

Image Analysis Assignment 1

Olof Harrysson

1.

By looking at the function you notice that the brightness only varies in the x-axis. You can therefore determine a value for a certain pixel and that whole y-column will have the same value. If you want to divide a line into n segments you need to make n - 1 cuts. I'm thinking of a pixel as a segment in two dimensions. If x, y ranges from 0 - 1 we can calculate the segment length in one dimension as

Segment length = $(b - a) / (n - 1)$ where $a=0$, $b=1$, $n=5$. This gives us a segment length of 0.25.

The values corresponding to the columns in the image will then have the values

$f(0,y) = 0$	$f(0.25,y) = 0.75$	$f(0.5,y) = 1$	$f(0.75,y) = 0.75$	$f(1,y) = 0$
--------------	--------------------	----------------	--------------------	--------------

We now want to quantify the image to 16 gray levels. As 0 is the lowest value found in the image it gets gray-level #0. Similarly 1 is the highest value and gets grey-level #15. The only other value found is 0.75 which gets grey-level #11 since $0.75 * (16 - 1) = 11.25$

0	11	15	11	0
0	11	15	11	0
0	11	15	11	0
0	11	15	11	0
0	11	15	11	0

2.

$$\int_0^r \frac{3}{2} \sqrt{t} \, dt = \left[t^{3/2} \right]_0^r = r^{3/2}$$

3.

$$\begin{pmatrix}
 3 & 3 & 2 & 2 & 2 & 3 & 3 & 3 & 0 & 2 & 2 & 2 \\
 0 & 3 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 3 & 2 & 3 \\
 1 & 2 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 2 & 3 \\
 3 & 0 & 3 & 1 & 0 & 3 & 3 & 0 & 2 & 3 & 3 & 2 \\
 3 & 2 & 1 & 0 & 1 & 3 & 1 & 3 & 2 & 0 & 3 & 2 \\
 0 & 3 & 3 & 1 & 1 & 1 & 1 & 0 & 2 & 3 & 3 & 3 \\
 2 & 0 & 2 & 1 & 1 & 0 & 1 & 0 & 2 & 2 & 0 & 2 \\
 2 & 1 & 3 & 0 & 0 & 3 & 2 & 1 & 2 & 0 & 3 & 2 \\
 2 & 0 & 3 & 1 & 0 & 3 & 2 & 3 & 2 & 2 & 0 & 3 \\
 2 & 2 & 2 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 2 & 1 \\
 0 & 2 & 3 & 0 & 0 & 1 & 0 & 0 & 0 & 3 & 3 & 2 \\
 0 & 2 & 3 & 0 & 3 & 3 & 2 & 1 & 2 & 1 & 0 & 2
 \end{pmatrix}$$

4.

```

function [S] = im2segment(im)
% [S] = im2segment(im)

nrofsegments = 5; % max nbr of segments
m = size(im,1); % nbr y px
n = size(im,2); % nbr x px
S = cell(1,nrofsegments);

% converts the image to a binarized image i.e. each pixel is either
% 0 or 1. Pixels who's value is above the specified threshold becomes 1
binary_img = imbinarize(im, 160);

% Inverts the image. Black becomes white
compl_img = imcomplement(binary_img);

% Segments the image based on their connectivity. Each pixel of a
% segment get's set to it's corresponding segment.
% Either the 4 or 8 neighbours can be used to determine connectivity
L = bwlabel(compl_img, 8);

% Loops over all the segments
for kk = 1:nrofsegments

    % Creates a black image the size of the original one
    segment_img = zeros(m,n);

    % Finds the pixel coordinates corresponding to a segment
    [r, c] = find(L==kk);

    % Colors all the pixels in a segment white
    for i = 1:size(r)
        segment_img(r(i), c(i)) = 1;
    end;

    % Set the final image
    S{kk} = segment_img;
end;

```

```
>> inl1_test_and_benchmark
```

```
You tested 10 images in folder ../datasets/short1
```

```
The jaccard scores for all segments in all images were
```

