Digital Image Processing CA

Kamyar Bagha AI student

Khatam University

Project 2-2:

Reducing the Number of Intensity Levels in an Image

```
def changeLevels(desiredLevels):
    k = np.log2(desiredLevels)
    intensity_level = 2**(8-k)
    target_compr_factor = 256/intensity_level
    image_reduced = np.floor(image/256 * target_compr_factor)
    return image_reduced
```

With usage of this function we can change the desired levels of intensity and get the corresponding image.

For image in Fig. 2.21 we can evaluate this function and show the results.

```
# %%
plt.figure(figsize=[30,30])
for k in [2**i for i in range(1,9)]:

    plt.subplot(3,3 ,int(np.log2(k)))
    plt.imshow(changeLevels(k), cmap='gray')
    plt.title('Grey-level  '+str(k), fontsize=30)
Run Cell | Run Above | Debug Cell
```

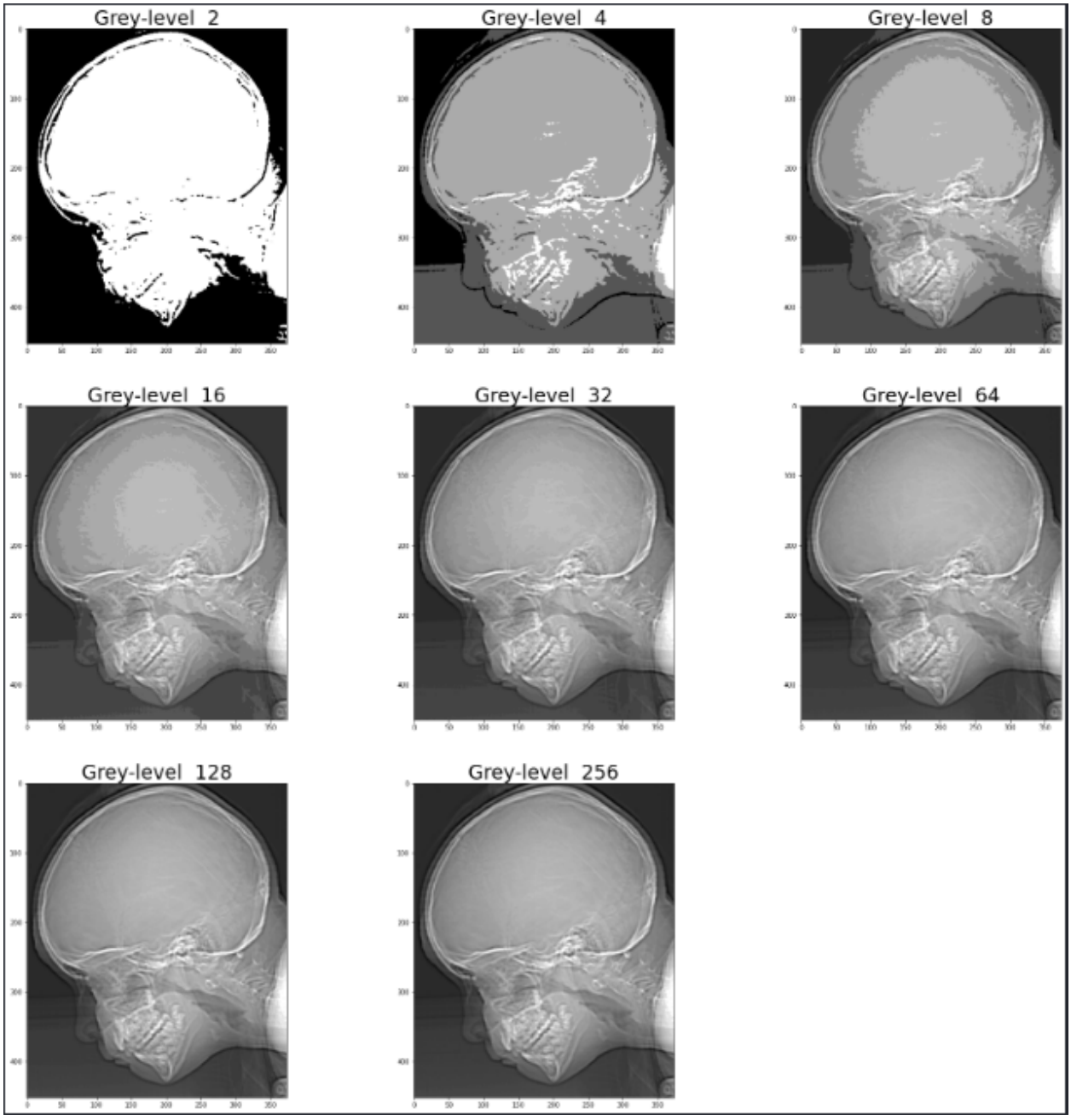


Fig - showing the image for each desired intensity levels

Project 2-3

Zooming and Shrinking Images by Pixel Replication

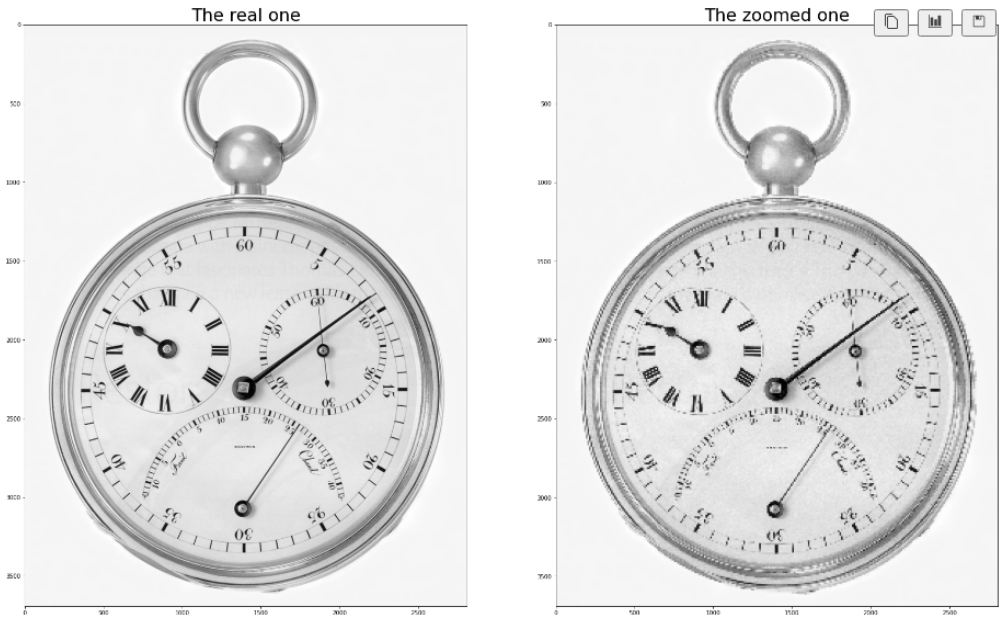


Fig - the real image and the result of shrinking and the zooming of the same pic

We can see that some information has been lost and the real image has much more details and information in the edges. But the final image was not too far from the real image.

```
k = 10
resized = cv2.resize(image, (image.shape[1]//k,image.shape[0]//k),
                     interpolation = cv2.INTER_NEAREST)


Run Cell | Run Above | Debug Cell | Go to [3]
#%%
for m in range(resized.shape[1]):
    for n in range(resized.shape[0]):
        if n==0:
            hstack = resized[n][m]+ np.zeros((10,10))
        else:
            hstack =np.concatenate([hstack , resized[n][m]+np.zeros((10,10))],
                                   axis=0)
    if m==0:
        vstack = hstack.copy()
    else:
        vstack = np.concatenate([vstack, hstack], axis=1)
```

This is the code snippet of this project. You can change the K- factor of shrinking and resizing the image.

