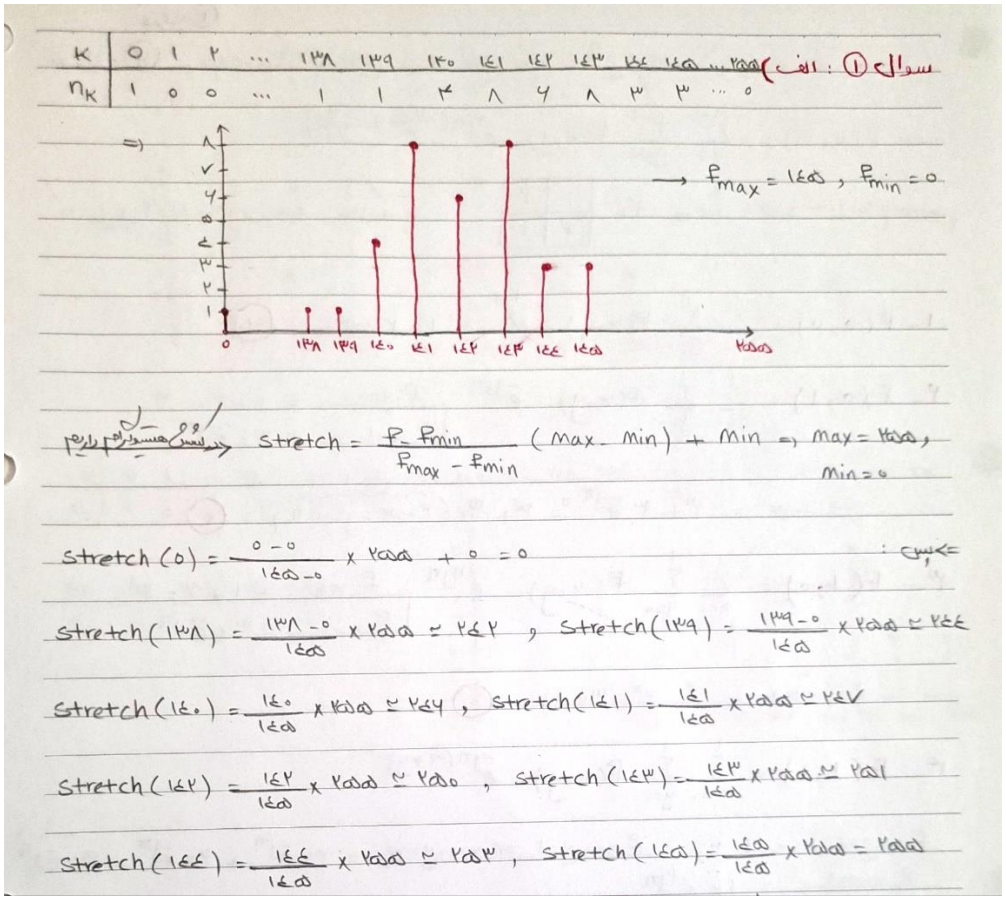
به نام خالق رنگین کمان

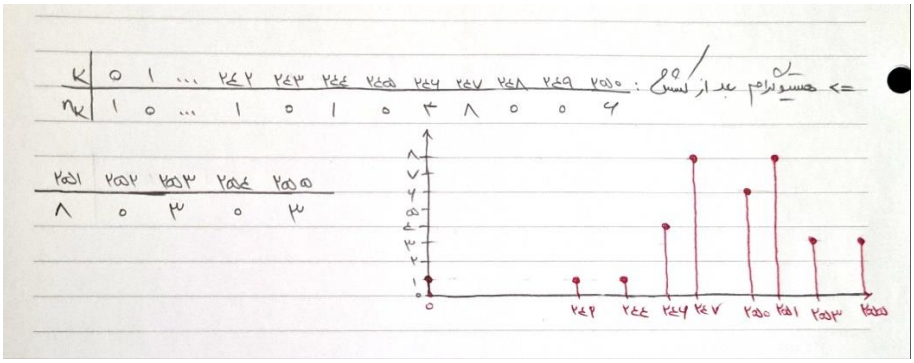
ستاره باباجانی – گزارش تمرین شماره 2

سوال 1: الف)



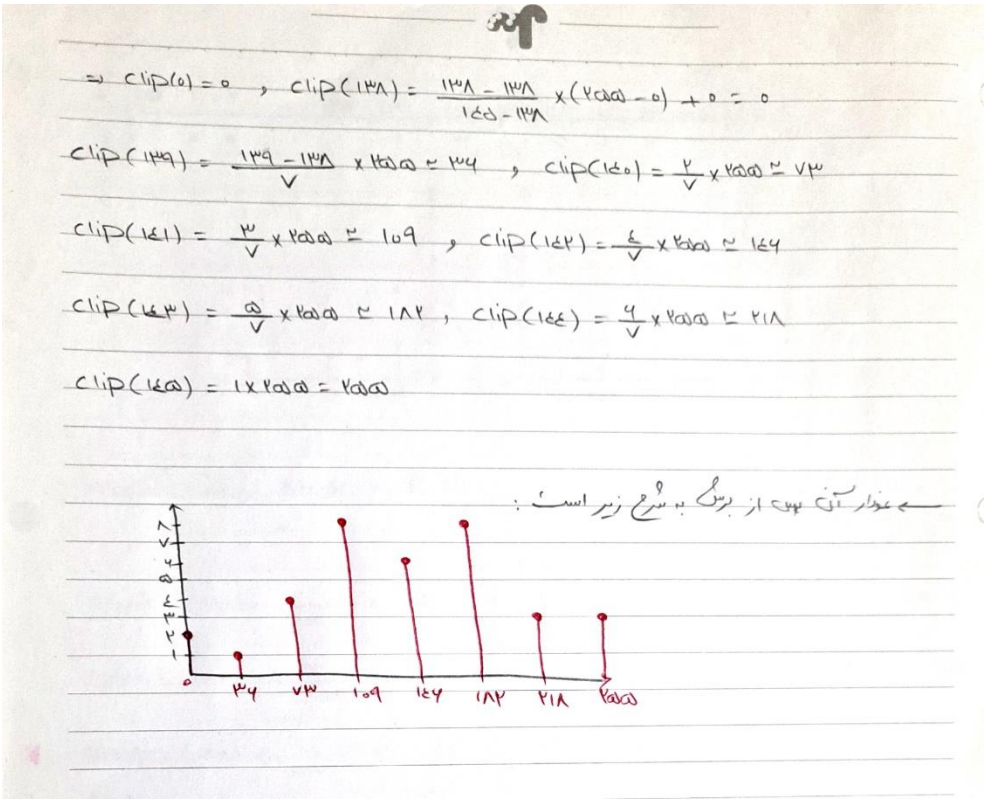
که تصویر حاصل بعد از کشش هیستوگرام به شرح زیر است:

247	0	242	251	251	251	253
247	246	246	250	250	251	251
246	255	255	253	250	250	255
247	247	247	251	250	247	251
244	246	247	247	250	251	253



برای برش هیستوگرام، عدد 0 را داده پرت در نظر میگیریم و محدوده هیستوگرام را از 138 تا 145 در نظر میگیریم و طبق فرمول گفته شده عمل میکنیم، که تصویر حاصل بعد از برش هیستوگرام به شرح زیر است:

109	0	0	182	182	182	218
109	73	73	146	146	182	182
73	255	255	218	146	146	255
109	109	109	182	146	109	182
36	73	109	109	146	182	218



ب) در این بخش توابع هیستوگرام، کشش و برش را تعریف کردیم:

- تعریف image1:

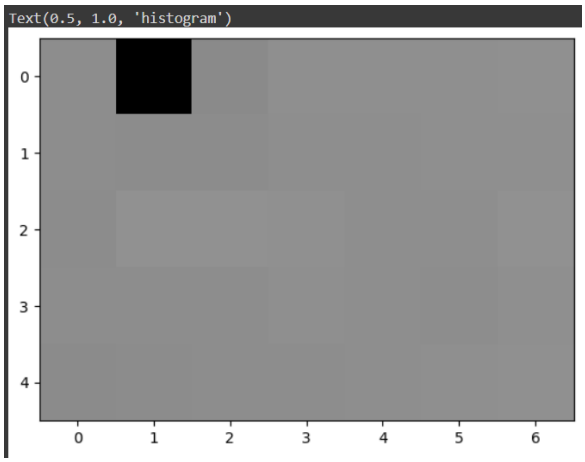
```
#define image1 here
image1 = np.array([
    [141, 0 , 138, 143, 143, 143, 144],
    [141, 140, 140, 142, 142, 143, 143],
    [140, 145, 145, 144, 142, 142, 145],
    [141, 141, 141, 143, 142, 141, 143],
    [139, 140, 141, 141, 142, 143, 144]
], dtype=np.uint8)
```

