DATA.ML.300 Computer Vision Exercise 1

Miska Romppainen
H274426

January 2021

1 Task: Homogeneous coordinates

Here is my solution to task 1. homogeneous coordinates. As stated in the exercise PDF "*You are free to use MATLAB for calculations.*" The following calculations were done using MATLAB:

Task 1. Homogeneous coordinates.
Converting the Cartesian points to Homogeneous coordinates as follows:

$$x1 = (2, -1) \rightarrow (2, -1, 1)$$
$$x2 = (1, -2) \rightarrow (1, -2, 1)$$
$$x3 = (1, 1) \rightarrow (1, 1, 1)$$
$$x4 = (-1, 0) \rightarrow (-1, 0, 1)$$

Now the matrices of the coordinates can be presented as following

$$x1 = [2; -1; 1]$$
$$x2 = [1; -2; 1]$$
$$x3 = [1; 1; 1]$$
$$x4 = [-1; 0; 1]$$

We can create a line between two coordinate points by taking cross product of the homogeneous coordinates.

$$Line(ij) = point(i) \times point(j)$$

We get $l1$ by taking cross product of points $x1$ and $x2$

$$l1 = x1 \times x2$$

We get

$$l1 = [1, -1, -3]$$

Then by repeating the same for $x3$ and $x4$ we get $l2$

$$l2 = x3 \times x4$$

We get

$$l2 = [1, -2, 1]$$

We can get the intersection point of lines k and l by taking the cross product lines k and l

$$point(kl) = line(k) \times line(l)$$

By taking the cross product of $l1$ and $l2$

$$x = l1 \times l2$$

We get the intersection point at

$$x = [-7, -4, -1]$$

We can convert these Homogeneous coordinates back to Cartesian points by dividing the elements u and v with w, w can be seen as the "weight" or "scale-factor" of the coordinates

$$p = [u, v, w] \rightarrow (u/w, v/w)$$

Therefore the intersection point x is

$$x = [-7/-1, -4/-1]$$

$$x = (7, 4)$$

2 Task: Image denoising

Here is my solution to task 2. image denoising. The goal of the task was to implement three different filters to a image in order to denoise it. These three filters were:
a) Gaussian filtering
b) Median filtering
c) Bilteral filtering

I used Python as my choice of programming language. The figure 1 portrays the original picture of *Einstein* without any image denoising.

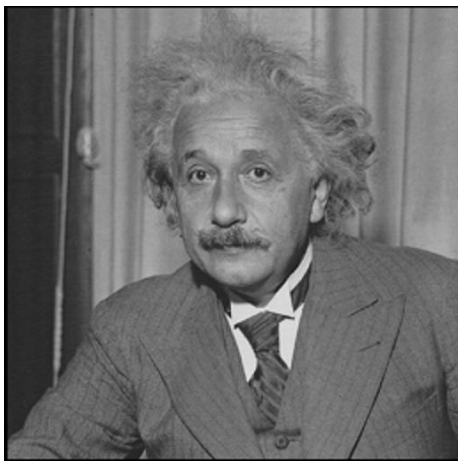

Figure 1: Original picture of Einstein

Two denoising techniques were used in the code, *"salt and pepper" noise* and *zero-mean Gaussian noise*. The output of these noises can be seen in the figure 2.