Project 2-4

Zooming and Shrinking Images by Bilinear Interpolation

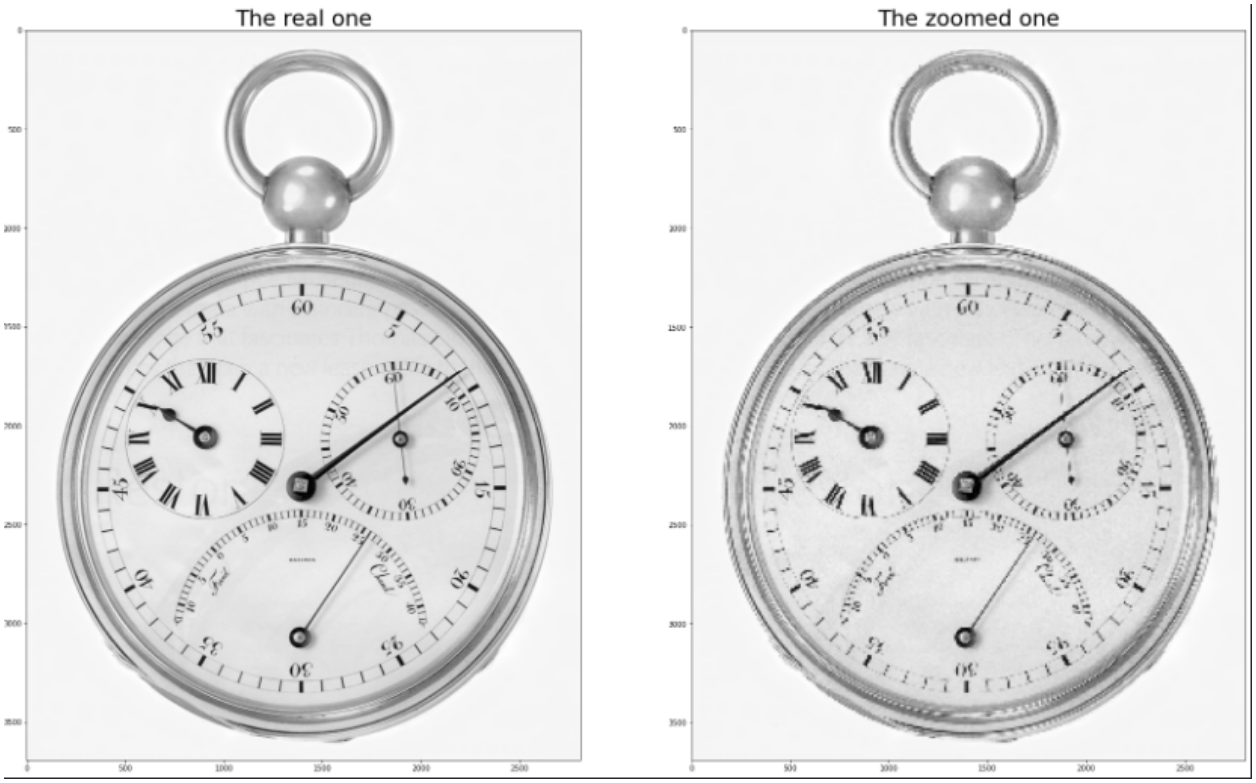


Fig - with interpolation in zooming

```
✓ print(im.info['dpi']) ···

(1250.0, 1250.0)
```

```
k = 10
resized = cv2.resize(image, (image.shape[1]//k,image.shape[0]//k),
                     interpolation = cv2.INTER_LINEAR)
```

We used the function in the CV2 library which handles linear interpolation by itself.

We saw that the result of the image is much better than the previous project.

Project 2-5

Arithmetic Operations

```python
import cv2
import matplotlib.pyplot as plt
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
from PIL import Image


Run Cell | Run Above | Debug Cell
# %%
def arthimeticOp_add(img1 , img2):
    return cv2.add(img1, img2)

def arthimeticOp_multiply(img1, img2):
        return cv2.multiply(np.array(img1),img2)

def arthimeticOp_divide(img1, img2):
    return cv2.multiply(np.array(img1),img2)

def arthimeticOp_subtract(img1, img2):
    return cv2.subtract(np.array(img1),img2)
Run Cell | Run Above | Debug Cell
```

We can see that easily we can implement the arithmetic operations by making four functions.

Project 3-1

Image Enhancement Using Intensity Transformations

The focus of this project is to experiment with intensity transformations to enhance an image.

Fig - the original image in book

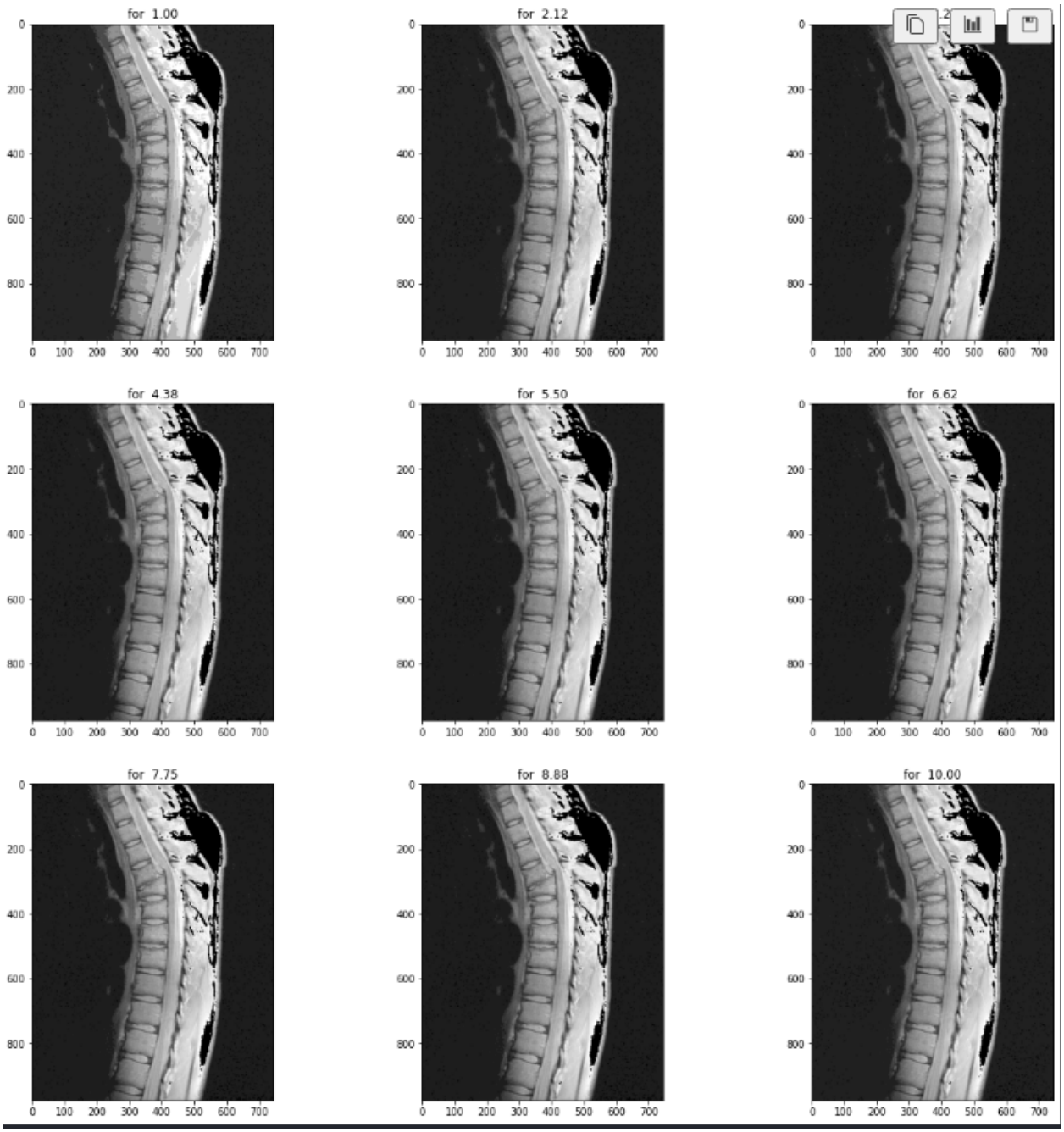


Fig - for changing c between 1 and 10

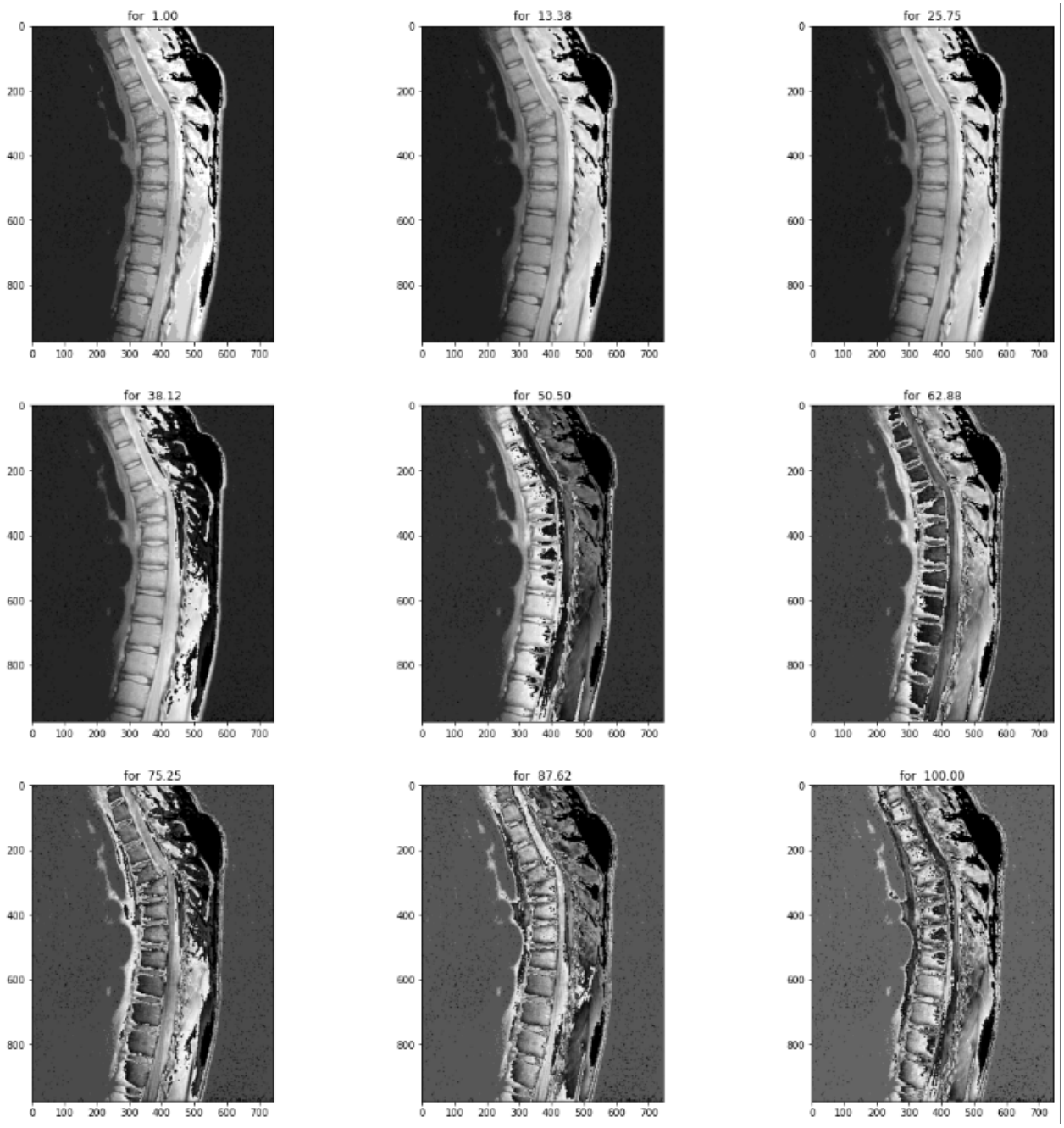


Fig - for changing c between 1 and 100

We can see that for the Log-transformation if C were bigger than 15 the image does not imply good information, so we can settle for c about 2 to 5.

