# COMP27112: Visual Computing

# Lab 3 – Using OpenCV


Terence Morley

## 1 Introduction

This lab is formative — please work through it at your own pace ensuring that you understand the basics ready for the summative lab that you will do later.

The aim of this lab is to get you started with OpenCV and to do some simple image processing. OpenCV is a large, popular library, and learning how to use it will allow you to quickly perform image processing functions and also to learn a bit more about the practical use of image processing techniques.

OpenCV (https://opencv.org/) is an open source image processing and computer vision library that is released under the Apache 2 licence (from OpenCV v4.5.0; earlier versions are under the 3-clause BSD licence). The licence allows you to use and modify the library and freely distribute it, including using the library in any commercial software that you create and sell.

The original library was created by Intel and the first version was released in the year 2000. It now has a user community of over 47,000 users and the library has been downloaded over 18 million times. OpenCV is written in C++ and has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and MacOS.

Since it is a well-established and widely used library, there are a lot of resources on the internet. If you run into problems with the assignment, this should be one of the first places to seek help.

> This document asks you do do a number of things to gain practical experience in programming with OpenCV. This includes writing code and running it. But there are also things for you to experiment with. These are highlighted with a ★ symbol to make sure that you don't miss them.

## 2 Intended Learning Outcomes

By the end of this assignment, you should be able to:

- Install the OpenCV library
- Implement image processing code using OpenCV with C/C++
- Compile and execute your programs
- Investigate the use of image processing techniques using OpenCV

# 3 Set Up Your Development Environment

This section shows you how to install the OpenCV library on your PC if you want to do that.

## 3.1 Using a Kilburn Lab Machine

The lab machines in Kilburn already have OpenCV for C/C++ and Python installed. If you only want to develop on the Kilburn machines, you can jump to Section 4.

## 3.2 Linux and Mac

You can download the source code for OpenCV from https://opencv.org/releases/. This web-page has all of the versions for the different platforms.

The instructions on how to install on Linux can be found here:
https://docs.opencv.org/4.x/d7/d9f/tutorial_linux_install.html

The instructions on how to install on MacOS can be found here:
https://docs.opencv.org/4.x/d0/db2/tutorial_macos_install.html

For both of these operating systems, the steps are:

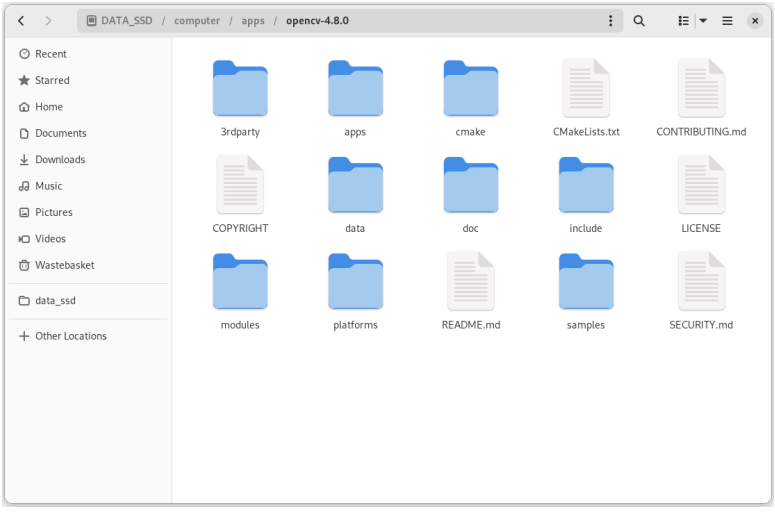
- Download the source code zip file
- Unzip the zip file
- Configure the installation using `cmake`
- Build and install the library using `make`

## 3.3 Example Linux Installation

If you want to install OpenCV on your own PC, this section will give you a quick look at how it is done on Linux.

