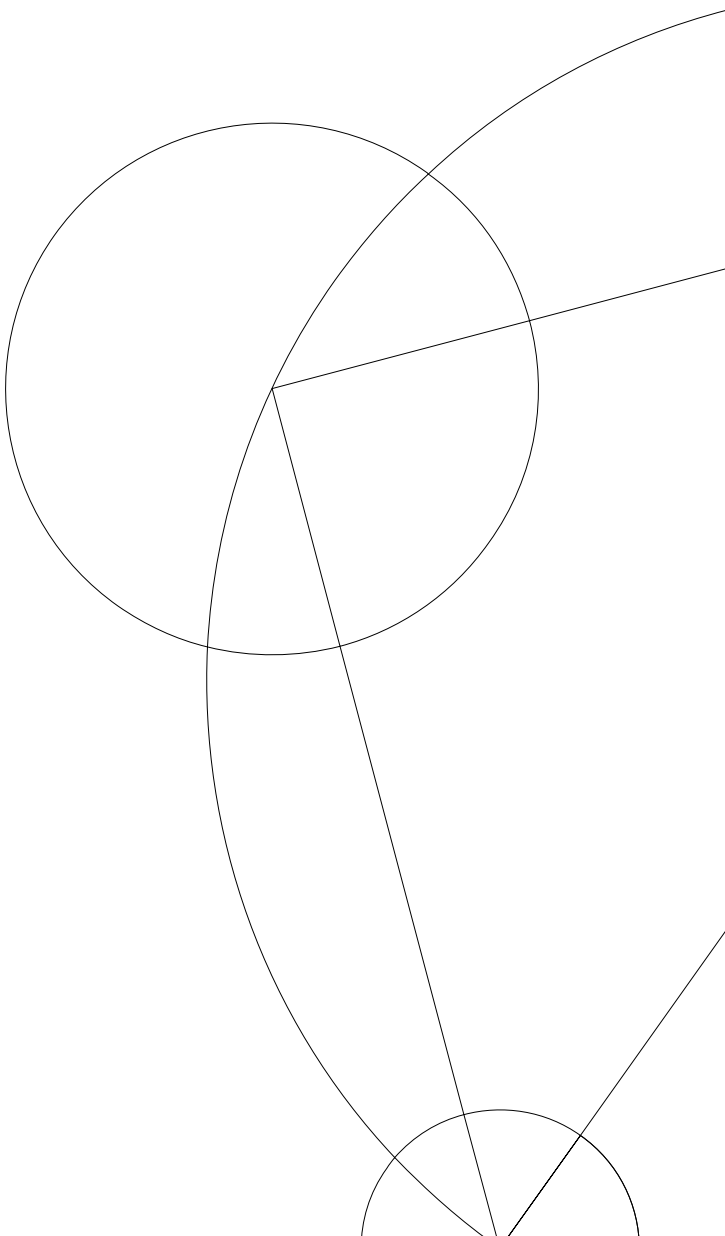
KØBENHAVNS UNIVERSITET

Signal and image processing Assignment 2

mnp553 tlq412 wvm400



21. februar 2022

1 Pixel-wise contrast enhancement

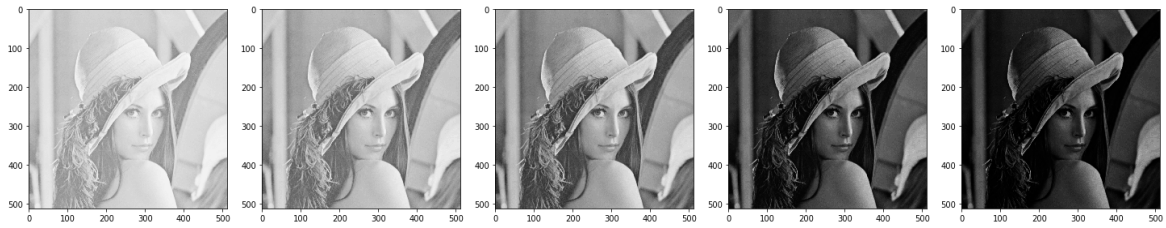
1.1

The implementation of the gamma-correction function is as follows:

```
def gamma_transform(image, gamma, c = 1):
    image = img_as_float(image)
    return np.array(c*image**gamma)
```

Listing 1: Gamma-correction function

We chose the Lena gray scale image as our example.



Figur 1: results, gamma = [0.3, 0.5, 0.8, 2.0, 3.0]

