- Assignment 2 -

Signal and Image Processing

Philip Rajani Lassen & Maria Luise da Costa Zemsch

February 17, 2019

Histograms and Filtering

---

1 Pixel-wise contrast enhancement

1.1 Gamma transform on a gray scale image

The gamma transform is a pixel-wise operation for intensity transformations on images. Its general function is:

$$J(x, y) = c[I(x, y)]^{\gamma} \tag{1}$$

$I(x, y)$ is the input and $J(x, y)$ the output pixel, c a constant (here set to $c = 1$). Figure 1 shows the result for several different gamma Values of the following function.

Listing 1: Gamma Transform

```
def gamma_transform(img, gamma):
    img_trans = np.power(img, gamma)
    img_trans = np.round((img_trans/np.max(img_trans))*255)
    return(img_trans)
```

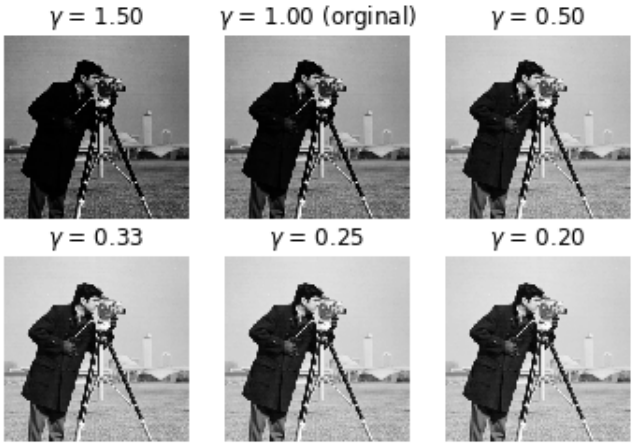

Figure 1: Gamma Correction of a gray scale image with different values of γ

The resulting image is darker for $\gamma > 1$, while $\gamma < 1$ results in a lighter image. The details in dark areas become more visible for $\gamma < 1$ and less visible for $\gamma > 1$, for details in light areas it is the other way around.

1.2 Gamma correction on a RGB image

To apply the gamma correction on a RGB image, we looked at each of the RGB components separately and used the same function as before (*gamma_transform()*).

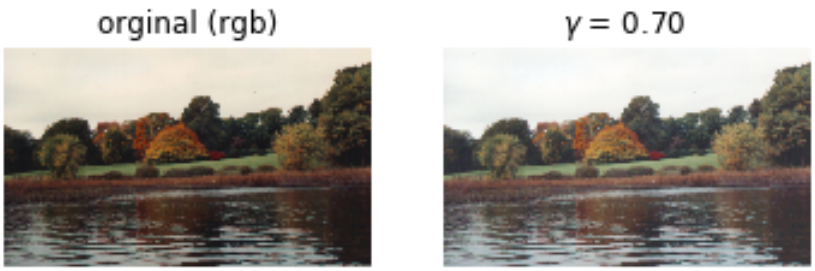

Figure 2: Gamma correction with $\gamma = 0.7$ on a RGB-image. Each colour channel was transformed separately.

Figure 2 shows the gamma correction applied to an RGB image with $\gamma = 0.7$. We can see that color is slightly faded after performing the gamma correction.

1.3 Gamma correction on a RGB-image by transforming it to HSV

We converted the image to HSV and applied *gamma_transform()* only on the v-value channel. The result was converted to RGB again.

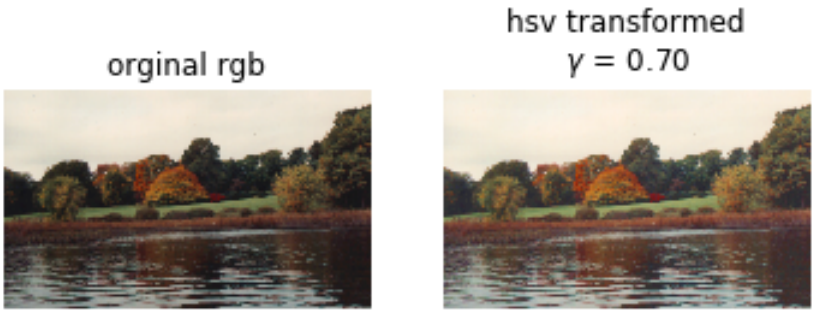

Figure 3: Gamma Correction with $\gamma = 0.7$ for a RGB-image. The image was converted to HSV before transforming only the v-value channel.

The gamma correction with the transformation to HSV doesn't change the saturation, that's why the image stays more colourful. Which resulting image is better, depends on what you want to achieve.
RGB: In this case it represents more the reality
HSV: Keeps more information about the colours and saturation

2 Histogram-based processing

2.1 Definition CDF

The cumulative distribution function defined with respect to the probability density function (p) is

$$f(x) = \int_{-\infty}^{x} p(t)dt$$

The variations in the Cumulative Distribution function correspond to changes in the probability density function. This is because the probability density function is the derivative of the cumulative distribution function (When the function is differential), and the derivative of a function captures its variation. Thus we can use this information to infer the distribution of the pixel intensities.

