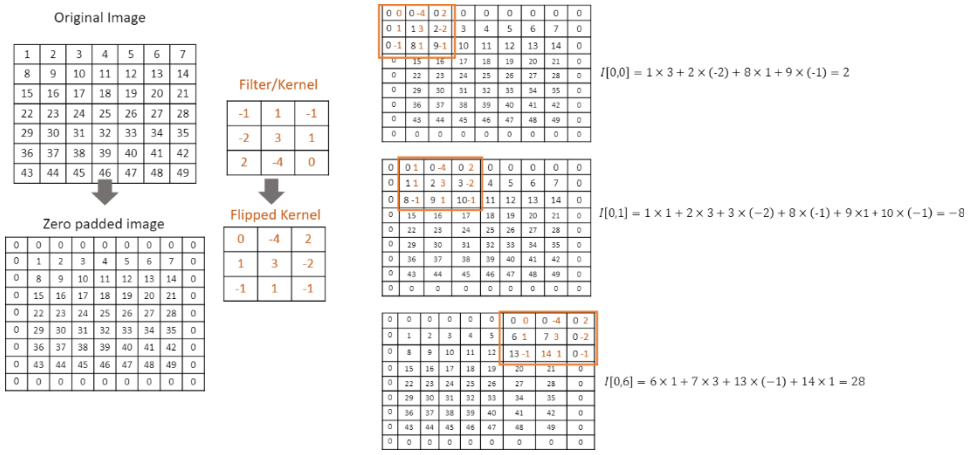
REPORT LAB 1: LINEAR FILTERING

1 CONVOLUTION OPERATION

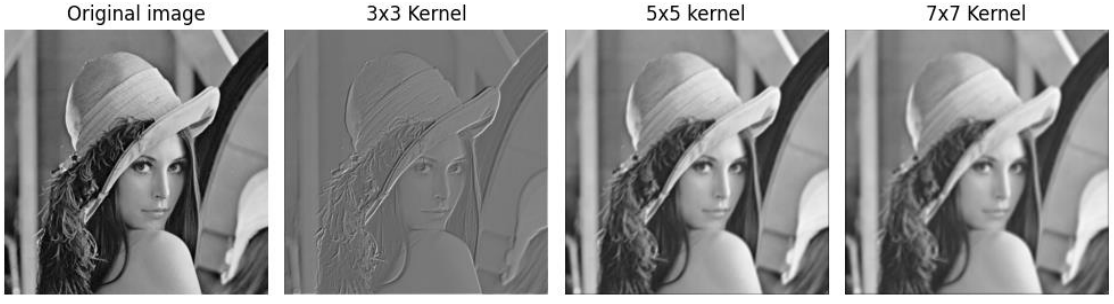
In mathematical terms, convolution is a mathematical operation on two functions (f and g) that produces a third function ($f * g$). This new function expresses how the shape of one is modified by the other. But, why are we applying convolution in image processing?

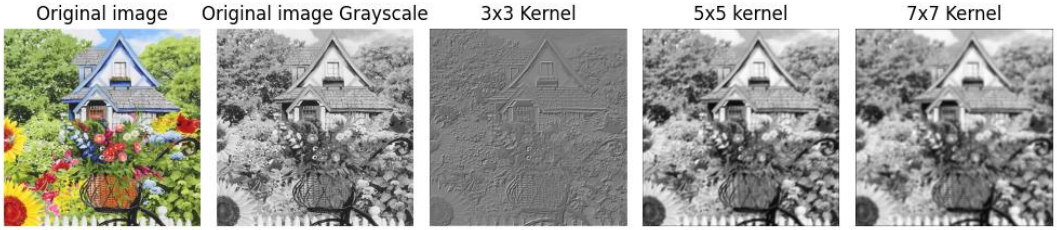
Convolution in image processing is used to apply a filter to an image in order to enhance, modify or suppress some of its features depending on the filter we use. Some of the various purposes of convolution are: smoothing or blurring, edge detection, sharpening, feature extraction…

Then, how do we apply a filter to an image? The convolution operation consists of moving the filter (also known as kernel) over the image, as a sliding window, and computing the weighted sum of the pixel values in the neighborhood of each pixel. The size of the kernel will define the nature of the filtering operation.



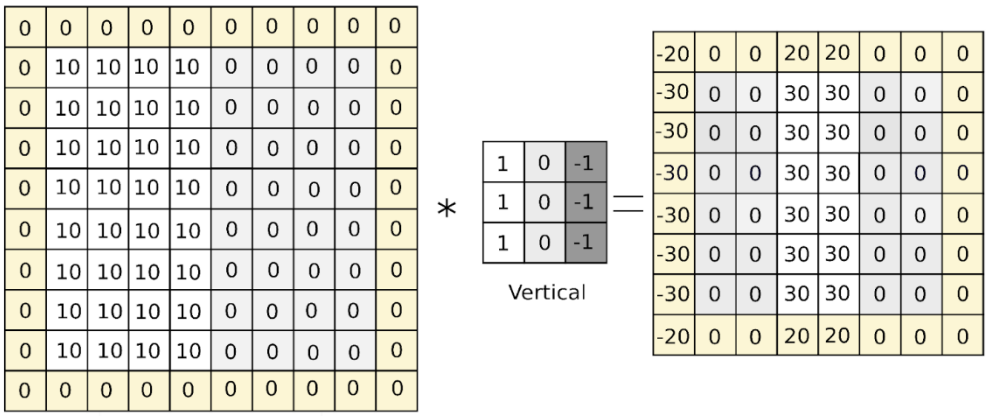
We often use odd-sized kernels because they have a center pixel, which makes it easier to apply the filter symmetrically around that center. Even-sized kernels can still be used but they require additional techniques for centering the kernel. However, it is important to know that by changing the size of the kernel, we can alter the degree of smoothness or feature extraction applied to the image. If the kernel is larger, it will cover more pixels at a time, resulting in more significant smoothing or feature enhancement, while smaller kernels have a more localized effect.





