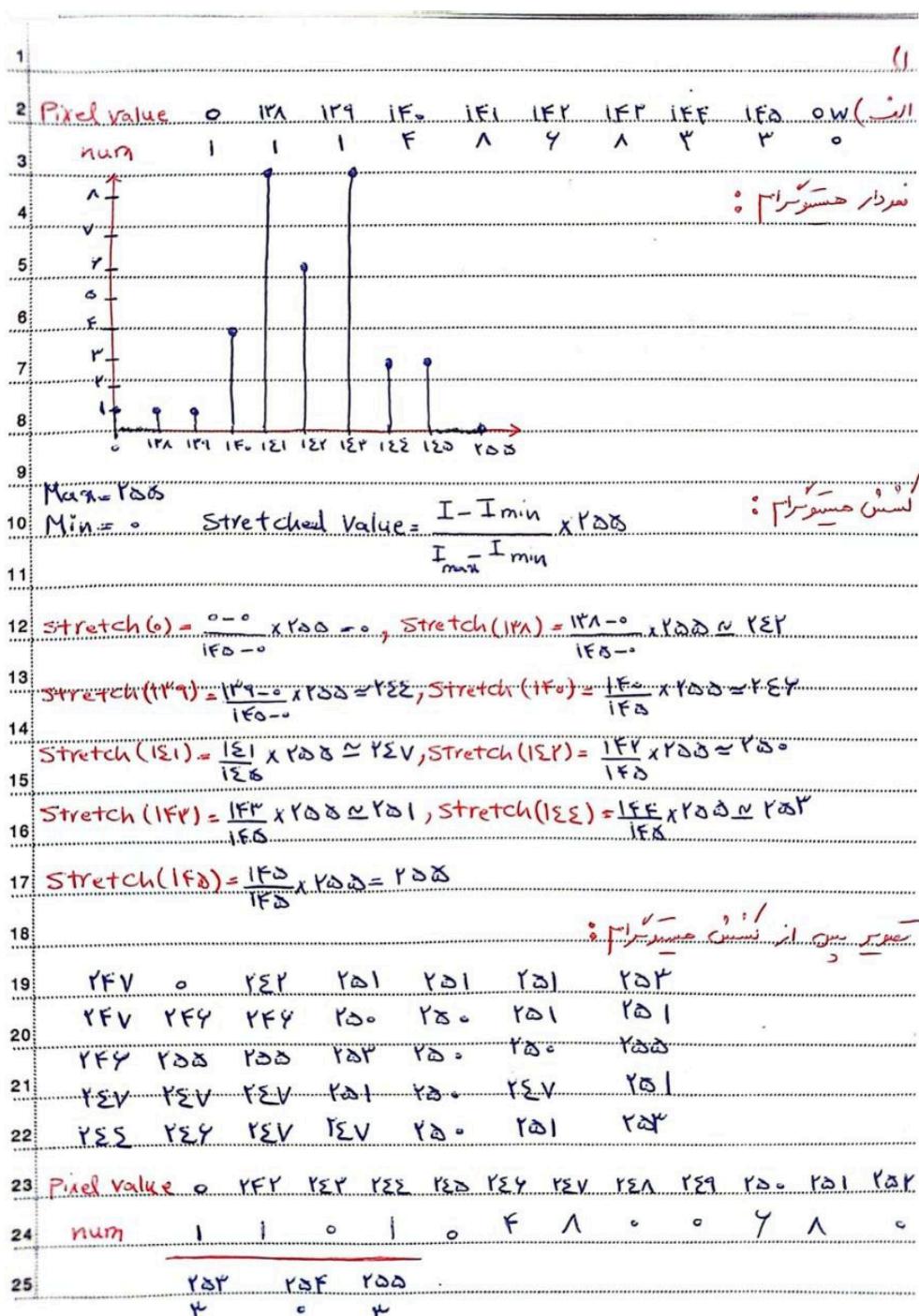
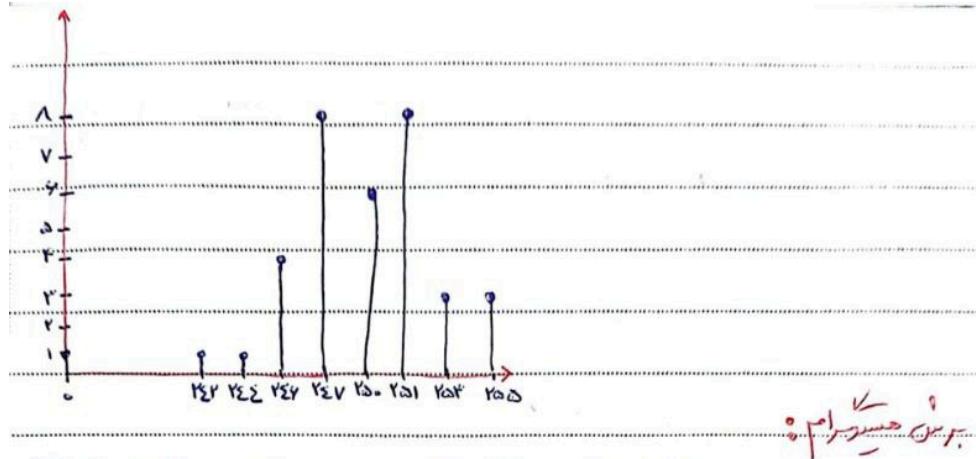


مليكا محمدی فخار
۹۹۵۲۲۰۸۶
تمرین دوم

.1

الف.





Pixel $\leq 128 \Rightarrow \text{clip} = 0$, pixel $> 128 \Rightarrow \text{clip} = 128$

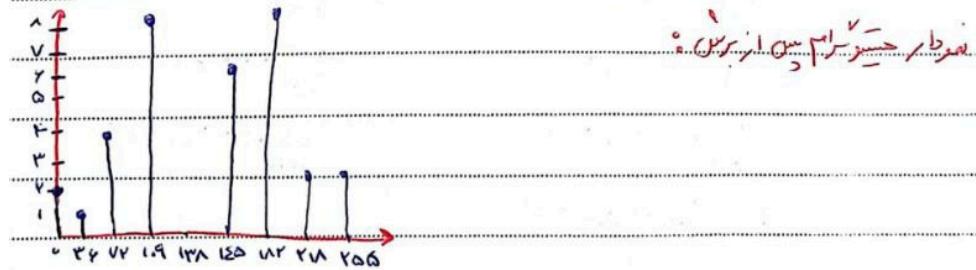
$$\text{clip}(129) = \frac{129 - 128}{128 - 128} \times 128 = 128, \text{clip}(120) = \frac{120 - 128}{128 - 128} \times 128 \approx 12$$

$$\text{clip}(121) = \frac{121 - 128}{128 - 128} \times 128 \approx 12, \text{clip}(122) = \frac{122 - 128}{128 - 128} \times 128 \approx 12$$

$$\text{clip}(123) = \frac{123 - 128}{128 - 128} \approx 12, \text{clip}(124) = \frac{124 - 128}{128 - 128} \approx 12$$

$$\text{clip}(125) = \frac{125 - 128}{128 - 128} \times 128 = 12, \text{clip}(126) = \frac{126 - 128}{128 - 128} \times 128 = 0$$

