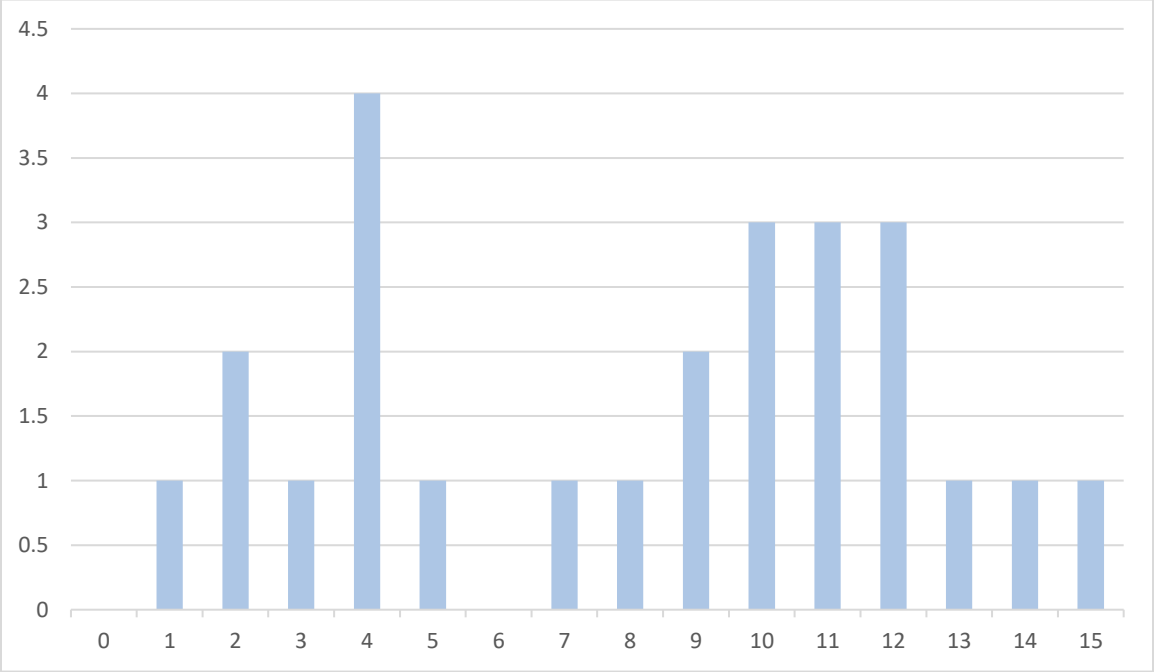
تمرین چهارم

نیکی مجیدی‌فرد 98522382

سوال اول

الف)



میانگین:

(1*1) + (2*2) + (1*3)+ (4*4)+(1*5)+(1*7)+(1*8)+(2*9)+(3*10)+(3*11)+(3*12)+(1*13)+(1*14)+(1*15) /25 = 8.12

میانه: 9

مد: 4

واریانس: 418.64/25 = 16.7456

کشش هیستوگرام :

$$g(x,y) = stretch[f(x,y)] = \left(\frac{f(x,y) - fmin}{fmax - fmin}\right)(Max - Min) + Min$$

ب) فرمول آستانه گذاری اتسو :

$$\sigma w^2 = \sigma 1^2 w1 + \sigma 2^2 w2$$

حد آستانه:9.5

$\sigma 1^2 = \quad 6.6391 \quad \sigma 2^2 = 2.3542 \qquad w1 = \frac{13}{25} \qquad w2 = \frac{12}{25}$

$\sigma w^2 = 6.6931 * \frac{13}{125} + 2.342 * \frac{12}{25} = 3.48041 + 1.12416 = 4.605 \text{ ، } \sigma w = 2.14$

حد آستانه:11.5

$\sigma 1 = \ 11.7175 \qquad \sigma 2 = 1.3333 \qquad w1 = \frac{19}{25} \qquad w2 = \frac{6}{25}$

$\sigma w^2 = 11.7174 * \frac{19}{25} + 1.333 * \frac{6}{25} = 8.903 + 0.319 = 9.22 \text{ ، ، } \sigma w = 3.03$

مقدار واریانس دومی از اولی بهتر است چرا که مقدار بیشتری دارد و واریانس بین کلاسی بیشتر است.

