1. Introduction

The ability to adequately process and analyse image data is of the essence of machine and computer vision, particularly in contexts where autonomous machine perception and perhaps artificial conceptualisation is patently required. On this basis, the applications for such vary extensively, from the automation of machinery in the industrial sector, to contemporary research regarding complex artificial intelligence systems. Moreover, the ability to process and analyse digital image data is paramount in contexts which require human interpretation of image data. Sufficient image refinement, enhancement or manipulation is a necessity for extracting relevant data within these human-machine interfaces. Such applications in this context are seen ubiquitously in all sectors. Some notable examples include: Medical imaging such as MRI/X-ray imagery, weather forecasting and UV sensing, encoding and transmission of image data and even mapping extensive topological surfaces on neighbouring planets [1][2]. So, conclusively speaking, one could make the assertion that digital image processing is an irrefutable linchpin in the symbiotic evolution of man and machine.

In this document, I explore some of the rudimentary techniques associated with digital image processing and analysis. The input images were provided by the lecturer preparatory to the conduction of the assignment, with the exception of one input image, which was provided by myself. This project was conducted via the Jupyter notebook (v.6.3.0) IDE and most notably, in conjunction with the Scikit-image (v.0.18.3) library [3]. The Sci-kit image library provides all the relevant algorithms required to process the input images. The project was segmented into three parts. with the first part involving rudimentary resizing, converting and filtering techniques of a raw input image provided by myself. Noise was added to the image which then relevant filters were applied to remove the supplemented noise. The second part involved creating a ***robust, fully automated*** and ***data-driven*** code to extract the smallest scissors from the input image. A series of techniques were employed in order to reduce the input image to a binary image, clear the borders and detect the relevant objects (scissors) within the image, in order to compute their relative centroids and areas and extract the smallest one. Once completed, a stress test demonstrating the codes ***'robustness'*** was created by implementing varying degrees of noise. The third and final part of the project involved colour matching of a reference image provided by the lecturer with a source image, which was no other than the input image provided in part. 1. The histograms of each image were plotted and analysed. Explicit detail of each procedure is undertaken in sec. 4.

2. Background and Theory

This section presents a brief insight upon the rudimentary principles regarding image processing that have been incorporated throughout the project.

2.1. Fundamental image representation

Once the image has been adequately acquired and formed, the image intensity data must be represented in such a way that it can be processed by various means.

Grey-scale images:: One such way of representing image intensity data is by means of some function $f : \mathbb{R}^2 \to \mathbb{R}$ that outputs an intensity at some input values (x, y), which typically correspond to the position in the image. The intensity scale or white level, denoted $'W'$, of the image ranges from 0 to 255, or 0 to 1 if normalised [4]. Thus $0 \leq f(x, y) \leq$ W, which ultimately corresponds to each pixel in the image. A 2-D array of successive positional inputs (x, y) thus forms the digital image with corresponding intensities $f(x, y)$.

Coloured images:: Conversely, representing coloured images is more complex. Instead of each pixel intensity being determined by a single function $f(x, y)$, pixel intensity in coloured images are determined by a series of functions, namely $R(x, y), G(x, y), B(x, y)$, which correspond to Red, green and blue [5]. So, ultimately, unlike gray-scale images, where each pixel being represented by some integer value between 0 - 255, instead each pixel is defined by a vector of three numbers $[R, G, B]$, which each range from 0 to 255.

2.2. Filtering methods.

Mean filtering: The mean filtering procedure operates by smoothening the input image as a direct consequence of reducing the relative intensity variations between neighbouring pixels [6]. Its most commonly employed in noise reduction practices. This particular method relies on the premise of the local convolution operator familiar to functional analysis. Briefly, the convolution operator is a mathematical operator, defined by [7]:

$$O(i, j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i + k - 1, j + l - 1)K(k, l) \qquad (1)$$

whereby the input image "I" has a total of M rows and N columns, and the kernel "K" has m rows and n columns. The output image "O" will have a total of M - m + 1 rows and N - n + 1 columns. The kernel "K" is typically a local 3 x 3 grid of pixels used to calculate the relative mean for the central pixel. This operation then iterates throughout the dimensions of the entire input image.

This method essentially provides a method for multiplying two arrays together, typically with varying dimensions, that being the input image and the kernel, resulting in third output image, the mean filtered image.

Median filtering: The median filter works by similar premise to the mean filter, however, instead of calculating the mean of neighbouring pixels, it calculates the median, precisely. It operates locally by sorting all the pixel values from the central pixels surrounding neighbours in ascending order. The median is then calculated by determining the midpoint of this list, which then is subsequently attributed to the central pixel [8]. This process is then iterated sequentially throughout he input image. Comparatively, median filtering tends to preserve edge features more adequately than mean filtering during noise reduction. This may prove beneficial in some contexts.

Gaussian filtering: A two dimensional isotropic Gaussian filter is most commonly employed in image filtering techniques. Its primary purpose is to blur images, remove

detail and diminish noise. It comprises of similar mechanisms to the mean filtering techniques, that being its a convolution operator in nature, however, its kernel constitutes of a different form, namely, the 2-D Gaussian [9]:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-x^2+y^2}{2\sigma^2}} \quad (2)$$

where *x* and *y* correspond to the respective distances along the horizontal and vertical axes from the origin of the image, and σ refers to the standard deviation of the distribution. However, considering the asymptotic nature of the Gaussian in probability space, practical application of the convolution matrix, or kernel, to the input image, requires discretisation of such function. This is done by truncating the function at some n^{th} standard deviation (σ) from the mean, where any contributions past the truncation are considered negligible. Typically 3σ serves an accurate truncation parameter, as contributions from pixels past this boundary are effectively zero. The kernel is then applied sequentially to the image by the standard convolution procedure outlined in eq. 1. with each central pixel now being replaced by a weighted mean calculated by the Gaussian function, in which their respective values tend to lie more towards the value of the central pixel, unlike the mean filter, which the weighted means are uniformly calculated. The degree of 'blurring' can be varied by the value of the standard deviation. Overall, the Gaussian filter provides for a softer 'blur' of the image and better edge preservation than the mean filter.

2.3. Image segmentation methods.

Image segmentation involves the partition of the input image into one or more subgroups comprised of image objects. These segmented objects are groups of pixels each with corresponding 'labels', such that they contain identical properties as one another. This certainly proves useful when attempting to reduce the image complexity and extract relevant image data. Segmentation methods such as image tresholding, edge and region based detection methods were employed within this project.

Otsu thresholding: The basic premise of image thresholding operates by converting a grey-scale image into a binary image, by replacing each pixel with a black pixel or white pixel depending on some global threshold value $'T'$ [4][10]. If the corresponding image intensity at some arbitrary pixel $'I'$ does not succeed the threshold value, its replaced with a black pixel. Otherwise, if $'I'$ exceeds $'T'$, its replaced with a white pixel. With Otsu thresholding on the other hand, the threshold value is instead optimised by measuring the variances between the pixels that fall on either side of the threshold values, namely the background and foreground. The threshold value is iterated throughout the image such that the optimal value for $'T'$ is configured when the variance between the background and foreground pixels is at some minimum. More specifically, this minimal variance is referred to as the 'within-class variance' and can be determined by:

$$\sigma_w^2 = n_B(T)\sigma_B^2(T) + n_F(T)\sigma_F^2(T) \quad (3)$$

Whereby n_B and n_F correspond to the weighted sum of the background and foreground pixels. and similarly, σ_b^2 and σ_f^2 corresponds to the variances associated with the foreground and background pixels. So, its quite evident to see that Otsu thresholding is particularly useful for segmenting the background and foreground of an image.

2.3. Digital image Histograms.

Digital image histograms provide for useful method of displaying the distributions of pixel colours or intensities within an image. These pixel colours or intensities are binned into corresponding groups of identical values and plotted. For the sake of clarity, Intensity histograms, correspond to monochromatic images, rather than coloured images. Moreover, Intensity histograms, are plots of the relative intensities of pixel data in monochromatic images, which take values between 0 and 255. These intensity histograms prove very useful in image manipulation, such as segmenting images via thresholding or edge detection. Conversely, colour histograms provide for a distribution of the types of colours present within a digital image within the constraints of some prescribed colour range[10]. Most commonly, colour histograms are built upon the RGB colour space, which is a 3-dimensional colour space comprised of Red, Blue and Green, each of which having a range from 0 - 255, similar to the intensity histogram. Colour histograms can comprise of N dimensions, with more dimensions representing a larger spectrum. However, the number of additional colours relative to the 3-dimensional RGB space isn't very much at all, thus why RGB is ubiquitously found in most practical applications. Some specific applications in colour mapping can be seen in colour calibration of images which may be necessary prior to digitally processing the image. In some circumstances, its necessary that the input images colours are calibrated before undertaking processing techniques, such as object recognition or segmentation.

Histogram matching: Digital image histogram matching is an image transformation technique that maps the histogram of one image to the histogram of another reference image. it proves particularly useful when trying to match images that have been taken simultaneously, but by different detectors, or at different times by the same detector, since external conditions may vary, having contrasting effects on the image data.
Briefly, the algorithm works by initially computing the histograms for the source and reference images. The the cumulative distribution function for each histogram is determined: $F_{source}()$, $F_{ref}()$, which is described by the probability of some random pixel taking on some intensity value, for monochromatic sources, or tonal value for coloured sources. Mapping the histogram is completed by finding some pixel value in the source image P_{source} such that for some pixel in the reference image P_{ref}, its true that $F_{source}(P_{source}) = F_{ref}(P_{ref})$, which matches the source to the reference histogram[11]. This is then applied to each pixel on the image until the algorithm is complete.