مخرج ١٢٨



مخرج ١٢٨

١٢٩ ٠ ٠ ١٢٨ ١٢٧ ٦٢ ٣٢ ٣١

١٢٩ ٦٢ ٦٢ ١٢٨ ١٢٧ ٦٢ ٣٢

٦٢ ٦٢ ٦٢ ٣٢ ٣٢ ٣٢

٣٢ ٣٢ ٣٢ ٣٢ ٣٢ ٣٢

٣٢ ٣٢ ٣٢ ٣٢ ٣٢ ٣٢

مخرج ٦٢

.ب

ابتدا تصویر را تعریف و آن را به تایپ `np.array` درآوردم:

```
image1 = [
    [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]
]

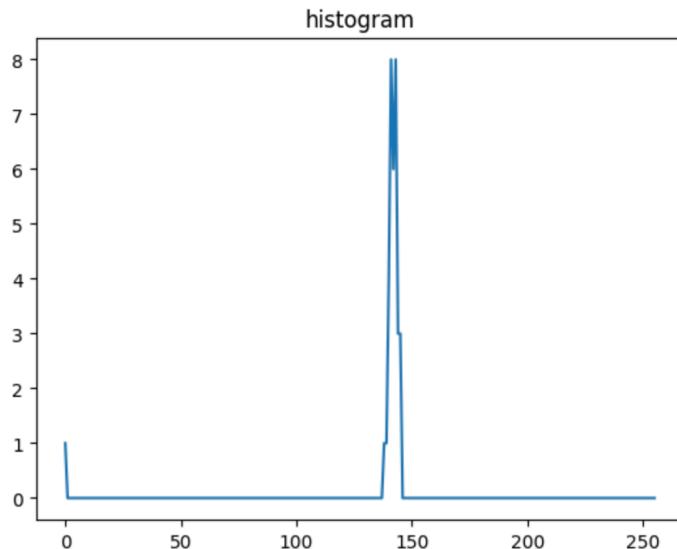
image1 = np.array(image1, dtype=np.uint8)
```

سپس در `calc_hist` یک لیست با اعضای ۰ و به طول ۲۵۶ می‌سازیم و با پیمایش بر روی تصویر ورودی به هر پیکسل که رسیدیم مقدار آن را در لیست یکی زیاد می‌کنیم.

```
# Initialize the histogram list with zeros. Assuming 256 bins for 8-bit images.
hist = [0] * 256

# Manually iterate through each row and pixel in the image
for row in image:
    for pixel in row:
        hist[pixel] += 1
```

حال اگر خروجی را ببینیم نمودار هیستوگرام دقیقاً مانند نموداری است که در بخش الف محاسبه کردیم.



سپس در تابع `np.array` ابتدا اطمینان حاصل می‌کنیم ورودی به صورت باشد، سپس مقدار مینیمم و ماکسیمم را در ورودی پیدا می‌کنیم. پس از آن فرمول کشیدگی هیستوگرام را بر روی تمام مقادیر ورودی اجرا کرده و حاصل را با تابع `np.round` گرد می‌کنیم:

```
# Ensure image is a NumPy array
if not isinstance(output_image, np.ndarray):
    image = np.array(output_image)

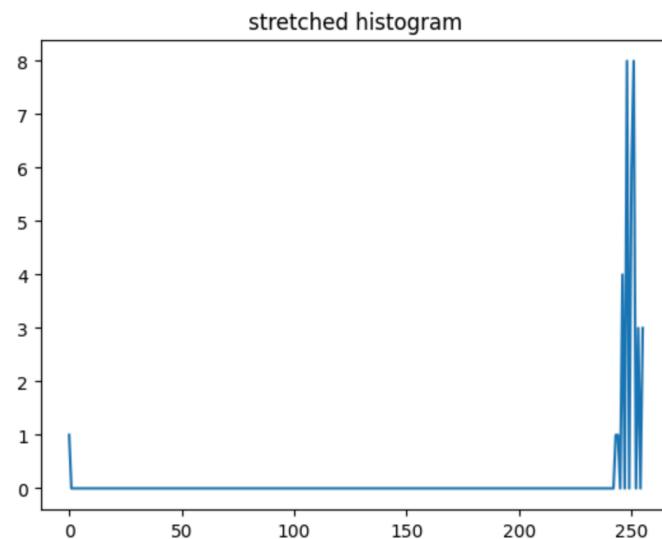
# Calculate the minimum and maximum pixel values
min_val = np.min(image)
max_val = np.max(image)

# Avoid division by zero
if max_val - min_val == 0:
    return image

# Apply histogram stretching
output_image = ((image - min_val) / (max_val - min_val)) * 255
output_image = np.round(output_image).astype('uint8') # Convert to uint8

return output_image
```

حال اگر خروجی را ببینیم نمودار هیستوگرام دقیقاً مانند نموداری است که در بخش الف محاسبه کردیم.



مرحله بعد پیاده سازی تابع `clip_hist` است: ابتدا مقدار ضریب مقیاس (`scale_factor`) را برای تغییر اندازه خطی محاسبه می‌کنیم. این کار با تقسیم ۲۵۵ بر تفاوت مقادیر حداقل و حداکثر انجام می‌شود.

سپس ماسک‌هایی برای پیکسل‌ها ایجاد می‌کنیم که مشخص می‌کند کدام پیکسل‌ها در بازه کلیپینگ قرار دارند. اگر پیکسلی بزرگتر یا مساوی `min_value` و کوچکتر یا مساوی `max_value` باشد، مقدار آن در ماسک `True` است.

تصویر خروجی را با استفاده از مقادیر مقیاس شده زمانی که ماسک `True` است، مقداردهی اولیه می‌کنیم.

در آخر مقادیری از تصویر که کمتر از `min_value` هستند را به صفر و مقادیری که بیشتر از `max_value` هستند را به ۲۵۵ تنظیم می‌کنیم.

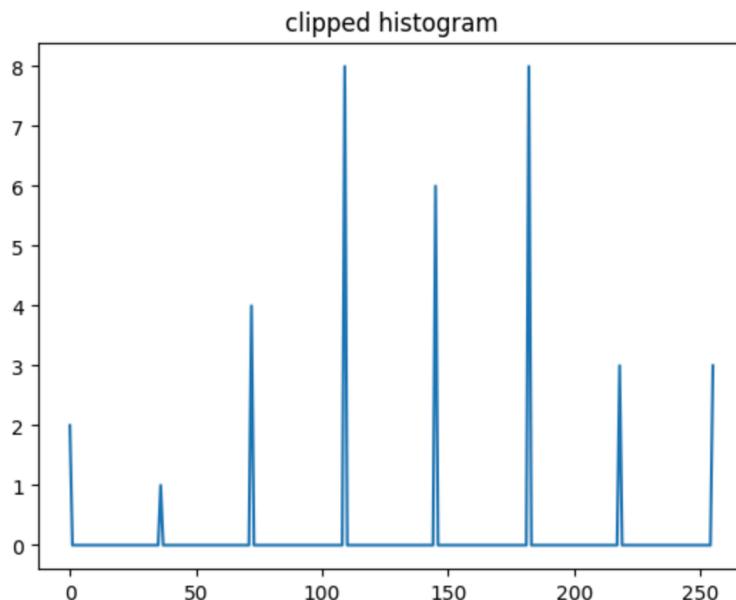
```
# Calculate the scale factor for linear scaling
scale_factor = 255.0 / (max_value - min_value)

# Create a mask for pixels within the clipping range
lower_mask = image >= min_value
upper_mask = image <= max_value

# Initialize the output image with the scaled values where the mask is True
output_image = (image - min_value) * scale_factor * lower_mask * upper_mask

# Set clipped values to 0 or 255 where the mask is False
output_image[image < min_value] = 0
output_image[image > max_value] = 255
```

حال اگر خروجی را ببینیم نمودار برش هیستوگرام دقیقاً مانند نموداری است که در بخش الف محاسبه کردیم.



