Laboratory 6: Introduction to OpenCV

The aims of today's lab are:

- Install OpenCV;
- Create a project using CMake;
- Display an image using OpenCV;
- Convert from RGB to luminance;
- Convert from `unsigned char` to `float`;
- Save an image into a file;
- Apply some basic filtering techniques.

To achieve these goals, we will create several programs:

1. `displayImage.cxx`: A simple program using OpenCV to open an image and display it in a window;
2. `rgb2grey.cxx`: A program to convert a RGB image in a greyscale image using OpenCV;
3. `meanFilter.cpp`: A program to perform the mean filter using OpenCV;
4. `logScale.cxx`: A program to display an image in the log scale;
5. `medianFilter.cpp`: A program to perform the median filter using OpenCV;
6. `gaussianFilter.cpp`: A program to perform the Gaussian filter using OpenCV.

Some test images are provided in the `images` directory.

1 Installing OpenCV

Before using OpenCV, you have to make sure it is installed on the machine you are using. If it is a University's PC, we will consider that it is not installed (an old version is already installed but it is out-dated for the version of MS Visual C++ that we are using).

- On Mac, OpenCV is available via Homebrew (`brew install opencv3`), Fink and Macports;
- Most Linux distributions have packages for OpenCV. Make sure you install the `-devel` package;

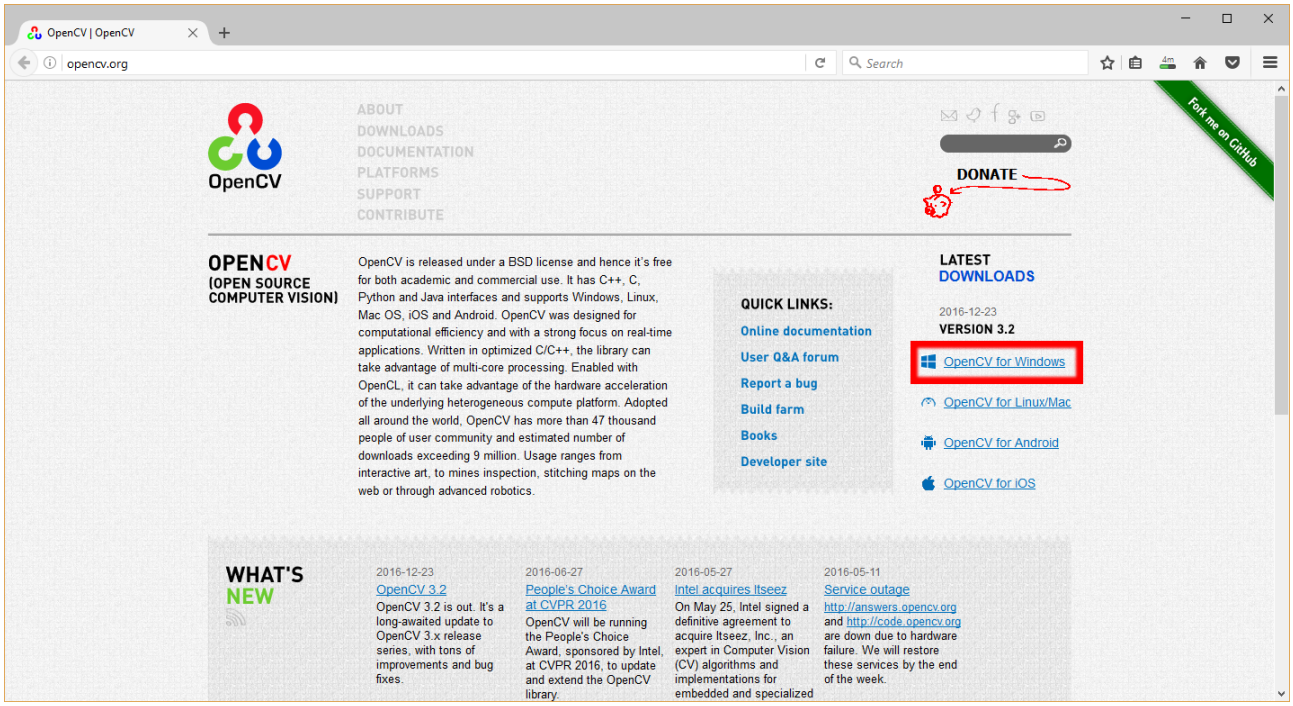

Figure 1: OpenCV's website.