2.2 PDF and CDF of a constant image

The PDF and CDF for an image of constant intensity is

$$pdf(x) = \begin{cases} 1 & \text{if } x = \alpha \\ 0 & \text{if } x \neq \alpha \end{cases}$$

$$cdf(x) = \begin{cases} 0 & \text{if } x < \alpha \\ 1 & \text{if } x \geq \alpha \end{cases}$$

In the discrete case the PDF for $X \sim U(a, b)$ is given by

$$pdf(x) = \begin{cases} \frac{1}{b-a+1} & \text{if } x \in \{a, a+1, \cdots, b\} \\ 0 & \text{if } x \notin \{a, a+1, \cdots, b\} \end{cases}$$

Then we get the CDF is

$$\begin{aligned} cdf(x) &= \sum_{i=a}^{\lfloor x \rfloor} pdf(x) \\ &= \begin{cases} 0 & \text{if } x < a \\ \frac{\lfloor x \rfloor - a + 1}{b-a+1} & \text{if } x \in [a, b] \\ 1 & \text{if } x > b \end{cases} \end{aligned}$$

2.3 Cumulative histogram

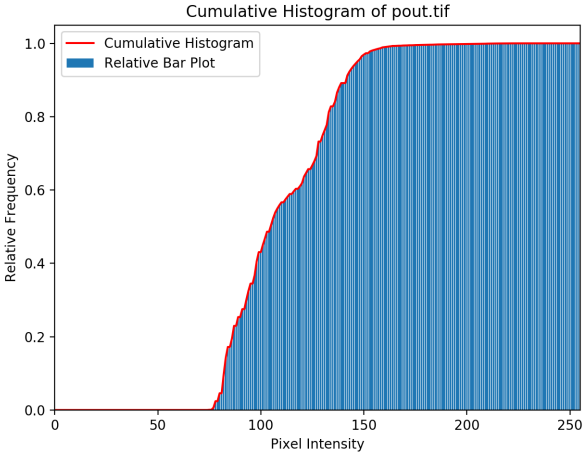

Figure 4: Relative Cumulative histogram plot for pout.tif with the relative frequencies overlayed.

In the plot the regions of fast increase correspond to pixel intensities which appear frequently in the image, where as the flat regions correspond to pixel intensities which don't have high frequency in the image or appear in the image at all.

Listing 2: Relative Cumulative Histogram Function

```
def cum_hist(histogram):
  cum_his = np.cumsum(histogram)
  total = sum(histogram)
  normalized = cum_his / total
  return np.round(normalized, 8)
```

The function takes a histogram of length 256 with integer values as input and returns the cumulative histogram represented as an array from 0 to 255 with elements normalized from 0 to 1.

2.4 CDF Application

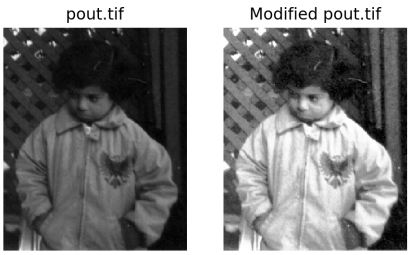

Figure 5: Comparison of the unmodified image and the image with histogram matching applied.

From the figure we can see that the image with histogram equalization has a greater range of values, which for the case of pout.tif makes the details of the image clearer.

Listing 3: Histogram Equalization Function

```
def C(I, CDF):
  CI = [[CDF[val] for val in row] for row in I]
  return CI / max(CDF)
```

The function takes a grey scale image and a cdf and returns their function composition. The code assumes the CDF provided as an argument is an array of length 256.

2.5 CDF Inverse

The CDF is not always invertible as the CDF is not neccesarily strictly increassing in which case it is possible that $f(x) = f(x+1)$, therefor making the CDF not injective, and thus not invertible.

Listing 4: Pseduo Inverse Function

```
def cdf_inverse(l, cdf):
  return min([s for s in range(256) if cdf[s] >= l])
```

The function assumes the cdf is represented as an array of 256 elements with values between 0 and 1. The function also assumes $l \in [0, 1]$.

2.6 Histogram Matching

Listing 5: Pseduo Inverse Function

```
def histogram_matching(im1, c1, c2):
  temp = C(im1, c1)
  result = [[cdf_inverse(val, c2) for val in row] for row in temp]
  return result
```

The Histogram matching function Takes an image, its CDF, and the target images CDF, and creates a new image using the histogram matching procedure described.

