Reyhane Shahrokhian 99521361

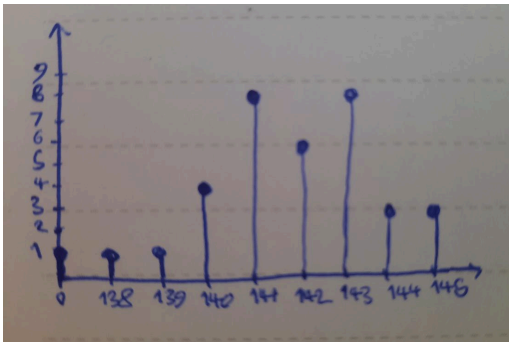
HomeWork2 of Computer Vision Course

Dr. Mohammadi

Q1:

1.1:

Pixel value	Histogram
0	1
138	1
139	1
140	4
141	8
142	6
143	8
144	3
145	3

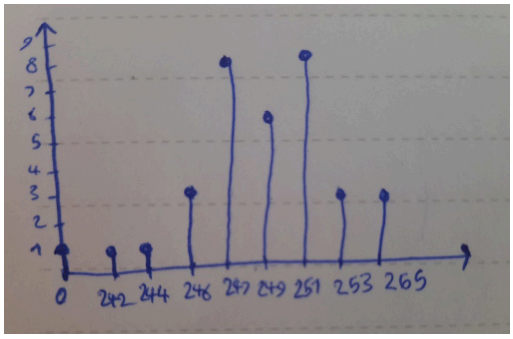


Stretching: I used linear stretching:

Values: $\frac{x-minimum}{maximum-minimum} \times 255$

Pixel value	Histogram	New value	Round value
0	1	0	0
138	1	$\frac{138}{145} \times 255 = 242.6$	242
139	1	$\frac{139}{145} \times 255 = 244.4$	244

140	4	$\frac{140}{145} \times 255 = 246.2$	246
141	8	$\frac{141}{145} \times 255 = 247.9$	247
142	6	$\frac{142}{145} \times 255 = 249.7$	249
143	8	$\frac{143}{145} \times 255 = 251.4$	251
144	3	$\frac{144}{145} \times 255 = 253.2$	253
145	3	255	255



New matrix: [247, 0, 242, 251, 251, 251, 253]

[247, 246, 246, 249, 249, 251, 251]

[246, 255, 255, 253, 249, 249, 255]

[247, 247, 247, 251, 249, 247, 251]

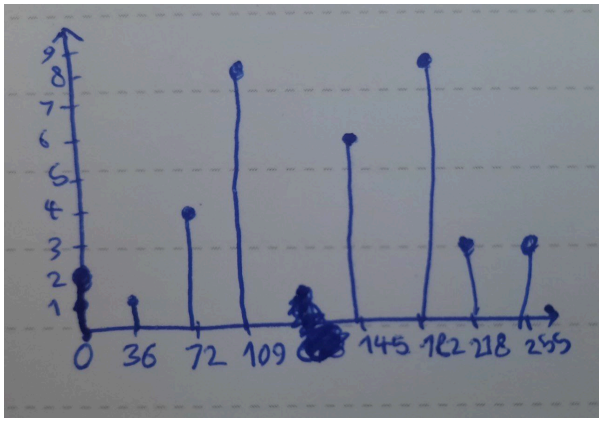
[244, 246, 247, 247, 249, 251, 253]

Clipping:

values less than 138 are considered as 138 and values more than 145(we don't have it) are considered as 145 and others are clipped from this formula:

Values: $\frac{x-minimum}{maximum-minimum} \times 255$

Pixel value	Histogram	New value	Round value
0	1	0	0
138	1	0	0
139	1	$\frac{1}{7} \times 255 = 36.4$	36
140	4	$\frac{2}{7} \times 255 = 72.8$	72
141	8	$\frac{3}{7} \times 255 = 109.2$	109
142	6	$\frac{4}{7} \times 255 = 145.7$	145
143	8	$\frac{5}{7} \times 255 = 182.1$	182
144	3	$\frac{6}{7} \times 255 = 218.5$	218
145	3	255	255



New matrix: [109, 0, 0, 182, 182, 182, 218]

[109, 72, 72, 145, 145, 182, 182]

[72, 255, 255, 218, 145, 145, 255]

[109, 109, 109, 182, 145, 109, 182]

[36, 72, 109, 109, 145, 182, 218]

1.2:

First we should implement the histogram calculator which is just counting each value:

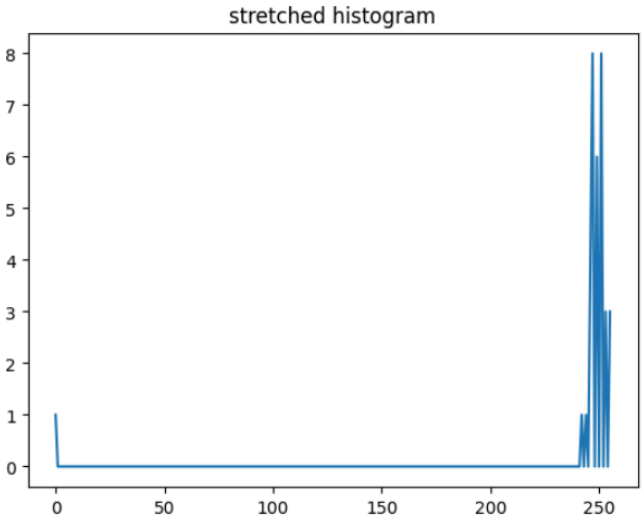
```
hist = np.zeros(256, dtype=int)
for row in image:
    for pixel in row:
        hist[pixel] += 1
return hist
```

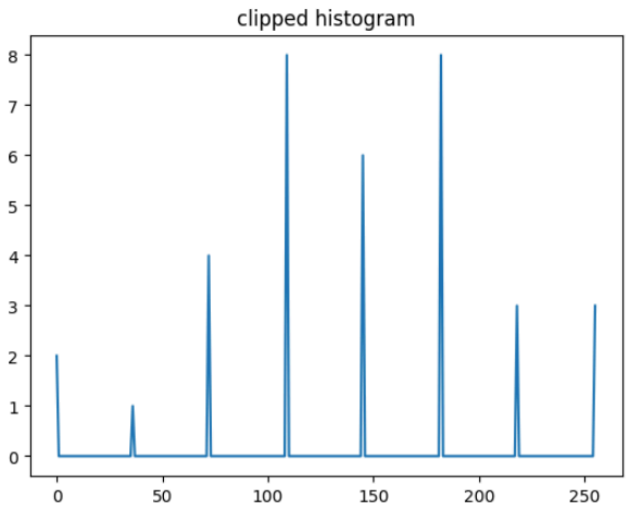
Then we are suppose to implement the stretching and clipping functions exactly as the formula that we had above:

```
output_image = image.copy()
# Start
minimum = min(map(min, output_image))
maximum = max(map(max, output_image))

output_image = [[(pixel - minimum) * 255 // (maximum - minimum) for pixel in row] for row in output_image]
# End
return output_image
```

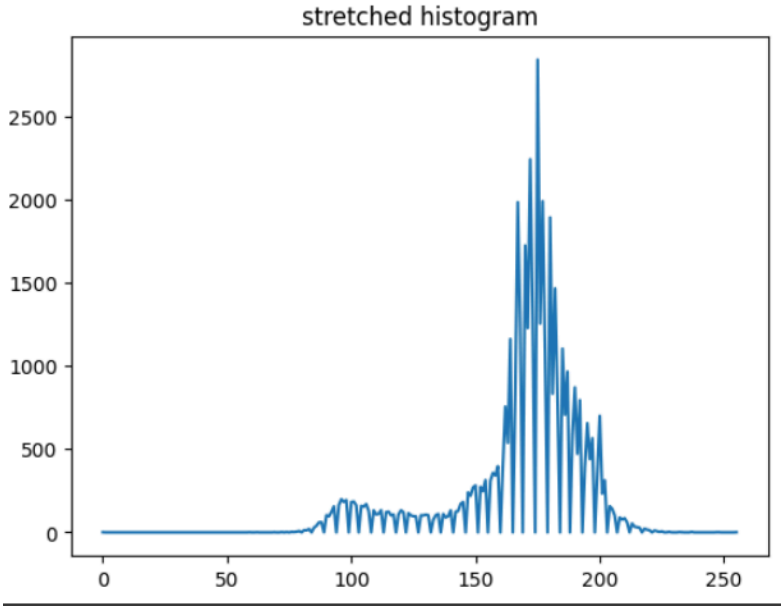
```
output_image = image.copy()
# Start
for i in range(len(image)):
  for j in range(len(image[i])):
    if image[i][j] < min_value:
      output_image[i][j] = 0
    elif image[i][j] > max_value:
      output_image[i][j] = 255
    else:
      output_image[i][j] = (image[i][j] - min_value) * 255 // (max_value - min_value)
# End
return output_image
```



