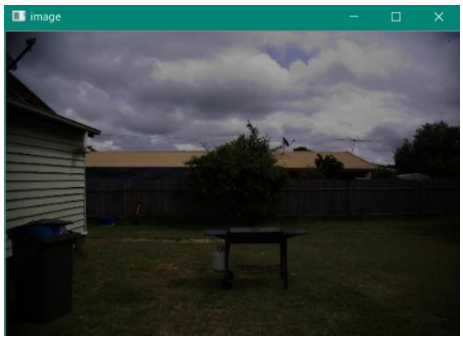
Lab 3 Pietro Balzan 1241795

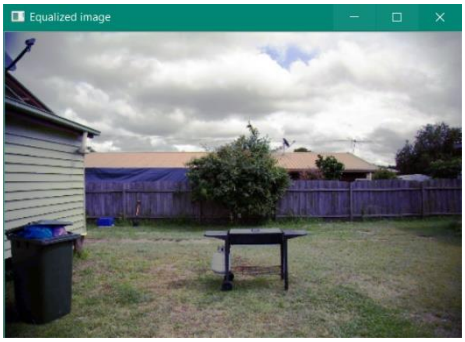
In this lab, I learnt how to equalize histograms and blur images in different ways, using OpenCV with c++.

In the project, the code is split between two different .cpp files (along with the filter.h header file).

The file lab3.cpp contains the main function to be executed. Inside it, the program reads the input image from the command line (`argv[1]`), before resizing and showing it.

The input image is very dark and it's difficult to see colors very well, but after equalizing the histograms of the red, green and blue channels, the result is still not very satisfactory, as some dark areas become unnaturally blue (see for example the wooden fence of the equalized image) and most other colors are "washed out" and faded.





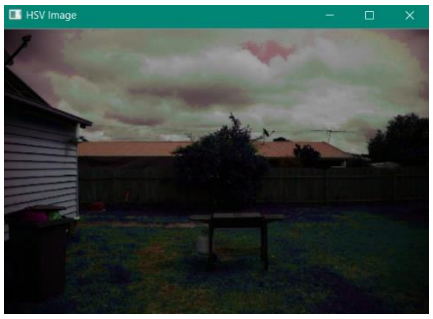
The histograms are calculated and shown through two methods:

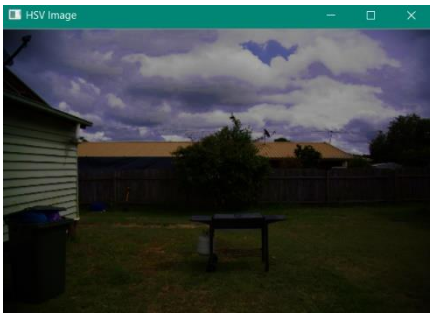
- `calculateHistograms(Mat bgr[3])` which takes a 3-channel image in the form of array of Mat and returns the vector that will be later used to show the histograms
- `showHistogram(std::vector<cv::Mat>& hists)` which was provided to us

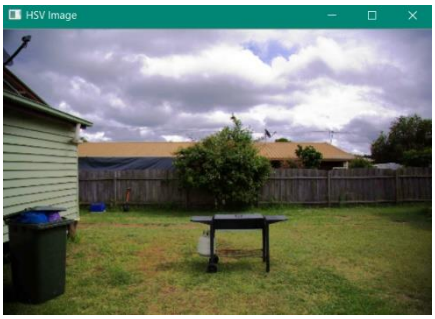
and are then equalized using the `equalizeHist()` OpenCV method.

To show the resulting image, the histograms are merged back together using the `pushandMerge()` method which just pushes the channels into a `vector<Mat>` and uses it to merge them into the returned `Mat` containing the image.

A similar process is followed when equalizing the HSV channels, but this time only one of them needs to be equalized in order to produce a satisfying result. Here are the different outputs when equalizing the three channels (from left to right: H is equalized, S is equalized, V in equalized):







As it's easy to see, the best result is obtained when equalizing the Value channel, which makes sense because the original image was very dark and we're equalizing Value, which is the dimension of lightness/darkness.

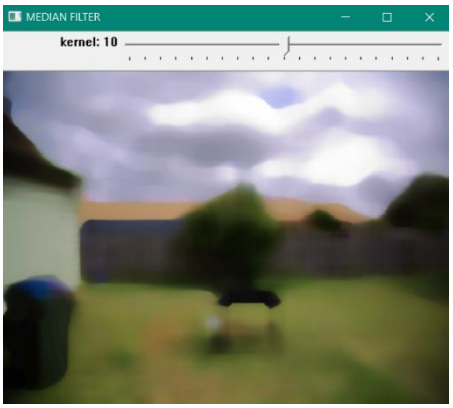
As for the blurring of the image, three different filters were applied to the equalized HSV image.

This is where the filter.h and filter.cpp files are used: in filter.h a Filter parent class was defined, along with three different classes that extend it, one for each type of filter, and each one with different variables that were needed to apply the filtering (such as the kernel size or sigma), and each overwriting the doFilter() function in order to apply the correct filter to the image. (filter.cpp just contains the implementation).

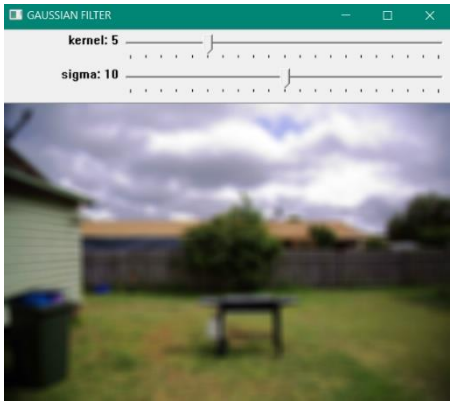
In the main function, for each filter, I created a namedWindow with trackbars that change the parameters of the filtering, which are then used to create the corresponding Filter objects, call the doFilter() function and show the result inside the callback methods of the trackbars.

These are the results for:

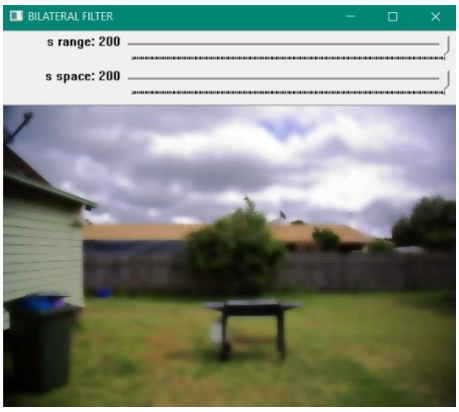
- Median Filter



- Gaussian Filter



- Bilateral Filter



We can see that the Median Filter gives more of a smoothing effect of the surfaces and areas, the Gaussian image almost seems like an out of focus image, with a reduction in detail and image noise, while the final filter provides smoothing of the image while preserving the edges (more so than median and gaussian, see the legs of the barbecue table, for example).