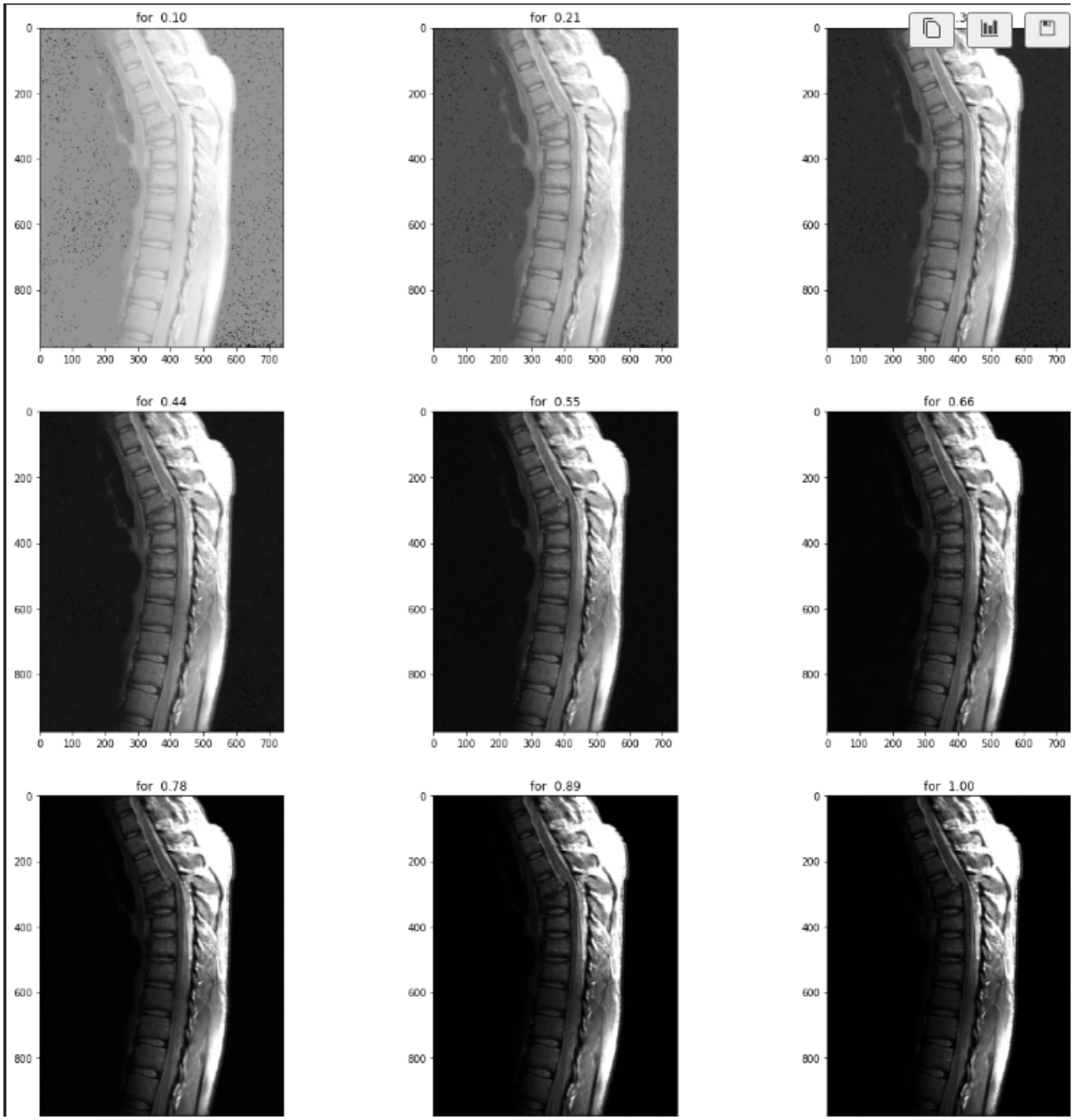


Fig - for power transformation changing respect to c

So we can choose 0.5 for the second parameter of power transformation.

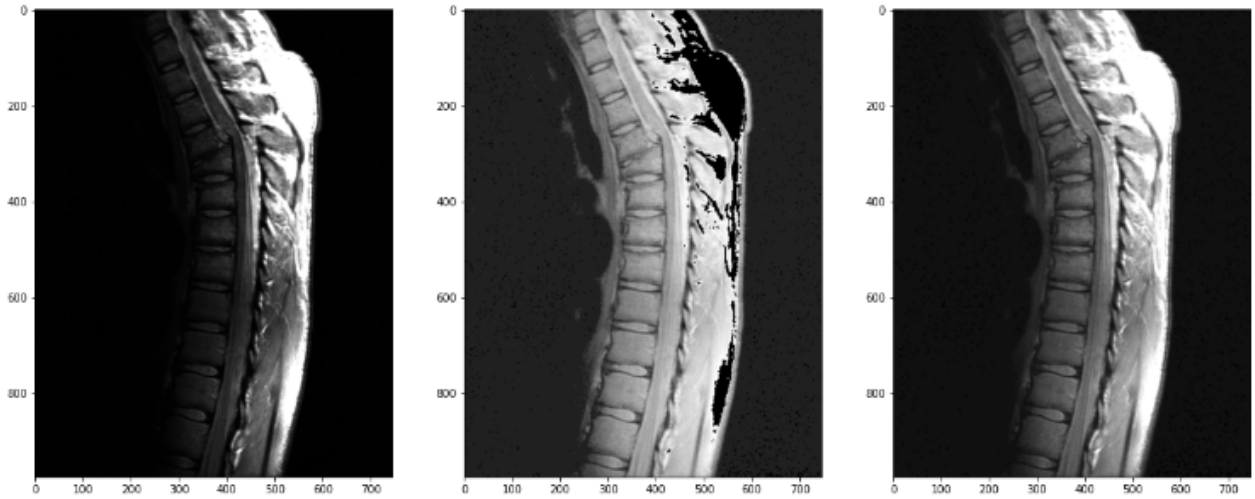


Fig - The final result for log transformation and power transformation

---

project 03-02

Histogram Equalization

Fig- the left is the original image and the right side is the output of histogram equalized

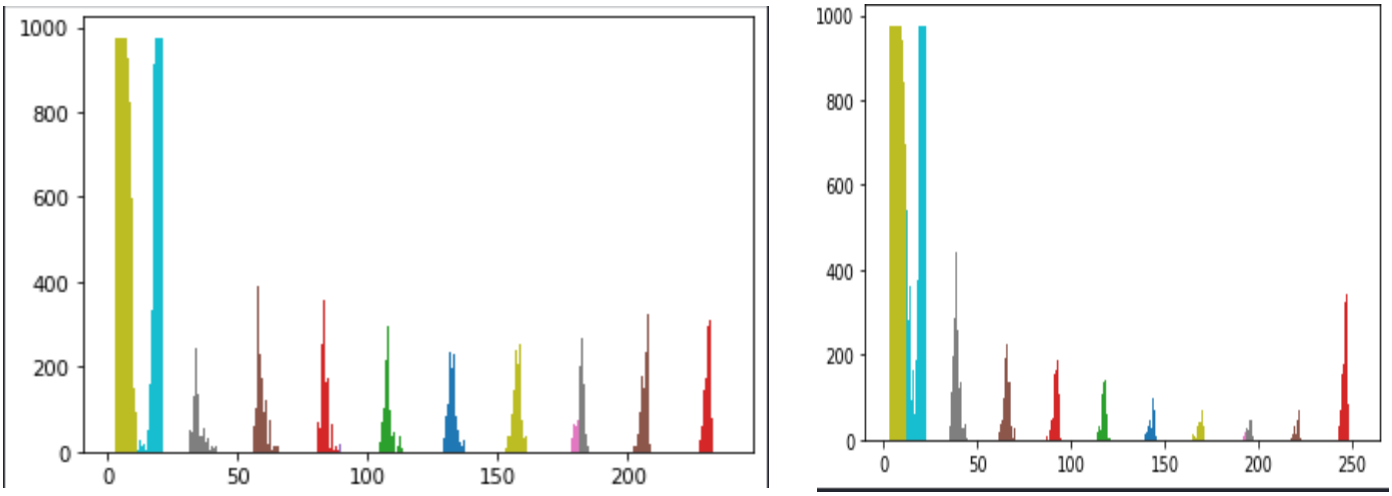


Fig- the right one is the main histogram and the left one is the equalized of image