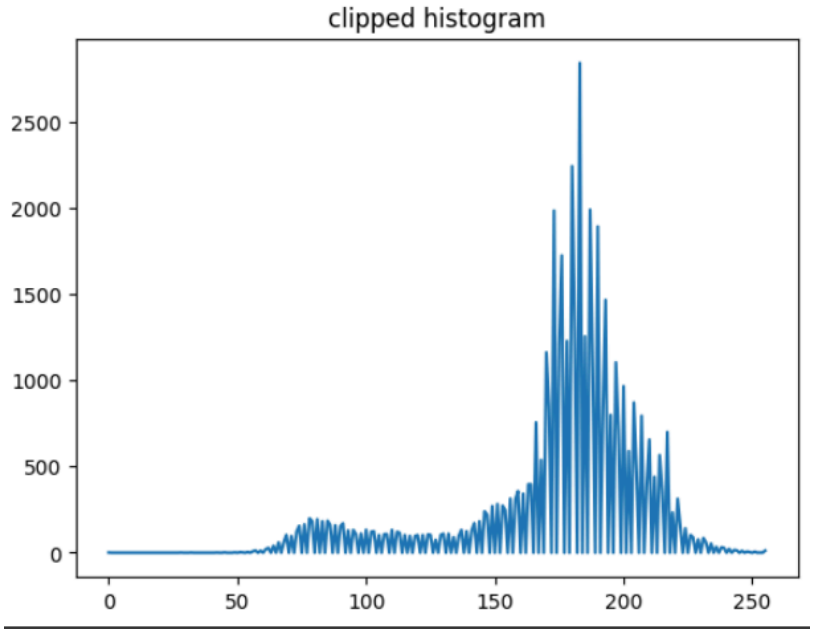


1.3:

After calling stretching:



After calling clipping:



The differences in the outcomes of stretching and clipping can be attributed to their distinct purposes and methodologies. Contrast stretching primarily focuses on improving the overall contrast and dynamic range of an image, making it more visually appealing by adjusting pixel values within a specified range. In contrast, image clipping is more about refining the composition of an image, removing distractions, enhancing specific elements, and creating a more polished and attractive visual representation.

Q2:

2.1:

Src:

Pixel value	Histogram	Equalized
0	8	8
1	32	40
2	24	64

Ref:

Pixel value	Histogram	Equalized
2	8	8
3	8	16
4	8	24
5	16	40
6	8	48
7	16	64

Mapping:

Src	Ref
$0 \rightarrow 8$	$8 \rightarrow 2$
$1 \rightarrow 40$	$40 \rightarrow 5$
$2 \rightarrow 64$	$64 \rightarrow 7$

New Src:

2	2	2	2	2	2	2	2
5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5
7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7
5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5

2.2:

In the first step to calculate the histogram I used np.bincount that counts occurrences of each value in the flattened image array and returns an array where the index represents the pixel value and the value at that index represents the count of occurrences. The minlenght argument ensures that the resulting histogram array has a length of 256, even if some pixel values are not present in the image. To flatten the image I used image.flatten() that flattens the 2D image array into a 1D array.

```
hist = np.bincount(image.flatten(), minlength=256)
```

For calculating the cumulative distribution function (CDF) of an input image channel, first I called calc_hist function to compute the histogram of the input image channel. Then np.cumsum calculates the cumulative sum of the histogram values. At the end we should normalize the cumulative sum array by dividing each element by the maximum value in the array.

```
hist = calc_hist(channel)

cdf = np.cumsum(hist)
cdf = cdf / cdf.max()
```

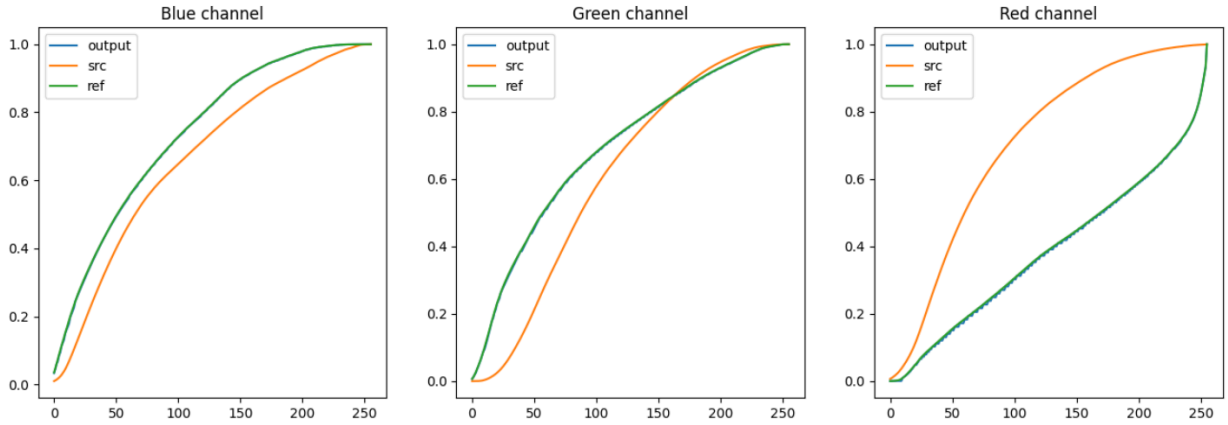
In the next function we should perform histogram matching between a source image and a reference image. So, for each channel, first the CDF of source and reference images are calculated. Then I used the np.searchsorted() function to find the indices in the reference CDF where the values from the source CDF would fit. This helps in mapping pixel intensities from the source to the reference distribution. Now it's time to update the channel in the output image by mapping the pixel values from the source image to the reference image based on the calculated mapping.

```
src_cdf = calc_cdf(src_image[:, :, channel])
ref_cdf = calc_cdf(ref_image[:, :, channel])

mapp = np.searchsorted(ref_cdf, src_cdf, side='right')

output_image[:, :, channel] = mapp[src_image[:, :, channel]]
```