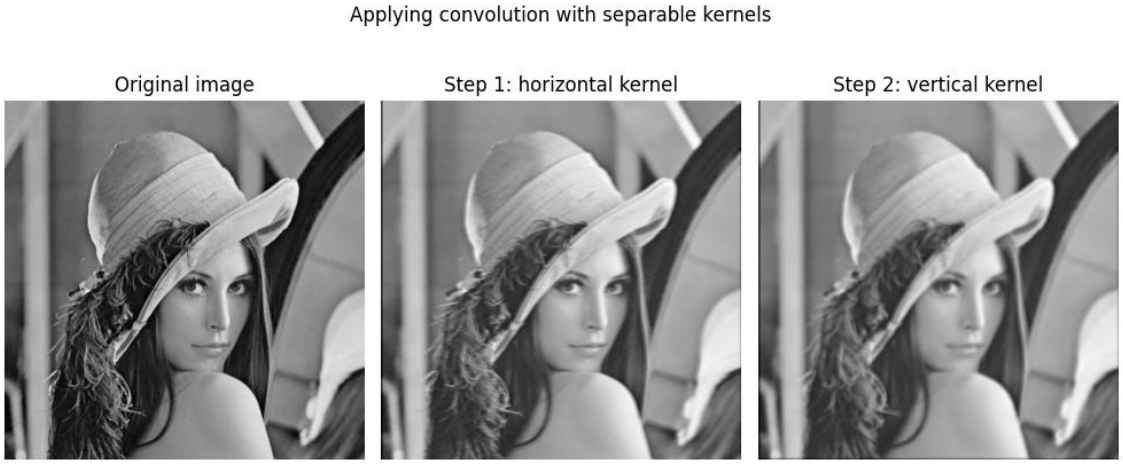
Before applying this filter to an image, we use a flipped version of the kernel to ensure that the operation is associative and commutative. By being associative we ensure that the order in which you apply multiple convolution operations with different kernels doesn't affect the final result. On the other hand, being commutative means the result of convolving a kernel K with an image I is the same as convolving I with K. If we didn't flip the kernel, we would be applying correlation, which is essentially convolution but with the original kernel.

Moreover, to ensure that the output feature map is the same size as the input image, we can apply padding, which consists on adding extra pixels around the boundary of the image matrix so the kernel, when it is near boundaries and corners, is able to perform a same size x same size multiplication of matrices. Normally we use 0-padding (the numbers added are 0) so we can keep the size of the output the same as the input. However, we can use other numbers for different interests. The required padding size each side needs is calculated by the *kernel_height//2* for the extra rows and *kernel_width//2* for the extra columns. When there is no padding, convolution is defined as "valid", and when there is padding is defined as "same".



In addition, *stride* another parameter in convolution that determines how the kernel moves across the image. This parameter is a number that defines the step size of the kernel. If we want to slide through all the pixels we would set *stride=1*. A larger stride will reduce the size of the output feature map, since it will skip pixels, which can be useful for downsampling an image. On the other hand, a smaller stride can lead to overlap, which means it would capture more detailed information of the image.

Now, let's introduce a new way to apply convolution: separable kernels. They idea behind it is that a 2D kernel can be separated into two 1D kernels, one applied horizontally and the other vertically. This method can improve the computational efficiency of convolution operations and it produces very similar results as non-separated kernels. Also, since they are 1D, they use less memory, which is an advantage when we are working with large images or limited storage.



For a better understanding of convolution, let's compute an easy example. We have the following 3x3 kernel and grayscale image:

```
image = np.array([[2,3,1,2,3],
                  [5,6,4,5,6],
                  [8,11,7,8,9],
                  [4,6,78,4,3],
                  [6,1,2,4,6]])

kernel = np.array([[-2, -1, 0],
                   [-1, 1, 1],
                    [0, 1, 2]])
```

To apply convolution, we would flip the kernel:

```
kernel_flipped = np.array([[ 2  1  0]
                           [ 1  1 -1]
                           [ 0 -1 -2]])
```

We would add padding to our image matrix to keep the output as the same size:

```
padded_image = np.array([[ 0.  0.  0.  0.  0.  0.  0.]
                         [ 0.  2.  3.  1.  2.  3.  0.]
                         [ 0.  5.  6.  4.  5.  6.  0.]
                         [ 0.  8. 11.  7.  8.  9.  0.]
                         [ 0.  4.  6. 78.  4.  3.  0.]
                         [ 0.  6.  1.  2.  4.  6.  0.]
                         [ 0.  0.  0.  0.  0.  0.  0.]])
```

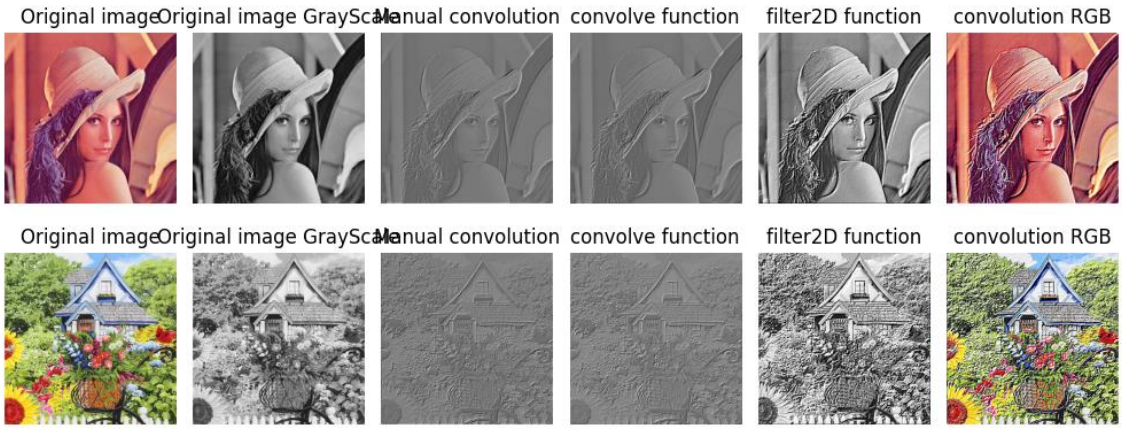
Finally, we would slide our kernel pixel by pixel on the image and apply the corresponding computations. The results would be the following:

```
convolution_matrix = np.array([[ -18.  -10.  -12.  -17.   -1.]
                               [ -29.  -11.  -11.  -19.    9.]
                               [ -14. -134.  -60.    9.   30.]
                               [  -2.  -46.   99.   85.   26.]
                               [   9.   19.   89.  160.   21.]])
```

These same steps (except for flipping the kernel) are applied correlation. The fundamental difference between convolution and correlation is that in convolution the kernel is flipped vertically and horizontally, but in correlation we use the original kernel. On the one hand, convolution's applications involve smoothing and sharpening images, detecting edges, etc. On the other hand, correlation measures the similarity between two signals. Therefore, it is often used for pattern matching and feature detection in images. For example, using the same kernel and image from the example done above, the result for correlation would be the following:

```
correlation_matrix = np.array([[  22.   16.   14.   21.    7.]
                               [  39.   23.   19.   29.    3.]
                               [  30.  156.   74.    7.  -12.]
                               [  10.   58.   57.  -77.  -20.]
                               [   3.  -17.  -85. -152.   -9.]])
```

In our code, to apply convolution we compared different methods: using CV implemented function cv2.filter2D, using convolve2D function and then using two manual implementations (one for grayscale images and the other for color images). We have compared the output of applying the same kernel with all the methods and these are the results:



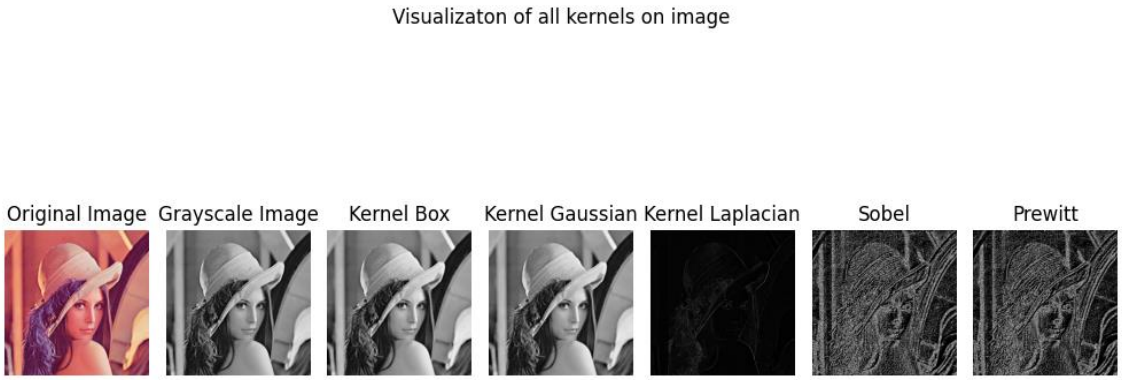
As we can observe, using manual convolution and convolve2D function gives us very similar results, but the output of filter2D function is way more different.

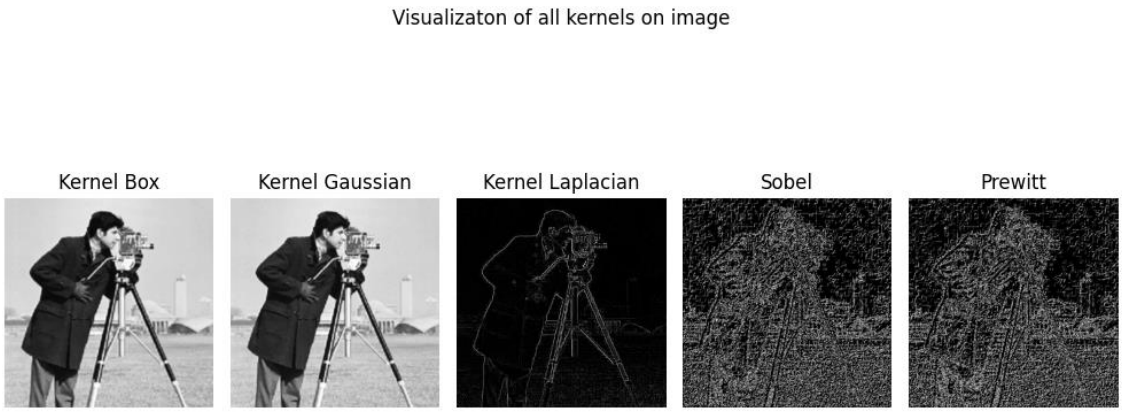
2 LINEAR FILTERING

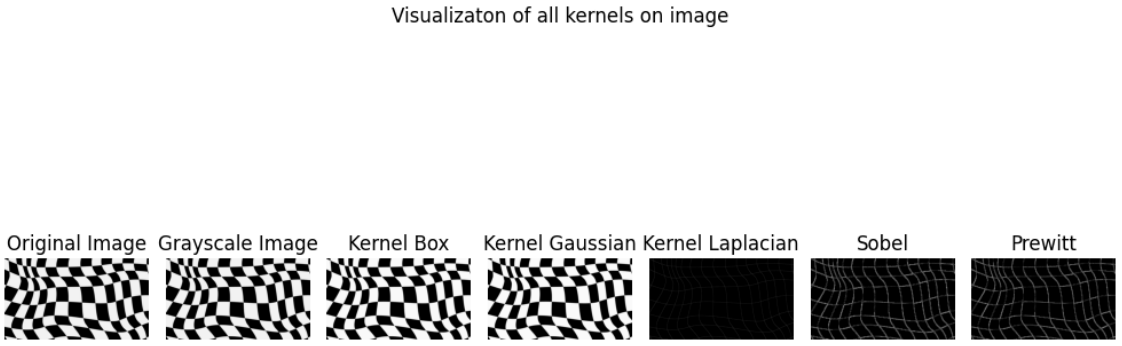
The purpose of image filtering is to modify or enhance image properties or to extract valuable information from the pictures, such as edges, corners or bulbs. We can achieve this by removing noise, sharpening edges or smoothing out textures. Some common applications of image filtering are: image restoration, feature extraction and object recognition. As we have seen before, convolution is a very useful technique to apply linear filtering. To apply these filters we have been looking at greyscale images as 2D arrays, whose values represent pixel intensities between 0 and 255, 0 being black and 255 being white.

To represent a color image, we would add a third dimension, each representing an intensity value in Red, Green and Blue. Since three colors have values from 0 to 255, there would be a total of 256*256*256 = 16,777,216 color combinations.

However, we will focus on applying filters to grayscale images. Depending on the filter values, the convolution can have a variety of effects. Here are some examples using different kernels:



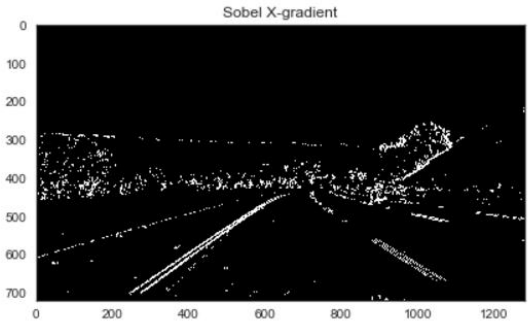




What do these filters do exactly? Box filters blur the image by replacing each pixel with the average value of its neighboring pixels; Gaussian filters are also smoothing filters that can help reduce noise while preserving edges better than a box filter; Laplacian filters enhance the edges in an images, and so do Sobel and Prewitt filters. These three last filters are not specifically designed for noise reduction.

But, would these filters be relevant for a real-scenario application like a self-driving car vision system or a medical image analysis? The answer is yes! For example Gaussian filters can be used to reduce noise captured by cameras, improving object detection and lane tracking, and also they can be used to remove noise in radiological images, improving diagnostic accuracy.

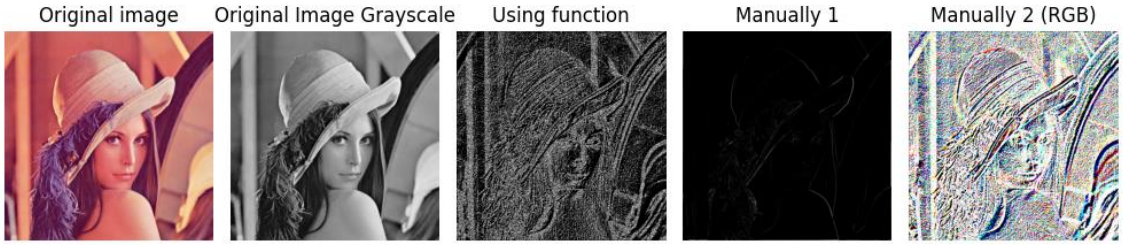
Sobel filters, jointly with Canny filters are used for edge detection, which is essential for identifying lane marks, other vehicles, obstacles… In medical imaging, it helps identifying organ boundaries, tumors, and other anatomical features.