0.9224	0.9588	0.9783	0.9692	0.9524
0.9671	0.9811	0.9764	0.9905	1.0000
0.9895	0.9528	0.9550	0.9505	0.9895
0.9333	0.9570	0.9767	0.9545	1.0000
0.9671	0.9775	0.9420	1.0000	0.9747
0.9896	0.9307	0.9714	0.9277	0.9464
0.9625	0.9500	0.9811	0.9618	0.9884
0.9333	1.0000	0.9697	0.9818	0.9554
0.9710	1.0000	0.9562	0.9697	0.9353
0.9895	1.0000	0.9700	0.9709	0.9896

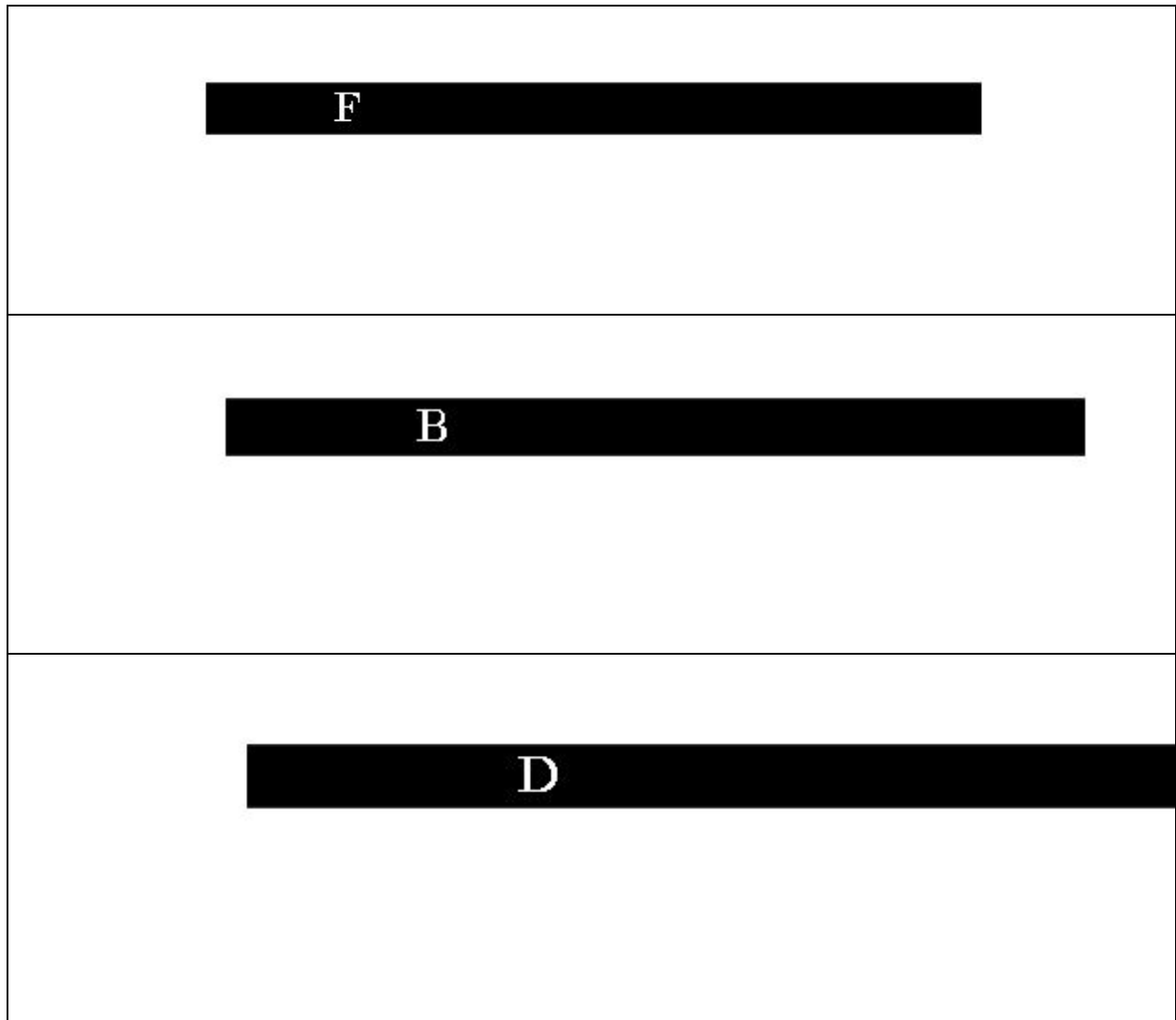
```
The mean of the jaccard scores were 0.96836
```

```
This is great!
```

Z A F B D

Z

A



5.

