Signal and image processing
Assignment 2

tkz347
Casper Bresdahl - Whs715

February 22, 2021

1 Introduction

For all images in this report, after reading the images we cast them to a numpy array to simplify further processing. This means that in the following, when we refer to the images we usually refer to the numpy arrays of the images.

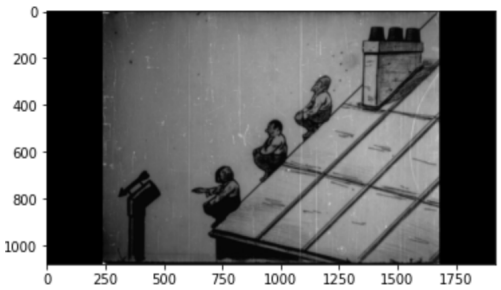
Exercise 1

1.1

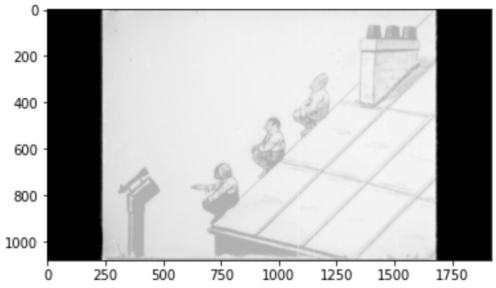
```
def gamma_map(im, gamma = 1.0):
    return (pow(im*(1/255), gamma)*255).astype(np.uint8)
```

Figure 1: Implementation of the gamma function.

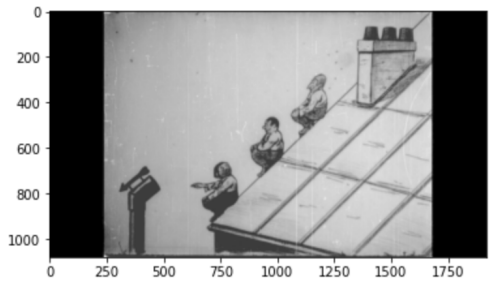
Our implementation of the gamma function can be seen in Figure 1, and the filter applied to an example image (*'movie_flicker1.tif'*) can be seen in Figure 2. When we use a gamma value $0 < \gamma < 1$ we constraint the low intensities while



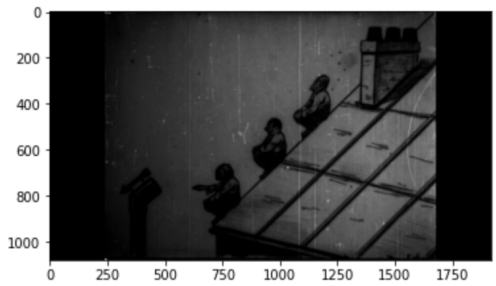
(a) Original image



(b) $\gamma = 0.1$



(c) $\gamma = 0.4$



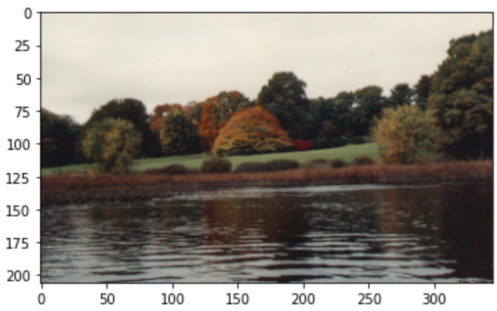
(d) $\gamma = 2.0$

Figure 2: Sample image and effect of applying the gamma filter.

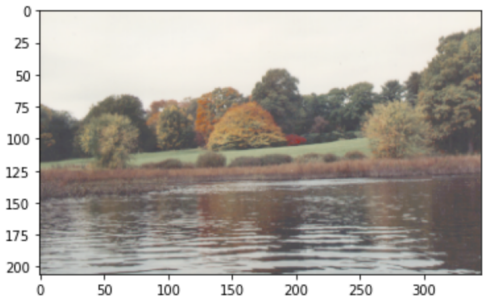
stretching the high intensities thus making the image brighter. When choosing $\gamma > 1$ we do the opposite, namely constraint the high intensities while stretching the low intensities thus making the image darker.

1.2

To apply our gamma function to each of the color channels we simply call our *gamma_map* function on the color image, and numpy's broadcasting ensures we apply the function to each channel. The resulting image can be seen in Figure 3.



(a) Original image

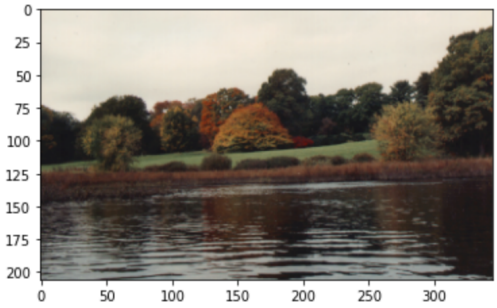


(b) $\gamma = 0.5$

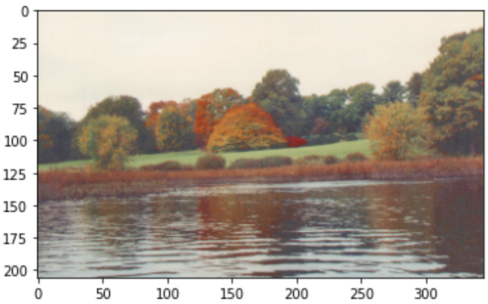
Figure 3: Autumn image and autumn image filtered with the gamma filter on all color channels.

1.3

Applying the gamma correction to the value channel can be seen in Figure 4.



(a) Original image



(b) $\gamma = 0.5$

Figure 4: Autumn image and autumn image filtered with the gamma filter on the value channel.

Comparing the two resulting images we see that the color image becomes more mat, whereas the hsv image keeps a lot more color, especially we note the red leaves in the middle of the image. Because the image keeps a lot more color we find the hsv approach provides the best results. This is note entirely unexpected as the hsv color scheme represents colors a lot more closely to what the human eye perceive, and so, filtering only on the color values would preserve more color, and result in more contrast between colors.

