Zakkarija Micallef

Zakkarija.micallef.18@um.edu.mt

Principles of Computer Vision for AI

Assignment Lab Report

Table of Contents

[Principles of Computer Vision for AI 0](#_Toc42805676)

[Assignment Lab Report 0](#_Toc42805677)

[Point Processing – Tutorial 1 2](#_Toc42805678)

[Area Processing – Tutorial 2 5](#_Toc42805679)

[Morphology – Tutorial 3 8](#_Toc42805680)

[Feature Detectors – Tutorial 4 12](#_Toc42805681)

# Point Processing – Tutorial 1

**Abstract**

The main aim of these exercises is to learn about point processing and applying functions such as thresholding, power transform

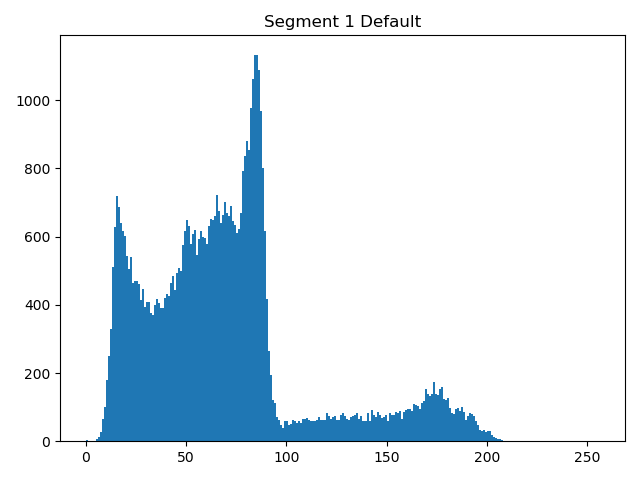
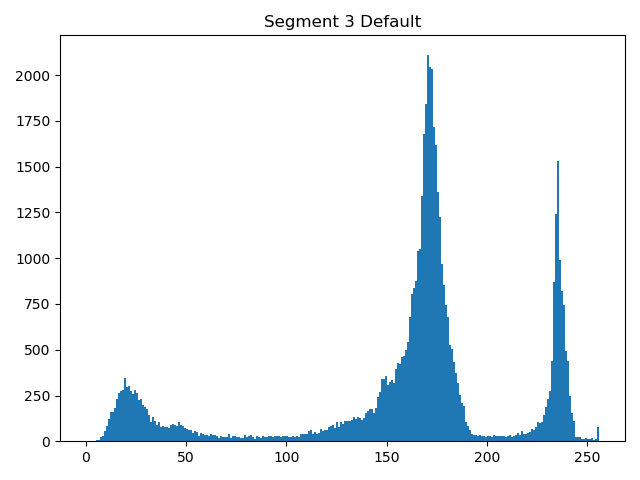
**Method**

1. Initially the read image had to be segmented into 4 section. This was achieved by reading only ¼ of the section of the array that represents the image. Each image section was given its own 2D array variable. When each of these variables are shown separately through a function such as *cv2.imshow(),* 1/4th of the image issuccessfully outputted.
2. A thresholding function had to be applied to every section to identify if it’s a dark or a light region. The function *isThreshold()* returns a Boolean value, depending if the region is dark or light. The function basically takes the images section’s mean value using the NumPy’s mean function and compares it to an threshold value that is set to 127 (Chosen since it returned half of the mage as light and the other dark, ideal for the upcoming exercises). Another threshold() function is present that returns a binary image where all pixel values that are above a certain threshold value are set to 1.0 and the others to 0. By comparing the image to the threshold value using the ‘>’ operator, it returns an array of the same shape that is 1 where values are more than specified and 0 otherwise.
3. Wherever an operation on a image is done, there will be a call to the *histogram()* function which accepts the title of the histogram and the image that the histogram will be based on. *cv2.calchist()* is used to calculate the histogram and is plotted using the *matplotlib.pyplot.plot* function and using it’s show function it displays all figures.[ <https://docs.opencv.org/2.4/modules/imgproc/doc/histograms.html>][ <https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.hist.html>]
4. Anywhere where there was a light/dark segment a power transform has to be performed to make it lighter/darker. This was done by varying the gamma of the image segment. This was done by taking the image as an array and applying an adjustment formula on the image as the array object and the desires data type was set to ‘uint8’. The gamma and image are sent as parameters of the function so they could be varied. [notes]
5. In bit-plane slicing the image is divided into bit planes. This is done by first converting the pixel values in the binary form and then dividing it into bit planes. Each pixel in the image was iterated over, converted to an 8-bit binary representation using *np.binary\_repr* and append to a 1D array. From an 8-bit array we can get the 8 bit planes. To get the nth bit plane, the 1D array is iterated over and from each element the nth value is taken and converted from it’s binary value to its decimal value. For e.g. a random element in the 1D Array [01101101], by taking the second value ‘1’ it is converted to it’s decimal value which is 2. By doing this to every array we end up with a 1D array that is reshaped to a 2d Array. For ease of viewing, all these image are concatenated to the same image window using *cv2.hconcat* and *cv2.hconcat* [<https://theailearner.com/2019/01/25/bit-plane-slicing/>]
6. The application was transformed to video simply by taking as an input the webcam using cap = cv2.VideoCapture(0) method and an infinite while loop consistently read the capture.