In our code, we tried to apply these filters using implemented library functions but also using our own convolution function, and these are the results for some of the filters:

Kernel Gaussian

Sobel

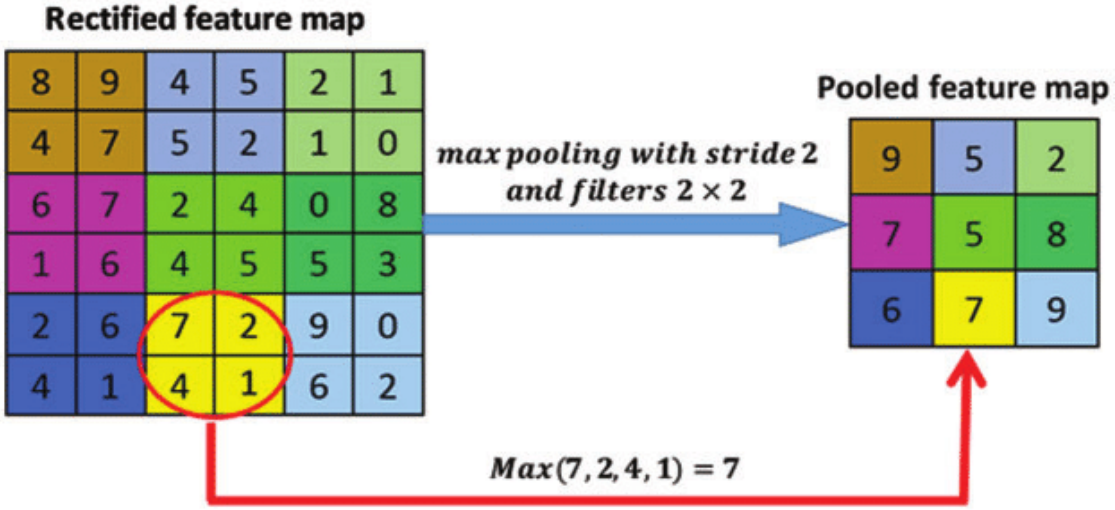


Linear filters are designed to work well with stationary noise, which is the noise that remains constant over time. If the noise we are working with is not stationary, linear filters may not be effective. Also, if there are different levels of noise in the image, a fixed-size kernel won't adapt to the context. Moreover, they are designed with the assumption that the noise is Gaussian, so they won't work well with non-Gaussian noise, such as salt-and-pepper noise.

If we wanted to remove "salt and pepper" noise, these linear filters could try to do the job, but non-linear filters such as median filters would be more suitable. "Salt and pepper noise" is a type of noise which is random, isolated pixels that are either completely black (pepper) or white (salt). For example, a box filter could help to some extent to reduce this noise, but it may also blur the image.

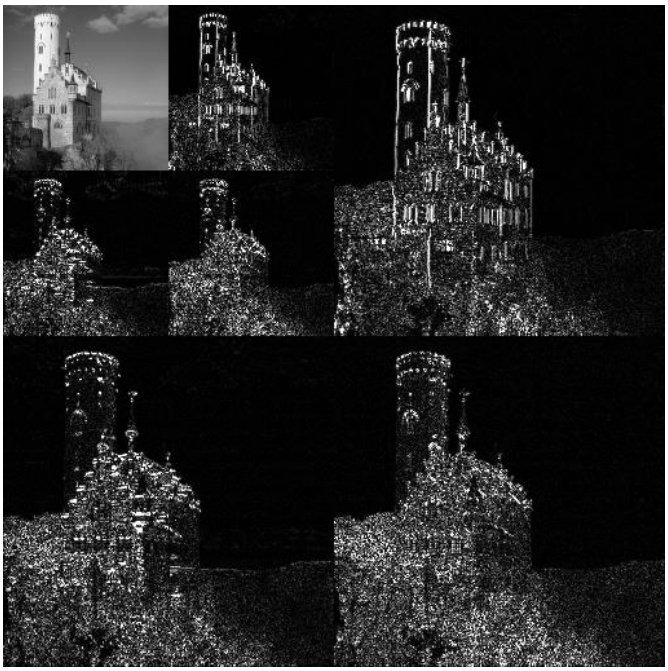
Apart from applying these filters using convolution, there are other methods we can use:

1. Pooling: a filter reduces the resolution of the image by selecting maximum or average value in a specific region. It is used for down sampling. It reduces complexity and memory usage, but there is also a loss of spatial information.



2. Filter Response: a filter is applied to an image and we measure its response to each pixel. This allows us to capture specific features, so it's mainly use for feature extraction. However, it is computationally expensive for certain filters.

3. Wavelet Transform: it decomposes an image into different frequency components using wavelets. This method can be useful to do a multiscale representation of an image or for feature extraction and compression. However, it is complex to implement and it may not be as intuitive as spatial filtering.



