Project

May 1, 2022

1 Introduction to Image and Video Processing - 1st Project

1.1 Spencer Matei Olson - i6260490

[ ]:
```
# Importing necessary libraries
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

plt.rcParams['figure.figsize'] = [20, 10]

# Custom function to display images nicely using pyplot
def show_images(*images):
    plt.figure()
    if len(images) == 1:
        plt.axis('off')
        plt.imshow(images[0], vmin=0, vmax=255, cmap='gray')
    else:
        f, axarr = plt.subplots(1, len(images))

        for i, a in zip(images, axarr):
            a.axis('off')
            a.imshow(i, vmin=0, vmax=255, cmap='gray')
```

1.2 Exercise 1

1 - In this first exercise, I simply used OpenCV's built-in function to convert the RGB image to HSV (cv.cvtColor()).

If you look at the resulting image, you can see that there are a lot of blues visible. When converting RGB to HSV, the Blue-channel is converted into the Value-channel, therefore, a large amount of blue pixels indicates that the value of the image is relatively high (the image is bright). Looking at the parrots head, this portion of the image is very green. The Green-channel corresponds to the Saturation of the image. Because the color of the birds' feathers are very 'colorful' or saturated, this is what we would expect.

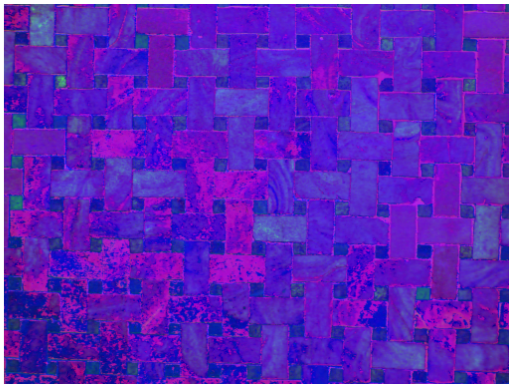
The second image is similar. In the original image of the stone tiles, we notice that the colors are very pale and greyed-out. This corresponds to an HSV image that has a lot of blue but very little green in it!

[ ]:
```
# Loading and converting images from BGR to RGB
birds = cv.imread('./img/birds.jpg')
birds = cv.cvtColor(birds, cv.COLOR_BGR2RGB)
stone = cv.imread('./img/stone.jpg')
stone = cv.cvtColor(stone, cv.COLOR_BGR2RGB)

# Conversion of RGB to HSV image
birds_converted = cv.cvtColor(birds, cv.COLOR_RGB2HSV)
stone_converted = cv.cvtColor(stone, cv.COLOR_RGB2HSV)

print("Images chosen for this exercise:")
show_images(birds, stone)
show_images(birds_converted, stone_converted)
```

```
Images chosen for this exercise:
```

```
<Figure size 1440x720 with 0 Axes>
```

```
<Figure size 1440x720 with 0 Axes>
```



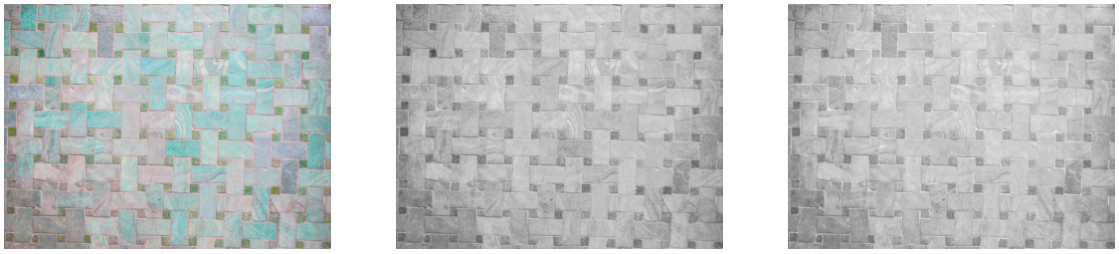
2 - In the second exercise, I used simple NumPy aggegrate functions to calculate the Intensities and Values of the two images. To calculate the Intensities, I take the average/mean of the three color channels. To calculate the Values, I take the largest value of the three color channels per pixel. This results in the following greyscale images.