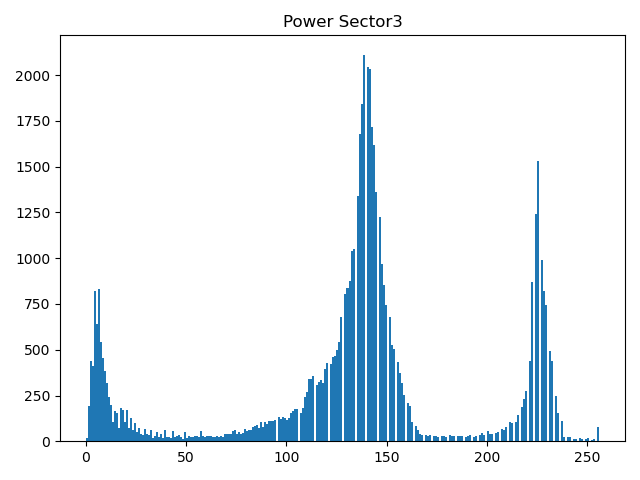
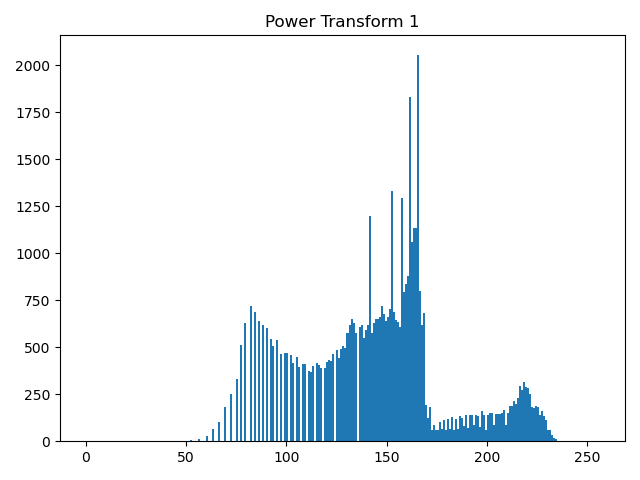
**Data and Calculation**

In this section we will be looking at the histograms and outputs of different image segments and operations.

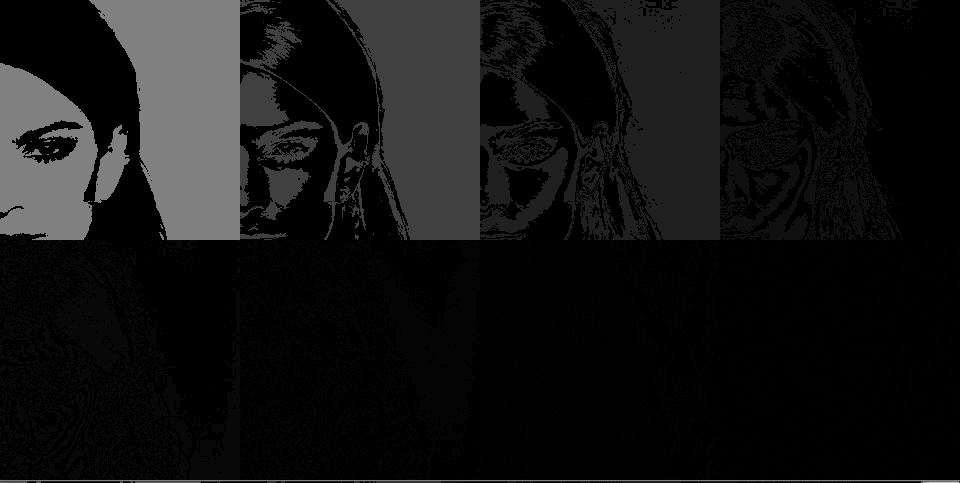
Default Image Histogram:

Power Transformed Images Hsitograms



Sector 4 Bitplane (top right)



**Discussion of Results**

As we can see when we look at the histograms of segment 1 and segment 3 is a difference of where the peak of the graphs is. Segment 1 has a more pixel values focuses around the lower values area as compared to segment 3 meaning that the image would be darker overall. This is further confirmed when these segments are sent to the *isTreshold* function as previously mentioned. So, when the power transform function is applied to the image, the segment 1 image can be seen to get lighter and segment 3 darker, so now both images have a fairly central pixel value peak in the histogram.