*Exercise 2

2.1

```python
def hist_homebrew(arr, bins):
    count = np.zeros(bins)
    hist = np.arange(bins)
    for i in hist:
        count[i] = sum(arr[arr == i])
    return count, hist


def hist_pre_processing(im, bins = 256):
    count, intensity = hist_homebrew(np.ravel(im), bins=bins)

    prob = count/sum(count)
    return (((bins-1) * np.cumsum(prob)) / (bins-1)), intensity
```

Figure 5: Implementation of the histogram and cdf functions.

In Figure 5 we can see the *hist_pre_processing* function which takes an image and first computes its histogram by using the function *hist_homebrew* and then computes the cumulative histogram (the cdf). We have chosen to let *hist_pre_processing* take an image instead of a histogram to simplify the following exercises.
The cdf of *'pout.tif'* can be seen in Figure 6.

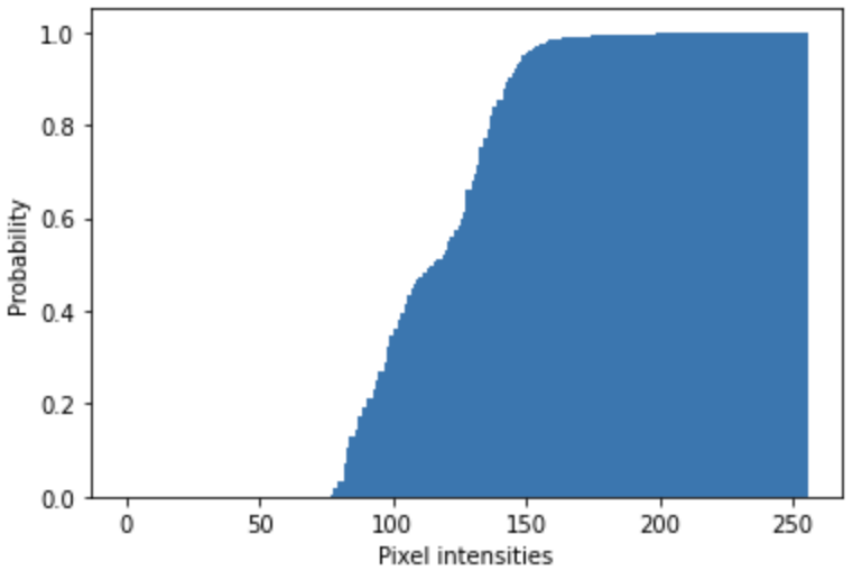


Figure 6: Cdf of *'pout.tif'*.

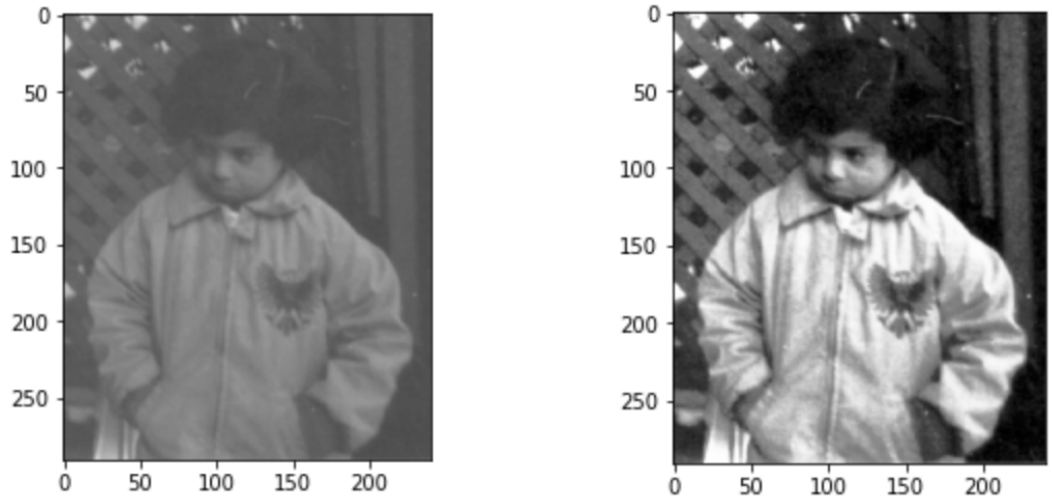
The cdf can be interpreted as the probability of seeing a value less than or equal to some value. That is, when the cdf is flat, there are no values and thus the probability does not change. However, when the cdf is steep, it means there are a lot of values in that range, and so the probability rapidly increases to see value which is less or equal to it. To give a concrete example, in Figure 6 we see flat spots in the range 0 to about 75. That is, there are no to few pixel intensities in that range in the *'pout.tif'* image. However, in the range 75 to about 150 we see a steep incline, which means a lot of pixel have intensities in that range.

2.2

```
def hist_processing(im, bins = 256):
    c = im.copy()
    sums, intensity = hist_pre_processing(c, bins)
    for i, s in zip(intensity, sums):
        c = np.where(im == i, s*(bins-1), c)

    return np.round(c)

plt.imshow(hist_processing(A4) / 255, cmap='gray', vmin=0, vmax=1)
```

Figure 7: Implementation of histogram equalization.

In Figure 7 we see our implementation for histogram equalization. In Figure 8 we see the effect of applying histogram equalization.



(a) Original image

(b) Histogram equalization.

Figure 8: The original *'pout.tif'* image and *'pout.tif'* after histogram equalization.

