Problem 1. Adaptive Histogram Equalization (10 points)

It is often found in image processing and related fields that real world data is unsuitable for direct use. This warrants the inclusion of pre-processing steps before any other operations are performed. An example of this is *histogram equalization* (HE) and its extension *adaptive histogram equalization* (AHE).

The goal of this problem is to implement a function for AHE as described in Chapter 1 of *Adaptive Histogram Equalization - A Parallel Implementation*[1]. The function has the following specifications:

(i) The desired function $AHE()$ takes two inputs: the image im and the contextual region size win_size.

(ii) Using the pseudocode in Algorithm 1 as a reference, compute the enhanced image after AHE.

(iii) You may use loops if necessary. You should not make use of any inbuilt Python functions for AHE or HE.

(iv) The function returns one output: the enhanced image after AHE.

Evaluate your function on the image *beach.png* for $win\_size = 33,\ 65,$ and 129. In your report, include the original image, the 3 images after AHE, and the image after simple HE (you may use Python inbuilt function for this final HE image only). Make sure to resize all images to ensure they do not take up too much space. Additionally, include your answers (no more than three sentences each) to the following questions:

- How does the original image qualitatively compare to the images after AHE and HE respectively?
- Which strategy (AHE or HE) works best for *beach.png* and why? Is this true for any image in general?

Code:

```python
import numpy as np
import cv2 as cv

def AHE(im, win_size):
    output = im # Creating an output image variable

    # Padding the input image to allow for the window function
    padded_image = np.pad(im, (win_size//2,), 'symmetric')

    # Iterating though the pixels of the original image (non-padded areas)
    for x in range(win_size//2,len(im)+win_size//2):
        for y in range(win_size//2,len(im[0])+win_size//2):
            rank = 0
            contextual_region = padded_image[x-win_size//2:x+win_size//2, y-win_size//2:y+win_size//2]

            # Iterating through pixels in contextual region (window in padded_image)
            for i in contextual_region:
                for j in i:
                    if im[x-win_size//2,y-win_size//2] > j:
                        rank = rank+1

            # Assigning updated pixel value to the output image variable
            output[x-win_size//2, y-win_size//2] = rank*255/(win_size*win_size)

    return output


im = cv.imread('beach.png')
img = cv.cvtColor(im, cv.COLOR_BGR2GRAY) # Converting beach.png to a greyscale image

cv.imshow('Original image', img)

win_size = 33 # Adjust the window size of AHE using this variable
ahe_img = AHE(img, win_size)
cv.imshow('AHE with win_size = 33', ahe_img)

im = cv.imread('beach.png')
img = cv.cvtColor(im, cv.COLOR_BGR2GRAY) # Converting beach.png to a greyscale image

# Computing histogram equalization using an inbuilt python function from OpenCV
he_img = cv.equalizeHist(img)
cv.imshow('Image after Histogram Equalization', he_img)

cv.waitKey(0)
cv.destroyAllWindows()
```

Note: Make sure to use the equalizeHist() function separately from the AHE function, because output is initialized to the input image

Original Image:

Image after AHE with win_size = 33:

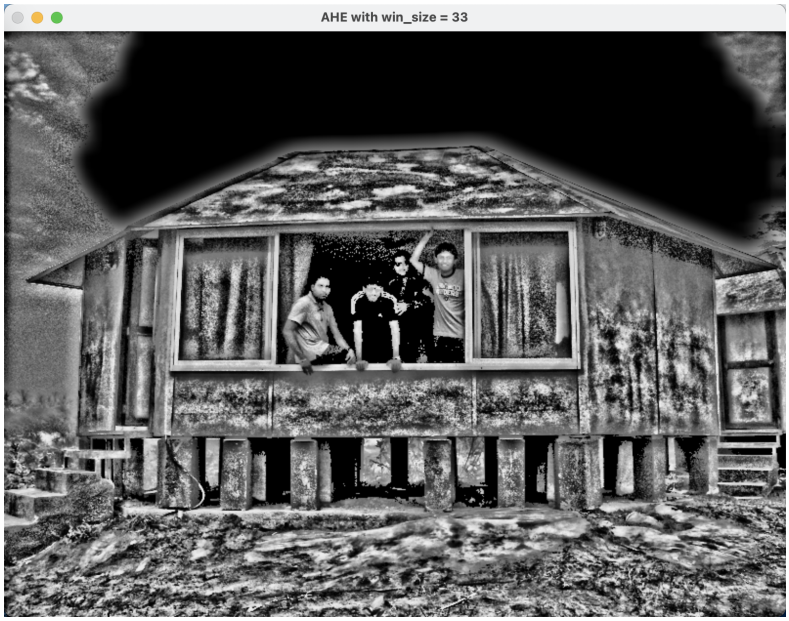
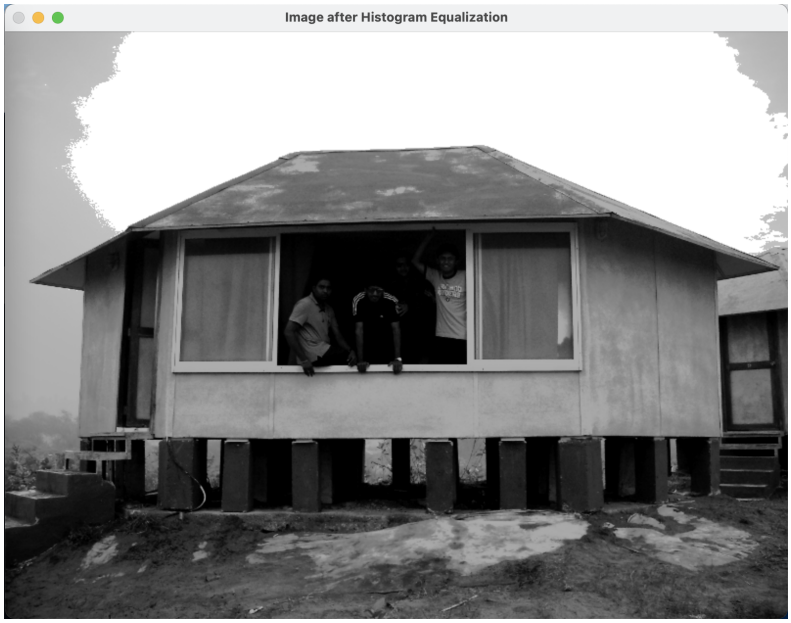


Image after AHE with win_size = 65:

Image after AHE with win_size = 129:

Image after Histogram Equalization:



- Compared to the images after AHE and HE, the original image does not have as much contrast in general. Additionally, the people inside the cabin are underexposed in the image, so they are not as visible compared to after the image has been equalized.

- The AHE strategy works best for beach.png because the subjects are significantly easier to distinguish compared to the original image and the image after HE. The details such as the curtains behind the windows and the texture of the ground are also more visible after AHE, and the image becomes smoother for bigger window sizes when applying AHE. No, it is not true that AHE would work better than HE for any image in general.

Problem 2. Binary Morphology (10 points)

In this problem, you will be separating out various objects in binary images using morphology, followed by connected component analysis to gather more information about objects of interest.