---

Project 3-3

Spatial Filtering

```
def convolution2d(image, kernel, pad):
    m, n = kernel.shape
    image = cv2.copyMakeBorder(image, pad, pad, pad, pad, cv2.BORDER_CONSTANT, value=0)
    y, x = image.shape
    y_out = y - m + 1
    x_out  = x - n + 1
    new_image = np.zeros((y_out, x_out))
    for i in range(y_out):
        for j in range(x_out):
            new_image[i][j] = np.sum(image[i:i+m, j:j+n]*kernel)
    return new_image
```

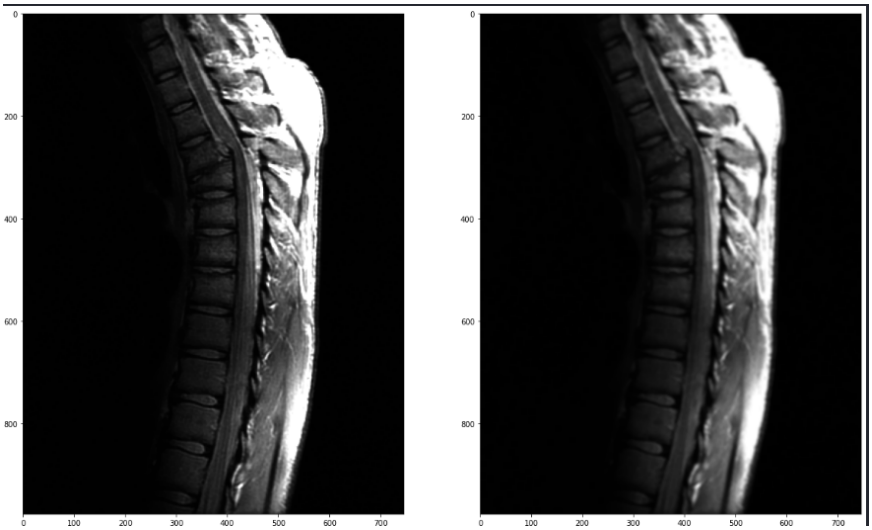


Fig- Using kernel for blurring(with convolution)

PROJECT 03-04

Enhancement Using the Laplacian

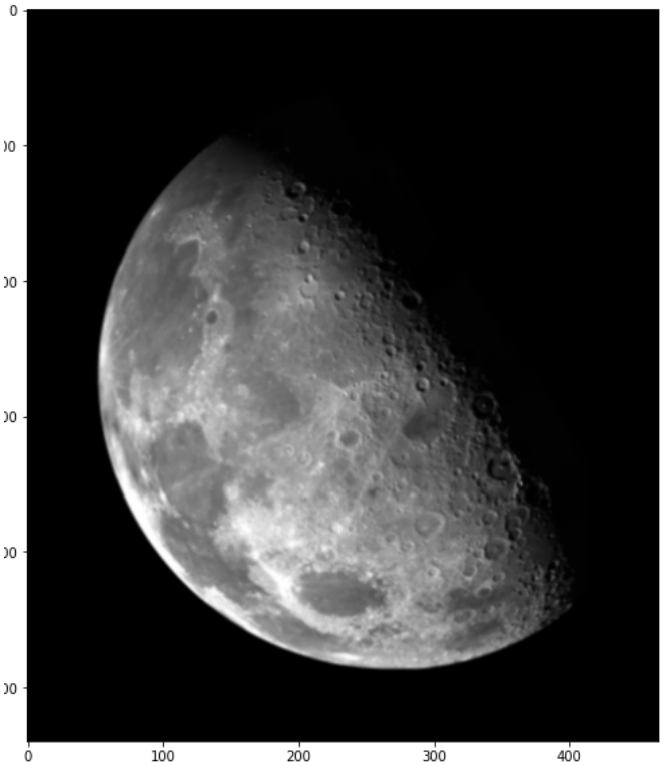


Fig - the original image

```
img = plt.imread('./Fig0338(a)(blurry_moon).tif')
c = 1
m= 3
laplace_kernel= np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]])
laplacian = convolution2d(img, laplace_kernel,1)
laplacian += min([min(a) for a in laplacian])
img2 = img +np.clip(c* laplacian, 0, 255)
plt.figure(figsize=(10,10))

plt.imshow(img2, cmap='gray')
plt.figure(figsize=(10,10))

plt.imshow(laplacian, cmap='gray')
```

With use of this code snippet we can filter with an amount corresponding to the Laplacian filter.

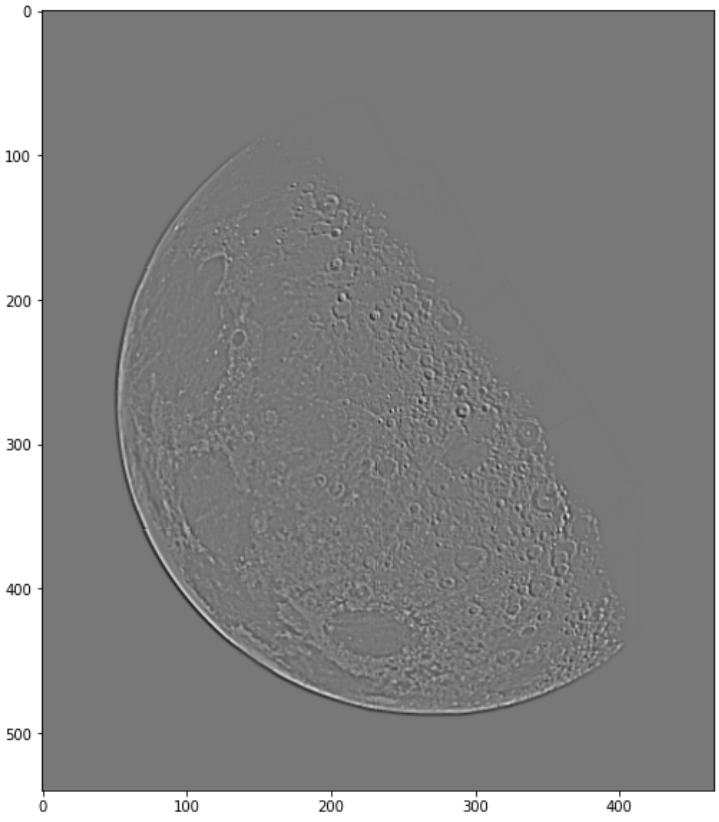


Fig - the final result of the Laplacian filter

PROJECT 04-01
Two-Dimensional Fast Fourier Transform

The function that calculates the 2D Fourier transform in Python is np. fft. fft2() . FFT stands for Fast Fourier Transform and is a standard algorithm used to calculate the Fourier transform computationally.

numpy.fft.fft2

`fft.fft2(a, s=None, axes=(- 2, - 1), norm=None)` [source]

Compute the 2-dimensional discrete Fourier Transform.

This function computes the *n*-dimensional discrete Fourier Transform over any axes in an *M*-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

```
def multiply_minus1(img):

    df = pd.DataFrame(np.zeros((img.shape[0], img.shape[1])), dtype=int)
    for i in range(img.shape[0]):
        if i%2==0:
            df.loc[i,:] = [ 1 if i%2==0 else -1 for i in range(img.shape[1])]
        else:
            df.loc[i,:] = [ -1 if i%2==0 else 1 for i in range(img.shape[1])]

    df = df* img
    return df.to_numpy()
```

With use of this function we can multiply pixelvise by $(-1)^{x+y}$
Multiply the resulting (complex) array by a real filter function. Then, compute the inverse Fourier transform and finally Multiply the result by $(-1)^{x+y}$ and take the real part. Also we know that numpy.fft has the the fftshift and the ifftshift for exactly doing the same approaches.

```
dft = np.fft.fft2(multiply_minus1(img))
img_fft= 50*np.log(np.abs(dft))
inverse_img= multiply_minus1(np.fft.ifft2((dft))).real
```