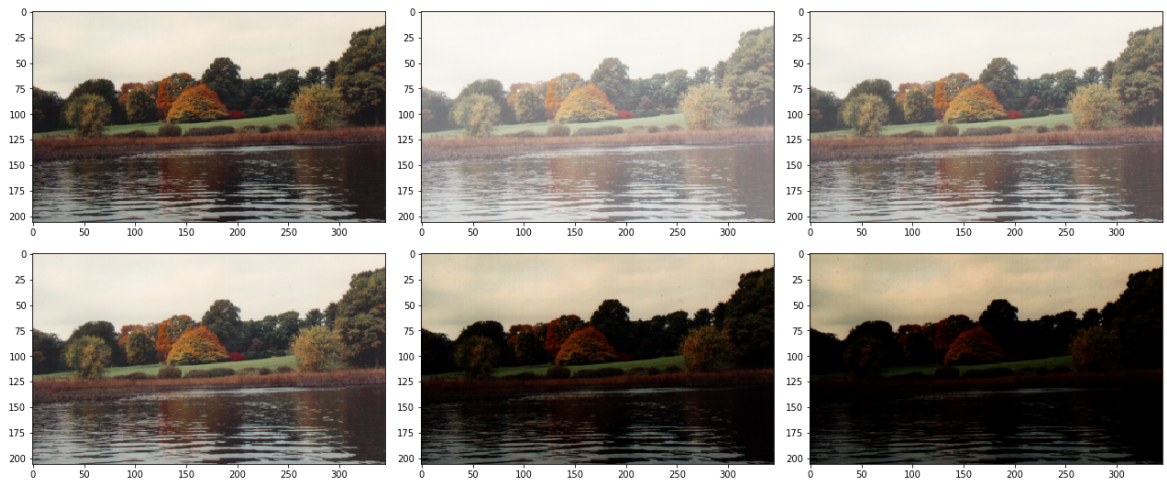
When gamma is larger than 1, the correction makes the shadows darker, while when gamma is smaller than 1, it makes dark regions lighter and more details visible.

1.2

The code is as follows:

```
autumn = imread('autumn.tif')
imshow(autumn)
imshow(gamma_transform(autumn, 0.3)) # gamma = [0.3, 0.5, 0.8, 2.0, 3.0]
```

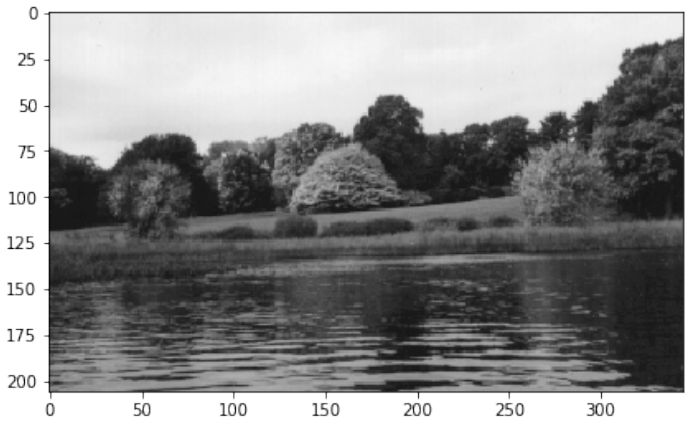
Listing 2: Apply gamma-correction on each of the RGB component separately



Figur 2: Orginal picture(top left) and results, gamma = [0.3, 0.5, 0.8, 2.0, 3.0]

