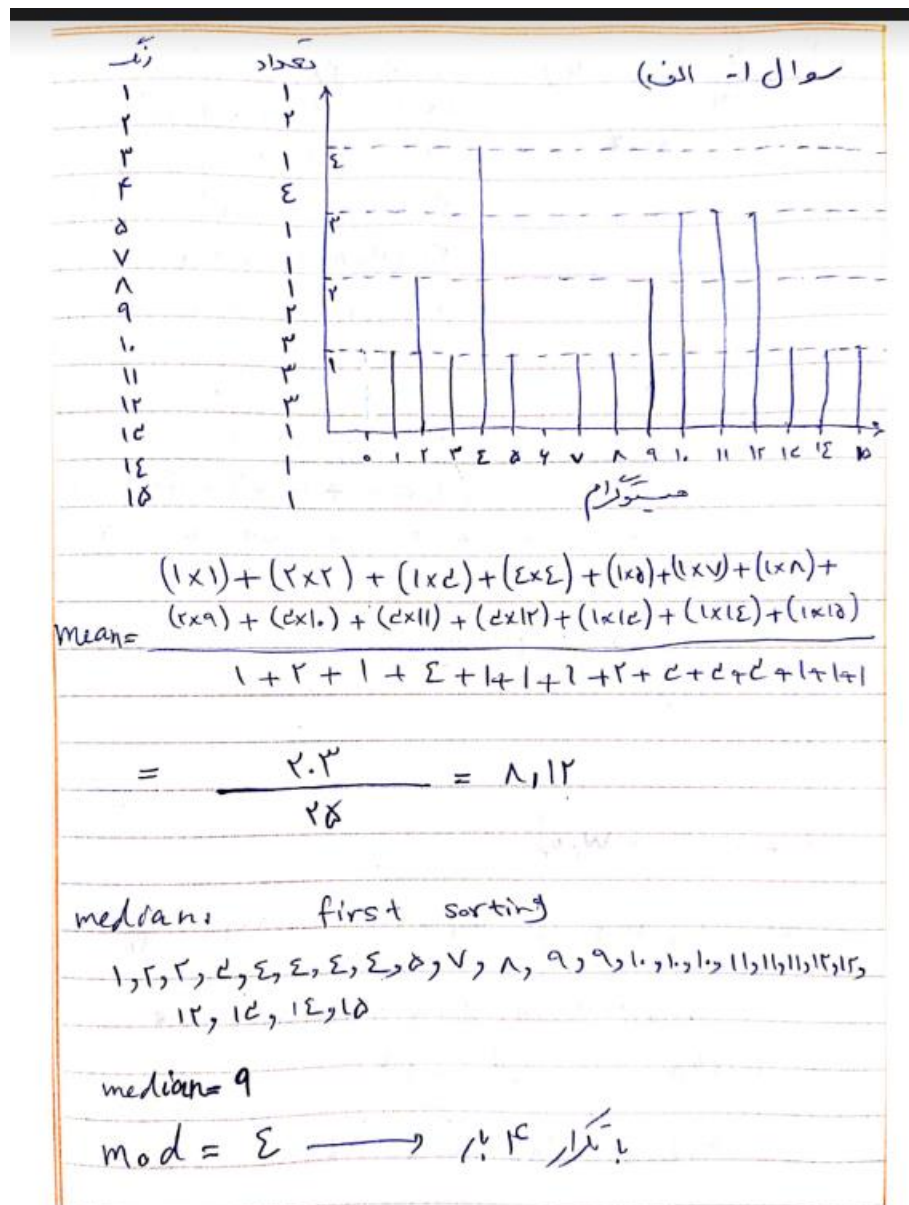


به نام خدا

تمرین سری چهارم
درس بینایی کامپیوتر

سینا علی‌نژاد

شماره دانشجویی: ۹۹۵۲۱۴۶۹



$$\sigma^2 = \frac{\sum_{i=1}^N (a_i - \mu)^2}{n} = \frac{(\mu - 1)^2 + 2 \times (\mu - 2)^2 + (\mu - 3)^2 + \dots + (\mu - 12)^2}{28}$$

$$= \frac{8,12944 + 74,9088 + 24,2144 + 47,8976 + 9,7224 + 1,2888 + 0,0144 + 1,8488 + 10,4032 + 25,8824 + 58,1424 + 22,8144 + 25,8752 + 27,2224}{28}$$

$$= \frac{518,44}{28} = 18,5157$$

(ب)

$$\sigma_w^2 = w_1 \sigma_1^2 + w_2 \sigma_2^2$$

thresh = 11,8 , $w_1 = \frac{19}{28}$, $w_2 = \frac{4}{28}$

• نکته: طبق رابطه بالا اگر آستانه را تغییر دهیم، می‌توانیم σ_1^2 و σ_2^2 را تغییر دهیم.

$$\sigma_1^2 = 12,328 \quad \sigma_2^2 = 1,4$$

$$\sigma_w^2 = 9,2 + 0,382 = 9,582$$

Scanned

Date: _____ Subject: _____

$$\text{thresh} = 9,5 \quad , \quad w_1 = \frac{12}{28} \quad w_2 = \frac{12}{28}$$

$$\sigma_1^2 = 7,19 \quad \sigma_2^2 = 2,548$$

$$\sigma_w^2 = 2,7588 + 1,2324 = 3,9912$$

هرچه معیار otsu به صفر نزدیکتر باشد، آستانه عددی
بهتری برای threshold است. اگر قرار بود بین این دو عدد
یکی را انتخاب کنیم، عدد ۹,۵ را انتخاب می‌کنیم.

$$3,9912 < 9,582$$









در روش Gaussian otsu میانگین هر دسته را بدست آورده که اختلاف به توان دوی این میانگین‌ها را واریانس بین دسته‌ای می‌گوییم و هرچه این میانگین‌ها اختلاف بیشتری داشته باشند، نشان‌دهنده این است که عدد threshold ما عدد بهتری بوده و معیار Gaussian otsu عدد بزرگتری خواهد شد. فرمول این روش بدین صورت است که وزنه‌های دو دسته که تعداد اعضای آن دسته نسبت به کل اعضا است، در هم ضرب شده و حاصل در توان دوی اختلاف میانگین دو دسته ضرب می‌شود. برای حالتی که میانگین هر دو دسته بالا باشند یعنی اختلافشان نزدیک به صفر شود، این معیار نزدیک صفر می‌شود که این یعنی مقدار خوبی برای threshold انتخاب نکردیم.

$$\sigma_B^2(t) = \sigma^2 - \sigma_\omega^2(t) = \omega_b(t) * (\mu_b(t) - \mu)^2 + \omega_f(t) * (\mu_f(t) - \mu)^2$$

$$= \omega_b(t) * \omega_f(t) * (\mu_b(t) - \mu_f(t))^2$$

الف) سرعت Gaussian otsu همواره بیشتر است زیرا اگر threshold را تغییر دهیم، بدست آوردن میانگین جدید دسته‌ها با کم و زیاد کردن مقادیری که از دسته سمت راست کم شد و به دسته چپی اضافه شد، قابل محاسبه است.