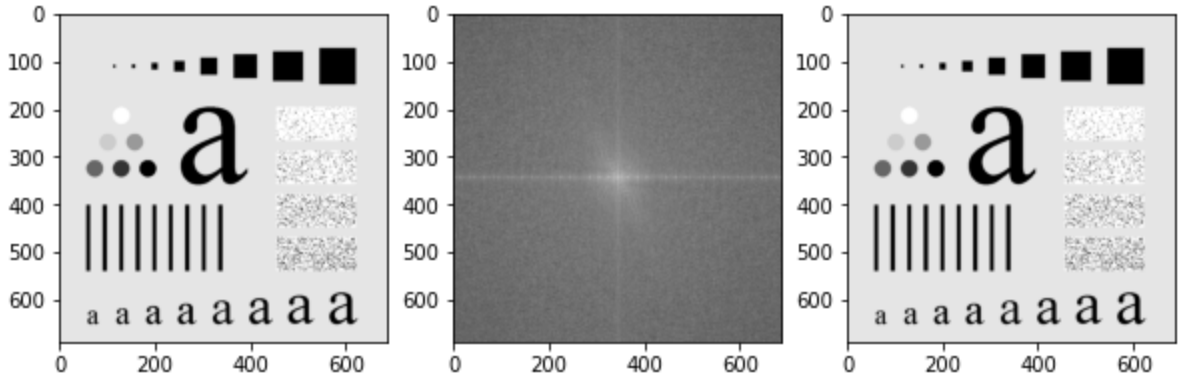
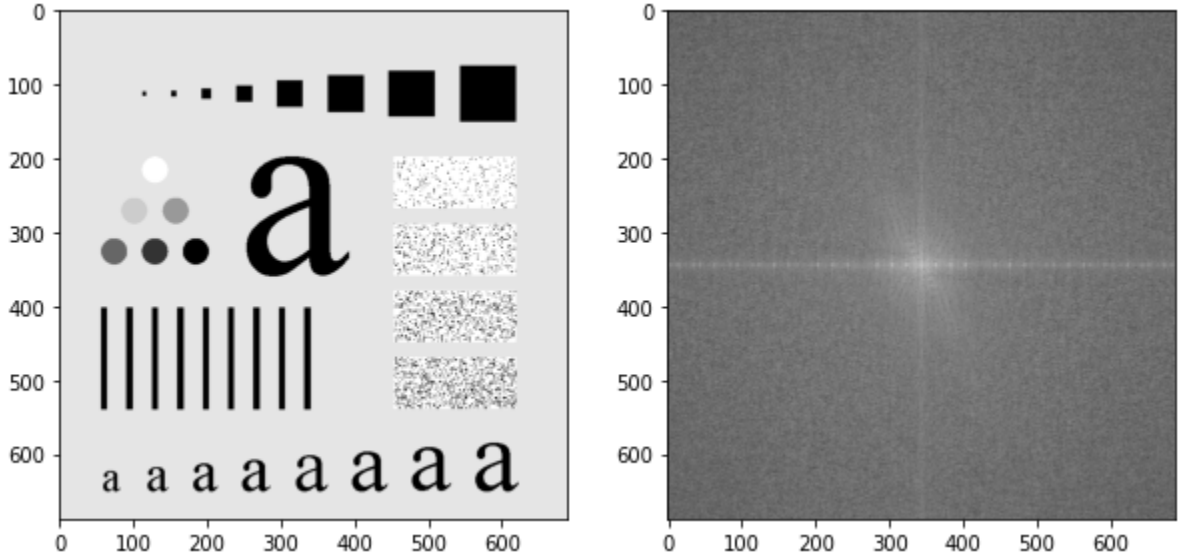


Fig- the left one is the original image, in the middle is the spectrum of image
and the later one is The inverse FFT and get the image exactly the same as Original output.

PROJECT 04-02

Fourier Spectrum and Average Value



```
the average value of the image is : 207.31
 the sum value of the image is : 98131169.00
 the value of spectrum in center is  : 98131169.00
```

PROJECT 04-03
Lowpass Filtering

Gaussian low pass filter in Python
A Gaussian Filter is a low pass filter used for reducing noise (high frequency components) and blurring regions of an image. The filter is implemented as an Odd sized Symmetric Kernel (DIP version of a Matrix) which is passed through each pixel of the Region of Interest to get the desired effect.

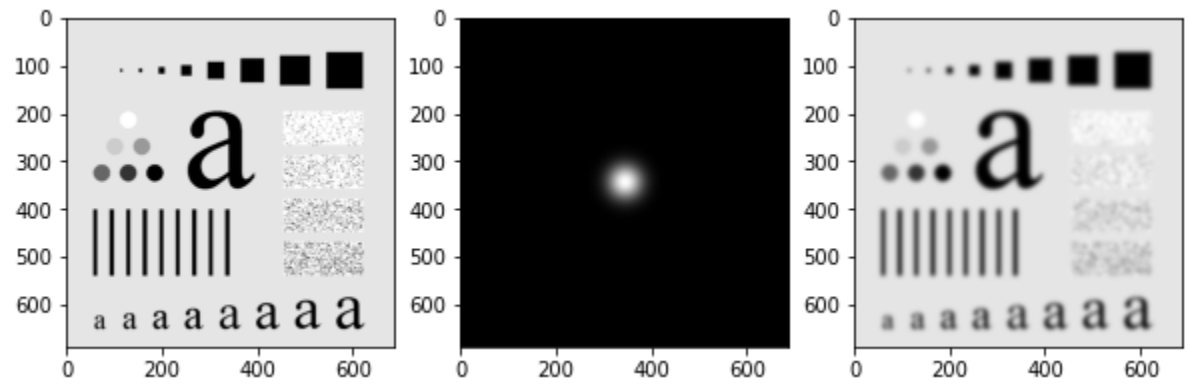


Fig- the left is the original the middle is the gaussian filter
Which only allows low frequencies, the last one is the result
Of applying this filter to the original image.

```
def get_gauss_kernel(size=[4,4],sigma=1, center=None):
    if center is None:
        center=(size[0]//2 ,size[1]//2)
    kernel=np.zeros((size[0],size[1]))
    for i in range(size[0]):
       for j in range(size[1]):
          diff=np.sqrt((i-center[0])**2+(j-center[1])**2)
          kernel[i,j]=np.exp(-((0.4*diff)**2)/(2*sigma**2))
    return kernel/np.sum(kernel)
```

```
gaussianLPF= get_gauss_kernel([img.shape[0], img.shape[1]] , sigma=10)
Run Cell | Run Above | Debug Cell | Go to [6]
# %%
filterdspectrum = gaussianLPF * dft
inverse_img= multiply_minus1(np.fft.ifft2((filterdspectrum))).real

plt.figure(figsize=(10,10))
plt.subplot(131),plt.imshow(img, cmap = 'gray')
# plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(gaussianLPF, cmap = 'gray')
# plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(inverse_img, cmap = 'gray')
```

PROJECT 04-04

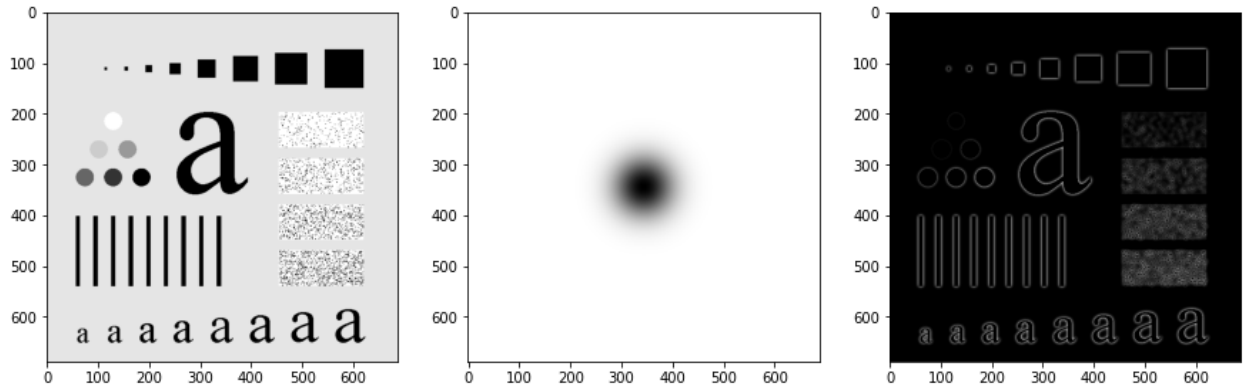
High Pass Filtering

High pass filter" is a very generic term. There are an infinite number of different "highpass filters" that do very different things (e.g. an edge detection filter, as mentioned earlier, is technically a highpass (most are actually a bandpass) filter.

```
def GaussHPF(size=[4,4],sigma=1, center=None):
    if center is None:
        center=(size[0]//2 ,size[1]//2)
    kernel=np.zeros((size[0],size[1]))
    for i in range(size[0]):
       for j in range(size[1]):
          diff=np.sqrt((i-center[0])**2+(j-center[1])**2)
          kernel[i,j]= 1 - np.exp(-((0.5*diff)**2)/(2*sigma**2))
    return kernel/np.sum(kernel)
```

```
dft = np.fft.fft2(multiply_minus1(img))
gaussianHPF= GaussHPF([img.shape[0], img.shape[1]] , sigma=20)
filterdspectrum = gaussianHPF * dft
inverse_img= multiply_minus1(np.fft.ifft2((filterdspectrum))).real
```

Defining the Gaussian filter for filtering the image and finally apply it to the image spectrum and get the inverse fourier transform and getting the final results.