2.3

In general the cdf is not invertible due to the possible flat spots where it would be ambiguous which value the inverse should map to. It is for this reason we decide to always map to the lowest value in the pseudo inverse.
In Figure 9 we see our implementation of the inverse cdf. Here we assume a cdf is a tuple of the cumulative sum and the intensity range.

```
def inverse(cdf, l = 0.5):
    s, i = cdf
    return np.min(i[s >= l])
```

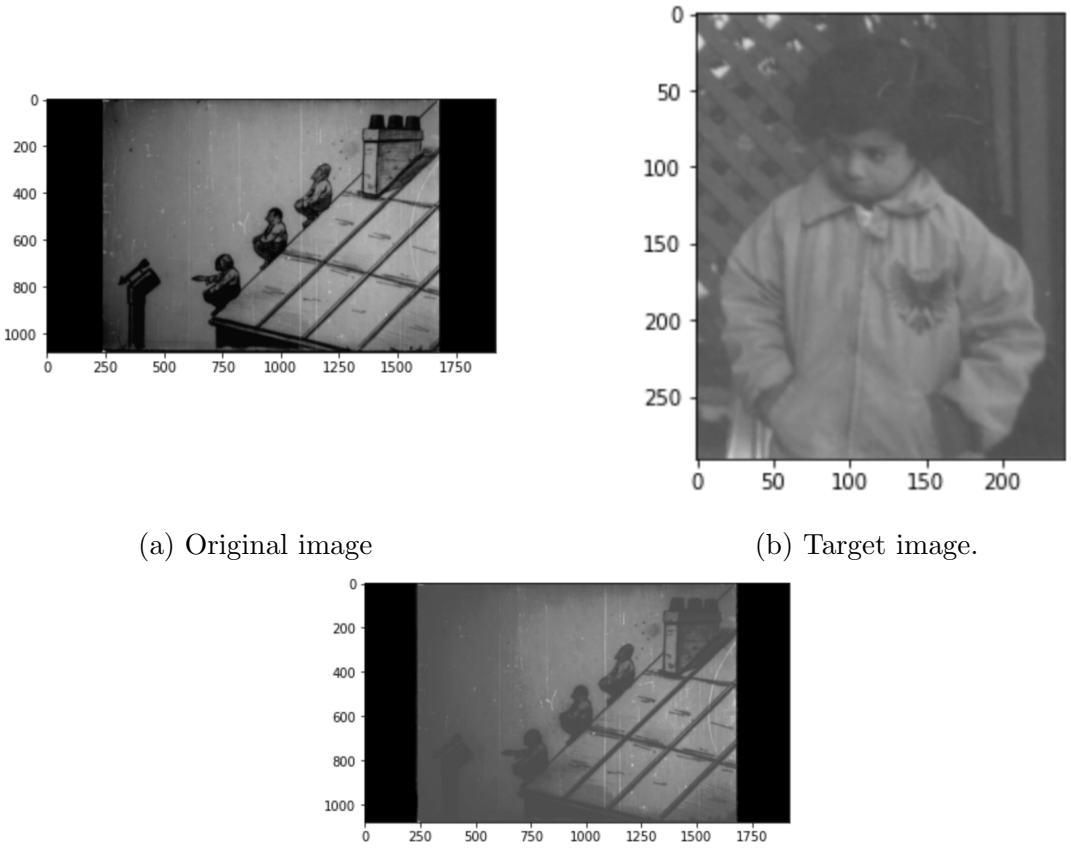
Figure 9: Implementation of the inverse cdf.

2.4

```python
def hist_match(im1, im2):
    cdf1,_ = hist_pre_processing(im1)
    cdf2 = hist_pre_processing(im2)

    J = im1.copy()
    for x in range(len(im1)):
        for y in range(len(im1[0])):
            try:
                J[x,y] = inverse(cdf2, l = cdf1[im1[x,y]])
            except ValueError:  #raised if `y` is empty.
                J[x,y] = 255
    return J
```

Figure 10: Implementation of the histogram matching.

In Figure 10 we see our implementation of histogram matching. An application of the function can be seen in Figure 11. The resulting histograms can be seen in Figure 12.

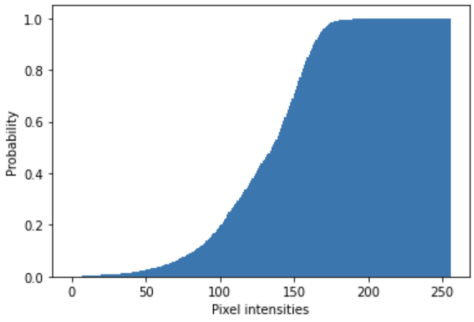


(a) Original image

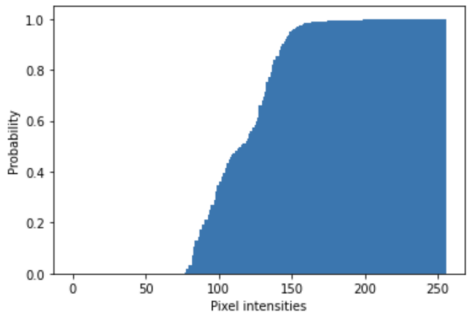
(b) Target image.

(c) Histogram matching result.

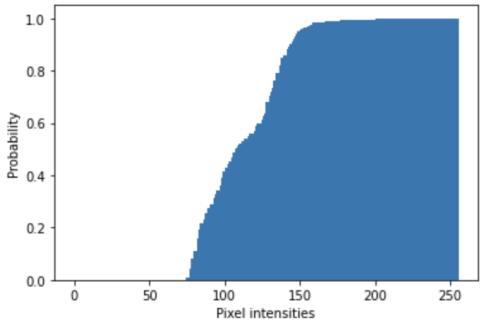
Figure 11: Histogram matching between *'movie_flicker1.tif'* and *'pout.tif'*.



(a) Histogram for original image



(b) Histogram for target image.



(c) Histogram matching result.