[ ]:
```
hsi_birds = birds.mean(axis=2)
hsv_birds = birds.max(axis=2)

hsi_stone = stone.mean(axis=2)
hsv_stone = stone.max(axis=2)

print("Images are (from left to right): original, 'I' image, 'V' image")
show_images(birds, hsi_birds, hsv_birds)
show_images(stone, hsi_stone, hsv_stone)
```

```
Images are (from left to right): original, 'I' image, 'V' image
```

```
<Figure size 1440x720 with 0 Axes>
```

```
<Figure size 1440x720 with 0 Axes>
```



1.3 Exercise 2

1 - For the first part of this exercise, I chose two images that were given. One low contrast and one high contrast. Below the images you may see their corresponding histograms.

What we can notice is that the histogram of the low contrast image is concentrated in one spot in the 0-255 range of greys. Looking at the histogram of the second images, you can see that a

majority of greyscale values fall on the very ends of the 0-255 range, meaning that there are a lot of very dark and very bright values.

These two observations support the idea that the first image is low contrast and the second image is high contrast!

2 - Transforming both images using the negative-pointwise transformation, results in an inverted image. So pixels with values x will become 255-x, therefore resulting in dark pixels becoming light and vice versa. You can see those images below the original ones.

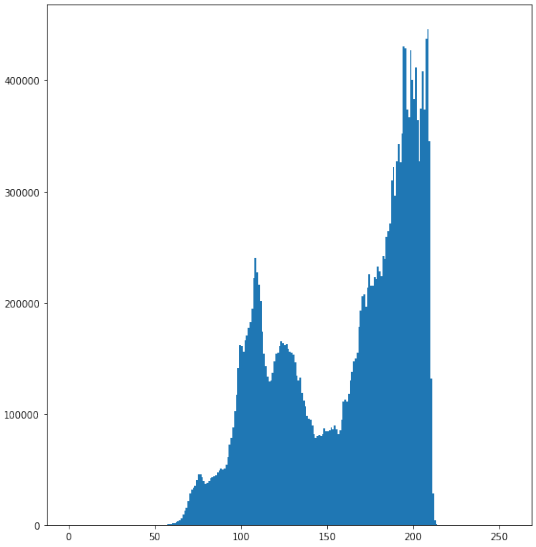
3 - If we take a look at the histograms of these 'inverted' images, in essence, the values of the histogram will be flipped around the midpoint of the x-axis, resulting in the histograms you see below. Because of this, the contrast of the images should not change by applying a negative point-wise transformation!

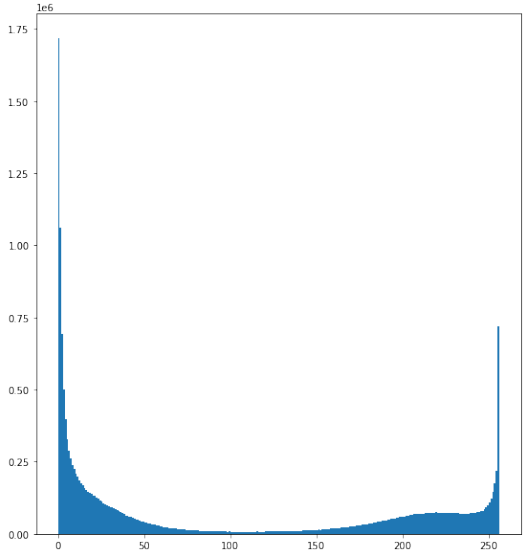
[ ]:
```
# Loading and converting images from BGR to RGB
fog = cv.imread('./img/fog.jpg')
fog = cv.cvtColor(fog, cv.COLOR_BGR2RGB)
shadows = cv.imread('./img/shadows.jpg')
shadows = cv.cvtColor(shadows, cv.COLOR_BGR2RGB)

print('Original images and their corresponding histograms')
show_images(fog, shadows)

# Plotting histograms for images
plt.figure()
_, (a1, a2) = plt.subplots(1, 2)
a1.hist(fog.ravel(), 256, [0, 256])
a2.hist(shadows.ravel(), 256, [0, 256])

# Use negative point-wise transform
#   (I'm choosing to only use one channel because the images are greyscale and␣
 ↪therefore it's irrelevant which channel we consider)
neg_fog = 255-fog[:, :, 0]
neg_shadows = 255-shadows[:, :, 0]

print('"Inverted" images and their corresponding histograms')
show_images(neg_fog, neg_shadows)

# Plotting histograms of negative images
plt.figure()
_, (a1, a2) = plt.subplots(1, 2)
a1.hist(neg_fog.ravel(), 256, [0, 256])
a2.hist(neg_shadows.ravel(), 256, [0, 256])

plt.show()
```

```
Original images and their corresponding histograms
"Inverted" images and their corresponding histograms
```

```
<Figure size 1440x720 with 0 Axes>
```