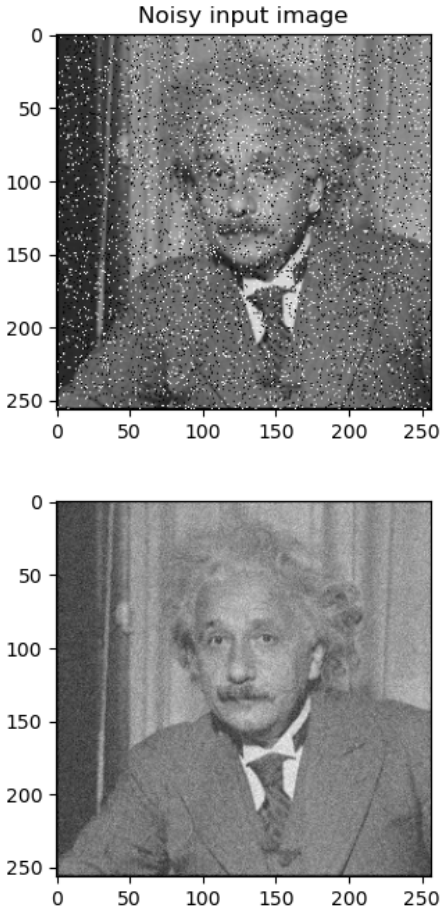

Figure 2: Noised images; *"salt and pepper" noise* (top), *zero-mean Gaussian noise* (bottom)

a) Gaussian filtering:
CODE SNIPPET FROM: *image_denoising.py*

```
# Decompose the filter into its horizontal and vertical components
U, S, V = np.linalg.svd(g)
h_filter = U[:, 0]
v_filter = V[0, :]

# Normalize the filters
h_filter /= np.sum(h_filter)
v_filter /= np.sum(v_filter)
```

```
# Apply the filters to the "salt and pepper"
# noised image one axis at the time
image_out_imns = conv1(imns, h_filter, axis=1 , mode='reflect')
gflt_imns = conv1(image_out_imns, v_filter, axis=0 , mode='reflect')

# Apply the filters to the zero-mean Gaussian
# noised image one axis at the time
image_out_imng = conv1(imng, h_filter, axis=1 , mode='reflect')
gflt_imng = conv1(image_out_imng, v_filter, axis=0 , mode='reflect')
```

Gaussian implementation done by instead of directly filtering with Gaussian filter, it is created with a separable implementation, where horizontal and vertical 1D convolutions are used. (Results are stored to $gflt\_imns$ and $gflt\_imng$)