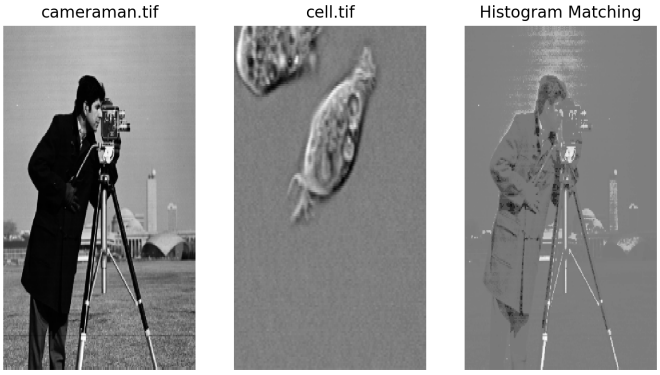

Figure 6: Comparison of original image, target image, and result of histogram matching

We can see that the histogram matching on the cameraman.tif with the target image of cell.tif changes the cameraman so that the cameraman.tif has the same distribution of pixel intensities. In this case it causes the majority of the image to be close to the grey pixels found in the background of the cell image.

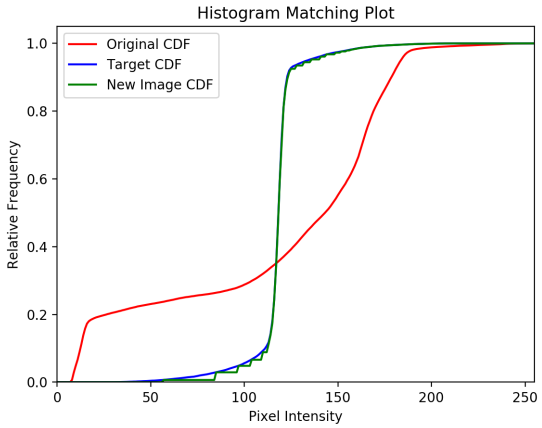

Figure 7: Plot of the Relative CDFs of the original image, target image, and resultant image from histogram matching

We can see that the cell image and the generated image from the histogram matching have approximately the same CDF. While the original cameraman image has a substantially different CDF. This helps explain why the new image has such similarities with the cell.tif in terms of pixel intensity distribution.

2.7 Applied Histogram Matching

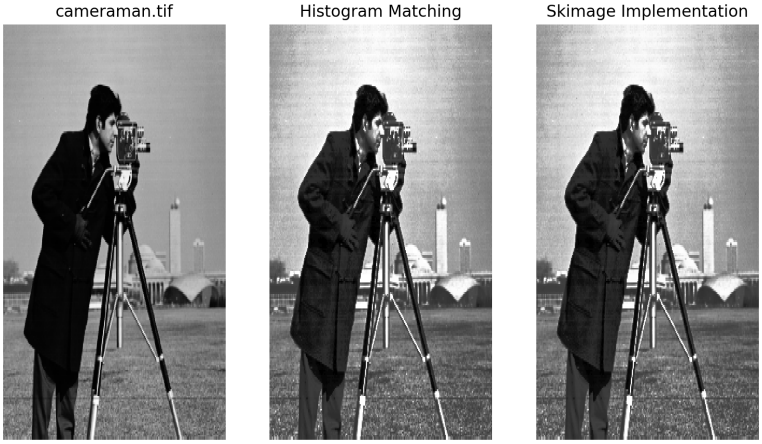

Figure 8: Image comparison of cameraman.tif, cameraman.tif with histogram matching of uniform distribution of range 0 to 85, and with skimage equalize_hist function of range 0 to 85

We can see that using the histogram matching algorithm on the cameraman with a a target CDF coming from a uniform distribution with pixel intensities from 0 to 85, that we shrink the range of pixel intensities which causes the image to lose detail. This can be seen in the

background where the the changing of color isn't very fluid. The histogram matching the we implemented and the skimage implementation of equalize_hist seem to render to the same image when applied to the cameraman.

2.8 Midway Specification

For constant image $I_1(x,y) = a, I_2(x,y) = b$. We observe that $C_1(I_1(x,y)) = 1, C_2(I_2(x,y) = 1$ and $f_1^{(-1)}(1) = a, f_2^{(-1)}(1) = b$, which follows from our definition of the pseudo-inverse described in 2.7. Then we get

$$\begin{aligned}\tilde{I_1} &= \frac{1}{2}(C_1^{(-1)} + C_2^{(-1)})(C_1(I_1)) \\ &= \frac{1}{2}(C_1^{(-1)}(C_1(I_1)) + C_2^{(-1)}(C_1(I_1)))\end{aligned}$$

Then for every pixel $I(x,y)$

$$\begin{aligned}\tilde{I_1}(x,y) &= \frac{1}{2}(C_1^{(-1)}(C_1(I_1(x,y))) + C_2^{(-1)}(C_1(I_1(x,y)))) \\ &= \frac{1}{2}(C_1^{(-1)}(1) + C_2^{(-1)}(1)) \\ &= \frac{1}{2}(a+b)\end{aligned}$$

Thus $\tilde{I_1} = \frac{(a+b)}{2}$. Following the exact same step but replacing $(C_1(I_1))$ with $(C_2(I_2))$ we also get $\tilde{I_2} = \frac{(a+b)}{2}$

The cumulative histogram of the midway specification is not equal to the average of the cumulative histograms. Consider cumulative histogram for constant images with values a and b where $a < b$. The average of their normalized cumulative histogram is

$$f(x) = \begin{cases} 0 & \text{if } x < a \\ 1/2 & \text{if } a \leq x < b \\ 1 & \text{if } x \geq b \end{cases}$$

Which does not equal the normalized cumulative histogram of the constant image $\frac{(a+b)}{2}$.