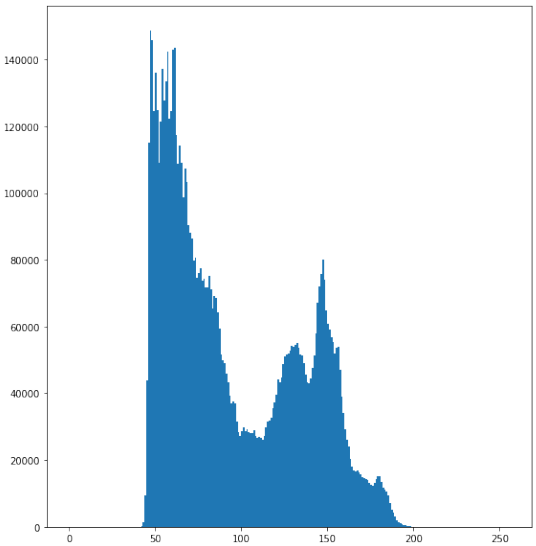
```
<Figure size 1440x720 with 0 Axes>
```

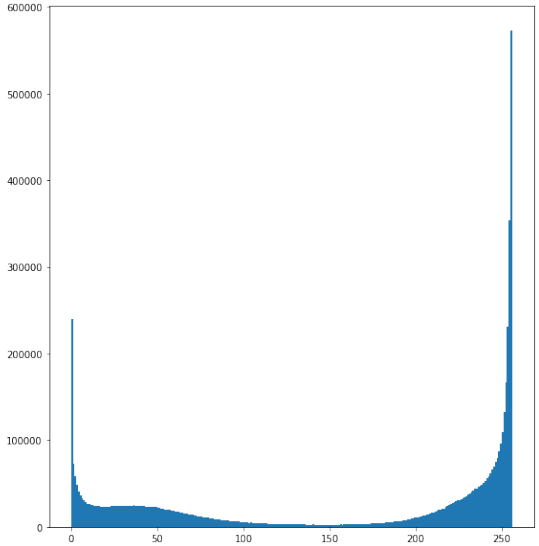





```
<Figure size 1440x720 with 0 Axes>
```

```
<Figure size 1440x720 with 0 Axes>
```





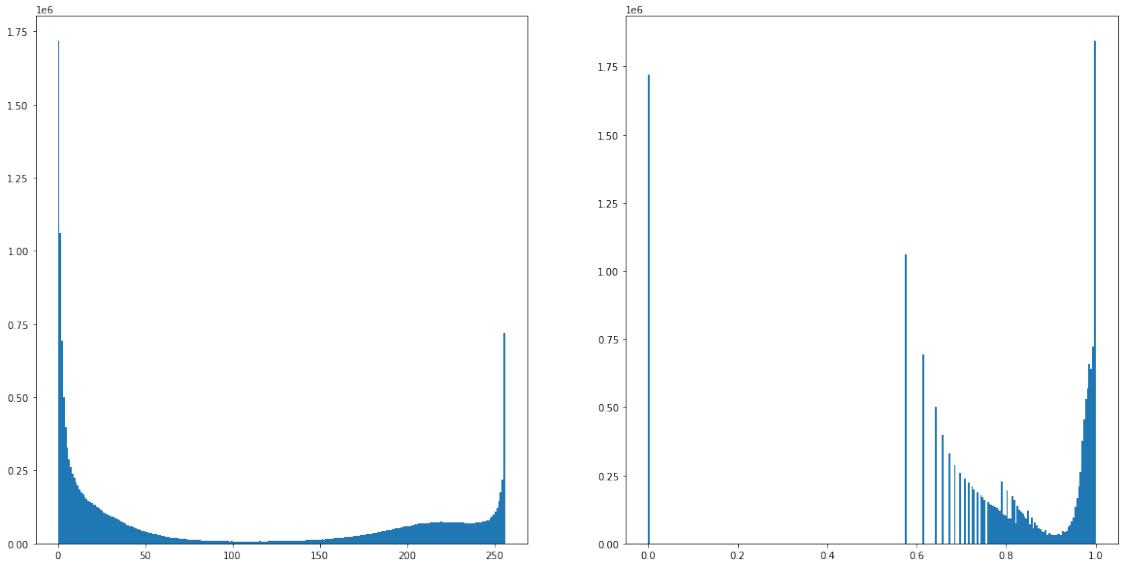
4 - For this exercise, I chose to modify the high-contrast image using the power-rule point-wise transform. To turn this image into a low contrast, I use the power-rule transform with n $<$ 1, where s = r^n. This is to increase the values of the low-valued pixels significantly, while leaving the high-valued pixels relatively unchanged. After performing the transform, I normalize the image back to the range of 0-255.