The output of the Gaussian implementation can be seen in the figure 3

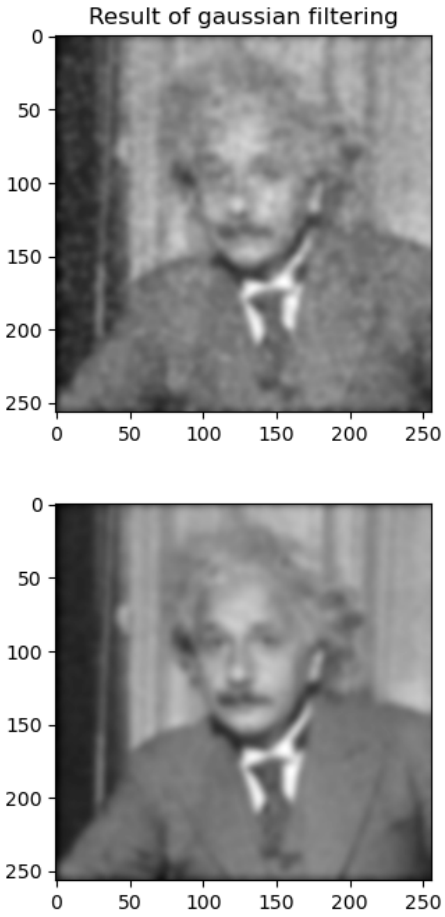

Figure 3: Gaussian filtered images

b) Median filtering:
CODE SNIPPET FROM: *image_denoising.py*

```
# Median filtering is done by extracting
# a local patch from the input image
# and calculating its median
def median_filter(img, wsize):
    nrows, ncols = img.shape
    output = np.zeros([nrows, ncols])
    k = (wsize - 1) / 2

    for i in range(nrows):
        for j in range(ncols):
```

```
                # Calculate local region limits
                iMin = int(max(i - k, 0))
                iMax = int(min(i + k, nrows - 1))
                jMin = int(max(j - k, 0))
                jMax = int(min(j + k, ncols - 1))

                # Use the region limits to extract
                # a patch from the image,
                # calculate the median value
                # (e.g using numpy) from the extracted
                # local region and store it to
                # output using correct indexing.

                ##--your-code-starts-here--##

                # Extracting a patch from the image
                patch = img[iMin:iMax+1, jMin:jMax+1]

                # Calculating the median value of the patch
                median =  np.median(patch)

                # Storing the median value in the output image
                output[i, j] = median

                ##--your-code-ends-here--##

    return output

# Apply median filtering, use neighborhood size 5x5
# Store the results in medflt_imns and medflt_imng
# Use the median_filter function above

##--your-code-starts-here--##

medflt_imns = median_filter(gflt_imns, 5)
medflt_imng = median_filter(gflt_imng, 5)

##--your-code-ends-here--##
```

Median filtering is done by extracting a local patch from the input image and calculating its median. Local region limits are used to *extract* a path from from the image and to calculate the extracted patches median value and storing it to the *output* image

The output of the Median filter implementation can be seen in the figure 4

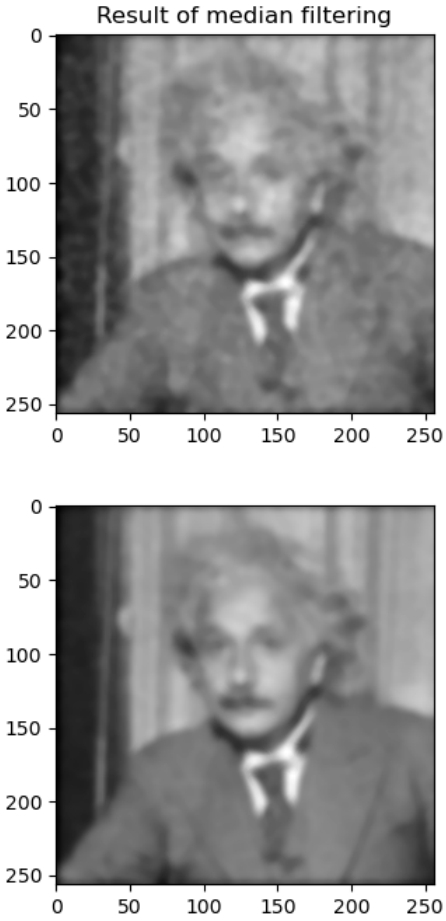

Figure 4: Median filtered images

c) Bilateral filtering:
CODE SNIPPET FROM: *image_denoising.py*

```
wsize = 11
sigma_d = 2.5
sigma_r = 0.1

bflt_imns = bilateral_filter(imns, wsize, sigma_d, sigma_r)
bflt_imng = bilateral_filter(imng, wsize, sigma_d, sigma_r)
```

Bilateral filter is a "edge-preserving" filter similar to Median filter - which performs the best in denoising the *zero-mean Gaussian noise*, but when trying to denoise the *salt and pepper" noise*, the output is worse compared to the original

image.