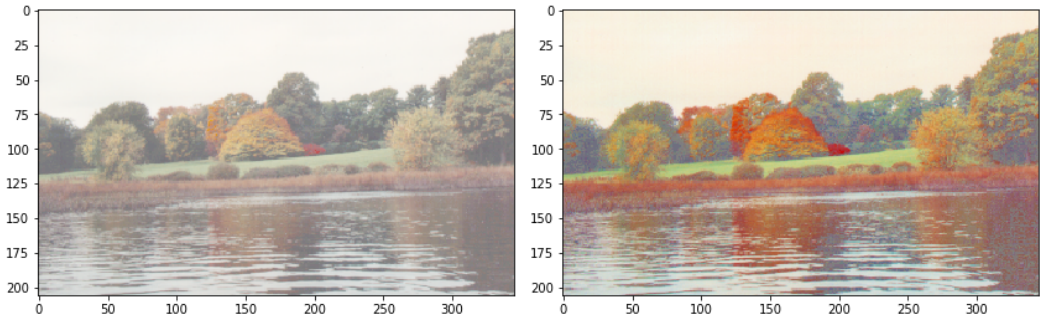
1.3

First converting to HSV color representation

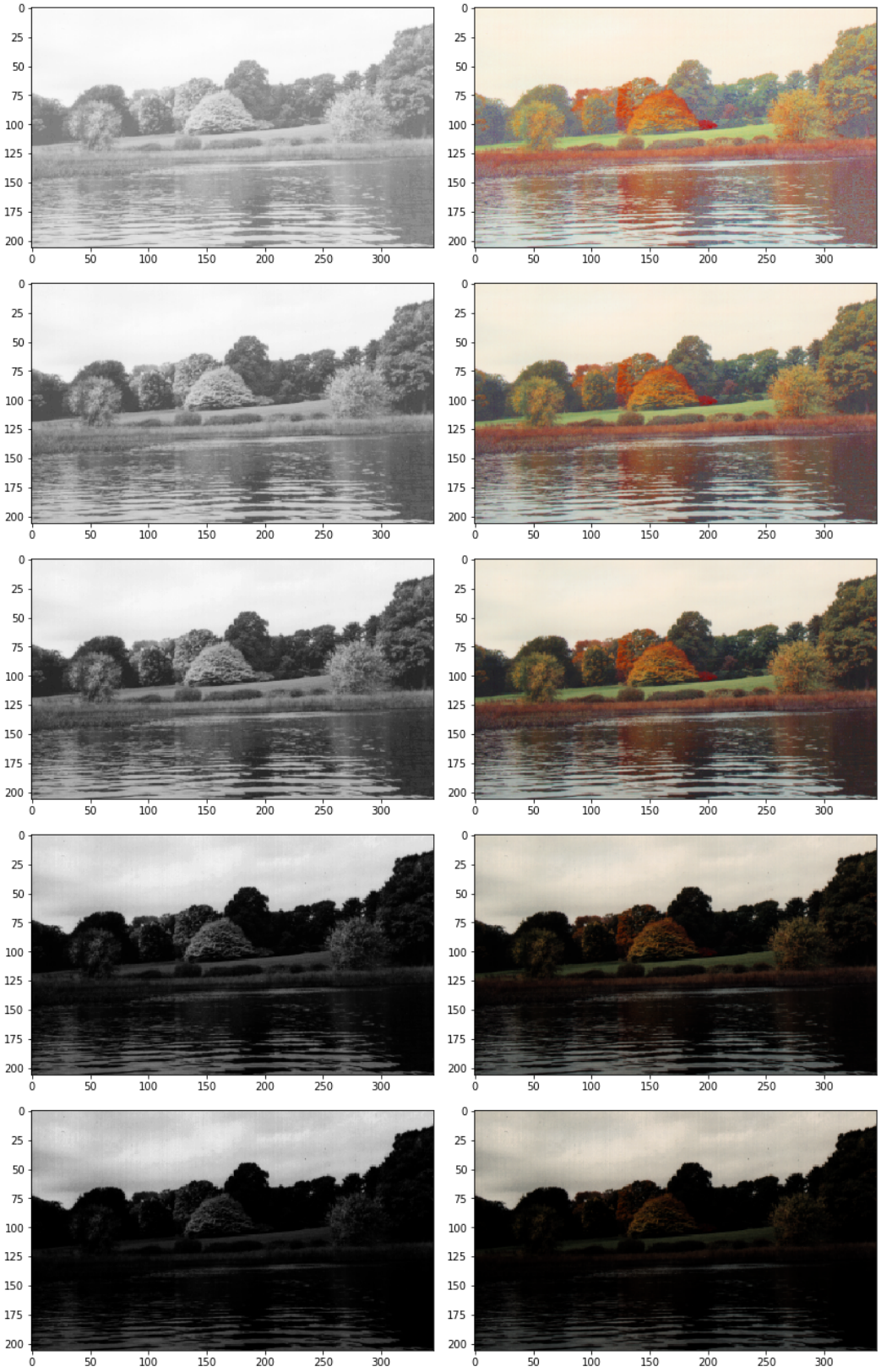


Figur 3: v-channel of original image

Then apply the gamma correction to the v-channel, and finally convert back to RGB representation(see Figure 5).
The approach used in 1.3 is better than the one used in 1.2 since it improves the details in the dark part(see Figure 4).



Figur 4: Comparison with 1.2 and 1.3 when gamma=0.3



Figur 5: v channel and corresponding RGB representation, gamma = [0.3, 0.5, 0.8, 2.0, 3.0]

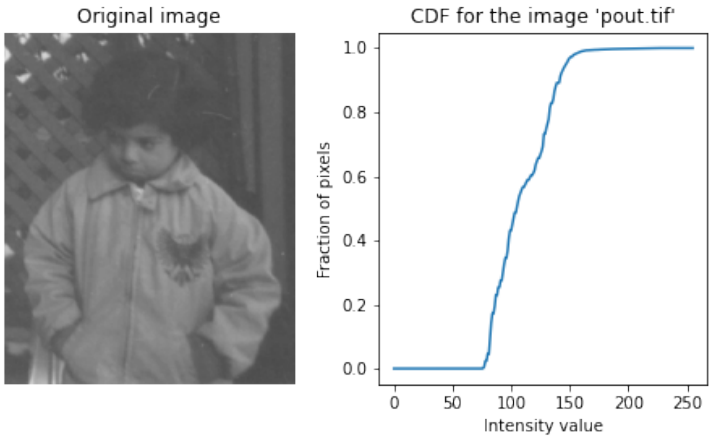
2 Histogram-based processing

2.1

The CDF for the pout.tif can be seen in Figure 6. The essential code is included in Listing 3. The regions of fast increase of the CDF means there are a lot of pixels having that value of intensity. And the flat regions means basically no pixels with that value of intensity.

```
def cumulative_histogram(hist):
  cumsum_hist = np.cumsum(hist)/np.sum(hist)
  return cumsum_hist
```

Listing 3: Code for computing cumulative histogram



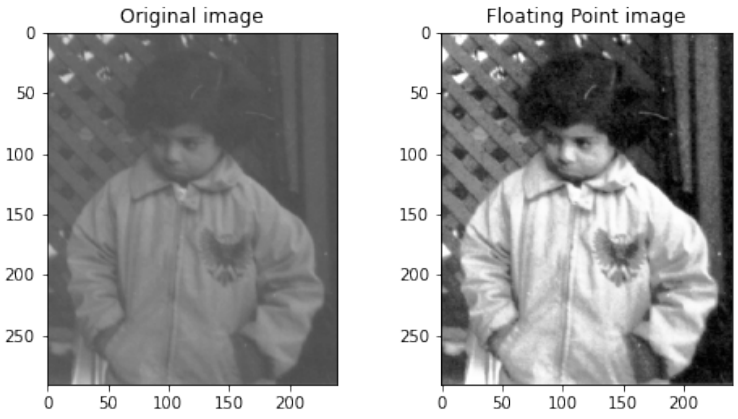
Figur 6: Original image of pout.tif and its CDF