3 TEMPLATE MATCHING

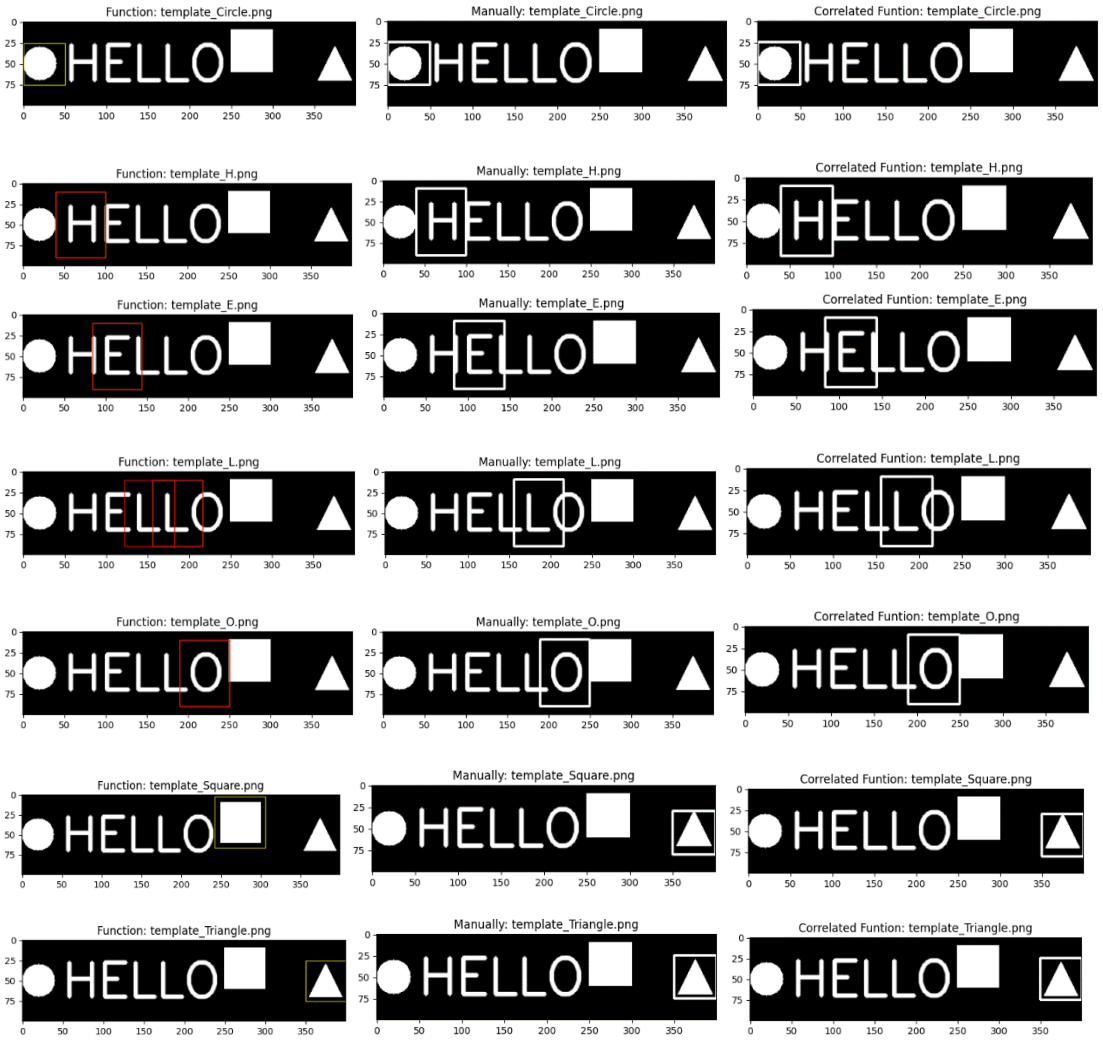
Template matching is a technique used to find instances of a template (a small image) within a larger image. The process involves sliding the template over the search image and calculating the normalized cross-correlation at each position. The position with the highest normalized cross-correlation represents the best match, indicating the location of the template within the image.

Before starting the template matching process, both the image and the template must be normalized, which involves subtracting the mean and dividing by the standard deviation. At each position, the normalized cross-correlation is calculated, which is defined as follows:

$$NCC(f,g) = \frac{\sum_{x,y}(f(x,y)-\mu_f)(g(x,y)-\mu_g)}{\sqrt{\sum_{x,y}(f(x,y)-\mu_f)^2 \sum_{x,y}(g(x,y)-\mu_g)^2}}$$

Where: *f(x,y)* and *g(x,y)* are the pixel intensities at position *(x,y)* in image *f* and template *g* respectively. μf is the mean pixel intensity of image *f*; and μg is the mean pixel intensity of template *g*. As the template slides over the image, a match score map is actualized with the results of each position.

We have had several difficulties applying this method into our code, but here are some results. We tried 3 different approaches: using library functions (column 1), doing it manually (column 2) and finally combining manual operations with using implemented cross-correlation functions. In our results, we can observe there where some problems with the square template and the "L" letters, since our manual implementations had trouble detecting them.



4 FOURIER TRANSFORM

The Fourier Transform is used to represent an image in terms of its frequency components by breaking down the image into a combination of different sinusoidal (with waveform) components with different frequencies and amplitudes. By examining these frequency components we can detect and isolate important features, like edges, textures and specific patterns. It can also be used to apply filters in the frequency domain to be able to remove noise, enhance details or perform sharpening. We could also use Fourier Transform for data compression without losing essential information.

In Fourier Transform, convolution is simplified in the frequency domain as a simple multiplication, which makes the operation computationally more efficient:

$$F(f * g) = F(f) \times F(g)$$

We might choose to operate in the frequency domain instead of the spatial domain when processing an image because of several advantages:

1. We are able to examine low-frequency components (representing smooth areas or large-scale patterns) and high-frequency components (fine details or edges).
2. Since noise in images often appears as a high-frequency component, we can reduce it while preserving important image information.
3. Deconvolution (reversing the effects of convolution for image restoration) is more manageable in the frequency domain.

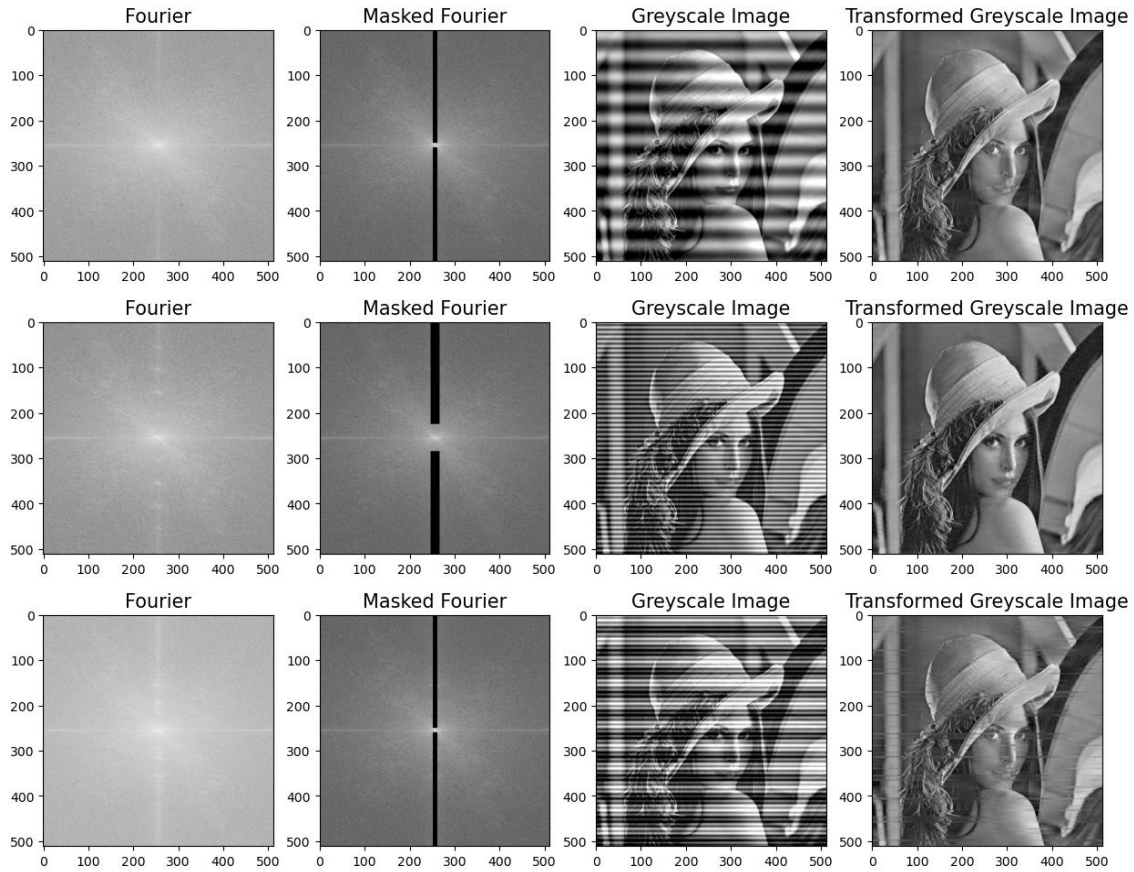
There are two types of Fourier Transform: discrete FT and fast FT:

1. Discrete Fourier Transform (DFT) converts a discrete sequence of N complex numbers into another sequence of complex numbers, representing the signal's frequency components. It has a complexity of O(N^2), which makes it slow for large inputs. Therefore, DFT is used when computational efficiency is not required, but instead we want a high accuracy in frequency.

2. Fast Fourier Transform (FFT) takes advantage of the symmetry and periodicity properties of sinusoidal functions to make it more efficient, having a complexity of O(N log N). It is used in real-time signal processing, audio and image compression, communication systems… where processing speed is important.

As we can see, Fourier Transform is used in various fields outside of image processing. For example, speech and audio processing: FT is used to analyze the frequency components of spoken words to identify phonemes and words, and also for audio effects, like reverb and equalization, that involves modifying the frequency content of audio signals.

In our code we had to clean three images that had been corrupted by one ore more periodic frequencies. We did this by applying Fourier to the image (1) and then creating a mask (2) that returned a more clean look of the image (4). This are our results:



For this part of the code we had a bit of trouble finding the correct mask to apply to each image. At first we tried to create a function that, depending on the magnitude spectrum of each image, created a specific mask for that image. However, after several failed tries to implement this function, we went safe and adjusted the values of the mask manually. For the first and third image we obtained medium good results with the same configuration, but for the second image we used different values.

5 EXTRA TOPIC: WORKING WITH COLOR IMAGES (RGB)

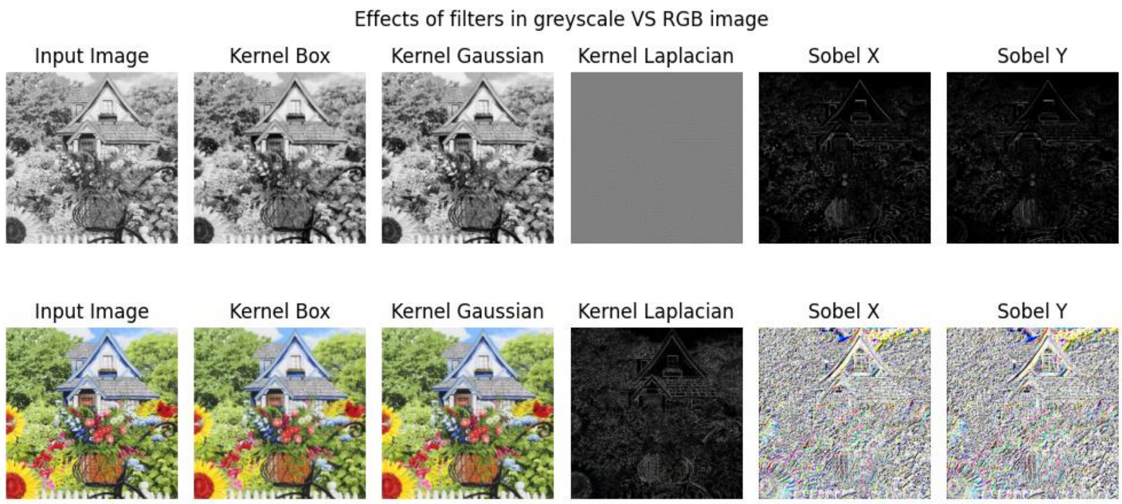
On this lab we were told to work with greyscale images because the convolution operation is more simple. However, we wanted to take a step forward and try making a function to compute convolution on RGB images too.

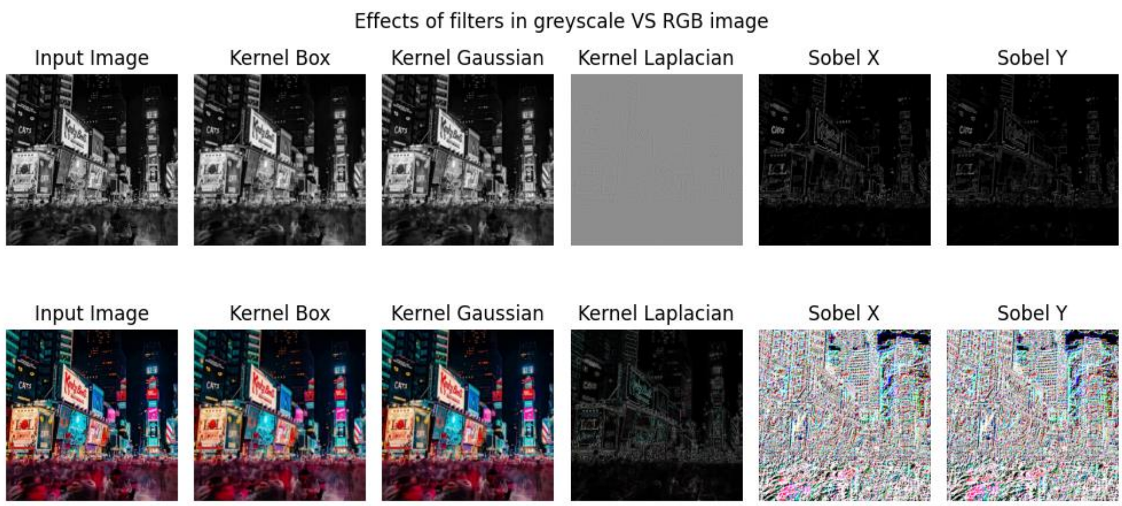
How is convolution different in color images? Basically, the operation works the same way, but we need to apply it to each color channel separately (R,G and B) and then combine the results to produce a final output. Dealing with multiples channels increases the computational cost of the operations because the input are the 3 channels, but you might have multiple output channels.

To show our results working with color images we added a section on the code called *"Working with color images"*. We first had to convert the original color images to RGB representation for a better visualization. And after that, we implemented different functions to play with those images.