2.9 Midway Specification Implementation

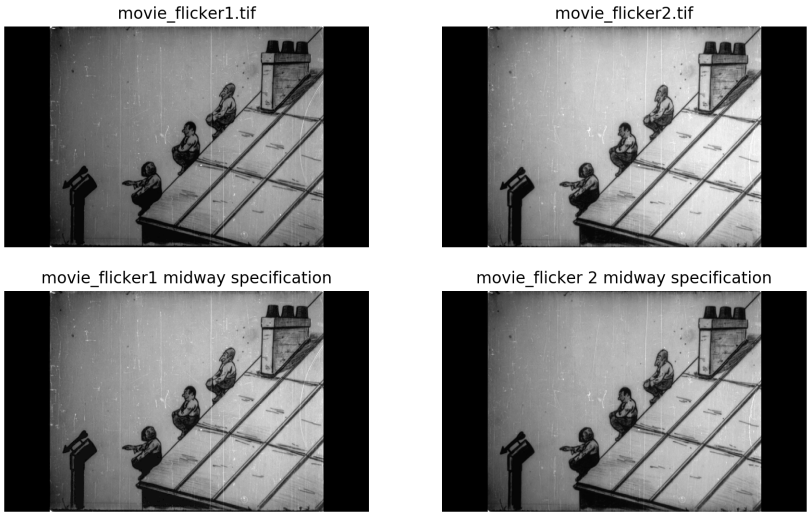

Figure 9: Comparison of movie_flicker1.tif, movie_flicker2.tif and both filtered with the Midway Specification.

In Figure 9 we can see that the the original movie_flicker1.tif is darker than movie_flicker2.tif. When we compare the two images below which have been filtered with the midway specification we see that their backgrounds appear to be the same shade.

Generalizing to n images we would just need to adjust ϕ to the following

$$\phi = \frac{1}{n}(C_1^{(-1)} + C_2^{(-1)} + \cdots + C_n^{(-1)})$$

We can think of this new ϕ as simply being a midway between all n images.

3 Image filtering and enhancement

3.1 Approximating image derivatives: Convolution and Correlation

Filtering an image using correlation, the kernel implementing the approximate derivative looks like the following.

Derivative to x-axis	Derivative to y-axis
$\begin{bmatrix} -1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ -1 \end{bmatrix}$

If we compare the convolution and the correlation integral (see equation 2 and 3), the main difference is whether we flip h or not.

$$g(x,y) = \{f * g\}(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x',y')h(x-x',y-y')dx'dy' \tag{2}$$

$$g(x,y) = \{f \circ g\}(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x',y')h(x+x',y+y')dx'dy' \tag{3}$$

To implement these in Python we only have to flip the filter kernel, which is the same as multiplying the kernel with -1. So we get the following convolution kernel:

Derivative to x-axis	Derivative to y-axis
$\begin{bmatrix} 1 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 1 \end{bmatrix}$

For symmetrical filters like the mean or the Gaussian filter it does not make a difference if the kernel is flipped.

3.2 The Prewitt and Sobel filters

The Prewitt and Sobel filters can be seen as the first derivative between the neighbour pixel values. If the difference between them is large you get a higher output value. Furthermore it also computes the diagonal gradient, which is included to the output value calculation. Thus we get filters, which are more robust to noise than the simple finite difference approximation. The Sobel filter is even more robust to noise, because the pixels in the middle of the kernel are weighted higher than the outer pixel.

3.3 Mean and median filtering

The mean/median filter takes the mean/median of the pixel in the kernel to compute the output pixel value. To take care of the border pixel values, we pad the image with zeros. The kernel is defined out of the padded image. We only consider odd kernel sizes, as we want to

compute the output value out of symmetric distributed neighbour values (means the center pixel needs to be exactly in the center).

Listing 6: Median Filter

```
def median_filter(img, N):
  size = img.shape
  #pad image
  border = int((N-1)/2)
  img_filter = np.zeros(size)
  img_pad = np.pad(img, pad_width = border, mode='constant', constant_values=0)
  #running the kernel over the image
  for x in range(size[0]):
    for y in range(size[1]):
      kernel = img_pad[x:x+N, y:y+N]
      img_filter[x, y] = np.median(kernel)
  return(img_filter)
```

The only difference between the method *mean_filter* and the *median_filter* is in line 10: the computation of the output pixel.