The amount of dimensions correspond to the number of pixels. For a 2x2 image, imagine you want to represent $\begin{bmatrix} 7 & 0 \\ 2 & 3 \end{bmatrix}$. This could be done by scaling and adding a combination of these images. $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$. 7 of the first one, 2 of the third one and 3 of the fourth one. Similarly, a 2000 x 3000 image would need 6 million dimensions.

6.

$$\begin{aligned} \|u\| &= \sqrt{u \cdot u} = \sqrt{27} \\ \|v\| &= \sqrt{v \cdot v} = 1 \\ \|w\| &= \sqrt{w \cdot w} = 1 \\ u \cdot v &= 1/2 \times (1 - 3 - 4 + 1) = -2.5 \\ u \cdot w &= 1/2 \times (1 + 3 + 4 + 1) = 4.5 \\ v \cdot w &= 1/4 \times (1 - 1 - 1 + 1) = 0 \end{aligned}$$

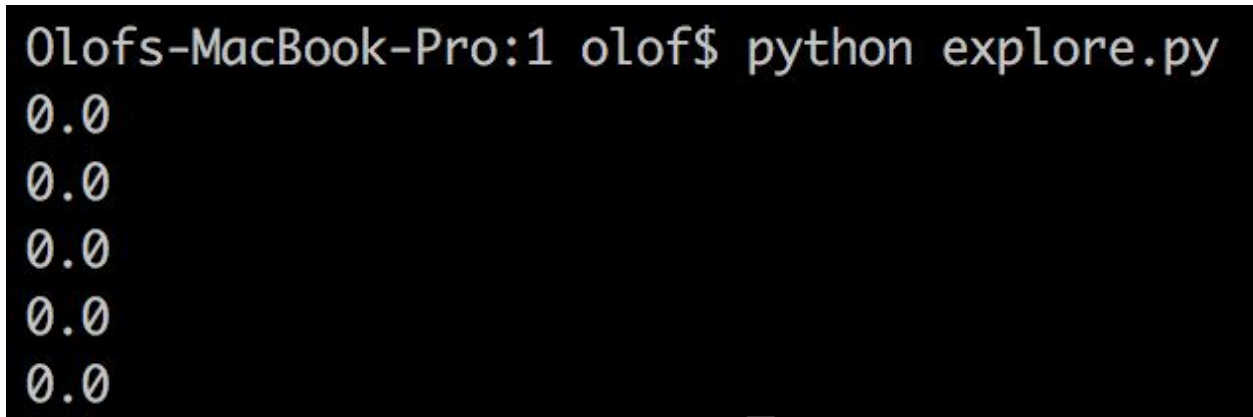
Yes, $v \cdot w$ is orthogonal since their elementwise multiplication when $i \neq j$ is 1 and 0 when $i = j$

The projected $up = -2.5v + 4.5w = [1, -3.5; 3.5, -1]$

7.

```
o1 = np.array([[0, 1, 0], [1, 1, 1], [1, 0, 1], [1, 1, 1]]) / 3
o2 = np.array([[1, 1, 1], [1, 0, 1], [-1, -1, -1], [0, -1, 0]]) / 3
o3 = np.array([[1, 0, -1], [1, 0, -1], [0, 0, 0], [0, 0, 0]]) * 0.5
o4 = np.array([[0, 0, 0], [0, 0, 0], [1, 0, -1], [1, 0, -1]]) * 0.5

print(np.sum(np.multiply(o1, o2)))
print(np.sum(np.multiply(o1, o3)))
print(np.sum(np.multiply(o1, o4)))
print(np.sum(np.multiply(o2, o3)))
print(np.sum(np.multiply(o3, o4)))
```



```
Olofs-MacBook-Pro:1 olof$ python explore.py
0.0
0.0
0.0
0.0
0.0
```

My take on this assignment is that we want to save bandwidth output from the camera. We therefore have the images $\phi_1 - \phi_4$ saved on the camera which will be used to create a representation of an approximation of the image $x_1 - x_4$. This x_s can be sent with a low bandwidth and then be used along side $\phi_1 - \phi_4$ to recreate the approximated image. The $\phi_1 - \phi_4$ are not sent but rather stored in both the camera and wherever the approximated image is recreated.