### 3.3.1 Get Source Code

This example uses OpenCV 4.8.0. Download the source code zip file and unzip it into a directory:




Open a terminal in this directory, create a new directory called `build` and go into it:

```
mkdir build
```

```
cd build
```

### 3.3.2 Configure the Installation

You need the GNU C++ compiler (to compile the OpenCV library and to compile your programs), so install it:

```
sudo dnf install g++
```

This example uses `dnf` to install programs on Fedora Linux. Your distribution may have a different installer such as `apt` or `yum`.

There are many options that can be set up for the OpenCV installation such as for enabling GPU support and installing example code. The configuration is managed with cmake so install that:

```
sudo dnf install cmake
```

Now we can configure the installation. For our purposes, we will set the option `OPENCV_GENERATE_PKGCONFIG`. We will see later what that option does for us.

```
cmake -DOPENCV_GENERATE_PKGCONFIG=ON ..
```

This is run in the build directory, so '..' tells cmake to look in the directory above.

You should now see lots of messages displayed while cmake sets up the build.

### 3.3.3 Build and Install

Now compile OpenCV:

```
make -j4
```

Note that the '-j4' tells it to use four cores of your CPU. In the terminal, type in **nproc** and this will tell you how many cores you have got available. If it says 8, then you can change this to '-j8' to speed up the make.

You should see lots of messages being display with a progress percentage displayed on each line. This will take a few minutes altogether.

Once the build is finished, install the library:

```
sudo make install
```

OpenCV should then be ready for you to use in your programs.

## 3.4 Microsoft Windows

The easiest way to use OpenCV on Windows is by using the pre-compiled binaries (dll – dynamic link libraries). I have written a guide called `opencv_on_visual_studio.pdf` which is available on Blackboard. This guide shows you how to get the OpenCV libraries and how to set up projects using Visual Studio.

You can also follow the instructions on the OpenCV website:
https://docs.opencv.org/4.8.0/d3/d52/tutorial_windows_install.html

## 3.5 Editor/IDE

You can use any editor or IDE for this lab. For example,

- A simple text editor

- Atom or Sublime Text
- Visual Studio or Visual Studio Code
- Eclipse, NetBeans, etc.

# 4 Your First OpenCV Program

> It is better to type in these first few programs because you will read what you are writing, you will make mistakes and will have to read it again to find the correction. This will help burn it in to your memory.

Type the following code into a file called `version.cpp` or get a copy from Blackboard.

version.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>

int main(int argc, char *argv[])
{
        // Print the OpenCV version
        printf("OpenCV version: %d.%d\n", CV_MAJOR_VERSION, CV_MINOR_VERSION);
        return 0;
}
```

Now we need to compile the program `version.cpp`, but we need to tell the compiler where the header files are (`core.hpp`, in this case) and where the libraries are. We do this with the following command:

```
g++ version.cpp -o version `pkg-config --cflags --libs opencv4`
```

**Note:** In this command, the quotes are back-quotes (the key that is usually next to your '1' key).

This command runs the GNU C++ compiler (`g++`) on the source file `version.cpp`. It specifies the name of the produced executable, `version`, with the '-o' option.

The pkg-config part finds out the location of the headers and libraries. This information is available because the `OPENCV_GENERATE_PKGCONFIG` option was set in the configuration above.

If there are no errors in your `version.cpp` file, running the `g++` command will compile your program and the directory will now contain an executable called `version`. You can run your program with the following:

```
./version
```

You should see the following output from your program (but with it showing the actual version of OpenCV that you installed):

```
OpenCV version: 4.8
```

> The code you have just written contains only C statements, but you can write C++ code if you want. In either case, the program **must** be compiled as a C++ program.

## 4.1 Extra Information

**pkg-config** is a separate program that you can run in the terminal:

```
pkg-config --cflags --libs opencv4
```

Running this command will produce something like the following:

```
-I/usr/local/include/opencv4 -L/usr/local/lib64 -lopencv_gapi -lopencv_highgui -lopencv_ml
    -lopencv_objdetect -lopencv_photo -lopencv_stitching -lopencv_video -lopencv_calib3d
    -lopencv_features2d -lopencv_dnn -lopencv_flann -lopencv_videoio -lopencv_imgcodecs
    -lopencv_imgproc -lopencv_core
```

This ouput is passed to the `g++` command. The '-I' specifies the location of the include files, the '-L' specifies the location of the library files, and each of the '-l' items specifies the libraries to use.

If we didn't use pkg-config in the `g++` command (for example, if we didn't add it in the OpenCV configuration), we could just put those items on the `g++` command:

```
g++ version.cpp -o version -I/usr/local/include/opencv4 -L/usr/local/lib64 -lopencv_core
```

This program didn't do much. It just printed out the version number from the OpenCV library. You compiled it with a command that specified the header files and the libraries. However, the program didn't make any use of the OpenCV libraries. It could have been compiled with the following, without mentioning the libraries:

```
g++ version.cpp -o version -I/usr/local/include/opencv4
```

But, we will normally use the OpenCV libraries in our programs so we need to specify them, and this is done most easily using the pkg-config program as shown above.

## 5 Loading and Displaying Images

This program will use OpenCV to open an image and display it in a window. OpenCV's function to read images, `imread()`, is declared in `imgcodecs.hpp` and the image is displayed with the `imshow()` function which is declared in the `highgui.hpp` header file. So, these files need to be included in the source code. Type in the following code into a file called `display.cpp`:

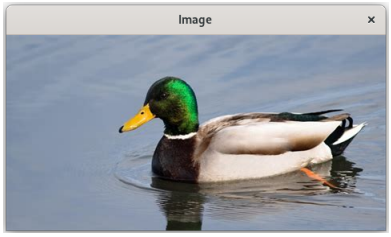
display.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>

int main(int argc, char *argv[])
{
        cv::Mat img; // Images are stored in Mat class (Mat = matrix)

        img = cv::imread(argv[1]);

        // Check if the image was successfully loaded
        if (img.empty()) {
                        printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                        return -1;
        }

        cv::namedWindow("Image", cv::WINDOW_NORMAL);
        cv::imshow("Image", img);

        // Wait for a key press before quitting
        cv::waitKey(0);

        return 0;
}
```

Compile and run the program as we did earlier, but remember that the filename is different. Pass in an image filename as a command-line parameter:

```
g++ display.cpp -o display `pkg-config --cflags --libs opencv4`

./display duck.png
```

You should now see the image displayed in a window as shown below. If you press a key, the window will close and the program will end.



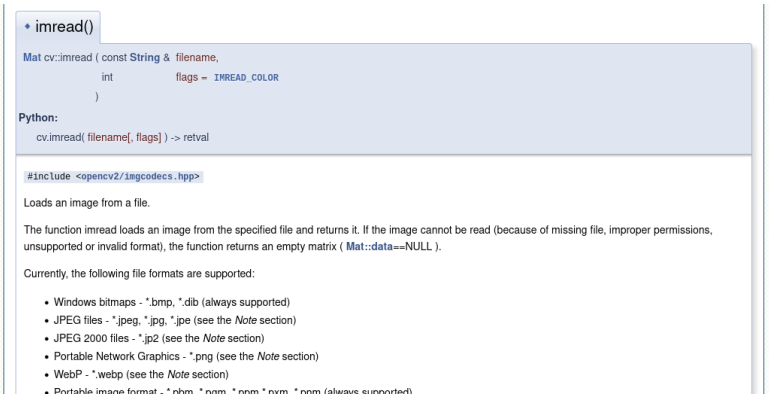

**Note:** The image `duck.png` is provided on Blackboard, but you can use any image. Later in this section, you will investigate using different images.

## 5.1 Header Files

How do you know which header files to include in your programs?

Go to the OpenCV documentation webpages for your version of OpenCV, for example: https://docs.opencv.org/4.8.0/.

Type the name of a function into the search box (e.g. `imread`) and you will see the documentation for that function:




The required header file is stated just under the blue title section.