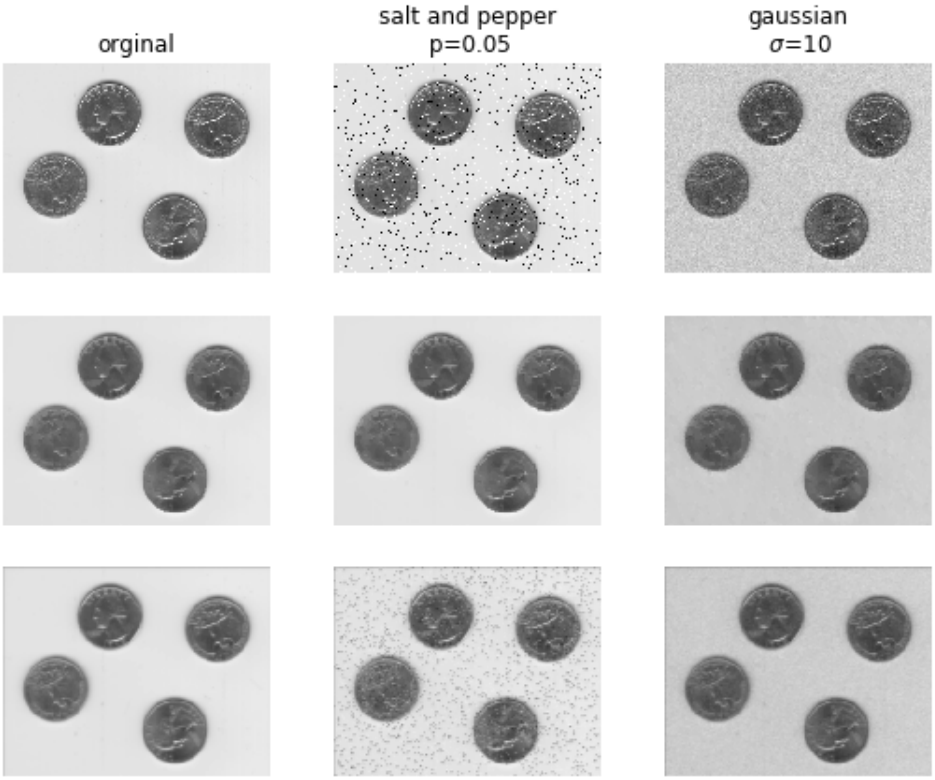

Figure 10: Comparison of a median filter (second row) and a mean filter (third row) both with a kernel size of $N = 3$. Both filters were applied on the original image and two noisy images (salt and pepper and Gaussian noise).

We applied these two filters on a noisy image for salt and pepper and Gaussian noise (see figure 10). If you compare both filters with the same kernel size, the median filter gets a significantly better output. While the mean filter fades the salt and pepper noise, the median filter reduces almost all of it (even with a kernel size of N = 3). However, both filters cannot remove all the Gaussian noise.

Increasing the kernel size you blurs the image. An example you can see in figure 11, where different kernel sizes of the median filter were applied on an image with salt and pepper noise.

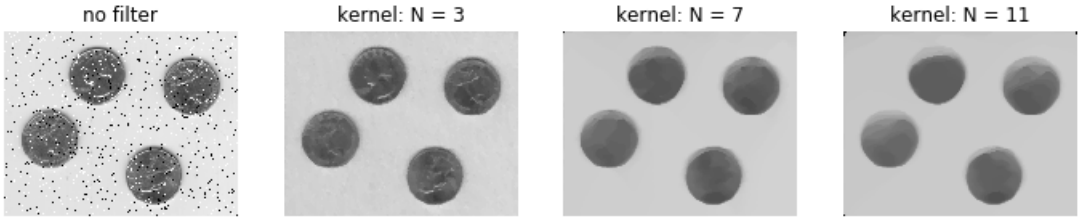

Figure 11: Comparison of different kernel sizes of the median filter. The larger the kernel gets the more blury the image is (the less details you can see).

We want to compare the increase of the computational time regarding the kernel size of both filters. Therefore different computational times obtained for N=3 to N=25 are stored, each time for 100 executions. The mean time is shown in figure 12.

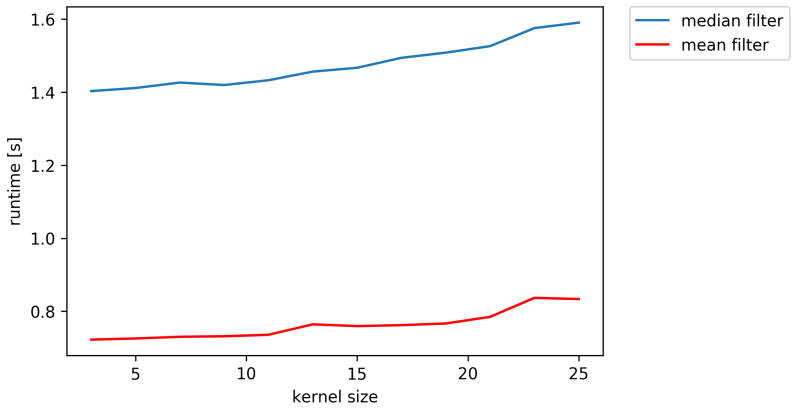

Figure 12: Comparison of the runtime for different kernel sizes for the *median_filter* and the *mean_filter*. Both filters have an increase in runtime with respect to the kernel size, while the *mean_filter* is almost double as fast as the *median_filter*.