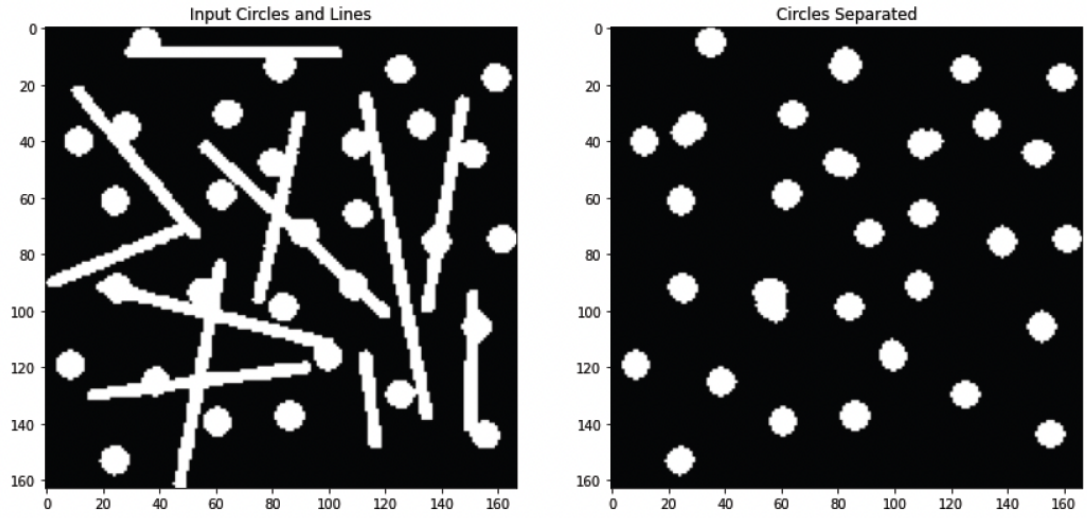
(i) For the binary image *circles_lines.jpg*, your aim is to separate out the circles in the image, and calculate certain attributes corresponding to these circles.

In [ ]:
```
from skimage import morphology
from scipy.ndimage import measurements
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import cv2
```

Part (a) - Remove the lines and label components. There should be 30 circles after opening the image.

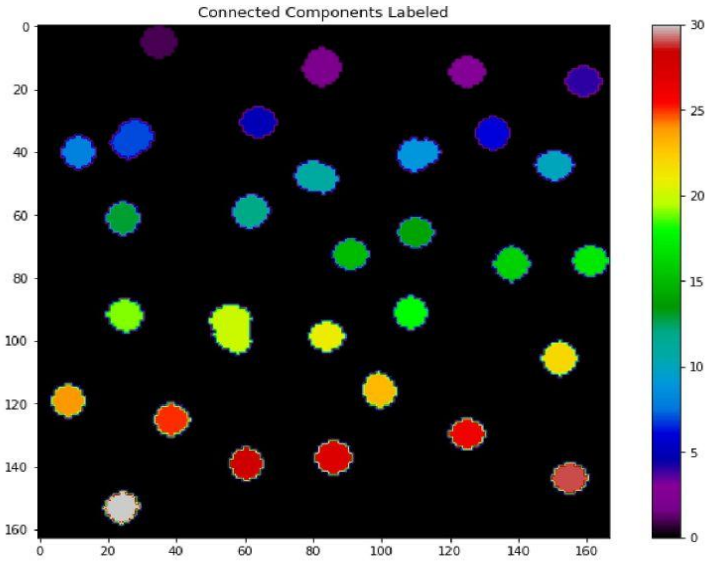
1. Convert image to {0,1} binary using threshold ~ 128
2. Use a disk structuring element with size=4 in the opening step

In [ ]:
```
circles_lines = cv2.imread('circles_lines.jpg')
circles_lines = cv2.cvtColor(circles_lines,cv2.COLOR_BGR2GRAY)
_,circles_lines = cv2.threshold(circles_lines, 128, 1, cv2.THRESH_BINARY)

opened_img = morphology.opening(circles_lines, morphology.disk(4))


f, ax = plt.subplots(1,2, figsize=(14,7))
ax[0].imshow(circles_lines,cmap='gray')
ax[0].set_title('Input Circles and Lines')
ax[1].imshow(opened_img,cmap='gray')
ax[1].set_title('Circles Separated')
plt.show()
```



In [ ]:
```
labeled_img, num_components = measurements.label(opened_img)
print('Number of Circles = ',num_components)
plt.figure(figsize=(12,8))
plt.imshow(labeled_img,cmap='nipy_spectral')
plt.title('Connected Components Labeled')
plt.colorbar()
plt.show()
```

```
Number of Circles =  30
```



