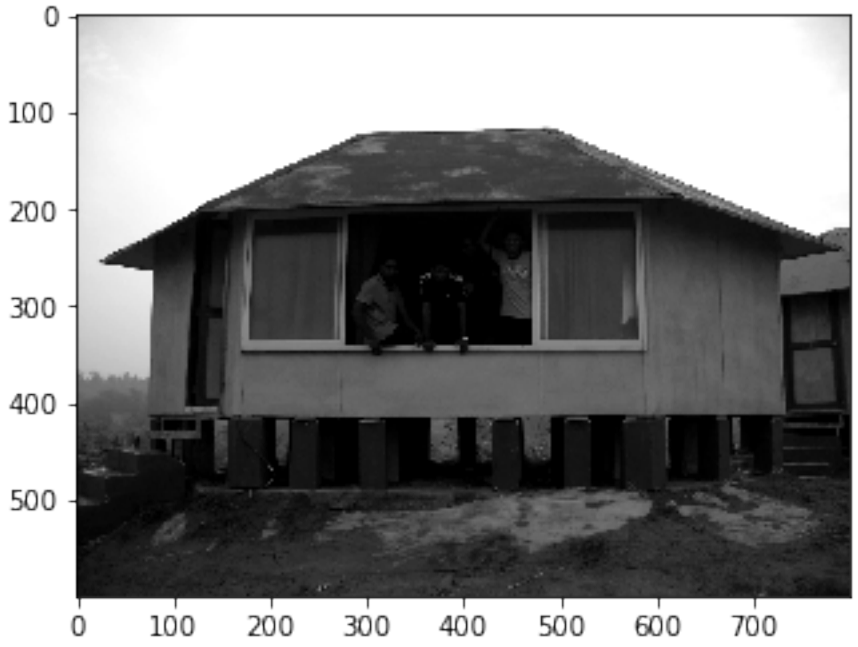
HW2.1

November 4, 2019

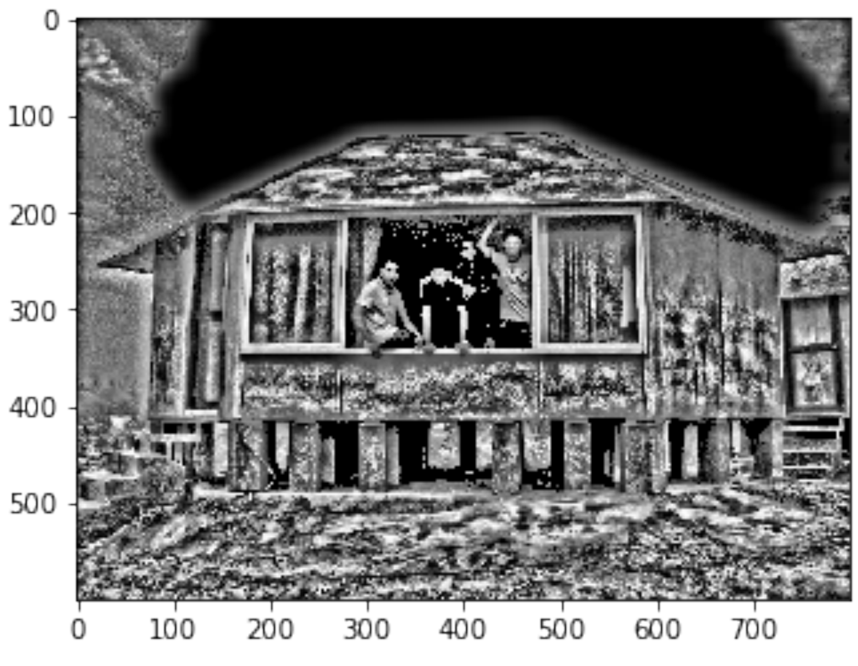
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