2.2

The essential code is included in Listing 4. The graphical output can be seen in Figure 7. To show the result in gray figure, the floating-point image has been multiplied with 255.

```
def CDF_map(intensity, CDF):
    return CDF[intensity]

def Image_to_CDFimage(img, CDF):
    CDFimage = CDF_map(img, CDF)
    return CDFimage
```

Listing 4: Code for computing the floating-point image of the its CDF



Figur 7: Original image and Floating-point image of the CDF

2.3

Generally the CDF is not invertible, because in a figure may not have all values of intensity. Then the PDF of these points would be zero. This will cause some points have same value in CDF. So CDF could maps to the same value, it is not possible in all cases to calculate the exact one value that caused a certain value of CDF. Therefore, the CDF is in general not invertible. The essential code is included in Listing 5.

```
def inverse_cdf(l, cdf):
  subset  = np.take(range(256), np.where(cdf >= l))
  return np.min(subset)
```

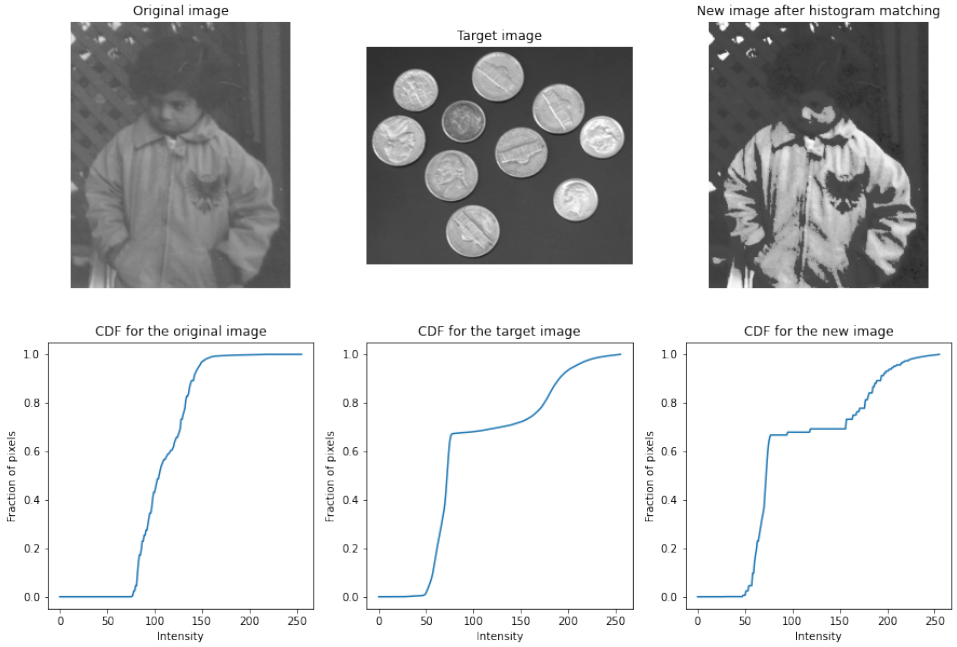
Listing 5: Code for pseudo-inverse of any given CDF

2.4

The essential code is included in Listing 9. The graphical output can be seen in Figure 8. The CDF for the new image is pretty same as the target image. But it is not as smooth as the target one because of our function of pseudo-inverse of CDF.

```
def histogram_matching(original_img, target_img):
  orignal_hist, _ = np.histogram(original_img.ravel(),256,[0,256])
  target_hist, _ = np.histogram(target_img.ravel(),256,[0,256])

  original_CDF = cumulative_histogram(orignal_hist)
  target_CDF = cumulative_histogram(target_hist)
  original_CDFimg = Image_to_CDFimage(original_img, original_CDF)

  (X,Y) = original_img.shape
  output_img = np.zeros((X,Y))
  for x in range(0,X):
    for y in range(0,Y):
      output_img[x,y] = inverse_cdf(original_CDFimg[x,y], target_CDF)
  return output_img
```

Listing 6: Code for histogram matching



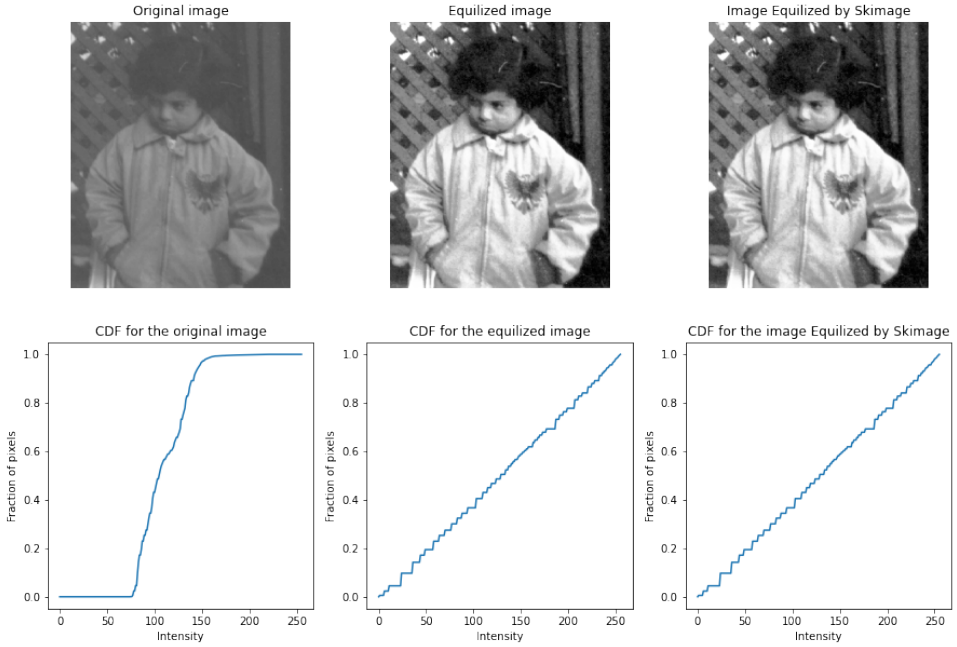
Figur 8: Two input and the result of histogram matching images with its CDF