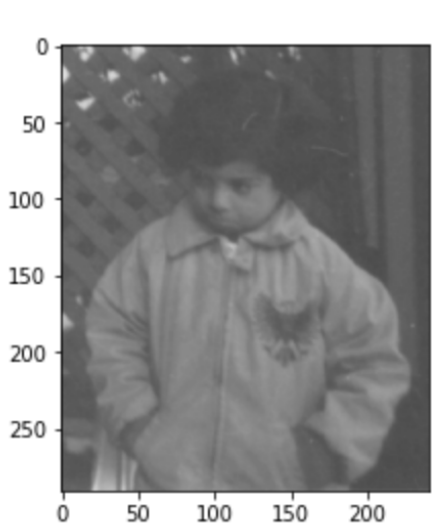
Figure 12: Histogram matching between *'movie_flicker1.tif'* and *'pout.tif'*.

2.5

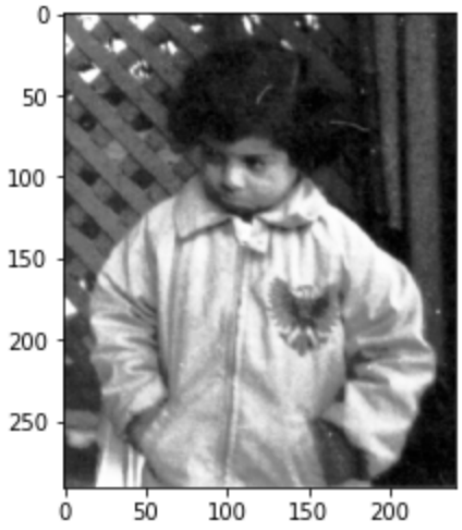
```python
def hist_match_uni(im1):
    cdf1,_ = hist_pre_processing(im1)
    uni = np.full((256), 1/256)
    unicdf = np.cumsum(uni)

    J = im1.copy()
    for x in range(len(im1)):
        for y in range(len(im1[0])):
            try:
                l = cdf1[im1[x,y]]
                I = np.min(np.where(unicdf >= l))
                J[x,y] = I
            except ValueError:  #raised if `y` is empty.
                J[x,y] = np.average(J)
    return J
```

Figure 13: Implementation of the histogram matching with an uniform histogram.

In Figure 13 we see our implementation of histogram equalization with an uniform histogram. In Figure 14 we see the effect of applying histogram equalization with a uniform histogram. As expected the result is the same as histogram equalization. In Figure 15 we see the corresponding histograms.

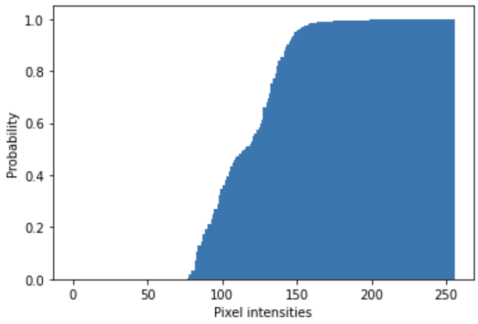


(a) Original image

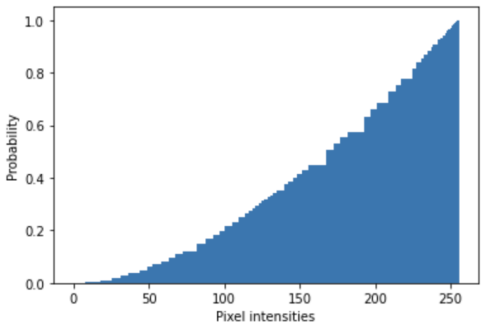


(b) Histogram equalization with uniform histogram.

Figure 14: The original *'pout.tif'* image and *'pout.tif'* after histogram equalization with an uniform histogram with value 1/256.



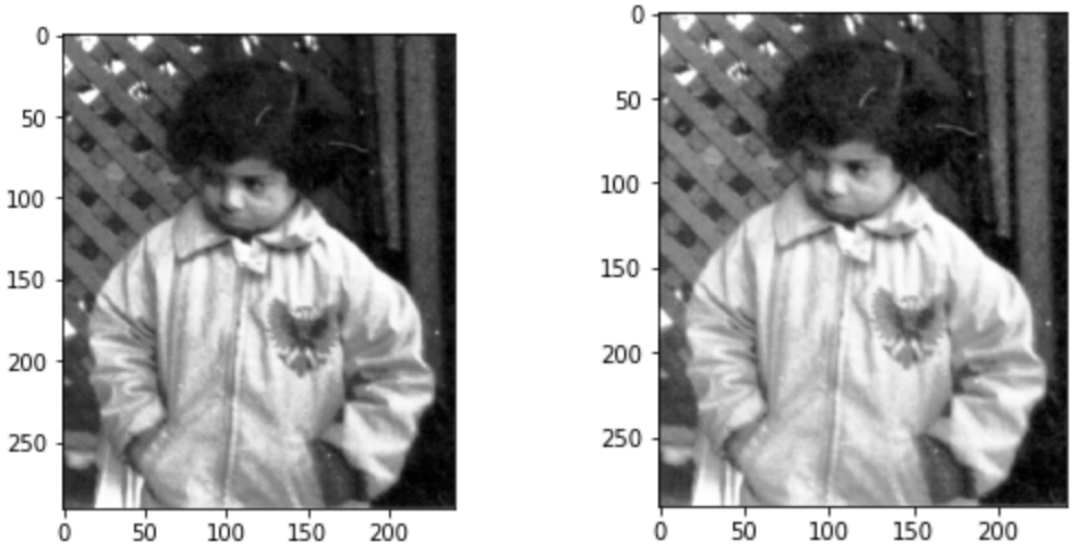
(a) Histogram of original image



(b) Histogram after histogram matching with uniform histogram.

Figure 15: Histograms of the original *'pout.tif'* image and *'pout.tif'* after histogram equalization with an uniform histogram with value 1/256.