دقت: دقت Gaussian otsu در حالتی که دو حالت یا دو mode داشته باشیم بیشتر است، منظور از mode محدوده‌ای است که هیستوگرام مقادیر زیادی دارند. در حالات دیگر دقت یکسان دارند. همچنین اگر عکس ما دارای یک background و یک foreground باشد، الگوریتم otsu درست عمل نمی‌کند، و مقدار threshold صفر می‌دهد.

Original image	Histogram	thresholding value	
		Otsu	G.Otsu
 Walkbridge		0	124
 Woman and dark hair		0	119
 Woman blonde		121	121
 Lena		116	116

همانطور که مشخص است، برای دو عکس اول که دارای دو mode هستند، الگوریتم otsu مقدار اشتباه صفر می‌دهد در حالیکه Gaussian otsu درست عمل می‌کند. برای عکسهای دیگر مثل عکسهای سوم و چهارم، هر دو روش یک عدد را خروجی می‌دهند.

ب) بله، زیرا بیشینه شدن واریانس بین کلاسی بدین معناست که اختلاف میانگین دو دسته بیشتر است و این یعنی دو دسته ما با توجه به این threshold هر کدام در یک سمت افتاده که در سمت چپ مقادیر یک دسته و در سمت راست مقادیر دسته دیگر است، و چون thresholding به درستی انجام شده پس هر دسته داده پرت ندارد و در نتیجه واریانس درون آن کلاس کمینه است.

برای این سوال که رشد ناحیه را باید پیاده‌سازی کنیم، من از یک نقطه seed شروع کردم و با الگوریتم BFS اطرافینش را که با این بذر رنگ تقریباً مشابهی دارند را در صف اضافه کرده و هر بار از این صف یکی را برداشته و برای همسایه‌های آن این عمل را تکرار میکنم.

کد تابع BFS:

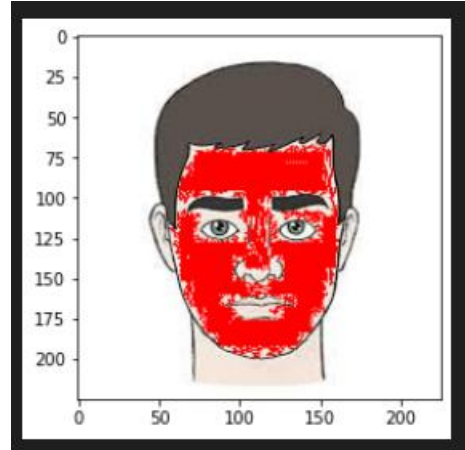
```
def BFS(original_image, image, x, y, color, thresh):
    visited = [[False] * image.shape[1] for i in range(image.shape[0])]
    queue = [(x,y)]
    visited[x][y] = True
    image[x, y] = color
    w, h = image.shape[0], image.shape[1]
    while(len(queue) > 0):
        i, j = queue.pop(0)
        if(i+1 < w and (not visited[i+1][j]) and areSameColors(original_image, x, y, i+1, j, thresh)):
            image[i+1, j] = color
            visited[i+1][j] = True
            queue.append((i+1, j))
        if(i-1 >= 0 and (not visited[i-1][j]) and areSameColors(original_image, x, y, i-1, j, thresh)):
            image[i-1, j] = color
            visited[i-1][j] = True
            queue.append((i-1, j))
        if(j+1 < h and (not visited[i][j+1]) and areSameColors(original_image, x, y, i, j+1, thresh)):
            image[i, j+1] = color
            visited[i][j+1] = True
            queue.append((i, j+1))
        if(j-1 >= 0 and (not visited[i][j-1]) and areSameColors(original_image, x, y, i, j-1, thresh)):
            image[i, j-1] = color
            visited[i][j-1] = True
            queue.append((i, j-1))
```

```
def segment(image, x, y, color, thresh):
    segmented_image = image.copy()
    BFS(image, segmented_image, x, y, color, thresh)
    return segmented_image
```

```
segmented_image_face = segment(image, image.shape[0]//2, image.shape[1]//2, [255,0,0], 10)
segmented_image_hair = segment(segmented_image_face, 50,100, [0,255,0], 10)
segmented_image_neck = segment(segmented_image_hair, 200, 85, [0,0,255], 8)
segmented_image_all = segment(segmented_image_neck, 10,10, [84,4,140], 7)
plt.imshow(segmented_image_all)
```

پارامتر اول تابع segment عکس است و پارامتر دوم و سوم، مختصات نقطه بذر هستند. پارامتر چهارم رنگی است که می‌خواهیم به ناحیه رشد داده شده بدهیم و پارامتر آخر threshold است که می‌خواهیم برای مشابه بودن دو رنگ از آن استفاده کنیم. از آنجا که خواستیم همه بخش‌های انسان و محیط اطراف را مشخص کنیم، ممکن بود به مقادیر مختلفی برای threshold نیاز پیدا کنم.

برای تابع مشابهت دو رنگ ابتدا خودم یک تابعی پیاده‌سازی کردم که از فاصله اقلیدسی دو رنگ استفاده میکرد و برای مثال برای صورت خروجی زیر را دریافت میکردم:



برای نقاط گوشه خوب عمل نمی‌کرد چون رنگ‌ها به صورت آهسته در حال تغییر بودند در حالیکه از نظر درک انسان آن رنگ‌ها نیز جزو صورت بود. برای همین در اینترنت سرچ کردم که چجوری مشابهت دو رنگ را از نظر human perception در بیارم و راهی که وجود داشت تغییر فضای رنگی rgb به lab بود که درباره این فضا چیزی نمی‌دانم.