## 5.2 Investigations

### 5.2.1 Window Style

Use the program that you wrote in Section 5 to open the file `big_duck.jpg` that you can find on Blackboard:

```
./display big_duck.jpg
```

Notice that this image is in jpeg format whereas the previous one was a png file. OpenCV can read many types of image file.

But, the main thing that you might have noticed is that the image doesn't fit on your screen. The file was named `big_duck.jpg` because it is a big image of a duck, not because it is an image of a big duck.

In your program, `display.cpp`, there is an instruction (`namedWindow()`) that creates a window for you to later

put an image in with the `imshow()` function. The parameters for this function are the name to display in the window title bar and a window style, which in this case was set to `WINDOW_NORMAL`:

```
cv::namedWindow("Image", cv::WINDOW_NORMAL);
```

★ Modify your program so that the window style is set to `WINDOW_AUTOSIZE` and try again.

That makes no difference? The `WINDOW_NORMAL` option allows the window to be resized by dragging the borders. The `WINDOW_AUTOSIZE` option makes the window fit the image but doesn't allow you to manually resize the window. Probably the better option is to set the window to `WINDOW_NORMAL` and then resize the window in code. This still allows the user to manually resize the window afterwards:

```
cv::namedWindow("Image", cv::WINDOW_NORMAL);
cv::resizeWindow("Image", 800, 500);
cv::imshow("Image", img);
```

You are probably thinking that you might like the image to be zoomed in and out and for scroll bars to appear when the image is too big for the window. This is possible, but you need to write the code to do the resizing of the image and to manage scroll bars, this is not done automatically by OpenCV.

### 5.2.2 Greyscale Images

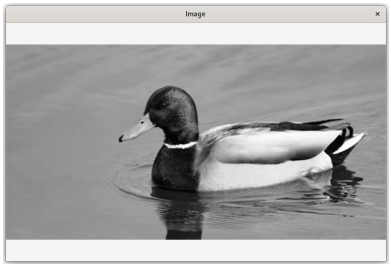
In image processing, a lot of processing is performed on greyscale images rather than colour ones. Greyscale images just show the intensity in the image as shades of grey from black to white. Reasons for using greyscale images are that the processing is faster and a lot of the information in an image is stored in the intensities.

Duplicate your `display.cpp` file as `displaygrey.cpp`, then modify the `imread()` function call so that is looks like the one below:

displaygrey.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>

int main(int argc, char *argv[])
{
        cv::Mat img; // Images are stored in Mat class (Mat = matrix)

        img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

        // Check if the image was successfully loaded
        if (img.empty()) {
                        printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                        return -1;
        }

        cv::namedWindow("Image", cv::WINDOW_NORMAL);
        cv::resizeWindow("Image", 800, 500);
        cv::imshow("Image", img);

        // Wait for a key press before quitting
        cv::waitKey(0);

        return 0;
}
```

The `cv::IMREAD_GRAYSCALE` flag tells OpenCV to load the image in greyscale format rather than colour. When you run this program on `duck.png`, it should look like this:

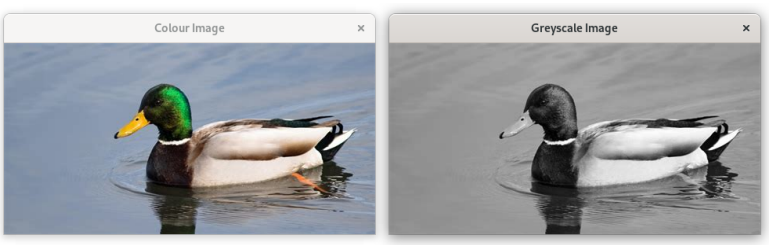



Often, in image processing, you want to do some processing on the greyscale image: maybe you want to find a contour around a shape. But you might want to show this contour drawn in red on the original colour image, rather than the greyscale one. For this reason, you normally read the image in as colour and then make a greyscale copy of it in your program. That way, you have got the colour version and the greyscale one to work with. You can do this using the `cvtColor()` function as shown below.

displaygrey2.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

int main(int argc, char *argv[])
{
        cv::Mat img;

        img = cv::imread(argv[1]); // Load image in colour

        // Check if the image was successfully loaded
        if (img.empty()) {
                        printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                        return -1;
        }

        // Make a greyscale copy of the image
        cv::Mat grey_img;
        cv::cvtColor(img, grey_img, cv::COLOR_BGR2GRAY);

        cv::namedWindow("Colour␣Image", cv::WINDOW_NORMAL);
        cv::imshow("Colour␣Image", img);

        cv::namedWindow("Greyscale␣Image", cv::WINDOW_NORMAL);
        cv::imshow("Greyscale␣Image", grey_img);

        // Wait for a key press before quitting
        cv::waitKey(0);

        return 0;
}
```