- There are pre-compiled versions on `http://www.opencv.org/` that can be used on Windows.

  1. **Go to OpenCV's website** at `http://www.opencv.org/`.
  2. **Download the latest version** (i.e. 3.2.0): On the right hand side (see Figure 1), there is a link called "OpenCV for Windows". Clic on it to download `opencv-3.2.0-vc14.exe`.
  3. **Move the file** on `D:` drive using the File Explorer to avoid using the network (to speed-up the process).
  4. **Run the file** and extract the library in `D:` drive (see Figure 2).
  5. **Job done:** There is a new directory called `opencv` in the `D:` drive.

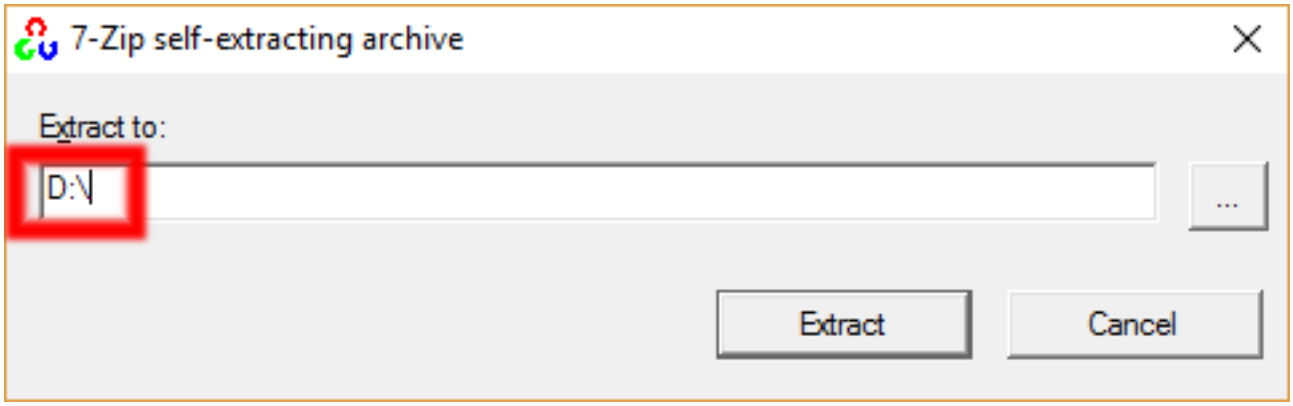

Figure 2: Installing OpenCV on the D: drive.

2 Using CMake

As some of you will use

- MS Windows on the University's computers,
- MS Windows on their own computer(s),
- Mac OS X, and
- Linux

it is important to keep in mind portability and we saw that toolchains can help us achieve it. CMake is an example of toolchain that is user friendly.

I provided a `CMakeLists.txt` file that will work on most platforms.

Listing 1: CMakeLists.txt.

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
PROJECT(ICP3038--Lab6--Intro2OpenCV)

# Add where OpenCV might be installed (look in D: first, then in C:)
IF (WIN32)
    SET (CMAKE_PREFIX_PATH ${CMAKE_PREFIX_PATH} "D:\\opencv\\build")
    SET (CMAKE_PREFIX_PATH ${CMAKE_PREFIX_PATH} "C:\\opencv\\build")
ENDIF (WIN32)


