به نام خدا

تمرین دوم – بینایی کامپیوتر **نیایش خانی ۹۹۵۲۱۲۳۵**

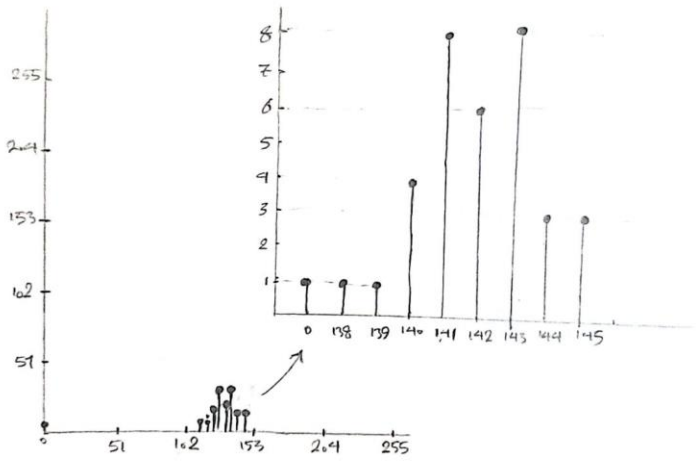
سوال ۱) الف) برای محاسبه هیستوگرام یک تصویر کافیست تعداد پیکسل هایی که دارای یک سطح روشنایی هستند را بدست آوریم. هیستوگرام با فرمول زیر تعریف می شود :

$$h(r_k) = n_k$$

بنابراین داریم:

$$h(0) = 1\ ,\ h(138) = 1\ ,\ h(139) = 1\ ,\ h(140) = 4\ ,\ h(141) = 8\ ,\ h(142) = 6\ ,$$
$$h(143) = 8\ ,\ h(144) = 3\ ,\ h(145) = 3$$

نمودار هیستوگرام :



برای اعمال کشش هیستوگرام از فرمول زیر استفاده می کنیم :

$$stretch[f(x,y)] = \left(\frac{f(x,y) - f_{min}}{f_{max} - f_{min}}\right)(MAX - MIN) + MIN \qquad MIN = 0\ , MAX = 255$$

بنابراین خواهیم داشت :

$$stretch(0) = \frac{0-0}{145-0} \times 255 = 0$$

$$stretch(138) = \frac{138-0}{145-0} \times 255 \approx 242$$

$$stretch(139) = \frac{139-0}{145-0} \times 255 \approx 244$$

$$stretch(140) = \frac{140-0}{145-0} \times 255 \approx 246$$

$$stretch(141) = \frac{141-0}{145-0} \times 255 \approx 247$$

$$stretch(142) = \frac{142-0}{145-0} \times 255 \approx 250$$

$$stretch(143) = \frac{143-0}{145-0} \times 255 \approx 251$$

$$stretch(144) = \frac{144-0}{145-0} \times 255 \approx 253$$

$$stretch(145) = \frac{145-0}{145-0} \times 255 \approx 255$$

تصویر حاصل پس از کشش هیستوگرام به شکل زیر می شود :

[247, 0 , 242, 251, 251, 251, 253]

[247, 246, 246, 250, 250, 251, 251]

[246, 255, 255, 253, 250, 250, 255]

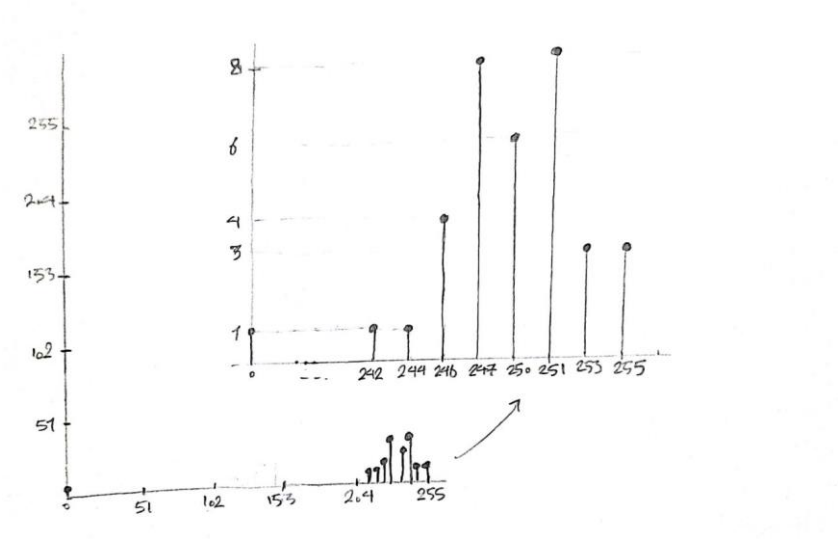
[247, 247, 247, 251, 250, 247, 251]

[244, 246, 247, 247, 250, 251, 253]

برای محاسبه هیستوگرام خواهیم داشت :

$h(0) = 1$, $h(242) = 1$, $h(244) = 1$, $h(246) = 4$, $h(247) = 8$, $h(250) = 6$, $h(251) = 8$, $h(253) = 3$, $h(255) = 3$

نمودار هیستوگرام :



برای اعمال برش هیستوگرام از فرمول زیر استفاده می کنیم :

$$clip[f(x,y)] = \left(\frac{f(x,y) - f_1}{f_{99} - f_1}\right)(MAX - MIN) + MIN \qquad f(x,y) < 138 = 0\,, f(x,y) > 145 = 255$$

بنابراین خواهیم داشت :

$clip(0) = 0$

$clip(138) = 0$

$clip(139) = \frac{139-138}{145-138} \times 255 \approx 36$

$clip(140) = \frac{140-138}{145-138} \times 255 \approx 72$

$clip(141) = \frac{141-138}{145-138} \times 255 \approx 109$

$clip(142) = \frac{142-138}{145-138} \times 255 \approx 145$

$clip(143) = \frac{143-138}{145-138} \times 255 \approx 182$ $clip(144) = \frac{144-138}{145-138} \times 255 \approx 218$

$clip(145) = \frac{145-138}{145-138} \times 255 \approx 255$

برای محاسبه هیستوگرام پس از برش خواهیم داشت :

$h(0) = 2$, $h(36) = 1$, $h(72) = 4$, $h(109) = 8$, $h(145) = 6$, $h(182) = 8$, $h(218) = 3$, $h(255) = 3$

بنابراین تصویر بعد از برش به این شکل خواهد شد:

[109, 0 , 0 , 182, 182, 182, 218]

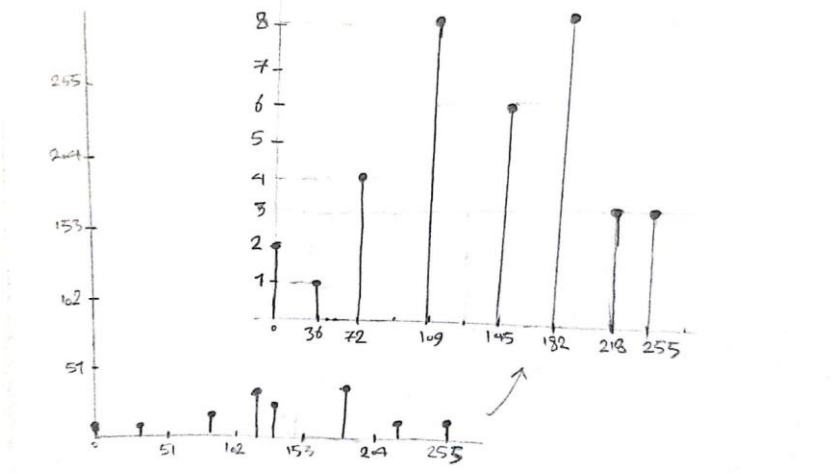
[109, 72 , 72 , 145, 145, 182, 182]

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

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

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

نمودار هیستوگرام پس از برش:



ب) ابتدا تصویر خود را به صورت زیر تعریف می کنیم.

```
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]
])
```

سپس با استفاده از کد زیر هیستوگرام تصویر را محاسبه می کنیم.

```
def calc_hist(image):
    # Flatten the image into 1 dimension: row after row
    flat_image = image.flatten()
    # Calculate histogram
    hist = np.bincount(flat_image, minlength=256)
    return(hist)
```

ابتدا با استفاده از تابع flatten() ماتریسی که از تصویر خود در اختیار داریم را به صورت ردیف هایی پشت سر هم یک بعدی می کنیم. سپس تابع np.bincount آرایه ای را برمی گرداند که در آن مقدار شاخص i تعداد دفعاتی است که i در تصویر ظاهر شده است.