In [ ]:

```
header = ['Component ID','row centroid','column centroid','area']
df = pd.DataFrame(columns=header)
for i in range(1,num_components+1):
    comp = np.where(labeled_img == i)
    area = len(comp[1])
    x_ = round(sum(comp[0])/area,0)
    y_ = round(sum(comp[1])/area,0)
    df = df.append({header[0]: i, header[1]:x_, header[2]:y_, header[3]:area}, ignore_index=True)
df
```

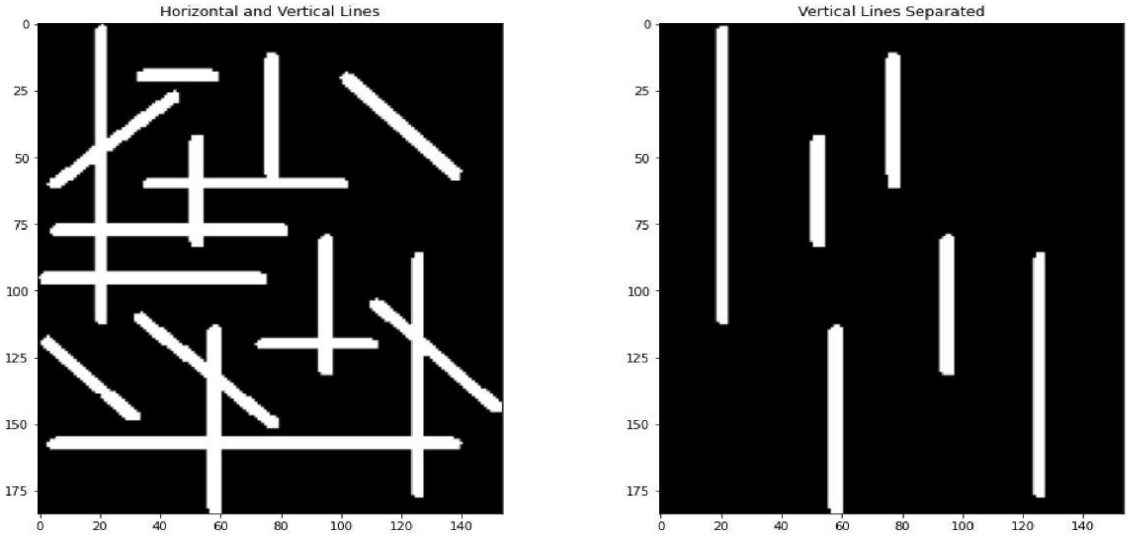
Out[ ]:

	Component ID	row centroid	column centroid	area
0	1.0	5.0	35.0	89.0
1	2.0	13.0	83.0	106.0
2	3.0	14.0	125.0	78.0
3	4.0	18.0	159.0	78.0
4	5.0	30.0	64.0	78.0
5	6.0	34.0	132.0	78.0
6	7.0	36.0	27.0	109.0
7	8.0	40.0	12.0	78.0
8	9.0	41.0	111.0	97.0
9	10.0	44.0	150.0	80.0
10	11.0	48.0	81.0	97.0
11	12.0	59.0	62.0	85.0
12	13.0	61.0	24.0	78.0
13	14.0	66.0	110.0	78.0
14	15.0	72.0	91.0	78.0
15	16.0	76.0	138.0	84.0
16	17.0	74.0	161.0	78.0
17	18.0	91.0	108.0	78.0
18	19.0	92.0	25.0	85.0
19	20.0	96.0	56.0	148.0
20	21.0	98.0	84.0	78.0
21	22.0	106.0	152.0	84.0
22	23.0	116.0	99.0	84.0
23	24.0	119.0	8.0	78.0
24	25.0	125.0	39.0	81.0
25	26.0	130.0	125.0	78.0
26	27.0	137.0	86.0	89.0
27	28.0	139.0	60.0	78.0
28	29.0	144.0	155.0	78.0
29	30.0	153.0	24.0	77.0

(ii) In this part, we are interested in performing similar operations on the binary image *lines.jpg*. Your aim now is to separate out the vertical lines from the horizontal ones in the image, and then calculate certain attributes corresponding to these vertical lines.