سوال دوم

الف)

دقت: دقت عملکرد اوتسو گاوسی بهتر و دقیق تر از اوتسو معمولی است .

اوتسو گاوسی شامل مرحله روشن سازی و کانوالو کردن فیلتر گاوسی است و هنگامی که تصویر نویز بیشتری دارد و دقت بالا در استانه گذاری اهمیت دارد، استفاده از اوتسوی گاوسی نتیجه بهتری خواهد داشت

سرعت: به طور کلی اوتسو به طور مستقیم بر تصویر اعمال می شود و به علت این که تنها یک بار محاسبه‌ی آستانه استفاده می شود از روش گاوسی سریع تر است و و همچنین انجام کانولوشن تصویر در گاوسی که اشاره شد زمان بیشتری می گیرد.

ب) فرمول کلی به دست اوردن واریانس بین کلاسی :

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

همان طور که در فرمول دیده میشود. واریانس بین کلاسی از کم کردن واریانس درون کلاسی از واریانس کلی است.

هر چقدر که واریانس بین کلاسی کم تر باشد، مقدار واریانس بین کلاسی بیشتر می شود.در نتیجه کم ترین مقدار واریانس درون کلاسی ، بیشترین مقدار واریانس بین کلاسی را حاصل می شود.

سوال سوم)

الگوریتم رشد ناحیه: مبنای اصلی این الگوریتم اختلاف پیکسل های اطراف با نوع connectivityهای مختلف است. طبق الگوریتم گفته شده در کلاس، می توانیم از دو سطح آستانه استفاده کنیم که اختلاف پیکسل های همسایه با پیکسل وسط و همچنین اختلاف با پیکسل بذر را بسنجد.

این اختلاف و مقایسه برای تمام کانال های rgb انجام می پذیرد.

پیاده سازی این الگوریتما کمک bfs (استفاده از پشته) برای تمام پیکسل های متصل به یک دیگر امکان می پذیرد.

مراحل الگوریتم)

initializing: -1

تعریف یک پشته ، یک محموعه برای تمام پیکسل هایی که دیده شده اند (visited)،و رنگ کردن نقطه ی بذر.

```
Q = Queue(maxsize = image.shape[0] * image.shape[1])
    segmented_image = image.copy()
    q.put(seed)
    visited = {seed}
    rs, gs, bs = image[seed[0],seed[1]]
    segmented_image[seed[0] , seed[1]] = color
```

2- تا زمانی که پشته خالی نباشد، یک پیکسل را از ابتدای صف برمیدارد و تمام همسایگی های 8 تایی آن را محاسبه می کند.

```
def get_pixels(image , p , x , y):
          pixels = []
          if(p[0] +1 < x):
               pixels.append((p[0] +1 , p[1]))
               if(p[1] +1 < y):
                    pixels.append((p[0] +1 , p[1]+1))
               if(p[1] -1 >= 0):
                   pixels.append((p[0] +1 , p[1]-1))
          if(p[1] +1 < y):
               pixels.append((p[0] , p[1] + 1))
               if(p[0] -1 >= 0):
                   pixels.append((p[0]-1 , p[1] + 1))
```

```
        if(p[1] -1 >= 0 ):
            pixels.append((p[0] , p[1] - 1))
            if(p[0] -1 >=0):
                pixels.append((p[0]-1 , p[1] - 1))
        if(p[0] -1 >= 0):
            pixels.append((p[0]-1 , p[1] ))
        return pixels
```

3- بعد از به دست اوردن پیکسل های همسایه، اختلاف هر یک از پیکسل ها را با پیکسل وسط ، و همچنین اختلاف هر یک را با پیکسل بذر را مقایسه می کند.