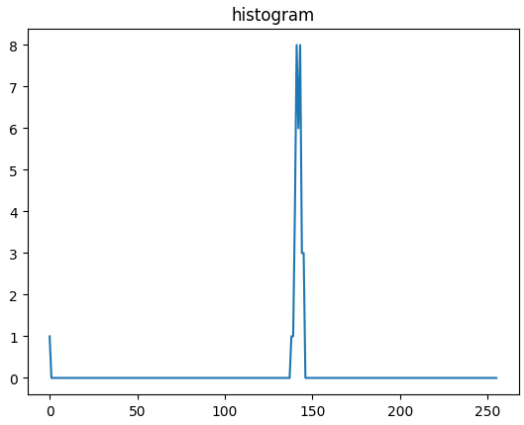


Figure ۱ - هیستوگرام

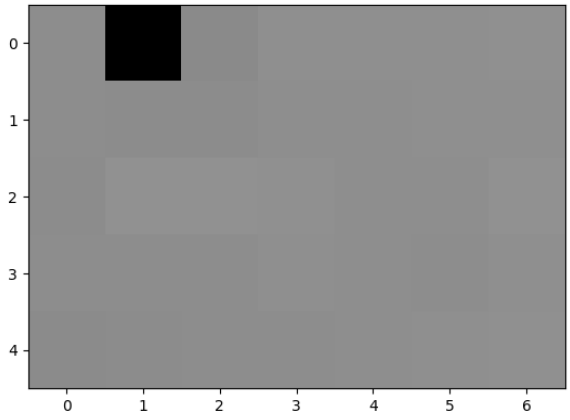


Figure ۲ - تصویر خروجی

حال با استفاده از کد زیر کشش هیستگرام را پیاده سازی می کنیم.

```python
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()
  min_val = min([min(row) for row in image])
  max_val = max([max(row) for row in image])

  for i in range(len(image)):
    for j in range(len(image[0])):
      output_image[i][j] = (image[i][j] - min_val) * 255 / (max_val - min_val)
  return output_image
```

ابتدا کمترین و بیشترین مقدار پیکسل موجود در تصویر را پیدا می کنیم. سپس به ازای تمامی پیکسل های تصویر فرمول کشش هیستوگرام که در قسمت قبل ذکر شده است را اعمال می کنیم

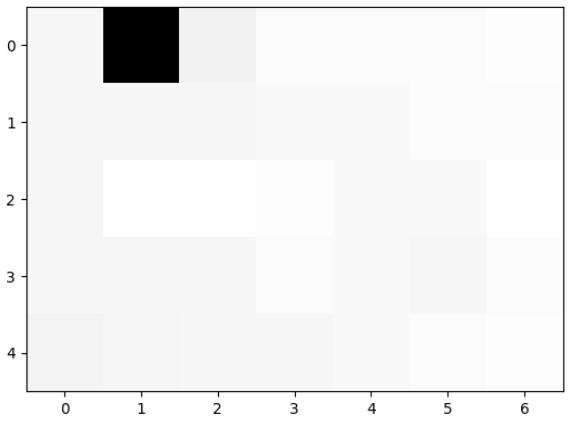


Figure ٤- تصویر خروجی پس از کشش

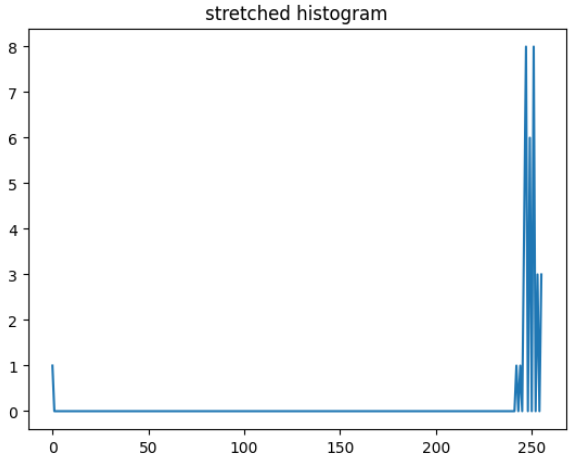


Figure ٣ - کشش هیستوگرام

بخش بعدی اعمال Clipping است که با کد زیر پیاده سازی شده است.

```
def clip_hist(image, min_value, max_value):
    output_image = image.copy()
    # Traverse all pixels of the input image
    for i in range(len(image)):
        for j in range(len(image[0])):
            # Clip pixel values below min_value to 0
            if image[i, j] < min_value:
                output_image[i, j] = 0
            # Clip pixel values above max_value to 255
            elif image[i, j] > max_value:
                output_image[i, j] = 255
            # Perform linear scaling for pixel values within the range
            else:
                output_image[i, j] = int((image[i, j] - min_value) / (max_value -
min_value) * 255)

    # Return the resulting clipped image
    return output_image
```

برای اعمال برش هیستوگرام به ازای هر کدام از پیکسل های تصویر سه شرط را چک می کنیم :

۱. اگر پیکسل مورد نظر کمتر از کمترین مقدار بود، آن را صفر قرار می دهیم.
۲. اگر پیکسل مورد نظر بیشتر از بیشترین مقدار بود، آن را ۲۵۵ قرار می دهیم.
۳. در غیر این دو صورت، فرمول clipping که در بخش قبلی ذکر شده است را روی آن اعمال می کنیم.

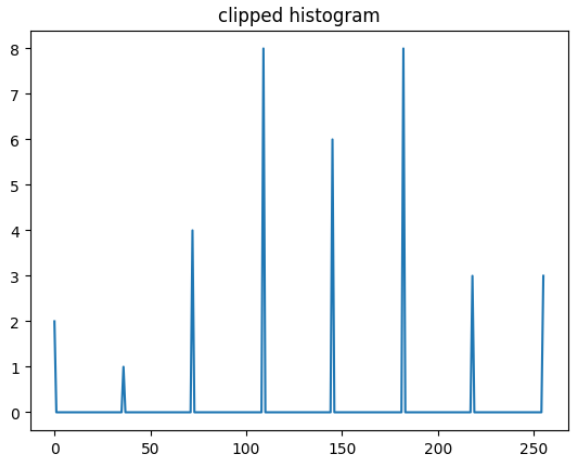


Figure ۵ - برش هیستوگرام

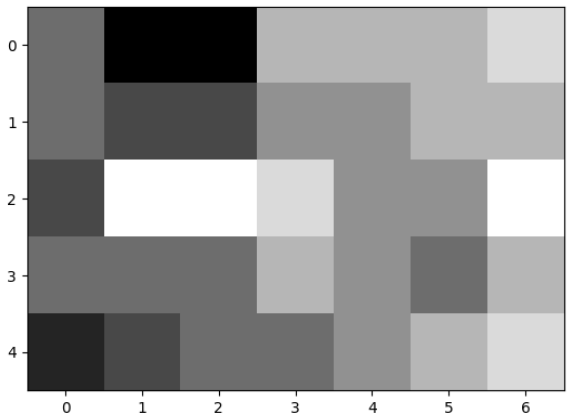


Figure ۶ - تصویر خروجی بعد از برش

ج) با استفاده از دستور تصویر را خوانده و آن را نمایش می دهیم. (از plt.axix برای از بین بردن محور های مختصات استفاده شده است.)

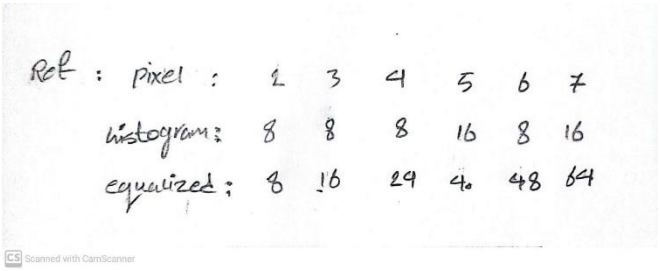
```
# first read the image and show it.(image2)
image2 = cv2.imread('image2.png', cv2.IMREAD_GRAYSCALE)
plt.axis('off')
plt.imshow(image2, cmap='gray', vmin=0, vmax=255)
plt.title('Original Image')
```

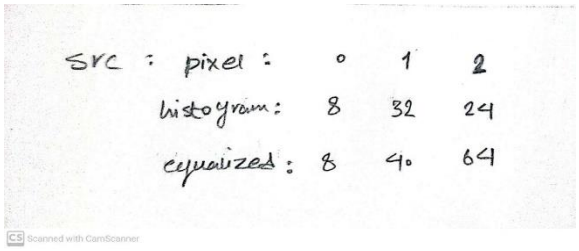
در کشش هیستوگرام، مقادیر پیکسلی تصویر به طور خطی مقیاس داده می‌شوند تا مقادیر تمامی هیستوگرام در یک بازه جدید مشخص شده (معمولاً ۰ تا ۲۵۵) قرار گیرند. این باعث می‌شود یک توزیع متوازن تر از مقادیر پیکسل به دست آید. همچنین می‌تواند منجر به افزایش کنتراست و بهبود دیداری تصویر شود، اما ممکن است جزئیاتی از تصویر از دست برود. همانطور که در تصویر نیز مشاهده می شود، خروجی روشن تر شده است که این نتیجه کشش هیستوگرام است.