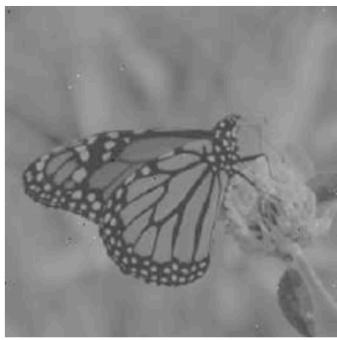
ج.

ابتدا تصویر را می‌خوانیم:

```
image = cv2.imread('image2.png', cv2.IMREAD_COLOR)
```

سپس با استفاده از `cv2.imshow` تصویر اولیه و حاصل کشش و برش هیستوگرام را نمایش می‌دهیم:

Original Image



Stretched Image



Clipped Image



کشش هیستوگرام:

در کشش هیستوگرام، تمام محدوده داده‌های تصویر (مقادیر پیکسل‌ها) به گونه‌ای تغییر می‌کنند که کل گستره ممکن (مثلاً از ۰ تا ۲۵۵ برای تصاویر ۸ بیتی) را پوشش دهند. این روش برای تصاویری که دامنه کنترast محدودی دارند مفید است، زیرا باعث می‌شود جزئیات در مناطق تاریک و روشن بهتر دیده شوند. نتیجه کشش هیستوگرام این است که جزئیات بیشتری از تصویر در تمام محدوده دیداری ما قابل مشاهده می‌شوند و تصویر "روشن‌تر" به نظر می‌رسد.

کلیپینگ هیستوگرام:

در کلیپینگ هیستوگرام، فقط مقادیر پیکسل‌هایی که خارج از محدوده تعیین شده‌اند (به عنوان مثال، در صد بسیار کمی از پیکسل‌های خیلی تیره یا خیلی روشن) تغییر می‌کنند. این روش برای تصحیح پیکسل‌هایی با شدت‌های بسیار بالا یا بسیار پایین به کار می‌رود که ممکن است جزئیات تصویر را بپوشانند. نتیجه کلیپینگ هیستوگرام این است که کنترast کلی تصویر بهبود پیدا می‌کند و جزئیات در مناطق بسیار روشن و تیره بهتر قابل مشاهده می‌شوند.

خروجی‌های متفاوت:

با کشش هیستوگرام، تصویر نهایی دارای کنتراست بیشتری خواهد بود و محدوده وسیع تری از شدت‌ها را نشان می‌دهد. این کار می‌تواند باعث شود که جزئیات ناپدید شده در مناطق تاریک یا روشن بیشتر به چشم بیایند.

با کلیپینگ هیستوگرام، تصویر نهایی تمرکز بیشتری بر روی محدوده خاصی از شدت‌ها دارد و می‌تواند به حذف نویز یا تأثیرات ناخواسته نوری کمک کند.

.۲
الف.

Src	Pixel	Histogram	Equalized	(الف)
0	3	3	3	
1	32	40		
2	24	64		
Ref				
2	8	8		
3	8	16		
4	8	24		
5	16	40		
6	8	48		
7	16	64		
Mapping :	Src	ref		
	0 → 8	8 → 2		
	1 → 40	40 → 5		
	2 → 64	64 → 7		
Src after mapping:				
Y Y Y Y Y	Y Y Y			
Δ Δ Δ Δ Δ	Δ Δ Δ			
Δ Δ Δ Δ Δ	Δ Δ Δ			
V V V V V	V V V			
V V V V V	V V V			
V V V V V	V V V			
Δ Δ Δ Δ Δ	Δ Δ Δ			
Δ Δ Δ Δ Δ	Δ Δ Δ			

.ب

تابع calc_hist

```
# Initialize the histogram array with zeros
hist = np.zeros(256, dtype=int)

# Calculate histogram using np.bincount and ensure it has a length of 256
hist = np.bincount(image.flatten(), minlength=256)
```

مقداردهی اولیه هیستوگرام: در ابتدای کد، یک آرایه به طول ۲۵۶ ساخته می‌شود که هر عنصر آن نشان‌دهنده تعداد پیکسل‌های تصویر باشد روشنایی معین (از ۰ تا ۲۵۵) است. این آرایه با صفر مقداردهی اولیه می‌شود که بیانگر این است که هنوز هیچ پیکسلی شمارش نشده است.

محاسبه هیستوگرام: برای محاسبه هیستوگرام، از تابع `np.bincount` استفاده می‌کنیم. این تابع، تعداد وقوع هر مقدار در آرایه ورودی را محاسبه می‌کند. با استفاده از `image.flatten()`، تصویر دو بعدی به یک آرایه یک بعدی تبدیل می‌شود تا بتوان تعداد وقوع هر شدت روشنایی را به دست آورد. `minlength=256` اطمینان می‌دهد که آرایه خروجی حداقل ۲۵۶ عنصر دارد، پوشش داده شدن تمام مقادیر ممکن برای شدت‌های پیکسلی.

تابع calc_cdf

```
# Use calc_hist to calculate the histogram of the channel
hist = calc_hist(channel)

# Calculate the CDF by cumulatively summing the histogram values
cdf = np.cumsum(hist)

# Normalize the CDF to make its last element equal to 1
cdf_normalized = cdf / cdf.max()
```

محاسبه هیستوگرام کanal: ابتدا، با استفاده از تابع `calc_hist` که پیش‌تر توضیح داده شد، هیستوگرام کanal تصویر ورودی محاسبه می‌شود. این هیستوگرام نشان‌دهنده تعداد پیکسل‌ها در هر سطح شدت روشنایی در آن کanal است.

محاسبه CDF: پس از به دست آوردن هیستوگرام، تابع توزیع تجمعی (CDF) را با جمع‌زنی تجمعی مقادیر هیستوگرام محاسبه می‌کنیم. این عمل باعث می‌شود که هر نقطه در نشان‌دهنده مجموع تمام مقادیر هیستوگرام تا آن نقطه باشد، که این مجموع تجمعی نمایانگر توزیع تجمعی شدتهای روشنایی در کanal است.

نرمال‌سازی CDF: برای استفاده آسان‌تر و استاندارد کردن CDF، آن را نرمال‌سازی می‌کنیم به گونه‌ای که عنصر آخر آن برابر با ۱ شود. این کار با تقسیم تمام مقادیر CDF بر بزرگ‌ترین مقدار در آن (که معمولاً مجموع تمام پیکسل‌ها است) انجام می‌شود. نتیجه، یک CDF نرمال‌شده است که مقادیر آن در بازه $[0, 1]$ قرار دارند.

تابع hist_matching

```
for channel, title in channels:
    # Calculate the Cumulative Distribution Function (CDF) of the source image's current channel.
    cdf_src_image = calc_cdf(src_image[:, :, channel])

    # Calculate the CDF of the reference image's corresponding channel.
    cdf_ref_image = calc_cdf(ref_image[:, :, channel])

    # This maps each pixel intensity in the source to a new intensity such that
    # the CDF of the source matches the CDF of the reference.
    res = np.interp(cdf_src_image, cdf_ref_image, np.arange(256))

    # Apply the mapping to the source image's current channel,
    # effectively transforming the source image to match the reference histogram.
    output_image[:, :, channel] = res[src_image[:, :, channel]]
```

محاسبه CDF برای کانال‌های تصویر منبع و مرجع: برای هر کانال از تصویر منبع و مرجع، تابع توزیع تجمعی (CDF) محاسبه می‌شود. این کار با استفاده از تابع `calc_cdf` انجام می‌شود که پیش‌تر توضیح داده شد.