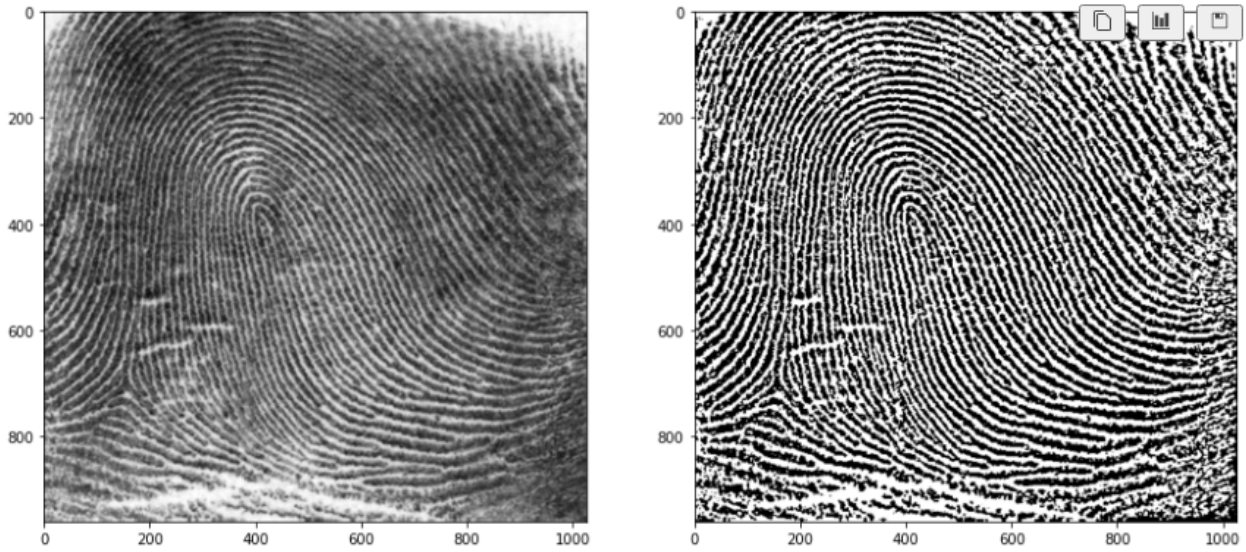


PROJECT 04-05

High Pass Filtering Combined with Thresholding

Everything is similar to the previous project, the only difference is to make a threshold for reaching the binary image. We just make this thresholding using the median of intensities.

```
inverse_img[ inverse_img > np.median(inverse_img)] = 255
inverse_img[ inverse_img < np.median(inverse_img)] = 0
```



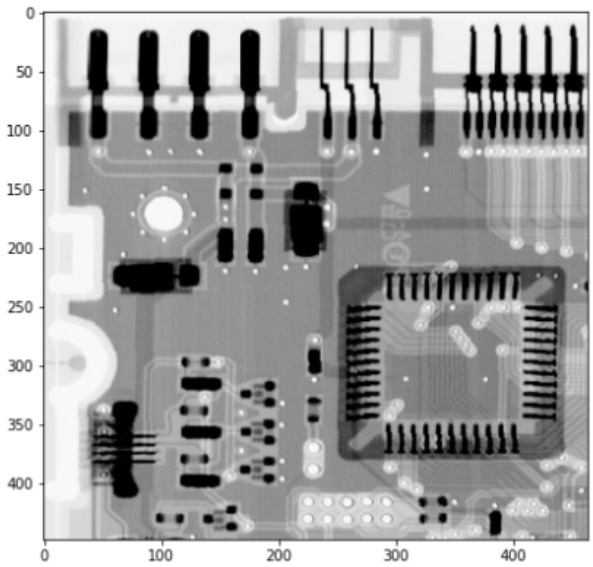
PROJECT 05-01

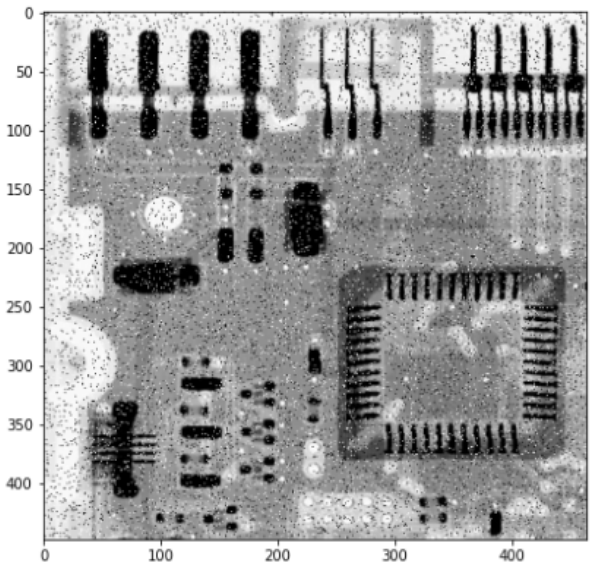
Noise Generators

The Function adds gaussian , salt-pepper , poisson and speckle noise in an image

```
def noisy(noise_typ,image, mean=0 , sigma=1, sprob=0.5,pprop=0.5, amount_sp=0.004):
    if noise_typ == "gauss":
      row,col= image.shape
      gauss = np.random.normal(mean,sigma,(row,col))
      gauss = gauss.reshape(row,col)
      noisy = image + gauss
      return noisy
    elif noise_typ == "s&p":
      row,col = image.shape
      out = np.copy(image)
      # Salt mode
      num_salt = np.ceil(amount_sp * image.size * sprob)
      coords = [np.random.randint(0, i - 1, int(num_salt))
              for i in image.shape]
      out[coords] = 255

      # Pepper mode
      num_pepper = np.ceil(amount_sp* image.size * (pprop))
      coords = [np.random.randint(0, i - 1, int(num_pepper))
              for i in image.shape]
      out[coords] = 0
      return out
```

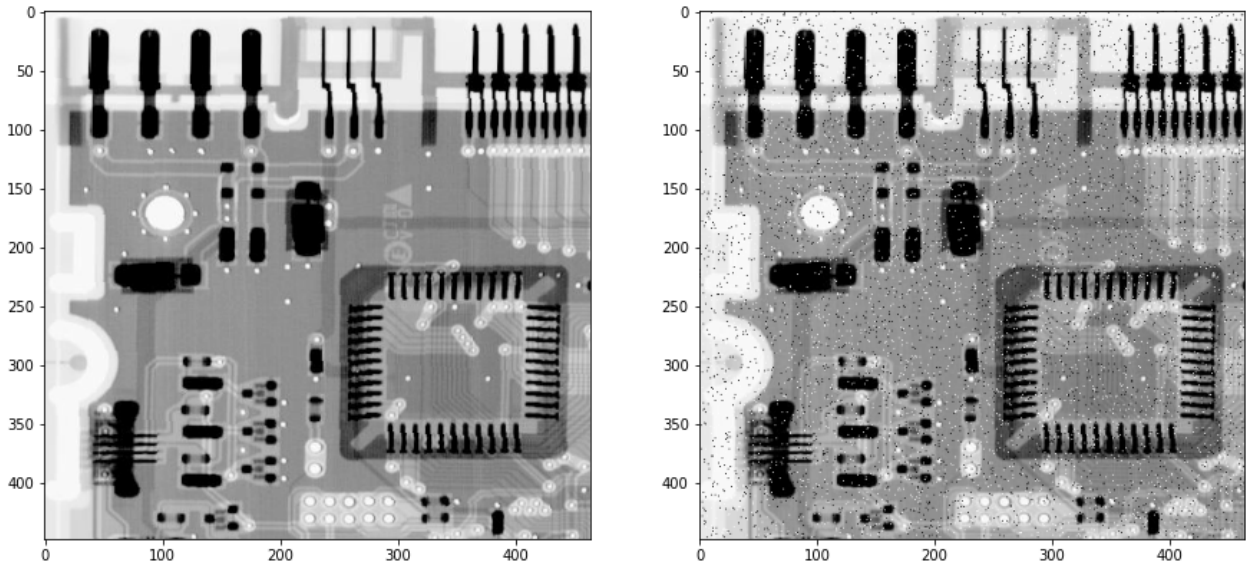




PROJECT 05-02

Noise Reduction Using a Median Filter

First, like in the previous project we have just added noise to the original image.



Then we want to use the median filter for noise reduction.

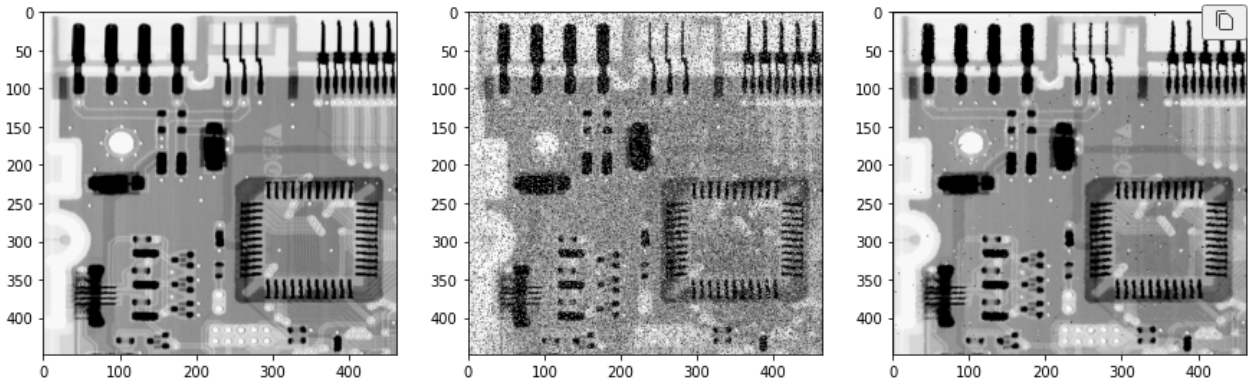


Fig - the left one is the original image, the middle is noise added image and
The right one is a denoised image with median filtering.

```
img  = plt.imread('./Fig0507(a)(ckt-board-orig).tif')

img_noisy = noisy('s&p', img, mean=10, sigma=20,
                  amount_sp=0.6, sprob=0.2,
                  pprop=0.2)

newimg = np.zeros((img_noisy.shape[0], img_noisy.shape[1]))

for i in range(2,img_noisy.shape[0]):
    for j in range(2,img_noisy.shape[1]):
        temp = []
        try:
            temp.append(img_noisy[i-1:i+2, j-1:j+2].ravel())
        except:continue
        temp.sort()
        newimg[i][j] = np.median(temp)
```