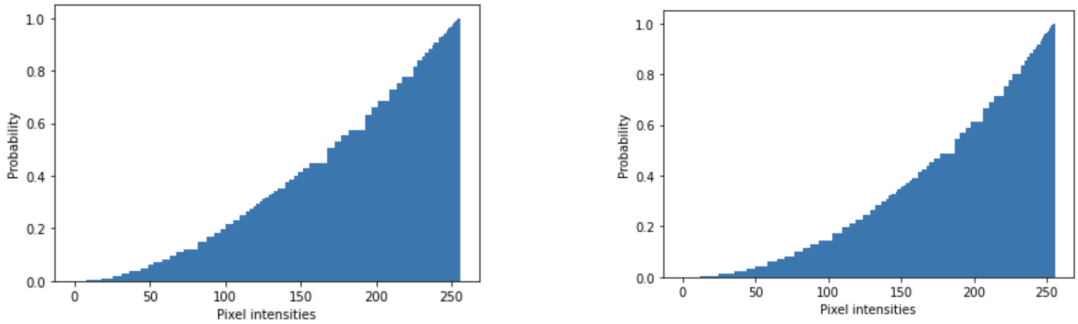
We can now compare our results with *skimage.exposure.equalize_hist*. In Figure 16 we see the resulting images. We note that the images looks very much alike with Skimage's result being a little brighter. In Figure 17 we see the resulting histograms which confirms that the results follows almost the same distribution, with Skimage's results being a little brighter.



(a) Our histogram equalization.

(b) Skimage's histogram equalization.

Figure 16: *'pout.tif'* after our histogram equalization and Skimage's histogram equalization.



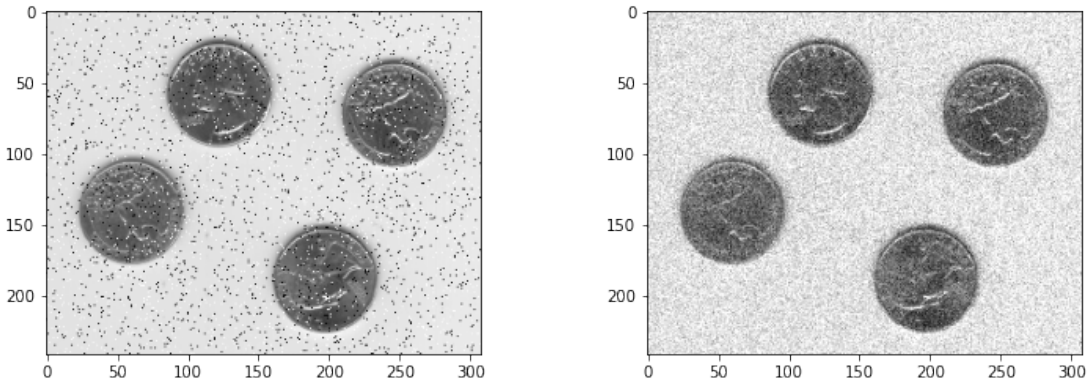
(a) Our histogram after histogram equalization.

(b) Skimage's histogram after histogram equalization.

Figure 17: Our and Skimage's histograms of the original *'pout.tif'*.