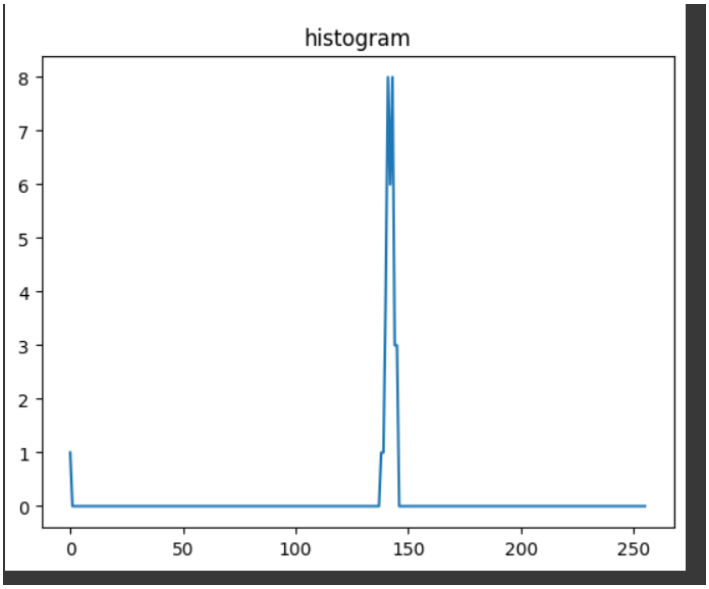
- تعریف تابع هیستوگرام: برای تعریف این تابع، از تابع آماده calcHist موجود در OpenCV استفاده کردم:

```python
#code here
#first define a function for calculating histogram
#you are free to use libraries
def calc_hist(image):
    '''
    you are free to use libraries
    calculate image histogram
    input(s):
      image (ndarray): input image
    output(s):
      hist (ndarray): computed input image histogram
    '''
    hist = cv2.calcHist([image], [0], None, [256], [0, 256])

    return hist

```

حال به نمایش هیستوگرام image1 میپردازیم(همان طور که مشاهده میشود با نمودار رسم شده در بخش الف، تطابق دارد):

```python
#dont change this cell
plt.imshow(image1,cmap='gray',vmin=0,vmax=255)
plt.figure()
plt.plot(calc_hist(image1))
plt.title('histogram')
```





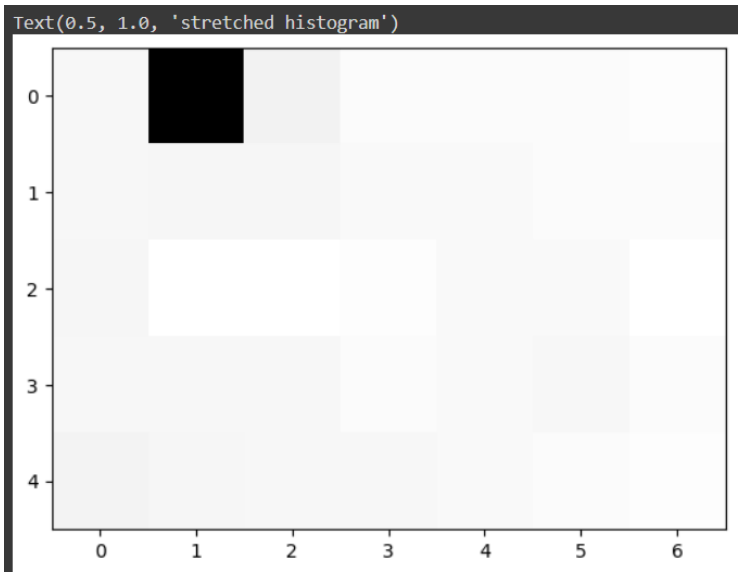
- تعریف تابع کشش: برای تعریف این تابع، طبق فرمول زیر، مقادیر مشخص شدند و در اخر چون حاصل تقسیم عدد اعشار میشد، به سمت پایین گرد کردم:

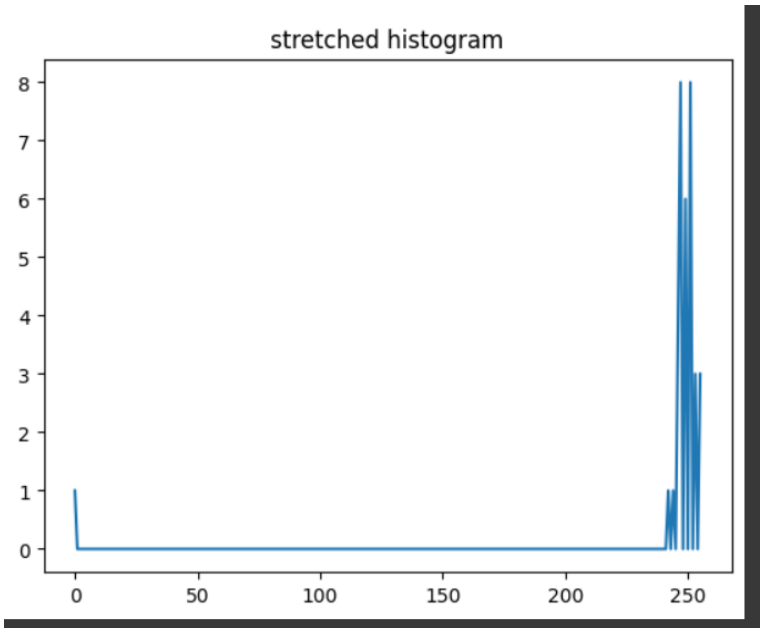
$$g(x,y) = stretch[f(x,y)] = \left(\frac{f(x,y) - f_{min}}{f_{max} - f_{min}}\right)(MAX - MIN) + MIN$$

```python
#code here
#define a function (stretch) for stretching(input:image , output: stretched image)

def stretch_hist(image):
  '''
  don't use libraries
  input(s):
    image (ndarray): input image
  output(s):
    output_image (ndarray): enhanced image with histogram stretching
  '''
  output_image = image.copy()
  # Start

  # Calculating minimum and maximum intensity values
  f_min = np.min(output_image)
  f_max = np.max(output_image)

  # Defining desired output intensity range
  min = 0
  max = 255

  # Stretch the histogram
  output_image = np.floor(((output_image - f_min) / (f_max - f_min)) * (max - min) + min).astype(np.uint8)

  # End
  return output_image
```

حال کشش هیستوگرام را روی image1 اعمال میکنیم:

```
#dont change this cell
plt.imshow(stretch_hist(image1),cmap='gray',vmin=0,vmax=255)
plt.figure()
plt.plot(calc_hist(stretch_hist(image1)))
plt.title('stretched histogram')
```





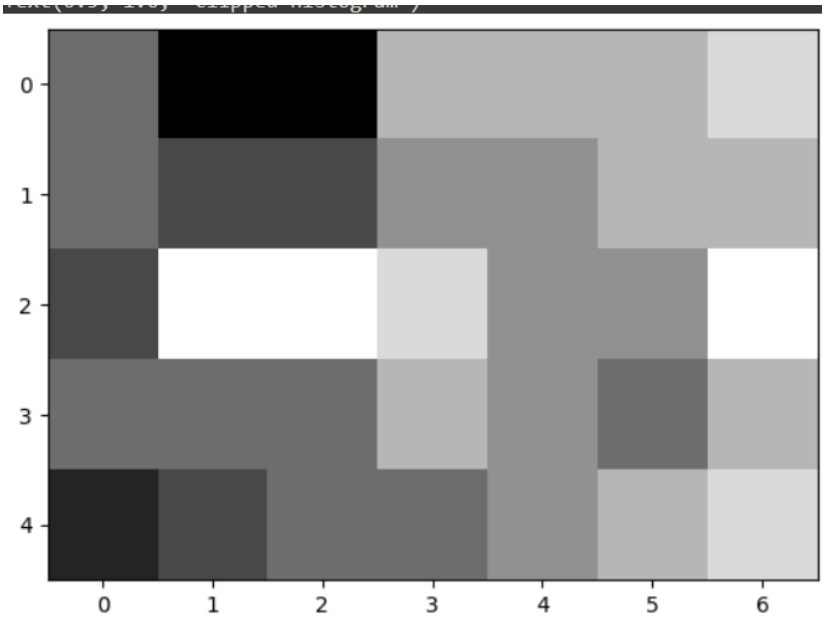
- تعریف برش هیستوگرام: برای تعریف این تابع، همان طور که گفته شد محدوده را بین 138 تا 145 میکنیم و از فرمول زیر استفاده میکنیم:

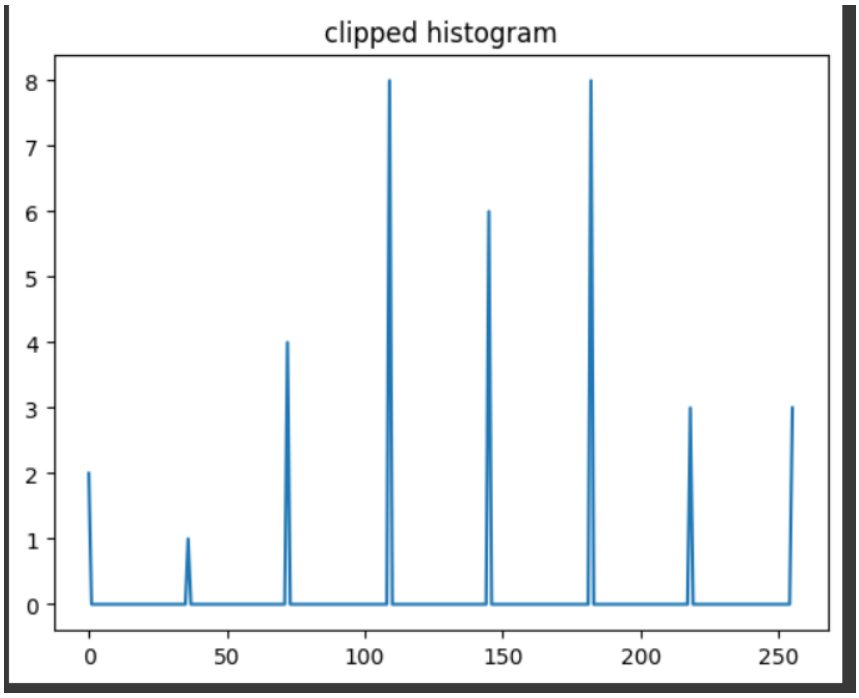
$$g(x,y) = clip[f(x,y)] = \left(\frac{f(x,y) - f_1}{f_{99} - f_1}\right)(MAX - MIN) + MIN$$

```python
def clip_hist(image, min_value, max_value):
    '''
    don't use libraries
    input(s):
      image (ndarray): input image
      min_value : min value of the histogram which you wanna clip.
      max_value : max value of the histogram which you wanna clip.
    output(s):
      output_image (ndarray): enhanced image with histogram clipping
    '''
    output_image = image.copy()
    # Start

    # Clip the intensity values to desired range
    output_image = np.clip(output_image, min_value, max_value)

    min = 0
    max = 255

    # cliping formula for the histogram
    output_image = np.floor(((output_image - min_value) / (max_value - min_value)) * (max - min) + min).astype(np.uint8)


    # End
    return output_image
```

حال برش را روی image1 اعمال میکنیم:

```python
#dont change this cell
min_value = 138 # your min value here
max_value = 145 # your max value here
plt.imshow(clip_hist(image1, min_value, max_value),cmap='gray',vmin=0,vmax=255)
plt.figure()
plt.plot(calc_hist(clip_hist(image1, min_value, max_value)))
plt.title('clipped histogram')
```

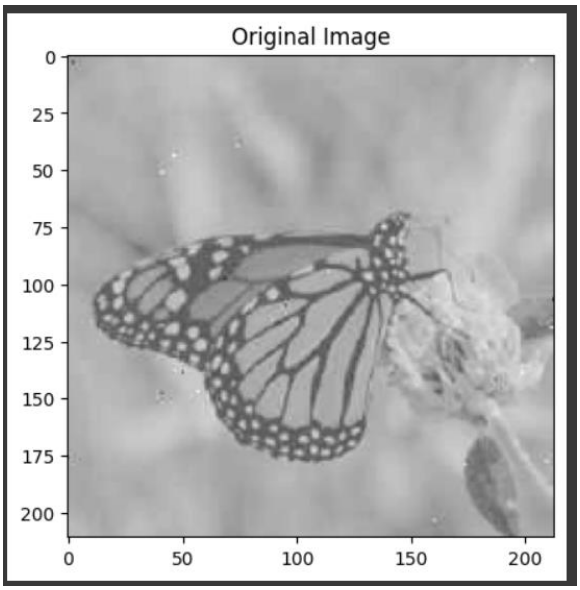




ج) حال در این بخش، توابع تعریف شده را روی تصویر جدید اعمال میکنیم:

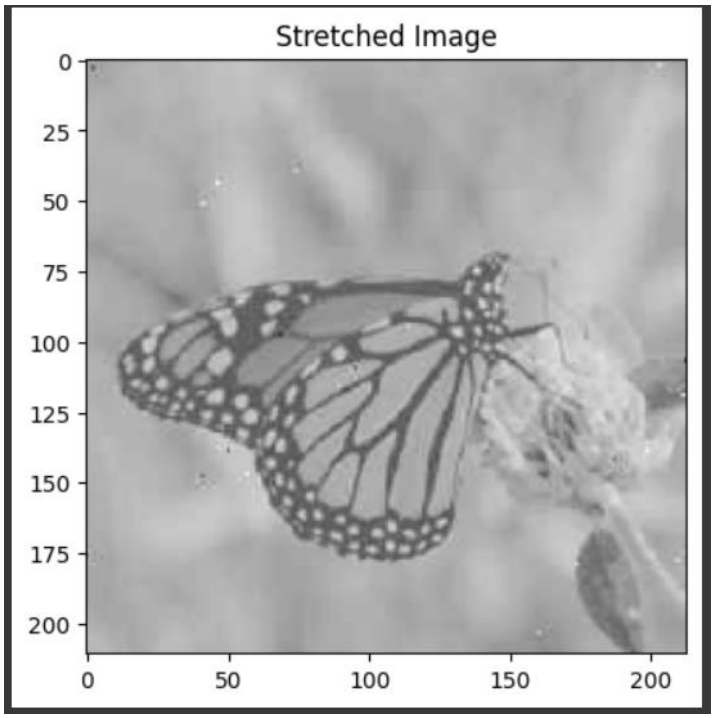
- تعریف و نمایش تصویر جدید:

```
# first read the image and show it.(image2)
image2 = cv2.imread("image2.png", cv2.IMREAD_GRAYSCALE)

# Show the original image
plt.imshow(image2, cmap='gray')
plt.title('Original Image')
plt.show()
```



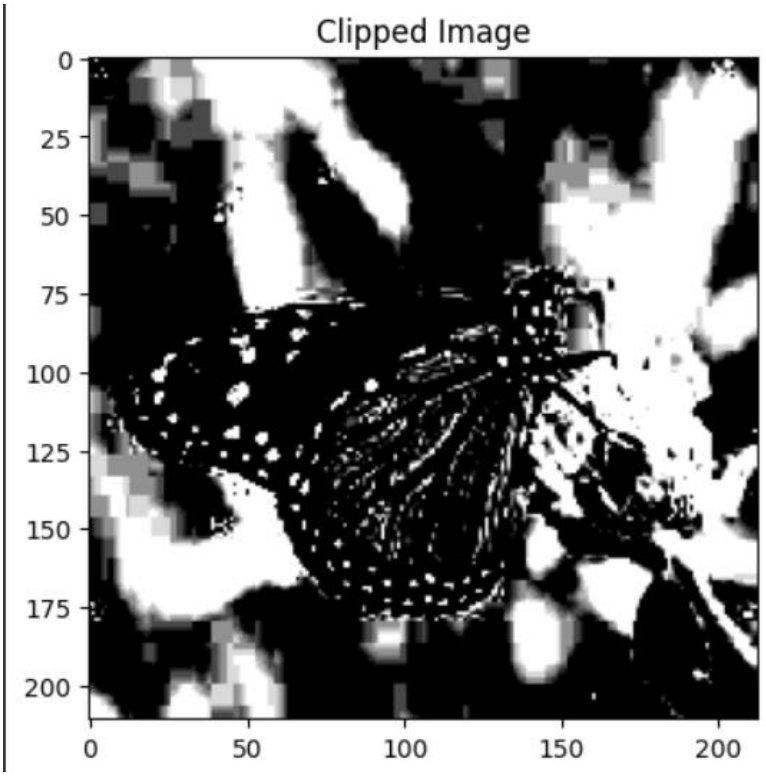
- اعمال کشش روی تصویر:

```
#use stretch_hist function to improve quality of the image and show it
stretched_image2 = stretch_hist(image2)

# Show the stretched image
plt.imshow(stretched_image2, cmap='gray')
plt.title('Stretched Image')
plt.show()
```



- اعمال برش روی تصویر:

```
#use clip_hist function to improve quality of the image and show it
min_value = 138  # Based on the part2
max_value = 145  # Based on the part2
clipped_image2 = clip_hist(image2, min_value, max_value)

# Show the clipped image
plt.imshow(clipped_image2, cmap='gray')
plt.title('Clipped Image')
plt.show()
```



1. کشش هیستوگرام: کشش هیستوگرام کنتراست تصویر را با پخش مقادیر شدت در کل محدوده افزایش می دهد. تمایل دارد تا روشنایی و کنتراست کلی تصویر را افزایش دهد و جزئیات را بیشتر نمایان کند و ظاهر بصری کلی را بهبود بخشد. با اعمال یک تبدیل خطی به مقادیر شدت، نگاشت محدوده شدت اصلی به محدوده دینامیکی کامل (به

عنوان مثال، 0 تا 255 برای یک تصویر 8 بیتی در مقیاس خاکستری) به دست می آید.

مزایا:

- برای تصاویر با کنتراست کم یا محدوده شدت محدود موثر است.
- جذابیت بصری تصویر را با افزایش جزئیات و کنتراست بهبود می بخشد.

2. برش هیستوگرام: مقادیر شدت تصویر را به یک محدوده مشخص محدود می کند و به طور موثر توزیع شدت را کوتاه می کند. با تنظیم مقادیر شدت زیر یک آستانه معین به حداقل مقدار محدوده مورد نظر و مقادیر شدت بالاتر از آستانه دیگر به حداکثر مقدار محدوده مورد نظر به دست می آید.