Projection can be used to create this approximated image. Doing the projection minimizes the 2-norm which means the approximated image is as close as it could be to the original.

```
f = np.array([[-2, 6, 3], [13, 7, 5], [7, 1, 8], [-3, 3, 4]])
o1 = np.array([[0, 1, 0], [1, 1, 1], [1, 0, 1], [1, 1, 1]]) / 3
o2 = np.array([[1, 1, 1], [1, 0, 1], [-1, -1, -1], [0, -1, 0]]) / 3
o3 = np.array([[1, 0, -1], [1, 0, -1], [0, 0, 0], [0, 0, 0]]) * 0.5
o4 = np.array([[0, 0, 0], [0, 0, 0], [1, 0, -1], [1, 0, -1]]) * 0.5
```

```
print(np.sum(np.multiply(f, o1)))
print(np.sum(np.multiply(f, o2)))
print(np.sum(np.multiply(f, o3)))
print(np.sum(np.multiply(f, o4)))
```

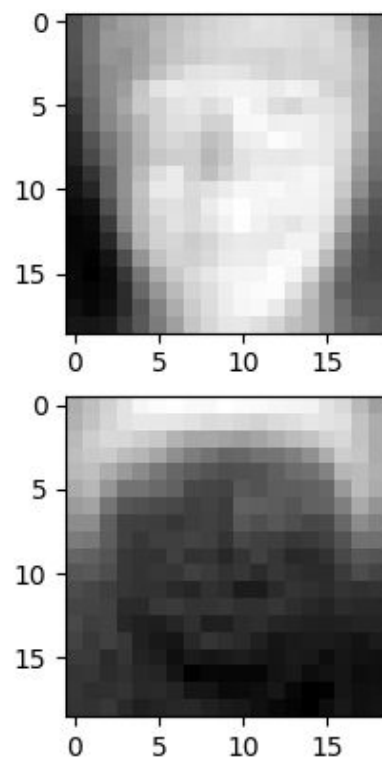
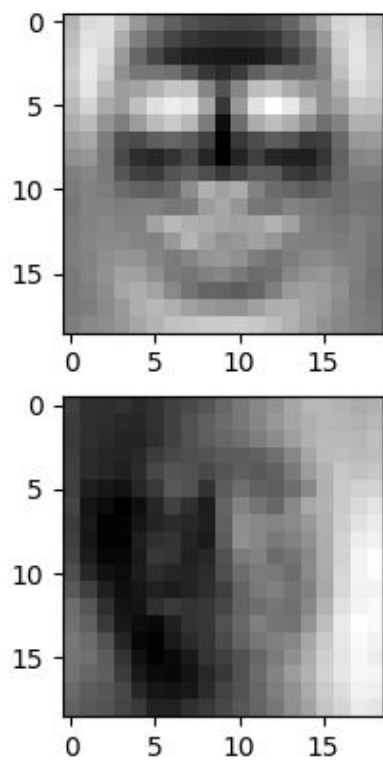
```
16.6666666667
2.0
1.5
-4.0
Olofs-MacBook-Pro:1 olof$
```

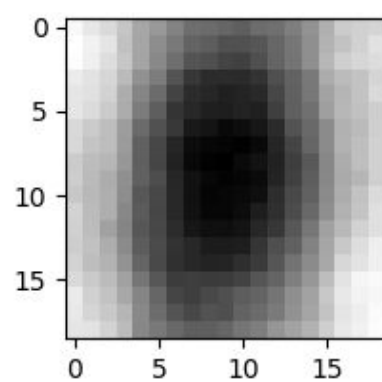
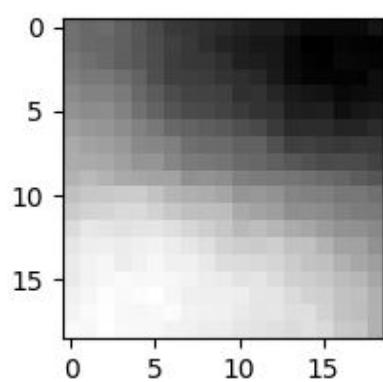
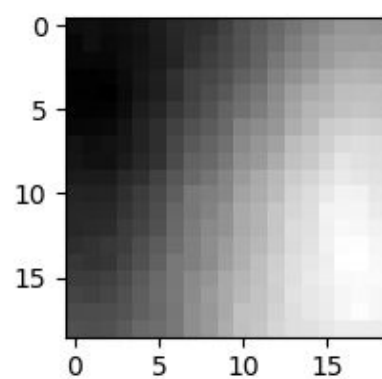
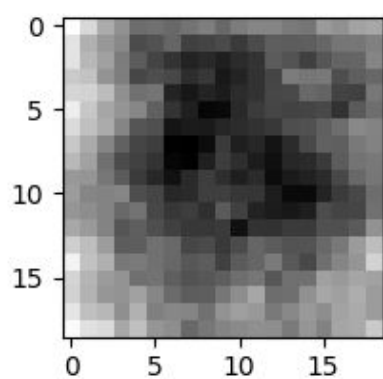
$$up = 16 \phi_1 + 2 \phi_2 + 1.5 \phi_3 - 4 \phi_4 =$$

