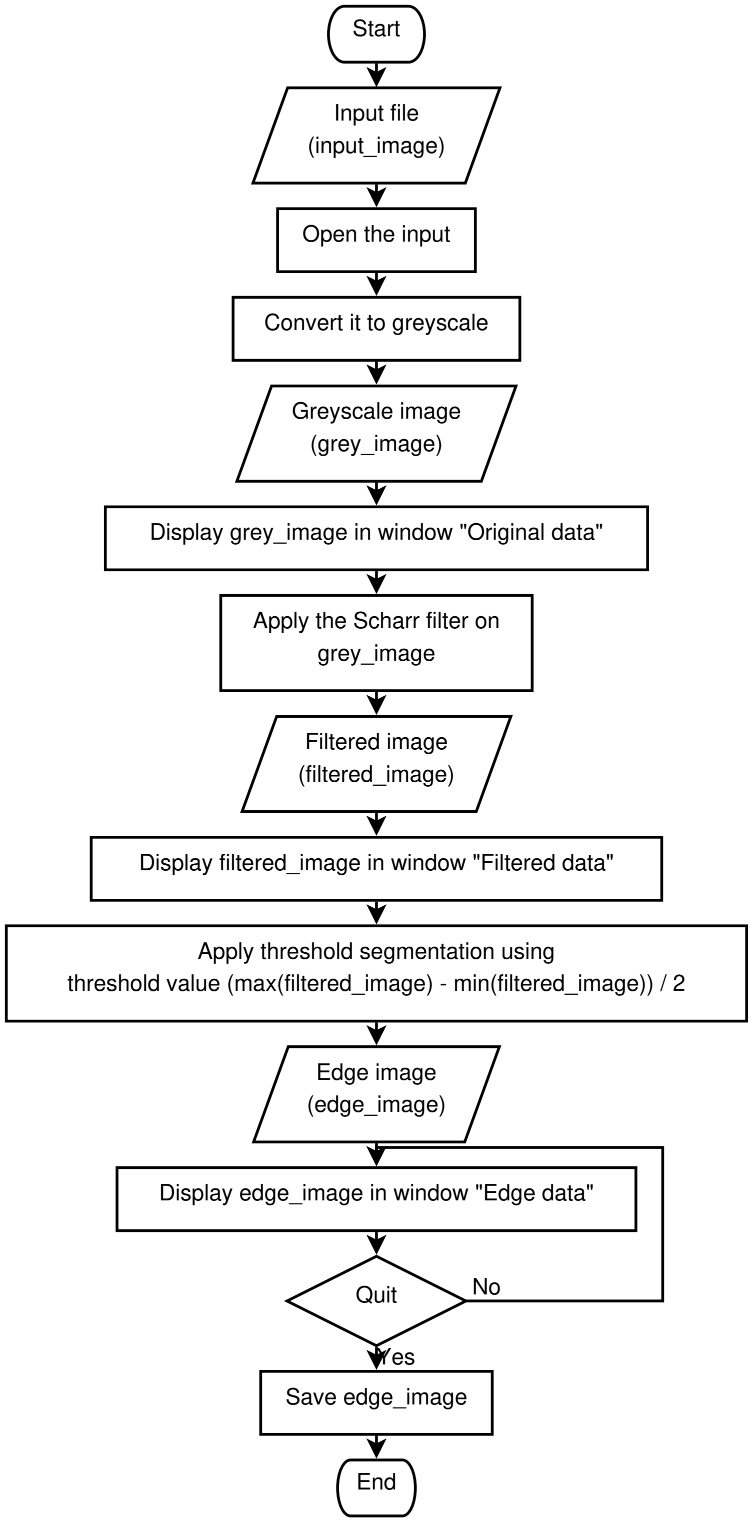


Figure 1: Flow chart of *edgeDetection1.cxx*.

(b) Compute the absolute value of the gradient along the X-axis (see `cv::abs`);
(c) Apply the Scharr filter on the blurred image along the Y-axis;
(d) Compute the absolute value of the gradient along the Y-axis;
(e) Combine the two images together so that:

$$gradient(x, y) = 0.5 \times |scharr_x(x, y)| + 0.5 \times |scharr_y(x, y)|$$

`operator*` and `operator+` have been overloaded in OpenCV. You can achieve the blending operation using them.

6. Find edges using a binary threshold filter (see `cv::threshold`).
7. Write the output. Remember to normalise the image between 0 and 255 before writing the file.

Last week, we used:

- `cv::cvtColor` in `rgb2grey.cxx`;
- `cv::Mat::convertTo` in `rgb2grey.cxx`;
- `cv::normalize` in `logScale.cxx`; and
- `cv::GaussianBlur` in `gaussianFilter.cxx`.