مزایا:

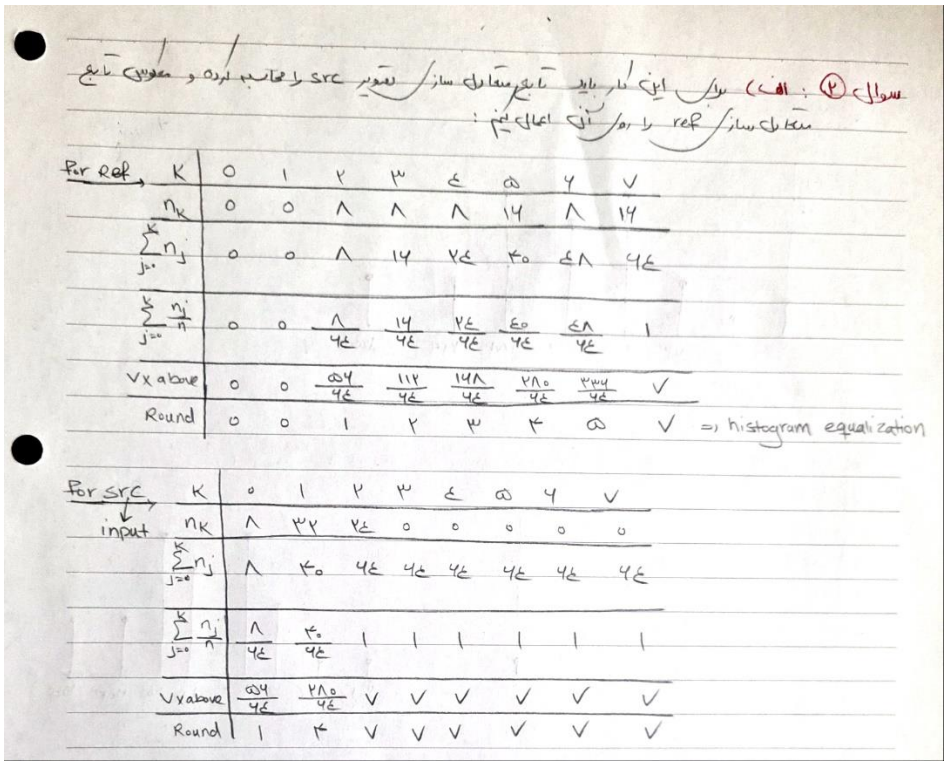
- امکان کنترل بر روی محدوده های شدت خاص در تصویر را فراهم می کند.
- مفید برای برنامه هایی که حفظ مقادیر شدت در یک محدوده خاص مهم است.

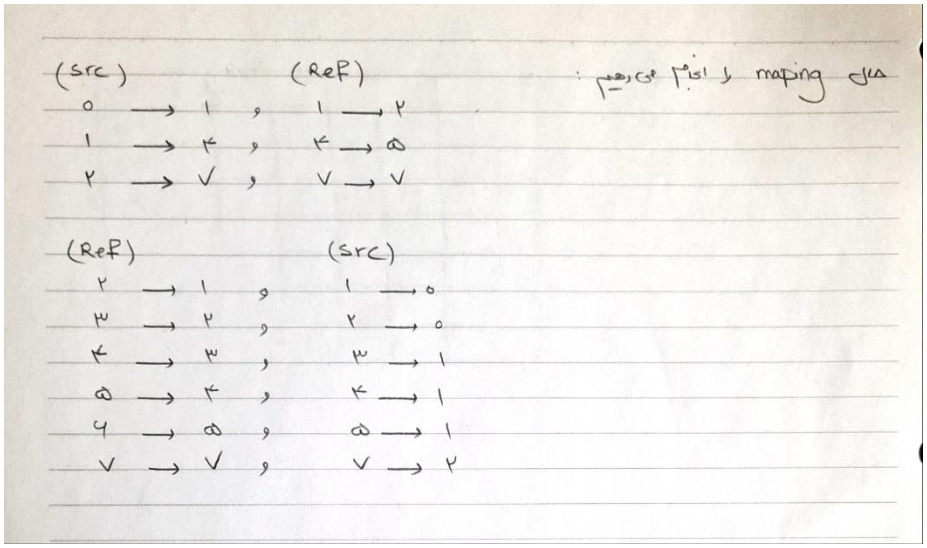
تفاوت:

- کشش هیستوگرام کنتراست کلی و روشنایی تصویر را با استفاده از محدوده دینامیکی کامل افزایش می دهد، در حالی که برش هیستوگرام مقادیر شدت را بدون تغییر کنتراست کلی به محدوده مشخص محدود می کند.
- کشش هیستوگرام تمایل به ایجاد یک تصویر بصری جذاب با کنتراست و جزئیات بهبود یافته دارد، در حالی که برش هیستوگرام برای کنترل محدوده‌های شدت خاص در تصویر بدون تغییر کنتراست کلی مفید است.
- کشش هیستوگرام بر کل توزیع شدت تصویر تأثیر می گذارد، در حالی که برش هیستوگرام به طور انتخابی مقادیر شدت را در محدوده مشخص شده تغییر می دهد.

به طور کلی طبق نتایج به دست آمده از اعمال کشش و برش روی image2، کشش روی این تصویر خیلی مفید نبوده و باعث تغییرات بسیار کم و برجسته شدن بعضی نواحی تصویر شده است. همچنین محدوده برش باعث حذف مقادیر و جزئیات مهمی در تصویر شده است.

سوال 2: الف)





در نهایت تصاویر اینگونه خواهند شد:

New Ref:

1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0
1	2	1	1	0	1	2	0

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

با کمک این لینک:

https://en.wikipedia.org/wiki/Histogram_matching

ب) حال در این بخش به پیاده سازی توابع خواسته شده میپردازیم:

- تابع محاسبه هیستوگرام: تابع Bincount تعداد حضور هر عنصر داخل آرایه را می شمارد که در واقع همان هیستوگرام خواسته شده می‌باشد.

```python
def calc_hist(image):
    '''
    Do not use libraries
    calculate image histogram
    input(s):
      image (ndarray): input image
    output(s):
      hist (ndarray): computed input image histogram
    '''
    hist = np.zeros(256,dtype=int)

    ######################
    #    your code here   #

    #Count number of occurrences of each value in array of non-negative ints.
    hist = np.bincount(image.flatten(), minlength=256)

    ######################

    return hist
```

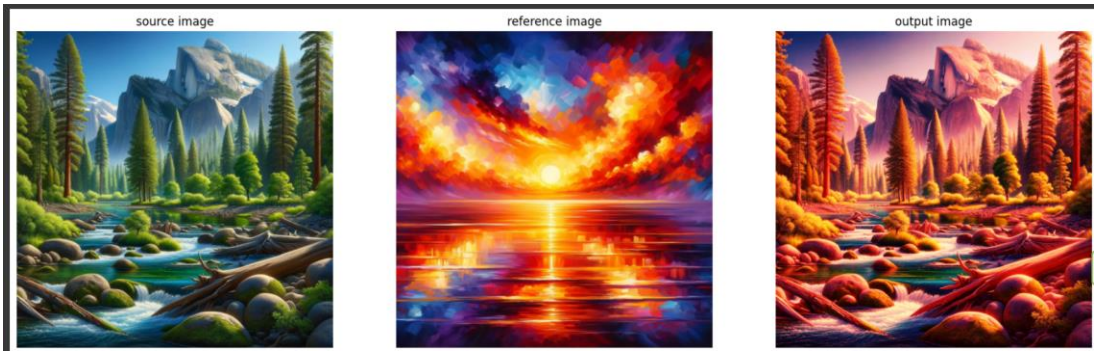
- تابع محاسبه cdf: ابتدا هیستوگرام را با استفاده از تابع قسمت قبل به دست آورده و سپس از تابع cumsum استفاده کردم که جمع عناصر را به صورت انباشته محاسبه می کند. سپس برای اینکه cdf normalize داشته باشیم(رنج بین 0 تا 1)، مقادیر به دست آمده بر بیشینه مقدارشان تقسیم می‌کنیم:

```python
def calc_cdf(channel):
  '''
    Do not use libraries
    calculate image cdf
    input(s):
      channel (ndarray): input image channel
    output(s):
      cdf (ndarray): computed cdf for input image channel
  '''

  ########################
  #    your code here    #

  # Compute the histogram of the image
  hist = calc_hist(channel)
  # Calculate the cumulative sum of the histogram
  cdf = np.cumsum(hist)

  # Normalize the CDF to have values between 0 and 1
  cdf_normalized = cdf / cdf.max()

  ########################

  return cdf_normalized
```