و کدی که وجود داشت، به شکل زیر بود:

```
def similar_colors(rgb1, rgb2, threshold=50):
    # Convert RGB colors to LAB colors
    lab1 = rgb_to_lab(rgb1)
    lab2 = rgb_to_lab(rgb2)

    # Calculate the Euclidean distance between the LAB values
    delta_e = math.sqrt((lab1[0]-lab2[0])**2 + (lab1[1]-lab2[1])**2 + (lab1[2]-lab2[2])**2)

    # Return True if the Euclidean distance is below the threshold
    return delta_e <= threshold

def rgb_to_lab(rgb):
    # Convert RGB values to the LAB color space using the sRGB color space as a reference white point
    r = rgb[0] / 255.0
    g = rgb[1] / 255.0
    b = rgb[2] / 255.0

    # Apply a gamma correction to the RGB values
    r = pow((r + 0.055) / 1.055, 2.4) if r > 0.04045 else r / 12.92
    g = pow((g + 0.055) / 1.055, 2.4) if g > 0.04045 else g / 12.92
    b = pow((b + 0.055) / 1.055, 2.4) if b > 0.04045 else b / 12.92

    # Convert the RGB values to the XYZ color space
    x = r * 0.4124 + g * 0.3576 + b * 0.1805
    y = r * 0.2126 + g * 0.7152 + b * 0.0722
    z = r * 0.0193 + g * 0.1192 + b * 0.9505

    # Convert the XYZ values to the LAB color space using the D65 white point reference
    x /= 0.95047
    y /= 1.00000
    z /= 1.08883

    x = pow(x, 1/3) if x > 0.008856 else (7.787 * x) + (16/116)
    y = pow(y, 1/3) if y > 0.008856 else (7.787 * y) + (16/116)
    z = pow(z, 1/3) if z > 0.008856 else (7.787 * z) + (16/116)

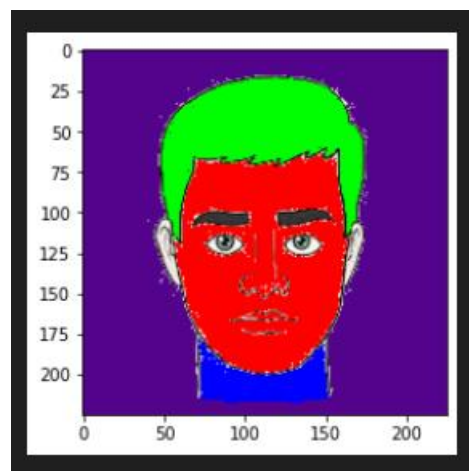
    l = (116 * y) - 16
    a = 500 * (x - y)
    b = 200 * (y - z)

    return (l, a, b)
```

تابع `rgb_to_lab` برای تبدیل یک رنگ و بردن آن به فضای `lab` است و تابع `similar_colors` فاصله اقلیدسی دو رنگ در فضای `lab` را مقایسه میکند، برعکس روش قبل که فاصله اقلیدسی در فضای `rgb` را محاسبه کرده بودیم.

```
def areSameColors(image, x1, y1, x2, y2, thresh):  
    r1,g1,b1 = image[x1, y1]  
    r2,g2,b2 = image[x2, y2]  
    return similar_colors([r1,g1,b1], (r2,g2,b2), thresh)  
    # if (r2-r1)**2 + (g2-g1)**2 + (b2-b1)**2 < 1500:  
    #     return True  
    # return False
```

این کد هم صرفاً از همان دو تابع بالا استفاده میکند. کد کامنت شده مربوط به روش قبلی بود که پیاده کرده بودم. و در نهایت خروجی آخر به شکل زیر است:



روی سیستم من ۷ ثانیه طول کشید تا خروجی نهایی ظاهر بشه. همانطور که میبینید، تابع مشابهت دو رنگ خیلی بهتر عمل کرده است.

سوال ۴ - بخشی الف

گسترش

۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۸.	۷.	۸.	۷.	۷.	۸.	۷.
۸.	۸.	۸.	۸.	۸.	۸.	۸.	۷.
۷.	۷.	۷.	۸.	۷.	۸.	۷.	۷.
۷.	۸.	۸.	۸.	۸.	۸.	۸.	۷.
۷.	۸.	۸.	۸.	۸.	۸.	۸.	۷.

$$\text{dilate}(x, y) = \max_{(x', y') \in SE} \text{src}(x+x', y+y')$$

سای

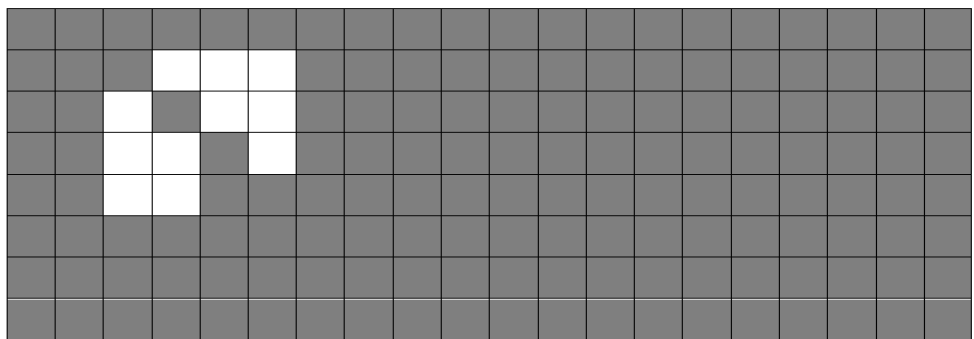
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.
۷.	۷.	۷.	۷.	۷.	۷.	۷.	۷.

$$\text{erode}(x, y) = \min_{(x', y') \in SE} \text{src}(x+x', y+y')$$

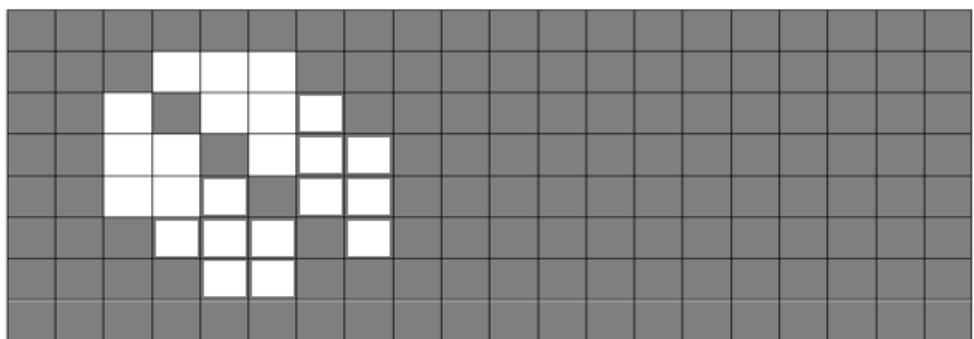
(ب)

ابتدا عملگر باز را میزنیم:

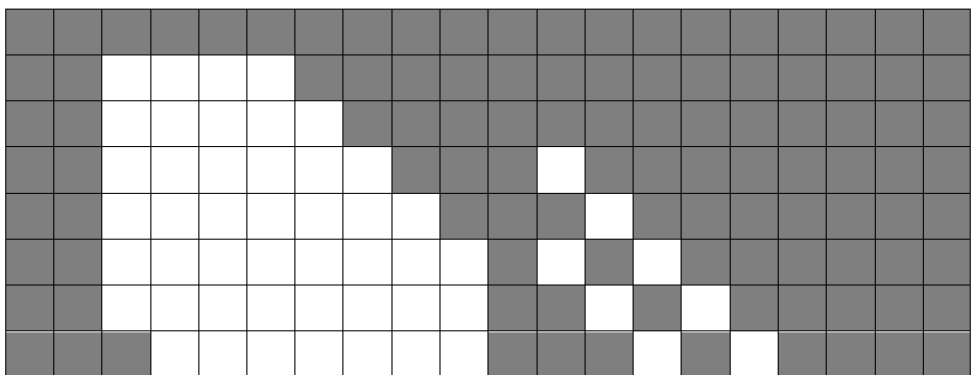
عملگر باز: ابتدا سایش و سپس گسترش را اعمال می‌کنیم. حاصل عملیات سایش:



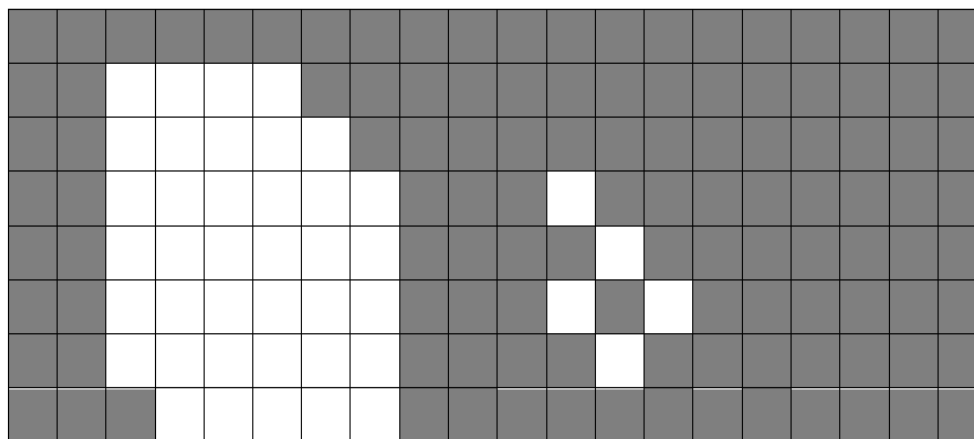
حال روی حاصل عملیات سایش، عملیات گسترش میزنیم که در نتیجه حاصل عملیات باز به شکل زیر بدست می آید:



عملگر بسته: ابتدا عملگر گسترش و سپس عملگر سایش را اعمال می کنیم. حاصل عملگر گسترش:



اکنون روی تصویر حاصل از گسترش، عملیات سایش را اعمال می کنیم.

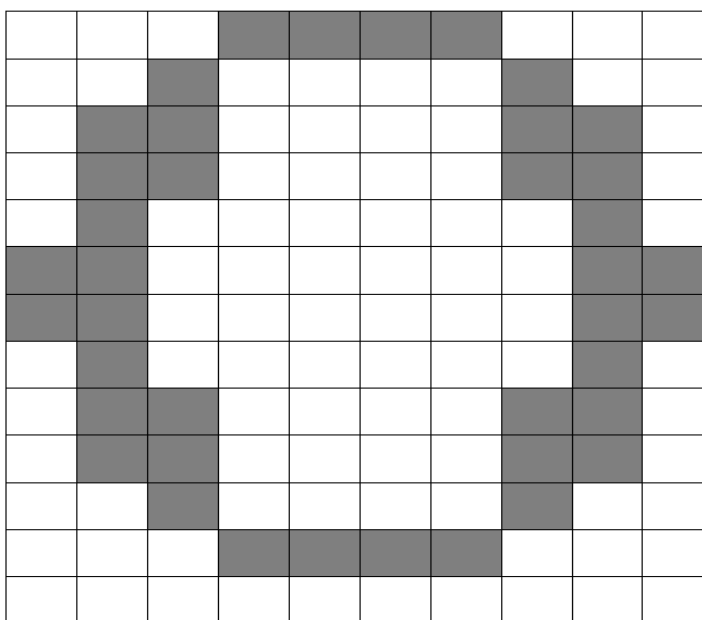


برای این قسمت مجبور بودم از padding استفاده کنم و نوع padding رو reflect انتخاب کردم.

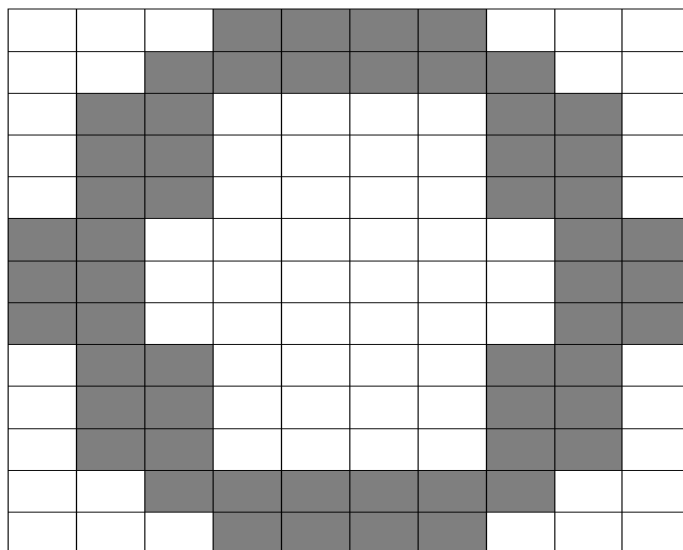
نکته: در این سوال پیکسل های سفید را ۱ و پیکسل های سیاه را ۰ گرفتم.

۵- با کرنل پایین سمت چپ عملیات باز میزنیم. سیاه ها را یک و سفید ها را صفر میگیریم.

برای عملیات باز ابتدا عملیات سایش را با کرنل انجام داده که حاصل به صورت زیر میشود.



سپس این تصویر حاصل را با همان کرنل گسترش می دهیم. البته باید ابتدا کرنل را ۱۸۰ درجه بچرخانیم.



تصویر حاصل به گونه‌ای است که خط وسط حذف شده است. اینکه باید با کرنل پایین سمت چپ استفاده کنم و عملیات باز بزنم، تجربی بدست آمد و روش خاصی نبود. برای بقیه کرنل ها نتوانستم عملیاتی که باعث حذف خط وسط و باقی ماندن عدد صفر شود، پیدا کنم.