برش هیستوگرام، از سوی دیگر، یک فرآیند است که توزیع شدت پیکسل تصویر را تغییر می دهد. در کلیپ کردن هیستوگرام، مقادیر پیکسلی که خارج از محدوده‌ی مشخص شده‌ای (بین *min_value* و *max_value*) قرار دارند، به حداقل یا حداکثر مقدار ممکن برای محدوده جدید تغییر می‌یابند. به عبارت دیگر، تمامی مقادیر پیکسلی که کمتر از *min_value* هستند، به *min_value* تنظیم می‌شوند و مقادیری که بیشتر از *max_value* هستند، به *max_value* تنظیم می‌شوند. کلیپ کردن معمولاً برای حفظ جزئیات مهم در تصویر، مانند لبه‌ها یا نقاط روشن و تاریک، استفاده می‌شود.

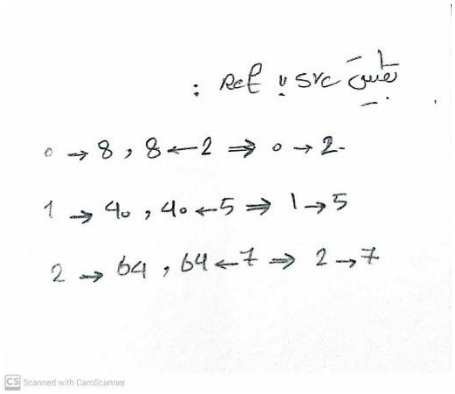
هر دو اینها روش هایی هستند که برای بهبود کنتراست تصویر استفاده می شوند، اما به روش های مختلف این کار را انجام می دهند و می توانند تاثیرات مختلفی بر تصویر داشته باشند. کشیدن هیستوگرام به طور کلی کنتراست کلی تصویر را با استفاده از کل محدوده شدت پیکسل ها بهبود می بخشد، در حالی که برش هیستوگرام می تواند کنتراست در مناطق خاصی از تصویر را در حالی که به نفع مناطق دیگر است، بهبود بخشد.

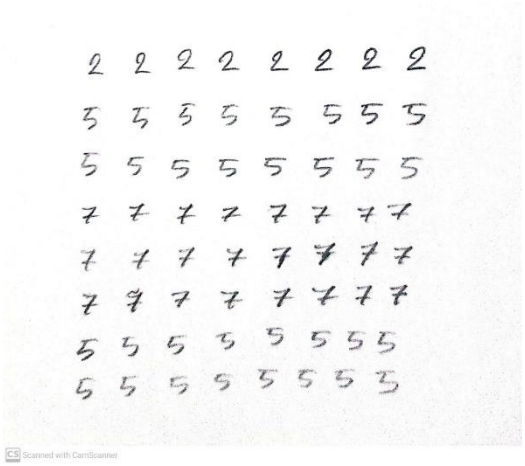
سوال ۲) الف) برای تطبیق هیستوگرام ابتدا هیستوگرام دو تصویر را محاسبه می کنیم.





سپس src را بر اساس Ref تطبیق می دهیم و تصویر زیر حاصل می شود.





ب) برای محاسبه هیستوگرام از کد زیر استفاده شده است.

```
def calc_hist(image):
    hist = np.zeros(256, dtype=int)
    # Iterate through each pixel in the image and update the histogram
    for i in range(len(image)):
        for j in range(len(image[0])):
            intensity = image[i, j]
            hist[intensity] += 1

    return hist
```

ابتدا یک آرایه به طول ۲۵۶ می سازیم که تمامی مقادیر آن صفر هستند. سپس به ازای هر پیکسلی که در تصویر وجود دارد، به تعداد دفعاتی که موجود است هیستوگرام آن را افزایش می دهیم و در نهایت هیستوگرام را برمی گردانیم.

محاسبه CDF :

```
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
    '''
    hist = calc_hist(channel)
    # Calculate cumulative distribution function (CDF)
    cdf = hist.copy()
    cdf[0] = hist[0]
    for i in range(1, len(hist)):
        cdf[i] = cdf[i-1] + hist[i]

    return cdf
```

ابتدا هیستوگرام کانال تصویر ورودی را با استفادهتاعی که در بخش ثبلی پیاده سازی شد، محاسبه می کنیم. سپس، یک آرایه cdf را برای ذخیره تابع توزیع تجمعی (*CDF*) مقداردهی اولیه می کنیم. *CDF* را با جمع کردن مقادیر هیستوگرام به صورت تجمعی محاسبه می کنیم. در نهایت *CDF* محاسبه شده را برمی گردانیم.

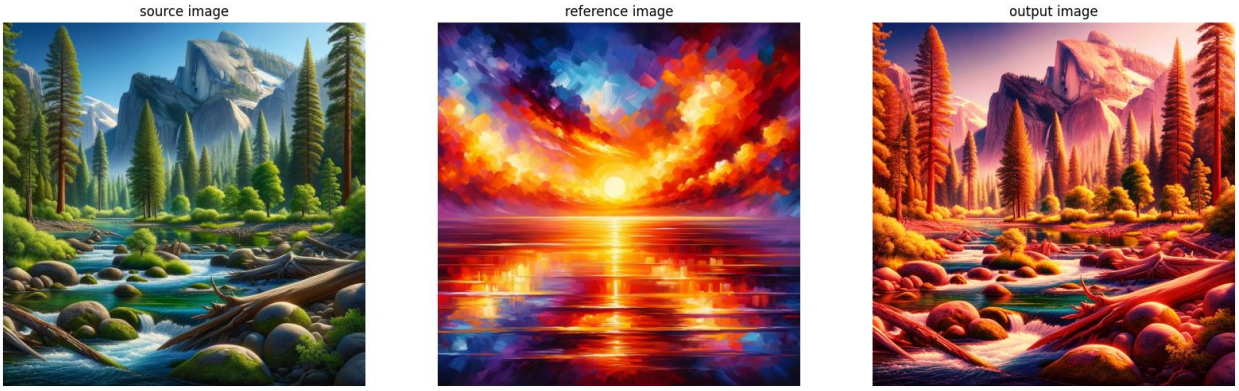
```
def hist_matching(src_image, ref_image):
    output_image = src_image.copy()
    channels = [(0, 'Blue channel'), (1, 'Green channel'), (2, 'Red channel')]
    # Iterate through each channel
    for channel, title in channels:
        # Compute the cumulative distribution function (CDF) for the source and
reference images
        src_cdf = calc_cdf(src_image[:, :, channel])
        ref_cdf = calc_cdf(ref_image[:, :, channel])

        # Compute the histogram mapping function
        mapping_func = np.zeros(256, dtype=np.uint8)
        for i in range(256):
            mapping_func[i] = np.argmax(ref_cdf >= src_cdf[i])
        # Apply the mapping function to each pixel value in the source image
        for i in range(src_image.shape[0]):
            for j in range(src_image.shape[1]):
                output_image[i, j, channel] = mapping_func[src_image[i, j,
channel]]

    return output_image
```

ابتدا یک کپی از تصویر منبع با نام **output_image** ایجاد می کنیم. سپس لیستی به نام **channels** تعریف می کنیم که شامل تاپل‌هایی برای هر کانال رنگی است: (اندیس کانال، نام کانال). در این کد، کانال‌ها به ترتیب آبی، سبز و قرمز مرتب شده‌اند. کد در هر کانال از تصویر (آبی، سبز و قرمز) حرکت می‌کند. برای هر کانال، توابع توزیع تجمعی (CDF) برای هر دو تصویر منبع (**src_image**) و تصویر مرجع (**ref_image**) با استفاده از تابع **calc_cdf**(موجود در کد قبل) محاسبه می‌شود. درون حلقه کانال، یک آرایه خالی به نام **mapping_func** با ۲۵۶ عنصر ایجاد می کنیم که می‌تواند مپینگ از شدت پیکسل‌های منبع به شدت‌های همسان از هیستوگرام تصویر مرجع را ذخیره کند. سپس برای هر سطح شدت (از ۰ تا ۲۵۵)، تطابق مربوطه با پیدا کردن شاخص اولین عنصر در توزیع تجمعی مرجع که بزرگتر یا مساوی عنصر متناظر در توزیع تجمعی منبع است، تعیین می‌شود. در نهایت، تابع تطابقی را به هر مقدار پیکسل در تصویر منبع اعمال می‌کند و **output_image** را به‌روزرسانی می‌کنیم.