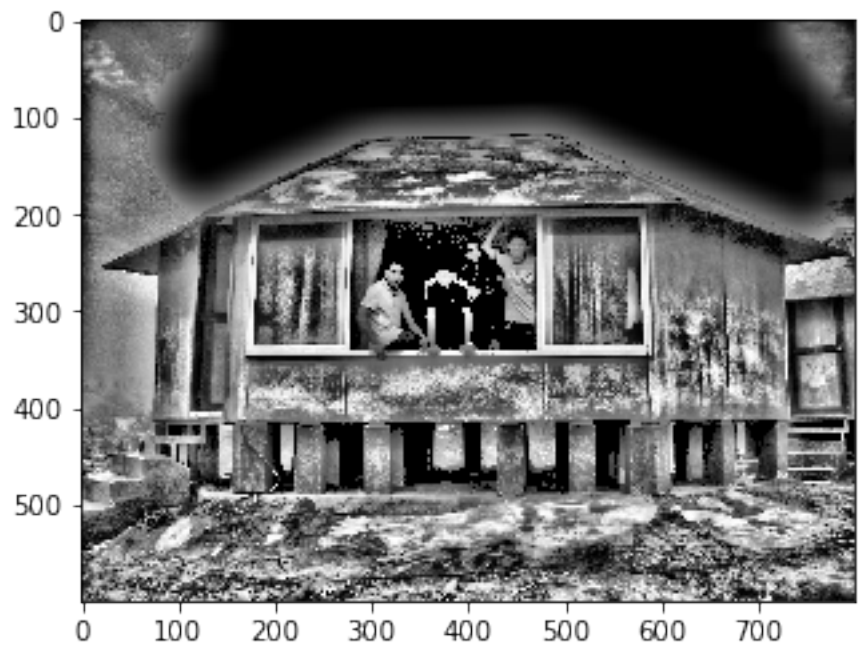
[2]:
```
# def AHE(img, win_size):
win_size = 33
img = plt.imread("data/beach.png")
plt.imshow(img, cmap = 'gray')
plt.savefig("p1_original_image.jpg")
# plt.show()
```

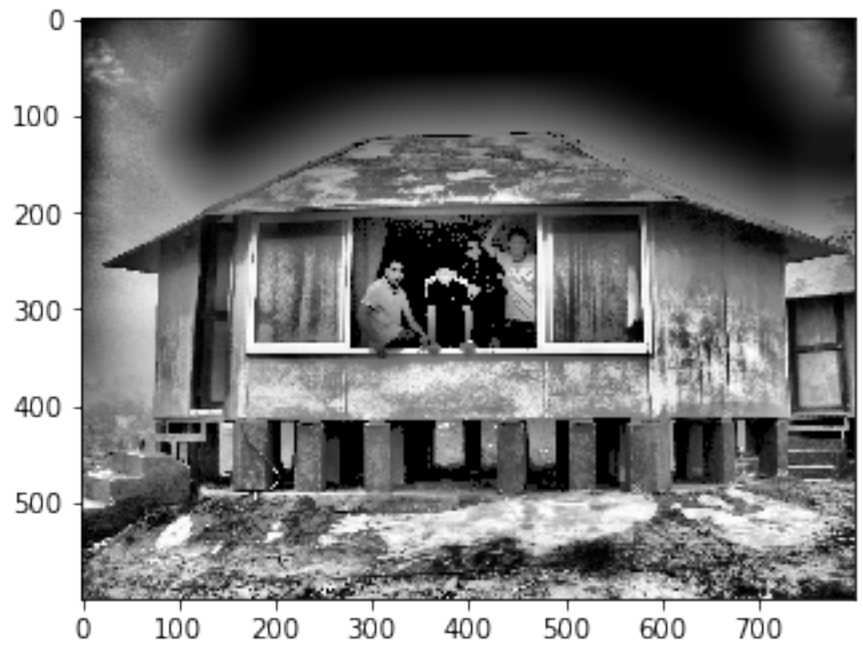


[3]:
```
def AHE(img, win_size):
    height = img.shape[1]
    width = img.shape[0]

    pad_w = int(win_size / 2)
    pad_h = win_size - pad_w - 1
    img_pad = np.pad(img, (pad_w, pad_h), mode = 'symmetric')

    output = np.zeros((width, height))
    for x in range(0, width):
        for y in range(0, height):
            rank = 0
            pixel = img_pad[x + pad_w][y + pad_h]
            context_reg = img_pad[x : x + win_size, y : y + win_size]
            rank = np.sum(context_reg < pixel)
            output[x][y] = int(rank * 255 / win_size ** 2)

    return output
```