(ب)

برای بدست آوردن مرز از چهار کرنل استفاده کردم:

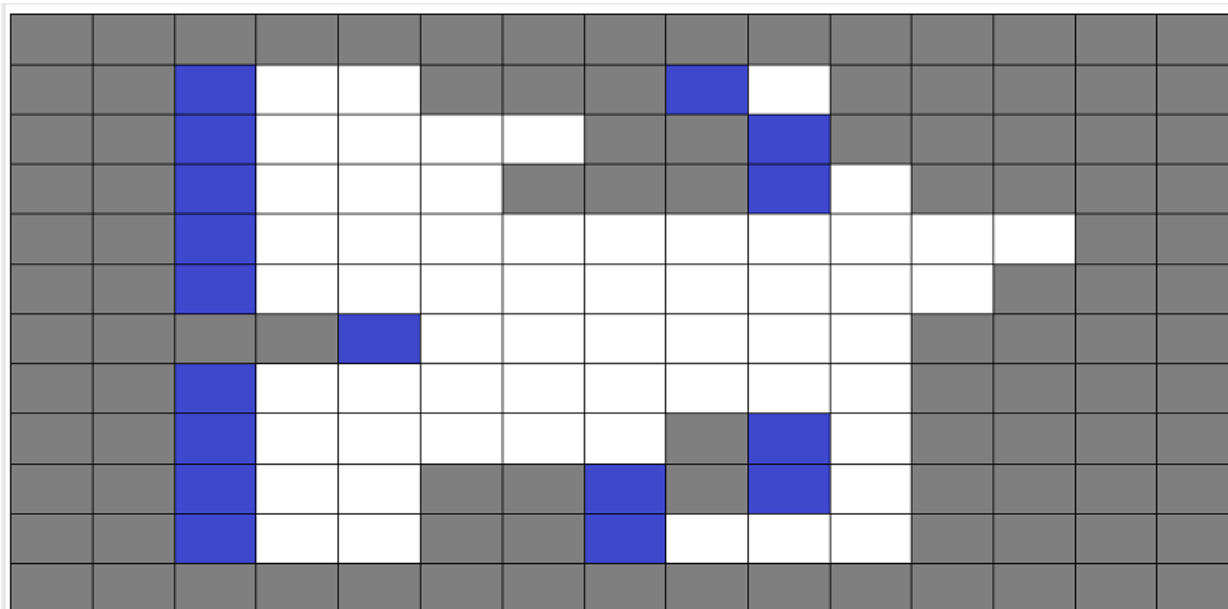
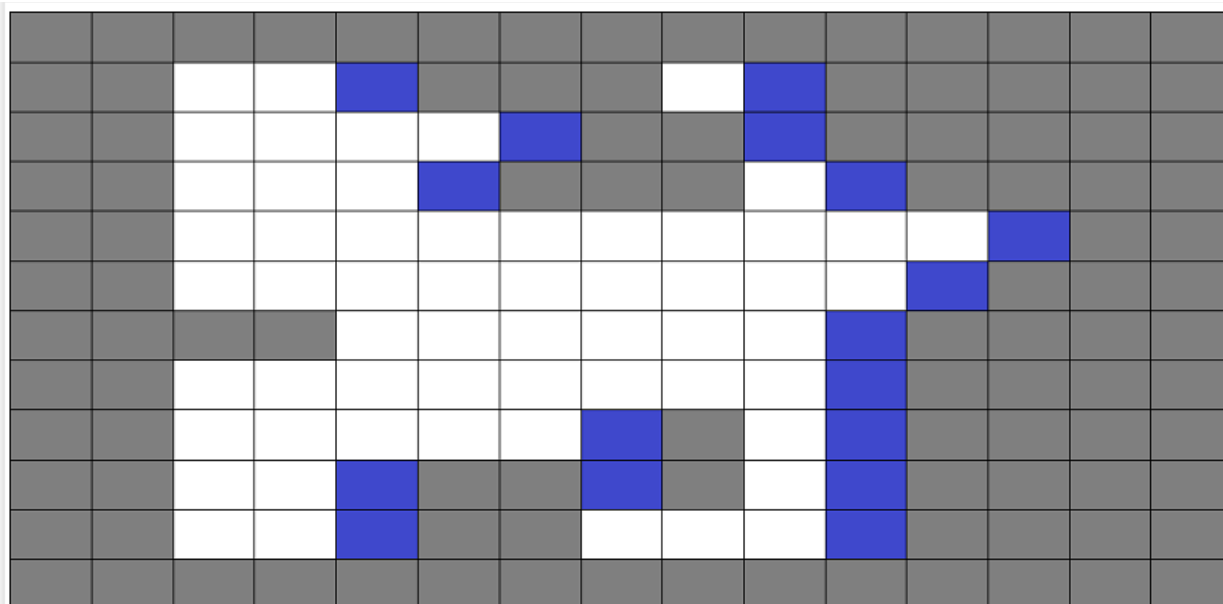
- حالتی که سمت راست یک سفید، سیاه باشد.