خروجی تصویر به این شکل خواهد بود :

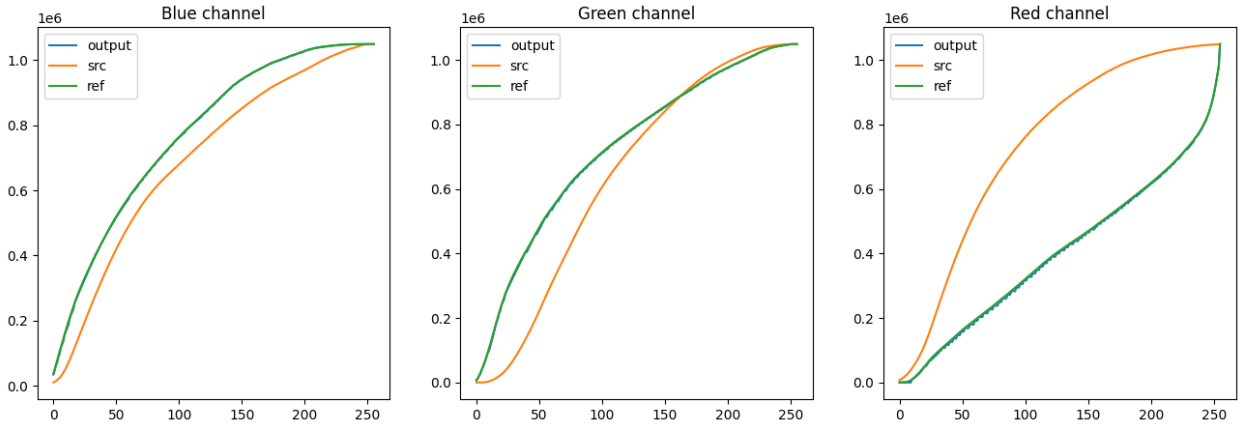


همانطور که می بینیم تصویر منبع را به گونه‌ای تغییر یافته است که هیستوگرام آن با هیستوگرام تصویر مرجع همسان شده است. این کار به ما امکان می‌دهد که رنگ‌ها و کنتراست تصویر منبع را به گونه‌ای تغییر دهیم که شبیه تصویر مرجع شود.

به عبارت دیگر، با اعمال تطبیق هیستوگرام، تصویر منبع به گونه‌ای تغییر می‌کند که نقاط مهم هیستوگرام مرجع در آن تقریباً با همان مقادیر در هیستوگرام تصویر منبع تطبیق پیدا کنند. این اقدام می‌تواند منجر به تغییر کنتراست، روشنایی و دیداری تصویر شود، به طوری که هیستوگرام تصویر خروجی نزدیک به هیستوگرام تصویر مرجع باشد.

به طور مثال، اگر تصویر منبع دارای توزیع پیکسل‌هایی با کنتراست پایین باشد و تصویر مرجع دارای توزیع پیکسل‌هایی با کنتراست بالا، با اعمال تغییر هیستوگرام، ما می‌توانیم کنتراست تصویر منبع را افزایش دهیم تا به تصویر مرجع نزدیک‌تر شود. در مورد تغییر رنگ، ما می‌توانیم تغییراتی اعمال کنیم که رنگ‌های تصویر منبع را به سمت رنگ‌های تصویر مرجع بیشتر ببریم. این تغییرات می‌تواند شامل تعویض رنگ‌ها، تغییر روشنایی یا تغییر کنتراست باشد. به طور کلی، با تغییر هیستوگرام، ما می‌توانیم تصویر منبع را به گونه‌ای تغییر دهیم که با تصویر مرجع مطابقت داشته باشد، بیشترین شباهت را با تصویر مرجع داشته باشد و اصلاحات لازم را برای بهبود کیفیت تصویر اعمال کنیم.

بنابراین همانطور که مشاهده می شود رنگ ها، کنتراست و به طور کلی هیستوگرام تصویر منبع شبیه تصویر مرجع می شود بدون اینکه خود تصویر تغییری پیدا کند و اطلاعاتی از دست برود.



همانطور که مشاهده می شود نمودار کانال های آبی و سبز تصویر منبع و مرجع تقزیبا نزدیک به هم هستند زیرا توزیع رنگ کانال های آبی و سبز در هردو تصویر تقریبا نزدیک به هم است. در مورد کانال قرمز به دلیل تفاوت تفاوت رنگی این کانال در تصویر مبدا و مرجع، همانطور که مشاهده می شود نمودار این دو تصویر نزدیک به هم نیستند. و اما در مورد تصویر خروجی همانطور که مشاهده می شود نمودار آن کاملا منطبق بر تصویر مرجع است که این نشان دهنده این است که با اعمال تطبیق هیستوگرام، کنتراست، سطح روشنایی و کانال های رنگی تصویر خروجی کاملا منطبق بر تصویر مرجع بدست آمده است.

سوال ۳) برای متعادل سازی هیستوگرام با استفاده از opencv از کد زیر استفاده شده است.

```
equalize_image = cv2.equalizeHist(image)
```

نتیجه :

استفاده از این تابع معایبی دارد :

۱. Global adjustment : متعادل سازی هیستوگرام کل تصویر را تنظیم می‌کند، به این معنی که ممکن است در مواردی که تصویر شامل مناطق با شرایط روشنایی مختلف یا اشیاء متمایز با سطوح روشنایی متفاوت است، نتایجه مناسبی را تولید نکند.

۲. افزایش نویز : متعادل سازی ممکن است نویز را در تصویر افزایش دهد، به خصوص در مناطق با کنتراست کم یا روشنایی پایین.
۳. افت کنتراست محلی : در برخی موارد، متعادل سازی هیستوگرام می‌تواند منجر به افت کنتراست محلی و از بین رفتن جزئیات دقیق در تصویر شود. این به این دلیل است که این تابع مقادیر پیکسل‌ها را به یکنواخت بین تمام هیستوگرام توزیع می‌کند که ممکن است برای حفظ جزئیات و بافت در مناطق خاص تصویر مناسب نباشد

هزینه محاسباتی، تغییر غیرطبیعی تصویر و ... نیز از معایب دیگر این روش هستند.

ب) ابتدا به سراغ **پیاده سازی ACE1** می رویم.

```
import cv2

def ACE1(image, gridSize):
    x, y = image.shape
    output_image = image.copy()

    # Apply histogram equalization to each grid
    for i in range(0, x, gridSize):
        for j in range(0, y, gridSize):
            grid = image[i:i+gridSize, j:j+gridSize]
            if grid.shape[0] > 0 and grid.shape[1] > 0:
                equalized_grid = cv2.equalizeHist(grid)
                output_image[i:i+gridSize, j:j+gridSize] = equalized_grid

    return output_image
```

ابتدا تصویر را به چند گرید با سایزی که در ورودی تابع داده می شود تقسیم بندی می کنیم. سپس به ازای هر یک از این گرید ها متعادل سازی هیستوگرام را اعمال کرده و قسمت متناظر را در تصویر خروجی قرار می دهیم. در نهایت تصویر بهبود یافته را بر می گردانیم.