In the next figure we can see the 8 images from the bitwise splicing concatenated together. The first image has the most detail and it keeps getting darker and the further away from the original image. Since the first image is made up of the MSB, not as much information is lost as the next images where it made up of binary values that hold less value as more information is lost.

[<https://theailearner.com/2019/01/25/bit-plane-slicing/>]

# Area Processing – Tutorial 2

**Abstract**

The purpose of this experiment was to preform convolution on images with kernels using sliding windows.

**Method**

1. 2 functions are used in order to slide through the image. A helper function *slidingWindow* which actually loops (slide) through the pixels in the image and the *slide()* function which calls the *slidingWindow()* in an iterative for loop, each time showing the output that is given. The *slide()* function accepts 3 parameters: the image, the n\*n window size and finally the stride length. The rectangle chich shows where the sliding is is done by using the *cv2.rectangle* function.[<https://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/>]
2. This function takes an n\*n region of the image starting from (0,0) and takes it, iterates over it and preforms convolution with a predefined Sobel kernel (3x3 2D-Arrays). The RIO part of the image is taken by taking that part of the NumPy array of the image as done in the previous tutorial. An all-zero image was created in the size of the ROI to house the final Sobel convoluted image. Every pixel in the region is looped using two nested for loops. For each pixel, it and its immediate neighbours (3x3) is taken and convoluted by the Sobel kernel. Initially convoluted with the x-axis Sobel Kernel, then with the y-axis Sobel kernel. In order to convolute them, the kernels are multiplied, and each element of the resulting matrix is summed up for a final pixel value in x and y directions. The Pythagoras theorem is then applied to these values to get the final magnitude. The final pixel value is finally added to the all-zero 2D NumPy array created before. This array is finally plotted alongside the original image using *matplotlib.pyplot* function.
3. This one is very similar to the previous one but instead the ROI is the entire image. Two functions are provided, one that just displays the entire completed Sobel image along with a graph that shows the histogram of the Sobel image compared with the histogram of the original image, and one which shows the sliding windows while the Sobel convolution is being performed. There is an issue with how OpenCV shows the output of the Sobel. In the previous question *matplotlib.pyplot* was used instead.
4. The *slidingWindow()* was used in order to iterate over the full image exactly as the previous excercies but instead of applying the *Sobel* , the *cv2.gaussianBlur* function was used. a *makeGaussian*() function was also written but was not used. The cv2 function convolves the image with a Gaussian kernel. A great advantage of the gaussian blur is that it preserves well-defined edges. This is done by having the centre value of the kernel having a higher weight than the ones in the edges, so near an edge there won’t be a lot of data taken from a side of the edge and being applied to the other.

**Data and Calculation**

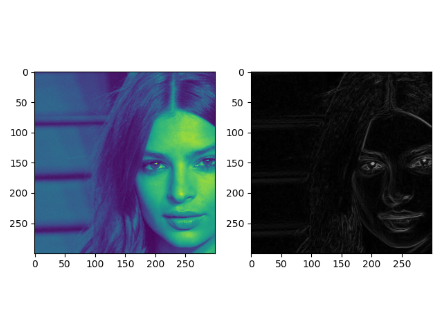


Figure 1

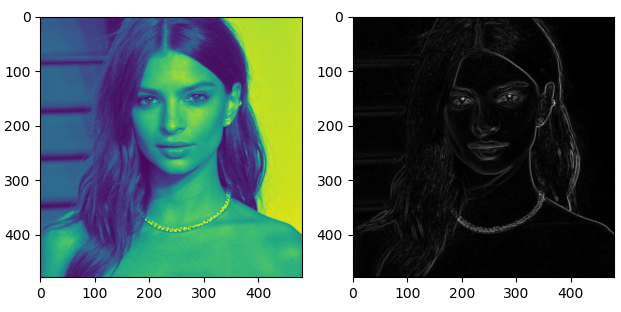


Figure 2

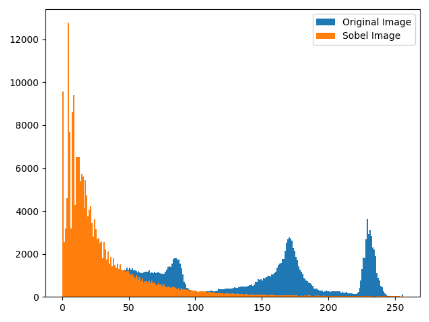


Figure 3

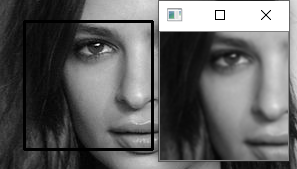


Figure 4

**Discussion of Results**

In figure 1, we can see that a specified part of the image (the Region of interest starting from 0,0 to 300,300) on the right has the Sobel kernel applied to it. We can see how most of the edges are in white as the Sobel kernel works by taking wherever there is a sharp change in intensity and would detects it as an edge. This is done as explained previously in the method section part 2. Figure 2 is exactly the same method as figure 1 but instead it is done on the entire image instead of the specified region of interest. Looking at the intensity histogram for the Sobel filtered image obtained from *matplotlib.pyplot*, in figure 3, most pixels are centred around an low intensity, mostly because it’s a rather simple image with a relatively uniform colour palette without a lot of hard edges. Figure 4 shows a section of the image while the gaussian filter was being applied resulting in a blurred section.