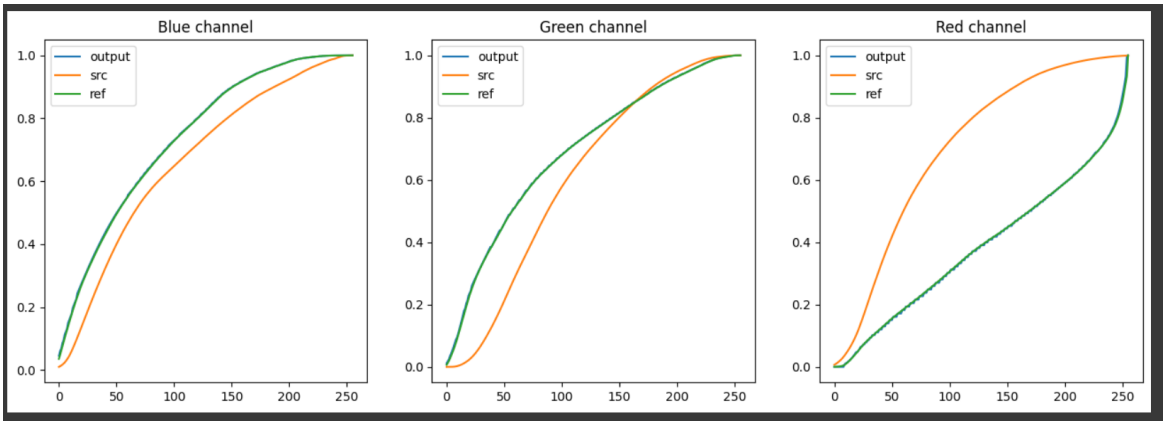
- تابع انطباق هیستوگرام: ابتدا cdf عکس src و ref با توجه به کانالی که در آن هستیم و با استفاده از توابعی که قبلا پیاده سازی کردیم، محاسبه میکنیم و سپس با استفاده از تابع interp که تطبیق خواسته شده را برای ما انجام می‌دهد، تصویر خروجی را تشکیل میدهیم:

```python
def hist_matching(src_image,ref_image):
  '''
  don't use libraries
  input(s):
    src_image (ndarray): source image
    ref_image (ndarray): reference image
  output(s):
    output_image (ndarray): transformation of source image so that its histogram matches histogram of refrence image
  '''
  output_image = src_image.copy()
  channels = [(0, 'Blue channel'), (1, 'Green channel'), (2, 'Red channel')]
  for channel, title in channels:

    ######################
    #    your code here   #

    # compute cdf src given channel
    cdf_src_image = calc_cdf(src_image[:,:,channel])

    # compute cdf ref given channel
    cdf_ref_image = calc_cdf(ref_image[:,:,channel])

    # find piecewise linear interpolant to a function with given Loading... ata points (cdf_ref_image, 0-255), evaluated at cdf_src_image
    res = np.interp(cdf_src_image , cdf_ref_image , np.arange(256))

    # place it in the right channel of output_image
    output_image[:,:,channel] = res[src_image[:,:,channel]]

    ######################

  return output_image
```

حال نتیجه این فرآیند را میبینیم که همان طور که مشاهده میشود تصویر ورودی تا حد زیادی شبیه تصویر رفرنس ما شده است:



اگر هر کانال رنگ را برای دو تصویر داده شده و تصویر خروجی رسم کنیم، خواهیم داشت: (همان طور که مشاهده میشود، نمودار تصویر خروجی به خوبی روی نمودار تصویر رفرنس، منطبق شده است)



سوال 3: الف) ابتدا متعادل سازی هیستوگرام را با استفاده از تابع آماده opencv طراحی میکنیم:

```
1 image = cv2.imread('img_improvment.png',cv2.IMREAD_GRAYSCALE)
2 output_image = image.copy()
3 equalize_image = cv2.equalizeHist(image)
```

که تصویر حاصل شده به شرح زیر است:

چون تصویر اولیه به طور کلی تصویر تیره‌ای می‌باشد، باعث شده تا نواحی که تیره بودند روشن تر شوند و جزئیات بهتر دیده شود. اما روشن کردن تصویر باعث می‌شود نواحی که از ابتدا روشن بوده اند، روشن تر شود و کلی از

جزئیات نواحی دیگر قابل دیدن نیستند و انگار پیکسل های آن اشباع شدند. پس به طور کلی تصویر بهبود نیافته است.

معایب این روش به شرح زیر است:

1. محدود به مقیاس خاکستری: تابع یکسان سازی هیستوگرام OpenCV فقط روی تصاویر در مقیاس خاکستری کار می کند. برای استفاده بر روی تصاویر رنگی باید آنها را به یک فضای رنگی متفاوت (مانند YUV یا HSV) تبدیل کنیم، کانال شدت را یکسان کنیم و سپس دوباره به RGB تبدیل کنیم.
2. یکسان سازی کلی: OpenCV یکسان سازی هیستوگرام کلی را اعمال می کند پس وقتی هیستوگرام تصویر در محدوده کامل توزیع نشده باشد، ممکن است نتایج خوبی به همراه نداشته باشد.
3. از دست دادن جزئیات: یکسان سازی هیستوگرام کنتراست تصویر را افزایش می دهد، اما گاهی اوقات می تواند منجر به از دست دادن جزئیات در مناطقی که کنتراست از قبل بالا است، شود.
4. تقویت بیش از حد نویز: اگر تصویر دارای نویز باشد، یکسان سازی هیستوگرام می تواند این نویز را به خصوص در مناطق نسبتاً همگن بیش از حد تقویت کند.

ب) حال به تعریف توابع خواسته شده میپردازیم:

- متد اول ACE: در این روش تصویر را به نواحی مختلف تقسیم می‌کنیم و هر ناحیه را به طور جداگانه ارتقا می‌دهیم. مشکلی میتواند در نواحی ای باشد که اگر در یکی در تصویر روشن باشد، تابعی که به دست می اوریم تصویر را تیره میکند و کنتراست را افزایش میدهد. اما در ناحیه ای که هم نواحی روشن و هم تیره داریم، به شدت تصویر تیره نمیشود و باعث مشخص شدن نواحی مرزی میشود. در اینجا میتوان کاملا خطوط grid را مشاهده کرد.

```python
def ACE1(image, gridSize):
  '''
  you can use the equalize function of OpenCV for each grid
  Use first method for ACE implementation (calculating transition function for each grid)
  input(s):
    image (ndarray): input image
    gridSize (int): window size for calculating histogram equalization
  output(s):
    output (ndarray): improved image
  '''
  x,y = image.shape
  output_image = image.copy()

  ######################
  #    your code here   #

  # spliting the whole image into parts and improving each part independently
  for i in range(0, x, gridSize):
    for j in range(0, y, gridSize):
      output_image[i:i+gridSize,j:j+gridSize] =cv2.equalizeHist(image[i:i+gridSize,j:j+gridSize])

  ######################

  return output_image
```

به طور کلی مزایا و معایب این روش به شرح زیر است:

مزایا:

- بهبود موضعی: با محاسبه یک تابع انتقال برای هر شبکه، ACE می تواند کنتراست محلی را افزایش دهد، که برای تصاویر با شرایط نوری متفاوت در مناطق مختلف مفید است.
- کارایی: از نظر محاسباتی کارآمدتر از محاسبه تابع انتقال برای هر پیکسل است، زیرا تعداد محاسبات مورد نیاز را کاهش می دهد.

معایب:

- مصنوعات: این روش ممکن است به دلیل تغییرات ناگهانی در تابع انتقال از یک شبکه به شبکه دیگر، مصنوعاتی را در مرزهای شبکه ها معرفی کند.
- حفظ جزئیات کمتر: ممکن است به اندازه روش های هر پیکسل جزئیات را حفظ نکند، به خصوص در مناطقی با تغییرات کنتراست ظریف.

- متد دوم ACE: برای پیاده سازی ACE روش دوم، اول به اندازه نصف gridsize که داریم، ابعاد پدینگ را به دست آورده و سپس با استفاده از copyMakeBorder اطراف تصویر را با حاشیه ای به ابعاد padding و با مقدار 255 پر می‌کنیم. سپس cdf هر خانه را محاسبه کرده و هیستوگرام آن را در می اوریم. همان طور که مشاهده میشود، خروجی عکس به طور کلی بهبود پیدا کرده و کنتراست عکس در نواحی روشن(مثل مجسمه)به خوبی دیده می‌شود و خطوط جدا کننده نواحی عکس

وجود ندارند چون پیکسل به پیکسل جلو رفته ایم. اما مشکل در قسمت هایی می‌باشد که تفاوت پیکسل ها خیلی کم است یعنی نواحی که مقادیر پیکسل هایش بهم نزدیک اند و چشم ما قادر به تشخیص تفاوت کوچکش نیست. در این نواحی به دلیل آن کششی که در هیستوگرامش به دست می‌آید، پیکسل های خیلی روشن یا خیلی تیره در آن نواحی خواهیم داشت که باعث افزایش نویزها میشود.

```python
def ACE2(image, gridSize):
    '''
    you can just use the equalize function of OpenCV for each grid
    You can use OpenCV built-in tools for applying padding
    Use second method for ACE implementation (calculating transition function for each pixel)
    input(s):
      image (ndarray): input image
      gridSize (tuple): window size for calculating histogram equalization
    output(s):
      output (ndarray): improved image
    '''
    output = image.copy()

    ######################
    #    your code here   #

    x,y = image.shape

    # tuple for padding
    padding = (int(gridSize[0]/2) , int(gridSize[1]/2))

    # completing the around of the picture with that padding and giving the value of 255 to it
    image = cv2.copyMakeBorder(image,padding[0], padding[0], padding[1], padding[1], cv2.BORDER_CONSTANT,value=255)

    for i in range(padding[0], x+padding[0]):
      for j in range(padding[1], y+padding[1]):
        # calculating the histogram
        hist = np.bincount(image[i-padding[0]:i+padding[0]+1,j-padding[1]:j+padding[1]+1].flatten(), minlength=256)

        # calculating CDF and normalizing it
        cdf = np.cumsum(hist)
        cdf = cdf / cdf.max()

        # put the result in the range of 256
        output[i-padding[0], j-padding[1]] = cdf[image[i, j]] * 255

    ######################

    return output
```

به طور کلی مزایا و معایب این روش به شرح زیر است:

مزایا:

- حفظ جزئیات: محاسبه یک تابع انتقال برای هر پیکسل امکان کنترل بسیار خوب بر افزایش کنتراست را فراهم می کند و جزئیات بیشتری را در تصویر حفظ می کند.
- انتقال هموار: این روش از مصنوعات مرزی شبکه ای که در رویکرد مبتنی بر شبکه دیده می شود اجتناب می کند و در مقابل، منجر به انتقال روان تر می شود.