3. Conceptualisation

This section comprises of a brief conceptualisation of each question being asked. In each of the sections, namely: *Part. 1: Filters, Part. 2: Region properties* and *part. 3: Colour matching,* Pseudo code is constructed alongside concise commentary displaying procedures being undertaken. Explicit details regarding how the code operates, in conjunction with an analysis of how its expected to operate is contrasted in section. 4.

3.1. Part 1: Filters.

a) "Acquire a colour image of your own face (front facing). Develop a programme to resize it to 512x512 pixels. Convert this image to greyscale and display your result. Find and display the minimum, maximum and mean greyscale values in this image."

We are required to import and read an image into our IDE which is stored in some local folder within our working directory. We can attribute any arbitrary name to this folder, but for simplicity, it will be called 'images'. We then utilise the Sci-kit image packages, which are scripts which enable us to conveniently resize the image and convert it to greyscale and display it as desired.

Figure 1. Part 1. a) Pseudocode

```
import all relevant packages

Work directory ="Enter path of directory where"
"-image folder is stored"

Image directory = "Create an image directory"

Raw image = "Read in input image"

Resize function (Raw image):
    resize image = resize(Raw image, 512 x 512)
    return resize image

Greyscale function (Resized image)
    greyscale image = rgb2gray(Resized image)
    return greyscale image

Resized = Resize funtion (Raw image)
Greyscale = Greyscale function (Resized)

'plot the image data with matplotlib'
```

Each sci-kit process (resize, rgb2gray) is embedded with in a defined python function, with each being declared at the end. The argument for the resize function is simply the raw input image data from the image folder, whilst the greyscale image function is a composite function of the resize image function. Results can be then configured on a plot and displayed using matplotlib.

b) "Taking the output image from part (a) and add Gaussian noise (mean=0.0, variance=0.01). Display your result."

Utilising the same packages and input files from our previously established work directory, we apply Gaussian noise to the resized, greyscale output image obtained from the previous section "part a". The resulting image is then plotted using matplotlib, where it can be qualitatively analysed.

Figure 2. Part 1. b) Pseudocode

```
Gaussian(output image , variance, mean)
    noise = random_noise(output image,
                         variance, mean)
    return noise

Gaussian image = Gaussian(output image,
                         variance, mean)

'plot the image data with matplotlib'
```

c) "Apply 10x10 mean and median rank filter based noise reduction techniques to the noisy image generated in part (b). Display and discuss your results, are they as expected from your theoretical understanding of these filters."

Mean and median filtering techniques are then applied to the image with Gaussian noise, such that qualitative analysis can be undertaken for both techniques and results can be discussed. Once again, the relevant packages from sci-kit image are imported and utilised.

Figure 3. Part 1. c) Pseudocode

```
Mean function(Gaussian img, rank):
    Mean filtered image = mean(Gaussian img,
                               disk(rank))
    return Mean filtered image
Median function(Gaussian img, rank):
    Median filtered image = median(Gaussian img
                               disk(rank))
    return Median filtered image
Mean image = Mean function(Gaussian img,
                                    rank)
Median image = Median function(Gaussian img,
                                    rank)
'plot the image data with matplotlib'
```

d) "Again taking the noisy image generated in part (b) apply noise reduction using a Gaussian filter based noise reduction for various values of standard deviation (mean=0.0). What standard deviation produces the cleanest image for your data? How does this filter perform in comparison to the mean and median filters discussed in part (c)?"

The Gaussian filtering technique is then applied to the image with Gaussian noise. Its required to qualitatively analyse which value of standard deviation provides for the optimum noise reduction. A list of standard deviation values are then iterated throughout the function, and each corresponding images are plotted.

Figure 4. Part 1. d) Pseudocode

```
standard deviation list = [1,2,3,4]

  for index in standard deviation list

    Gaussian_filter(Gaussian image, index):
            Gaussianfilter = gaussian(image,
                                      index)
            return Gaussianfilter

    filter = Gaussian_filter(Gaussimg, index)
'plot the image data with matplotlib'
```

3.2. Part 2: Region properties.

a) "Develop a robust, fully automated and data driven code to locate, highlight and display the smallest scissors in the colour image illustrated in Figure 1 (scissors.jpg). Find and display the area and centroid of this item."

Once again, the same parameters are utilised to import and read our desired image from our work directory. The underlying task at hand is to extract and display the smallest scissors object from the input image. Its specified that the code must be fully automated and data driven, which infers that there cannot be any user input at any point during run-time, rather the code does the work and completes the desired task completely autonomously. It's robustness stems from the programs ability to operate adequately even with significant noise introduced. This notion will be revisited in part b. The program should be constructed sequentially, by means of passing the raw image file into the initial function, then its subsequent product is passed to the next function, and so on until we've accomplished the extracted scissors as our output file.

For ease of the reader, I will construct the pseudocode into two components, the first establishing the labelled region comprising of all objects within the image and the second being the method for which only the smallest scissors is selected and displayed.

Figure 5. Part 2. a) Pseudocode

```
import all relevant packages

Work directory ="Enter path of directory where"
"-image folder is stored"

Image directory = "Create an image directory"

Raw image = "Read in input image Scissors.jpg"

Greyscale function(image):
    greyscale image = rgb2gray(image)
    return greyscale image

Greyscale = Greyscale function(Scissors)

threshold function(image):
    threshold = threshold_otsu(image)
    binary = image > threshold
    return binary, threshold

Threshold = threshold function(Greyscale)

Clear border function(image):
    clear border = clear_border(image)
    return clear border

Border = clear border function(Threshold[0])

label function(image):
    labelled image = label(image)
    label overlay = label2rgb(labelled image)
    return labelled image, label overlay

image label = label function(Border)
'plot image using matplotlib'
```

As displayed, the raw input image undergoes four crucial procedures prior to the scissor selection procedure. These four procedures help to form what is essentially a labelled binary image, which is just a separation between the background and foreground pixels, with the foreground pixels being separate objects stored in a list. Objects in contact with the image border have been omitted from the labelling process.

Its now required to extract and display the scissors of smallest area, that being; the scissors comprised of the fewest pixels. Its also required to print the numerical value of the scissors area and its corresponding centroid.

Figure 6. Part 2. a) Pseudocode

```
list = []

for objects in regionprops(labelled image):
    if the euler number == -1:
        if area == minimum:
            append area to list

            area = min(list)

            centroid = objects.centroid

            rectangle around smallest scissors

crop function(image):
    image crop = image[rectangle]
    return image crop

cropped image = crop function(labelled image)

print(Area of scissors)
print(Centroid of scissors)

'Use matplotlib to display the labelled image
'region, with a rectangle around the smallest
'scissors, and a cropped image of the smallest
'scissors.
```

The idea is to determine which objects in the list of labelled objects meets the criteria of having an Euler number corresponding to -1. Once this criteria is met, then the areas of each object within the list is calculated. The following criteria now has to be met, which states that if the are is equivalent to the minimum area in the list, then highlight the object with the smallest area by placing a box around that object. The smallest scissors is then cropped and displayed.

b) "Design an experiment to illustrate the robustness of your final programme (as developed in part (a))to an appropriate range of image noise."

An experiment to test the programs robustness would involve implementing some noise to the image up to some reasonable degree. Similar to part. 1 d), a list of varying degrees of noise; Gaussian noise in this context, which can be established and iterated through up until some reasonable degree. In order to combat this noise, a mean or median filter can be introduced to act on the noisy image, in order for the object detection parameters to work adequately.

Figure 7. Part 2. b) Pseudocode

```
variance = [0.1, 0.3, 0.5]

    for index in variance:
            noise = random_noise(image, index)

            meanfilter = mean(noise,disk(rank))
```

The key idea is to filter the image adequately, such that it can operate with considerable noise. The for loop will print out a list of images each containing noise outlined by the variance list, then each image is subsequently filtered. From this point on the code outlined in figure 5. and 6. will suffice, with the difference being each filtered image is called through each function and printed out separately. The quantity of pictures that need to be processed corresponds to the length of the variance list.

3.3. Part 3: Colour matching.

a)"Apply the colour profile of the reference image in Figure 2 (reference.jpg) to the resized colour face image used in Part 1 (a) (we will refer to this as the input image). Plot the histograms of all three images (reference, your colour input image, and matched output image). Comment on your resultant matched image and histogram."

A program must be constructed that firstly, imports a new reference image and resizes it to the same size as the source image used in part. 1. Conveniently, the resized image generated in part. 1 can be used and called throughout the program. secondly, the program must match the histogram distribution of the source image to the reference image, generating a new image file. The source image, reference image and the matched image must then be displayed. Once that has been completed its required that the histograms of each image is printed in their red, green and blue channels.

Figure 8. Part 3. a) Pseudocode

```
ref = read reference image 'reference.jpg'

Resize function(image):
    resized = resize(image, (512,512))
    return resized

Histogram match(source, ref)):
    match = match_histograms(source ,ref)
    retrun match

ref image = Resize function(ref)
matched image = Histogram match(ref, ref image)

'Plot images with matplotlib'
```

Plotting the corresponding histograms of each image:

Figure 9. Part 3. a) Pseudocode

```
fig, axes = subplots(rows, columns)

for i, images in enumerate((Source, Ref
                    image, matched image)
    for c, colour in enumerate(Red, green,
                                        Blue)

    image histograms = exposure.histograms(img)
    image cdf = exposure.cumulative_dist(img)

    axes[z, c].plot image histogram
    axes[z, c].plot image cdf
    axes[z, 0].set label to say 'colour'

'display each image using matplotlib'
```


4. Implementation, Results and Analyses

Once the tasks were fully conceptualised and thoroughly understood, The implementation of the code could begin. As mentioned previously, version 6.3.0 of Jupyter was used in conjunction with Sci-kit image version 0.18.3. The results are segmented into their corresponding parts outlined in the assignment brief; *Part. 1: Filters, Part. 2: Region properties* and *Part. 3: Colour matching*. Firstly, the code is demonstrated, along with brief commentary on how some components work, then the results are shown and analysed.