# Morphology – Tutorial 3

**Abstract**

The purpose of this experiment was to preform mathematical morphology on different types of images.

**Method**

1. For *cv2.connectedComponets()* function to work, the image must be binary and the objects should be white so the Inverted Binary threshold function along with Otsu's Binarization was applied to the image using *cv2.treshhold* function. *cv2.connectedComponenets* functionis called on the threshold image which returns the number of components found and a return value. A for loop iterates for every component returned and the mask is outputted. The mask is gotten from the return value of the connected components function.
2. In order to dilate the image, the cv2 *dilate()* function was called. The function accepted three parameters: the image itself, the kernel, which is a NumPy 2D Array of ones and the finally the number of iterations the dilation will take place on the image. Dilation usually consists of convoluting an image A with some kernel (B), which can have any shape or size, usually a square or circle. Kernel B has a defined anchor point, usually the centre. When scanning over the image, the maximal pixel value in the current scan ‘frame’ is replaced by the ancho point of kernel B. This causes the image to grow (dilate).[ <https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html>]
3. The erosion procedure was exactly the same as the dilation but instead the *cv2.erode()* function was used.
4. To achieve opening, firstly erosion was executed flowed exactly by dilation on the same eroded image.

**Data and Calculation**

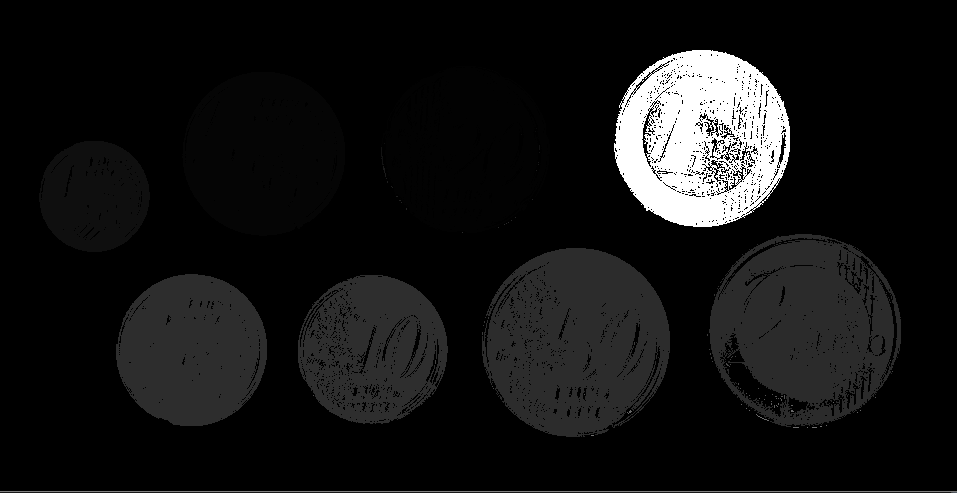


Figure 5

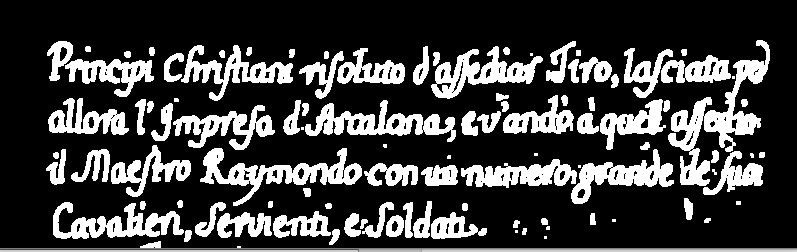


Figure 6

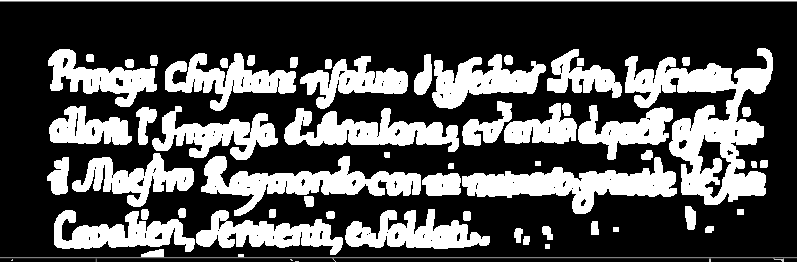
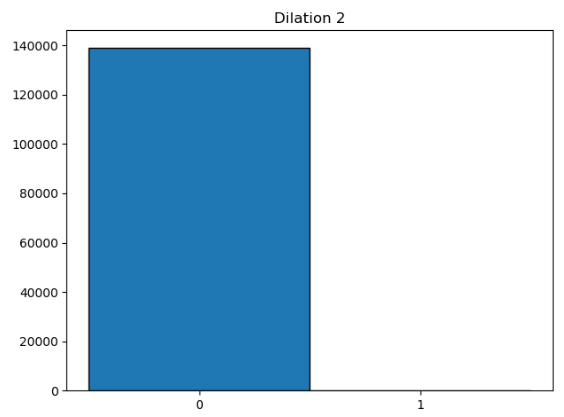
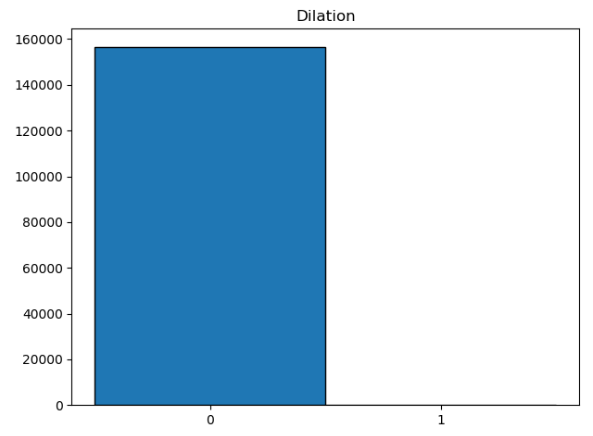
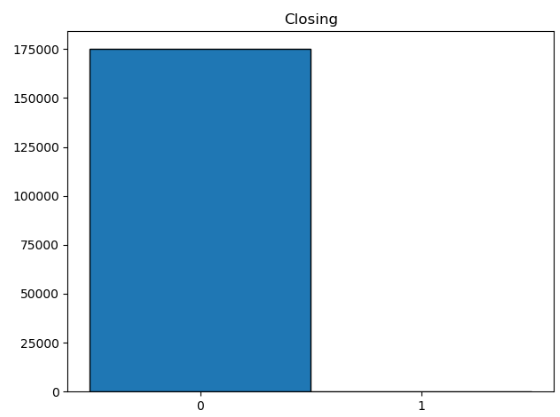
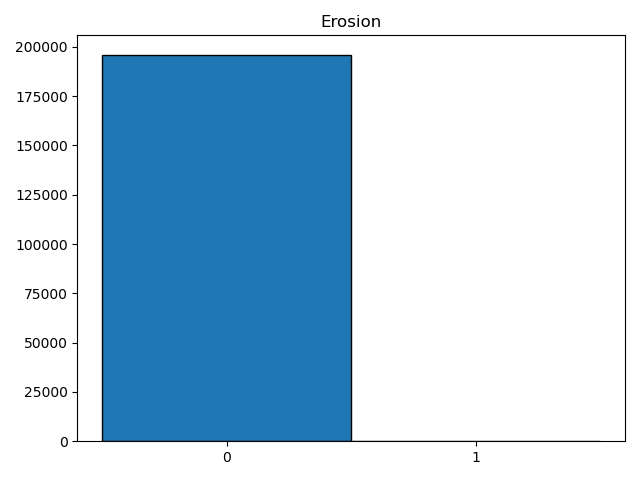
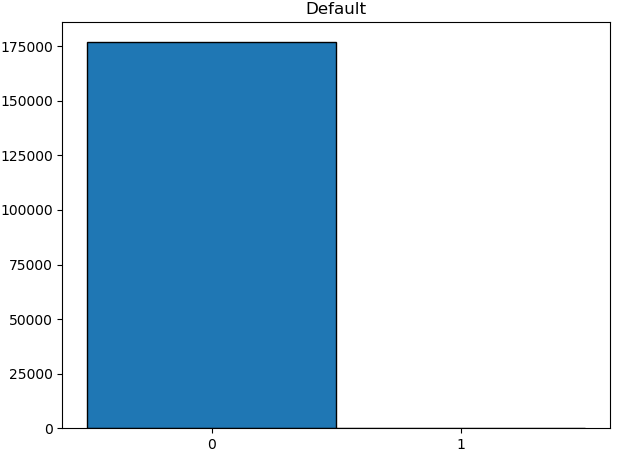