PROJECT 05-03

Periodic Noise Reduction Using a Notch Filter

Periodic noise can be reduced significantly via frequency domain filtering. On this page we use a notch reject filter with an appropriate radius to completely enclose the noise spikes in the Fourier domain. The notch filter rejects frequencies in predefined neighborhoods around a center frequency. The number of notch filters is arbitrary. The shape of the notch areas can also be arbitrary (e.g. rectangular or circular). On this page we use three circular shape notch reject filters. Power spectrum density of an image is used for the noise spike's visual detection.

```
def add_sine_noise(img, A,u0,v0):
    width, height = img.shape
    noisy_img = img.copy()
    for i in range(width):
        for j in range(height):
            noisy_img[i][j] += A * np.sin(v0 * i + u0*j )
    return noisy_img
```

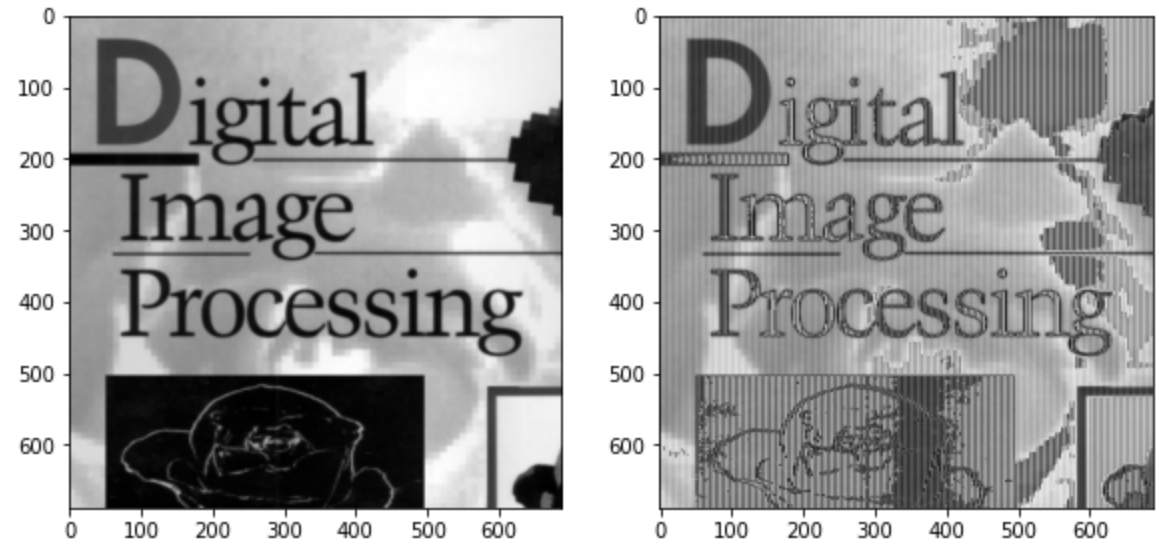


Fig- Added sinusoidal noise to the image

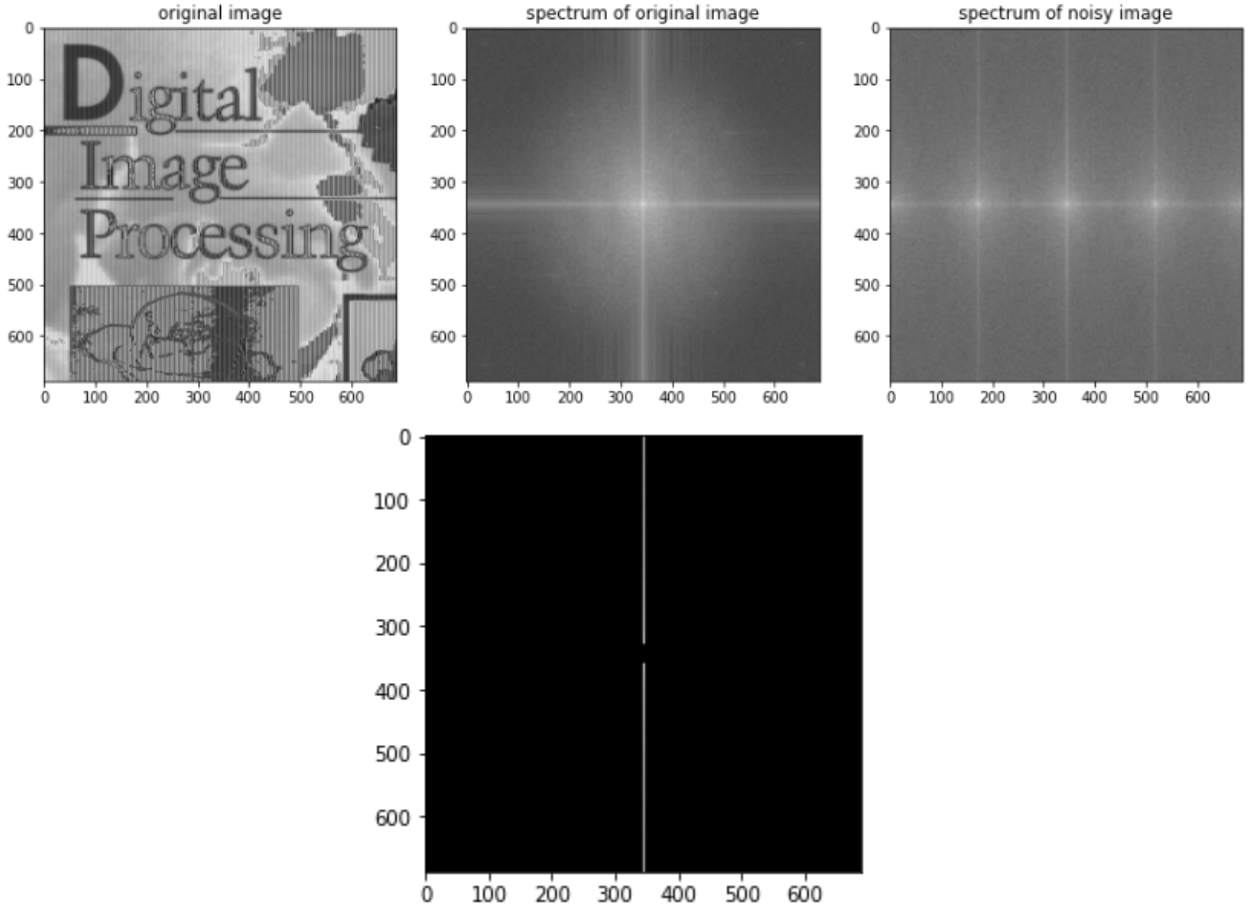


Fig - the notch filter

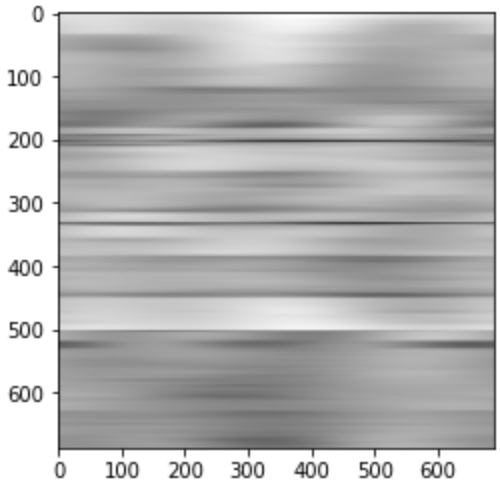


Fig - the result of the output

PROJECT 05-04

Parametric Wiener Filter

If the motion variables x0(t) and y0(t) are known, the transfer function H(u, v) can be obtained directly from Eq. (5.6-8). As an illustration, suppose that the image in question undergoes uniform linear motion in the x-direction only, at a rate given by x0(t) = at>T. When t = T, the image has been displaced by a total distance a. With y0(t) = 0

With this function

$$H(u,v) = \frac{T}{\pi(ua + vb)} \sin[\pi(ua + vb)]e^{-j\pi(ua+vb)}$$

We can multiply this in the DFT of image and then take the inverse fourier transform.

```
img  = cv2.imread('./Fig0526(a)(original_DIP).tif',0)
# dft = np.fft.fft2(img)
dft = cv2.dft(np.float32(img), flags=cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
dft = (dft_shift[:,:,0]*1+1j*dft_shift[:,:,1])
# H = np.fft.fftshift(H)
fshift = H*dft
fshift = np.stack([np.real(fshift), np.imag(fshift)], axis=-1)
f_ishift= np.fft.ifftshift(fshift)

img_back = cv2.idft(f_ishift,flags=cv2.DFT_COMPLEX_OUTPUT)
# img_back = np.fft.ifft2(f_ishift)
# img_back = np.abs(img_back)
img_back = cv2.magnitude(img_back[:,:,0], img_back[:,:,1])
plt.imshow(img_back, cmap='gray')
```

```
T = 1
a = .1
b= .1
width, height = img.shape
H = np.zeros((width, height), dtype='complex')
H[0][0]=1 + 0j
for u in range(1,width):
    for v in range(1,height):
        m = (T/ ((np.pi)*(u*a+v*b)))*\
                (np.sin(np.pi*(u*a+v*b)))*(np.exp((-1j*np.pi*(u*a+v*b))))
        H[u][v] = m
```