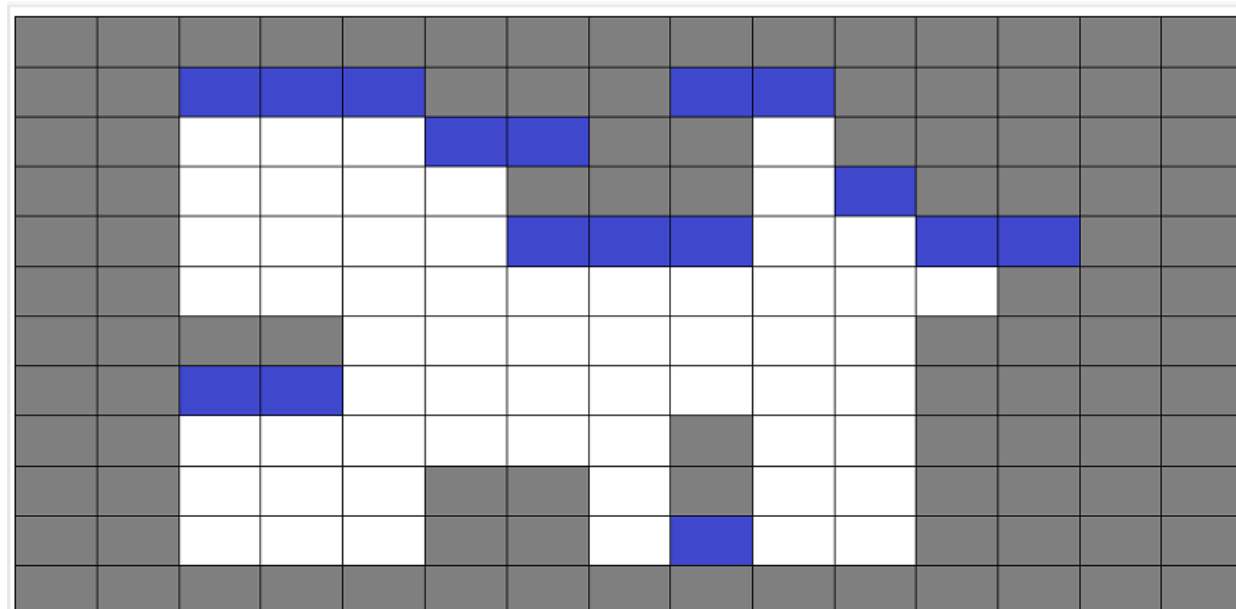
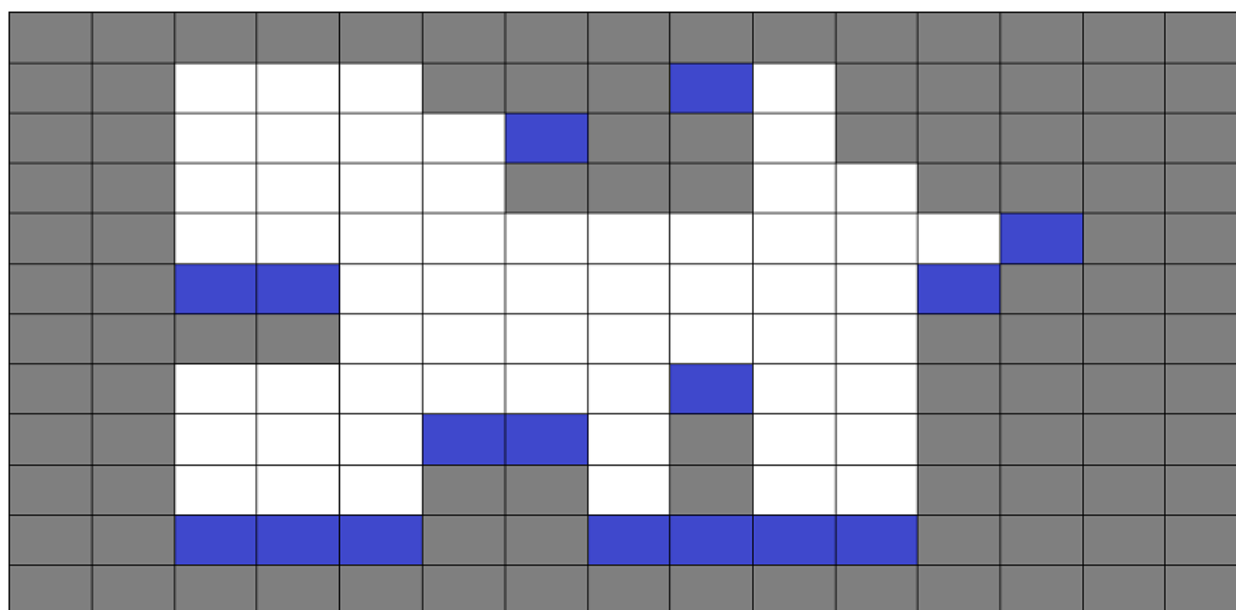
- حالتی که سمت چپ یک سفید، سیاه باشد.

- حالتی که بالای یک سفید، سیاه باشد.

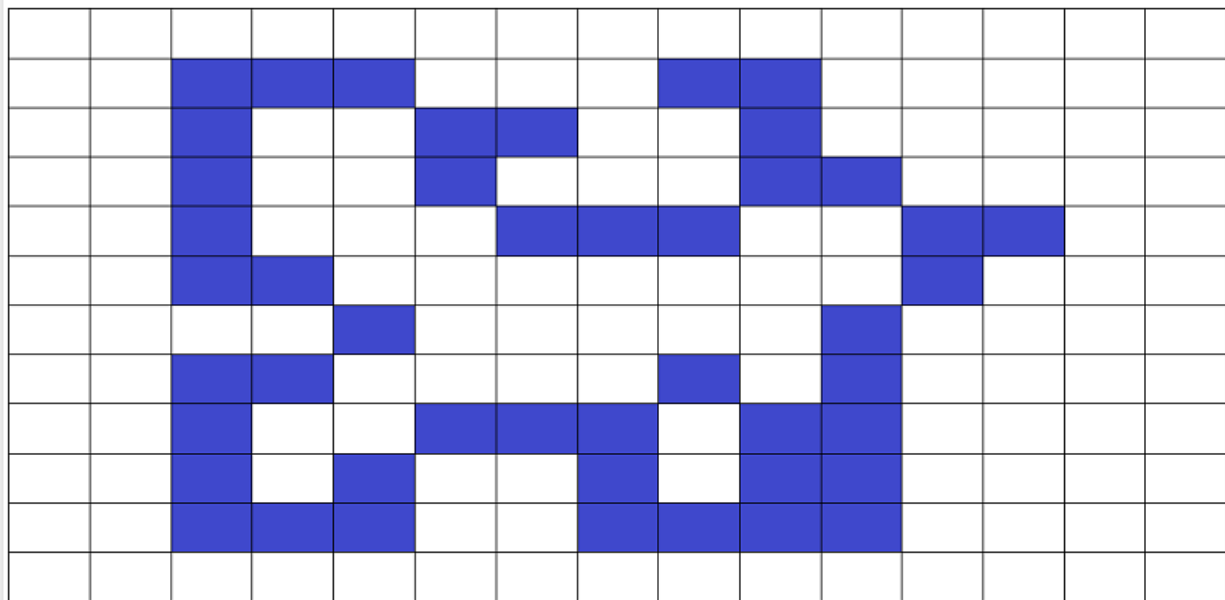
- حالتی که پایین یک سفید، سیاه باشد.

لنگر همی این کرنل‌ها روی رنگ سفید است. خانه‌هایی که در مراحل میانی به رنگ آبی هستند، نشان‌دهنده مرزهایی هستند که در آن مرحله بدست آمده‌اند، در انتها همه این خانه‌های آبی را باید اجتماع بگیریم تا جواب نهایی بدست آید. در این سوال خانه‌های سفید را ۱ و خانه‌های سیاه را ۰ گرفتم.





حال اجتماع همه نقاط آبی به شکل زیر خواهد بود:



-٤

(الف)

```
def dilate(img, kernel):
    """
    Dialates image with given kernel.

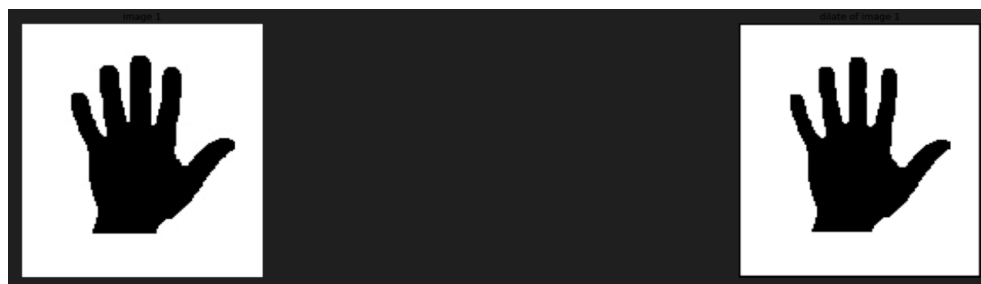
    Parameters:
        img (np.ndarray): The image to dialate.
        kernel (np.ndarray): The kernel to dialate image with.

    Returns:
        np.ndarray: The dialated image.
    """
    copy_img = img.copy()
    copy_img[copy_img < 128] = 0
    copy_img[copy_img >= 128] = 1
    kernel = np.rot90(kernel, k=2)
    img_dialated = np.zeros(img.shape)
    w, h = img.shape
    k_w, k_h = kernel.shape
    for i in range(k_w//2, w-k_w//2):
        for j in range(k_h//2, h-k_h//2):
            res = np.logical_and(copy_img[i-k_w//2:i+k_w//2+1, j-k_h//2:j+k_h//2+1], kernel)
            if np.any(res):
                img_dialated[i, j] = 1
    return img_dialated * 255
```