The output of the Bilateral filter can be seen in the figure 5

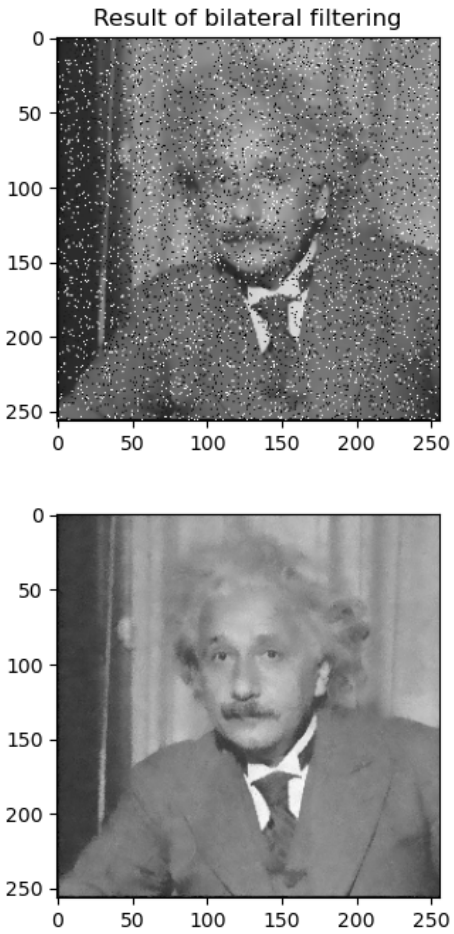

Figure 5: Bilateral filtered images

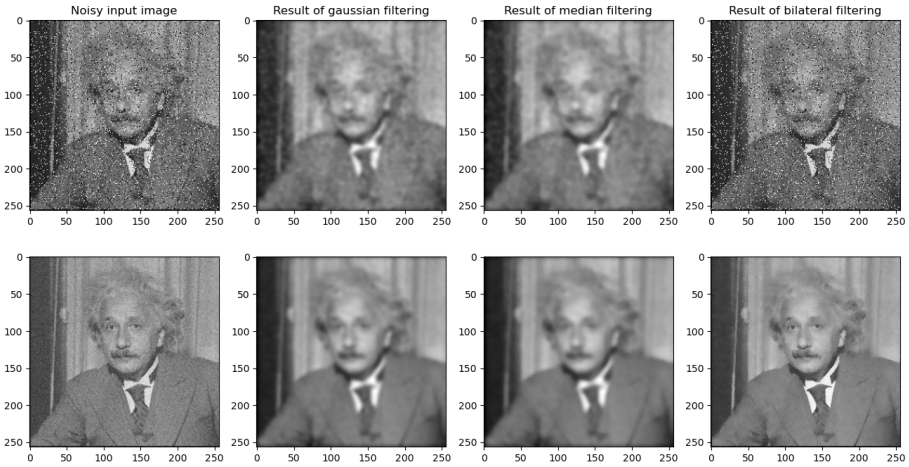

Figure 6: Output of the code for task 1

3 Task: Hybrid images

Here is my solution to task 3. hybrid images. The goal of the task was to create a hybrid image combining image of a wolf and a man. Original input images can be seen in the figure 7 and the output images can be seen in the figure 8.

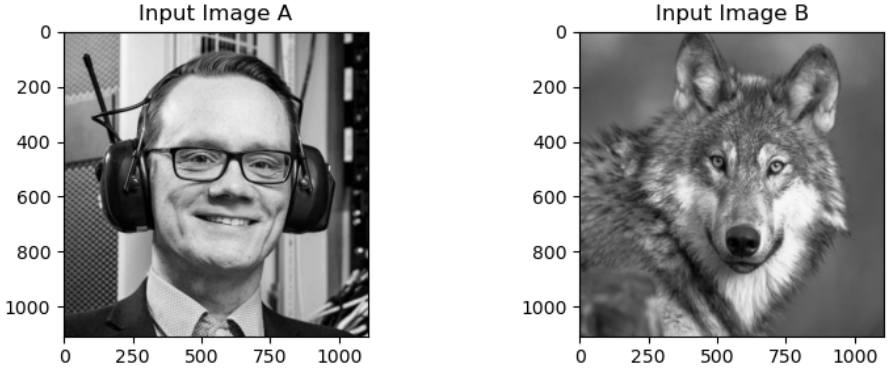

Figure 7: Input images for task 3; man (left, input image A) and wolf (right, input image B)

In order to create hybrid image, first the images need to be transformed in away that the eyes overlap correctly. This is done by using an affine transformation which is a geometric transformation which preservers lines and parallelism but not necessarily Euclidean distances and angles [**1**]. Affine transformation is used to fit the wolf and the mans eyes and transforming the images corre-

spondingly. A using simple blending method we can create aligned images using additive superimposition, output image can be seen in the 8.
In order to create hybrid image, we need to get low-pass and high-pass filtering for the two images. Low-pass filters have been created using *scipy.ndimage* packages $gaussian_filter$-packet. But to implement high-pass filter we need to remove low-pass filtered result from the original image. After which the low-pass filtered image of the man and high-pass filtered image of the wolf could be combined. The result of the hybrid image can be seen in figure 8 or in the figure 9. The code snippet of the implementation can be seen below.