With a kernel size of $N = 1$ the input image is the same as the output image, thus the mean/median of one value is that value. Therefore we only computed the filtered image with odd kernel sizes from $N = 3$ to $N = 25$.
The first significant thing you notice, is that the median filter is much faster than the mean filter, almost double as fast as the other one. However, the increase in runtime of both filters is around 0.2s for one execution.

3.4 Gaussian filter: Kernel size N

We implemented a Gaussian filter, which can be applied on a gray scale image. The sum of the values in the Gaussian kernel is set to 1 to ensure the output pixel value to be in the

correct interval ([0, 255]). The input image was padded with zeros depending on the kernel size N. To improve the runtime, the kernel was seperated linearly.

Listing 7: Gaussian Filter

```
def gaussian_filter(img, N, sigma):
  size = img.shape
  img_filter = np.zeros(size)
  #pad image
  border = int((N-1)/2)
  img_pad = np.pad(img, pad_width = border, mode='constant', constant_values=0)
  #computing the gaussian kernel
  kernel = np.arange(-N//2 +1, N//2 + 1)
  kernel = np.exp(-(kernel**2) / (2 * sigma**2))
  summe = sum(kernel)
  kernel = kernel/summe
  #linear fitting along the x-axis
  for x in range(size[0]):
    for y in range(size[1]):
      kernel_img = img_pad[x:x+N, y + N//2]
      img_filter[x, y] = sum(kernel*kernel_img)
  #linear fitting along the y-axis
  for y in range(size[1]):
    for x in range(size[0]):
      kernel_img = img_pad[x + N//2, y:y+N]
      img_filter[x, y] = sum(kernel*kernel_img)
  return(img_filter)
```

While increasing the kernel size N the image gets blurrier and we get a worse edge accuracy:

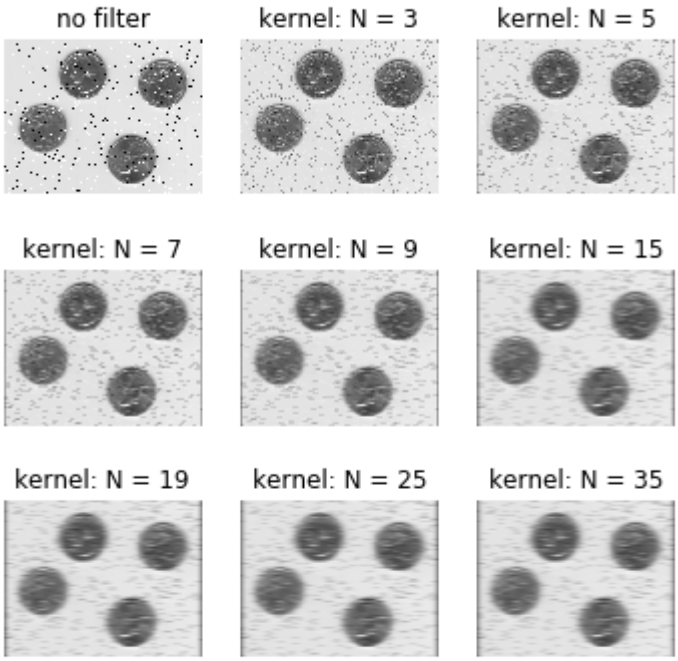

Figure 13: Comparison of different kernel sizes of a Gaussian filter with a standard deviation of $\sigma = 5$.

As of a $\sigma = 3N$, the image does not get significantly blurrier. This is because the σ is already very large meaning the border pixels wouldn't significantly contribute to the output pixel. This follows from the fact the Gaussian distribution goes to 0 very quickly as we get farther from the center.

3.5 Gaussian filter: Standard derivation σ

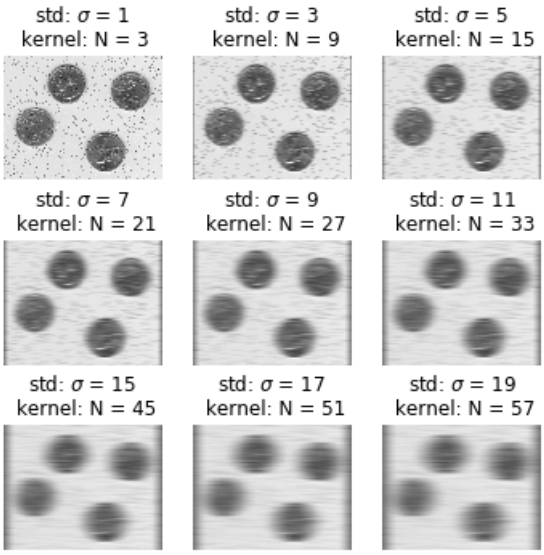

Figure 14: Comparison of different standard derivations σ of the Gaussian filter. The kernel size N was set to $N = 5\sigma$. The original image is transformed with salt and pepper noise.