As we can see in the resulting histogram, what this transformation does is concentrate the values of the pixels on the higher end of the 0-255 range. From our observations in the previous exercise, we know that when the histogram of an image is concentrated into one range, then the corresponding image appears to have low contrast.

We can see in the images below, that this observation seems to be correct!

[ ]:
```
# Use power-rule point-wise transform
pow_shadows = shadows.copy()
n = 0.1

# Applying power rule, then converting to 0-1 range
pow_shadows = (shadows**n)/255
# Normalizing image
pow_shadows = pow_shadows*(1/np.max(pow_shadows))
show_images(shadows, pow_shadows)

# Plotting histograms of power-rule images
f, (a1, a2) = plt.subplots(1, 2)
plt.figure()
a1.hist(shadows.ravel(), 256, [0, 256])
a2.hist(pow_shadows.ravel(), 256, [0, 1])

plt.show()
```

```
<Figure size 1440x720 with 0 Axes>
```




```
<Figure size 1440x720 with 0 Axes>
```

1.4 Exercise 3

1 - For this exercise, I chose the thin petal-ed flower to demonstrate what converting an image to polar coordinates look like.

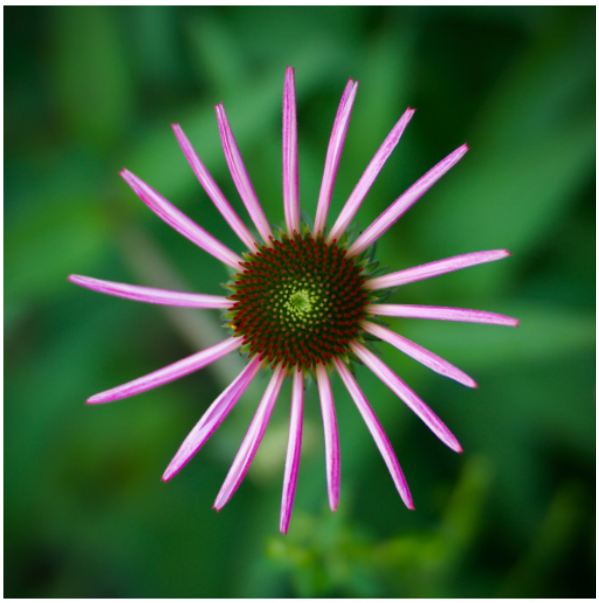
To achieve this conversion, I used one of opencv-s built-in functions, warpPolar(). For this transformation, I tell the function to use the center of the image as the 0-radius point, then I take the height of the image as the max-radius.

The function then returns an image where the horizontal axis corresponds to the radius of a pixel, and the vertical axis corresponds to the angle of a pixel. Because the flower is nicely centered in the image, the center of the flower is converting into a rectangle along the vertical-axis and radiating petals line-up perpendicular to the center of the flower.

Note: the image returned by warpPolar seems to have a vertical-axis that ranges from 0 (at the top) to -360 (at the bottom).