CODE SNIPPET FROM: *hybrid_images.py*
$Sidenote : sigmaA = 16, sigmaB = 8.$

```
# Replace the zero image below with a high-pass filtered
# version of 'wolft'
##---your-code-starts-here---##
wolft_highpass = wolft - wolft_lowpass
##---your-code-ends-here---##

# Replace also the zero image below with the correct
# hybrid image using your filtered results
##---your-code-starts-here---##
hybrid_image = man_lowpass + wolft_highpass
##---your-code-ends-here---##
```

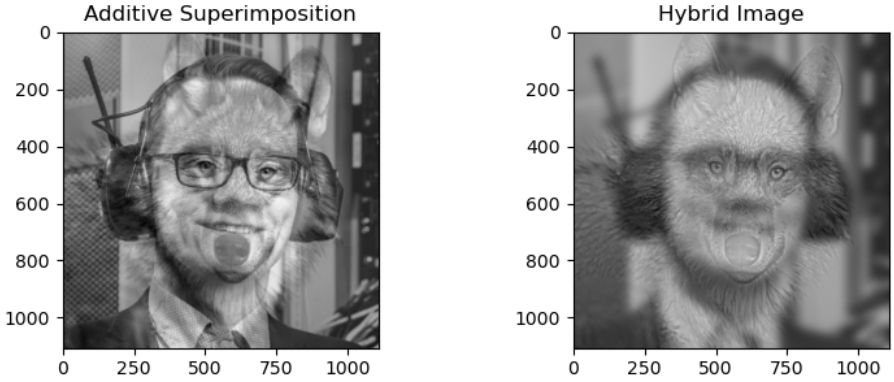

Figure 8: Output images for task 3; Additive superimposition (left) and hybrid image (right)

The visualized log magnitudes of the Fourier transforms of the original images can be seen in the figure 10 and figure 11