## 1.2 ImageDerivative (High-Pass Filter)

To calculate the image derivative, you need to call the Scharr operator:

```
void cv::Scharr(const cv::Mat& src, cv::Mat& dst, int ddepth, int dx, int dy)
```

with:

**src:** input image;

**dst:** output image of the same size and the same number of channels as `src`;

**ddepth:** output image depth, you can use `CV_32F`;

**dx:** order of the derivative x (= 0 or 1);

**dy:** order of the derivative y (= 0 or 1).

## 1.3 Pixel-wise Absolute Value Filter

To calculate the absolute image of the derivative, you need to call:

```
cv::Mat cv::abs(const cv::Mat& src)
```

with:

**src:** input image;

**return:** output image of the same size and the same number of channels as `src`;

### 1.4 Binary Threshold Filter

To calculate the absolute image, so that

$$dst(x,y) = \begin{cases} max\_value & \forall\, x\ \&\ y, \text{if } src(x,y) > threshold \\ 0 & otherwise \end{cases}$$

with $max\_value = 1$ in our case as we are dealing with pixel values stored using floating point numbers. You need to call:

```
cv::Mat cv::threshold(const cv::Mat& src, cv::Mat dst, double threshold, double
    max_value, int threshold_type)
```

with:

**src:** input image;

**dst:** output image of the same size and the same number of channels as `src`;

**threshold:** the threshold value;

**max_value:** the maximum value in the output;

**threshold_type:** the threshold type, here 0 for a binary threshold.

Try different value of threshold to get an acceptable result.

## 2 Improve the previous program

Copy paste some of the code from the `main` function of `edgeDetection1.cxx` into `edgeDetection2.cxx`. We are going to improve the program. Finding the best value of threshold is not easy. We can use an adjustable slider to do so. Also, we can display the 3 images side by side. In other words, we want to create a single window that looks like Figure 2.

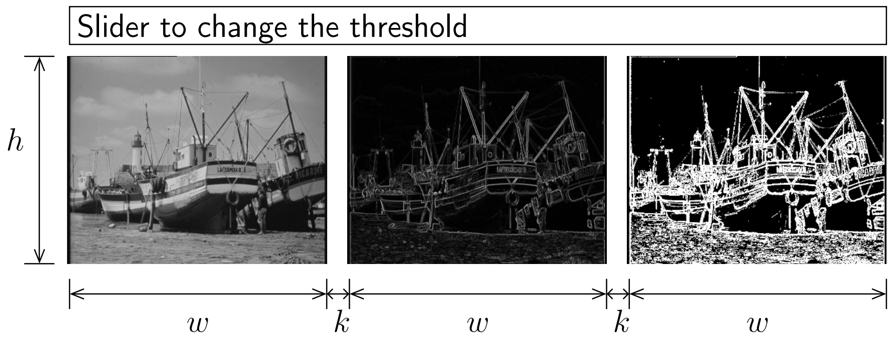


Figure 2: User interface. $w$ is the image width in pixels, $h$ is the image height in pixels, and $k$ is the number of empty pixels between two successive images.

## 2.1 Global variables

The slider will require a callback function that is why we need to move

```
cv::Mat g_scharr_image;
cv::Mat g_edge_image;
std::string g_image_window_title("Edge␣detection");
```

as global variables.

## 2.2 Displayed image

We also need to create a new image, e.g. g_display_image, as a global variable to hold the data to be displayed. Its size is $N \times w + (N-1) \times k$ with: $w$ the image width in pixels, $h$ the image height in pixels, $k$ the number of empty pixels between two successive images, and $N$ the number of images displayed in a row. It is created as follows:

```
// Create the displayed image
g_display_image = cv::Mat(rgb_image.rows, rgb_image.cols * N + (N - 1) * k,
    CV_32FC1, cv::Scalar(0.5, 0.5, 0.5));
```

Data from the images (grey_image, g_scharr_image and g_edge_image) will be copied in g_display_image. To do it, we first need to define the region of interest in the target image, for example with:

```
// Create the ROI in the target image
cv::Mat targetROI = g_display_image(cv::Rect(offset_x, offset_y, width, height))
```

then copy from the source to the target with:

```
grey_image.copyTo(targetROI);
```

- For grey_image, offset_x is $0 \times grey\_image.cols + 0 \times k$;
- For g_scharr_image, offset_x is $1 \times g\_scharr\_image.cols + 2 \times k$; and
- For g_edge_image, offset_x is $2 \times g\_edge\_image.cols + 2 \times k$. offset_y is null.

## 2.3 Slider

The slider is created with:

```
cv::createTrackbar(const string& aLabel, const string& aWindowTitle, int
    * aSliderPosition, int aSliderCount, void (*aCallback)(int, void*));
```

**aLabel:** the text describing the slider;

**aWindowTitle:** the title of the window to which the slider will be attached;

**aSliderPosition:** a pointer on the slider position;

**aSliderCount:** the number of ticks;

**aCallback:** the pointer on the callback function called when the slider moves (in C/C++ it is the name of the function without its parameters).

You can create two global variables such as:

```
int g_slider_count(256);
int g_slider_position(g_slider_count / 2);
```

This way, they will be available in the callback function.

### 2.4 Callback

The type of the call back is:

```
void callback(int, void*)
```

We do not use the parameters, ignore them. In the callback there are 4 steps:

- Get the threshold from `g_slider_count` and `g_slider_position` using linear interpolation.
- Compute the new image.
- Copy the results in `g_display_image`.
- Display `g_display_image` in the window.

## 3 Still some time left?

If you are done and still have a bit of time, you can look at displaying videos using OpenCV. Copy the code of `edgeDetection2.cxx` into `edgeVideo.cxx`. Checkout `http://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture` to use a video stream from a file (e.g. an AVI file) or from the webcam.