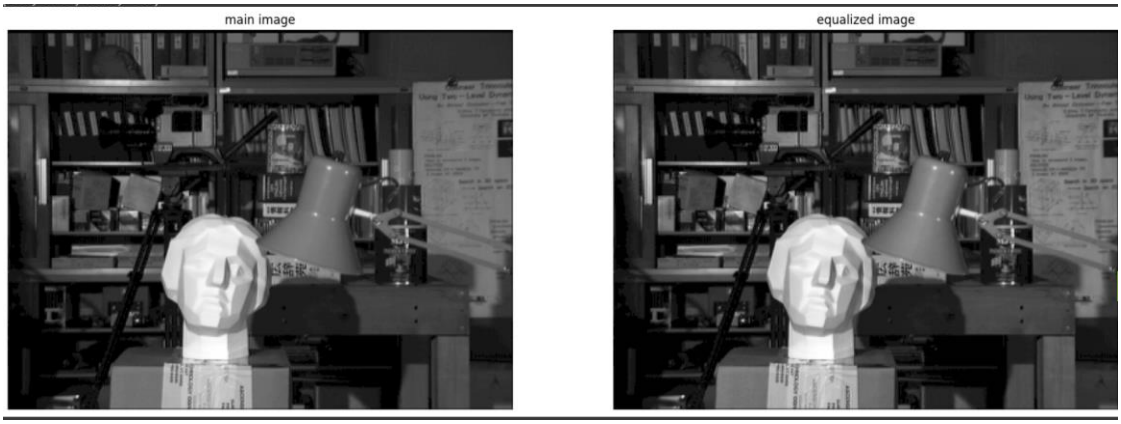
معایب:

- هزینه محاسباتی: به طور قابل توجهی از نظر محاسباتی فشرده تر است، زیرا نیاز به یک تابع انتقال برای هر پیکسل دارد.
- تقویت نویز: خطر افزایش نویز بیشتر است زیرا این روش در سطح پیکسل عمل می کند و به طور بالقوه نویز را همراه با جزئیات تصویر افزایش می دهد.

- روش CLAHE: برای پیاده سازی آن ابتدا padding اطراف تصویر را به اندازه نصف سایز grid به صورت یک تاپل به دست می‌آوریم سپس با استفاده از copyMakeBorder ، دور تصویر اولیه padding با مقدار 255 می‌گذاریم. سپس دوباره هیستوگرام آن را به دست می‌آوریم. در این روش با استفاده از تابع clip مقادیری که بیشتر از حد تعریف شده بودند را با خود clip_limit جایگزین می‌کنیم و آن ها را به هیستوگرام خود اضافه کرده و cdf نرمالایز آن را محاسبه می‌کنیم و آن را در رنج 0 تا 255 درآورده و در عکس خروجی قرار می‌دهیم.

```python
def CLAHE(image, gridSize, clip_limit):
    '''
    you can just use opencv library for calculate histogram and applying padding
    you can't use the equalize function of opencv
    Use second method for ACE implementation (calculating transition function for each pixel)
    input(s):
      image (ndarray): input image
      gridSize (tuple): window size for calculating histogram equalization
      clip_limit (int): threshold for contrast limiting
    output(s):
      output (ndarray): improved image
    '''
    output = image.copy()

    ######################
    #    your code here   #

    x,y = image.shape

    # tuple for padding
    padding = (int(gridSize[0]/2) , int(gridSize[1]/2))

    # completing the around of the picture with that padding and giving the value of 255 to it
    image = cv2.copyMakeBorder(image,padding[0],padding[0],padding[1],padding[1],cv2.BORDER_CONSTANT,value=255)

    for i in range(padding[0],x+padding[0]):
      for j in range(padding[1],y+padding[1]):
        # calculating the histogram
        hist = np.bincount(image[i-padding[0]:i+padding[0]+1, j-padding[1]:j+padding[1]+1].flatten(), minlength=256)

```

```
    for i in range(padding[0],x+padding[0]):
      for j in range(padding[1],y+padding[1]):
        # calculating the histogram
        hist = np.bincount(image[i-padding[0]:i+padding[0]+1, j-padding[1]:j+padding[1]+1].flatten(), minlength=256)

        # cliping(removing the values higher than clip_limit)
        hist = np.clip(hist, 0, clip_limit)

        # adding them to the histogram
        hist = hist + clip_limit

        # calculating CDF and normalizing it
        cdf = np.cumsum(hist)
        cdf = cdf / cdf.max()

        # put the result in the range of 256
        output[i-padding[0], j-padding[1]] = cdf[image[i,j]] * 255

    ######################

    return output
```



به طور کلی مزایا و معایب این روش به شرح زیر است:

مزایا:

- محدود کردن کنتراست: CLAHE شامل یک مرحله محدود کننده کنتراست است که از تقویت بیش از حد نویز که در تکنیک‌های یکسان سازی هیستوگرام استاندارد رایج است، جلوگیری می‌کند.
- تقویت موضعی با مصنوعات کاهش یافته: کنتراست را به صورت محلی مانند ACE افزایش می دهد، اما معمولاً شامل درون یابی بین شبکه ها برای کاهش مصنوعات می شود.

معایب:

- حساسیت پارامتر: نتایج CLAHE به انتخاب پارامترهایی مانند اندازه شبکه و محدودیت کلیپ حساس هستند، که ممکن است برای نتایج بهینه نیاز به تنظیم دقیق داشته باشد.
- افزایش پیچیدگی: الگوریتم به دلیل مراحل اضافی برش و توزیع مجدد هیستوگرام، پیچیده تر از یکسان سازی هیستوگرام استاندارد است.

ج) در این بخش میخواهیم روش CLAHE را با OpenCV پیاده سازی کنیم و از تابع creatCLAHE که دو ورودی clip_limit و سایز فیلتر را میگیرد و apply استفاده کردیم:

```python
def CLAHE(image, gridSize, clipLimit):
    '''
    use opencv library for CLAHE.
    input(s):
      image (ndarray): input image
      gridSize (tuple): window size for calculating histogram equalization
      clip_limit (int): threshold for contrast limiting
    output(s):
      output (ndarray): improved image
    '''

    # Convert image to grayscale if it's a color image
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Create CLAHE object
    clahe = cv2.createCLAHE(clipLimit=clipLimit, tileGridSize=gridSize)

    # Apply CLAHE to the image
    clahe_output = clahe.apply(image)

    return clahe_output
```

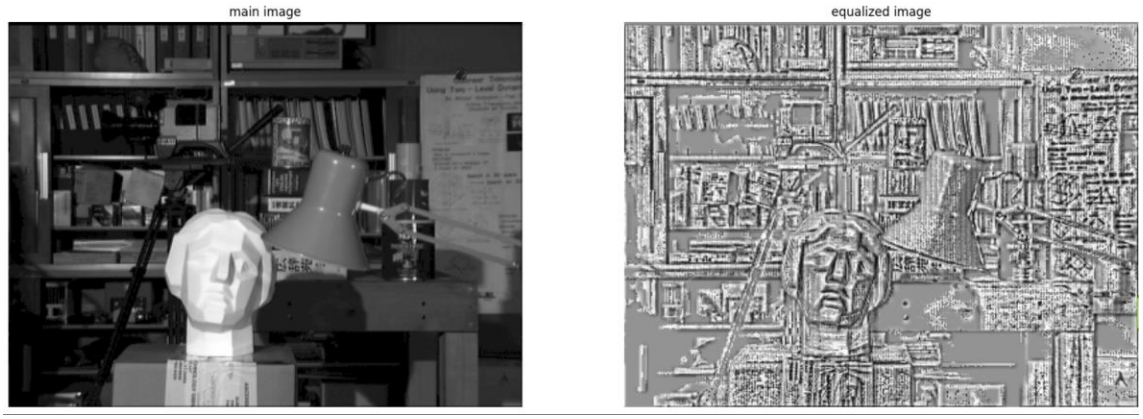
در این روش، دو پارمتر مهم که در راستای پیشرفت کیفیت عکس نقش دارند، وجود دارد:

1. سایز فیلتر: مشخص می‌کند که تصویر به چه اندازه‌ای باید به بلوک‌های کوچکتر تقسیم شود. سایز بلوک‌های کوچکتر باعث می‌شود توزیع پیکسل‌ها در هر بلوک به شکل همگن‌تری تبدیل شود. برای تصاویر با رزولوشن بالا، سایز بلوک باید بزرگتر انتخاب شود تا عملیات بر روی تصویر سریع‌تر صورت بگیرد. اما برای تصاویر با رزولوشن پایین، باید سایز بلوک کوچکتری انتخاب شود تا جزئیات بیشتری حفظ شود.
2. Clip limit: پارامتری است که تعیین می‌کند که در هر بلوک، چه تعداد از پیکسل‌ها باید به مقدار بالایی محدود شوند. با افزایش این پارامتر، مقدار بیشتری از پیکسل‌ها در هر بلوک به مقدار بالایی محدود خواهند شد و تغییرات توزیع پیکسل‌ها به شدت کاهش خواهد یافت. اما با کاهش این پارامتر، تصویر با جزئیات بیشتری بهبود پیدا خواهد کرد اما خطر افزایش نویز نیز وجود دارد.