Your program's windows should look like this:




Notice the flag on the `cvtColor()` function call:

```
        cv::cvtColor(img, grey_img, cv::COLOR_BGR2GRAY);
```

It says BGR2GRAY (BGR to grey). You might be used to colour images being described as RGB (red, green, blue). This refers to the order of the channels used to represent each pixel: a byte for red, followed by a byte for green, and then a byte for blue. In OpenCV, the channels are stored in the opposite order (BGR: blue, green, red).

# 6 Image Processing with OpenCV

We have looked at opening image files and displaying the image in a window. We can do that in lots of different programs. We want to do some proper processing on images that is not available in the usual image manipulation applications. (Well, in this lab, we are going to investigate a couple of techniques that *are* available in image manipulation applications, but this will get you started and you can investigate the massive number of functions that are available in OpenCV at your leisure.)

# 7 Blurring Images

There are a number of reasons why you would want to blur images. I would blur photographs of myself so that people cannot see all of my wrinkles. You might blur images of circuit boards to help you with edge detection to find the boards – without blurring, edge detection would find the edges around the individual components.

An simple way to blur an image is by using a box filter, or averaging filter. This is a process where each pixel value in an image is replaced with the average value of the pixels surrounding it. The number of pixels used to calculate the average has an effect on how much blurring is performed. This is normally done by stating the size of a rectangle that should be used, centred on the pixel. You might use a 3×3 rectangle to do just a bit of blurring, and you might use 15×15 to blur the image more.

The following code shows how to load an image as greyscale and blur it. Type this code into a file called `blur.cpp` and try it on the image `woman.jpg` that is supplied on Blackboard.

blur.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

int main(int argc, char *argv[])
{
        cv::Mat img;
        cv::Mat out;

        img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

        // Check if the image was successfully loaded
        if (img.empty()) {
                        printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                        return -1;
        }

        // Blur the image
        cv::blur(img, out, cv::Size(5, 5));

        cv::namedWindow("Image", cv::WINDOW_NORMAL);
        cv::imshow("Image", out);

        // Wait for a key press before quitting
        cv::waitKey(0);
```

```
        cv::imwrite("blurred_image.jpg", out);

        return 0;
}
```

★ In the `blur()` function call, try different values for the size of the kernel (rectangle of coefficients) — try 5×5, 15×15 and 25×25, for example.

The following is the result for 25×25:

Notice the `imwrite()` function call. This is how you write an image to a file. Check your directory after running the program and you should see the file `blurred_image.jpg`.

# 8 Thresholding

A common image processing technique is segmentation. This is where you split an image into various parts. That is, you give a label to distinct areas of an image. A simple way of doing this is with binary thresholding. This is where you look at each pixel intensity and if it is above a certain value (threshold), you set the pixel to white (255), otherwise you set it to black (0).

## 8.1 Binary Thresholding

Copy your `blur.cpp` (which loads an image as greyscale) to `threshold.cpp` and modify the main part of it to the following:

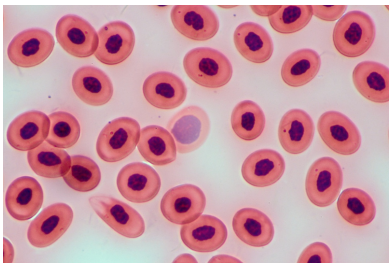
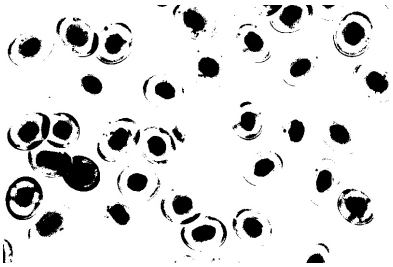
threshold.cpp