[ ]:
```
flower = cv.cvtColor(cv.imread('img/flower.jpeg'), cv.COLOR_BGR2RGB)

flower_polar = cv.warpPolar(flower, (0, 0), (flower.shape[1]/2, flower.shape[0]/
  ↪2), flower.shape[0]/2, None)

show_images(flower, flower_polar)
```

```
<Figure size 1440x720 with 0 Axes>
```



2 - For the second exercise, I chose a different flower than the first one. To demonstrate the cartoonification of an image, I choose a flower with 'fuller' shapes to demonstrate the monotonic colors inside the black borders better.

My approach of cartoonification involves several steps. In broad terms, I first manually quantize the image into 3 bins per color channel.

After quantizing, I use the built-in Canny() function to detect the edges of the quantized image. This function returns a black image where white values indicate edges in the image. To get larger and more defined lines, I blur (using built-in kernel blur) then quantize this image to either 0 or 255 to ensure that all pixels are either black or white.

I then set all pixels in the quantized image to black wherever white pixels appear in the edge-detection image.

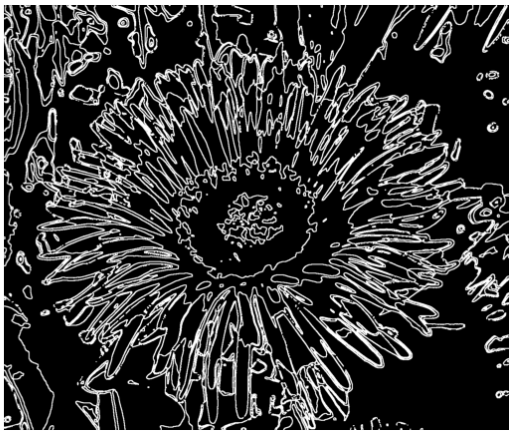
After some experimenting, I found that I also received cleaner quantized images if I blurred the original image beforehand

[ ]:
```
pink = cv.cvtColor(cv.imread('./img/pink.jpg'), cv.COLOR_BGR2RGB)

# Cartoonifying flower

# Blur original image
pink_blur = cv.blur(pink, ksize=(11, 11))

# Quantize image
n = 2.0
quant = (np.round(pink_blur*n/255)*(255/n)).astype(np.uint8)
```

```
# Get edges of quantized image, then blur them
pink_edges = cv.Canny(quant, 200, 200)
pink_edges = cv.GaussianBlur(pink_edges, ksize=(3, 3), sigmaX=1, sigmaY=1)
# Quantize edges
pink_edges[pink_edges > 0] = 255

quant_copy = quant.copy()
# Set all pixels that are white in edges to black on quantized image
quant[pink_edges == 255] = 0

# Original image and Cartoonified image
show_images(pink,quant)

# Quantized image without edges, edges after blurring and quantizing
show_images(quant_copy,pink_edges)
```

```
<Figure size 1440x720 with 0 Axes>
```

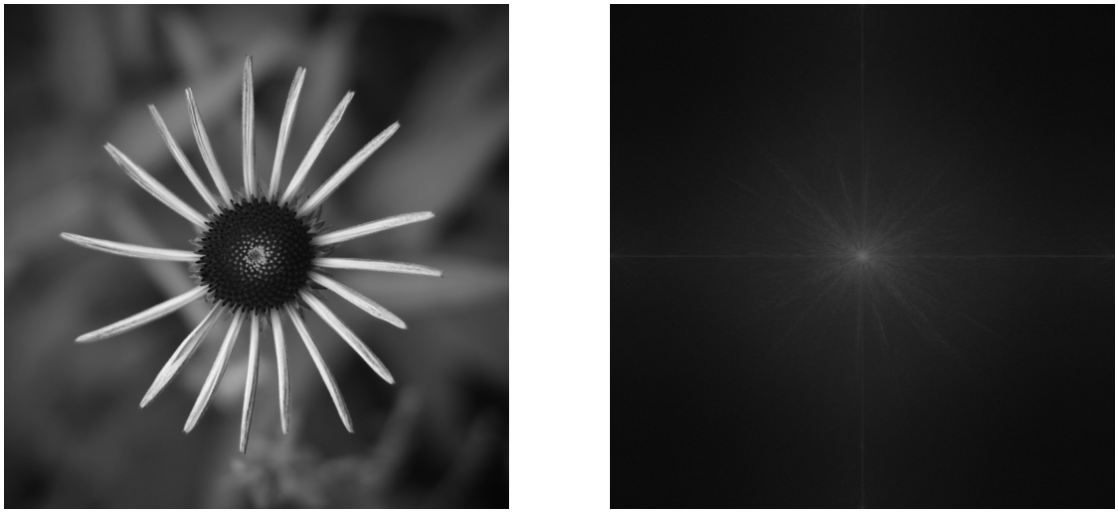
```
<Figure size 1440x720 with 0 Axes>
```