In the code preamble, A working directory in conjunction with an image directory was configured, and both were used to store and access image data files throughout the entirety of the project. Configuring the working directory required the Python "os" module, which enables the user to interact with the operating system in order to create and interact with local directories. All relevant packages were imported from sci-kit images and matplotlib libraries respectively. The code outlined in figure. 8 below demonstrates all of the necessary packages required to complete the assignment.

Figure 10. Code preamble

```
%matplotlib inline

#-------------------------------------------------------------------------------------------#
#                                        preamble.                                          #
#-------------------------------------------------------------------------------------------#
import os

print("Current: {0}".format(os.getcwd()))
#os.chdir('/tmp')
```

```
#Local Work Directory.
work_dir = 'C:/Users/cianr/OneDrive/Desktop/IPA/'
print(work_dir)


#Image Directory.
image_dir = os.path.join(work_dir, 'images/')
print(image_dir)

!pip show scikit_image

#-------------------------------------------------------------------------------------------#
#                                Importing packages.                                        #
#-------------------------------------------------------------------------------------------#
from skimage import data, io, filters, color, exposure
from skimage.filters.rank import mean, median
from skimage.filters import gaussian, threshold_otsu
from skimage.util import random_noise
from skimage.color import rgb2gray, label2rgb
from skimage.exposure import match_histograms
from skimage.transform import resize
from skimage.morphology import disk, closing, square
from skimage.util import img_as_ubyte, crop
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops, euler_number

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy as np

from tabulate import tabulate
```


4.1. Part 1: Filters.

Part. A: The raw front facing image data file, is stored in the 'images' folder in the IPA directory. The raw image data is read into the script using the 'io' python module and assigned to a variable: 'Facial_image', for ease of calling during the build. Before printing the final image, the raw image is read through three individual functions, each using the function previous as their respective arguments.
The first function 'Imageresize' uses the sci-kit package 'resize' to transform the raw image into a 512 x 512 image. The defined function then returns the desired image. The function is assigned to a variable for convenience sake, namely 'imgR', as the functions output will be passed through the next function, and so on.

Figure 11. Part 1. a) Image resizing

```
#Loading the raw image file and printing dimensions.
Facial_image = io.imread(os.path.join(image_dir, 'Front_facing_image.jpg'))


#Resizing the image to 512 x 512 pixels.
def Imageresize(Raw_image):
    resizeimage = resize(Raw_image, (512, 512))
    return resizeimage
```

The output resized image is then passed through the 'Greyscaleconversion' function, which converts the coloured image into a monochromatic image file. This function is once again assigned to a variable, 'imgG'.

Figure 12. Part 1. a) Greyscale conversion

```
#Converting the image to grayscale and returning image.
def Grayscaleconversion(Resized_image):
    grayscaleimage = rgb2gray(Resized_image)
    return grayscaleimage
```

The resized greyscale image is now passed through the final function, 'finalimage', which calculates the minimum, maximum and mean greyscale values and prints the respective values in a table using the 'tabulate', python

package. The values are normalised to unity by default an can be simply converted to the conventional range from 0 - 255 by the following: $f(x, y) * 255$. Both images were plotted using matplotlib. Two axes were configured for the subplots and labelled accordingly. Each images was then saved directly to the 'images folder in the working directory. The plotting mechanisms code was partially sourced from the tutorial material provided.[12]

Figure 13. Part 1. a) Image plotting

```
#Printing the final result and corresponding minimum, maximum and mean values.
def finalimage(grayscaleimage):

    print(tabulate([[grayscaleimage.min(), grayscaleimage.max(), grayscaleimage.mean()]], headers=[
                                                                    'Min', 'Max', 'Mean '],
    tablefmt='orgtbl'))
    print("|Raw dimensions:", Facial_image.shape,"|")
    print("|final Dimensions:", grayscaleimage.shape,"|")


#Assigning a function to a variable for convenience when the output of the function is needed.
imgR = Imageresize(Facial_image)
imgG = Grayscaleconversion(imgR)
imgF = finalimage(imgG)


#Rendering the image plots.
fig, ax = plt.subplots(ncols = 2, figsize = (24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)


ax[0].imshow(imgR, cmap=plt.cm.gray)
ax[0].set_title("Original image (Resized - 512 x 512)", fontsize = 20, color = 'blue')
ax[0].axis("off")

ax[1].imshow(imgG, cmap=plt.cm.gray)
ax[1].set_title(" Grayscale image (Resized - 512 x 512)",fontsize = 20, color = 'red')
ax[1].axis("off")


plt.savefig(os.path.join(image_dir, 'part1a'))

plt.show()
```


Figure 14. *Part. 1. a: Original resized image vs resized greyscale image.*

The corresponding mean, minimum and maximum greyscale values alongside the raw and processed image dimensions were outputted from the program and seen below in figure 15.

	Min	**Max**	**Mean**
Normalised	0.01155	1	0.36354
Unnormalised	3	255	93
Raw dimensions	(937, 470, 3)	**Final dimensions**	(512, 512)

Figure 15. Part 1. a) *Table containing greyscale values.*

Part. B: Gaussian noise was subsequently added to the previous output image file from part. a. The 'random_noise' package from sci-kit image was used to add such noise. Since the noise is Gaussian, meaning that the probability density function of any given pixel throughout the image can be modelled by the Gaussian distribution, simply increasing the variance or the spread of the function will subsequently increase the noise in the image. In this context, the variance is set to 0.01 and the mean set to 0.0 as prescribed in the assignment brief.The code and output image can be seen in figure 16 and 17 respectively.

Figure 16. Part 1. b) Gaussian noise

```
def Gaussian(outputA, variance, M):
    GaussianNoise = random_noise(outputA, mode = 'gaussian', var = variance**2, mean = M)
    return GaussianNoise


#Assigning a function to a variable for convenience when the output of the function is needed.
Gaussimg = Gaussian(imgG, 0.1, 0.0)


#Rendering the image plots.
fig, ax = plt.subplots(ncols = 2, figsize = (24, 9))

ax[0] = plt.subplot(1, 3, 2)

ax[0].imshow(Gaussimg, cmap=plt.cm.gray)
ax[0].set_title("Output image with Gaussian noise(512 x 512)", fontsize = 20, color = 'green')
ax[0].axis("off")

plt.savefig(os.path.join(image_dir, 'part1b'))

plt.show()
```


Figure 17. Part 1. b). *Gaussian image, variance = 0.01, mean = 0.0.*

Part. C: A 10 x 10 pixel region mean filter and median filter technique is then applied to the Gaussian image created in part. c and the results are contrasted. The kernel, as briefly mentioned in section 2.2, corresponds to this 10 x 10 pixel region or 'rank', which is used to calculate the corresponding mean and median values of the central pixels. The mean and median sci-kit image modules were imported and used.

Figure 18. Part 1. c) Mean and Median filtering

```
def MeanFilter(outputB, FilterRank):
    uint8 = img_as_ubyte(outputB, force_copy = False)
    meanfil = mean(uint8, disk(FilterRank))
    return meanfil


def MedianFilter(outputB, FilterRank):
    uint8 = img_as_ubyte(outputB, force_copy = False)
    medianfil = median(uint8, disk(FilterRank))
    return medianfil


#Assigning a function to a variable for convenience when the output of the function is needed.
MeanF = MeanFilter(Gaussimg, 10)
MedianF = MedianFilter(Gaussimg, 10)

#Rendering the image plots.
fig, ax = plt.subplots(ncols = 2, figsize = (24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)


ax[0].imshow(MeanF, cmap=plt.cm.gray)
ax[0].set_title("Image with mean rank filtering", fontsize = 20, color = 'blue')
ax[0].axis("off")

ax[1].imshow(MedianF, cmap=plt.cm.gray)
ax[1].set_title("Image with Median rank filtering",fontsize = 20, color = 'red')
ax[1].axis("off")


plt.savefig(os.path.join(image_dir, 'part1c'))

plt.show()
```

The Gaussian image is of a float64 data type, which has a normalised pixel intensity range from 0 - 1. It suggested the image be converted to a uint8 data type, which has a corresponding pixel intensity range from 0 - 255, in order to avoid precision loss when using these filters. Sci-kit image documentation suggested to convert the data type by using the function 'img_as_ubyte_'.
Increasing the rank dimensions, which increases the kernel size leads to greater 'smoothening' of the image, however the details of the image become completely indistinguishable at such large values. Once the images have been processed, they are plotted using the matplotlib code used previously, with the exception of different axes labels.
The median filter, in contrast to the mean filter is a more robust mechanism in determining the average pixel value at a given point.Its true in the sense that it enables the omission of sparse outliers from each pixel calculation as opposed to the calculating the mean, which on the other hand takes the weighted some of each data set. As a direct result of this, the median filter is a more optimum candidate for edge preservation when filtering images as it preserves step edges between highly variant pixel intensities at some point.

So, the output images are expected to differ slightly in terms of their edge preservation. It should be apparent that the mean filter will have greater blurring on the edges of the image, in comparison to the median filter which should have slightly sharper edges.

Both images are displayed in figure 19 below.

Figure 19. Part 1. c). *Mean and median filtered images. Rank = 10 x 10.*

Its evident that the median image has much better edge preservation, as can be seen contrasting the left and centre foreground and also the right middle ground of both pictures in figure 19.

Part. D: A Gaussian filter was applied to the noisy image produced in part. c, for vary degrees of noise and the results were qualitatively contrasted with each-other and also with the mean and median filtering techniques.