مزیت و معایب این روش :

مزیت :

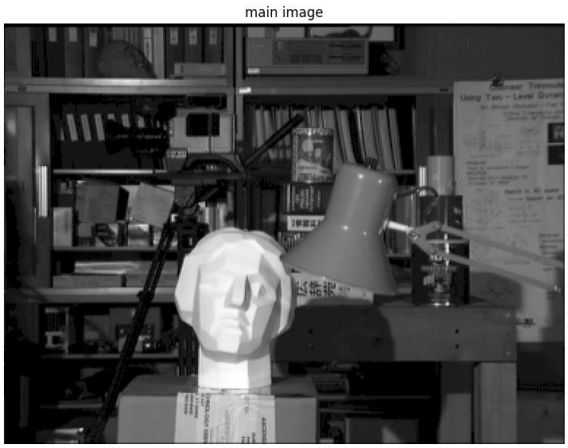
- انعطاف‌پذیری بالا: این روش انعطاف‌پذیری بالایی را در اعمال تطبیق کنتراست به هر گرید از تصویر فراهم می‌کند. با تعیین اندازه گرید، می‌توانیم کنتراست را به طور دقیق برای هر قسمت از تصویر تنظیم کنیم.
- محاسبات پارالل: این روش امکان اجرای محاسبات موازی بر روی هر گرید از تصویر را فراهم می‌کند. این امر می‌تواند سرعت پردازش را افزایش دهد، به خصوص برای تصاویر با ابعاد بزرگ.

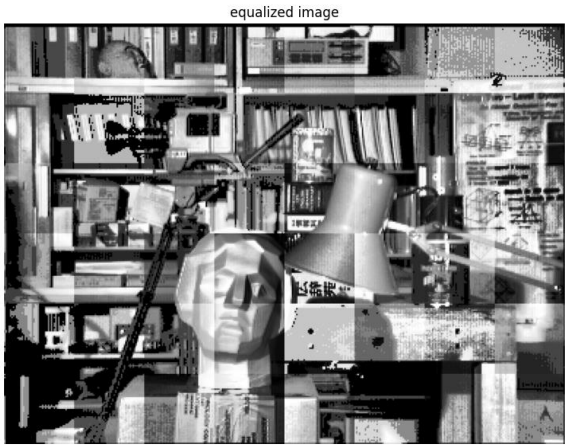
معایب :

- پیچیدگی محاسباتی: این روش ممکن است نیاز به محاسبات زیادی داشته باشد، به ویژه برای تصاویر با ابعاد بزرگ و اندازه شبکه‌های کوچک. این محاسبات می‌توانند زمان پردازش را افزایش دهند و نیاز به منابع محاسباتی بیشتری داشته باشند.

- ممکن است منجر به ناهمواری‌های تصویری شود: اعمال تطبیق کنتراست به هر شبکه می‌تواند منجر به ناهمواری‌های تصویری در ترکیب تصویر اصلی شود، به خصوص در مواردی که مرزهای گرید محاسبه شده و مقادیر پیکسلی تصویر تغییر می‌کنند.

خروجی :





بنابراین همانطور که در خروجی مشاهده می شود متعادل سازی هیستوگرام به ازای هر کدام از تقسیم بندی ها، به صورت محلی ، به نحو خوبی انجام شده است اما مرز این تقسیم بندی ها مشخص است که نتیجه دلخواهی نیست.

پیاده سازی ACE2 :

```
import cv2

def ACE2(image, gridSize):
    output = image.copy()

    # Apply padding to the image
    padded_image = cv2.copyMakeBorder(image, gridSize[0]//2, gridSize[0]//2,
gridSize[1]//2, gridSize[1]//2, cv2.BORDER_REFLECT)

    # Iterate over each pixel of the image
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Extract neighborhood defined by gridSize
            neighborhood = padded_image[i:i+gridSize[0], j:j+gridSize[1]]
            # Apply histogram equalization to the neighborhood
            equalized_neighborhood = cv2.equalizeHist(neighborhood)
            # Replace the central pixel value with the equalized value
            output[i, j] = equalized_neighborhood[gridSize[0]//2, gridSize[1]//2]

    return output
```

ابتدا از ابزار Opencv برای اعمال پدینگ به تصویر استفاده می کنیم زیرا پدینگ لازم است تا اطمینان حاصل شود که همه پیکسل‌ها محیطی با اندازه تعیین شده توسط `*gridSize*` دارند. سپس برای هر پیکسل، محیط تعریف شده توسط `*gridSize*` را استخراج کنیم و متعادل سازی هیستوگرام را اعمال می کنیم. در آخر مقدار پیکسل را با مقدار متعادل شده جایگزین می کنیم و تصویر خروجی را بر می گردانیم.

مزایا و معایب این روش :

مزایا :

- سادگی الگوریتم: در این روش با استفاده از توابع داخلی*OpenCV* ، فرآیند تطبیق کنتراست برای هر پیکسل به سادگی انجام می‌شود.
- عملکرد سریع: استفاده از توابع داخلی *OpenCV* معمولاً با سرعت بالا و عملکرد بهتری صورت می‌گیرد. این موجب افزایش سرعت عملیات *ACE* بر روی تصاویر بزرگ می‌شود.

معایب :

- کمبود انعطاف‌پذیری: این روش کمبود انعطاف‌پذیری در اعمال تطبیق کنتراست دارد. زیرا تنها از یک تابع داخلی برای اعمال تطبیق کنتراست بر روی هر شبکه استفاده می‌کند و فرآیند را برای هر پیکسل به طور مستقل انجام نمی‌دهد.
- احتمال ایجاد نویز: استفاده از تساوی هیستوگرام بر روی هر شبکه ممکن است منجر به افزایش نویز در تصویر شود، به ویژه در مناطق با نویز پایین که نیازی به تطبیق کنتراست ندارند.

همچنین استفاده از حافظه بیشتر اعمال پدینگ به تصویر از دیگر معایب این روش است، زیرا نیاز است که تصویر را با ابعاد بزرگتری بسازیم تا پیکسل‌ها محیطی با اندازه `*gridSize*` داشته باشند.

خروجی:

همانطور که در تصویر مشاهده می شود، کنتراست تصویر به خوبی افزایش پیدا کرده است اما همانطور که در معایب نیز ذکر شد تصویر حاصل دارای نویز در قسمتی است که در تصویر اصلی از کنتراست پایینی برخوردار بود و نیازی به تطبیق نداشت.

پیاده سازی روش CLAHE :

```
import cv2

def CLAHE(image, gridSize, clip_limit):
    output = image.copy()
    # Apply padding to the image
    padded_image = cv2.copyMakeBorder(image, gridSize[0]//2, gridSize[0]//2,
gridSize[1]//2, gridSize[1]//2, cv2.BORDER_REFLECT)

    # Iterate over each pixel of the image
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Extract neighborhood defined by gridSize
            neighborhood = padded_image[i:i+gridSize[0], j:j+gridSize[1]]
            # Calculate histogram for the neighborhood
            hist, _ = np.histogram(neighborhood, bins=256, range=(0, 256))
            # Apply contrast limiting to the histogram
            clipped_hist = np.clip(hist, 0, clip_limit)
            # Calculate cumulative distribution function (CDF) for the clipped
histogram
            cdf = np.cumsum(clipped_hist)
            # Calculate transition function for each pixel
            transition_func = (cdf[neighborhood] - cdf[neighborhood.min()]) * 255
/ (neighborhood.size - cdf[neighborhood.min()])
            # Replace central pixel value with equalized value
            output[i, j] = transition_func[gridSize[0]//2, gridSize[1]//2]

    return output
```

مانند قبل ابتدا به تصویر پدینگ می دهیم و برای هر پیکسل، محیط تعریف شده توسط `*gridSize*` را استخراج کرده و برای هر محیط، هیستوگرام را محاسبه می کنیم. سپس برای جلوگیری از افزایش بیش از حد نویز محدودیت کنتراست را با کلیپ کردن بازه‌های هیستوگرام بر اساس `*clip_limit*` انجام می‌دهیم. در مرحله بعد تابع توزیع تجمعی (*CDF*) را برای هیستوگرام کلیپ شده اعمال کرده و تابع transition را برای هر پیکسل اعمال می کنیم. در آخر مقدار پیکسل با مقدار متعادل سازی شده جایگزین می کنیم تا تصویر بهبود یابد.

مزایا و معایب این روش :

مزیت :

- حفظ جزئیات : *CLAHE* قادر به افزایش کنتراست در نواحی مختلف تصویر است، در حالی که جزئیات را حفظ می‌کند.
- کنترل کنتراست: با استفاده از محدودیت کنتراست، این روش قادر است از افزایش بیش از حد نویز در نواحی با کنتراست پایین جلوگیری کند، در حالی که همچنان افزایش کنتراست در نواحی با کنتراست بالا را اعمال می‌کند.