ایجاد نگاشت میان CDF‌ها: برای هر پیکسل در هر کانال تصویر منبع، یک نگاشت جدید بر اساس مقایسه CDF‌های تصویر منبع و مرجع ایجاد می‌شود. این نگاشت به گونه‌ای است که CDF تصویر منبع با CDF تصویر مرجع مطابقت پیدا کند.

اعمال نگاشت به تصویر منبع: این نگاشت به کانال‌های مربوطه از تصویر منبع اعمال می‌شود تا تصویر خروجی ایجاد شود که هیستوگرام آن با تصویر مرجع مطابقت دارد.

تحلیل خروجی‌ها:

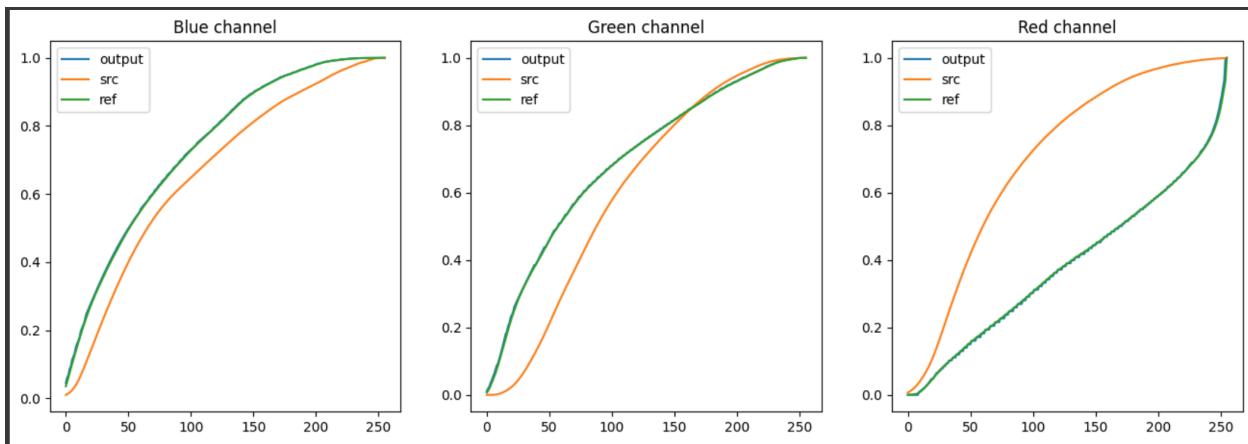


همانطور که از خروجی مشخص است، ظاهر کلی یا سبک بصری تصویر خروجی تحت تأثیر تصویر مرجع می‌باشد.

در حالی که هیستوگرام تصویر منبع تغییر می‌کند، محتوای تصویر ثابت مانده است. هیچ عنصر جدیدی از صحنه در تصویر منبع اضافه یا حذف نشده است. در عوض، عناصر موجود ممکن است روشن تر، تیره تر یا با کنتراست تغییر یافته به نظر برسند تا با هیستوگرام تصویر مرجع مطابقت داشته باشند.

تغییر رنگ: برای تصاویر رنگی، تطبیق هیستوگرام اغلب برای هر کanal رنگی به طور مستقل اعمال می‌شود (به عنوان مثال، کanal های قرمز، سبز و آبی در یک تصویر RGB). این می‌تواند منجر به تغییر رنگ در تصویر منبع شود تا با تن رنگ کلی تصویر مرجع مطابقت داشته باشد. به عنوان مثال، اگر تصویر مرجع دارای تن گرم باشد، تصویر خروجی نیز ممکن است به سمت رنگ های گرم تر تغییر کند.

به طور خلاصه اشکال تصویر source حفظ شده‌اند و رنگ‌های آن به رنگ‌های تصویر reference تغییر یافته‌اند.



همانطور که از تصویر فوق مشخص است، خروجی و مرجع عملکرد مشابهی در کanal های رنگی خود دارند و با تصویر منبع متفاوتند. اما چرا؟

هنگامی که تطبیق هیستوگرام با استفاده از یک تصویر مرجع به تصویر منبع اعمال می‌شود، هدف این است که تصویر منبع را تنظیم کنیم تا هیستوگرام آن برای هر کanal RGB با تصویر

مرجع مطابقت داشته باشد. این فرآیند شدت پیکسل ها را در تصویر منبع تغییر می دهد و تصمین می کند که توزیع این شدت ها در کانال های قرمز، سبز و آبی دقیقاً منعکس کننده توزیع موجود در تصویر مرجع است. در نتیجه، زمانی که کانال های RGB منبع، مرجع و تصاویر خروجی رسم می شوند، نمودارهای تصویر مرجع (reference) و خروجی به خوبی در یک راستا قرار می گیرند که نشان دهنده تطبیق موفقیت آمیز هیستوگرام است. این تراز به دلیل توزیع مجدد شدت پیکسل در تصویر منبع برای مطابقت با شدت پیکسل در تصویر مرجع در هر کانال است.

.۳
الف.

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

نتیجه تابع فوق:



استفاده از `cv2.equalizeHist` عمدهاً به دلیل رویکرد سراسری آن در تنظیم کنترast تصویر با چالش‌هایی مواجه است که می‌تواند منجر به چندین مشکل شود. ابتدا، برای تصاویر با شرایط نورپردازی متفاوت در مناطق مختلف ایده‌آل نیست، زیرا همان سطح تنظیم را در سراسر تصویر اعمال می‌کند، که ممکن است منجر به از دست دادن جزئیات در نقاط روشن و تاریک شود. این روش انعطاف‌پذیری محدودی دارد، بدون کنترل بر شدت یا مکانیزم **equalization**، که آن را برای تنظیمات ظرفی کمتر مناسب می‌کند. علاوه بر این، می‌تواند نویز را در مناطق با کنترast پایین یا یکنواخت تقویت کند. در صحنه‌های پیچیده، ماهیت سراسری این **equalization** ممکن است به طور یکنواخت کنترast را بهبود نبخشد، منجر به یک نتیجه بصری نامتوازن می‌شود. جایگزین‌هایی مانند تساوی هیستوگرام تطبیقی (ACE) و تساوی هیستوگرام تطبیقی با محدودیت کنترast (CLAHE) به دلیل توانایی آنها در ارائه بهبودهای کنترast محلی و مدیریت بهتر شرایط متنوع تصویر، اغلب ترجیح داده می‌شوند.

```
# Iterate over the image with the specified grid size
for i in range(0, x, gridSize):
    for j in range(0, y, gridSize):
        # Define the region (grid) to apply histogram equalization
        grid = output_image[i:min(i + gridSize, x), j:min(j + gridSize, y)]

        # Apply histogram equalization to the grid
        equalized_grid = cv2.equalizeHist(grid)

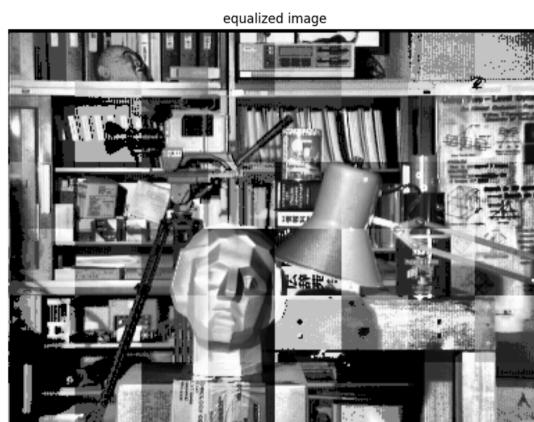
        # Place the equalized grid back into the output image
        output_image[i:min(i + gridSize, x), j:min(j + gridSize, y)] = equalized_grid
```

ایتریشن روی تصویر: این کد در حلقه‌های تکرار برای ایجاد گریدها یا پنجره‌های کوچک بر اساس gridSize روی تصویر ورودی اجرا می‌شود. برای هر گرید، تساوی هیستوگرام با استفاده از تابع cv2.equalizeHist() اعمال می‌شود.