And that's the final output:



As it is clear, color channels in the output image match those of the reference image, this occurs because the process aims to adjust the pixel intensities in the source image to align with the distribution of the reference image.

Q3:

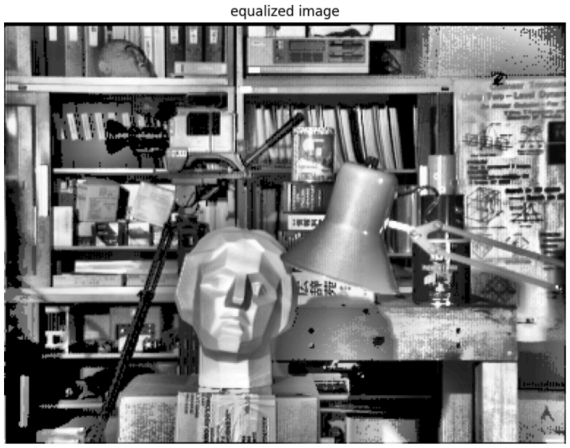
3.1:

The disadvantages of basic histogram equalization include the potential to increase noise in the image. We may also lose some details. To address this limitation, adaptive histogram equalization is used, which divides the image into smaller regions and applies histogram

equalization independently to each region, resulting in higher contrast images while ideally suppressing noise

3.2:

ACE1:

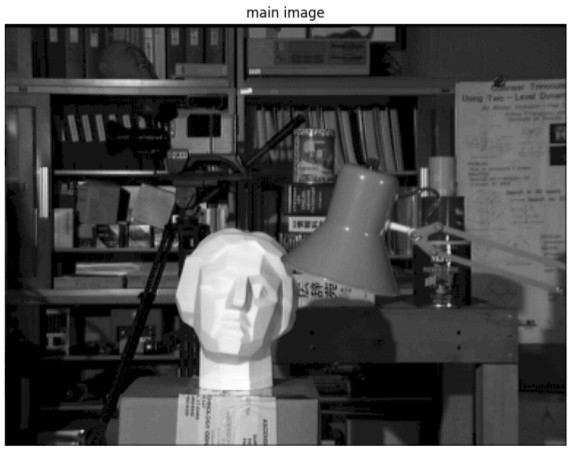


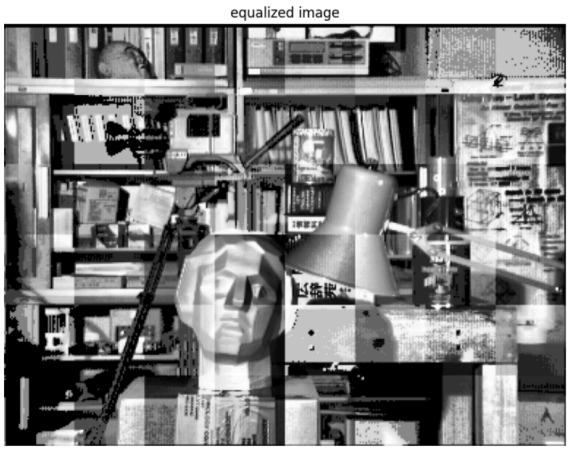
The advantages of the ACE1 method implemented in the provided Python function include the ability to enhance the contrast of an image by applying histogram equalization to smaller grid regions. By dividing the image into grids and performing histogram equalization on each grid independently, the ACE1 method can effectively improve the contrast of local regions, leading to enhanced details and clarity in the image. This approach is particularly useful for images with varying contrast levels across different parts, allowing for targeted contrast enhancement in specific areas.

On the other hand, a potential disadvantage of the ACE1 method is that it may not be suitable for images with significant noise or artifacts. Since histogram equalization amplifies intensity values without considering noise levels, applying this method to noisy images can lead to noise amplification and degradation of image quality. In such cases, additional preprocessing steps or

alternative methods may be necessary to address noise issues before applying histogram equalization for contrast enhancement.

ACE2:





The ACE2 method offers advantages such as simplicity in applying histogram equalization to each pixel by calculating transition functions for individual pixels. ACE2 simplifies the process of enhancing the contrast of an image at a pixel level. This method allows for fine-grained contrast adjustments, potentially leading to improved image quality and clarity. Additionally, by incorporating padding using OpenCV's built-in tools, ACE2 ensures that the histogram equalization process considers the surrounding pixel values, enhancing the accuracy of contrast enhancement for each pixel.