The 'Gaussian' filtering module was imported from the sci-kit image library in the code preamble and used to create the Gaussian filter. In order to qualitatively determine which standard deviation value rendered the 'cleanest' output image, a list of selected standard deviation values ranging from 1 - 2, incremented in values of 0.2 was configured. A for loop was constructed containing the Gaussian filtering function and the matplotlib plotting mechanism. The loop iterates throughout the list and plots the image data for each value of the standard deviation. There is a total of 18 images plotted and can be seen in fig. 21 and 22. For each value of the standard deviation, the original image, the unfiltered noisy image and the filtered noisy image is plotted for qualitative comparison.

Figure 20. Part 1. d) Gaussian filtering

```
stdlist = [1, 1.2, 1.4, 1.6, 1.8, 2]

for i in stdlist:
    def GaussianNoiseFilter(outputB, std):
        GaussianFiltered = gaussian(outputB, sigma = std, mode ='nearest')
        return GaussianFiltered



    imgGF = GaussianNoiseFilter(Gaussimg, i)

#Rendering the image plots.
    fig, ax = plt.subplots(ncols=3, figsize=(24, 9))

    ax[0] = plt.subplot(1, 3, 1)
    ax[1] = plt.subplot(1, 3, 2)
    ax[2] = plt.subplot(1, 3, 3, sharex=ax[0], sharey=ax[0])

    ax[0].imshow(imgR, cmap=plt.cm.gray)
    ax[0].set_title('Original Image (512 x 512)', fontsize = 20, color = 'blue')
    ax[0].axis('off')

    ax[1].imshow(Gaussimg, cmap=plt.cm.gray)
    ax[1].set_title('Gaussian Noise Image (512 x 512)', fontsize = 20, color = 'red')
    ax[1].axis('off')
```

```
    ax[2].imshow(imgGF, cmap=plt.cm.gray)
    ax[2].set_title('Gaussian Filtered Image (512 x 512)', fontsize = 20, color = 'green')
    ax[2].axis('off')

    fig.tight_layout()

# save figure
    plt.savefig(os.path.join(image_dir, 'part1d'))

    plt.show()
```

The program outputs the following images:

Figure 21. Part 1. d) *Gaussian filtered images for σ = 1, 1.2, 1.4 respectively.*

Figure 22. Part 1. d) *Gaussian filtered images for σ = 1.6, 1.8, 2 respectively.*

Qualitatively, between each of the Gaussian images its difficult to determine which standard deviation provides for the best filter. However, close inspection suggests the higher value of $\sigma = 2$, reduces much more noise in comparison with the value $\sigma = 1$. There seems to be less notable grain in the image with a higher standard deviation.

In comparison with the Mean and Median filters used previously, its immediately evident that the Gaussian filter preserves much more detail than the former filters, as they induce a greater 'blurring effect' due to the intrinsic nature of how they operate. The noise reduction is significantly better in the mean and median filters as a direct result of this 'blurring' effect within good reason. The comparatively softer 'blur' from the Gaussian function is due to how the mean values are calculated. As mentioned in section 2.1, the average values determined by the Gaussian kernel tend to have values which lie closer to the central pixel value, as opposed to the mean filter, which uniformly calculates the weighted mean using the neighbouring pixels.

4.2. Part 2: Region properties.

Part. A: Some of the more notable, necessary modules imported from the sci_kit library were the'rgb2gray', 'threshold_otsu', 'clear_border', 'label', 'label2rgb', 'regionprops' and 'euler_number', which were used in succession throughout the functions within the program. The raw image 'scissors.jpg' was read in using the 'io' python module and assigned a corresponding variable name. In order for the detection of the scissors to be completely autonomous, the image was passed through several functions sequentially, each using the various sci_kit modules outlined above. Firstly, the scissors is passed through 'Grayscaleconverter' function, then through the 'Threshold_otsu' function, the 'clear' function where its then labeled by the 'labl' function. At this point, the scissors detection mechanism is employed and finally, the smallest scissors along with its respective area and centroid is outputted from the program.

Beginning with the first function, seen in figure 23. below. The scissors image is read in and passed through the 'Grayscaleconverter' function.

Figure 23. Part 2. a) Grayscale conversion

```
#Reading in the reference image.
Scissor = io.imread(os.path.join(image_dir, 'scissors.jpg'))

#Converting RGB to Greyscale.
def Grayscaleconverter(image):
    grayscaleimage = rgb2gray(image)
    return grayscaleimage

grayscale = Grayscaleconverter(Scissor)
```

Its necessary to convert the image into greyscale for the purpose of background and foreground segmentation that will occur in the next step.

The image is then passed through the 'Threshold_otsu' function, which segments the images background from its foreground by converting the image into a binary image. In this context, the background pixels are numerically less than the threshold values determined by the Otsu thresholding mechanism, thus take on a value of 0, or black to be specific. Conversely, the objects in the image succeed the threshold values and take on values of 255, or white, thus segmenting the image.

Passing the greyscale image through the thresholding function:

Figure 24. Part 2. a) Otsu thresholding

```
#Applying the otsu threshold for background segmentation.
def Threshold_otsu(image):
    Thresholdimg = threshold_otsu(image)
    binary = image > Thresholdimg
    return binary, Thresholdimg

Threshold = Threshold_otsu(grayscale)
```

The binary image output called 'Threshold' is then passed into the 'clear' function which uses the 'clear_border' module to omit any objects in contact with the image border, as specified by the assignment brief.

Figure 25. Part 2. a) Border clearing

```
#Clearing objects connected to the image border.
def Clear(image):
    ClearedBorder = clear_border(image)
    return ClearedBorder

ClearBorder = Clear(Threshold[0])
```

The cleared border image is then passed through the 'labl' function which makes use of sci-kits 'label' module, which enables the user to create an indexed list of all objects present within the image. This is particularly useful later on when attempting to discover certain characteristics of the image objects. The labeled image is also overlayed with a colour scheme for aesthetic purposes, which is provided by the 'label2rgb' sci-kit module. This can be seen in figure. 28.

Figure 26. Part 2. a) Object labeling

```
#Labelling image regions.
def labl(image):
    labelledimg = label(image)
    label_overlay = label2rgb(labelledimg, image = image, bg_label = 0)
    return labelledimg, label_overlay

labelledimage = labl(ClearBorder)
```

The scissors detection mechanism operates by the following procedure: Firstly, an empty list is established and given the the arbitrary name of 'target'. This list stores the areas of the scissors, provided the criteria is met. Secondly, A 'for loop' statement is constructed, which iterates throughout the labeled images list configured by the 'labl' function established previously. Located within this 'for loop', a nested 'if' statement is found, which provides the criteria that must be met in order for the object to firstly, be a scissors and secondly, to have the minimum area. The initial 'if' statement comprises od The 'euler_number' Sci-kit module, which has the property of determining the respective Euler number of each object in the labelled list. To briefly clarify, the Euler number is found by subtracting the number of holes away from the number of objects in each labeled region, thus we are looking for objects with a corresponding Euler number of -1. Once this criterion is met, the second if statement appends the minimum area into the target list. Finally, a small blue rectangular box is placed around the smallest scissors within the image. Both the scissors area and centroid are subsequently printed.

Figure 27. Part 2. a) Scissor detection

```
#Configuring axes for uncropped and cropped images.
fig, ax = plt.subplots(ncols=2, figsize=(24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)

ax[0].imshow(labelledimage[1], cmap=plt.cm.gray)
ax[0].set_title('Labelled image', fontsize = 20, color = 'blue')
ax[0].axis('off')



#fig, ax = plt.subplots(figsize = (10, 6))
#ax.imshow(labelledimage[1])


#Creating an empty area for the areas of the scissors.
target = []

#Iterating through the labelled images.
for i in regionprops(labelledimage[0]):
    if i.euler_number == -1:
        if i.area == A:

            target.append(i.area)

            A = min(target)

            C = i.centroid

            minr, minc, maxr, maxc = i.bbox
            Area = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                    fill = False, edgecolor = 'blue', linewidth = 1.5)
            ax[0].add_patch(Area)
```


The image highlighting the smallest scissors by means of a blue rectangular box is then displayed using matplotlib. Finally, the smallest scissor is then cropped from the highlighted image, using the 'imagecrop' function, which can be seen in figure 28 and 29 below. This function uses the parameters established by the rectangular box configured in the scissor detection mechanism. The images were then plotted using matplotlib.

The image plotting and cropping mechanism:

Figure 28. Part 2. a) Cropping / plotting the scissors

```
#Cropping the desired scissors.
def imagecrop (image):
    image_crop = image[minr:maxr, minc:maxc]
    return image_crop

imgCrop = imagecrop(labelledimage[0])

print('This is the area of the samllest scissor image:', A)

print('This is the centroid of the smallest scissor image:', C)

ax[1].imshow(imgCrop, cmap=plt.cm.gray)
ax[1].set_title('Cropped Image', fontsize = 20, color = 'red')
ax[1].axis('off')

plt.tight_layout()
plt.show()
```