# Find OpenCV
FIND_PACKAGE(OpenCV REQUIRED)

# Add OpenCV's header path
INCLUDE_DIRECTORIES (${OpenCV_INCLUDE_DIRS})

# Add the libraries for OpenCV
SET (requiredLibs ${requiredLibs} ${OpenCV_LIBS})
```

```
# The executable programs
ADD_EXECUTABLE (displayImage   displayImage.cxx)
ADD_EXECUTABLE (rgb2grey       rgb2grey.cxx)
ADD_EXECUTABLE (logScale       logScale.cxx)
ADD_EXECUTABLE (meanFilter     meanFilter.cxx)
ADD_EXECUTABLE (medianFilter   medianFilter.cxx)
ADD_EXECUTABLE (gaussianFilter gaussianFilter.cxx)


# Add OpenCV libraries to each executable programs
TARGET_LINK_LIBRARIES (displayImage   ${requiredLibs})
TARGET_LINK_LIBRARIES (rgb2grey       ${requiredLibs})
TARGET_LINK_LIBRARIES (logScale       ${requiredLibs})
TARGET_LINK_LIBRARIES (meanFilter     ${requiredLibs})
TARGET_LINK_LIBRARIES (medianFilter   ${requiredLibs})
TARGET_LINK_LIBRARIES (gaussianFilter ${requiredLibs})
```

Configuring the project on MS Windows using the lab machines is relatively straightforward after following the instructions provided in Section 1 (see Figure 3).

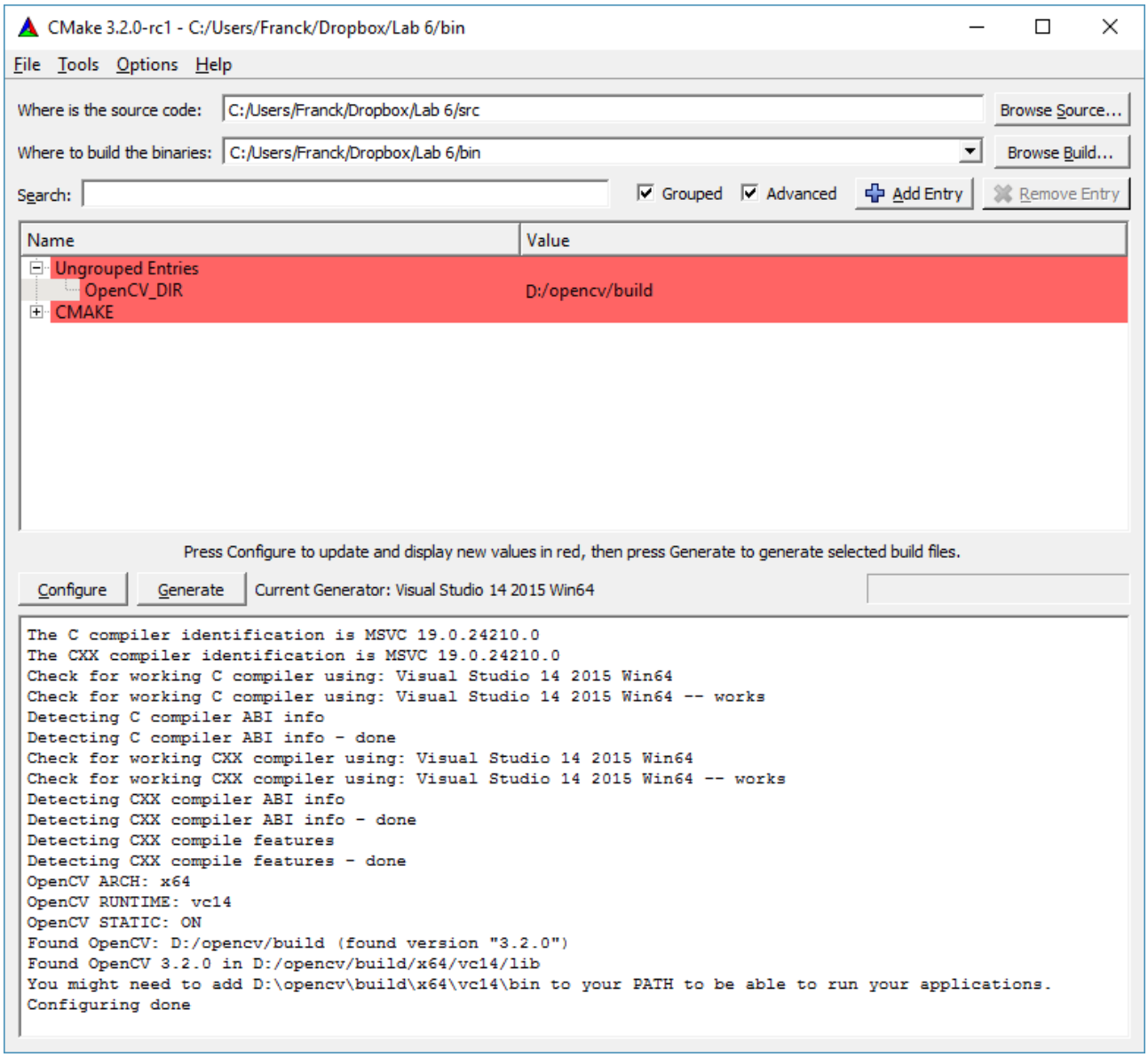

Figure 3: Project configuration using CMake.

3 Opening and Displaying an Image

3.1 Headers

Listing 2: Header files.