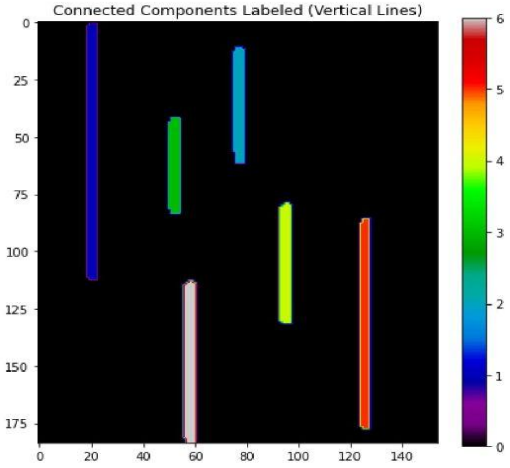
Part (b) - Use the lines.jpg and remove horizontal lines

In [ ]:
```
lines = cv2.imread('lines.jpg')
lines = cv2.cvtColor(lines,cv2.COLOR_BGR2GRAY)
_,lines = cv2.threshold(lines,128,1,cv2.THRESH_BINARY)

opened_img = morphology.opening(lines, morphology.rectangle(15,2))

f, ax = plt.subplots(1,2, figsize=(16,8))
ax[0].imshow(lines,cmap='gray')
ax[0].set_title('Horizontal and Vertical Lines')
ax[1].imshow(opened_img,cmap='gray')
ax[1].set_title('Vertical Lines Separated')
plt.show()
```



In [ ]:
```
labeled_img, num_components = measurements.label(opened_img)
print('Number of Vertical Lines = ',num_components)
plt.figure(figsize=(7,7))
plt.imshow(labeled_img,cmap='nipy_spectral')
plt.title('Connected Components Labeled (Vertical Lines)')
plt.colorbar()
plt.show()
```

```
Number of Vertical Lines =  6
```



In [ ]:
```
header = ['Component ID','row centroid','column centroid','Length']
df = pd.DataFrame(columns=header)
for i in range(1,num_components+1):
    comp = np.where(labeled_img == i)
    area = len(comp[0])
    length = max(abs(max(comp[0])-min(comp[0])),abs(max(comp[1])-min(comp[1])))
    x_ = round(sum(comp[0])/area,0)
    y_ = round(sum(comp[1])/area,0)
```

```
    df = df.append({header[0]: i, header[1]:x_, header[2]:y_, header[3]:length}, ignore_index=True)
df
```

Out[ ]:

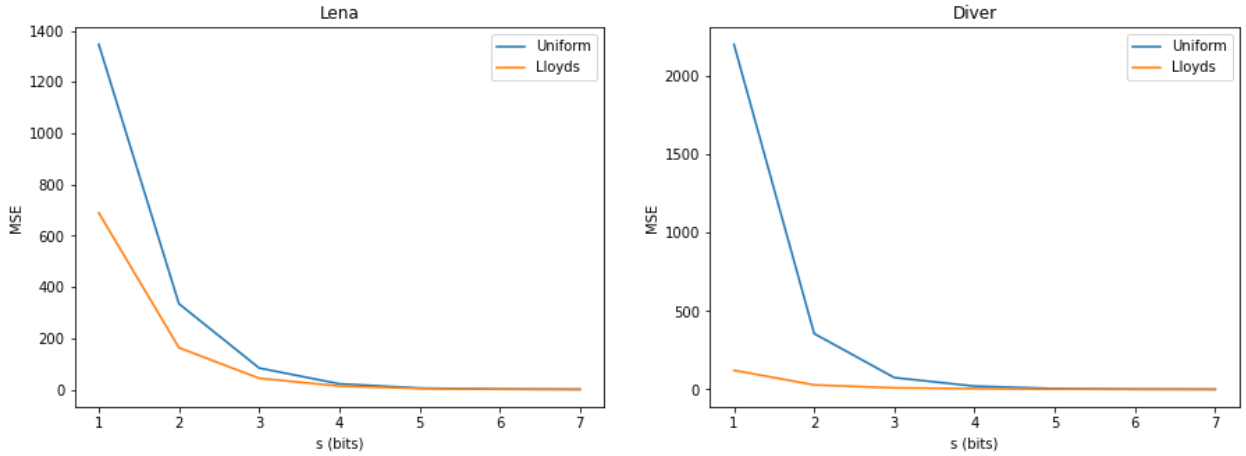
	Component ID	row centroid	column centroid	Length
0	1.0	56.0	21.0	111.0
1	2.0	36.0	77.0	50.0
2	3.0	62.0	52.0	41.0
3	4.0	105.0	95.0	52.0
4	5.0	132.0	126.0	91.0
5	6.0	148.0	58.0	70.0