However, a potential disadvantage of the ACE2 method is that it may be computationally intensive, especially when processing large images due to the pixel-wise calculation of transition functions. This approach could lead to increased processing time and resource utilization, which

may not be ideal for real-time applications or scenarios where efficiency is crucial. Additionally, the pixel-wise calculation may not be necessary for all images, and in some cases, grid-based histogram equalization methods might be more efficient and effective for contrast enhancement

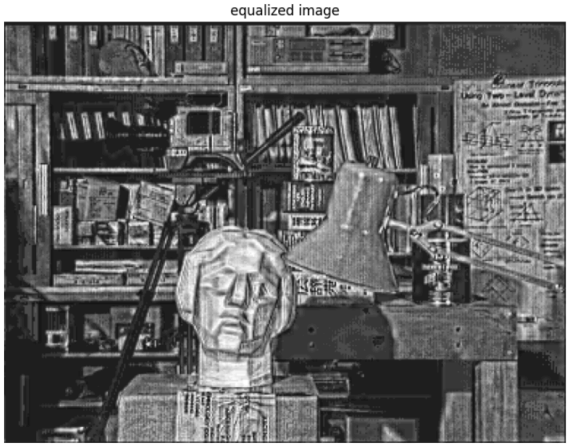
CLAHE:

CLAHE (Contrast Limited Adaptive Histogram Equalization) is particularly effective in enhancing the local contrast of images by limiting the contrast amplification in regions with high contrast, thereby preventing over-amplification of noise. CLAHE can significantly improve the quality of images, especially in scenarios where there are variations in contrast across different parts of the image. CLAHE helps in suppressing noise while enhancing contrast by limiting the amplification of intensity values, resulting in images with improved clarity and reduced noise artifacts.

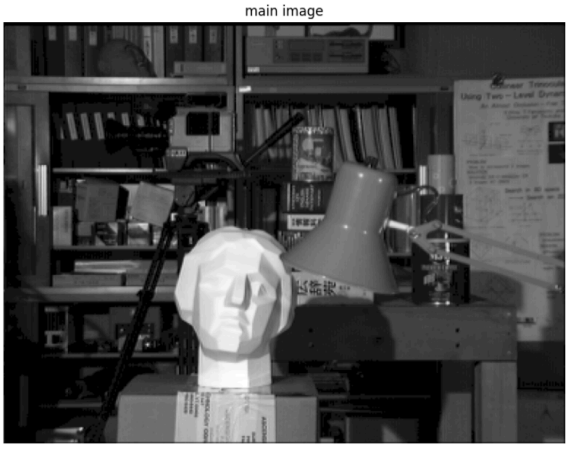
The CLAHE method can be computationally intensive, especially when processing large images pixel by pixel. Implementing CLAHE requires a more intricate process compared to basic histogram equalization methods. The need to calculate transition functions for each pixel and manage contrast limiting thresholds adds complexity to the implementation.

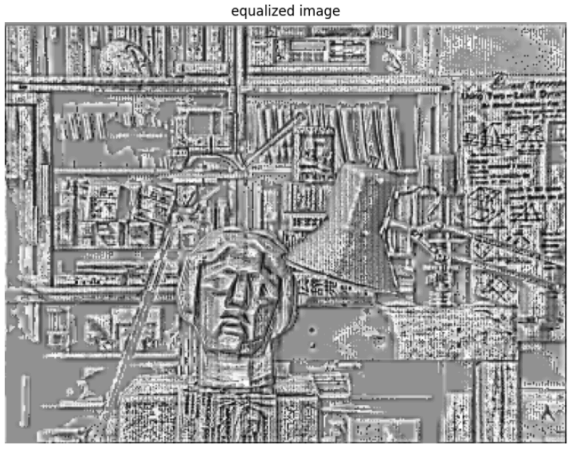
3.3:

Output of 128*128 with threshold=2:

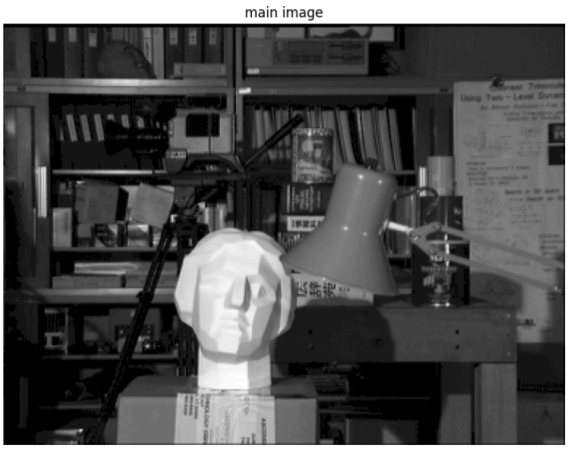


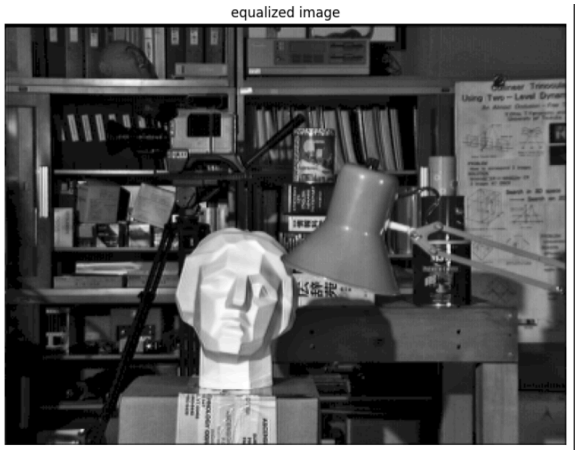
Output of 128*128 with threshold=128:



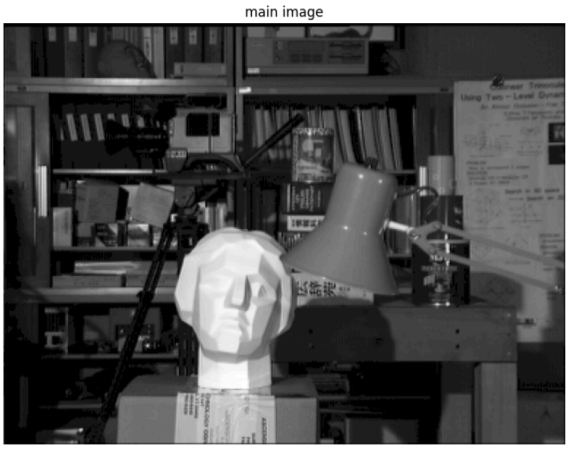


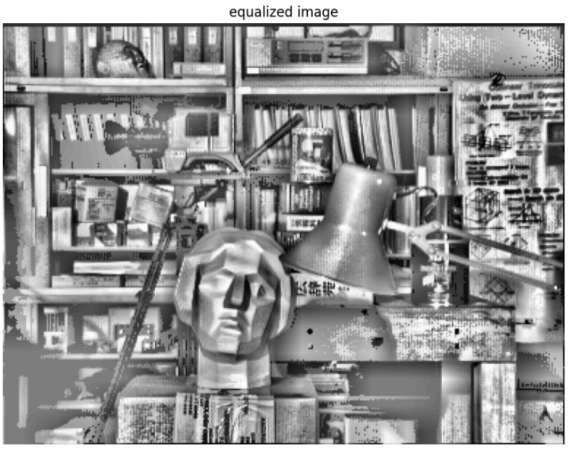
Output of 16*16 with threshold=2:





Output of 16*16 with threshold=128:





The window size and threshold values in CLAHE play a crucial role in balancing between global and local contrast enhancement as well as noise suppression.

A larger window size, such as (128, 128) pixels, allows for histogram equalization over a larger area, potentially leading to more global contrast enhancement but with a risk of over-smoothing and loss of local details. Conversely, a smaller window size, like (16, 16) pixels, focuses on local regions, enabling finer contrast adjustments and preserving more detailed information within smaller areas.

A higher threshold value, like 128, imposes stricter limits on contrast enhancement, preventing excessive amplification of intensity values. This can help in reducing noise amplification but may result in less pronounced contrast improvement. On the other hand, a lower threshold value, such as 2, allows for more aggressive contrast enhancement by permitting a wider range of intensity adjustments. This can lead to more significant contrast improvements but may also amplify noise in the image.

Q4:

4.1:

```python
def Add_Noise(img):
    """
    Add salt and pepper noise to the input image.
        Parameters:
          image: Input image (numpy array).
        Returns:
          Image with salt and pepper noise added.
    """
    noisy_image = np.copy(img)
    salt_vs_pepper = 0.2
    amount = 0.05

    num_salt = np.ceil(amount * img.size * salt_vs_pepper)
    num_pepper = np.ceil(amount * img.size * (1.0 - salt_vs_pepper))

    # Add Salt noise
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in img.shape]
    noisy_image[tuple(coords)] = 255

    # Add Pepper noise
    coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in img.shape]
    noisy_image[tuple(coords)] = 0

    return noisy_image
```