```
#include <exception> // Header for catching exceptions
#include <iostream>  // Header to display text in the console
#include <opencv2/opencv.hpp> // Main OpenCV header
```

OpenCV uses exceptions. To catch them, we need `<exception>`. To display text in the console `<iostream>` is required. The main OpenCV header is `<opencv2/opencv.hpp>`.

3.2 Main structure

As stated previously, OpenCV uses exceptions. We can (or should) catch them to handle errors. The structure of the main is shown in Listing 3.

Listing 3: Initial program to display an image using OpenCV.

```
//-----------------------------
int main(int argc, char** argv)
//-----------------------------
{
    try
    {
        // No file to display
        if (argc != 2)
        {
            // Create an error message
            std::string error_message;
            error_message  = "usage:␣";
            error_message += argv[0];
            error_message += "␣<input_image>";

            // Throw an error
            throw error_message;
        }

        // Write your own code here
        //....
        //....
        //....
    }
    // An error occured
    catch (const std::exception& error)
    {
        // Display an error message in the console
        cerr << error.what() << endl;
    }
    catch (const std::string& error)
    {
        // Display an error message in the console
        cerr << error << endl;
```

```
    }
    catch (const char* error)
    {
        // Display an error message in the console
        cerr << error << endl;
    }

    // Exit the program
    return 0;
}
```

3.3 Arguments of the Command Line

The first program only takes one parameter. It corresponds to the path of an image file. To make sure the number of arguments is correct, you can use:

Listing 4: Checking the number of command line arguments.

```
// No file to display
if (argc != 2)
{
    // Create an error message
    std::string error_message;
    error_message  = "usage: ";
    error_message += argv[0];
    error_message += " <input_image>";

    // Throw an error
    throw error_message;
}
```

To get the file name, you can use:

Listing 5: Getting the file name from the command line arguments.

```
std::string input_filename(argv[1]);
```

3.4 Reading the File

An image is stored in an instance of the class `Mat`. Note that OpenCV's namespace is `cv::`. To declare the variable that will hold our image, type:

```
// Create an image instance
cv::Mat image;
```

The image is loaded using:

Listing 6: Open an image.