Structuring Element to be used for part (i) was a disk with size 4 (can vary, but best results with 4) and for part (ii) was a rectangle (size can vary).

Problem 3

```
from lloyd_python import lloyds
import matplotlib.pyplot as plt
import numpy as np
import cv2

def uniformQuantization(image, s):
    bins = 2**s
    output = np.zeros_like(image)
    spacing = 255/bins
    for i in range(bins):
        output[image>=(spacing*i)] = spacing*(i+0.5)
    return output

def MSE(image, qimage):
    return np.mean((image.astype(float)-qimage)**2)

def lloydsMSE(image, partition, codebook):
    output = np.zeros_like(image)
    partition = np.insert(partition, 0, 0)
    for i in range(partition.shape[0]):
        output[image>=partition[i]] = codebook[i]
    return MSE(output, image)

lena = cv2.imread("lena512.tif", cv2.IMREAD_GRAYSCALE)
diver = cv2.imread("diver.tif", cv2.IMREAD_GRAYSCALE)
lena_flat = np.reshape(lena, [np.prod(lena.shape), 1])
diver_flat = np.reshape(diver, [np.prod(diver.shape), 1])

unifMSE_lena = np.zeros((7,1))
unifMSE_diver = np.zeros((7,1))
lloydsMSE_lena = np.zeros((7,1))
lloydsMSE_diver = np.zeros((7,1))

for s in range(1, 8):
    unif_lena = uniformQuantization(lena, s)
    unif_diver = uniformQuantization(diver, s)
    p_lena, cb_lena = lloyds(lena_flat, [2**s])
    p_diver, cb_diver = lloyds(diver_flat, [2**s])

    unifMSE_lena[s-1] = MSE(lena, unif_lena)
    unifMSE_diver[s-1] = MSE(diver, unif_diver)
    lloydsMSE_lena[s-1] = lloydsMSE(lena, p_lena, cb_lena)
    lloydsMSE_diver[s-1] = lloydsMSE(diver, p_diver, cb_diver)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,5))
ax1.plot(np.arange(1,8), unifMSE_lena, label="Uniform")
```