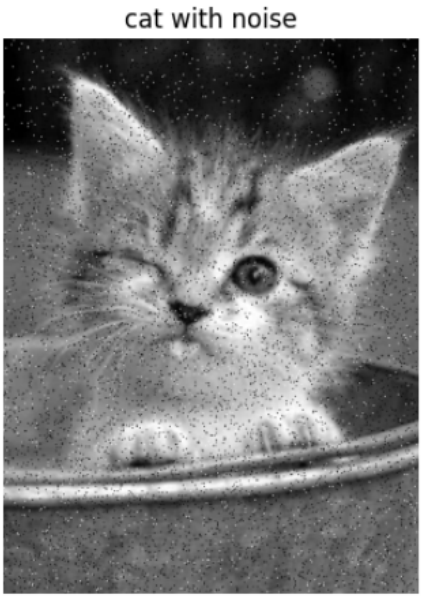
Firstly, I created a copy of the input image to avoid modifying the original image and set parameters for the noise generation:

salt_vs_pepper: Ratio of salt (white) to pepper (black) noise.

amount: Overall noise density in the image.

Then, the number of salt and pepper pixels are calculated to be added based on the image size, noise density, and the specified ratio. Random coordinates within the image dimensions are generated for salt noise.

The intensity value 255 (white) is assigned to these random coordinates in the noisy image, simulating salt noise. The intensity value 0 (black) is assigned to these random coordinates in the noisy image, simulating pepper noise.



4.2:

```
def Reflect101(img,filter_size):
  '''
    Do not use loop (like while and for)
    Do not use libraries
    calculate averaging filter
    input(s):
      img (ndarray): input image
      filter_size (ndarray): filter size
    output(s):
      image (ndarray): computed Reflect101
  '''

  ######################
  #    your code here   #
  image = np.pad(img, filter_size // 2, mode='reflect')
  ######################

  return image
```

This padding method extends the boundaries of the image by reflecting the pixels at the edges, effectively handling edge cases when applying filters or operations that require access to neighboring pixels.

```python
def Averaging_Blurring(img, filter_size):
    '''
    Do not use libraries
    input(s):
      img (ndarray): input image
      filter_size (ndarray): filter size
    output(s):
      result (ndarray): computed averaging blurring
    '''
    image = Reflect101(img, filter_size)
    result = np.zeros((img.shape))

    ########################
    #    your code here    #
    result = np.zeros_like(img)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            result[i][j] = np.mean(image[i:i+filter_size, j:j+filter_size])
    ########################

    return result
```

```python
def Median_Blurring(img, filter_size):
    '''
    Do not use libraries
    input(s):
      img (ndarray): input image
      filter_size (ndarray): filter size
    output(s):
      result (ndarray): computed median blurring
    '''
    image = Reflect101(img, filter_size)
    result = np.zeros((img.shape))

    ########################
    #    your code here    #
    result = np.zeros_like(img)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            result[i][j] = np.median(image[i:i+filter_size, j:j+filter_size])
    ########################

    return result
```

```python
def Gaussian_Blurring(img, filter_size, std):
    '''
    Do not use libraries
    input(s):
      img (ndarray): input image
      filter_size (tuple): filter size
      std (float): std of gaussian kernel
    output(s):
      result (ndarray): computed gaussian blurring
    '''
    kernel = np.zeros((filter_size,filter_size))
    ########################
    #    your code here    #
    for i in range(filter_size):
        for j in range(filter_size):
            kernel[i,j] = (1/ (2 * np.pi * std ** 2)) * np.exp(- ((i - (filter_size - 1) / 2) ** 2 + (j - (filter_size - 1) / 2) ** 2) / (2 * std ** 2))
    kernel /= np.sum(kernel)

    ########################
    output = img.copy()
    result = cv2.filter2D(src = output, ddepth = -1, kernel = kernel)
    return result
```

Average Blurring function calculates the average pixel value within a specified filter window by sliding the filter over the image.