```
        // Threshold the image
        int T = 128;
        cv::threshold(img, out, T, 255, cv::THRESH_BINARY);

        cv::namedWindow("Image", cv::WINDOW_NORMAL);
        cv::imshow("Image", out);

        // Wait for a key press before quitting
        cv::waitKey(0);
```

Compile your program and run it on the provided image of frog blood cells:

```
./threshold frog_blood_cells.jpg
```

The following shows the original colour image and the thresholded grey image obtained for the threshold value $T = 128$:

★ Modify the threshold value to different values and try to get a 'good' result:

```
int T = 128;
```

**Question:** What is a 'good' threshold result?

When you were just altering the threshold value, $T$, you were probably trying to get a clean image showing each complete cell with no incorrect detections. Or you might have been trying to isolate the cell nuclei.

But, a good result depends on your desired outcome. Your aim might be to locate the cell, or the nuclei, or maybe to locate cells that have got some sort of damage. So, a 'good' result is one that gives you what you need.

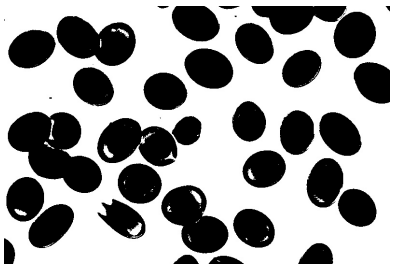
## 8.2 Automatic Thresholding

There are various methods for automatically choosing a threshold value which examine the statistical distribution of intensities in an image. One such method is Otsu thresholding which was developed by Nobuyuki Otsu in 1979.

★ Modify the thresholding line of your `threshold.cpp` program to the following:

```
cv::threshold(img, out, T, 255, cv::THRESH_BINARY | cv::THRESH_OTSU);
```

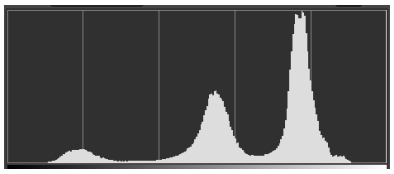
When you specify `OTSU`, OpenCV ignores the threshold value that you supply and automatically finds a reasonable value. This is the result for the blood cell image:



**Question:** Will Otsu always be a good thing to use when performing thresholding?

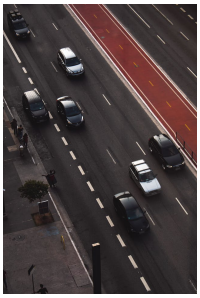
So, this looks like a good result — but, only if your intention is to find the whole cell. If you want the nuclei, then this won't work.

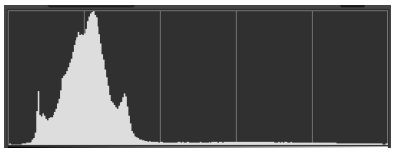
The image below shows the histogram for a greyscale version of the blood cells image. The small peak on the left shows the almost black nucleus pixels, the middle peak shows the cell body pixels, and the large peak on the right shows the almost white background pixels



**Question:** How do you think that the Otsu algorithm uses the information from the histogram? (Given that there are two output classes in the thresholded image: white and black.)

Imagine that you want to automatically count cars on a road. Below is an image of cars on a road and the histogram of the image. Do you think that using Otsu will isolate the cars? Do you think that you will be able to manually select a threshold value that will separate the cars from the background?





★ Try your Otsu thresholding program on the supplied `cars.jpg` image. Remove the `THRESH_OTSU` flag and try a few different threshold values.

## 9 Interactivity

As you have seen from the blurring and thresholding exercises, many image processing techniques need user-specified parameter values. The choice of these values depends on the image and your desired outcome.

When you were trying to find a good threshold value for the blood cell and cars images, you probably got fed up of entering different values and then examining the results. It would be good if you could vary parameter values and immediately see their effect.

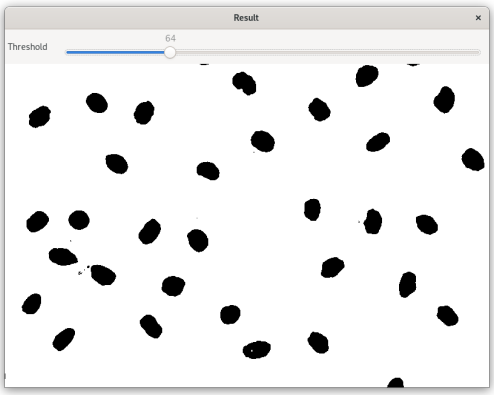
Type in the following code or use the file supplied on Blackboard:

threshold_interactive.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

cv::Mat img;
cv::Mat out;

void thresh_onchange(int val, void* userdata)
{
        cv::threshold(img, out, val, 255, cv::THRESH_BINARY);
        cv::imshow("Result", out);
}

int main(int argc, char *argv[])
{
        img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

        // Check if the image was successfully loaded
        if (img.empty()) {
```

```
                printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                return -1;
        }

        cv::namedWindow("Result", cv::WINDOW_NORMAL);
        cv::createTrackbar("Threshold", "Result", NULL, 255, thresh_onchange);

        // Call the callback function for the initial processing
        thresh_onchange(0, NULL);

        // Wait for a key press before quitting
        cv::waitKey(0);

        return 0;
}
```

This code creates a window as normal but then calls `createTrackbar()` which adds a trackbar to the window. The trackbar can be moved to vary a value. The window will look like this (where I have tried to find only the nuclei):




The first parameter in `createTrackbar()` is a name for the value being varied and the second is the name of the window to attach the trackbar to. The third parameter is not used in this example, but you can pass in the address of a variable to receive the trackbar value. The minimum value for a trackbar is always 0 and the fourth parameter specifies a maximum value; therefore, our trackbar goes from 0 to 255. The final parameter is the address of a callback function; that is, a function that OpenCV will call whenever the trackbar is moved. At the top of the program, the callback function is declared. This function performs the thresholding with the value from the trackbar and then displays the image in the window.

Because the two variables `img` and `out` are used in both `main()` and the callback, they were declared as global variables in this program.

You can attach multiple trackbars to a window.

★ Run the program on the images `frog_blood_cells.jpg` and `cars.jpg` and move the trackbar to obtain what you think might be useful segmentations for some purpose.

**Question:** How can you measure how well you have segmented an image? Are there any metrics to measure your success? What are these metrics measuring against?

## 10 Manipulating Pixels

Although there are many functions available in the OpenCV library, there will be times when you want to do something that OpenCV doesn't do. When processing images yourself, you will want to loop through pixels, examine their values and set them to new values. In this section, you will investigate how to access pixel data.

## 10.1 Greyscale Image

When an image is read in using `IMREAD_GRAYSCALE`, or you have converted a colour image to greyscale with `COLOR_BGR2GRAY`, the image (Mat) is made up of a two-dimensional array of unsigned bytes. A byte is an 8-bit number that can represent the numbers 0 to 255. Black is represented by 0 and white by 255. All of the shades of grey are the numbers between these.

If we want to invert an image (create an image-negative), we can simply replace each pixel intensity value, $I$, with $255 - I$. So a pixel with a value of 0 will change to 255, a value of 1 will change to 254, a value of 255 will change to 0.