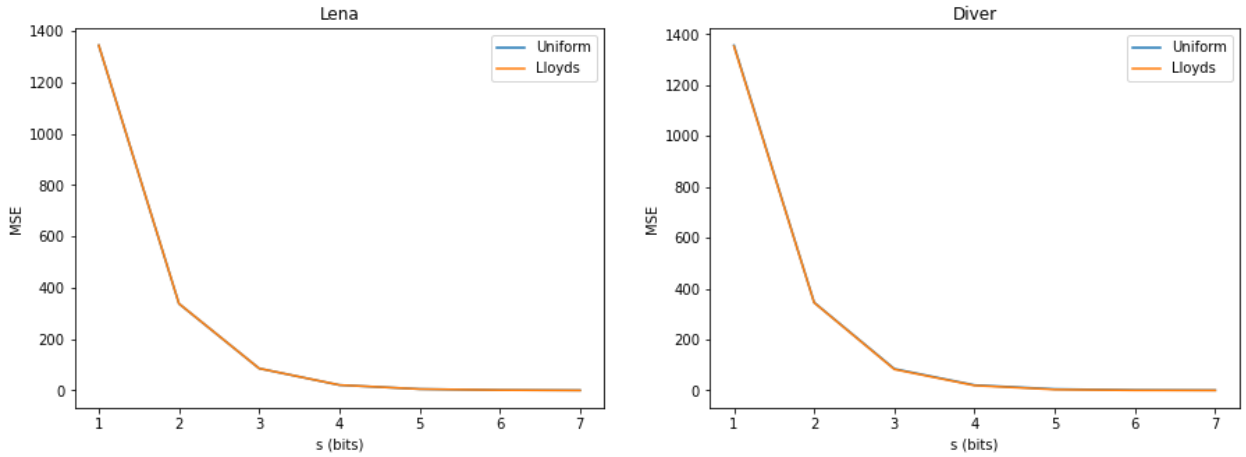
```
ax1.plot(np.arange(1,8), lloydsMSE_lena, label="Lloyds")
ax1.set_xlabel("s (bits)")
ax1.set_ylabel("MSE")
ax1.legend()
ax1.set_title("Lena")
ax2.plot(np.arange(1,8), unifMSE_diver, label="Uniform")
ax2.plot(np.arange(1,8), lloydsMSE_diver, label="Lloyds")
ax2.set_xlabel("s (bits)")
ax2.set_ylabel("MSE")
ax2.legend()
ax2.set_title("Diver")
plt.show();
```



Lloyd Max is optimal so it will always outperform other quantization methods in the mean square sense. There is less of a discrepancy in the Lena case since it has a more uniform histogram.

```
lena = cv2.equalizeHist(lena)
diver = cv2.equalizeHist(diver)
lena_flat = np.reshape(lena, [np.prod(lena.shape), 1])
diver_flat = np.reshape(diver, [np.prod(diver.shape), 1])

unifMSE_lena = np.zeros((7,1))
unifMSE_diver = np.zeros((7,1))
lloydsMSE_lena = np.zeros((7,1))
lloydsMSE_diver = np.zeros((7,1))

for s in range(1, 8):
    unif_lena = uniformQuantization(lena, s)
    unif_diver = uniformQuantization(diver, s)
    p_lena, cb_lena = lloyds(lena_flat, [2**s])
    p_diver, cb_diver = lloyds(diver_flat, [2**s])

    unifMSE_lena[s-1] = MSE(lena, unif_lena)
    unifMSE_diver[s-1] = MSE(diver, unif_diver)
    lloydsMSE_lena[s-1] = lloydsMSE(lena, p_lena, cb_lena)
```

```
    lloydsMSE_diver[s-1] = lloydsMSE(diver, p_diver, cb_diver)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,5))
ax1.plot(np.arange(1,8), unifMSE_lena, label="Uniform")
ax1.plot(np.arange(1,8), lloydsMSE_lena, label="Lloyds")
ax1.set_xlabel("s (bits)")
ax1.set_ylabel("MSE")
ax1.legend()
ax1.set_title("Lena")
ax2.plot(np.arange(1,8), unifMSE_diver, label="Uniform")
ax2.plot(np.arange(1,8), lloydsMSE_diver, label="Lloyds")
ax2.set_xlabel("s (bits)")
ax2.set_ylabel("MSE")
ax2.legend()
ax2.set_title("Diver")
plt.show();
```



They have the same MSE after histogram equalization. Uniform quantization attains optimal performance in the special case of an equalized histogram. But Lloyd Max is also always optimal, so they end up with the same MSE.