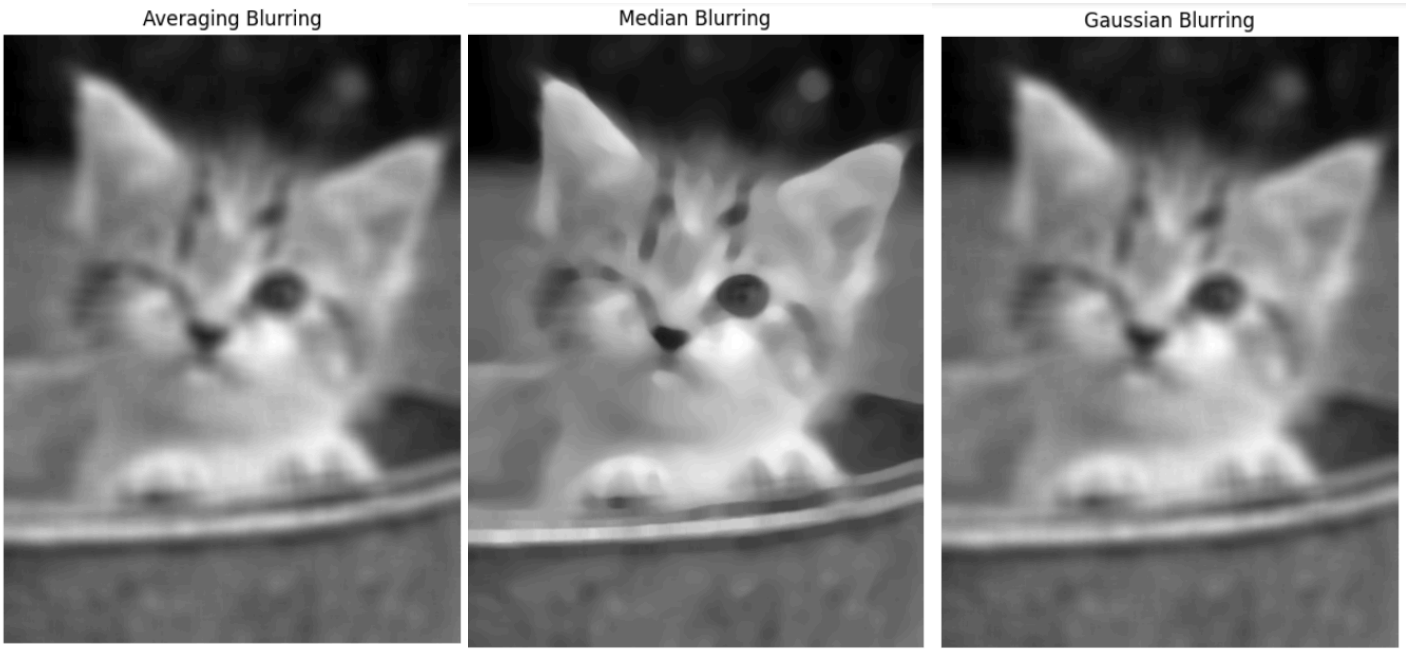
The kernel size significantly impacts the output of average blurring. As the kernel size increases, the image becomes progressively more blurred. Larger kernel sizes lead to a more extensive averaging of pixel values within the filter window, resulting in a smoother and more blurred image.

Median Blurring function computes the median pixel value within the filter window. Larger kernel sizes in median blurring operations can lead to increased blurring of the image, affecting the level of detail and sharpness. Specifically, using larger kernel sizes in median blurring can result in a more pronounced blurring effect, potentially reducing the image's clarity and fine details.

Gaussian Blurring applies a Gaussian kernel to the input image for blurring. It generates a 2D Gaussian kernel based on the specified standard deviation and filter size. The function convolves the Gaussian kernel with the input image using a filter2D operation to produce a Gaussian blurred image. Larger kernel sizes result in more extensive blurring of the image, affecting the level of detail and sharpness in the output. When the kernel size is increased, the blurring effect becomes more pronounced, leading to a smoother and more blurred image.

4.3:

As it is clear in notebook file the output from libraries are exactly the same as what I had written by myself.



Q5:

Kernel = [[0, 1, 0],

[1, -4, 1],

[0, 1, 0]]

We should apply the Laplacian kernel to the image using convolution.

As the pixel values except the central one are the same, we can make it easier by just calculating 3 different cases:

1. The center of kernel is on the center of image:

   $0 + 10 + 0 + 10 - 48 + 10 + 0 + 10 + 0 = -8$

2. The center of kernel have one pixel difference from the center of image:

   $0 + 10 + 0 + 10 - 40 + 12 + 0 + 10 + 0 = 2$

3. Other places of kernel:

$$0 + 10 + 0 + 10 - 40 + 10 + 0 + 10 + 0 = 0$$

New matrix:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	2	0	0	0
0	0	0	2	-8	2	0	0
0	0	0	0	2	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Q6:

The 2D discrete Fourier transform (DFT) of f(x, y) is:

$$F(u, v) = \sum_{x=0}^{1} \sum_{y=0}^{1} f(x, y) e^{-i2\pi(\frac{ux}{2} + \frac{vy}{2})}$$

$$F(0, 0) = 1 \times e^{0} + 2 \times e^{0} + 2 \times e^{0} + 1 \times e^{0} = 6$$

$$F(0, 1) = 1 \times e^{0} + 2 \times e^{-i\pi} + 2 \times e^{0} + 1 \times e^{-i\pi} = 1 - 2 + 2 - 1 = 0$$

$$F(1, 0) = 1 \times e^{0} + 2 \times e^{0} + 2 \times e^{-i\pi} + 1 \times e^{-i\pi} = 1 + 2 - 2 - 1 = 0$$

$$F(1, 1) = 1 \times e^{0} + 2 \times e^{-i\pi} + 2 \times e^{-i\pi} + 1 \times e^{-i2\pi} = 1 - 2 - 2 + 1 = -2$$

New matrix:

6	0
0	-2