- انعطاف‌پذیری: این روش انعطاف‌پذیری بالایی دارد و می‌تواند برای تصاویر با اندازه و مشخصات مختلف مورد استفاده قرار بگیرد.

معایب :

- مصرف حافظه: این روش نیازمند مصرف حافظه بالاست، به ویژه برای تصاویر با ابعاد بزرگ.
- زمان اجرا : اجرای *CLAHE* ممکن است زمان‌بر باشد، به خصوص برای تصاویر با ابعاد بزرگ.
- آرتیفکت‌های ناشی از تایل‌بندی: استفاده از تایل‌بندی ممکن است منجر به ظهور آرتیفکت‌های ناشی از انتقال ناگهانی کنتراست در مرزهای تایل‌ها شود، که ممکن است به شکل‌دهی به لبه‌های مصنوعی یا پدیده‌های دیگر منجر شود.

خروجی :

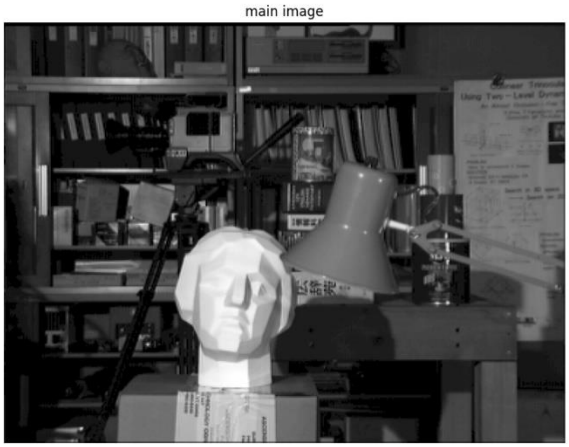
همانطور که مشاهده می شود این روش کنتراست تصویر را به خوبی افزایش داده است و نویزی نیز تقویت نکرده است. اما همانطور که ذکر شد، این فرایند زمان بر بود.

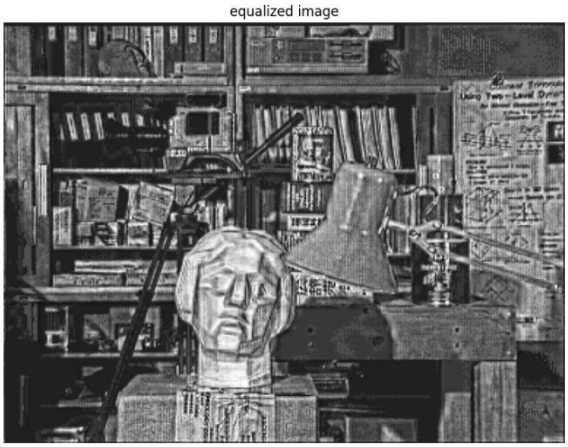
ج) پیاده سازی CLAHE با استفاده از کتابخانه opencv

```
def CLAHE(image, gridSize, clipLimit):
    clahe = cv2.createCLAHE(clipLimit=clipLimit, tileGridSize=gridSize)
    clahe_output = clahe.apply(image)
    return clahe_output
```

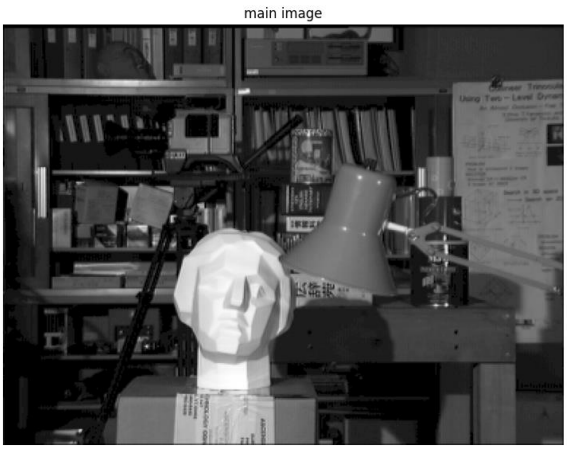
در این کد با استفاده از تابع createCLAHE و با محدودیت برش و سایز گرید این متود را روی تصویر اعمال می کنیم.

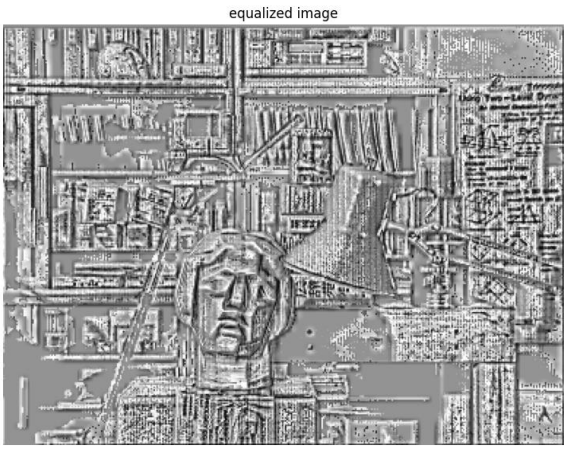
۱) ابعاد پنجره ۱۲۸ * ۱۲۸ و حد برش ۲



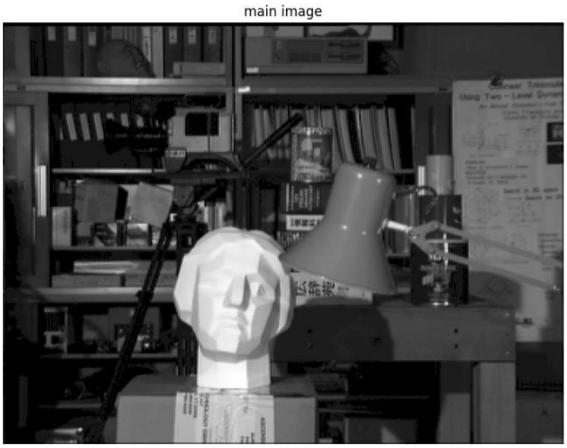


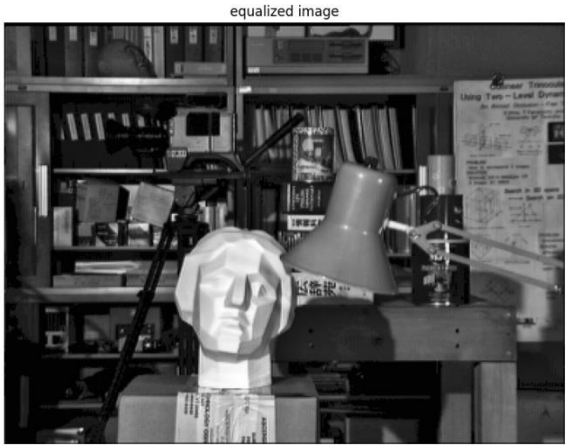
۲) ابعاد پنجره ۱۲۸ * ۱۲۸ و حد برش ۱۲۸



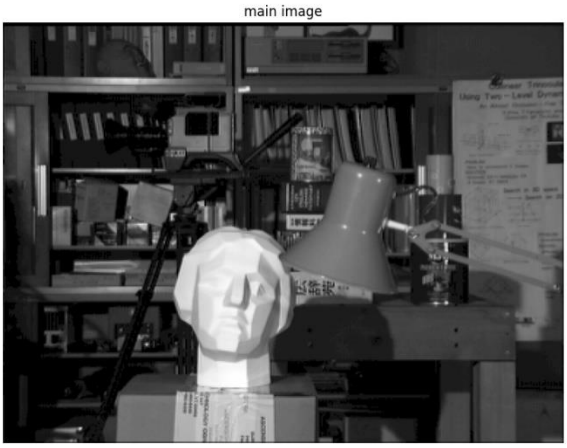


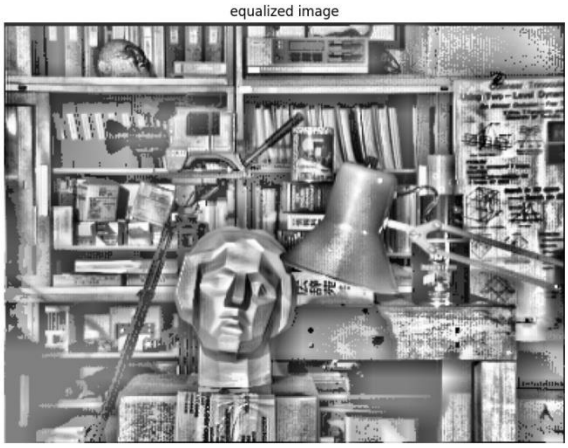
۳) ابعاد پنجره ۱۶ * ۱۶ و حد برش ۲





۴) ابعاد پنجره ۱۶ * ۱۶ و حد برش ۱۲۸





تاثیر ابعاد پنجره (*gridSize*) و حد برش (*clipLimit*) بر خروجی تابع *CLAHE* :

ابعاد پنجره (*gridSize*):

- ابعاد پنجره تعیین کننده اندازه تایل‌هایی است که بر روی تصویر برای اعمال افزایش تطبیقی کنتراست استفاده می‌شود.