2.5

The essential code is included in Listing 7. The graphical output can be seen in Figure 9. The result of histogram equalized image generated by Skimage is a floating-point image. Its range is $[0, 1]$. To show the figure of histogram equalized image by Skimage, the output image has been multiplied with 255. Our result is basically same as applying the skimage.exposure.equalize function.

```
def histogram_equalization(img):
    target = np.arange(0, 256)
    new_img = histogram_matching(img, target)
    return new_img
```

Listing 7: Code for histogram equalization

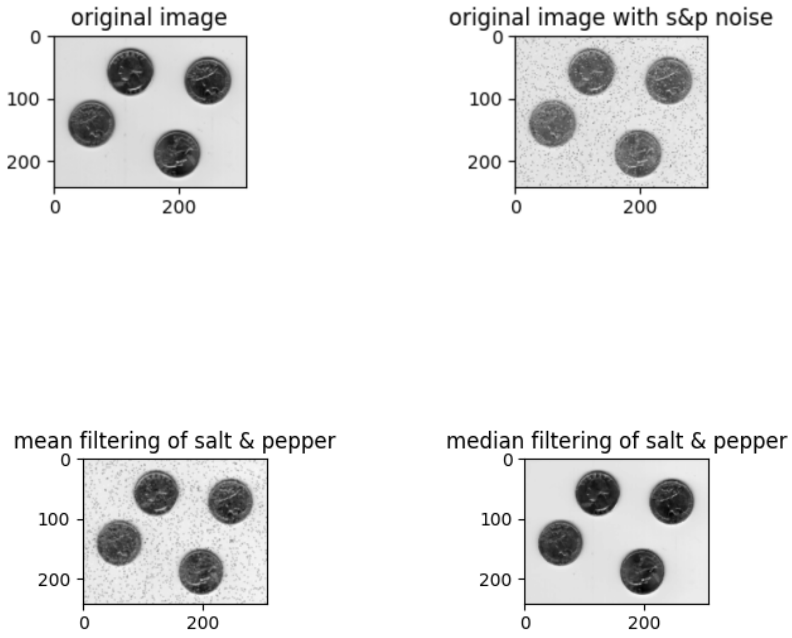


Figur 9: Original image and histogram equalized image by our function and Skimage with their CDF

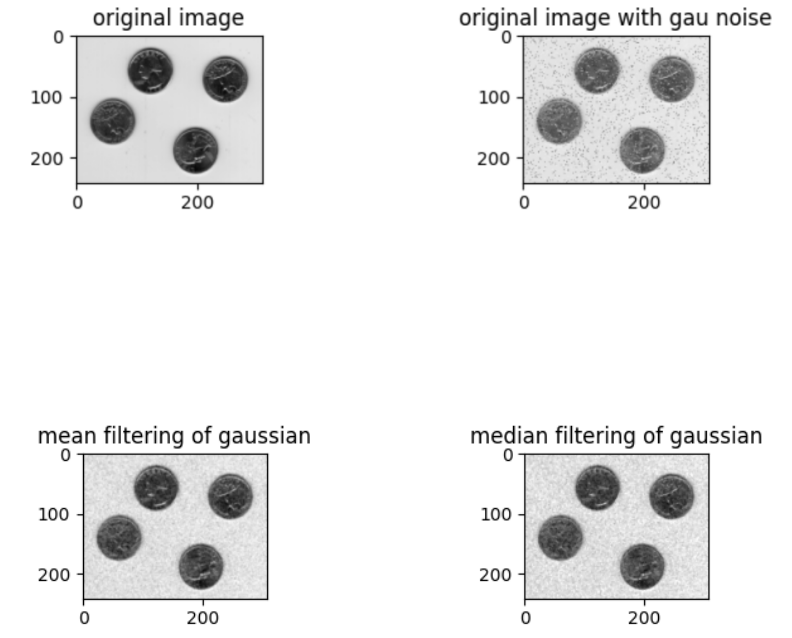
3 Image filtering and enhancement

3.1

The filtered images after the effect of mean and median filtering on the salt and pepper and gaussian noise versions of `eight.tif` is as follow,