```
// Open and read the image
image = cv::imread( input_filename, CV_LOAD_IMAGE_COLOR );
```

It is a good practice to check if any error occurred, e.g. to avoid unspecified behaviours and crashed. If the image is not loaded, its `data` field is empty. If it is the case we can throw an error as follows:

Listing 7: Check that the image contains data

```
// The image has not been loaded
if (!image.data)
{
    // Create an error message
    std::string error_message;
    error_message  = "Could␣not␣open␣or␣find␣the␣image␣\"";
    error_message += input_filename;
    error_message += "\".";

    // Throw an error
    throw error_message;
}
```

3.5 Displaying the Image

There are four steps to create a window and display and image:

1. Create a string to contain the window title (it is used to identify the window);
2. Create the window;
3. Show the image in the window;
4. Wait for a user input to leave the window.

It can be done as follows:

Listing 8: Create an image.

```
// Create a string to contain the window title
string window_title;
window_title  = "Display␣\"";
window_title += input_filename;
window_title += "\"";

// Create the window
cv::namedWindow(window_title, cv::WINDOW_AUTOSIZE);

// Show the image in the window
cv::imshow(window_title, image);

// Wait for a user input to leave the window
cv::waitKey(0);
```

The program is now complete. You can compile it and run it with different image files to test it. Figure 4 shows a screenshot of the program.

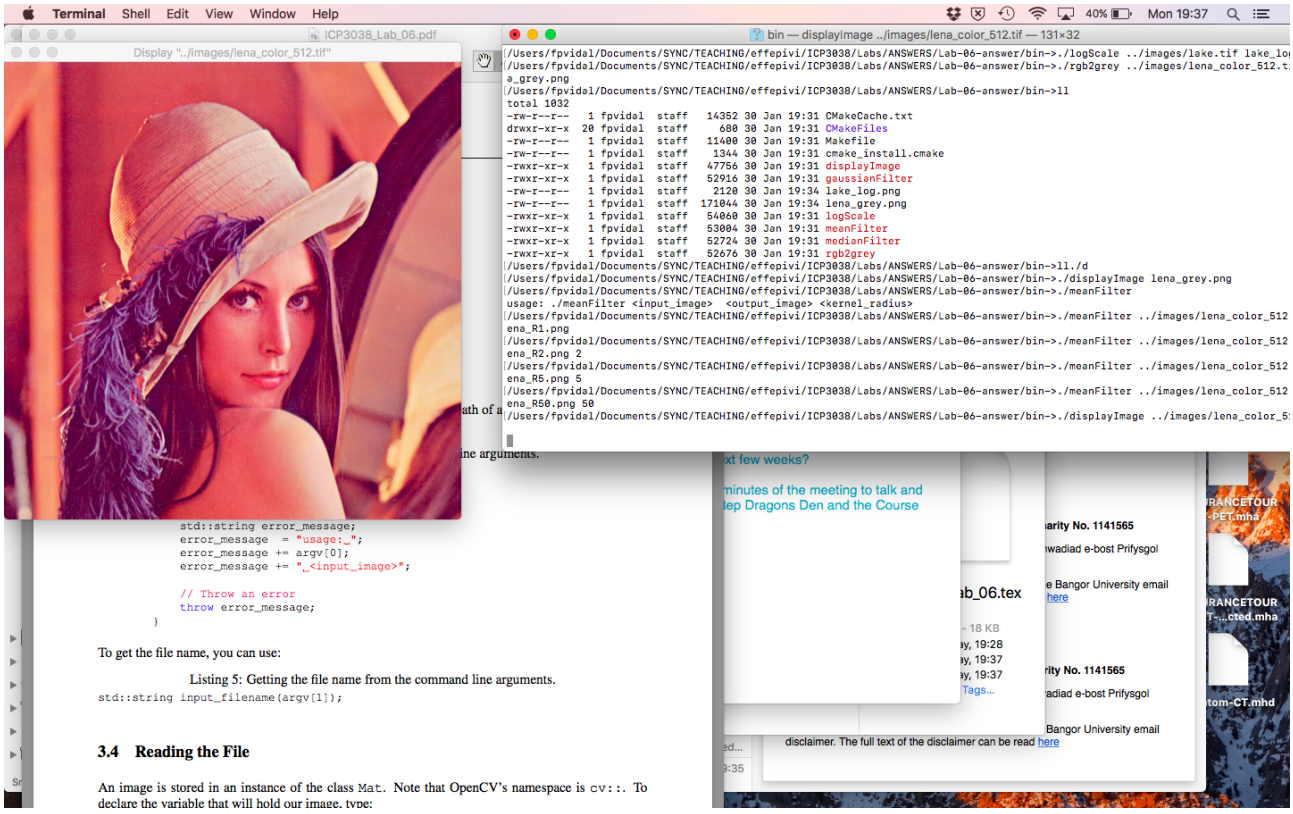

Figure 4: Screenshot of **displayImage**.

4 Convert a RGB Image in a Greyscale Image

Copy the main function of `displayImage.cxx` into `rgb2grey.cxx`.

4.1 Arguments of the Command Line

The second program takes two parameter:

1. The path of the input RGB image file, and
2. The path of the output greyscale image file.

Modify the code accordingly.

4.2 Converting from RGB to Greyscale

After displaying the RGB image and BEFORE `cv::waitKey(0)`, create a new image called `grey_image`. To convert the original image in greyscale, simply type:

Listing 9: Convert the color model of the image.