توضیحات تابع dilate:

در این تابع ابتدا رنگ پیکسل‌های تصویر را باینری کردم، بیش از ۱۲۸ را برابر با ۱ و کمتر را برابر با ۰ گرفتم. سپس کرنل یا عنصر ساختاری را با استفاده از تابع `np.rot90` ۱۸۰ درجه چرخاندم، پارامتر دوم این تابع نشان‌دهنده ضریبی است که در ۹۰ ضرب میشود. ابتدا همه مقادیر تصویر گسترش یافته را صفر میگیریم. حال کرنل را روی تصویر لغزانده و با استفاده از تابع `np.logical_and` کرنل را با آن قسمت از عکس که روی آن قرار گرفته، `and` میکنیم، حاصل ماتریسی است که در خانه‌هایی که کرنل مقدار ۱ داشت، مقدار خودش را میگیرد و در بقیه نقاط صفر میشود. سپس با استفاده از تابع `np.any` که بررسی میکند ماتریس ورودی حداقل یک عدد غیر صفر که اینجا ۱ است داشته باشد. اگر این اتفاق بیفتد، یعنی آن قسمت از تصویر حداقل بدونه ۱ در خانه‌هایی که کرنل ۱ بود دارد پس برای این خانه در تصویر گسترش یافته عدد ۱ یعنی سفید را قرار میدهیم. نکته‌ای که هست تصاویر ما (هواپیما و گاو و ...) سیاه هستند و بکگراند سفید است، در نتیجه عملیات `dilate` همانطور که انتظار میرود، قسمت‌های سفید را گسترش داده و در نتیجه قسمتهای سیاه کوچک میشود و در نتیجه عکس شی ما کوچکتر میشود.

خروجی dilate:



```
def erode(img, kernel):
    """
    Erodes image with given kernel.

    Parameters:
        img (np.ndarray): The image to erode.
        kernel (np.ndarray): The kernel to erode image with.

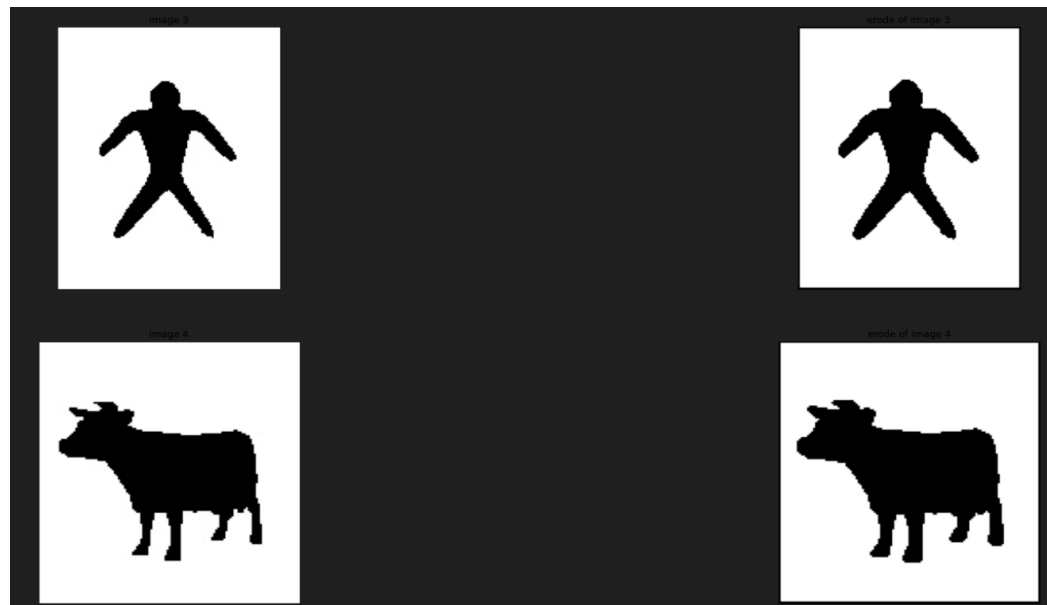
    Returns:
        np.ndarray: The eroded image.
    """
    img_eroded = np.zeros(img.shape)
    copy_img = img.copy()
    copy_img[copy_img < 128] = 0
    copy_img[copy_img >= 128] = 1
    w, h = img.shape
    k_w, k_h = kernel.shape
    for i in range(k_w//2, w-k_w//2):
        for j in range(k_h//2, h-k_h//2):
            if np.array_equal(np.logical_and(copy_img[i-k_w//2:i+k_w//2+1, j-k_h//2:j+k_h//2+1], kernel), kernel):
                img_eroded[i, j] = 1
    return img_eroded * 255
```

توضیحات تابع erode:

همه چیز همانند تابع `dilate` به غیر از اینکه کرنل را نمیچرخانیم و اینکه در قسمت بررسی شرط سفید شدن هر خانه به شکل زیر عمل میکنیم:

با استفاده از تابع `np.logical_and` کرنل را با آن قسمت از تصویر `and` کرده و با استفاده از تابع `np.array_equal` بررسی میکنیم حاصل این `and` شدن با کرنل، با خود کرنل برابر شود، زیرا در آن صورت یعنی آن قسمت از تصویر در همه خانه‌هایی که کرنل ۱ بود، مقدار ۱ دارد. در آخر تصویر را سایش یافته را در ۲۵۵ ضرب کردم که تصویر دارای مقادیر ۰ و ۲۵۵ باشد.