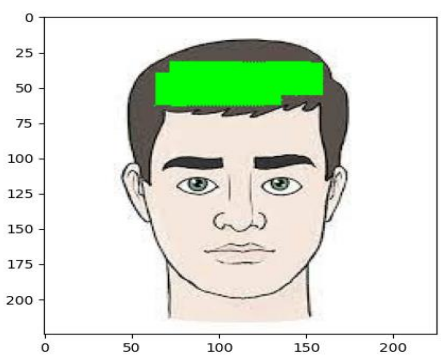
```
while not q.empty():
            p = q.get()
            pixels = get_pixels(image , p , image.shape[0] , image.shape[1]
            for n in pixels:
                if n not in visited:
                    r , g , b = image[p[0] , p[1] ]
                    ri ,gi, bi = image[n[0] , n[1]]
                    x1 ,x2 ,x3 ,x4 ,x5,x6 = diff(r, ri) , diff(g, gi), diff(b,
bi) , diff(ri,rs) , diff(gi, gs) , diff(bi , bs)
```

4- تمام مقادیر را مقایسه می کند. در صورتی که اختلاف ها کم تر سطح آستانه های تعیین شده بود، آن پیکسل به پیکسل های تحت پوشش اضافه میشود و به کیو و مجموعه visited اضافه می شود.

```
 if (x1 <= t1 and x2 <= t1 and x3 <= t1 and x4 <= t2 and x5 <= t2 and x6 <= t2):
                        segmented_image[n[0] , n[1]] = color
                        visited.add(n)
                        q.put(n)
```

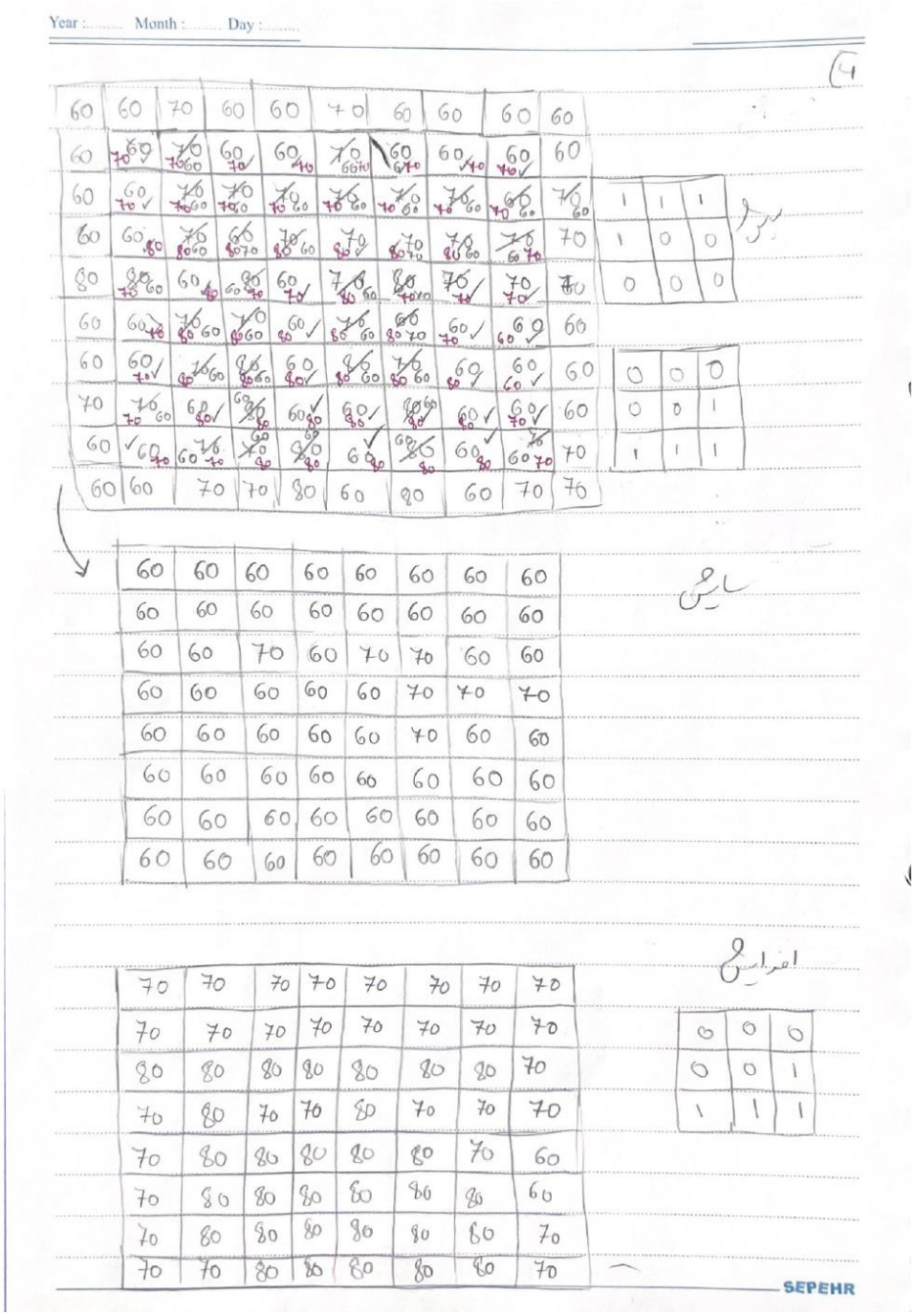
امتحان نتیجه:

```
segmented_image = segment(image , 30 , 100 , (40 , 110) , (0,255,0))
```