```
// If the image is not a greyscale image, then convert it.
cv::Mat grey_image;
cv::cvtColor(image, grey_image, CV_RGB2GRAY);
```

In OpenCV in general, the first argument is the input image; the second argument is the output image; other arguments are the parameters of the function. Now create another window where to display the new image.

4.3 Saving an Image into a File

The function to save an image is `cv::imwrite(file_name, image)`. It returns true if the file has been successfully written; false otherwise. We can use the return value to handle possible errors:

Listing 10: Save an image.

```
// Write the image
if (!cv::imwrite(argv[2], grey_image))
{
    // The image has not been written

    // Create an error message
    std::string error_message;
    error_message  = "Could not write the image \"";
    error_message += argv[2];
    error_message += "\".";

    // Throw an error
    throw error_message;
}
```

Calling `rgb2grey lena_color_512.tif lena.png` should produce the output presented in Figure 5.

(a) Input image. (b) Output image.

Figure 5: Input and output of **rgb2grey**.

5 Mean Filter

Let us consider the mean filter. Copy parts of the main function of `rgb2grey.cxx` into `meanFilter.cxx`. The new program will take 3 inputs:

1. The input image;
2. The output image; and
3. The convolution kernel radius.

To convert a C string into an integer, use the `atoi` function from the `<cstdlib>` header. It will be needed to get the kernel radius from the command line argument. To set the kernel size, you need to use an instance of the `cv::Size` class. You also have to specify its size. You can use:

```
// Filter size
cv::Size filter_size(kernel_width, kernel_height);
```

or

```
// Filter size
cv::Size filter_size;
filter_size.width = kernel_width;
filter_size.height = kernel_height;
```

Note that

- If the radius is 0, then the kernel size is 1×1
- If the radius is 1, then the kernel size is 3×3
- If the radius is 2, then the kernel size is 5×5
- ...
- If the radius is 7, then the kernel size is 15×15
- etc.

Now you are ready to filter the input image. Use either `cv::blur` or `cv::boxFilter`. They are the same. The first argument is the input image; the second is the output image; and the third one is the kernel size. Display and save the output image. Try different kernel sizes to see the differences (see Figure 6).

(a) Radius = 1. (b) Radius = 2.

(c) Radius = 5. (d) Radius = 50.

Figure 6: Outputs of **meanFilter**.

6 Display an Image in the Log Scale

The main function of `rgb2grey.cxx` into `logScale.cxx` as it is important to use a greyscale image in this new program. Fig. 7 shows the shape of the $\log$ function. Looking at the y axis, we note that it is important to store the image using floating point numbers. If we don't, there will be enormous quantisation problems.

To convert the greyscale image from `unsigned char` to `float`, we use:

Listing 11: Convert an image into floating point numbers.