The bigger the kernel, the larger the black edge at the y-axis. This is because we use zero padding on the image for kernel input.
The denoising gets better with larger kernels. Still, with a standard deviation of $\sigma = 9$ you can still see the background noise, even the image is already so blurry that you cannot identify much.

4 Bonus questions: bilateral filtering

4.1 Theory

The bilateral filter is non-linear. This is because the weight from the filter in the one dimensional domain is dependent on the pixel difference. This depends on the pixels around the image, making it shift variant.

We can think of the τ parameter as a way to specify how much weight we would like to put on the pixel difference. As τ increases the affect of $g_\tau(u)$ on the bilateral filter decreases.

In the limit case where $\tau \to \infty$ we get that $lim_{\tau\to\infty} \exp(-\frac{u^2}{2\tau^2}) = 1$ for all u. In which case

the bilateral filter becomes

$$\tilde{I}(x,y) = \frac{\sum_{i=-l/2}^{l/2}\sum_{j=-k/2}^{k/2} f_\sigma(x,y)I(x+i,y+j)}{\sum_{i=-l/2}^{l/2}\sum_{j=-k/2}^{k/2} f_\sigma(x,y)}$$

Where $\frac{1}{\sum_{i=-l/2}^{l/2}\sum_{j=-k/2}^{k/2} f_\sigma(x,y)}$ is simply a constant. Letting this constant equal C we get

$$\tilde{I}(x,y) = C\sum_{i=-l/2}^{l/2}\sum_{j=-k/2}^{k/2} f_\sigma(x,y)I(x+i,y+j)$$

4.2 Implementation

Each image pixel is computed by the following function:

$$\tilde{I}(x,y) = \frac{\sum_{i=-l/2}^{l/2}\sum_{j=-k/2}^{k/2} \omega(x,y,i,j)I(x+i,y+j)}{\sum_{i=-l/2}^{l/2}\sum_{j=-k/2}^{k/2} \omega(x,y,i,j)} \tag{4}$$

with $\omega(x,y,i,j) = f_\sigma(i,j)\cdot g_\tau(I((x+i,y+j)-I(x,y))$, k,l being the dimensions of the filter kernel. $f_\sigma(i,j)$ represents a 2D Gaussian function and g_τ a 1D Gaussian function.
We need to restrict the borders, as we include the neighbourhood pixel in the calculation. Therefore we do not filter the $N/2$ pixel on the corner rows/columns.

Listing 8: Bilateral Filter

```
def bilateral_filter(img, N, sigma, tau):
  size = img.shape
  img_filter = np.zeros((size[0], size[1]))
  z=0; n=0
  #running the kernel over the image
  for x in range(N//2, size[0]-N//2):
    for y in range(N//2, size[1]-N//2):
      #running pixel-wise over the kernel
      for j in range(-(N//2), int(np.ceil(N//2))):
        for i in range(-(N//2), int(np.ceil(N//2))):
          z += w(img, x, y, i, j, sigma, tau)*img[x+i, y+i]
          n += w(img, x, y, i, j, sigma, tau)
      img_filter[x, y] = z/n
      z=0; n=0
  return(img_filter)
```

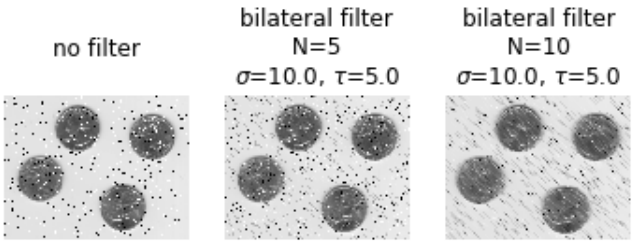

Figure 15: Bilateral filter on an image with salt and pepper noise. The kernel size needs to be increased to see an improvement of denoising.

In the figure we used a bilateral filter with parameters $\sigma = 10, \tau = 5$ and we used kernels of size $N = 5, N = 10$. The bilateral filter gives slight improvements on the unfiltered image. We can see that the salt and pepper noise gets slightly dampened. With a kernel size of 10 we see that we have a greater blurring effect.