Exercise 3

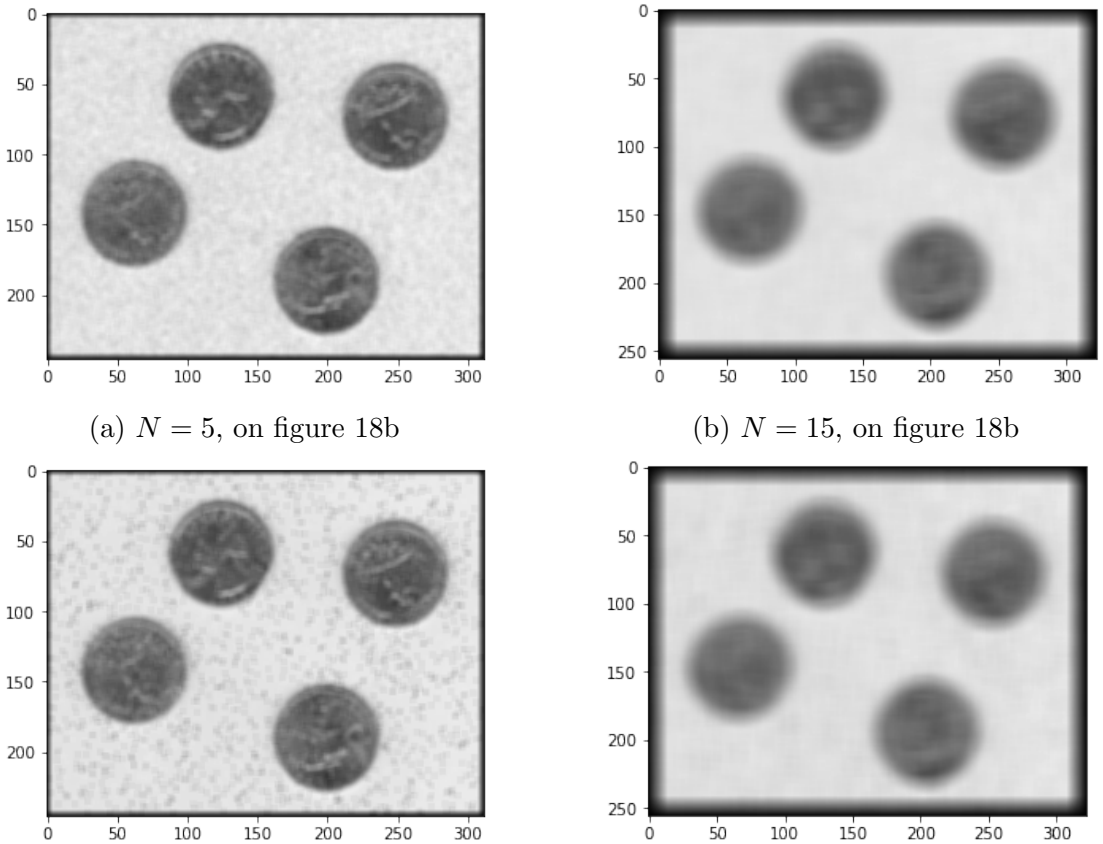
3.1



(a) Salt & Pepper noise

(b) Gaussian noise

Figure 18: The original images before filtering



(a) $N = 5$, on figure 18b

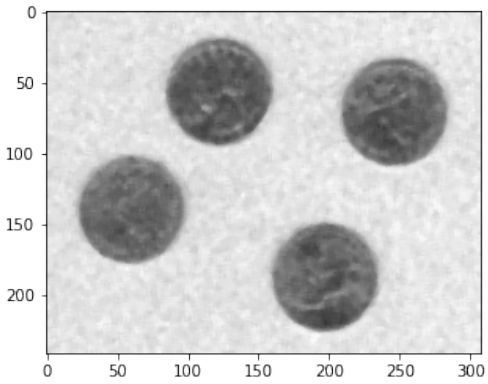
(b) $N = 15$, on figure 18b

(c) $N = 5$, on figure 18a

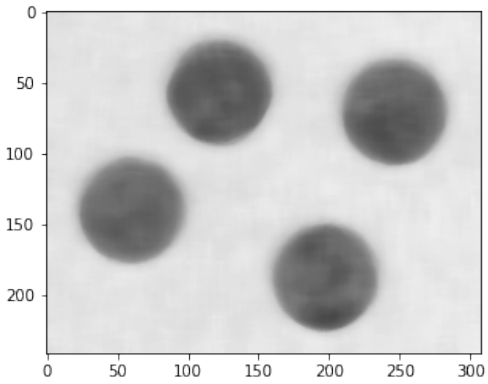
(d) $N = 15$, on figure 18a

Figure 19: Results of using mean filter.

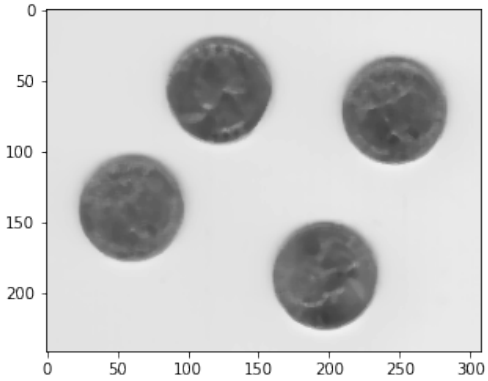
When applying the mean filter we see that at a small kernel size it works quite well on the Gaussian noise, but that it fails to remove the noise in the salt and pepper image. However once the kernel size is increased significantly enough it removes the the noise at the cost of detail.



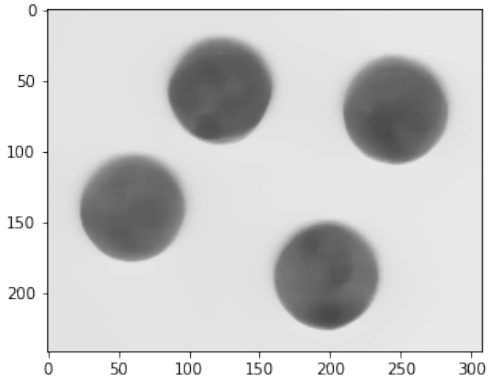
(a) $N = 5$, on figure 18b



(b) $N = 15$, on figure 18b



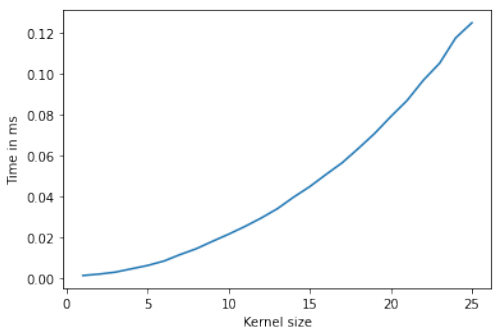
(c) $N = 5$, on figure 18a



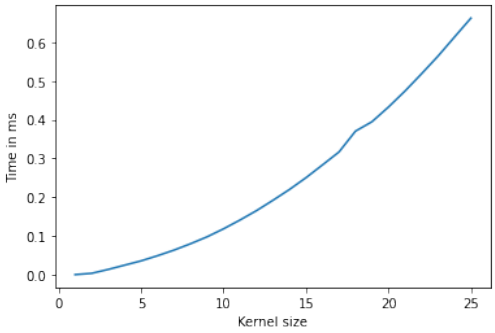
(d) $N = 15$, on figure 18a

Figure 20: Results of using median filter. We used scipy for the kernel due to optimization issues, and it did not add visible padding to the resulting image.

We see when applying our median filter that it very easily removes the salt and pepper noise, with even a very small kernel. But it does very little to the Gaussian noise. Even when scaling up the kernel it still has small areas that are not completely smoothed out.



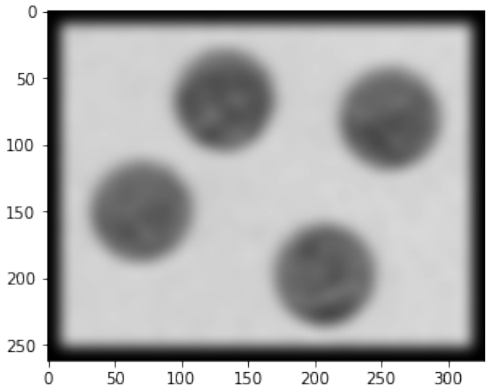
(a) Mean kernel average time over 100 rounds



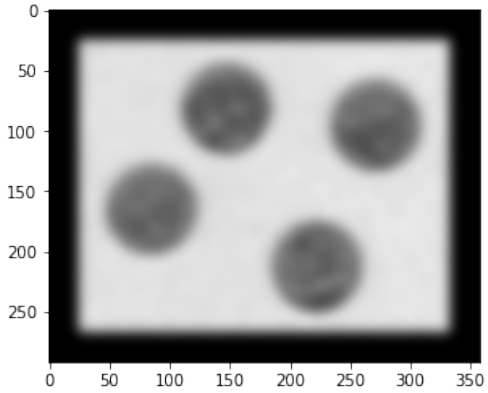
(b) Median kernel average time over 100 rounds

Figure 21: The performance of the kernels averaged over 100 runs each. They are both quite fast even at rather large kernel sizes if we use optimized kernel methods like numpy or scipy.