```
// Convert to float
cv::Mat float_image;
grey_image.convertTo(float_image, CV_32FC1);
```

It can be seen on the figure that $\log(x) \forall x \in]-\infty, 0]$ is undefined. In other word, if x is equal to zero or x is negative, then there is no y value. As the input image was using `unsigned char`, we do not have to worry about negative values. However, we have to make sure no 0 value is present in the

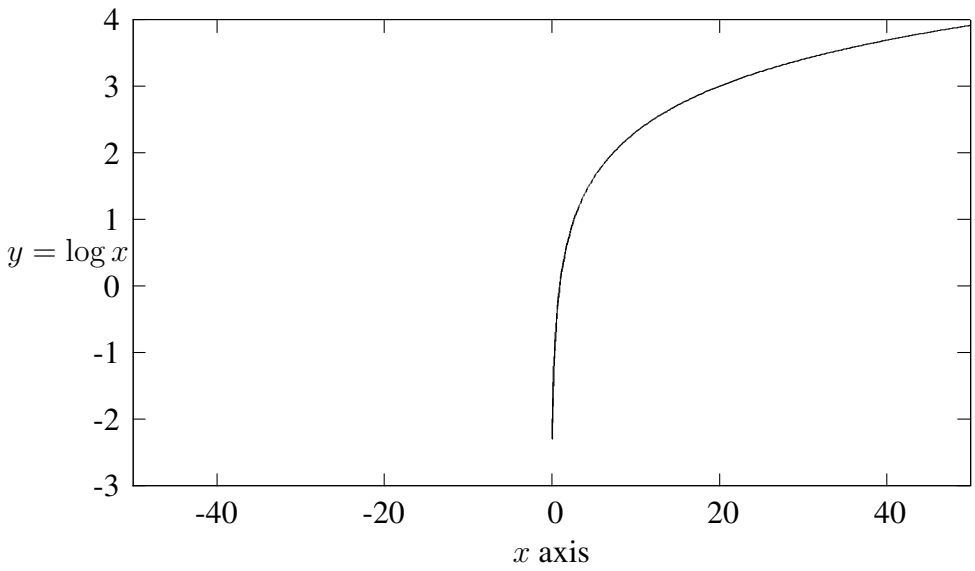

Figure 7: The $\log$ function.

image. To do so, we apply the following transformation:

$$f'(x, y) = \log(f(x, y) + 1) \tag{1}$$

using

```
// Log transformation
cv::Mat log_image;
cv::log(float_image + 1.0, log_image);
```

Looking at the curve, we notice another problem. In some case, $\log(x)$ may be negative. In this case, it is common to normalise the image so that its values lie in the range $[0, 1]$ using:

$$f''(x, y) = \frac{f'(x, y) - \min(f')}{\max(f') - \min(f')} \tag{2}$$

There are two ways to achieve this in OpenCV. You can implement Eq. 2 using:

```
double min, max;
cv::minMaxLoc(log_image, &min, &max);
cv::Mat normalised_image = 255.0 * (log_image - min) / (max - min);
normalised_image.convertTo(normalised_image, CV_8UC1);
```

or you can use OpenCV's function:

```
// Normalisation
cv::Mat normalised_image;
cv::normalize(log_image, normalised_image, 0, 255, cv::NORM_MINMAX,
    CV_8UC1);
```

Now you can display and save the image (see Figure 8.

(a) Input image. (b) Output image.

Figure 8: Input and output of **logScale**.

7 Additional tasks

If you still have some time left before the end of the lab, investigate the use of the median and Gaussian filters:

- `cv::medianBlur`, see `http://docs.opencv.org/3.2.0/d4/d86/group__imgproc_filter.html#ga564869aa33e58769b4469101aac458f9`
- `cv::GaussianBlur`, see `http://docs.opencv.org/3.2.0/d4/d86/group__imgproc_filter.html#gaabe8c836e97159a9193fb0b11ac52cf1`