[4]:
```
win_sizes = [33, 65, 129]
i = 1

for win_size in win_sizes:
    output = AHE(img, win_size)

    plt.imshow(output, cmap = 'gray')
    plt.savefig("q1_AHE_%s.jpg"%i)
    i += 1
    plt.show()
```



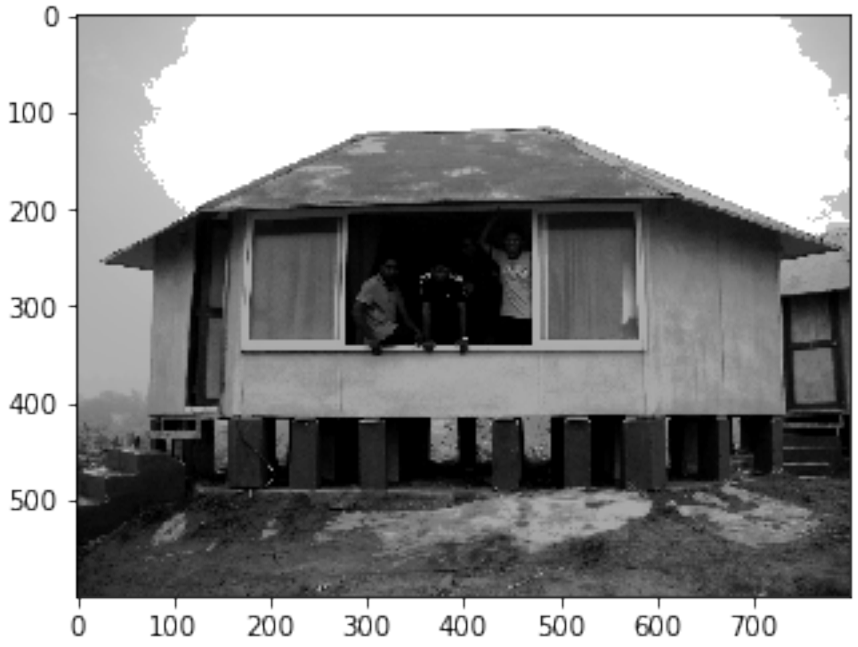




```
[5]: img = cv2.imread("data/beach.png",0)
     equ = cv2.equalizeHist(img)
     print(equ)
     plt.imshow(equ, cmap='gray')
```

```python
plt.savefig("p1_HE.jpg")
```

```
[[174 174 174 … 175 175 175]
 [174 174 174 … 176 176 176]
 [174 174 174 … 176 176 176]
 …
 [ 31  34  31 …  24  29  20]
 [ 27  29  34 …  22  18  18]
 [ 29  31  29 …  16  13  15]]
```



1.How does the original image qualitatively compare to the images after AHE and HE respectively?

The image after AHE and HE shows improved contrast.

2.Which strategy (AHE or HE) works best for beach.png and why? Is this true for any image in general?

AHE shows better-improved contrast. As the histograms the adaptive method computes correspond to a distinct section of the image, and is used to redistribute the lightness values of the image. Therefore it will show better performance on improving the local contrast and enhancing the definitions of edges in each region of an image.

However it is not true for any image in general as AHE has a tendency to overamplify noise in relatively homogeneous regions of an image. This drawback can also be shown in the AHE result of our image.