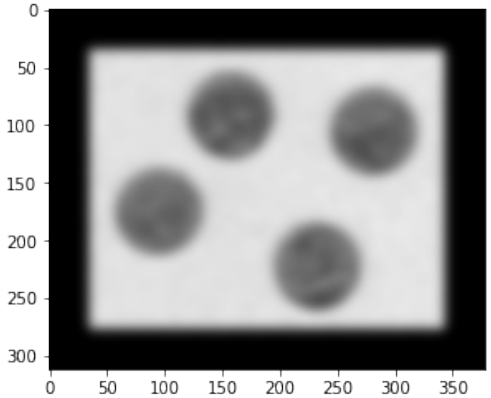
3.2



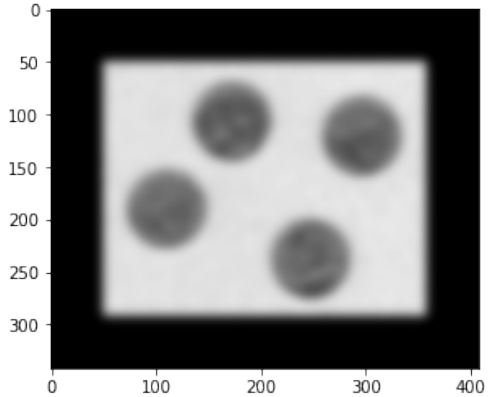
(a) $\sigma = 5, N = 10$



(b) $\sigma = 5, N = 25$



(c) $\sigma = 5, N = 35$

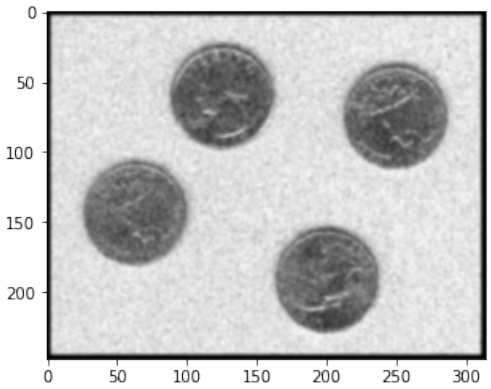


(d) $\sigma = 5, N = 50$

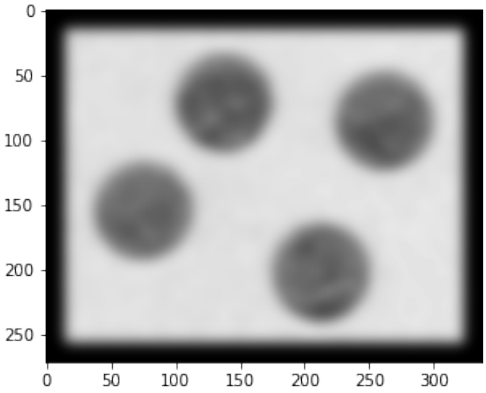
We see that as we increase N the smoothing approaches a point where increasing N has no more any real visual effect on the image aside from padding it more.

This is mainly due to σ staying constant which means the Gaussian kernel is not adding more weight to the pixels further away from the center of the kernel as the kernel size increases. This leads then to the stagnate smoothing after a large enough kernel is used.

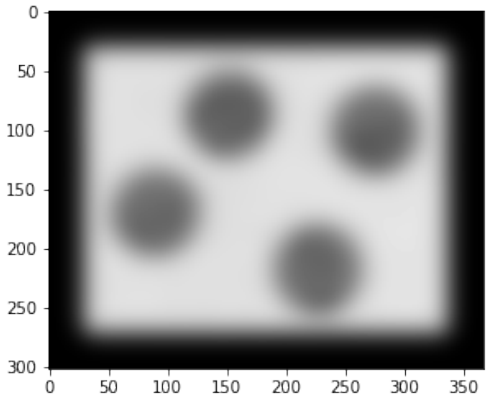
3.3



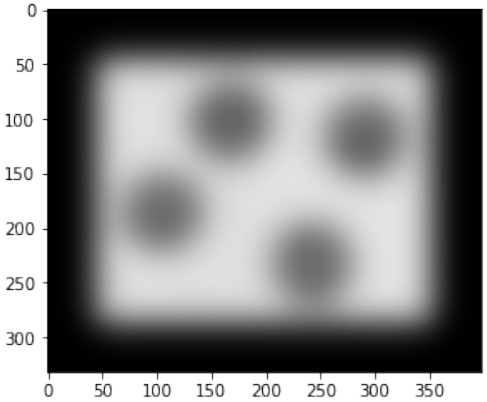
(a) $\sigma = 1, N = 3$



(b) $\sigma = 5, N = 15$



(c) $\sigma = 10, N = 30$



(d) $\sigma = 15, N = 45$

We see that increasing out σ leads to a trade off between smoothing and sharpness. If we increase σ it will lead to an overall more balanced smoothing akin to the mean filter, but if we continue increasing our sigma our N starts to grow significantly and since we continue to give high weight to pixels far from the center the smoothing becomes quite excessive.