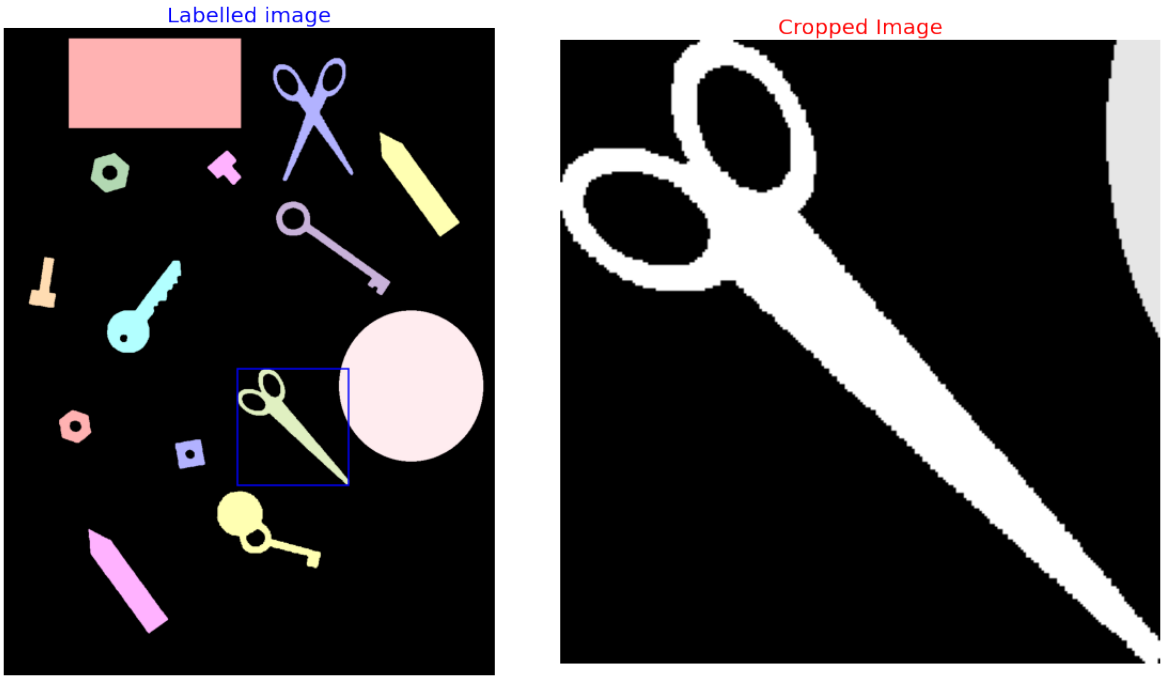

Figure 29. Part 2. a) *Highlighted scissors and cropped scissor image outputs.*

program determined the scissors to have a **Centroid** = (818.56, 593.56) and **Area** = 8555 pixels.

Part. B: To test the robustness of the program, varying magnitudes of Gaussian noise were introduced into the the image by implementing a Gaussian noise function to the scissors image. The original code in part a. was amended slightly by introducing a median filter after the Gaussian noise addition, to remove any noise that was present in the image, thus increasing its robustness.
The testing process consisted of varying the degrees of Gaussian noise. These values were initially added into a list called 'noise', as seen in figure 30 below. For each index in this list, or each value of noise to be more precise, the program ran its course as normal, as outlined previously, attempting to detect the scissors of smallest pixel area. This process was repeated for each index, or noise value in the list, and the results were subsequently displayed. The values for the noise indices were determined by means of brute force, meaning a succession of trials using different list sizes took place until a list containing a range of noise values was found which still permitted the detection of the smallest scissors.

In the case of the median filter, it gave moderately good results in the very low level noise range: σ = 0.1 - 0.3, with the list being arbitrarily incremented in values of 0.1. For the median filter, it completely breaks down for values of σ > 0.3. The median filter utilised a 5 x 5 kernel operator, which qualitatively gave the best results. Mean filtering techniques were applied, with several different kernel values but the results were not optimal in comparison to the median filter.

The Gaussian noise function was implemented into the original code constructed in part. a, and underwent a test run to determine if significant noise is present.

Figure 30. Part 2. b) Adding Gaussian noise

```
noise = [0.1, 0.2, 0.3]

for i in noise:
    def Gaussiannoise(grayscale, variance, M):
        Gaussian = randon_noise(grayscale, mode = 'gaussian', var = variance**2, mean = M)
        return Gausian

Gauss = Gaussiannoise(Scissor, i, 0.0)
```

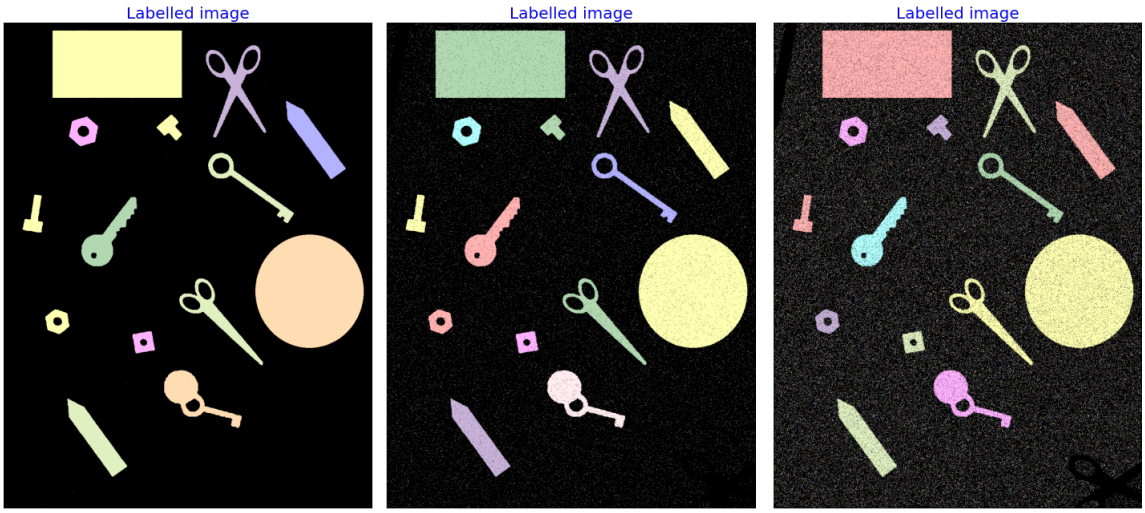

Figure 31. Part 2. b) *Gaussian noise for values σ = 0.1, 0.2 and 0.3 respectively.*

Figure 31. demonstrates the successive outputs of the program after Gaussian noise has been implemented for values of σ = 0.1, 0.2 and 0.3. Evidently, the program breaks down with the addition of noise, as the scissors fails to be detected. This can be rectified for small a small noise range by implementing the median filter.

Figure 32. Part 2. b) Adding the median filter

```
def MedianFilter(image, FilterRank):
    uint8 = img_as_ubyte(image, force_copy = False)
    medianfil = median(uint8, disk(FilterRank))
    return medianfil

Median = MedianFilter(Gauss, 5)
```

This function is implemented into the code established in part a. Once the image with Gaussian noise is outputted from the 'Gaussiannoise' function, its then subsequently passed through the median filter function as seen above in figure 32. Once this process has been completed and the median filter is has been applied sufficiently reducing or practically eliminating the noise such that the smallest scissors can be detected once again by means of the scissors detection mechanism previously established.Finally, the results are displayed in figure 33. for each of the values: σ = 0.1, 0.2 and 0.3. in the list.

As seen in figure 33, its evident that with the successive increase in Gaussian noise, the quality of the cropped scissor image deteriorates moderately. The edges of the scissors corresponding to the image with noise: σ = 0.3, are noticeably more jagged that those with lesser noise values.