Figure 7







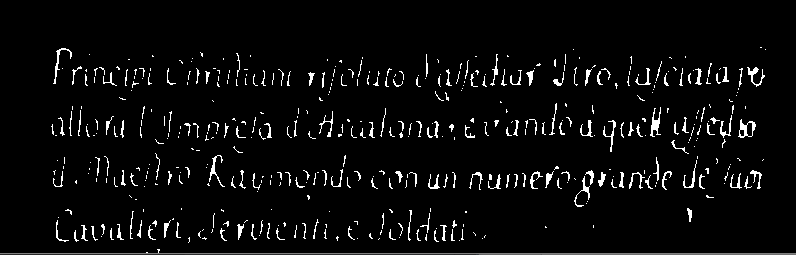


Figure 8

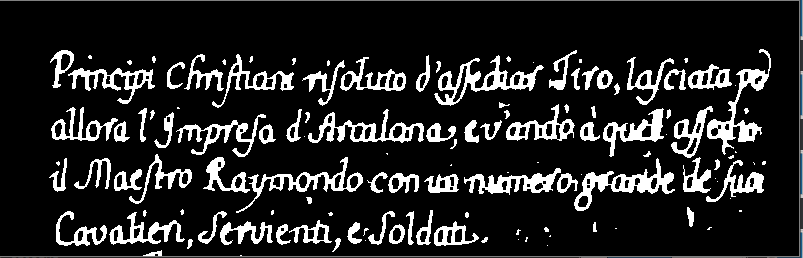


Figure 9

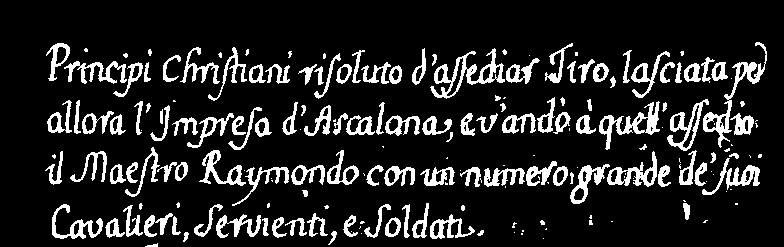


Figure 10

**Discussion of Results**

In the above figure, figure 10, is out invert binary image. Below figure 7, we have 2 histograms, the one on the left representing the text image with one iteration of dilation and the other with two iterations. The image that had one dilation has just below 16,000 black pixels, while the one with increased dilation had just below 14,000 black pixels. This can clearly be seen in the images. When the dilate function was called on it, the text can be seen to get thicker, see figure 6. On figure 7, the dilation function was iterated twice over the image, growing larger, hence fewer black pixels. This can also be also be seen when comparing the normal image with the eroded one which has the most black pixels at around 20,000, see figure 8. The opening was achieved by initially dilating the text then closing it. Due to its nature it will be the closest to the original image, with very minor alterations, mainly losing minute data. This can be seen in their histograms, were both have around the same number of black pixel values, with the default having slightly more. If a close look is taken at the differences between figure 10 and figure 9, wherever there are small gaps, they were filled, and the jagged outer part of the letters became slightly smoother.

# Feature Detectors – Tutorial 4

**Abstract**

The purpose of this experiment was to preform corner detection, feature detection and feature matching while researching about the differences about the different methods used to achieve this.

**Method**

1. The Corner Harris detector is used for corner detection. The *cv2.cornerHarris()* function accepts 4 parameters: the image that must be in greyscale and float32 type, (blockSize) the size of the neighbourhood for corner detection, (kSize) the aperture of the Sobel derivative used and k(Harris detector free parameter in the equation). [<https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html>] Corner Harris simple pseudocode
   1. Convert image to greyscale
   2. Apply Gaussian Filter to image for noise smoothening
   3. Apply Sobel operator for x and y gradient values
   4. Compute Harris Value of every pixel (Compute corner strength function of 3x3 neighbourhood of a pixel)
   5. Compute a feature descriptor for every pixel that exceed a certain threshold and are the local maxima within a certain window.
2. The Shi-Tomasi Corner detector is similar to the Corner Harris detector but instead of using R = \lambda_1 \lambda_2 - k(\lambda_1+\lambda_2)^2 it uses R = min(\lambda_1, \lambda_2) where if it’s greater that a threshold value, it’s considered a corner. The *cv2. goodFeaturesToTrack()* function was used since it uses the Shi-Tomasi detector. This function accepts: the image in greyscale, number of corners to find, the minimum quality threshold level of corners to find and the minimum Euclidean distance between the detected corners. The function returns the corners in an order, highest quality first. Using the *cv2.circle()* function, the dots are drawn onto the corners. [<https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_shi_tomasi/py_shi_tomasi.html>]
3. While corner Harris is rotation-invariant meaning, when an image is rotating the same corners can be found, it is not scale invariant. This is where SIFT comes in. In order to run Sift, the algorithm has to be loaded using *sift = cv2.xfeatures2d.SIFT\_create().* Using the *sift.detectAndCompute()* function, it will return key points and descriptors. A key point is a position where a feature (such as a corner) has been detected, while a descriptor is an array describing the features. The keypoints are then drawn on the image using *cv2.drawMatches()*. [<https://pysource.com/2018/03/21/feature-detection-sift-surf-obr-opencv-3-4-with-python-3-tutorial-25/>]
4. The procedure for the surf and Orb is exactly the same as described previously for the sift. **Surf** is an overall faster implementation than Sift, achieved by using different techniques [<https://medium.com/@shehan.a.perera/a-comparison-of-sift-surf-and-orb-333d64bcaaea>]
5. **ORB**is the fastest algorithms of the three. [<https://medium.com/@shehan.a.perera/a-comparison-of-sift-surf-and-orb-333d64bcaaea>]