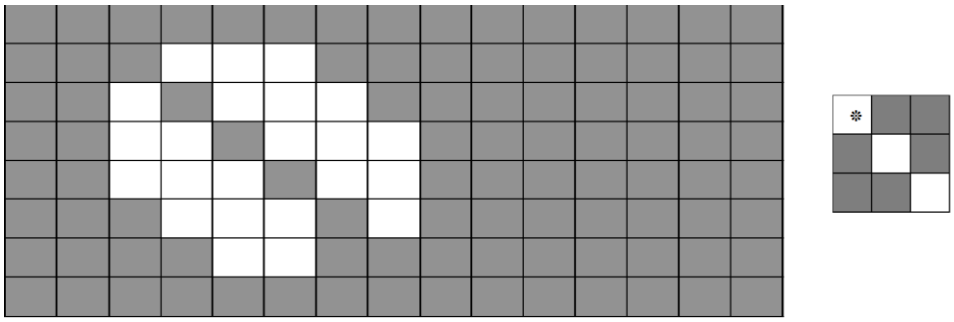
سوال چهارم

الف)

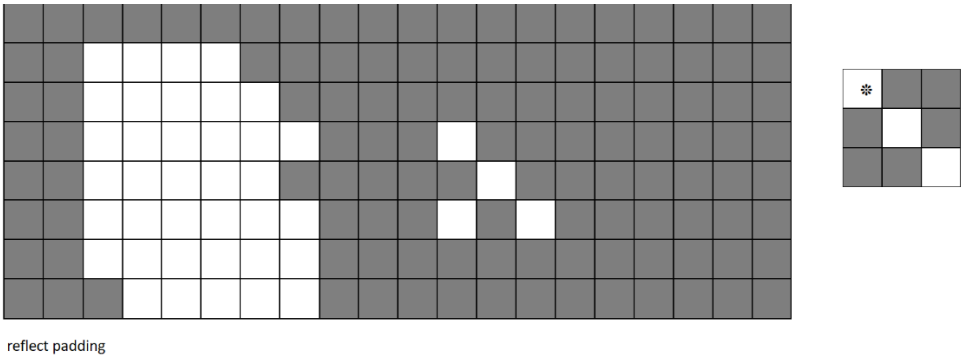


ب)

باز:



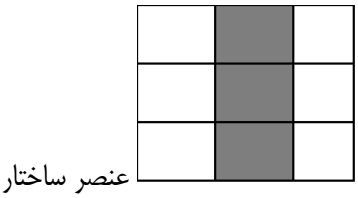
بسته:



سوال پنجم

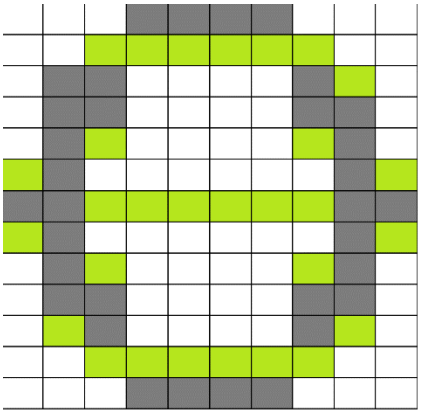
الف)

در تصویر بالا فرض می شود که پیکسل‌های سفید صفر و سیاه‌ها یک هستند. با کمک این فرض نیاز داریم به عملگری که خط سیاه وسط را از بین ببرد. عملگر باز این کار را برای ما انجام خواهد داد و خطوط اضافی را حذف می کند. ابتدا روی تصویر یک سایش انجام می دهیم(مرحله 1) و سپس افزایش را انجام می دهیم.(مرحله 2) همانطور که مشاهده شد خط وسط از بین رفت و صفر قابل مشاهده است.

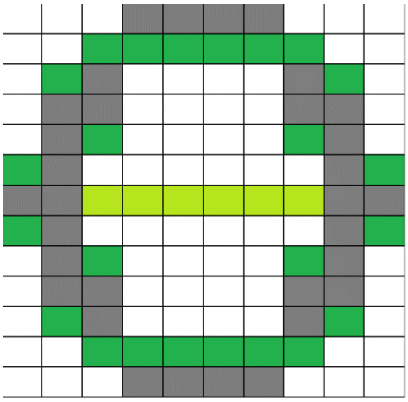


عنصر ساختار

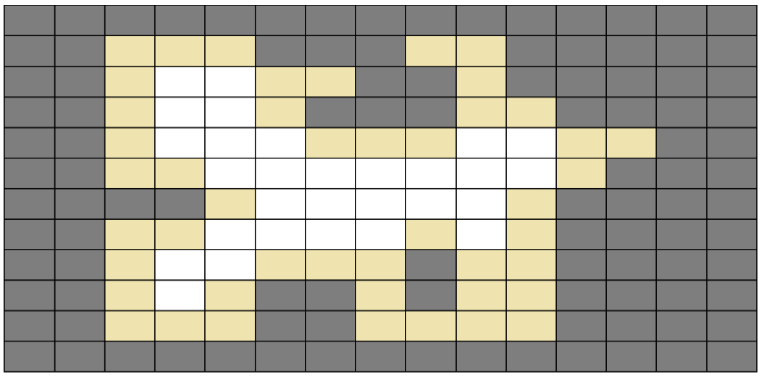
a)



b)



ب)



0	0	0
-1	1	0
0	0	0

0	0	0
0	1	0
0	-1	0

0	0	0
0	1	-1
0	0	0

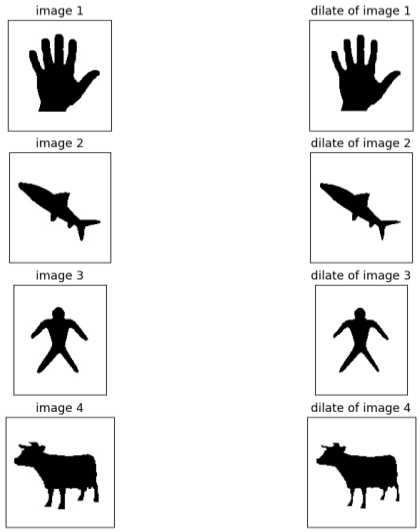
0	-1	0
0	1	0
0	0	0

سوال 6 الف) پیاده سازی عملگر های مورفولوژی

1- Dilate: این عملگر مورفولوژی را می توان با کمک الگوریتم convoloution پیاده سازی کرد. در این الگوریتم ، در صورتی که در تصویر یک پیکسل با ساختار مشترک باشد، آن پیکسل روشن خواهد شد(باینری) و برای پیکسل با سطوح روشنایی مختلف از بیشترین مقدار در نقاطی که 1 هستند استفاده می کنیم.