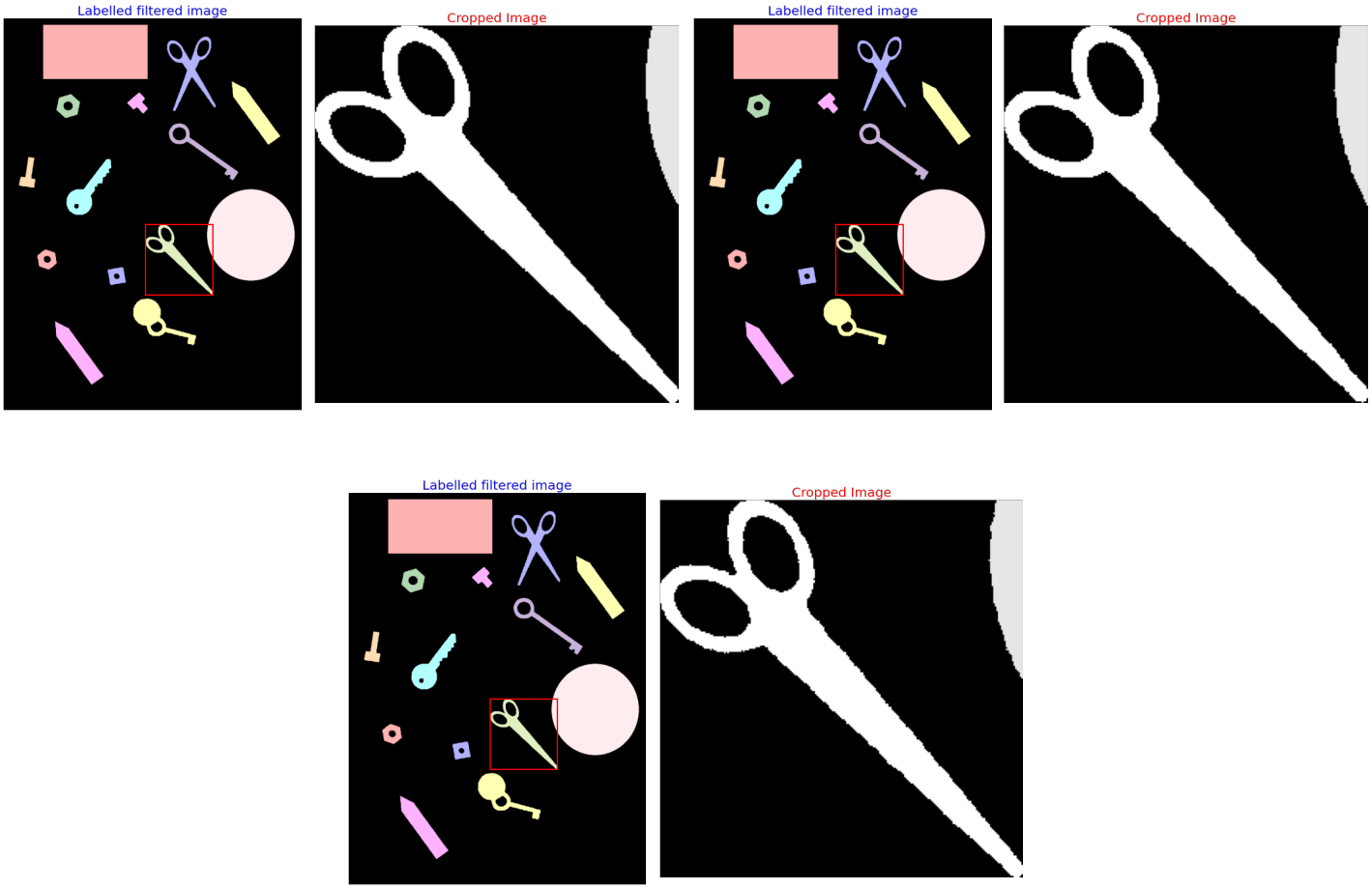

Figure 33. Part 2. b) *Scissors detection for σ = 0.1, 0.2 and 0.3 respectively*

4.3. Part 3: Colour matching.

Part. A: The 'match_histograms' module was required for this program in order to sufficiently match the source image's histogram to the reference image. The resized front facing image outputted from part 1. a, namely 'imgR' is called throughout this program, and has its respective histogram matched to that of the reference image. Each of the images, namely, the source, matched and reference image is displayed using matplotlib. The image histograms are then plotted for each of the red, green and blue channels of each image.

The reference image is read in from he 'images' folder and resized to match the dimensions of 'imgR' via the 'Reference' function in figure 34 below. The source image's histogram is then matched to the reference image's histogram via the 'HistogramMatch' function, which utilises the Sci-kit module, 'match_histograms'. The outputs of each function are assigned to the variables 'imgRef' and 'imgMat' respectively.

Figure 34. Part 3. a) Matching the histograms and plotting the images

```
#Reading in the reference image.
ref = io.imread(os.path.join(image_dir, 'reference.jpg'))

#Defining a function that resizes the reference image to 512 x 512 pixels.
def Reference (Reference):
    resizeimg = resize(Reference, (512, 512))
    return resizeimg

#Histogram matching function.
def HistogramMatch (source, reference):
    matchedhist = match_histograms(source, reference, multichannel = True)
    return matchedhist

imgRef = Reference(ref)
imgMat = HistogramMatch(imgR, imgRef)
.
.
.
```

```
#Rendering the image plots
fig, ax = plt.subplots(ncols=3, figsize=(24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)
ax[2] = plt.subplot(1, 3, 3, sharex=ax[0], sharey=ax[0])

ax[0].imshow(imgR, cmap=plt.cm.gray)
ax[0].set_title('Original Image (512 x 512)', fontsize = 20, color = 'blue')
ax[0].axis('off')

ax[1].imshow(imgRef, cmap=plt.cm.gray)
ax[1].set_title('Reference Image (512 x 512)', fontsize = 20, color = 'red')
ax[1].axis('off')

ax[2].imshow(imgMat, cmap=plt.cm.gray)
ax[2].set_title('Matched Image (512 x 512)', fontsize = 20, color = 'green')
ax[2].axis('off')

fig.tight_layout()

# save figure
plt.savefig(os.path.join(image_dir, 'part3a'))

plt.show()
```

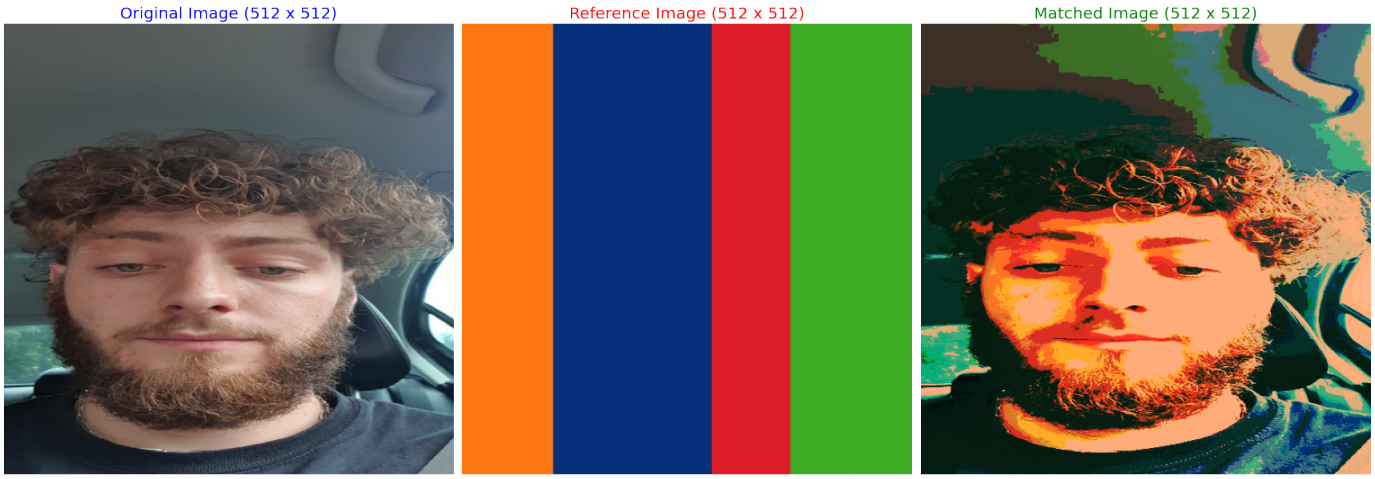

Figure 35. Part 3. a) *Source, reference and matched images.*

Figure 35 displays the source, reference and matched images outputs. The matched image now has an identical distribution of red, green and blue pixels as the reference image. It may have been easy to empirically deduce that the middle ground of the source image, or more precisely, the entire facial region, would be mapped to the majority of the red and orange pixels in the reference image, due to the colour distribution captured in the image. The distribution of the blue pixels around the darker regions of the source image would have been expected.

We can further this notion by observing the histograms comprised of the red, green and blue distributions for each of the image data sets. The histograms were plotted using matplotlib by means of the procedure outlined in figure 36.

Figure 36. Part 3. a) Plotting the histograms for the source, reference and matched images

```
#Plotting the Histograms, Cumalative Histograms and RGB channels.
fig, ax = plt.subplots(nrows = 3, ncols=3, figsize=(24, 24))

for i, images in enumerate((imgR, imgRef, imgMat)):
    for z, colour in enumerate (('Red', 'Green', 'Blue')):

        img_hist, bins = exposure.histogram(images[..., z], source_range = 'dtype')
        img_cdf, bins = exposure.cumulative_distribution(images[..., z])
```

```
        ax[z, i].plot(bins, img_hist / img_hist.max())
        ax[z, i].plot(bins, img_cdf)
        ax[z, 0].set_ylabel(colour)

ax[0, 0].set_title('Source Image', fontsize = 30)
ax[0, 1].set_title('Reference Image', fontsize = 30)
ax[0, 2].set_title('Matched Image', fontsize = 30)
ax[0, 0].set_ylabel('Red', fontsize = 30, color = 'Red')
ax[1, 0].set_ylabel('Green', fontsize = 30, color = 'Green')
ax[2, 0].set_ylabel('Blue', fontsize = 30, color = 'Blue')

plt.tight_layout()

plt.savefig(os.path.join(image_dir, 'part3a1'))

plt.show()
```