The Brute Force Matchers takes two images and tries to find common similar features between them. The BFMatcher function takes and compares the similarities between two separate descriptors. To initialize a BFMatcher, two parameters are required: the Norm Type and the cross check Boolean (false by default). If the cross check is true it will return fewer but higher quality features. The *match*() accepts the two descriptions and returns the matches. The matches are then drawn using the *cv2.drawMatches* function. Finally the matching results are outputted.



Figure 11

Figure 12

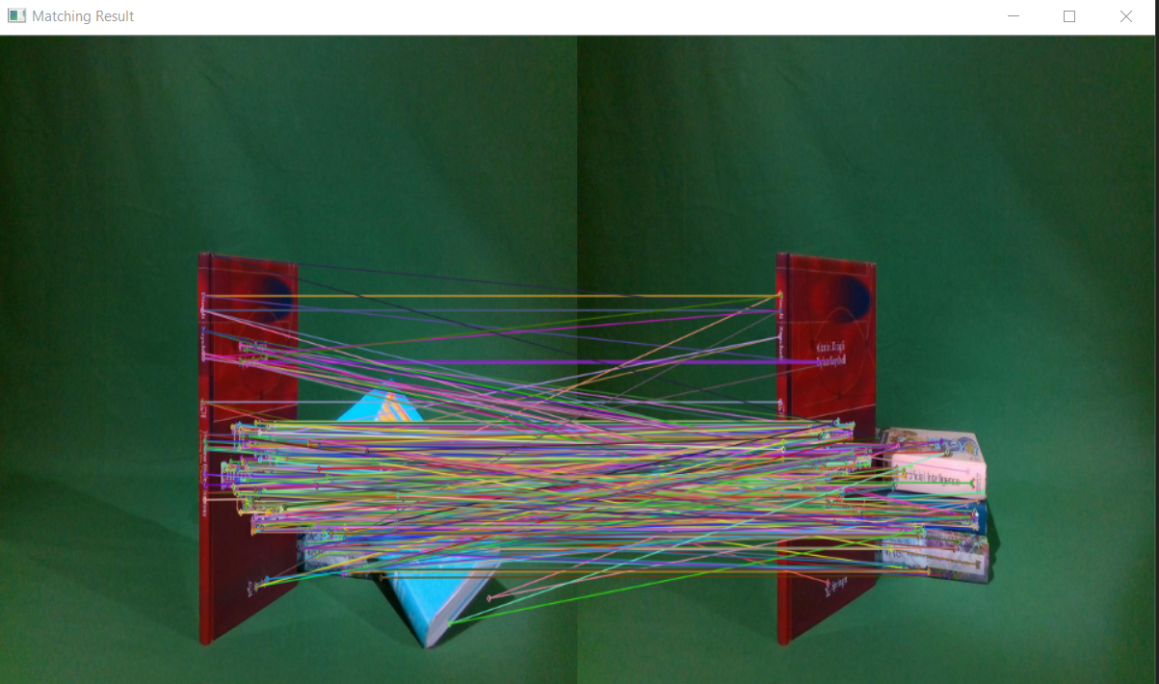
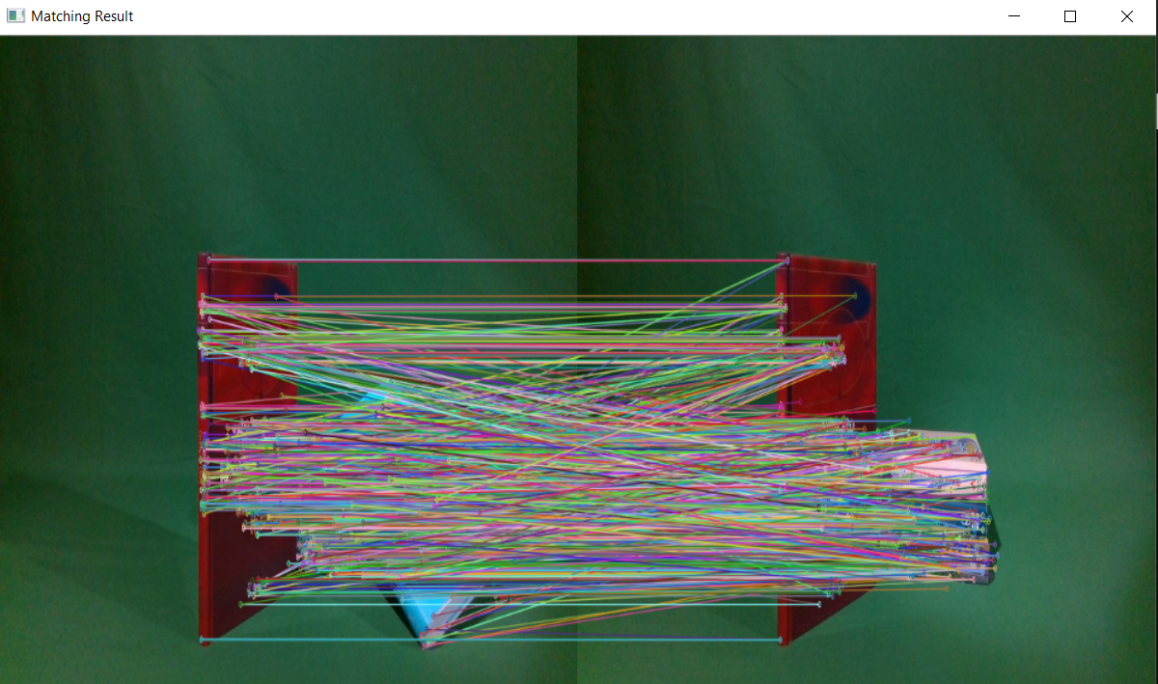
****