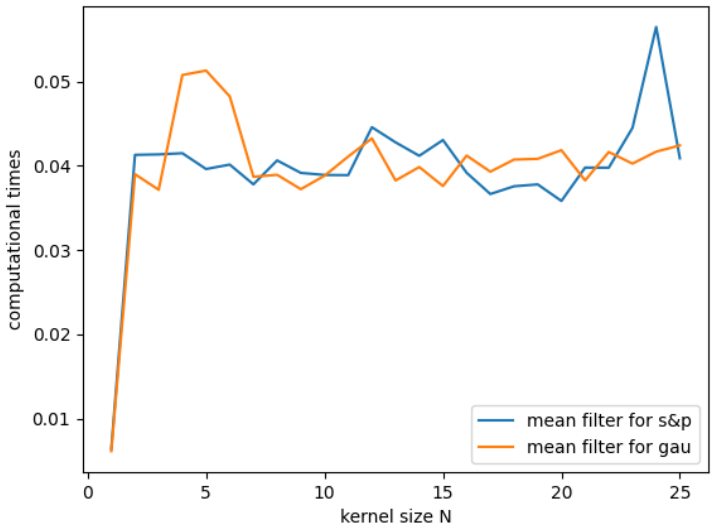
Figur 10: mean filtering and median filtering for salt & pepper noise image



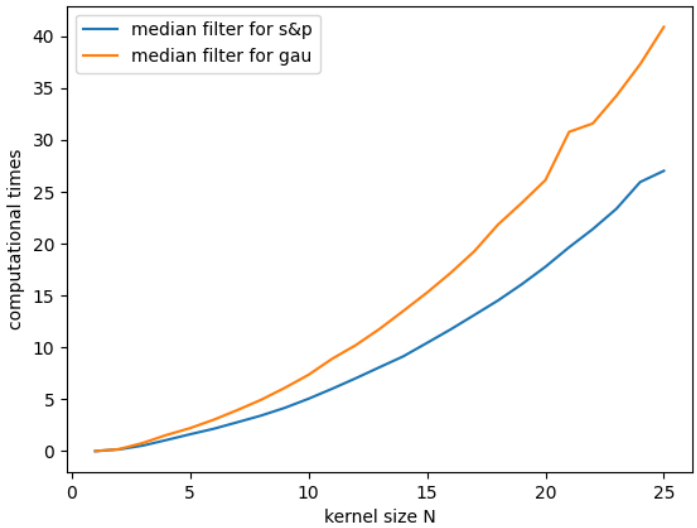
Figur 11: mean filtering and median filtering for gaussian noise image

As we can see, for the salt and pepper noise version, the performance of median filtering is much better than mean filtering.
For the gaussian noise version, they both perform bad.

I think computational times will be increasing with increasing the kernel size N, since obviously the larger kernel size N, it causes more computations for every pixel.
Below is the plot of the different computation times obtained for $N = 1$ to $N = 25$ with each time for 100 executions, the former is for mean filtering, the latter is for median filtering.



Figur 12: comparison of mean filter between s&p and gaussian noise image obtained N for 1 to 25

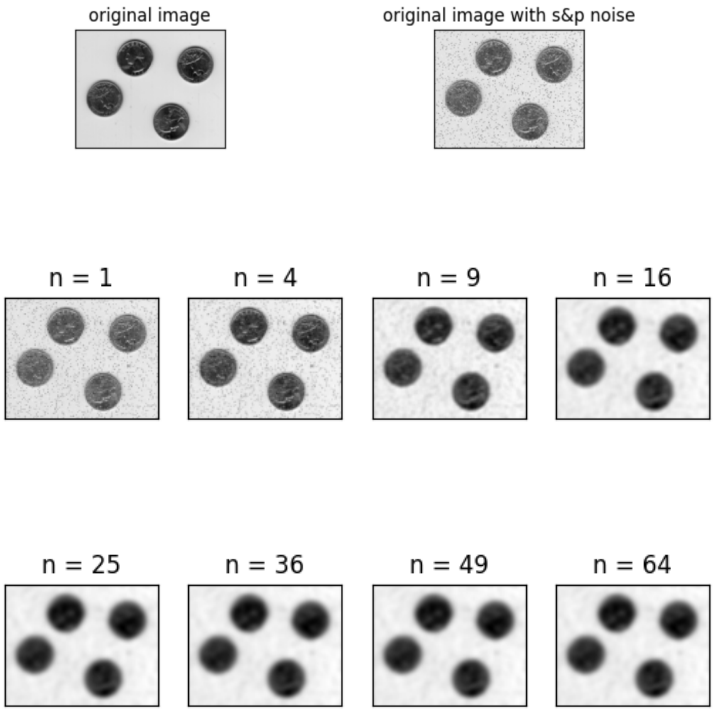


Figur 13: comparison of median filter between s&p and gaussian noise image obtained N for 1 to 25