HW2.2

November 4, 2019

1 Problem 2

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from skimage.morphology import opening, closing, dilation, erosion
from skimage.morphology import disk, rectangle
import pandas
from scipy.ndimage import label
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

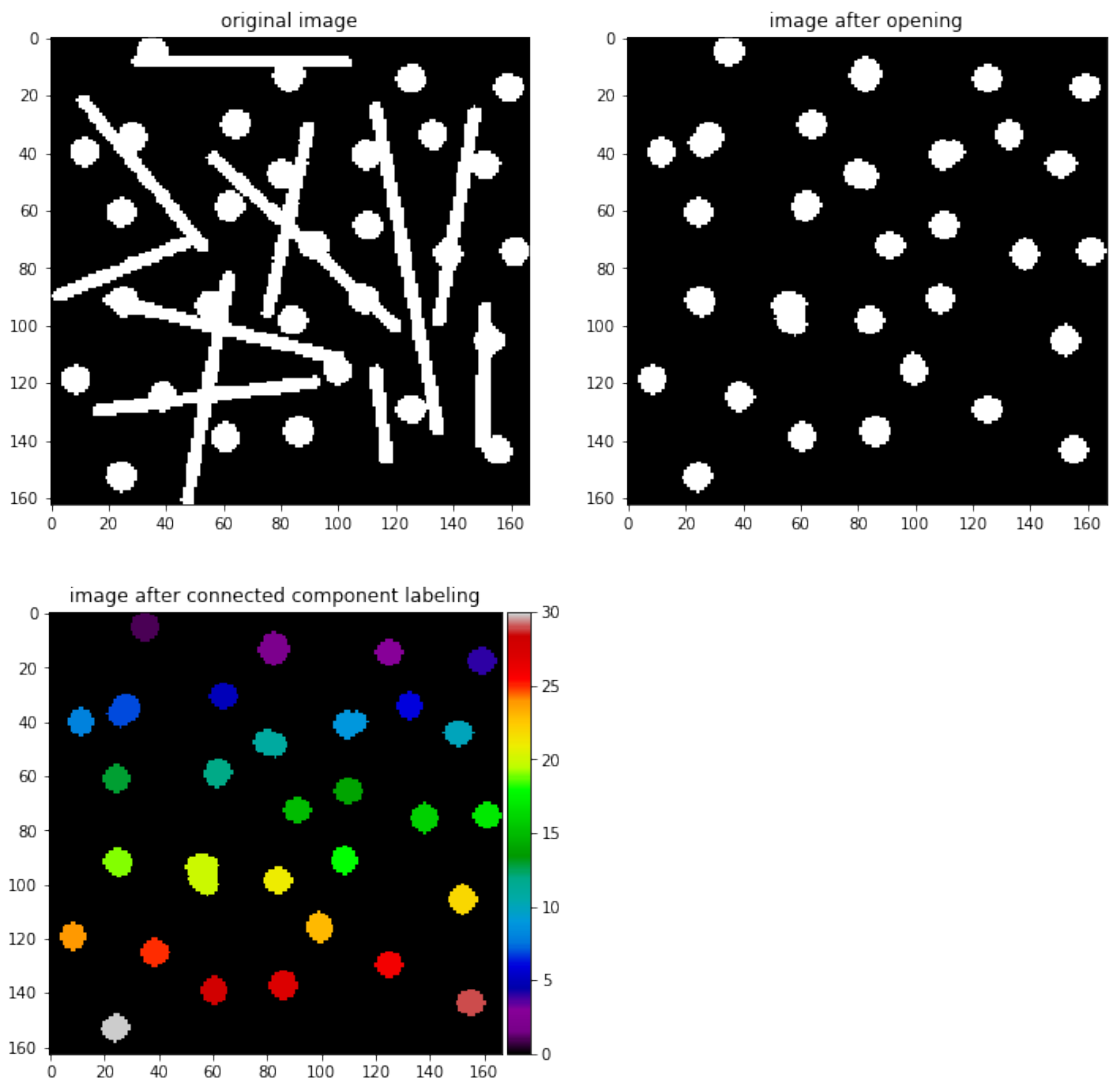
1.0.1 Part(i)

[2]:
```
img = cv2.imread("data/circles_lines.jpg", 0)
img[img <= 128] = 0
img[img > 128] = 1
```

[3]:
```
img_new = opening(img, disk(4))
labeled_array, num_features = label(img_new)
print(num_features)
```

```
30
```