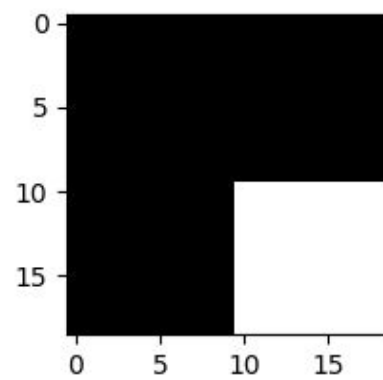
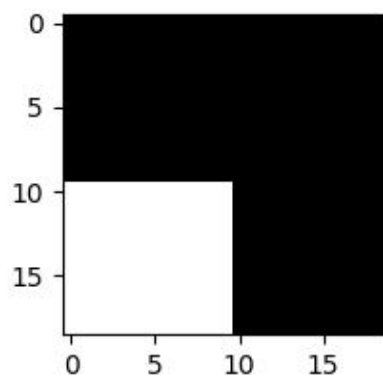
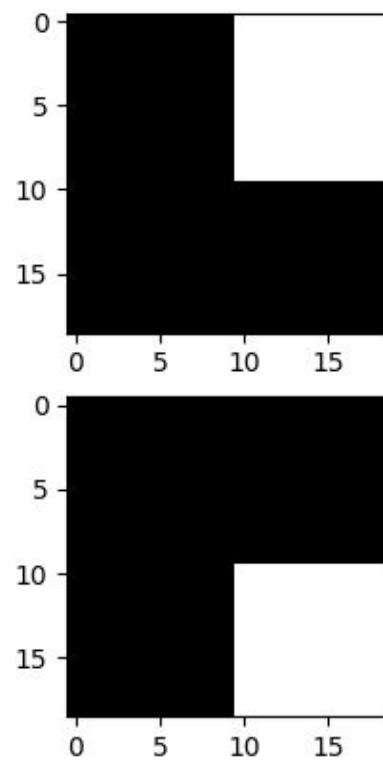
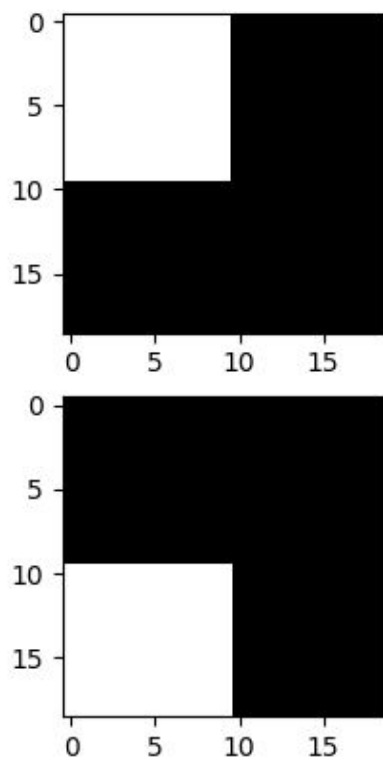
```
[[ 1.41666667  6.          -0.08333333]
 [ 6.75         5.33333333  5.25        ]
 [ 2.66666667 -0.66666667  6.66666667]
 [ 3.33333333  4.66666667  7.33333333]]
```

8.