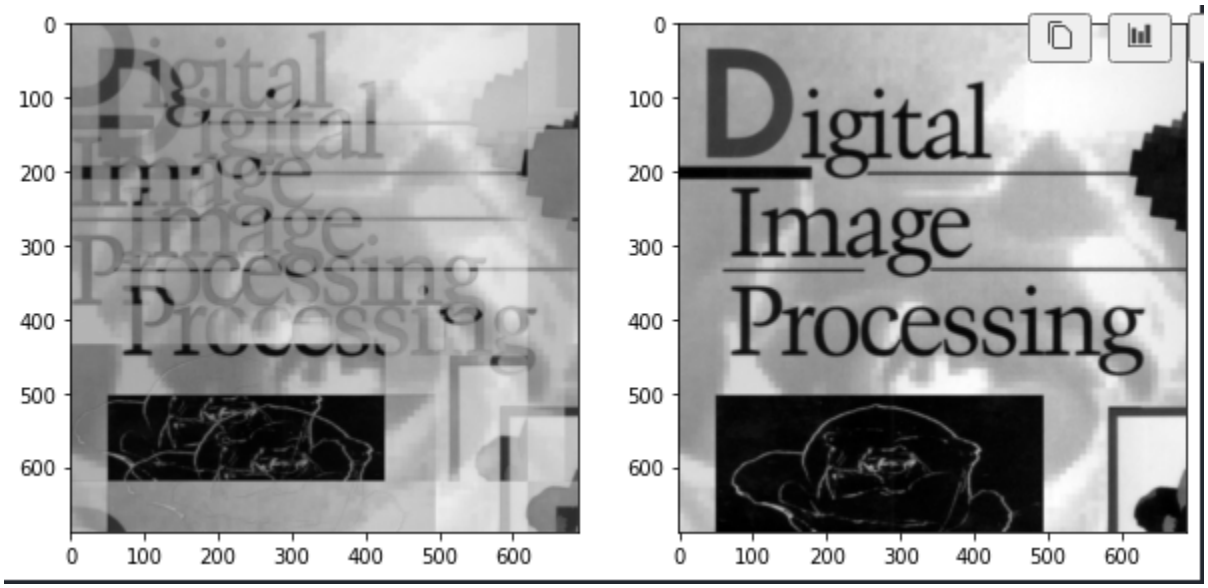
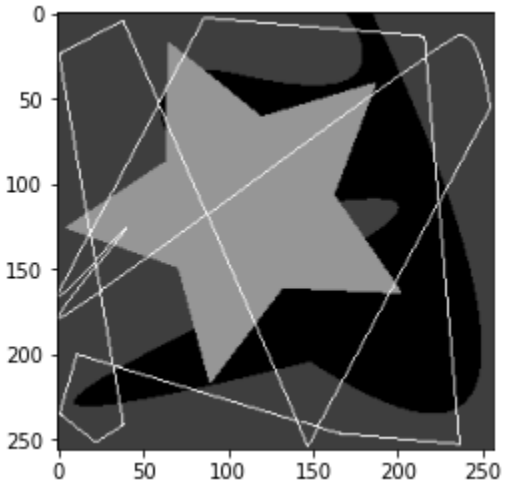


Fig - The final result implementing motion with a=0.1 and b=0.1

PROJECT 08-01

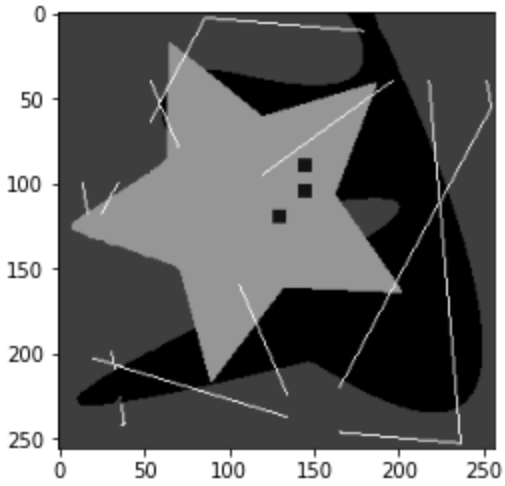
Objective Fidelity Criteria

The original image is:



The transformed this image to the goal I wrote following codes for making the goal:

```
img_2=median_filter(img, center=[50,27],
                    width=50, height=27, kernel=7)
img_2 = median_filter(img_2, center=[159,30],
                    width=40, height=30, kernel=7)
img_2 = median_filter(img_2, center=[120,90],
                    width=40, height=30, kernel=7)
img_2 = median_filter(img_2, center=[230,10],
                    width=40, height=10, kernel=7)
img_2 = median_filter(img_2, center=[250,25],
                    width=40, height=12, kernel=7)
img_2 = median_filter(img_2, center=[240,150],
                    width=40, height=15, kernel=7)
img_2 = median_filter(img_2, center=[20,220],
                    width=20, height=40, kernel=7)
img_2 = making_square(img_2, [120, 130],4,4)
img_2 = making_square(img_2, [90, 145],4,4)
img_2 = making_square(img_2, [105, 145],4,4)
```



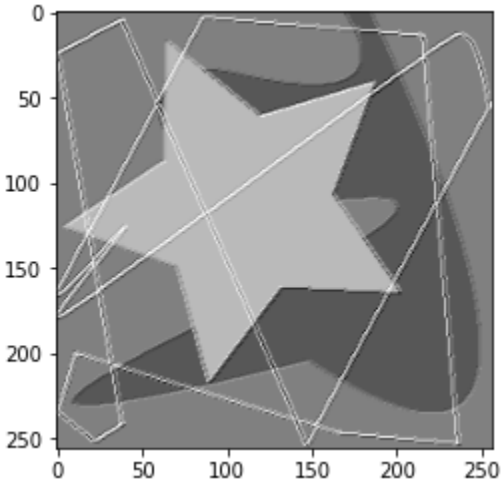

```
✓ findingErrors(img_2,img) ···

(1.1139115540272093, 36.77014646383905)
```

Fig- the first goal

```python
def findingErrors(compressed, decompressed):
    size = compressed.size
    diff = compressed - decompressed
    rmse = np.sqrt(np.square(diff).sum()/size)
    SNRms = np.square(compressed).sum()/np.square(compressed-decompressed).sum()
    return rmse, SNRms
```

This program computes the root-mean-square error and mean-square signal-to-noise ratio of a compressed- decompressed image.(RMSE , SNRms)



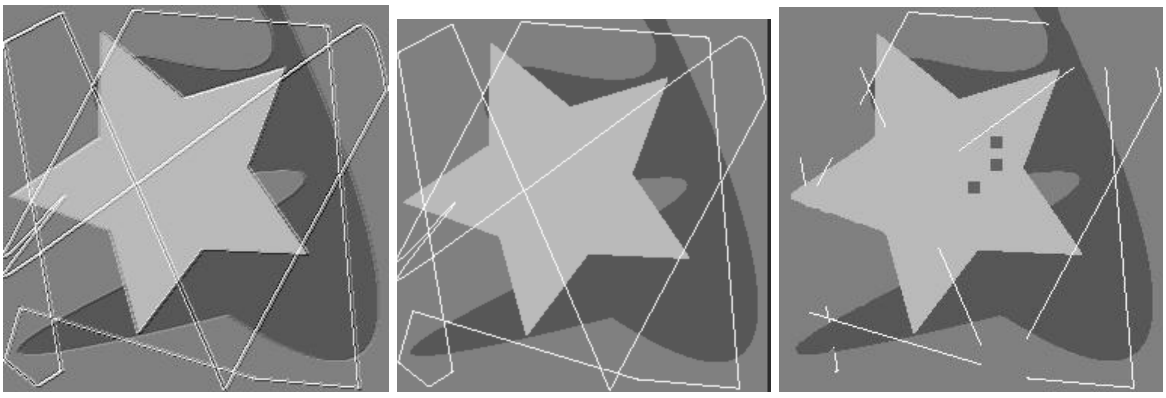
The second image

```
✓ findingErrors(img_3,img) ···
(3.41640812585702, 4.148499331961523)
```

We see that although the second image is much similar to the original image, when a transform or loss has been added to many pixels the error goes too large!
But when some information has been omitted from the original image the errors go much less than the former one.

PROJECT 08-02
Image Entropy

```
✓ for image in images: ···
1.6613969600735903
5.129396716200752
3.7975430635024825
```



The images for finding the entropy

```python
def entropycalc(img):
    marg = np.histogramdd(np.ravel(img), bins = 256)[0]/img.size
    marg = list(filter(lambda p: p > 0, np.ravel(marg)))
    entropy = -np.sum(np.multiply(marg, np.log2(marg)))
    return entropy

img  = plt.imread('./Fig0801(a).tif')
img_b = plt.imread('./img_b.jpeg')
img_c = plt.imread('./img_c.jpeg')
images= [img, img_b, img_c]
Run Cell | Run Above | Debug Cell | Go to [2]
# %%
for image in images:
    print(entropycalc(image))
```