[4]:
```
f1 = plt.figure(figsize=(12,12))
f1_ax1 = f1.add_subplot(221)
f1_ax2 = f1.add_subplot(222)
f1_ax3 = f1.add_subplot(223)
f1_ax1.imshow(img, cmap='gray')
f1_ax1.title.set_text("original image")
f1_ax2.imshow(img_new, cmap='gray')
f1_ax2.title.set_text("image after opening")
img3 = f1_ax3.imshow(labeled_array, cmap='nipy_spectral')
f1_ax3.title.set_text("image after connected component labeling")
divider = make_axes_locatable(f1_ax3)
cax = divider.append_axes('right', size='5%', pad=0.05)
plt.colorbar(img3, cax, orientation='vertical')
plt.savefig("p2_circles_lines.jpg")
plt.show()

data = []
for i in range(num_features):
    position = np.where(labeled_array == i + 1)
    area = len(position[0])
    x_cen = round(sum(position[0]) / area, 2)
    y_cen = round(sum(position[1]) / area, 2)
    data.append([area, (round(x_cen,2), round(y_cen,2))])

title = ["area", "centroid"]
circle = list(np.arange(1, 31, 1))
print(pandas.DataFrame(data, circle, title))
```



```
    area        centroid
1     89      (5.0, 35.0)
2    106   (13.04, 82.59)
3     78    (14.5, 125.0)
4     78    (17.5, 159.0)
5     78     (30.5, 64.0)
6     78    (34.0, 132.5)
7    109   (35.75, 27.25)
8     78     (40.0, 11.5)
9     97   (40.7, 110.57)
10    80   (44.22, 150.5)
11    97    (47.9, 80.99)
12    85   (58.85, 61.85)
13    78     (61.0, 24.5)
14    78    (65.5, 110.0)
15    78     (72.5, 91.0)
16    84    (75.5, 138.0)
17    78    (74.5, 161.0)
18    78    (91.0, 108.5)
19    85   (91.85, 25.15)
20   148   (95.86, 56.32)
21    78     (98.5, 84.0)
22    84   (105.5, 152.0)
23    84  (115.68, 99.37)
24    78     (119.0, 8.5)
25    81   (125.0, 38.68)
26    78   (129.5, 125.0)
27    89    (137.0, 86.0)
28    78    (139.0, 60.5)
29    78   (143.5, 155.0)
30    77  (152.83, 24.17)
```

i) The original image, the image after opening, the image after connected component labeling (plot with colorbar), and a table with the desired values for each component of part(i) is shown above.

ii) The structure used for the opening operation is disk and the size of it is 4.

iii) codes shown above

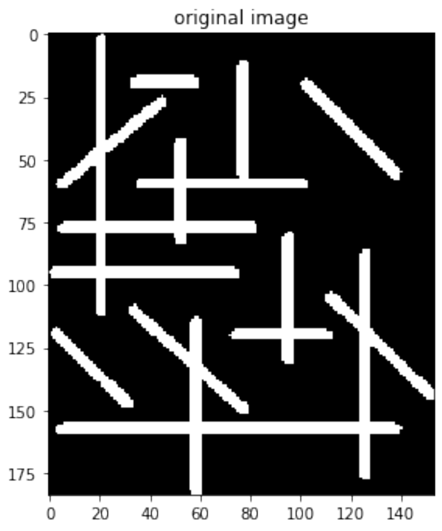
1.0.2 Part(ii)

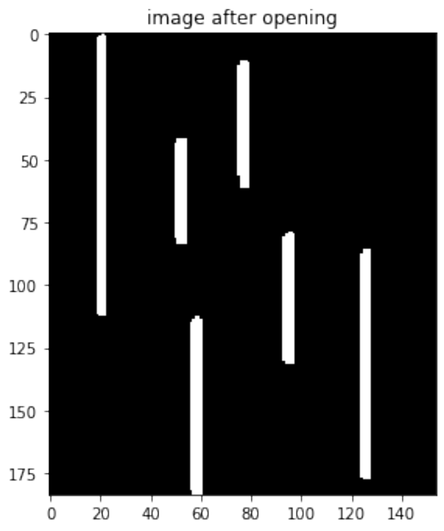
[5]:
```
img_l = cv2.imread("data/lines.jpg", 0)
img_l[img_l <= 128] = 0
img_l[img_l > 128] = 1
```