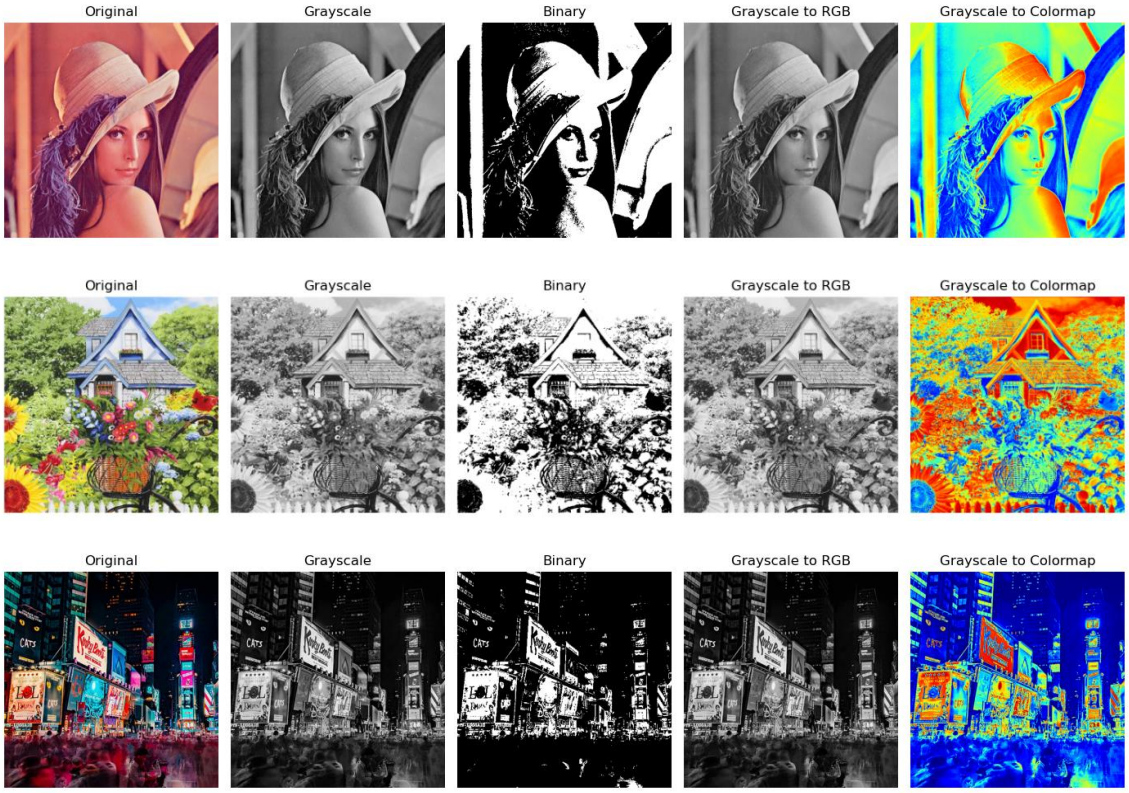
Firstly, we wanted to observe how convolution worked with RGB images and compare the results using convolution with different linear filters on greyscale and RGB images.


Effects of filters in greyscale VS RGB image
Input Image
Kernel Box
Kernel Gaussian
Kernel Laplacian
Sobel X
Sobel Y
Input Image
Kernel Box
Kernel Gaussian
Kernel Laplacian
Sobel X
Sobel Y

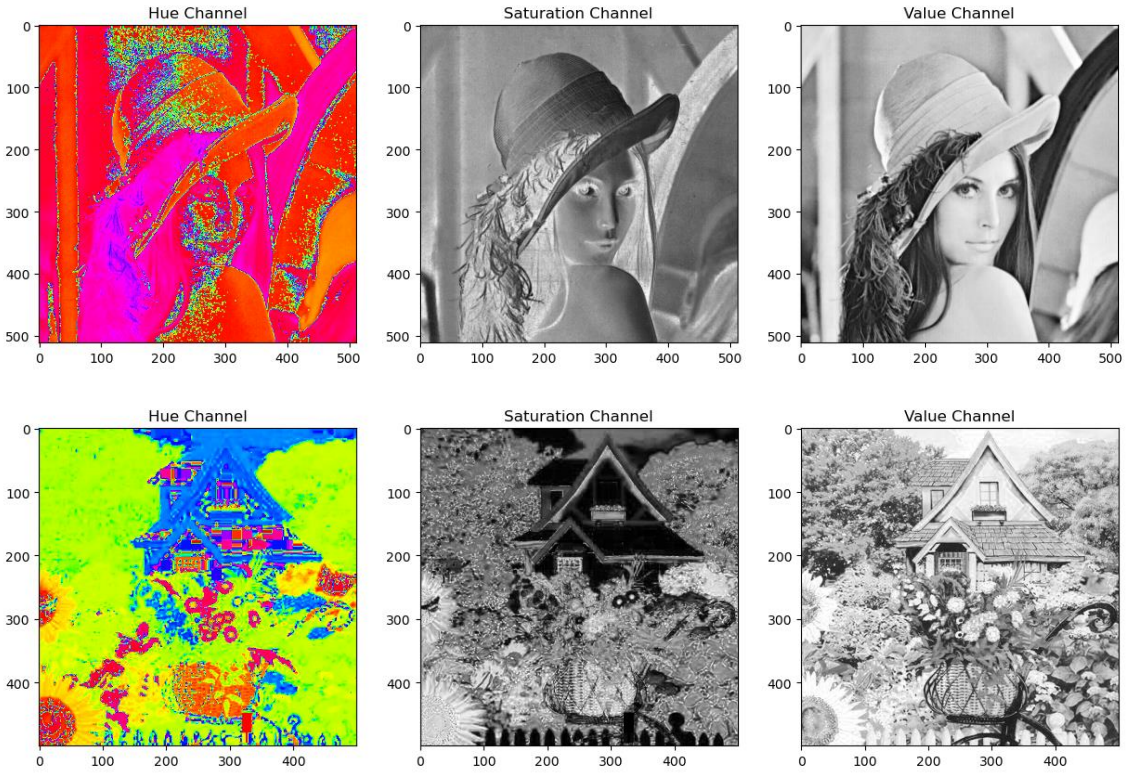

Effects of filters in greyscale VS RGB image
Input Image
Kernel Box
Kernel Gaussian
Kernel Laplacian
Sobel X
Sobel Y
Input Image
Kernel Box
Kernel Gaussian
Kernel Laplacian
Sobel X
Sobel Y