تساوی هیستوگرام در هر گرید: برای هر گرید مشخص شده، تابع cv2.equalizeHist() هیستوگرام را تساوی می‌کند. این به معنای بازنمایی بازنمایی بازنمایی در هر گرید برای بهبود کنتراست است.

بازگذاری گریدهای تساوی شده: گریدهایی که هیستوگرام آنها تساوی شده، دوباره در تصویر خروجی، در موقعیت‌های مربوط به خود قرار می‌گیرند.

خروجی: همانطور که از شکل زیر مشخص است خروجی این تابع، یک تصویر بهبود یافته است که در آن کنتراست به صورت محلی درون گریدها یا پنجره‌های کوچک تعیین شده بر اساس gridSize افزایش می‌یابد. این روش به ویژه برای تصاویری با نورپردازی نامناسب یا کنتراست پایین که جزئیات آنها به خوبی قابل تشخیص نیست، مفید است. با اعمال تساوی هیستوگرام به صورت محلی، هر بخش از تصویر به گونه‌ای تنظیم می‌شود که بهترین کنتراست ممکن را داشته باشد، بدون آنکه به قسمت‌های دیگر تصویر آسیبی برسد. این امر منجر به افزایش وضوح و شفافیت بصری در تصویر خروجی می‌شود، به طوری که جزئیات ریز و بافت‌های تصویر بهتر و واضح‌تر قابل مشاهده می‌شوند.



ACE۲

```
# Get image dimensions
x, y = image.shape

# Compute half the grid size for easier navigation
half_grid = gridSize // 2

# Pad the image using reflect101 padding
padded_image = cv2.copyMakeBorder(output, half_grid, half_grid, half_grid, half_grid,
cv2.BORDER_REFLECT_101)

# Iterate through each pixel in the output image
for i in range(half_grid, x + half_grid):
    for j in range(half_grid, y + half_grid):
        # Extract the local region around the current pixel
        local_region = padded_image[i - half_grid:i + half_grid + 1, j - half_grid:j + half_grid + 1]

        # Apply histogram equalization to this local region
        local_eq = cv2.equalizeHist(local_region)

        # Set the pixel in the output image to the center pixel in the equalized local region
        output[i - half_grid, j - half_grid] = local_eq[half_grid, half_grid]

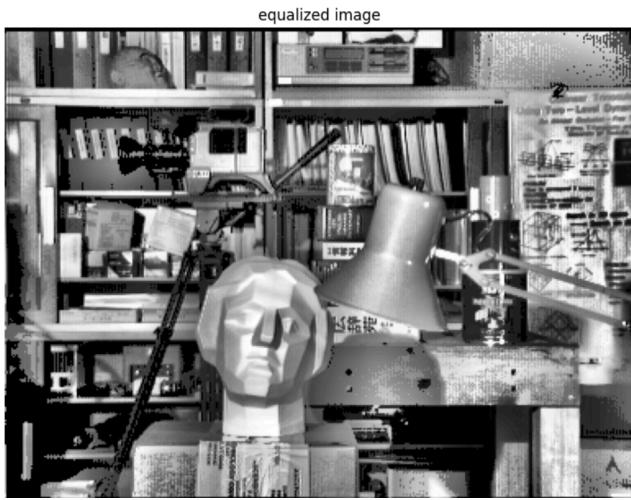
return output
```

بدست آوردن ابعاد تصویر: ابعاد تصویر برای تعیین محدوده ایتریشن‌ها محاسبه می‌شود. محاسبه نیمه اندازه گرید: نصف اندازه گرید محاسبه می‌شود تا **navigation** در اطراف پیکسل‌ها و استخراج مناطق محلی راحت‌تر انجام شود.

پدینگ تصویر: برای جلوگیری از بروز مشکلات در لبه‌های تصویر هنگام استخراج مناطق محلی، تصویر با استفاده از پدینگ **reflect101** پدینگ می‌شود. این امر اطمینان حاصل می‌کند که برای هر پیکسل در تصویر، یک منطقه محلی معتبر جهت اعمال تساوی هیستوگرام وجود دارد. ایتریشن بر روی هر پیکسل: برای هر پیکسل در تصویر خروجی، یک منطقه محلی با استفاده از پنجره‌ای به اندازه **gridSize** استخراج شده و تساوی هیستوگرام بر روی آن منطقه اعمال می‌شود.

اعمال تساوی هیستوگرام محلی: برای هر منطقه محلی استخراج شده، تساوی هیستوگرام انجام شده و سپس پیکسل مرکزی از این منطقه تساوی شده به عنوان مقدار جدید پیکسل در تصویر خروجی قرار داده می‌شود.

خرожی: تصویر خروجی از اجرای این تابع، یک تصویر بهبود یافته است که در آن کنتراست محلی به صورت قابل توجهی در سراسر تصویر افزایش می‌یابد. این افزایش کنتراست محلی از طریق اعمال تساوی هیستوگرام بر روی مناطق کوچک‌تر اطراف هر پیکسل به دست می‌آید، به جای تمرکز بر روی بلوک‌های بزرگ‌تر. نتیجه، بهبود در وضوح و شفافیت بصری تصویر خروجی است، جایی که جزئیات ریز و بافت‌ها بهتر و واضح‌تر قابل مشاهده می‌شوند. این روش مخصوصاً برای تصاویری که در بخش‌هایی از آن نورپردازی نامناسب یا کنتراست پایین دارند، مفید است، زیرا امکان بهبود دیداری بدون افراط در اعمال تغییرات بر روی کل تصویر را فراهم می‌کند.



CLAHE

کپی تصویر ورودی: این کد با کپی کردن تصویر ورودی آغاز می‌شود تا تمام تغییرات بر روی کپی اعمال شوند و تصویر اصلی تغییر نکند.

محاسبه محدوده کلیپ مؤثر: برای هر گرید، یک محدوده کلیپ مؤثر بر اساس `clip_limit` و اندازه گرید محاسبه می‌شود. این امر به جلوگیری از افزایش بیش از حد کنترast در هر گرید کمک می‌کند.

پدینگ تصویر: تصویر با استفاده از پدینگ `BORDER_REFLECT_101` آماده سازی می‌شود تا در حین استخراج گریدها، مشکلی از بابت لبه‌های تصویر به وجود نیاید. پیمایش گریدها و اعمال تساوی هیستوگرام: برای هر گرید، هیستوگرام محاسبه شده، سپس با استفاده ازتابع `clip_histogram` کلیپ می‌شود. برای محاسبه تابع توزیع تجمعی (CDF) از هیستوگرام کلیپ شده استفاده می‌شود.