1.5 Exercise 4

[ ]:
```
tower = cv.imread('./img/flower.jpeg')
tower = cv.cvtColor(tower, cv.COLOR_RGB2GRAY)
tower_rot = cv.warpAffine(tower, np.float32([[1, 0, tower.shape[1]/2], [0, 1,␣
↪0]]), tower.shape[::-1])

def getFFT(image):
    image = image/255
    result = np.log(abs(np.fft.fftshift(np.fft.fft2(image))))
    # result = (result * 255 / np.max(result)).astype(np.uint8)
    return result*10

fft_tower = getFFT(tower)
fft_tower_rot = getFFT(tower_rot)

show_images(tower, fft_tower)
show_images(tower_rot, fft_tower_rot)
```

```
<Figure size 1440x720 with 0 Axes>
```



<Figure size 1440x720 with 0 Axes>

1.6 Exercise 5

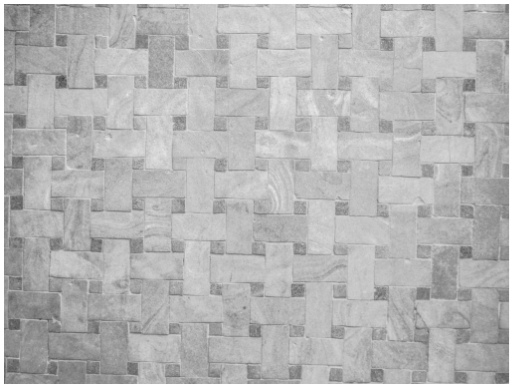
[ ]:
```
shadows = cv.cvtColor(cv.imread('./img/stone.jpg'), cv.COLOR_BGR2GRAY).
 ↪astype(np.uint32)

angle = 0
wavelength = 2.5

x = np.arange(0, shadows.shape[1], 1)
y = np.arange(0, shadows.shape[0], 1)
```

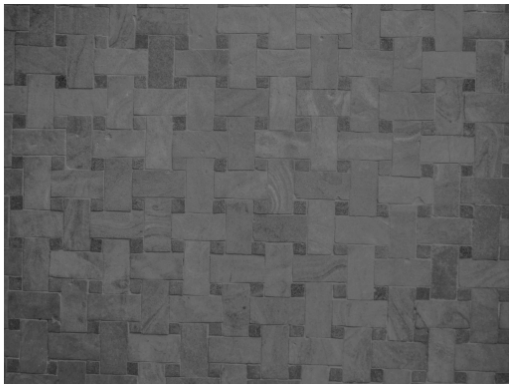
```
X, Y = np.meshgrid(x, y)

magnitude = 0.9
noise = np.sin(
    (2*np.pi*X/wavelength)
)*magnitude/2+0.5
# noise = noise + cv.randu(np.zeros_like(noise), -0.1, 0.1)
print(np.max(noise))
print(np.min(noise))

noisy_image = np.multiply(noise, shadows)
noisy_image = noisy_image - np.min(noisy_image)
noisy_image = noisy_image*255/np.max(noisy_image)

show_images(shadows, noisy_image)
show_images(noise*255)

fft = getFFT(shadows)
noisy_fft = getFFT(noisy_image)
show_images(fft, noisy_fft)
plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, noisy_fft)
```

```
0.9279754323329603
0.07202456766695697
```