4.3 Comparison of different std parameters with the Gaussian filter

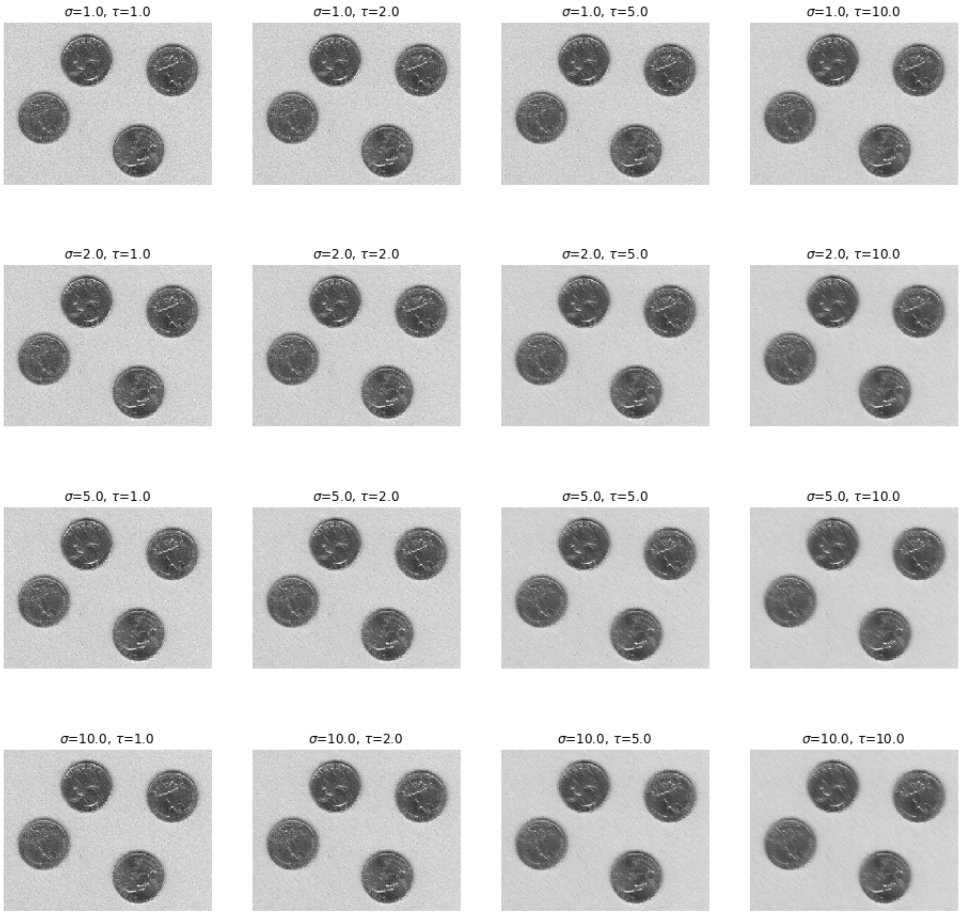

Figure 16: Variation of the Gaussian parameters σ and τ of the bilateral filter with kernel size 7. The image has a Gaussian noise of $\sigma = 10$.

With increasing σ and τ the noise diminishes. In all the above images the kernel size used on the filter was 7.

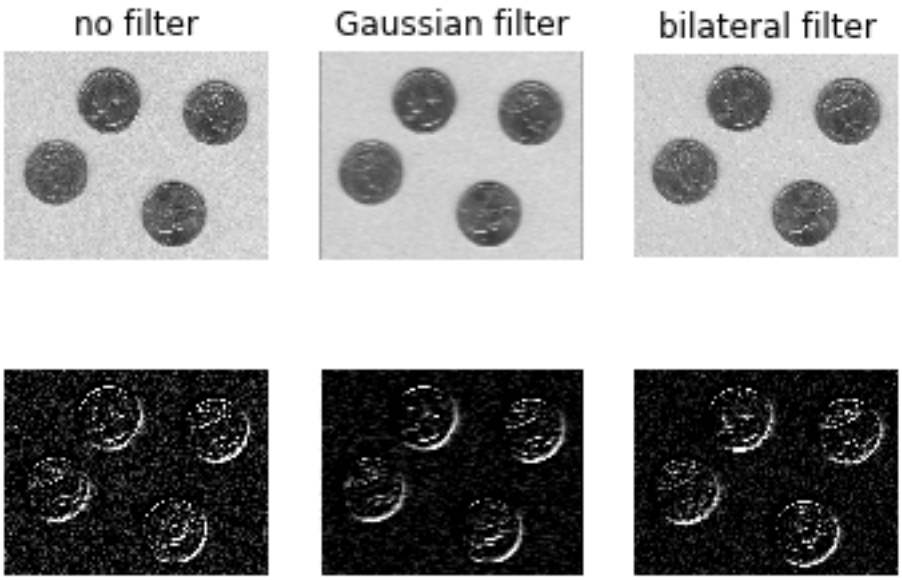

Figure 17: Edge detection on an image with Gaussian noise ($\sigma = 10$. The detection was used on the original noisy image, on a Gaussian filtered one and on an image with an bilateral filter.

The edge detection of an image with Gaussian noise gets better if we use a bilateral filter before applying the edge method. The Gaussian filter smooths the edges too much, so they fade into the background.