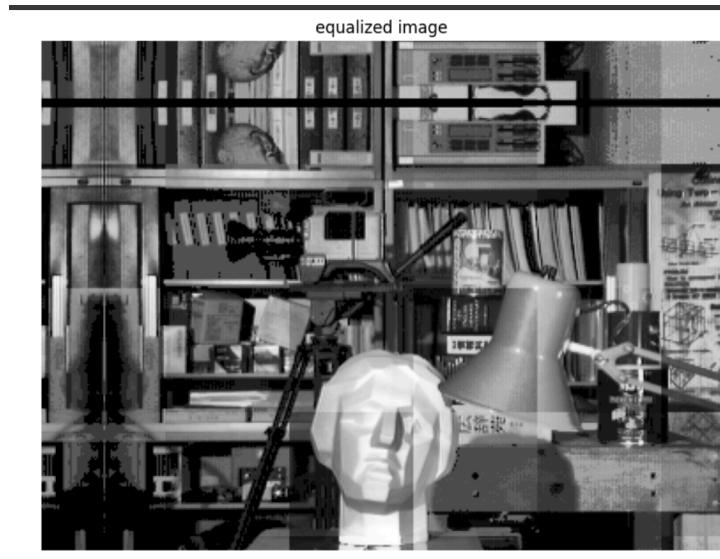
نگاشت مجدد مقادیر پیکسل‌ها: با استفاده از CDF محاسبه شده، مقادیر پیکسل‌ها در هر گرید به مقادیر جدیدی نگاشت می‌شوند که کنتراست بهبود یافته‌ای را نشان می‌دهند. این مقادیر جدید به تصویر خروجی اعمال می‌شوند. کد مد نظر داخل نوتبوک ضمیمه شده است.

خروجی:

همانطور که از تصویر پایین مشخص است، خروجی تابع CLAHE، تصویری با کنتراست بهینه‌سازی شده در سطح محلی است. با استفاده از این روش، مناطقی از تصویر که کنتراست پایینی دارند، به گونه‌ای تقویت می‌شوند که جزئیات بیشتری در آن‌ها قابل مشاهده باشد، در حالی که از افزایش بی‌رویه کنتراست که می‌تواند به از دست رفتن جزئیات منجر شود، جلوگیری می‌کند. این امر به ویژه در مناطق تاریک و روشن تصویر که ممکن است جزئیات به دلیل نورپردازی نامناسب از دست رفته باشند، کاربرد دارد.

روش CLAHE با اعمال محدودیت بر هیستوگرام و توزیع مجدد دقیق تر مقادیر پیکسل‌ها در سطح محلی، امکان مشاهده بهتر تفاوت‌های روشنایی و تیرگی را فراهم می‌کند. در نتیجه، تصویر

خروجی نه تنها دارای کنتراست بهبود یافته است، بلکه بافت‌ها و جزئیات بیشتری را نیز در سراسر تصویر نشان می‌دهد.



مقایسه:

CLAHE (تساوی هیستوگرام تطبیقی با محدودیت کنترast):

مزایا:

بهبود کنتراست محلی با حفظ جزئیات در مناطق روشن و تاریک.
جلوگیری از افزایش بیش از حد کنتراست که می‌تواند به از دست رفتن جزئیات منجر شود.
قابلیت اطمینان و عملکرد خوب برای تصاویر با کنتراست پایین یا نورپردازی نامناسب.

معایب:

ممکن است در برخی مناطق به افزایش نویز منجر شود.
نیاز به تنظیم دقیق پارامترها (مانند اندازه گرید و محدودیت کلیپ) برای دستیابی به نتایج بهینه.

ACE۱ (تساوی هیستوگرام تطبیقی - روش اول):

مزایا:

سادگی پیاده‌سازی و اعمال تساوی هیستوگرام بر اساس گریدهای تعریف شده.
مناسب برای تصاویر با نیاز به بهبود کنتراست یکنواخت در سراسر تصویر.

معایب:

کمتر انعطاف‌پذیر نسبت به **CLAHE** در جلوگیری از افزایش بیش از حد کنتراست.
ممکن است در برخی مناطق به کاهش جزئیات یا ظهر نویز منجر شود.

ACE۲ (تساوی هیستوگرام تطبیقی - روش دوم):

مزایا:

افزایش دقت در بهبود کنتراست محلی با تمرکز بر روی هر پیکسل.
قابلیت بهبود جزئیات در سطح پیکسل، که برای تصاویر با جزئیات پیچیده مفید است.

معایب:

پیچیدگی و زمان محاسباتی بیشتر نسبت به ACE و CLAHE به دلیل نیاز به پردازش محلی پیچیده‌تر. ممکن است به تنظیم دقیق‌تر پارامترها برای جلوگیری از تأثیرات ناخواسته نیاز داشته باشد.

.ج

CELAHE by opencv

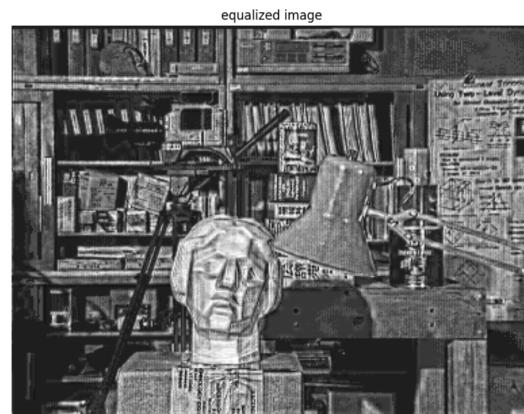
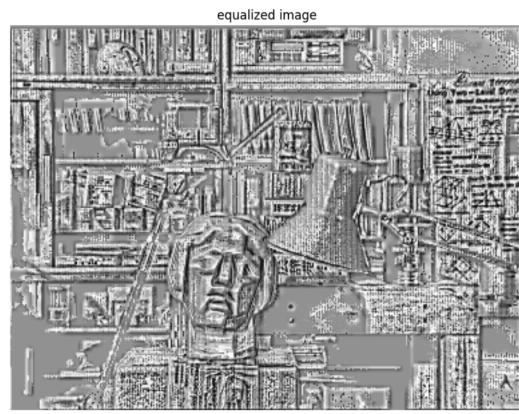
```
# Create a CLAHE object with specified gridSize and clipLimit
clahe = cv2.createCLAHE(clipLimit=clipLimit, tileGridSize=gridSize)

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

ایجاد یک شی CLAHE: اولین قدم، ایجاد یک شی CLAHE با استفاده از تابع cv2.createCLAHE است. این شی با مقادیر clipLimit و tileGridSize که از ورودی‌ها دریافت می‌شود، تنظیم می‌شود. این شی با کنتراست محدودیت clipLimit برای جلوگیری از افزایش بیش از حد کنتراست در گریدها را تعیین می‌کند، در حالی که tileGridSize اندازه هر گرید (پنجره) برای اعمال تساوی هیستوگرام را مشخص می‌کند. اعمال CLAHE apply شی CLAHE با استفاده از متدهای تساوی هیستوگرام تطبیقی روی تصویر ورودی اعمال می‌شود. این عمل باعث بهبود کنتراست محلی در سراسر تصویر می‌شود، بدون آنکه به جزئیات ظرفی تصویر آسیب برسد.

خروجی‌ها:





با تغییر مقادیر `clipLimit` و `gridSize` در تابع `CLAHE`, خروجی‌های مختلفی به دست می‌آید که هر کدام ویژگی‌های خاص خود را دارند.

$$\bullet \quad \text{output_image1} = \text{CLAHE}(\text{image}, (128, 128))$$

این خروجی با استفاده از گریدهای بزرگ (128×128 پیکسل) و یک محدودیت کنترast نسبتاً پایین (۲) ایجاد شده است. گریدهای بزرگ‌تر منجر به تساوی هیستوگرام در ناحیه‌های وسیع‌تر می‌شوند که می‌تواند به یکنواختی بیشتر در سراسر تصویر کمک کند، اما با کنترast کمتری نسبت به گریدهای کوچک‌تر. محدودیت کنترast پایین‌تر باعث می‌شود که تغییرات کنترast ملایم‌تر باشند.

$$\bullet \quad \text{output_image2} = \text{CLAHE}(\text{image}, (128, 128))$$

این تنظیم با همان اندازه گرید به عنوان `output_image1` اما با محدودیت کنترast بسیار بالاتر (۱۲۸) انجام شده است. این امر اجازه می‌دهد کنترast در هر گرید تا حد زیادی افزایش یابد، منجر به تصویری با کنترast بیشتر می‌شود، خصوصاً در مناطق با تفاوت‌های روشنایی بالا.

$$\bullet \quad \text{output_image3} = \text{CLAHE}(\text{image}, (16, 16))$$

با کاهش اندازه گرید به 16×16 پیکسل و حفظ همان محدودیت کنترast کم (۲)، این تنظیم اجازه می‌دهد که تساوی هیستوگرام به صورت بسیار محلی‌تر اعمال شود. این امر می‌تواند به افزایش دقیق در نمایش جزئیات کوچک و تقویت کنترast محلی کمک کند، اما با تغییرات کنترast نسبتاً محافظه کارانه.

$$\bullet \quad \text{output_image4} = \text{CLAHE}(\text{image}, (16, 16))$$

ترکیب گریدهای کوچک (16×16 پیکسل) با یک محدودیت کنترast بالا (۱۲۸) در `output_image4` منجر به ایجاد قوی‌ترین تغییرات کنترast می‌شود. این روش باعث افزایش چشمگیر کنترast محلی در سراسر تصویر می‌گردد و می‌تواند به بهبود قابل توجه در جزئیات و واضح تصویر کمک کند. این تنظیمات به ویژه برای نمایش واضح بهتر در مناطق با جزئیات پیچیده و کوچک مناسب است، اما ممکن است در برخی موارد به افزایش بیش از حد نویز یا ایجاد تأثیرات ناخواسته در کنترast تصویر منجر شود، به خصوص اگر تصویر از پیش دارای کنترast بالایی باشد.

الف.تابع Add_Noise