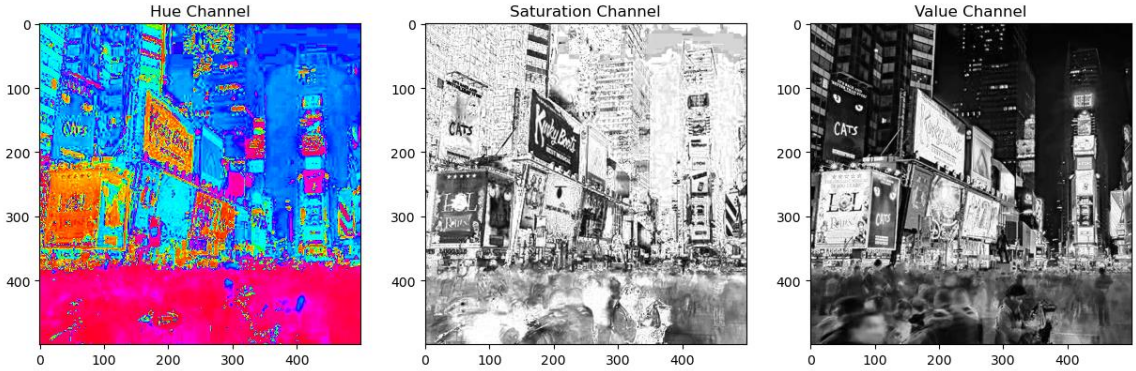

Effects of filters in greyscale VS RGB image
Input Image
Kernel Box
Kernel Gaussian
Kernel Laplacian
Sobel X
Sobel Y
Input Image
Kernel Box
Kernel Gaussian
Kernel Laplacian
Sobel X
Sobel Y

Secondly, we wanted to see the color image represented in different color typer: grayscale, binary, grayscale to RGB and grayscale to colormap.

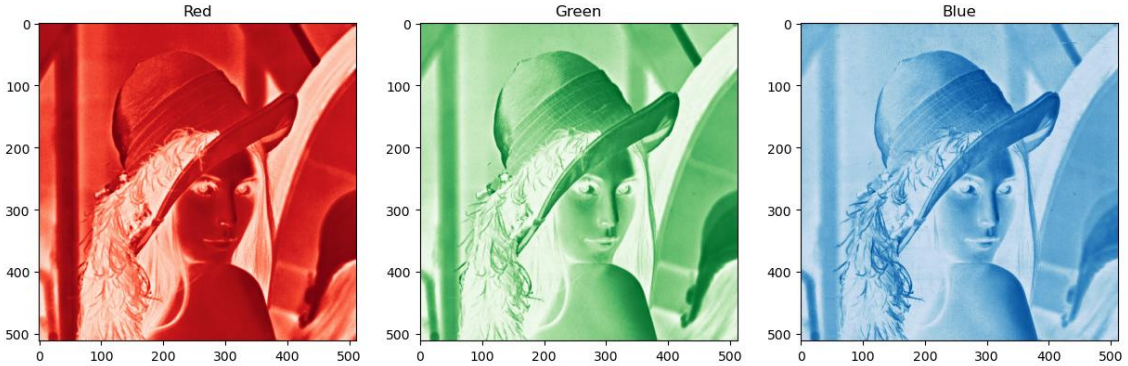


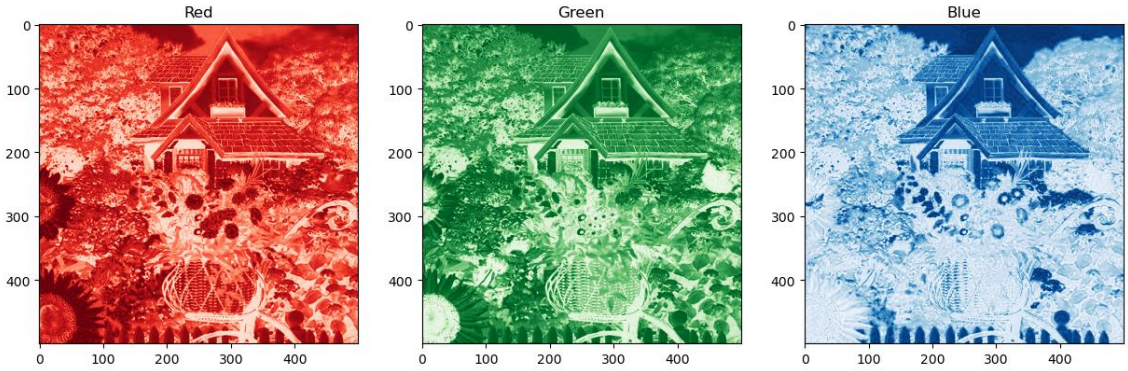
Then, we wanted to show HSV channels of the images:

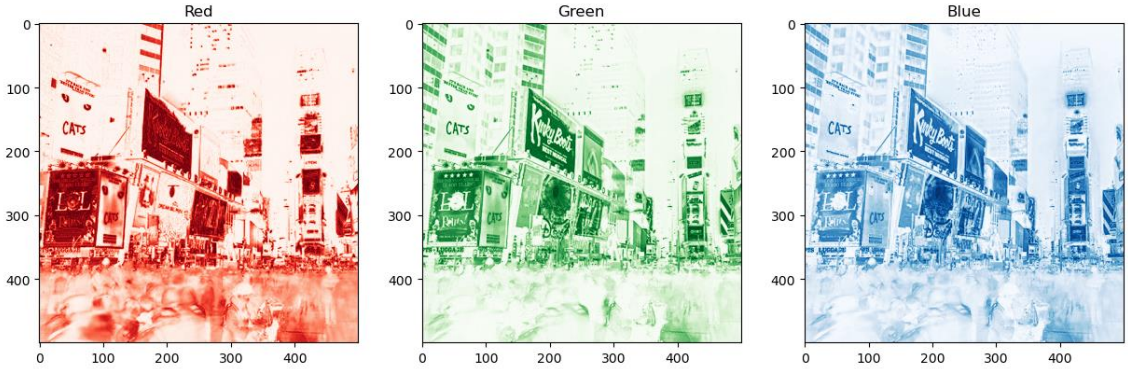




Finally, let's observe separately the Red, Green and Blue channels:







6 CONCLUSIONS

While doing this lab we have been able to understand how convolution works better while working with images, since we had to implement it step by step. We also realized the importance of choosing a kernel when applying convolution, since different kernels can provoke very diverse changes in an image, from obtaining a white image to a black one (it was the case when we applied Filter2D function).

Nevertheless, we also need to emphasize that for us Template Matching coding was the most difficult part of the lab, at the end we understood the process of how a smaller template is found in a bigger image. However, we have been unable to get our expected and desired results even after trying it. There are some problems with finding the geometrical figures and the elements which are represented twice (L).

Finally, we were asked to apply the Fourier Transform to clean different corrupted frequencies we realized how making modifications in the frequency domain worked and what are the advantages that could have working in that domain compared to the spatial domain.

All those concepts that we have treated have given us the chance to prove by doing how some mathematical concepts are related with image processing. Besides, we have found out how important is the structure of a code so it will be easier to interpret and compute, the better the less time and memory you will have to waste.

This lab has given us the opportunity to put into practice some important concepts explained during the course and also the communication and organization needed to elaborate a good Lab and Report.