- اگر ابعاد پنجره بزرگتر باشد، تصویر به ناحیه‌های بزرگتری تقسیم می‌شود و کنتراست هر ناحیه به صورت محلی افزایش می‌یابد.

حد برش (*clipLimit*):

- حد برش مقداری است که تعیین می‌کند تا چه حد کنتراست هیستوگرامی هر تایل باید افزایش یابد.

- افزایش این مقدار می‌تواند باعث افزایش کنتراست تصویر شود، اما اگر بیش از حد باشد ممکن است باعث افزایش نویز و تزاحم بین تایل‌ها شود.

بنابراین برای دو تصویر اول که ابعاد پنجره در آنها بزرگتر بود شاهد بودیم که به دلیل انتخاب تایل های بزرگتر در تصویر، کنتراست محلی به خوبی اعمال نمی شود. همچنین با افزایش حد برش دیدیم که نویز ها و در هم رفتگی تایل ها بیشتر شد.

برای دو تصویر بعدی که ابعاد پنجره مناسبی داشتند، مشاهده می شود که تصویر خروجی تصویر دلخواهی است زیرا تایل های تصویر به خوبی انتخاب شده اند و تقسیم بندی تصویر به صورتی بوده که کنتراست محلی به خوبی اعمال شده است. در تصویر اول که حد برش کمتری دارد می بینیم که تصویر خروجی مناسب است اما با افزایش حد برش، همانطور که در خروجی آخر پیداست، نویز و در هم رفتگی (تزاحم) بین تایل ها بیشتر شده است.

در نتیجه، انتخاب ابعاد پنجره و حد برش مناسب باید با توجه به ویژگی‌های خاص تصویر و نیازهای کاربردی صورت گیرد. در این انتخاب معمولاً به صورت تجربی و با آزمایش مقادیر مختلف این پارامترها انجام می‌شود تا بهترین نتیجه به دست آید. برای این تصویر و اکثر تصاویر ابعاد پنجره کمتر مناسب تر است. حد برش نیز بیشتر تجربی است که در این مثال حد برش کمتر نتیجه بهتری داشته است.

سوال ۴)

الف) پیاده سازی نویز نمک و فلفل :

```python
def Add_Noise(img):
    # Copy the input image to avoid modifying the original image
    noisy_img = np.copy(img)

    # Define probability of noise (adjust as needed)
    noise_prob = 0.01

    # Generate random noise mask
    salt_mask = np.random.rand(*img.shape) < noise_prob / 2
    pepper_mask = np.random.rand(*img.shape) < noise_prob / 2

    # Apply salt noise
    noisy_img[salt_mask] = 255

    # Apply pepper noise
    noisy_img[pepper_mask] = 0

    return noisy_img
```

۲. ابتدا احتمال نویز را تعیین می کنیم که مشخص می‌کند چقدر احتمال دارد هر پیکسل توسط نویز تحت تاثیر قرار گیرد. سپس برای ایجاد ماسک‌های تصادفی برای نویز نمک و فلفلی به صورت جداگانه از `np.random.rand` برای تولید اعداد تصادفی بین ۰ و ۱ استفاده کرده و آن‌ها را با احتمال نویز تقسیم بر ۲ مقایسه می‌کنیم تا اطمینان حاصل شود که کل احتمال نویز ثابت می‌ماند. سپس در جاهایی که میخواهیم نویز نمکی اعمال کنیم، با تنظیم مقدار پیکسل‌ها به حداکثر شدت (۲۵۵) و در نویز فلفلی با تنظیم مقدار پیکسل‌ها به حداقل شدت (۰) این کار را انجام می دهیم.

خروجی :

ب) پیاده سازی reflect101 :

```
def Reflect101(img, filter_size):
    pad_size = filter_size // 2
    # Create padded image with zeros
    padded_img = np.zeros((img.shape[0] + 2 * pad_size, img.shape[1] + 2 * pad_size),
dtype=img.dtype)
    # Center of the new image
    padded_img[pad_size:-pad_size, pad_size:-pad_size] = img
    # Reflect edges
    left_reflection = img[:, pad_size - 1::-1]
    right_reflection = img[:, -(pad_size + 1):-(2 * pad_size + 1):-1]
    top_reflection = img[pad_size - 1::-1, :]
    bottom_reflection = img[-(pad_size + 1):-(2 * pad_size + 1):-1, :]
    padded_img[pad_size:-pad_size, :pad_size] = left_reflection  # Left
    padded_img[pad_size:-pad_size, -pad_size:] = right_reflection  # Right
    padded_img[:pad_size, pad_size:-pad_size] = top_reflection  # Top
    padded_img[-pad_size:, pad_size:-pad_size] = bottom_reflection  # Bottom

    # Reflect corners
```

```
    top_left_reflection = img[pad_size - 1::-1, pad_size - 1::-1]
    top_right_reflection = img[pad_size - 1::-1, -(pad_size + 1):-(2 * pad_size + 1):-1]
    bottom_left_reflection = img[-(pad_size + 1):-(2 * pad_size + 1):-1, pad_size - 1::-1]
    bottom_right_reflection = img[-(pad_size + 1):-(2 * pad_size + 1):-1, -(pad_size + 1):-(2
* pad_size + 1):-1]
    padded_img[:pad_size, :pad_size] = top_left_reflection  # Top-left
    padded_img[:pad_size, -pad_size:] = top_right_reflection  # Top-right
    padded_img[-pad_size:, :pad_size] = bottom_left_reflection  # Bottom-left
    padded_img[-pad_size:, -pad_size:] = bottom_right_reflection  # Bottom-right

    return padded_img
```

ابتدا ابعاد تصویر و ابعاد فیلتر (ارتفاع و عرض تصویر و ارتفاع و عرض فیلتر) را استخراج می‌کنیم.

سپس با محاسبه اندازه پدینگ برای بالا، پایین، چپ و راست بر اساس اندازه فیلتر، تعیین می‌کنیم که چقدر باید از حاشیه‌های تصویر برای پدینگ استفاده کنیم. سپس برای انعکاس ردیف‌های بالا و پایین تصویر از اندیسگذاری معکوس (`[::-1]`) استفاده می‌کنیم تا ردیف‌های بالا و پایین را به ترتیب منعکس کنیم. به همان روش ستون‌های چپ و راست را نیز منعکس می کنیم. سپس این انعکاس‌ها را به تصویر اصلی برای ایجاد تصویر با پدینگ متصل می کنیم.

پیاده سازی average bluring :

```
def Averaging_Blurring(img, filter_size):
    image = Reflect101(img, filter_size)
    # Extract filter dimensions
    filter_height = filter_size
    filter_width = filter_size

    # Extract image dimensions
    height, width = img.shape
    # Initialize the result image
    result = np.zeros_like(img)
    # Compute the sum of pixel values in the filter
    filter_sum = filter_height * filter_width
    # Perform averaging blurring
    for i in range(height):
        for j in range(width):
            # Extract the region of interest (ROI)
            roi = image[i:i+filter_height, j:j+filter_width]
            # Calculate the average pixel value in the ROI
            avg_value = np.sum(roi) / filter_sum
            # Set the result pixel value
            result[i, j] = avg_value

    return result
```

برای پیاده سازی این تابع ابتدا پدینگ *Reflect101* را بر روی تصویر ورودی اعمال می کنیم. سپس ابعاد فیلتر (ارتفاع و عرض) را با استفاده از `*filter_size*` بدست می آوریم. همچنین ابعاد تصویر را نیز بدست می آوریم. در ادامه مجموع مقادیر پیکسلی در فیلتر را محاسبه کرده و به ازای هر پیکسل مراحل زیر را انجام می دهیم:

- در هر مرحله از تصویر، به ازای هر پیکسل، ما ناحیه‌ای از تصویر با اندازه فیلتر را استخراج می‌کنیم (*ROI*).
- سپس، میانگین مقادیر پیکسلی در *ROI* را با تقسیم مجموع مقادیر پیکسلی در *ROI* بر تعداد کل پیکسل‌های فیلتر محاسبه می‌کنیم.
- مقدار محاسبه شده به عنوان مقدار پیکسل در result ذخیره می‌شود.