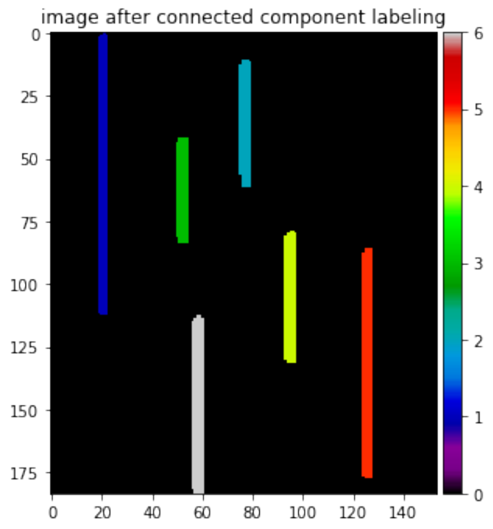
[6]:
```
img_nl = opening(img_l, rectangle(8,1))
labeled_array_l, num_features_l = label(img_nl)
print(num_features_l)
```

```
6
```

[7]:
```
f2 = plt.figure(figsize=(14,12))
f2_ax1 = f2.add_subplot(221)
f2_ax2 = f2.add_subplot(222)
f2_ax3 = f2.add_subplot(223)
f2_ax1.imshow(img_l, cmap='gray')
f2_ax1.title.set_text("original image")
f2_ax2.imshow(img_nl, cmap='gray')
f2_ax2.title.set_text("image after opening")
imgl_3 = f2_ax3.imshow(labeled_array_l, cmap='nipy_spectral')
f2_ax3.title.set_text("image after connected component labeling")
divider = make_axes_locatable(f2_ax3)
cax = divider.append_axes('right', size='5%', pad=0.05)
plt.colorbar(imgl_3, cax, orientation='vertical')
plt.savefig("p2_lines.jpg")
plt.show()

data_l = []
for i in range(num_features_l):
    position = np.where(labeled_array_l == i + 1)
    x_len = max(position[0]) - min(position[0])
    y_len = max(position[1]) - min(position[1])
    length = np.sqrt(x_len**2+y_len**2)
    x_cen = sum(position[0]) / len(position[0])
    y_cen = sum(position[1]) / len(position[0])
    data_l.append([round(length, 2), (round(x_cen,2), round(y_cen,2))])

title = ["length", "centroid"]
circle = list(np.arange(1, 7, 1))
print(pandas.DataFrame(data_l, circle, title))
```







```
   length         centroid
1  112.04   (56.37, 20.51)
2   50.16   (35.83, 77.05)
3   41.19    (62.5, 52.04)
4   52.15   (105.3, 95.02)
5   91.05 (131.62, 125.51)
6   70.11   (148.2, 58.02)
```

i) The original image, the image after opening, the image after connected component labeling (plot with colorbar), and a table with the desired values for each component of part(ii) is shown above.

ii) The structure used for the opening operation is rectangle and the size of it is width 1 with height 8.