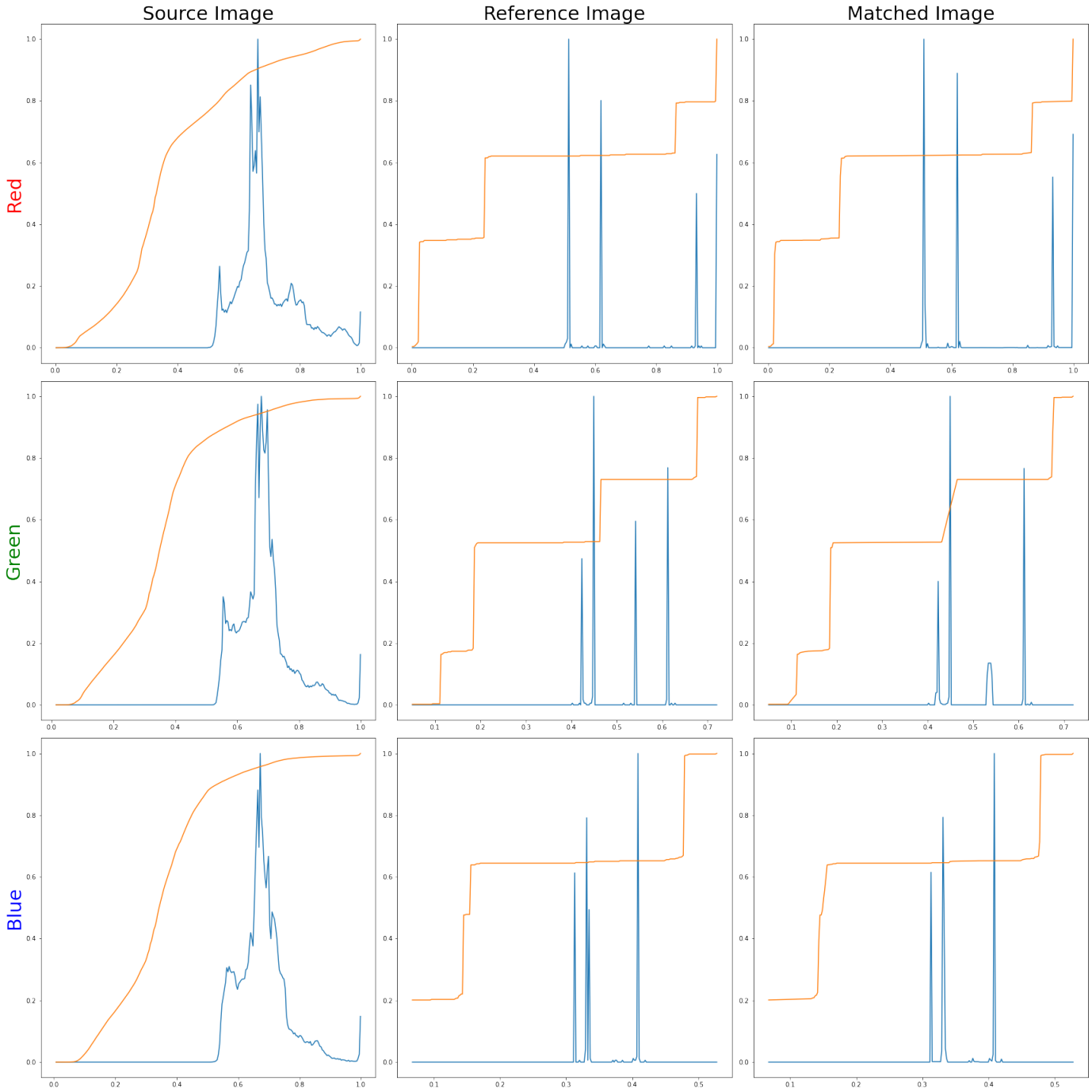

Figure 37. Part 3. a) *Red, green and blue channel Histograms of the Source, reference and matched images respectively.*

Observing figure 37, the histograms distributions belonging to the reference image are effectively identical to the matched image, with perhaps some discrepancies present due to pragmatic limitations regarding *'exact mapping'* of the source image's histogram to the reference image's. In practicality, its often a difficult task to map an image with discretised values to match a reference histogram distribution with exact precision, so in the majority of cases, its intuitive to use approximations to achieve the desired histogram.

5. Conclusion and Discussion

Part. 1: The tasks were completed, the source image 'Front_facing_image.jpg' was successfully imported, resized and converted from RGB to greyscale, with corresponding mean, min and max values being 93, 3, 255 respectively. Gaussian noise was then successfully added and subsequently removed using various filtering techniques, namely the mean median and Gaussian filters. Each of filtered images were displayed and qualitatively discussed. It was discovered that the median filter has better edge preservation, whilst the Gaussian filter preserves specific details with softer blurring.

Part. 2: The smallest scissors was autonomously detected in the prescribed image and had shown to have a corresponding pixel area of 8555 pixels and a centroid of (88.56, 593.56). Figure 29 displays the highlighted scissors object, however I'm unsure whether the segment of the circle is included in the total pixel count for the area of the segmented scissors region. The robustness of the code was tested by the introduction of Gaussian noise. Implementing the median filter permitted the program to operate within the range of noise σ = 0 - 0.3. There is major room for improvement here, which could be done by using several other useful sci-kit modules, such as 'skimage.morphology.binary_closing' module, which may have aided with better scissor detection in the case of larger noise ranges. Another useful module that perhaps would have rendered far better results is the 'skimage.morphology.dilation' module, which allows for expansion of labeled regions, giving a larger border, so in circumstances where the image is particularly noisy, the objects and their distinct features can still be identified.

Part. 3: The histogram belonging to the resized source image from part 1. was successfully mapped to the resized reference image, each image was displayed along with their corresponding histograms. A total of 9 histograms were printed, as RGB histograms are 3-dimensional. The pixel distributions of the reference and matched images were qualitatively discussed.

6. References

1. Planetary Imaging: How to Process Planetary Images [Internet]. Sky Telescope. 2021. Available from: https://skyandtelescope.org/astronomy-resources/how-to-process-planetary-images/

2. What is Medical Image Processing | Synopsys [Internet]. Synopsys.com. 2021. Available from: https://www.synopsys.com/glossary/what-is-medical-image-processing.html

3. scikit-image: Image processing in Python — scikit-image [Internet]. Scikit-image.org. 2021. Available from: https://scikit-image.org/

4. Whelan P. cipa$_{m}anual$.2012.

5. Color image - Wikipedia [Internet]. En.wikipedia.org. 2021. Available from: https://en.wikipedia.org/wiki/Color$_{i}mage$

6. Fisher R, Perkins S. Spatial Filters - Mean Filter [Internet]. Homepages.inf.ed.ac.uk. 2003. Available from: https://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm

7. Fisher R, Perkins s. Glossary - Convolution [Internet]. Homepages.inf.ed.ac.uk. 2003 . Available from: https://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm

8. Median filter - Wikipedia [Internet]. En.wikipedia.org. 2021 . Available from: https://en.wikipedia.org/wiki/Median$_{f}ilter$

9. Fisher R, Perkins S. Spatial Filters - Gaussian Smoothing [Internet]. Homepages.inf.ed.ac.uk. 2003 . Available from: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

10. Otsu's method - Wikipedia [Internet]. En.wikipedia.org. 2021 . Available from: https://en.wikipedia.org/wiki/Otsu

11. Histogram matching - Wikipedia [Internet]. En.wikipedia.org. 2021 . Available from: https://en.wikipedia.org/wiki/Histogram$_{m}atching$

12. P. Whelan. tutorial videos - EE425

7. Appendix

Part 1: Filters.

```
%matplotlib inline

#-------------------------------------------------------------------------------------------------#
#                                          preamble.                                              #
#-------------------------------------------------------------------------------------------------#
import os

print("Current: {0}".format(os.getcwd()))
#os.chdir('/tmp')

#Local Work Directory.
work_dir = 'C:/Users/cianr/OneDrive/Desktop/IPA/'
print(work_dir)

#Image Directory.
image_dir = os.path.join(work_dir, 'images/')
print(image_dir)

!pip show scikit_image

#-------------------------------------------------------------------------------------------------#
#                                    Importing packages.                                          #
#-------------------------------------------------------------------------------------------------#
from skimage import data, io, filters, color, exposure
from skimage.filters.rank import mean, median
from skimage.filters import gaussian
from skimage.util import random_noise
from skimage.color import rgb2gray
from skimage.exposure import match_histograms
import matplotlib.pyplot as plt
from skimage.transform import resize
from tabulate import tabulate


#-------------------------------------------------------------------------------------------------#
#                                      Part 1: Filters.                                           #
#-------------------------------------------------------------------------------------------------#
#a)

 #Acquire a colour image of your own face (front facing). Develop a programme to resize it to
 #512x512 pixels. Convert this image to greyscale and display your result. Find and display the
 #minimum, maximum and mean greyscale values in this image.


#Loading the raw image file and printing dimensions.
Facial_image = io.imread(os.path.join(image_dir, 'Front_facing_image.jpg'))


#Resizing the image to 512 x 512 pixels.
def Imageresize(Raw_image):
    resizeimage = resize(Raw_image, (512, 512))
    return resizeimage


#Converting the image to grayscale and returning image.
def Grayscaleconversion(Resized_image):
    grayscaleimage = rgb2gray(Resized_image)
    return grayscaleimage
```

```
#Printing the final result and corresponding minimum, maximum and mean values.
def finalimage(grayscaleimage):

    print(tabulate([[grayscaleimage.min(), grayscaleimage.max(), grayscaleimage.mean()]],
                                    headers=['Min', 'Max', 'Mean '], tablefmt='orgtbl'))
    print("|Raw dimensions:", Facial_image.shape,"|")
    print("|final Dimensions:", grayscaleimage.shape,"|")


#Assigning a function to a variable for convenience when the output of the function is needed.
imgR = Imageresize(Facial_image)
imgG = Grayscaleconversion(imgR)
imgF = finalimage(imgG)


#Rendering the image plots.
fig, ax = plt.subplots(ncols = 2, figsize = (24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)


ax[0].imshow(imgR, cmap=plt.cm.gray)
ax[0].set_title("Original image (Resized - 512 x 512)", fontsize = 20, color = 'blue')
ax[0].axis("off")

ax[1].imshow(imgG, cmap=plt.cm.gray)
ax[1].set_title(" Grayscale image (Resized - 512 x 512)",fontsize = 20, color = 'red')
ax[1].axis("off")


plt.savefig(os.path.join(image_dir, 'part1a'))

plt.show()

#b)

 #Taking the output image from part (a) and add Gaussian noise (mean=0.0, variance=0.01).
 # Display your result.


def Gaussian(outputA, variance, M):
    GaussianNoise = random_noise(outputA, mode = 'gaussian', var = variance**2, mean = M)
    return GaussianNoise


#Assigning a function to a variable for convenience when the output of the function is needed.
Gaussimg = Gaussian(imgG, 0.1, 0.0)


#Rendering the image plots.
fig, ax = plt.subplots(ncols = 2, figsize = (24, 9))

ax[0] = plt.subplot(1, 3, 2)

ax[0].imshow(Gaussimg, cmap=plt.cm.gray)
ax[0].set_title("Output image with Gaussian noise(512 x 512)", fontsize = 20, color = 'green')
ax[0].axis("off")

plt.savefig(os.path.join(image_dir, 'part1b'))

plt.show()
```

```
#c)

 #Apply 10x10 mean and median rank filter based noise reduction techniques to the noisy image
 #generated in part (b). Display and discuss your results, are they as expected from your
 #theoretical understanding of these filters.


def MeanFilter(outputB, FilterRank):
    uint8 = img_as_ubyte(outputB, force_copy = False)
    meanfil = mean(uint8, disk(FilterRank))
    return meanfil


def MedianFilter(outputB, FilterRank):
    uint8 = img_as_ubyte(outputB, force_copy = False)
    medianfil = median(uint8, disk(FilterRank))
    return medianfil


#Assigning a function to a variable for convenience when the output of the function is needed.
MeanF = MeanFilter(Gaussimg, 10)
MedianF = MedianFilter(Gaussimg, 10)

#Rendering the image plots.
fig, ax = plt.subplots(ncols = 2, figsize = (24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)


ax[0].imshow(MeanF, cmap=plt.cm.gray)
ax[0].set_title("Image with mean rank filtering", fontsize = 20, color = 'blue')
ax[0].axis("off")

ax[1].imshow(MedianF, cmap=plt.cm.gray)
ax[1].set_title("Image with Median rank filtering",fontsize = 20, color = 'red')
ax[1].axis("off")


plt.savefig(os.path.join(image_dir, 'part1c'))

plt.show()

#d)
 #Again taking the noisy image generated in part (b) apply noise reduction using a Gaussian
 #filter based noise reduction for various values of standard deviation (mean=0.0).
 #What standard deviation produces the cleanest image for your data? How does this filter
 #perform in comparison to the mean and median filters discussed in part (c)?


stdlist = [1, 1.2, 1.4, 1.6, 1.8, 2]

for i in stdlist:
    def GaussianNoiseFilter(outputB, std):
        GaussianFiltered = gaussian(outputB, sigma = std, mode ='nearest')
        return GaussianFiltered



    imgGF = GaussianNoiseFilter(Gaussimg, i)
.
.
.
.
```

```
#Rendering the image plots.
    fig, ax = plt.subplots(ncols=3, figsize=(24, 9))

    ax[0] = plt.subplot(1, 3, 1)
    ax[1] = plt.subplot(1, 3, 2)
    ax[2] = plt.subplot(1, 3, 3, sharex=ax[0], sharey=ax[0])

    ax[0].imshow(imgR, cmap=plt.cm.gray)
    ax[0].set_title('Original Image (512 x 512)', fontsize = 20, color = 'blue')
    ax[0].axis('off')

    ax[1].imshow(Gaussimg, cmap=plt.cm.gray)
    ax[1].set_title('Gaussian Noise Image (512 x 512)', fontsize = 20, color = 'red')
    ax[1].axis('off')

    ax[2].imshow(imgGF, cmap=plt.cm.gray)
    ax[2].set_title('Gaussian Filtered Image (512 x 512)', fontsize = 20, color = 'green')
    ax[2].axis('off')

    fig.tight_layout()

# save figure
    plt.savefig(os.path.join(image_dir, 'part1d'))

    plt.show()
```