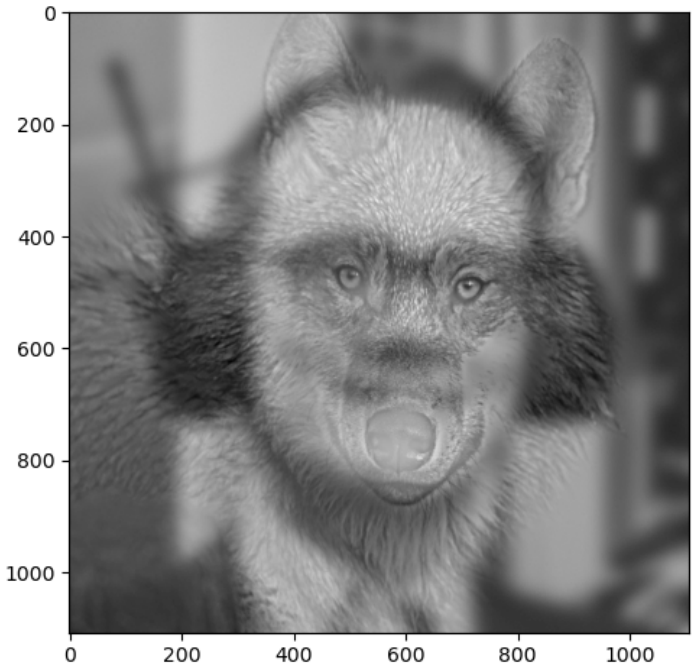

Figure 9: Output hybrid image with sigma values sigmaA = 16 and sigmaB = 8

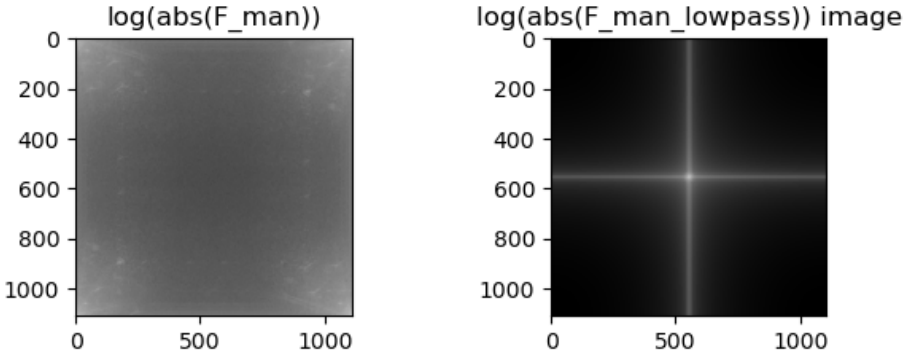

Figure 10: Calculated 2D Fourier transforms for the original and the filtered images of the man

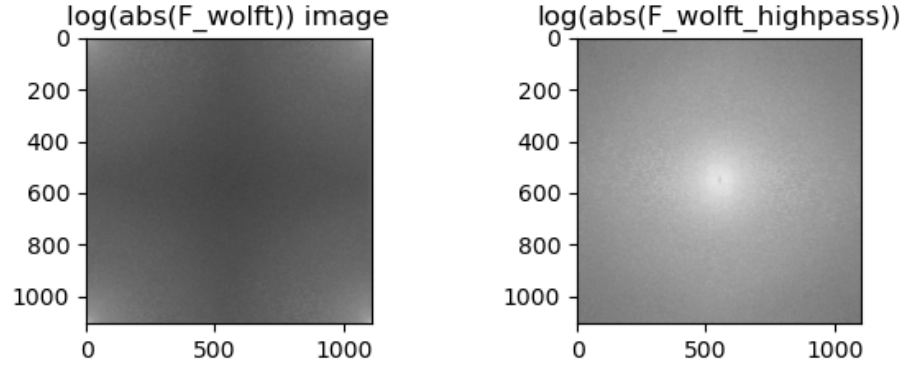

Figure 11: Calculated 2D Fourier transforms for the original and the filtered images of the wolf

To create the magnitude logs, first we need to apply the Fourier transforms of the original images. This is done using the $fft2$ function from $numpy.fft$. In order to create magnitude logs for the filtered images, first we apply the Fourier transform $fft2$ after which we need to shift the zero-frequency component to the center of the spectrum, which is done using the $fftshift$ function from the $numpy.fft$ package. The logs are created using *np.logs* to the Fourier transformed images absolute values element-wise using function *np.abs*.

CODE SNIPPET FROM: *hybrid_images.py*

```
# Calculate the 2D Fourier transforms of the
# original images
F_man = fft2(man)
F_wolft = fft2(wolft)

# Calculate the 2D Fourier transforms of the
# filtered images using fft2
# Shift the zero-frequency component to the
# center of the spectrum using fftshift
F_man_lowpass = fftshift(fft2(man_lowpass))
F_wolft_highpass = fftshift(fft2(wolft_highpass))
...
plt.imshow(np.log(np.abs(F_man)), cmap='gray')
plt.title("log(abs(F_man))")
plt.subplot(2,2,2)
plt.imshow(np.log(np.abs(F_man_lowpass)), cmap='gray')
plt.title("log(abs(F_man_lowpass)) image")
plt.subplot(2,2,3)
plt.imshow(np.log(np.abs(F_wolft)), cmap='gray')
plt.title("log(abs(F_wolft)) image")
plt.subplot(2,2,4)
plt.imshow(np.log(np.abs(F_wolft_highpass)), cmap='gray')
```

References

[1] Wikipedia, Affine transfomration (2022), https://en.wikipedia.org/wiki/Affine_transformation, Last accessed 15 January 2023