Problem 4

(i)

[31]:
```
def uniform_quantizer_color(image, num_levels):
    output = image.copy()
    incr = 255 / num_levels
    levels = []
    for i in range(0,num_levels):
        lower = i * incr
        output[image>=lower] = int((i+0.5)*incr)
        levels.append(int((i+0.5)*incr))
    levels = np.array(levels)
    return output, levels
```

(ii)

[32]:
```
def find_closest(v, levels):
    v = v.reshape((3,1))
    dist = np.abs(v - levels.astype(float))
    return levels[np.argmin(dist, axis=1)]
```

[33]:
```
def FSD(image, levels):
    output = image.copy()
    output = output.astype(float)
    for y in range(output.shape[0]):
        for x in range(0,output.shape[1]):
            old = output[y,x].copy()
            new = find_closest(old, levels).astype(float)
            output[y,x] = new
            quant_error = old - new
            if y+1 < output.shape[0]:
                output[y+1,x] = output[y+1,x] + quant_error * 5/16
                if x-1 > 0:
                    output[y+1,x-1] = output[y+1,x-1] + quant_error * 3/16
                if x+1 < output.shape[1]:
                    output[y+1,x+1] = output[y+1,x+1] + quant_error * 1/16
            if x+1 < output.shape[1]:
                output[y,x+1] = output[y,x+1] + quant_error * 7/16
    output = output.astype(np.uint8)
    return output
```

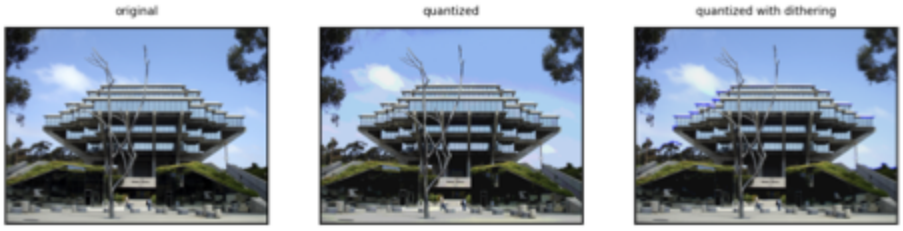
[34]:
```
in_image = cv2.imread("geisel.jpg", cv2.IMREAD_COLOR)
in_image = cv2.cvtColor(in_image, cv2.COLOR_BGR2RGB)
```

[35]:
```
quantized_image, levels = uniform_quantizer_color(in_image, 10)
```

[36]:
```
dithered_image = FSD(in_image, levels)
```

[40]:
```
# plotting
fig, axs = plt.subplots(1,3, figsize=(7,5), dpi=200)
axs[0].imshow(in_image)
axs[0].get_xaxis().set_visible(False)
axs[0].get_yaxis().set_visible(False)
axs[0].set_title("original", size=5)
axs[1].imshow(quantized_image)
axs[1].get_xaxis().set_visible(False)
axs[1].get_yaxis().set_visible(False)
```

```
axs[1].set_title("quantized", size=5)
im = axs[2].imshow(dithered_image)
axs[2].get_xaxis().set_visible(False)
axs[2].get_yaxis().set_visible(False)
axs[2].set_title("quantized with dithering", size=5)
plt.show()
```



[39]:
```
# plotting details
fig, axs = plt.subplots(1,3, figsize=(7,5), dpi=200)
axs[0].imshow(in_image[450:550,400:500])
axs[0].get_xaxis().set_visible(False)
axs[0].get_yaxis().set_visible(False)
axs[0].set_title("original", size=5)
axs[1].imshow(quantized_image[450:550,400:500])
axs[1].get_xaxis().set_visible(False)
axs[1].get_yaxis().set_visible(False)
axs[1].set_title("quantized", size=5)
im = axs[2].imshow(dithered_image[450:550,400:500])
axs[2].get_xaxis().set_visible(False)
axs[2].get_yaxis().set_visible(False)
axs[2].set_title("quantized with dithering", size=5)
plt.show()
```

By comparing the original image to the quantized image, we can observe a perceived lack of clarity as multiple similar colors are grouped to the same color. Although the second and the third image look very similar, there are subtle differences that can only be caught by zooming in to the image. The bottom row of images reveals this difference. The quantized image has clearly distinguishable color bands which are not so clear in the third image. The color bands have appeared to have blended with the addition of a dotted texture.

The cause for this modification can be best understood as a slight modification to some pixels in each color band to accommodate for the error in quantization. Values of neighboring pixels are modified relative to the quant error resulting in the biggest changes near the color band boundaries. In other word, quantization error is distributed across a larger area.