```
# Define the ratio of salt to pepper and the total amount of noise
salt_pepper_ratio = 0.5
amount = 0.04

# Copy the input image to avoid overwriting the original image
noisy_img = np.copy(img)

# Calculate the number of pixels to alter
num_salt = np.ceil(amount * img.size * salt_pepper_ratio)
num_pepper = np.ceil(amount * img.size * (1.0 - salt_pepper_ratio))

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

# Add pepper noise (black pixels)
coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in img.shape]
noisy_img[tuple(coords)] = 0
```

تعريف نسبت نمک به فلفل و مقدار کل نویز: در ابتدا، نسبت بین نمک (پیکسل‌های سفید) و فلفل (پیکسل‌های سیاه) به همراه میزان کلی نویز که قرار است به تصویر اضافه شود، تعیین می‌گردد. نسبت نمک به فلفل ۰.۵ و میزان کل نویز ۰.۰۴ (یعنی ۴ درصد از پیکسل‌ها) در نظر گرفته شده است.

کپی کردن تصویر ورودی: برای جلوگیری از تغییر تصویر اصلی، یک کپی از تصویر ورودی ایجاد می‌شود که تغییرات روی آن اعمال خواهد شد.

محاسبه تعداد پیکسل‌هایی که باید تغییر کنند: بر اساس میزان کل نویز و نسبت تعیین شده، تعداد دقیق پیکسل‌هایی که باید به نمک یا فلفل تبدیل شوند، محاسبه می‌گردد.

اضافه کردن نویز نمک: با انتخاب تصادفی مختصات، تعداد مشخصی پیکسل به طور تصادفی به رنگ سفید (۲۵۵) تغییر می‌کنند تا نویز نمک اضافه شود.

اضافه کردن نویز فلفل: به طریق مشابه، با انتخاب تصادفی مختصات، تعدادی پیکسل به طور تصادفی به رنگ سیاه (۰) تغییر می‌کنند تا نویز فلفل اضافه شود.

خروجی:



reflect101

```

pad_size = filter_size // 2
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

# Edges
padded_img[pad_size:-pad_size, :pad_size] = img[:, :pad_size][:, ::-1] # Left
padded_img[pad_size:-pad_size, -pad_size:] = img[:, -pad_size:][:, ::-1] # Right
padded_img[:pad_size, pad_size:-pad_size] = img[:pad_size, ::-1, :] # Top
padded_img[-pad_size:, pad_size:-pad_size] = img[-pad_size:, ::-1, :] # Bottom

# Corners
padded_img[:pad_size, :pad_size] = img[:pad_size, :pad_size][::-1, ::-1] # Top-left
padded_img[-pad_size:, :pad_size] = img[-pad_size:, :pad_size][::-1, ::-1] # Bottom-left
padded_img[:pad_size, -pad_size:] = img[:pad_size, -pad_size:][::-1, ::-1] # Top-right
padded_img[-pad_size:, -pad_size:] = img[-pad_size:, -pad_size:][::-1, ::-1] # Bottom-right

```

تعیین اندازه پدینگ: ابتدا، اندازه پدینگ بر اساس اندازه فیلتر محاسبه می‌شود. این اندازه نشان‌دهنده تعداد پیکسل‌هایی است که باید به هر طرف تصویر اضافه شود. ایجاد تصویر پدینگ شده خالی: یک تصویر خالی با ابعاد مورد نیاز برای نگه داشتن تصویر اصلی به همراه پدینگ ایجاد می‌شود. قرار دادن تصویر ورودی در مرکز: تصویر ورودی در مرکز تصویر خالی قرار داده می‌شود. انعکاس پیکسل‌ها برای ایجاد پدینگ: پیکسل‌های لبه سمت چپ و راست تصویر ورودی در طرفین تصویر پدینگ شده منعکس می‌شوند. پیکسل‌های لبه بالا و پایین تصویر ورودی در بالا و پایین تصویر پدینگ شده منعکس می‌شوند. گوشه‌های تصویر نیز به صورت مستقل منعکس می‌شوند تا پدینگ کامل شود.

Average Blurring

```

image = Reflect101(img, filter_size)
result = np.zeros(img.shape)

pad_size = filter_size // 2

# Iterate over each pixel of the original image size
for i in range(pad_size, image.shape[0] - pad_size):
    for j in range(pad_size, image.shape[1] - pad_size):
        # Compute the average value for the current window
        window = image[i - pad_size:i + pad_size + 1, j - pad_size:j + pad_size + 1]
        result[i - pad_size, j - pad_size] = np.sum(window) / (filter_size ** 2)

```

اعمال پدینگ ۱ Reflect101: ابتدا با استفاده از تابع Reflect101، پدینگ انعکاسی به تصویر اضافه می‌شود. این کار امکان پردازش لبه‌های تصویر را فراهم می‌آورد.

محاسبه میانگین برای هر پیکسل: سپس برای هر پیکسل در تصویر ورودی، یک پنجره به اندازه `filter_size` در نظر گرفته شده و میانگین مقادیر پیکسل‌های درون این پنجره محاسبه می‌گردد. این میانگین به عنوان مقدار جدید برای پیکسل در نظر گرفته می‌شود. تکرار برای کل تصویر: این فرآیند برای تمام پیکسل‌های تصویر انجام می‌شود تا کل تصویر با افکت تاری کردن میانگین پردازش شود.

خروجی:

خروجی این تابع، تصویری است که در آن تاری کردن میانگین اعمال شده، نتیجه در کاهش پسح و وضعیت "تار" تصویر نسبت به تصویر ورودی است. این روش می‌تواند برای کاهش جزئیات و نویز در تصویر مفید باشد، اما همچنین ممکن است باعث از دست رفتن پسح و جزئیات مهم در تصویر شود. تاری کردن میانگین به طور کلی برای ایجاد افکت‌های بصری خاص یا آماده‌سازی تصاویر قبل از پردازش‌های بعدی استفاده می‌شود.

Averaging Blurring



Median Bluring