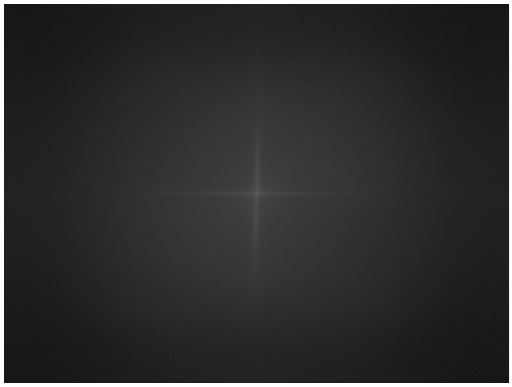
[ ]: `<matplotlib.contour.QuadContourSet at 0x7f0003b18430>`

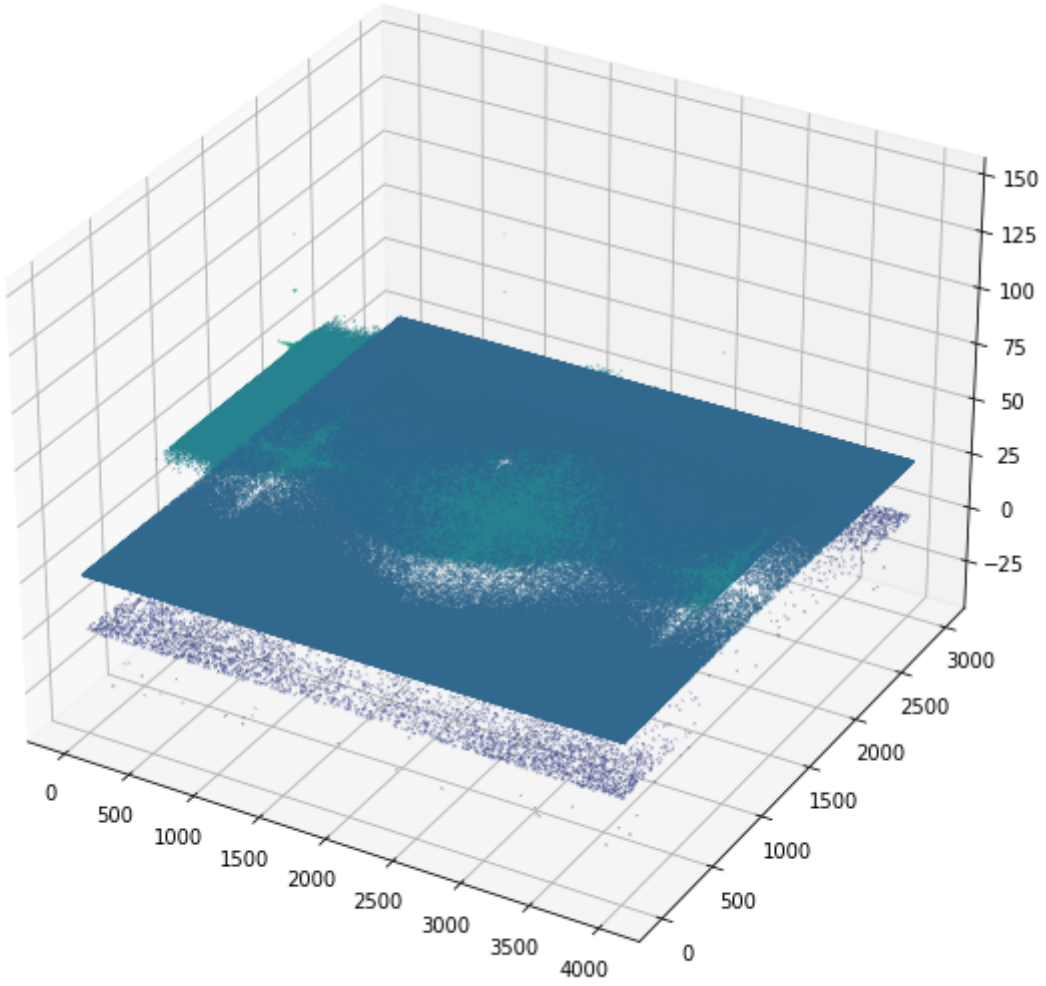
```
<Figure size 1440x720 with 0 Axes>
```





```
<Figure size 1440x720 with 0 Axes>
```




150
125
100
75
50
25
0
−25
0
500
1000
1500
2000
2500
3000
3500
4000
0
500
1000
1500
2000
2500
3000