Figure 13

Figure 14

Figure 15

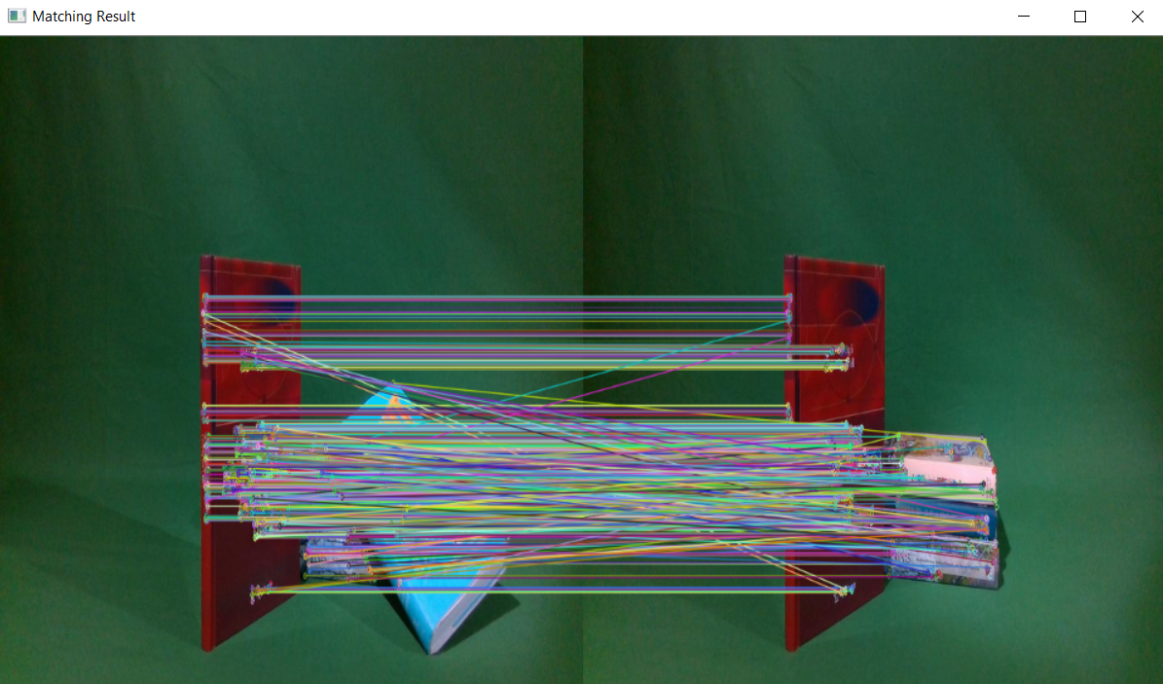
****

Figure 16

**Discussion of Results**

In figure 11 we can see the results of the Corner Harris Detector, where it mostly detected the edges of the titles of the books. In both figure 12 and figure 15, the Shi-Tomasi corner detector was used. In figure 15 it was limited to about 50 features and as can be seen it took the most prevalent corners in, the largest title with the highest contrast between book colour and title colour. While in figure 12, it was ‘limited’ to 500 features, and we can see most of the extra features it was going to add were similar to the ones already found by the corner Harris detector. From figure 13 to 15 we can see the different feature detectors in order: sift, surf and orb detecting the similar features between two similar images. All the detectors made a feature matches with parts of the image that were occluded in the other image, mainly the lower half of the stacked books. This could mostly be down to that the books are fairly uniform in their features such as colours, titles, etc along their lengths.