حال خروجی 4 حالت گفته شده را به ترتیب نمایش میدهیم:

1. ابعاد پنجره 128*128 و حد برش 2:

2. ابعاد پنجره 128*128 و حد برش 128:



3. ابعاد پنجره 16*16 و حد برش 2:

4. ابعاد پنجره 16*16 و حد برش 128:

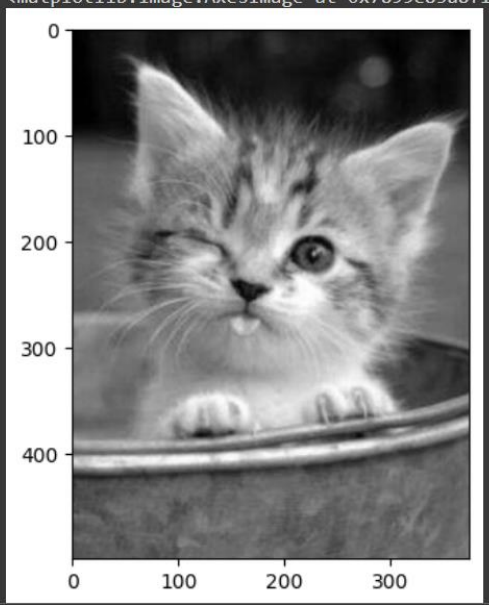
که همان طور که مشاهده میشود بهترین گزینه، ابعاد پنجره 16*16 و حد برش 2 میباشد.

سوال 4: الف) در این بخش از ما خواسته شده تا به تصویر ورودی نویز نمک و فلفل که نویزی غیر جمع شونده است، اضافه کنیم. ابتدا عکس اصلی را قبل از اعمال تغییرات نمایش میدهیم:

```
# Do not change this cell
image = cv2.imread('cat.png',cv2.IMREAD_GRAYSCALE)
```

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

```
<matplotlib.image.AxesImage at 0x7899e83a8f10>
```



حال تابع Add_Noise را تکمیل میکنیم:

- همان طور که در کد مشاهده میشود، 30 درصد از کل پیکسل ها برای نویز انتخاب شده اند.
- نصف پیکسل های نویزی، نمک و نصف دیگر آنها فلفل هستند.
- مقدار نویز نمک، 255 و مقدار نویز فلفل 0 میباشد.

```python
def Add_Noise(img):
    """
    Add salt and pepper noise to the input image.
        Parameters:
          img: Input image (numpy array).
        Returns:
          Image with salt and pepper noise added.
    """

    # Determine the number of pixels to be affected by salt and pepper noise
    num_pixels = img.size

    # Use 30% of the total number of pixels as the noise level
    noise_level = 0.3
    num_salt = int(num_pixels * noise_level / 2)  # Half of the noise level for salt
    num_pepper = int(num_pixels * noise_level / 2)  # Half of the noise level for pepper

    # Generate random coordinates for salt noise
    salt_coords = [np.random.randint(0, i - 1, num_salt) for i in img.shape]

    # Set the pixel values at the salt coordinates to maximum intensity (255)
    img[salt_coords[0], salt_coords[1]] = 255

    # Generate random coordinates for pepper noise
    pepper_coords = [np.random.randint(0, i - 1, num_pepper) for i in img.shape]

    # Set the pixel values at the pepper coordinates to minimum intensity (0)
    img[pepper_coords[0], pepper_coords[1]] = 0

    return img
```

حال تصویر را پس از اعمال نویز گفته شده، نمایش میدهیم(همان طور که مشاهده شده است، بسیاری از جزئیات دیگر قابل تشخیص نمی باشند):

```
1 image = Add_Noise(image)
2 plt.imshow(image,cmap='gray')
3 plt.title("cat with noise")
4 plt.axis("off")
```

```
(-0.5, 374.5, 499.5, -0.5)
```

ب) در این بخش میخواهیم سه فیلتر هموارساز متوسط گیر، میانه و گاوسی را طراحی کنیم. ابتدا تابع Reflect101 که یک تابع پدینگ است را طراحی میکنیم. در این تابع همان طور که مشاهده میشود ابتدا عرض پدینگ را مشخص کردیم(که برابر با نصف سایز فیلتر گذاشتیم چون تضمین می کند که وقتی فیلتر روی هر پیکسل از تصویر اصلی متمرکز می شود، به طور یکسان در همه جهات گسترش می یابد و کل ناحیه فیلتر را بدون خارج شدن از مرزهای تصویر پد شده پوشش می دهد) و سپس با استفاده از تابع pad از کتابخانه numpy، به 4 جهت پدینگ اضافه کردیم:

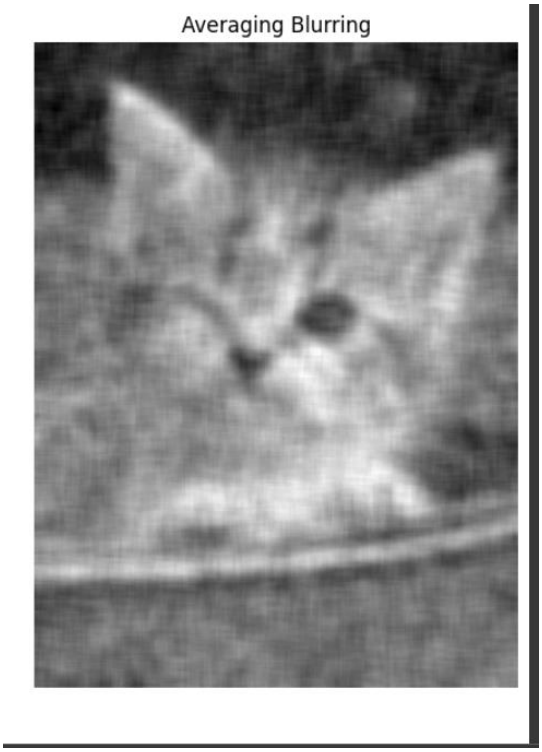
```python
def Reflect101(img,filter_size):
    '''
      Do not use loop (like while and for)
      Do not use libraries
      calculate averaging filter
      input(s):
        img (ndarray): input image
        filter_size (ndarray): filter size
      output(s):
        image (ndarray): computed Reflect101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

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

    return reflected_img
```

در ادامه نوبت طراحی توابع فیلترها میباشد:

1. فیلتر متوسط گیر: این فیلتر بین مقادیر میان گرفته و مقدار جدید را جایگزین میکند. در کد زیر ابتدا متغیر کرنل تعریف شد که برای تعریف درست همسایگی در عملیات کانولوشن است. سپس مقادیر همسایگی ها طبق سایز فیلتر با هم جمع شده و در کرنل تعریف شده ضرب میشوند(چون در تعریف کرنل، آن را تقسیم بر 2 برابر سایز فیلتر کردیم، میانگین گرفته میشود):

```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   #

    # Calculate the kernel by dividing each element by the total number of elements in the kernel
    kernel = np.ones((filter_size, filter_size)) / (filter_size ** 2)

    # Convolve the image with the kernel
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            result[i, j] = np.sum(image[i:i+filter_size, j:j+filter_size] * kernel)

    return result
    ######################

    return result
```

که نتیجه آن در تصویر نویزی داده شده به شرح زیر است:



2. فیلتر میانه: در این فیلتر میانه بین همسایگی ها گرفته میشود. در کد زیر طبق سایز فیلتر گفته شده، بین مقادیر میانه گرفته شد و در result جایگزین شد:

```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   #
  image = np.array(image)
  # Convolve the image with the filter
  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
  ########################

  return result
```

که نتیجه آن در تصویر نویزی داده شده به شرح زیر است:

3. فیلتر گاوسی: در این فیلتر مانند فیلتر متوسط گیر جمع فیلترهای همسایه محاسبه میشود و سپس در کرنل تعریف شده ضرب میشود. اما در این سوال، کرنل طبق فرمول گاوسی که به شرح زیر است، تشکیل شده است:

$$G(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

```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   #
    m = filter_size
    n = filter_size
    for x in range(-m // 2, m // 2 + 1):
        for y in range(-n // 2, n // 2 + 1):
            kernel[x + m // 2, y + n // 2] = np.exp(-(x ** 2 + y ** 2) / (2 * std ** 2))
    kernel /= np.sum(kernel)

    # Apply Reflect101 padding to the image
    image = Reflect101(img, filter_size)
    image = np.array(image)
    result = np.zeros_like(img)

    # Convolve the image with the kernel
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            result[i, j] = np.sum(image[i:i+m, j:j+n] * kernel)
    ######################
    output = result.copy()
    result = cv2.filter2D(src = output, ddepth = -1, kernel = kernel)
    return result
```

که نتیجه آن در تصویر نویزی داده شده به شرح زیر است:

حال اگر بخواهیم تاثیر سایز فیلتر بر نتیجه را بگوییم، به طور کلی، افزایش اندازه هسته منجر به تاری تهاجمی تر می شود، در حالی که کاهش آن منجر به تاری کمتر و حفظ بیشتر جزئیات می شود.