Exercise 4

4.1

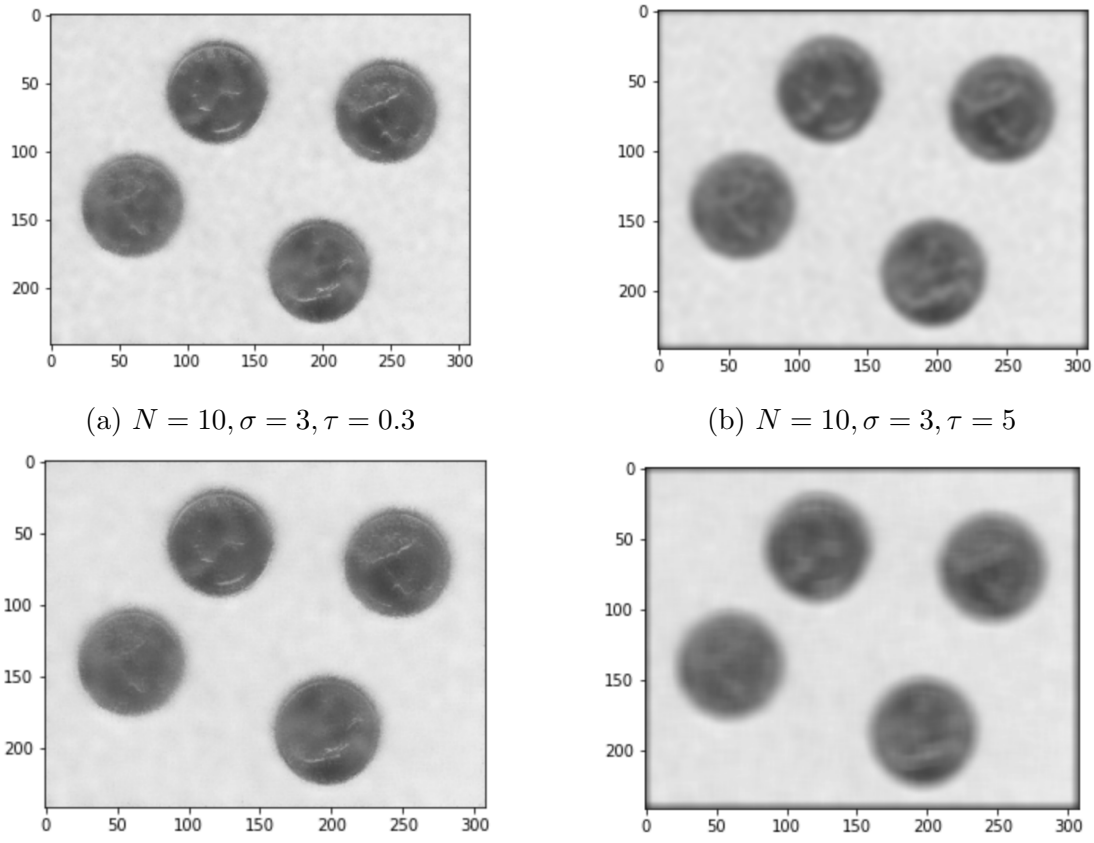
A linear filter can be described as a weighted summation of pixels, and given we can not describe this filter as a weighted summation it must be non-linear.
Bilateral filtering consists of two 'parts', a *domain kernel* and a *range kernel*. The domain kernel $f_\sigma(x, y)$ is defined as the 2D Gaussian function whereas the range kernel $g_\tau(u)$, where u is the difference in pixel intensities, is the 1D Gaussian function. The range kernel measures similarities between the center pixel and the neighbouring pixels. When τ is small we see that the fraction in g_τ becomes big, and we then take the exponential function to a big negative number which gives a response close to zero. When τ is big, the fraction becomes small, and we take the exponential function to a small negative number which gives a response close to one. The effect of τ is thus how g_τ scales f_σ, when τ is small, we get sharp edges as f_σ is scaled down, and when τ is big, we get results close to just using a Gaussian filter. Thus in the limit of $\tau \to \infty$ we get the usual Gaussian filter.

4.2

```python
def bilateral_filtering(image, n, sigma = 5, tau = 5):
    l = k = math.floor(n/2)
    im = np.pad(image.copy(), l)

    imr = image.copy()

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

    def g(u):
        return np.exp(-(u**2)/(2*tau**2))

    def w(x,y,i,j):
        o = f(i,j)
        p = g(im[x+i,y+j]-im[x,y])
        return o*p

    def I(x, y):
        s1 = 0
        s2 = 0
        for i in range(-l, l+1):
            for j in range(-k, k+1):
                v = w(x,y,i,j)
                s1 = s1 + v
                s2 = s2 + v*im[x+i,y+j]

        return s2/s1


    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            imr[x,y] = I(x+l,y+k) # Adding k and l removes padding from final image

    return imr
```

Figure 24: Python implementation of bilateral filtering

4.3



(a) $N = 10, \sigma = 3, \tau = 0.3$

(b) $N = 10, \sigma = 3, \tau = 5$

(c) $N = 10, \sigma = 20, \tau = 0.3$

(d) $N = 10, \sigma = 20, \tau = 5$

Figure 25: Results of using bilateral filtering on the Gaussian noise image with varying σ and τ.

When just comparing the differences in the variables, the τ variable controls how dramatic the contrast in blurring is around edges, specifically around the coins in Figure **??**. The σ variable controls the Gaussian blur that happens in the image, which makes the filter look very similar to that of the Gaussian filter. But the main difference is a lack of blurring between the coins and the background. It seems that the filter blurs regions of similar colors and thereby keeping the structure of the image intact even with a high σ. Specifically when comparing Figure 25a and Figure 25c we see when increasing σ the coins and the background gets more blurred, but we still have a clear separation between coins and background. Looking at and we see a much higher blurring effect and that the coins and background is starting to get mixed together a little. There is however still a distinct difference between these images and Figure 23b where we almost use the same parameters for σ and kernel size.