```
pad_size = filter_size // 2

# Iterate over each pixel of the original image size
for i in range(pad_size, image.shape[0] - pad_size):
    for j in range(pad_size, image.shape[1] - pad_size):
        # Extract the current window
        window = image[i - pad_size:i + pad_size + 1, j - pad_size:j + pad_size + 1]
        # Find the median of the current window and assign it to the corresponding pixel in result
        result[i - pad_size, j - pad_size] = np.median(window.flatten())
```

پدینگ تصویر: ابتدا، با استفاده از تابع `Reflect101`، پدینگ انعکاسی به تصویر اضافه می‌شود تا هنگام اعمال فیلتر میانه بر روی لبه‌های تصویر، داده کافی وجود داشته باشد. این امر به محاسبه میانه برای پیکسل‌های لبه تصویر امکان‌پذیر می‌کند.

محاسبه میانه برای هر پیکسل: سپس، برای هر پیکسل در تصویر ورودی، یک پنجره به اندازه `filter_size` اطراف آن پیکسل در نظر گرفته می‌شود و میانه مقادیر پیکسل‌های درون این پنجره محاسبه می‌گردد. مقدار میانه به عنوان مقدار جدید آن پیکسل در تصویر خروجی انتخاب می‌شود.

تکرار برای کل تصویر: این فرآیند برای تمام پیکسل‌های تصویر تکرار می‌شود تا تصویر کامل با افکت تار کردن میانه پردازش شود.

خروجی: خروجی این تابع، تصویر زیر که در آن نویز به طور مؤثری کاهش یافته، اما لبه‌ها و جزئیات مهم تصویر به خوبی حفظ شده‌اند. تاری کردن میانه به ویژه برای از بین بردن نویز نمک و فلفل یا نویزهای دیگری که ممکن است در تصویر وجود داشته باشند، بسیار مفید است. در نتیجه، تصویر خروجی نسبت به تصویر ورودی دارای ظاهری صاف‌تر و یکنواخت‌تر است، در حالی که ویژگی‌های اصلی و جزئیات کلیدی تصویر به خوبی قابل تشخیص باقی می‌مانند.

Median Blurring



Gaussian_Blurring

```
sum_val = 0

# Calculate kernel values
for x in range(filter_size):
    for y in range(filter_size):
        ex = 2 * np.pi * std * std
        kernel[x, y] = np.exp(-(x - (filter_size - 1) / 2) ** 2 + (y - (filter_size - 1) / 2) ** 2) / (2 * std * std) / ex
        sum_val += kernel[x, y]

# Normalize the kernel
kernel /= sum_val

# Apply Gaussian filter using OpenCV
output = img.copy()
result = cv2.filter2D(src=output, ddepth=-1, kernel=kernel)
```

ایجاد کرنل گوسی: ابتدا یک کرنل خالی به اندازه `filter_size` ایجاد می‌شود. سپس مقادیر این کرنل با استفاده از فرمول توزیع گوسی محاسبه و در آن قرار داده می‌شوند. مقادیر کرنل بر اساس فاصله از مرکز کرنل و انحراف معیار (`std`) تعیین می‌شوند.

نرمال‌سازی کرنل: مقادیر درون کرنل به گونه‌ای نرمال‌سازی می‌شوند که مجموع تمام مقادیر برابر با ۱ شود. این امر اطمینان حاصل می‌کند که کرنل تاثیری بر روشنایی کلی تصویر نخواهد داشت. اعمال فیلتر گوسی: کرنل گوسی محاسبه شده بر روی تصویر ورودی اعمال می‌شود. این کار با محاسبه میانگین وزن‌دار پیکسل‌های اطراف هر پیکسل بر اساس مقادیر کرنل انجام می‌گیرد.

خروجی: خروجی، تصویری است که در آن تاری کردن گوسی اعمال شده، باعث می‌شود که نویز کاهش یابد و تصویر نرم‌تر و طبیعی‌تر به نظر برسد. تاری کردن گوسی به خصوص برای کاهش نویز دیجیتالی در تصاویر و همچنین به عنوان پیش‌پردازش برای الگوریتم‌های تشخیص لبه و دیگر تکنیک‌های پردازش تصویر کاربرد دارد. افکت تاری ایجاد شده با کرنل گوسی نسبت به روش‌های دیگر تاری کردن، نرم‌تر و طبیعی‌تر است، که باعث می‌شود برای طیف وسیعی از کاربردها مناسب باشد.

Gaussian Blurring



تاثیر اندازه کرنل:

تاری بیشتر با کرنل‌های بزرگ‌تر:

هرچه اندازه کرنل بزرگ‌تر باشد، تأثیر تاری کردن بر روی تصویر وسیع‌تر می‌شود. کرنل‌های بزرگ‌تر منطقه بزرگتری از پیکسل‌ها را در محاسبات میانگین یا میانه در بر می‌گیرند، که منجر به افزایش میزان تاری در خروجی می‌شود.

کاهش جزئیات با کرنل‌های بزرگ‌تر:

استفاده از کرنل‌های بزرگ‌تر می‌تواند به کاهش قابل توجه جزئیات و وضوح تصویر منجر شود، زیرا

تفاوت‌های محلی کوچک بین پیکسل‌ها در میانگین یا میانه گم می‌شوند.

افزایش نرمی و یکنواختی با کرنل‌های بزرگ‌تر:

کرنل‌های بزرگ‌تر تأثیری نرم‌تر و یکنواخت‌تر ایجاد می‌کنند، زیرا تفاوت‌های نوری و رنگی در مناطق وسیع‌تری از تصویر همگن می‌شوند. این امر می‌تواند برای کاهش نویز دیجیتالی یا برای ایجاد افکت‌های بصری خاص مفید باشد.

تاری کمتر با کرنل‌های کوچک‌تر:

کرنل‌های کوچک‌تر تأثیر تاری کمتری دارند و حفظ جزئیات بیشتری را امکان‌پذیر می‌سازند. این کرنل‌ها برای اعمال تغییرات نرم و ظریف بر تصویر مناسب هستند، بدون اینکه تأثیر قابل توجهی بر وضوح کلی تصویر داشته باشند.

.ج

```
filter_size = (15, 15)
std = 40
AveragingBlurring = cv2.blur(image, filter_size) ## your code here ##
MedianBlurring = cv2.medianBlur(image, filter_size[0]) ## your code here ##
GaussianBlurring = cv2.GaussianBlur(image, filter_size, std) ## your code here ##
```

با مقایسه خروجی‌ها، واضح است که توابع آماده نیز عملکرد یکسانی با توابعی که بالاتر پیاده‌سازی شد دارند و این امر تایید درستی پیاده‌سازی توابع در بخش ب می‌باشد.