این روش با استفاده از انعکاس‌های متقابل در حاشیه‌های تصویر و محاسبه میانگین مقادیر پیکسلی در نواحی همسایه، average bluring را بر روی تصویر اعمال می‌کند.

پیاده سازی Median Blurring

```
def Median_Blurring(img, filter_size):

  image = Reflect101(img, filter_size)
  result = np.zeros((img.shape))
  # Extract filter dimensions
  filter_height = filter_size
  filter_width = filter_size
  # Extract image dimensions
  height, width = img.shape
  # Perform median blurring
  for i in range(height):
      for j in range(width):
          # Extract the region of interest (ROI)
          roi = image[i:i+filter_height, j:j+filter_width]
          # Calculate the median pixel value in the ROI
          median_value = np.median(roi)
          # Set the result pixel value
          result[i, j] = median_value

  return result
```

در این کد، مانند قبل ابتدا پدینگ *Reflect101* را بر روی تصویر ورودی اعمال کرده و ابعاد فیلتر و ابعاد تصویر را بدست می آوریم. سپس برای هر پیکسل مراحل زیر را انجام می دهیم:

- در هر مرحله از تصویر، به ازای هر پیکسل، ما ناحیه‌ای از تصویر با اندازه فیلتر را استخراج می‌کنیم (*ROI*).
- سپس، مقدار میانگین مقادیر پیکسلی در *ROI* را با استفاده از تابع `*np.median*` محاسبه می‌کنیم.
- مقدار محاسبه شده به عنوان مقدار پیکسل در نتیجه ذخیره می‌شود.

این روش با استفاده از انعکاس‌های متقابل در حاشیه‌های تصویر و محاسبه مقدار میانه مقادیر پیکسلی در نواحی همسایه، اثر median blurring را بر روی تصویر اعمال می‌کند.

پیاده سازی guassian blurring :

```
def Gaussian_Blurring(img, filter_size, std):
    kernel = np.zeros((filter_size, filter_size))
    center_i = filter_size // 2
    center_j = filter_size // 2
    for i in range(filter_size):
        for j in range(filter_size):
            x = i - center_i
            y = j - center_j
            kernel[i, j] = np.exp(-(x**2 + y**2) / (2 * std**2))
    kernel /= np.sum(kernel)
    img = Reflect101(img, filter_size)
    output = img.copy()
    result = cv2.filter2D(src=output, ddepth=-1, kernel=kernel)
    return result
```

در این تابع ابتدا یک هسته گوسی که یک آرایه صفری به ابعاد مشخص شده توسط `*filter_size*` است ایجاد می‌کنیم. سپس برای هر پیکسل در این آرایه، فرمول تابع گوسی را محاسبه کرده و در آن قرار می‌دهیم. این کار باعث می‌شود که مقدار پیکسل‌های مرکزی بیشتر باشند و به تدریج با فاصله از مرکز کمتر شوند. پس از محاسبه هسته گوسی، ما آن را نرمال می‌کنیم تا مجموع تمام عناصر آن به یک برسد. این کار باعث می‌شود که اثر blurring گوسی معقول باشد و مقادیر پیکسل‌ها به درستی اثر داشته باشند. سپس، ما تصویر ورودی را با استفاده از تابع `*Reflect101*` پدینگ می‌کنیم. سپس هسته گوسی را با استفاده از تابع `*cv2.filter2D*` به تصویر اعمال می‌کنیم تا اثر گوسی بر روی تصویر اعمال شود.

این روش از اثر گوسی برای انعکاس هیستوگرام پیکسل‌های نزدیک به مرکز بیشتر و تاثیر کمتر بر پیکسل‌های دورتر استفاده می‌کند، که باعث ایجاد اثر ماتیسه گوسی با پخش تصویر و کاهش نویز می‌شود.

خروجی ها :

در فیلتر متوسط گیر میانگین مقادیر پیکسل‌ها را در نظر گرفته می شود و با اعمال آن، تصویر از نویزهای کوچک‌تر تمیز می شود.استفاده از کرنل بزرگتر باعث افزایش اثرات نرمال‌سازی و کاهش وضوح تصویر می‌شود.

در فیلتر میانه نویز به خوبی کاهش یافته است اما بهتر از فیلتر قبلی گوشه ها و جزئیات تصویر به خوبی حفظ شده اند. اما با افزایش اندازه کرنل، تاثیر فیلتر بر روی تصویر افزایش می‌یابد و می‌تواند باعث کاهش وضوح تصویر شود.این فیلتر با استفاده از یک هسته گوسی میانگین‌گیری انجام می‌دهد که باعث کاهش نویز و حفظ حفره‌های موجود در تصویر می‌شود.

استفاده از این فیلتر با کرنل‌های بزرگتر می‌تواند باعث کاهش وضوح تصویر شود، اما همچنین می‌تواند نویزهای موجود در تصویر را بهبود دهد.در فیلتر گوسی تاری ایجاد شده نسبت به روش های دیگر طبیعی تر است.

در مجموع با استفاده از کرنل‌های بزرگتر، تاری بیشتری روی تصویر اعمال می‌شود. کرنل‌های بزرگتر، منطقه‌ای وسیع‌تر از پیکسل‌ها را در محاسبات میانگین یا میانه درگیر می‌کنند، که نتیجه‌اش افزایش میزان تاری در خروجی است. همچنین، استفاده از کرنل‌های بزرگتر می‌تواند به کاهش محسوس جزئیات و وضوح تصویر بینجامد، چرا که تفاوت‌های محلی کوچک بین پیکسل‌ها در میانگین یا میانه از بین می‌روند. کرنل‌های بزرگتر، تأثیری نرم‌تر و یکنواخت‌تر ایجاد می‌کنند، زیرا تفاوت‌های نوری و رنگی در مناطق وسیع‌تری از تصویر همگن می‌شوند. در مقابل، کرنل‌های کوچکتر تأثیر تاری کمتری دارند و امکان حفظ جزئیات بیشتری را فراهم می‌آورند. این کرنل‌ها برای اعمال تغییرات نرم و ظریف بر تصویر مناسب‌اند، بدون اینکه تأثیر قابل توجهی بر وضوح کلی تصویر داشته باشند.

ب) پیاده سازی فیلتر ها با استفاده از opencv:

```
kernel_size = (15, 15)
AveragingBlurring = cv2.blur(image, kernel_size)
MedianBlurring = cv2.medianBlur(image, 15)
GaussianBlurring = cv2.GaussianBlur(image, kernel_size, 40)
```

خروجی :

همانطور که مشاهده می شود نتایج با قسمت قبلی یکسان است.

سوال ۵ و ۶)

5) ابتدا لاپلاس را روی تصویر و سپس با صفت روی تصویر حاصل کانولوشن انجام میدهیم.

$$kernel = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

① 12 حول -4 : $(10\times0)+(10\times1)+(10\times0)+(10\times1)+(12\times-4)+(10\times1)+(10\times0)+(10\times1)+(10\times0) = -8$

② حول 10 های دیگر : $(10\times0)+(10\times1)+(10\times0)+(10\times1)+(10\times-4)+(10\times1)+(10\times0)+(10\times1)+(10\times0) = 2$

③ : $(10\times0)+(10\times1)+(10\times0)+(10\times1)+(10\times-4)+(10\times1)+(10\times0)+(10\times1)+(10\times0) = 0$

$$\Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -8 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

6)

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

$$F(0,0) = \sum_{x=0}^{1}\sum_{y=0}^{1} f(x,y)\, e^{0} = 1+2+2+1 = 6$$

$$F(0,1) = \sum_{x=0}^{1}\sum_{y=0}^{1} f(x,y)\, e^{-\pi j} = 1+2-2-1 = 0$$

$$F(1,0) = \sum_{x=0}^{1}\sum_{y=0}^{1} f(x,y)\, e^{-\pi j} = 1+2-2-1 = 0$$

$$\Rightarrow \begin{bmatrix} 6 & 0 \\ 0 & -2 \end{bmatrix}$$

$$F(1,1) = \sum_{x=0}^{1}\sum_{y=0}^{1} f(x,y)\, e^{-\pi j(x+y)} = 1-2-2+1 = -2$$