iii) codes shown above

HW2.3

November 8, 2019

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from lloyds import lloyds
import glob
```

[2]:
```
# imgs = cv2.imread("data/*.tif", 0)
path = "data/*.tif"
imgs = []
for file in glob.glob(path):
    imgs.append(cv2.imread(file, 0))
plt.imshow(imgs[1], cmap = 'gray')
plt.show()
```

[3]:
```
def bit_change(img, ori, out):
    change = 2 ** (ori - out)
    img_new = img // change * change
    return img_new
```

(i) The function quantilize the 8-bit image to s-bit image is shown as above.

[4]:
```
def lloyds_img(img, ori, out):
    [M, N] = img.shape;
    img = img.astype(float)
    img_new_l = np.zeros(img.shape)
    training_set = np.reshape(img,N*M,1);
    partition, codebook = lloyds(training_set, [2**out])
    for i in range(M):
        for j in range(N):
            if (img[i][j] < partition[0]):
                img_new_l[i][j] = codebook[0];
            else:
                curdist = [img[i][j] - p for p in partition]
                curmin = min(i for i in curdist if i >= 0)
                minIndex = curdist.index(curmin)
                img_new_l[i][j] = codebook[minIndex + 1]
    return img_new_l
```

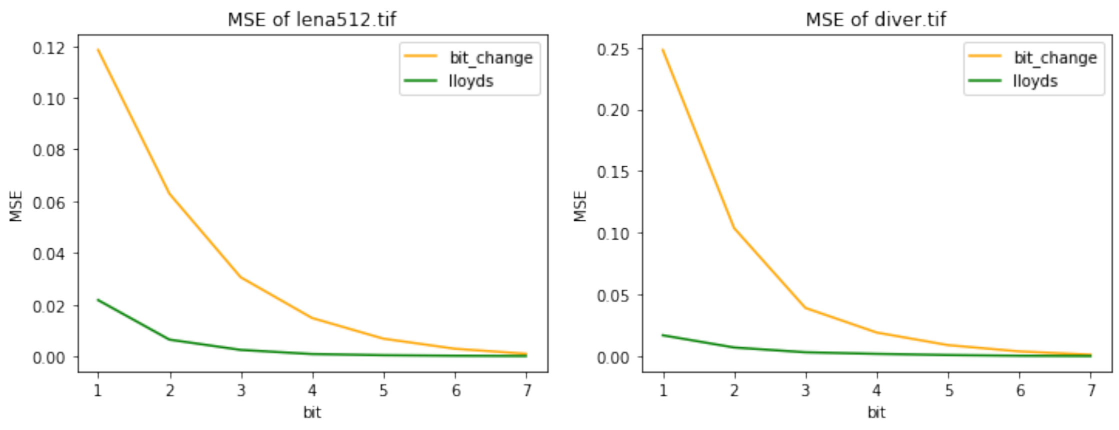
[5]:
```
def MSE(img, img_new):
    size = img.shape[0] * img.shape[1]
    return np.linalg.norm(img_new.astype(float) - img.astype(float), 2)  / size
```

[6]:
```
def mses_compute(img):
#     plt.imshow(img, cmap = 'gray')
#     plt.show()
    ss = np.arange(1, 8, 1)
    mses_bit_change = []
    mses_lloyds = []

    for s in ss:
        img_new_bit_change = bit_change(img, 8, s)
        mse_bit_change = MSE(img, img_new_bit_change)
        mses_bit_change.append(mse_bit_change)

        img_new_l = lloyds_img(img, 8, s)
        mse_lloyds = MSE(img, img_new_l)
        mses_lloyds.append(mse_lloyds)

#         f = plt.figure(figsize=(8,4))
#         ax1 = f.add_subplot(1,2,1)
#         ax1.imshow(img_new_bit_change, cmap = 'gray')
#         ax2 = f.add_subplot(1,2,2)
#         ax2.imshow(img_new_l, cmap = 'gray')
#         plt.show()
```

```
    return mses_bit_change, mses_lloyds
```

[7]:
```
msess_bit_change = []
msess_lloyds = []

for img in imgs:
    mses_bit_change, mses_lloyds = mses_compute(img)
    msess_bit_change.append(mses_bit_change)
    msess_lloyds.append(mses_lloyds)
```

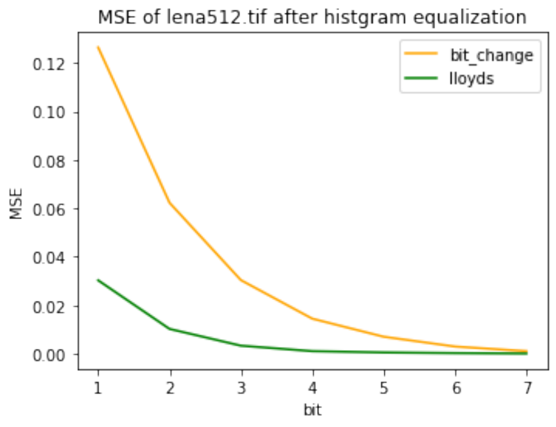
[8]:
```
f = plt.figure(figsize = (12, 4))
ss = np.arange(1, 8, 1)