تأثیرگذاری سایز فیلتر بر نتیجه هریک از فیلترها به شرح زیر است:

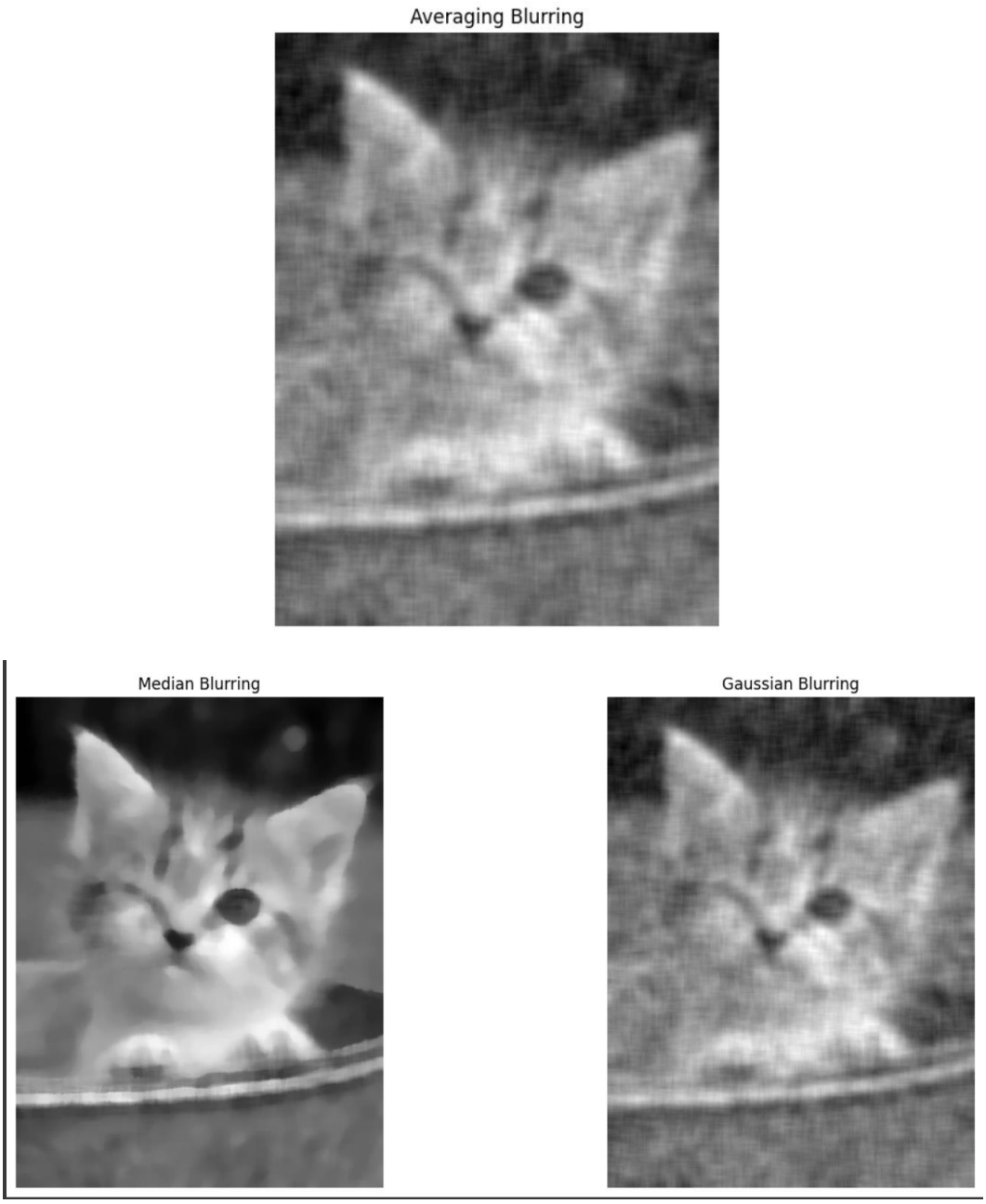
1. فیلتر متوسط گیر: با افزایش اندازه هسته پیکسل های مجاور بیشتری در فرآیند میانگین گیری گنجانده میشوند و اثر تاری بیشتر میشود که باعث حذف جزئیات کوچک در تصویر صاف و از دست دادن وضوح میشود. برعکس، کاهش اندازه هسته اثر تاری را کاهش می دهد و جزئیات دقیق تری را در تصویر حفظ می کند.

2. فیلتر میانه: مشابه فیلتر میانگین، افزایش اندازه هسته در فیلتر میانه منجر به صاف کردن تهاجمی تر می شود که می تواند در کاهش نویز به خصوص نویز نمک و فلفل کمک کند. با این حال، اندازه هسته بیش از حد بزرگ می تواند باعث از بین رفتن جزئیات و لبه های ریز در تصویر شود. اندازه هسته کوچکتر باعث ایجاد یکنواختی کمتر و حفظ جزئیات میشود.
3. فیلتر گاوسی: افزایش اندازه هسته در آن منجر به تاری قوی تر می شود و ویژگی های نویز و تصویر را صاف می کند همچنین اجزای با فرکانس بالا را در تصویر کاهش می دهد. برعکس، کاهش اندازه هسته اثر تاری را کاهش می دهد و جزئیات بیشتر تصویر را حفظ می کند.

ج) حال سه فیلتر گفته شده در مرحله قبل را با استفاده از توابع آماده OpenCV طراحی کرده و نتیجه هر یک را نمایش میدهیم:

```
# Perform Averaging Blurring
AveragingBlurring = cv2.blur(image, (15, 15))

# Perform Median Blurring
MedianBlurring = cv2.medianBlur(image, 15)

# Perform Gaussian Blurring
GaussianBlurring = cv2.GaussianBlur(image, (15, 15), 40)
```

نتایج:



که همان طور که مشاهده میشود، نتایج به نتایج به دست آمده از قسمت "ب" همخوانی دارند و پیاده سازی ها درست میباشند.

سوال 5: همان طور که میدانیم لاپلاسین تغییرات شدت روشنایی را برجسته میکند. همچنین تقویت پیکسل هایی که تغییرات دارند موجب تیز شدن تصویر میشود. به این منظور، از کرنل زیر برای sharpening استفاده شده است:

-1	-1	-1
-1	9	-1
-1	-1	-1

از آنجایی که در صورت سوال حرفی از پدینگ زده نشد، پدینگی در نظر نمیگیریم. حال باید تصویر داده شده را با کرنل مورد نظر، کانوالو کنیم که حاصل این عملیات به شرح زیر میشود:

10	10	10	10	10	10
10	10	10	10	10	10
10	10	8	8	8	10
10	10	8	28	8	10
10	10	8	8	8	10
10	10	10	10	10	10

سوال 6:

سوال ④ : همان طور که می دانیم فرمول تبدیل فوریه به شرح زیر است :

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

که در اینجا $M=N=2$ می باشد ، حال که

$$f = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

می باشد ، محاسبات را انجام می دهیم :

$$1- F(0,0) = \sum_{x=0}^{1} \sum_{y=0}^{1} f(x,y) . \cancel{e^{0}}^{\,1} = 1+2+2+1 = \textcircled{4}$$

$$2- F(0,1) = \sum_{x=0}^{1} \sum_{y=0}^{1} f(x,y) . e^{-yj\pi} \quad \xrightarrow{if\ y=0} = 1+2 = 3 \quad \xrightarrow{if\ y=1} = 3 \times e^{-j\pi}$$

$$\xrightarrow{so} = 3 + 3e^{-j\pi} = 3 + 3 \times (\underbrace{\cos\pi - j\sin\pi}_{=-1}) = \textcircled{0}$$

$$3- F(1,0) = \sum_{x=0}^{1} \sum_{y=0}^{1} f(x,y) . e^{-xj\pi} \quad \xrightarrow{if\ x=0} = 1+2 = 3 \quad \xrightarrow{if\ x=1} = 3 \times e^{-j\pi}$$

$$\xrightarrow{\text{عین قسمت ۲}} = 3 + (3 \times -1) = \textcircled{0}$$

$$4- F(1,1) = \sum_{x=0}^{1} \sum_{y=0}^{1} f(x,y) . e^{-j\pi(x+y)}$$

$$\xrightarrow{if\ x=y=0} = 1 \ , \quad \xrightarrow{if\ x=0, y=1\ or\ x=1, y=0} = 2 \times e^{-j\pi} + 2 \times e^{-j\pi} = 4 \times e^{-j\pi}$$

$$\xrightarrow{if\ x=y=1} = 1 \times e^{-2j\pi}$$

$$\xrightarrow{so} = 1 + 4 \times \underbrace{e^{-j\pi}}_{=-1} + e^{-2j\pi} = 1 - 4 + \overbrace{\cos 2\pi}^{1} - j \overbrace{\sin 2\pi}^{0}$$

$$= \textcircled{-2}$$

⇐ پس در نهایت ماتریس فوریه به این صورت خواهد بود :

$$\begin{bmatrix} 4 & 0 \\ 0 & -2 \end{bmatrix}$$