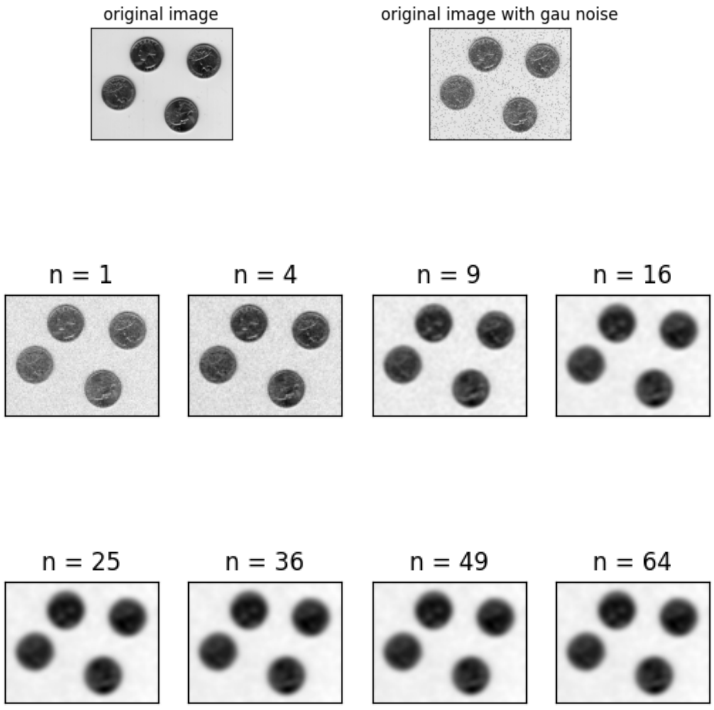
Compare with these 2 plots, apparently we can see mean filtering is far faster than median filtering. Observing each single plot, for mean filtering, after the first few increases, the computation times fluctuates in a range, the range remains the same. For median filtering, it keeps increasing stably.

3.2

The new images after Gaussian filter with increasing kernel size N is as below,



Figur 14: Gaussian filter for s&p as increasing kernel size N with $\sigma = 5$

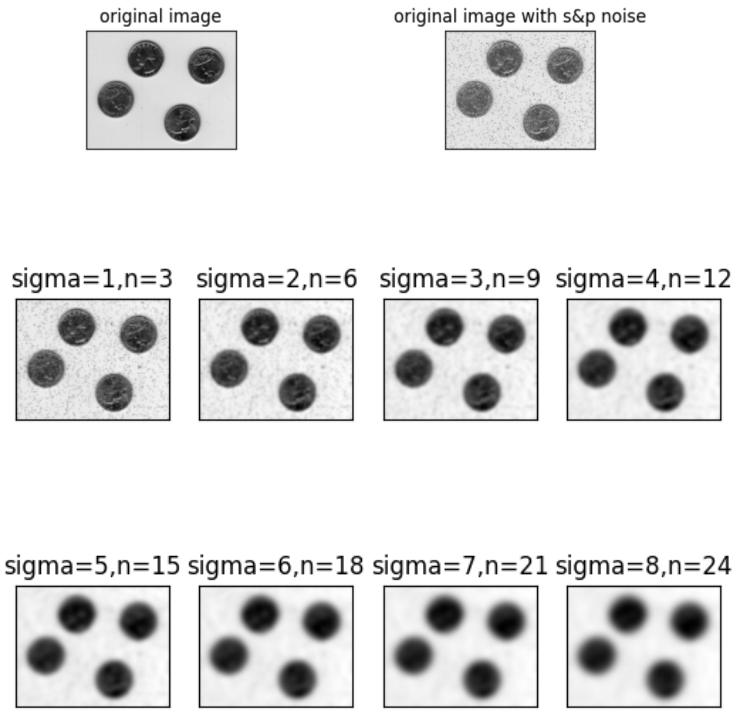


Figur 15: Gaussian filter for gaussian noise as increasing kernel size N with $\sigma = 5$

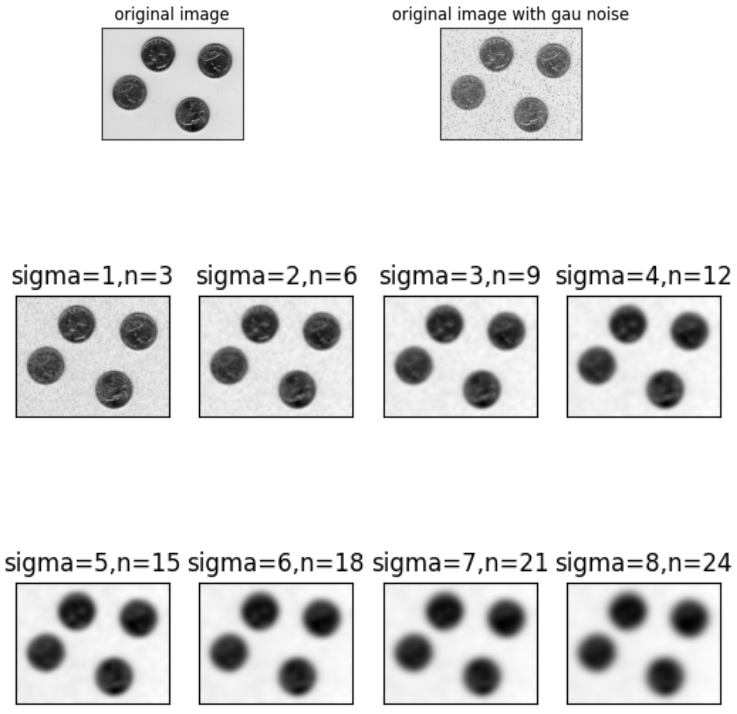
by comparing these 2 plots, we can see gaussian filter performs well for the gaussian noise rather than salt and pepper noise. Besides, by increasing the kernel size N with a fix deviation σ, we observed that the image becomes more and more blurred. The reason behind this is when the kernel size N increases, it smooths over more pixels. So each pixel in new image depends on more surrounding pixels, which makes image smoother.