ax1 = f.add_subplot(121)
ax1.title.set_text("MSE of lena512.tif")
ax1.plot(ss, msess_bit_change[0], color='orange', label='bit_change')
ax1.plot(ss, msess_lloyds[0], color='green', label='lloyds')
ax1.legend() #
ax1.set_xlabel('bit')
ax1.set_ylabel('MSE')

ax2 = f.add_subplot(122)
ax2.title.set_text("MSE of diver.tif")
ax2.plot(ss, msess_bit_change[1], color='orange', label='bit_change')
ax2.plot(ss, msess_lloyds[1], color='green', label='lloyds')
ax2.legend() #
ax2.set_xlabel('bit')
ax2.set_ylabel('MSE')
plt.savefig("p3_noHE.jpg")

plt.show()
```

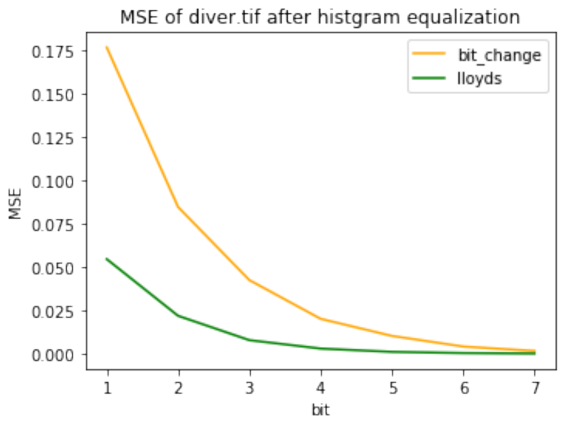


(ii) The results of lena512.tif and diver.tif are shown above which shows that the lloyd-max quantizer outperform the bit-change method as the lloyd-max quantizer will get much smaller MSE than bit-change quantizer and the lloyd-max quantizer's performace gap is much smaller than the bit-change method. The reason why lloyd-max quantizer is outperformed than bit-change method is that it will consider the detail situation of this whole image and decide the specific partition and codebook value for this specific image to be quantized. However, the bit-change method will not consider the specific situation for this image and will simply downsampled the value. So as the bit become smaller, the MSE for the bit-change method will become larger.

[9]:
```
imgs_histequs = []
for img in imgs:
    imgs_histequs.append(cv2.equalizeHist(img))

msess_histequ_bit_change = []
msess_histequ_lloyds = []

for img in imgs_histequs:
    mses_bit_change, mses_lloyds = mses_compute(img)
    msess_histequ_bit_change.append(mses_bit_change)
    msess_histequ_lloyds.append(mses_lloyds)
```

[10]:
```
f = plt.figure(figsize = (12, 4))
ss = np.arange(1, 8, 1)

ax1 = f.add_subplot(121)
ax1.title.set_text("MSE of lena512.tif after histgram equalization")
ax1.plot(ss, msess_histequ_bit_change[0], color='orange', label='bit_change')
ax1.plot(ss, msess_histequ_lloyds[0], color='green', label='lloyds')
ax1.legend() #
ax1.set_xlabel('bit')
ax1.set_ylabel('MSE')

ax2 = f.add_subplot(122)
ax2.title.set_text("MSE of diver.tif after histgram equalization")
ax2.plot(ss, msess_histequ_bit_change[1], color='orange', label='bit_change')
ax2.plot(ss, msess_histequ_lloyds[1], color='green', label='lloyds')
ax2.legend() #
ax2.set_xlabel('bit')
ax2.set_ylabel('MSE')
plt.savefig("p3_HE.jpg")

plt.show()
```





(iii) The gap in MSE between the two quantization approaches become smaller. This is because that for when using histogram equalization, the histogram will be flattened. And thus the MSE between this two method will be smaller as the MSE of lloyds-max method will be larger than it used to be.

(iv) Though the equalization will flatten the distribution, the differece between all the pixels will become larger and thus will make the lloyds-max quantizer performs well for 7-bit situation.