Bases

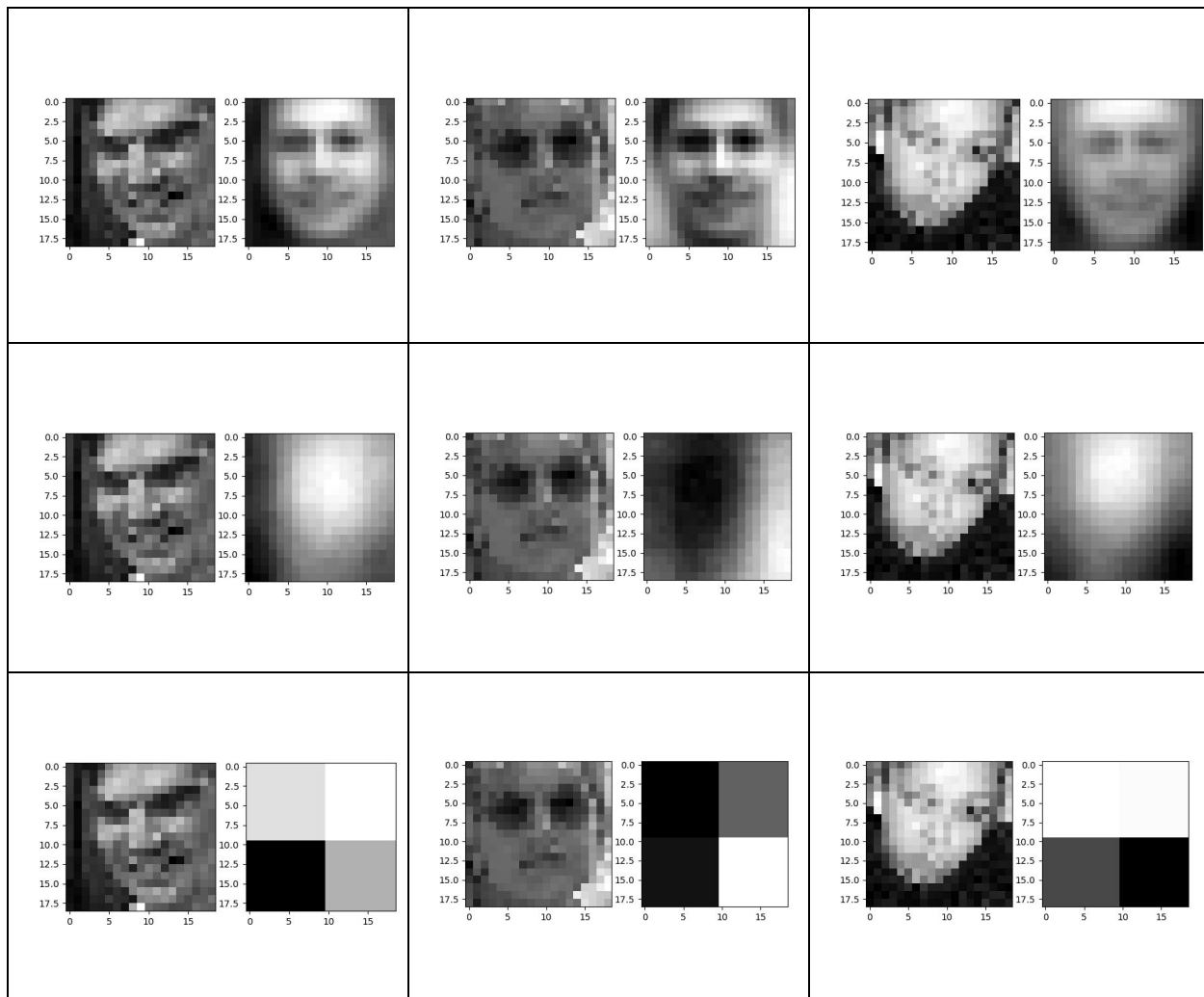




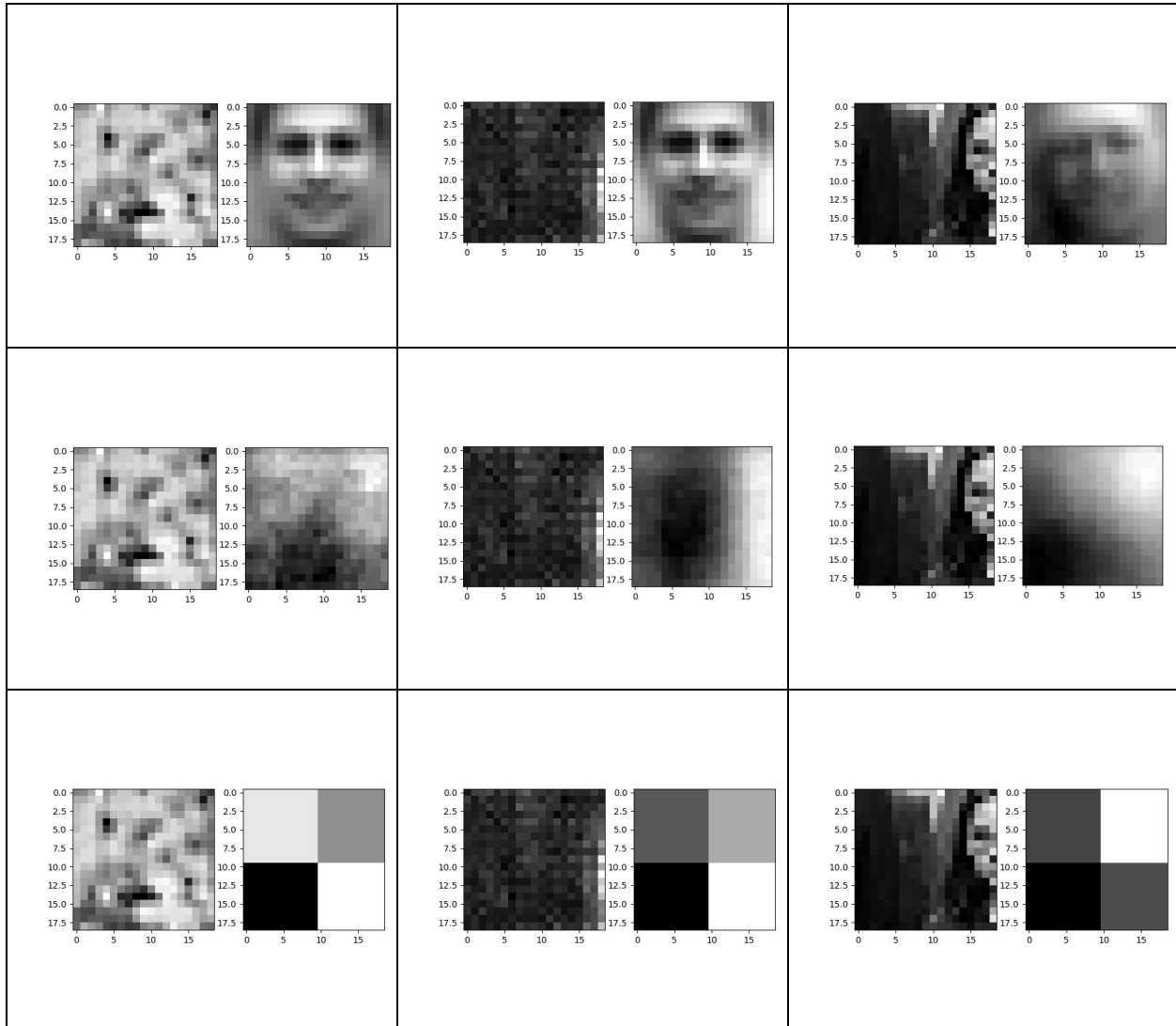


The first base looks somewhat like a face. The second base doesn't look like anything and the third one has a corner white with everything else black.

Stack 1



Stack 2



The left images are the original images whilst the right ones are the projected images. The top row is base #1, second row base #2 and last row base #3.

The images in stack #1 are of faces. The second set of images are a bit more difficult to nail down. In one of the images I can see half of a girl but the other ones doesn't look like anything to me.

Mean

In order of: stack1base1, stack1base2, stack1base3, stack2base1 etc.

```
Mean: 821.027080
Mean: 860.475350
Mean: 944.900862
Mean: 795.190152
Mean: 649.201270
Mean: 697.321408
```

For the first stack, the first base works best as the error norm is the lowest. This makes a lot of sense as both the images and the base represent faces.

For the second stack, the second base works best. This could be because this seemingly general images works better for a general base.

Code in image_basis.py