حاصل عملیات سایش:

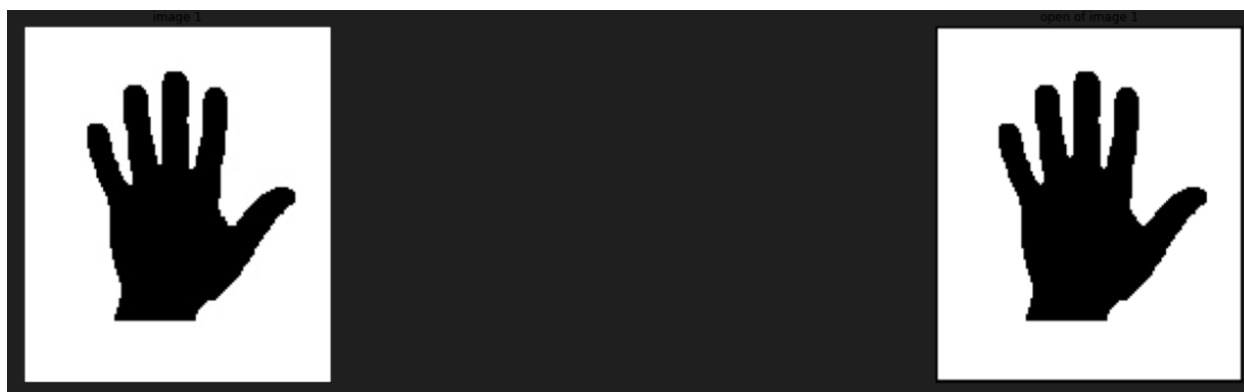


```
def open_morphology(img, kernel):  
    """  
    Performs opening morphology operation on the image.  
  
    Parameters:  
        img (numpy.ndarray): The image to perform opening morphology operation on.  
  
    Returns:  
        numpy.ndarray: The result image.  
    """  
    img_opened = dilate(erode(img, kernel), kernel)  
    return img_opened
```

توضیحات تابع open_morphology:

ابتدا سایش و روی حاصل سایش گسترش میزنیم.

خروجی عملگر باز:



تغییر خاصی مشاهده نشد چون قسمت سیاه و سفید تصویر کامل جدا هستند. اگر کرنل را خیلی بزرگ بگیریم، میتوانیم دست را کلاً حذف کنیم.

```
def close_morphology(img, kernel):
    """
    Performs closing morphology operation on the image.

    Parameters:
        img (numpy.ndarray): The image to perform closing morphology operation on.

    Returns:
        numpy.ndarray: The result image.
    """
    img_closed = erode(dilate(img, kernel), kernel)
    return img_closed
```

توضیحات تابع `close_morphology`:

ابتدا گسترش و روی تصویر حاصل سایش میزنیم.

خروجی عملگر بسته:



خروجی باقی عکسها هم در پوشه تمرین وجود دارند.

(ب)

استفاده از توابع آماده openCv:

تابع dilate:

```
image1_dilate = cv2.dilate(image1, np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]], dtype=np.uint8), iterations=1)
```

پارامتر اول عکس و پارامتر دوم کرنل است که باید به تایپ np.uint8 میبود وگرنه خطا میداد. پارامتر iterations تعداد تکرار عملیات را نشان میدهد، ما چون فقط یکبار میخواستیم گسترش را اعمال کنیم، عدد ۱ را قرار دادیم.

تابع erode:

```
image1_erode = cv2.erode(image1, np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]], dtype=np.uint8), iterations=1)
```

همه چیز مشابه dilate

تابع open:

```
image1_open = cv2.morphologyEx(image1, cv2.MORPH_OPEN, kernel, iterations=1)
```

پارامتر اول عکس، پارامتر دوم، نوع عملیات که اینجا عملگر باز است و پارامتر سوم کرنل و پارامتر چهارم همان iterations که در بالا گفتیم.

تابع close:

```
image1_close = cv2.morphologyEx(image1, cv2.MORPH_CLOSE, kernel, iterations=1)
```

همه پارامترها همانند تابع open

پارامتر دوم یعنی میخواهیم عملگر بسته را اعمال کنیم.

خروجی توابع آماده دقیقاً مشابه آنچه پیاده سازی کردم بود و فایل عکسهای حاصل در پوشه تمرین قرار دارد.

(ج)

برای بدست آوردن اسکلتون از فرمول زیر استفاده میکنیم:

$$S(A) = \bigcup_{k=0}^K S_k(A)$$

$$S_k(A) = (A \ominus kB) - (A \ominus kB) \circ B$$

$$A \ominus kB = ((A \ominus B) \ominus B) \ominus \dots)$$

$$K = \max\{k | (A \ominus kB) \neq \emptyset\}$$

$$A = \bigcup_{k=0}^K S_k(A) \oplus kB$$

```
def get_skeleton(image, kernel):
    """
    Finds the skeleton of the input image.

    Parameters:
    | image (numpy.ndarray): The input image.

    Returns:
    | numpy.ndarray: The skeleton image.
    """
    image_rev = image.copy()
    image_rev[image < 128] = 1
    image_rev[image >= 128] = 0
    res = np.zeros(image.shape)
    eroded_img = image_rev.copy()
    while(np.any(eroded_img)):
        opened_img = cv2.morphologyEx(eroded_img, cv2.MORPH_OPEN, kernel.astype(np.uint8), iterations=1)
        res = np.logical_or(res, (eroded_img - opened_img))
        eroded_img = cv2.erode(eroded_img, kernel.astype(np.uint8), iterations=1)

    return res
```

توضیحات تابع `get_skeleton`:

ابتدا مقادیر تصویر را صفر و یکی کردم و این کار را به گونه‌ای انجام دادم که نقاط سیاه، سفید و نقاط سفید، سیاه شوند. این کار را برای این کردم که شی ما قسمت سفید تصویر باشد تا کار با آن و فرمولی که بالا ذکر شد، راحت تر باشد.

سپس تصویر اسکلتون را با مقادیر اولیه صفر مقداردهی کردم و در `res` ریختم. همچنین یک متغیر `eroded_img` تعریف کردم که در هر بار `erode` کردن از آن استفاده میکنم، تا جایی این سایش پیش میرود که دیگر پیکسل سفیدی در تصویر نماند. برای شرط `while` هم همین شرط داشتن حداقل یک پیکسل روشن را بررسی کرده و به یک مرحله جلوتر میروم. روی تصویر سایش یافته با استفاده از کرنل عمل باز زده و در متغیر `opened_img` میریزم و سپس `res` را با استفاده از تابع `np.logical_or` با حاصل کم شدن تصویر سایش یافته با تصویر سایش یافته منهای تصویر حاصل عملیات باز، `or` میکنم و در `res` میریزم، این `or` کردن همان کار اجتماع را در فرمول انجام میدهد. در آخر حلقه نیز، یک سایش دیگر روی `eroded_img` میزنم تا یک مرحله به جلو برویم، هر موقع این سایش‌ها به قدری عکس را کوچک کند که چیزی از شی نماند، الگوریتم یافتن اسکلتون به پایان میرسد.

خروجی حاصل به شکل زیر بود:

image 1



skeleton of image 1



image 2



skeleton of image 2



image 3



skeleton of image 3



image 4



skeleton of image 4