3.3

The new images after Gaussian filter with increasing both σ and kernel size $N(\sigma = 3N)$ is as below,



Figur 16: Gaussian filter for s&p as increasing both N and σ



Figur 17: Gaussian filter for gaussian noise as increasing both N and σ

We already know that the image blurs as the kernel size N. However, as we can see from the plot, it blurred more seriously when they increases at the same time.
This reason is, as we know σ is the deviation over its surrounding pixel. So as the σ becomes larger the more variance allowed for those pixels around it, which means we will see more expansive scene rather than to care for the other details, namely more blurred.

4 Advanced filtering

4.1

Bilateral filter is a non-linear filter. The bilateral filter has one more Gaussian filter which is a function of pixel difference, $g_\tau(I(x+i, y+i) - I(x, y))$. The parameter τ determines to what extent should we preserve the edge in the image. The bilateral filtering becomes the usual Gaussian filtering when $\tau \to \infty$.

4.2

The implementation of our bilateral filtering is divided into 2 parts. The algorithm is as below,

```
def bilateralFilter(img,k,sigma,tau):
    w,h= img.shape
    size = k // 2

    # padding process for boundary pixels
    _img = np.zeros((w+2*size,h+2*size), dtype=float)
    _img[size:size+w,size:size+h] = img.copy()
    dst = _img.copy()

    #Filtering process
    for x in range(w):
        for y in range(h):
            dst[x+size,y+size] = filter_1pixel(_img, x+size, y+size, size, sigma, tau
    )

    dst = dst[size:size+w,size:size+h]

    return dst
```

Listing 8: Code for the window shifting with bilateral filtering

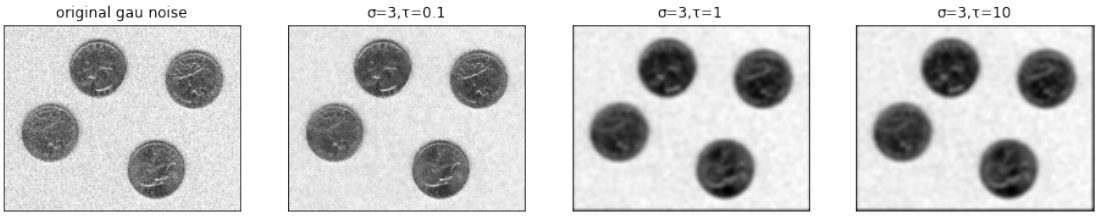
As we can see, firstly we should add some padding for those boundary pixels, since when compute the new boundary pixels, it will be over original size by $k//2$, so we fill these exceeded pixels with 0. The second step is just to compute new filtered pixels by `filter_1pixel()`. This function is implemented strictly according to the formula given. The detail is as follow,

```
def f(x, y, sigma):
    return math.exp(-(x**2+y**2)/(2*(sigma**2)))

def g(u, tau):
    return math.exp(-(u**2)/(2*(tau**2)))
def w(img, x, y, i, j, sigma, tau):
    return f(i, j, sigma)*g(img[x+i][y+j]-img[x][y], tau)

def filter_1pixel(img,x,y,size,sigma,tau):
    denmonator = 0
    numerator = 0
    for i in range(-size, size+1):
        for j in range(-size, size+1):
            w_result = w(img, x, y, i, j, sigma, tau)
            numerator += w_result*img[x+i][y+j]
            denmonator += w_result
    return numerator/denmonator
```

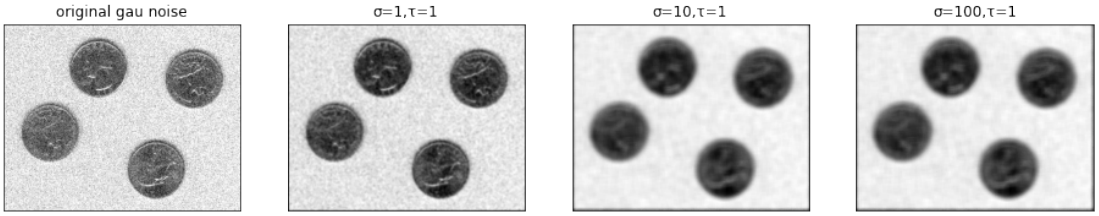
Listing 9: Code for the computation of just one pixel in bilateral filtering

4.3

The plots Below is applying bilateral filtering on the image `eight.tif` corrupted by Gaussian noise with gradually increasing σ and τ with kernel size $N = 9$.



Figur 18: Effect of bilateral filtering on the gaussian noise image for $\sigma = 3$ and increasing τ



Figur 19: Effect of bilateral filtering on the gaussian noise image for $\tau = 3$ and increasing σ

For bilateral filtering, When increasing τ, we could see in Figure 18, the images became smoother and if $\tau \to \infty$ it would become Gaussian filter. Edges are lost with high values of τ since more averaging is performed.
As we can see easily, bilateral filtering gets a good sharpness, in contrast, the Gaussian filtering causes the picture to become increasingly blurred as the figure 17 shown.
Hence, bilateral filtering is much better than Gaussian filtering.Bilateral filtering prevents averaging across edges. It will avoid the introduction of blur between objects while still removing noise in uniform areas.