Part 2: Region properties.

```
%matplotlib inline

#-----------------------------------------------------------------------------------------#
#                                        preamble.                                        #
#-----------------------------------------------------------------------------------------#
import os

print("Current: {0}".format(os.getcwd()))
#os.chdir('/tmp')

#Local Work Directory.
work_dir = 'C:/Users/cianr/OneDrive/Desktop/IPA/'
print(work_dir)

#Image Directory.
image_dir = os.path.join(work_dir, 'images/')
print(image_dir)

!pip show scikit_image

#-----------------------------------------------------------------------------------------#
#                                    Importing packages.                                  #
#-----------------------------------------------------------------------------------------#
from skimage import data, io, filters, color, exposure
from skimage.color import rgb2gray
from skimage.morphology import disk, closing, square
from skimage.filters import threshold_otsu
from skimage.util import img_as_ubyte, crop
from skimage.color import label2rgb
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops, euler_number
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
```

```
#-------------------------------------------------------------------------------------------#
#                                Part 2): Region properties.                                #
#-------------------------------------------------------------------------------------------#

#a)
 #Develop robust7, fully automated8 and data driven9 code to locate, highlight and display the
 #smallest scissors in the colour image illustrated in Figure 1 (scissors_col_2.jpg).
 #Find and display the area and centroid of this item.



#Reading in the reference image.
Scissor = io.imread(os.path.join(image_dir, 'scissors.jpg'))

#Converting RGB to Greyscale.
def Grayscaleconverter(image):
    grayscaleimage = rgb2gray(image)
    return grayscaleimage

grayscale = Grayscaleconverter(Scissor)

#Applying the otsu threshold for background segmentation.
def Threshold_otsu(image):
    Thresholdimg = threshold_otsu(image)
    binary = image > Thresholdimg
    return binary, Thresholdimg

Threshold = Threshold_otsu(grayscale)
#Clearing objects connected to the image border.
def Clear(image):
    ClearedBorder = clear_border(image)
    return ClearedBorder

ClearBorder = Clear(Threshold[0])

#Labelling image regions.
def labl(image):
    labelledimg = label(image)
    label_overlay = label2rgb(labelledimg, image = image, bg_label = 0)
    return labelledimg, label_overlay

labelledimage = labl(ClearBorder)


#Configuring axes for uncropped and cropped images.
fig, ax = plt.subplots(ncols=2, figsize=(24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)

ax[0].imshow(labelledimage[1], cmap=plt.cm.gray)
ax[0].set_title('Labelled image', fontsize = 20, color = 'blue')
ax[0].axis('off')

#Creating an empty area for the areas of the scissors.
target = []
.
.
.
.
.
.
```

```
#Iterating through the labelled images.
for i in regionprops(labelledimage[0]):
    if i.euler_number == -1:
        if i.area == A:

            target.append(i.area)

            A = min(target)

            C = i.centroid

            minr, minc, maxr, maxc = i.bbox
            Area = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                    fill = False, edgecolor = 'blue', linewidth = 1.5)
            ax[0].add_patch(Area)


#Cropping the desired scissors.
def imagecrop (image):
    image_crop = image[minr:maxr, minc:maxc]
    return image_crop

imgCrop = imagecrop(labelledimage[0])

print('This is the area of the samllest scissor image:', A)

print('This is the centroid of the smallest scissor image:', C)

ax[1].imshow(imgCrop, cmap=plt.cm.gray)
ax[1].set_title('Cropped Image', fontsize = 20, color = 'red')
ax[1].axis('off')

plt.tight_layout()
plt.show()


#b)
 #Design an experiment to illustrate the robustness of your final programme (as developed in
 # part (a)) to an appropriate range of image noise.
 #implement into the code in part a. Ensure variables are correctly called in each function. Also
 #appropriate the indentation in order for the loop to work.

noise = [0.1, 0.2, 0.3]

for i in noise:
    def Gaussiannoise(grayscale, variance, M):
        Gaussian = randon_noise(grayscale, mode = 'gaussian', var = variance**2, mean = M)
        return Gausian

Gauss = Gaussiannoise(Scissor, i, 0.0)

def MedianFilter(image, FilterRank):
    uint8 = img_as_ubyte(image, force_copy = False)
    medianfil = median(uint8, disk(FilterRank))
    return medianfil

Median = MedianFilter(Gauss, 5)
```

Part 3: Colour matching.

```
#-------------------------------------------------------------------------------------------#
#                                  Part 3): Colour matching.                                #
#-------------------------------------------------------------------------------------------#
#a)
 #Apply the colour profile of the reference image in Figure 2 (reference.jpg) to the resized
 #colour face image used in Part 1 (a) (we will refer to this as the input image). Plot the
 #histograms of all three images (reference, your colour input image, and matched output
 #image). Comment on your resultant matched image and histogram.


#Reading in the reference image.
ref = io.imread(os.path.join(image_dir, 'reference.jpg'))

#Defining a function that resizes the reference image to 512 x 512 pixels.
def Reference (Reference):
    resizeimg = resize(Reference, (512, 512))
    return resizeimg

#Histogram matching function.
def HistogramMatch (source, reference):
    matchedhist = match_histograms(source, reference, multichannel = True)
    return matchedhist

imgRef = Reference(ref)
imgMat = HistogramMatch(imgR, imgRef)

#Rendering the image plots
fig, ax = plt.subplots(ncols=3, figsize=(24, 9))

ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)
ax[2] = plt.subplot(1, 3, 3, sharex=ax[0], sharey=ax[0])

ax[0].imshow(imgR, cmap=plt.cm.gray)
ax[0].set_title('Original Image (512 x 512)', fontsize = 20, color = 'blue')
ax[0].axis('off')

ax[1].imshow(imgRef, cmap=plt.cm.gray)
ax[1].set_title('Reference Image (512 x 512)', fontsize = 20, color = 'red')
ax[1].axis('off')

ax[2].imshow(imgMat, cmap=plt.cm.gray)
ax[2].set_title('Matched Image (512 x 512)', fontsize = 20, color = 'green')
ax[2].axis('off')

fig.tight_layout()

# save figure
plt.savefig(os.path.join(image_dir, 'part3a'))

plt.show()
.
.
.
.
.
.
.
.
```

```
#Plotting the Histograms, Cumalative Histograms and RGB channels.
fig, ax = plt.subplots(nrows = 3, ncols=3, figsize=(24, 24))

for i, images in enumerate((imgR, imgRef, imgMat)):
    for z, colour in enumerate (('Red', 'Green', 'Blue')):

        img_hist, bins = exposure.histogram(images[..., z], source_range = 'dtype')
        img_cdf, bins = exposure.cumulative_distribution(images[..., z])

        ax[z, i].plot(bins, img_hist / img_hist.max())
        ax[z, i].plot(bins, img_cdf)
        ax[z, 0].set_ylabel(colour)

ax[0, 0].set_title('Source Image', fontsize = 30)
ax[0, 1].set_title('Reference Image', fontsize = 30)
ax[0, 2].set_title('Matched Image', fontsize = 30)
ax[0, 0].set_ylabel('Red', fontsize = 30, color = 'Red')
ax[1, 0].set_ylabel('Green', fontsize = 30, color = 'Green')
ax[2, 0].set_ylabel('Blue', fontsize = 30, color = 'Blue')

plt.tight_layout()

plt.savefig(os.path.join(image_dir, 'part3a1'))

plt.show()
```