Type in the following program or use the file supplied on Blackboard:

invert.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

int main(int argc, char *argv[])
{
	cv::Mat img;

	img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

	// Check if the image was successfully loaded
	if (img.empty()) {
		printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
		return -1;
	}

	// Invert the image
	for(int r = 0; r < img.rows; r++)
	{
		for (int c = 0; c < img.cols; c++)
		{
			img.at<uchar>(r, c) = 255 - img.at<uchar>(r, c);
		}
	}

	cv::namedWindow("Image", cv::WINDOW_NORMAL);
	cv::imshow("Image", img);

	// Wait for a key press before quitting
	cv::waitKey(0);

	return 0;
}
```

The thing that is different in this program is the nested loop in the middle of the file. You can see here that the size of the image is obtained via `img.rows` and `img.cols`. A nested loop is made to cycle through each row number and then each column number. The pixel intensities are obtained with the `at()` function providing a data type to the template, that is `img.at<uchar>(r, c)`. Using the template with a data type of `uchar` tells OpenCV to access the pixel as a `uchar` which is an unsigned byte.

We will see in the next section how this is slightly different if we want to access colour pixels.

## 10.2 Colour Image

In the example code below, notice that the `imread()` function is called with no flags and the default for the function is to read the image in colour (BGR).

This time, we need to access the pixels as a vector of three bytes (a byte each for the blue, green and red channels). This is done by passing `Vec3d` to the template. We can access the vector like an array using indices, for example `px[0]` accesses the blue channel.

In the following code, we cycle through every pixel in the image and set the blue and green channels to zero, leaving only the red channel with its original value. This will just show us the amount of red throughout the image.

red.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

int main(int argc, char *argv[])
{
        cv::Mat img;

        img = cv::imread(argv[1]);

        // Check if the image was successfully loaded
        if (img.empty()) {
                printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                return -1;
        }

        // Invert the image
        for(int r = 0; r < img.rows; r++)
        {
                for (int c = 0; c < img.cols; c++)
                {
                        cv::Vec3b px = img.at<cv::Vec3b>(r, c);
                        px[0] = 0; // Blue
                        px[1] = 0; // Green
                        // Leave Red channel untouched
                        img.at<cv::Vec3b>(r, c) = px;
                }
        }

        cv::namedWindow("Image", cv::WINDOW_NORMAL);
        cv::imshow("Image", img);

        // Wait for a key press before quitting
        cv::waitKey(0);

        return 0;
}
```

This program was compiled and was run on the supplied image `balloons.jpg`. The original and the output are shown here:


Comparing these two images, you can see that the deep blue of the sky has no red content so it appears black in the output. The white clouds do contain a lot of red and that can be seen. The left-hand balloon appears to be almost all red in the output image. This is because the yellow at the bottom of the balloon is made up of red and green so the high red content can be seen in the output.

## 11 Namespace

Are you fed up of typing `cv::` in front of everything? You can tell the compiler that you are using the `cv` namespace and then you don't need to keep adding it. See the program below noticing the `using namespace cv;` statement.

namespace.cpp

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

using namespace cv;

int main(int argc, char *argv[])
{
        Mat img;
        Mat out;

        img = imread(argv[1], IMREAD_GRAYSCALE);

        // Check if the image was successfully loaded
        if (img.empty()) {
                        printf("Failed␣to␣load␣image␣'%s'\n", argv[1]);
                        return -1;
        }

        // Blur the image
        blur(img, out, Size(25, 25));

        namedWindow("Image", WINDOW_NORMAL);
        imshow("Image", out);

        // Wait for a key press before quitting
        waitKey(0);

        imwrite("blurred_image.jpg", out);

        return 0;
}
```

# 12 Conclusion

After working through this lab you should now be comfortable using OpenCV with C++. You know how to read, display and write images; you know how to use a few of OpenCV's image processing functions; and you know how to manipulate pixel intensities in an image.

You are now in a position to investigate OpenCV further, and to do the next lab which will be assessed.