یک الگوریتم کانولوشن را مینویسیمم. با این تفاوت که در هر مرحله، ساختار را با پنجره ی مورد نظر ضرب کرده و در نقاط غیر صفر ، ماکسیمم را برای هر نقطه I,j از تصویر برگرداند.

به این علت که تصویر داده شده به طور کامل باینری نمی باشد.



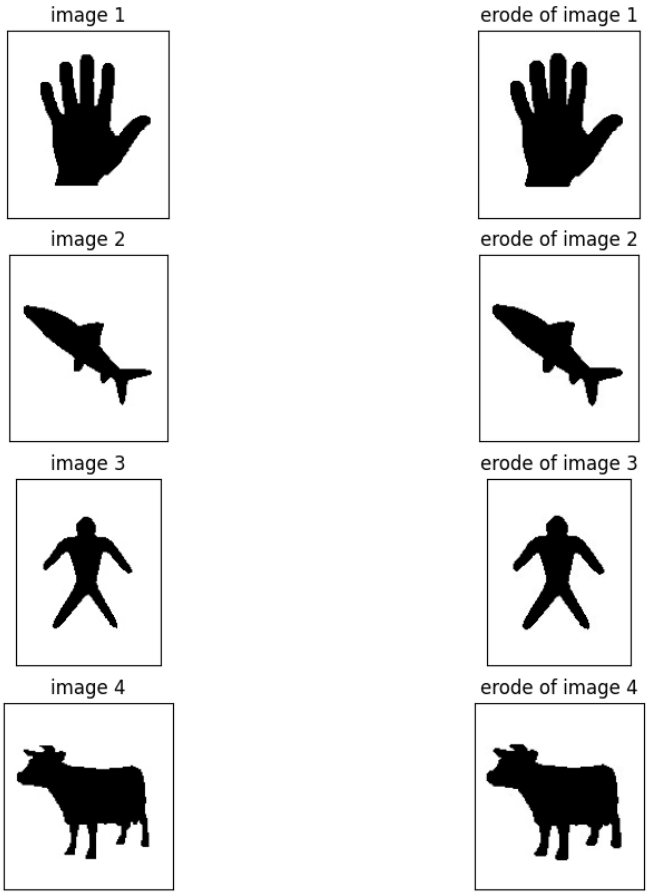
نتیجه

```
    reversed_kernel = np.flipud(np.fliplr(kernel))
    size = reversed_kernel.shape[0]
    w = image.shape[0]
    h = image.shape[1]
    ones = np.nonzero(kernel)
    #zero_padding
    matrix_padded = np.pad(image, (size//2, size//2), 'edge')
    m ,n = matrix_padded.shape
    result = np.zeros((w,h))
    for i in range( w):
        for j in range(h):
            if(i+size <= m and j+size <= n):
                window = matrix_padded[i:i+size, j:j+size]
                #return max value
                prdct = window * reversed_kernel
                max_value = np.max(prdct[ones])
                result[i,j] = max_value
    return result
```

2- سایش عملگری است که در صورتی که تمام نقاطی که در ساختارمان 1 باشند در تصویر نیز صفر نباشد، آن نقطه بدون تغییر می ماند اما در غیر این صورت، صفر می شود.

برای پیاده سازی این الگوریتم از کانولوشن استفاده میکنیم به این تفاوت که مینیمم مقدار در تمام نقاط غیر صفر ساختار را بر می گردانیم.

```
size = kernel.shape[0]
    w = image.shape[0]
    h = image.shape[1]
    ones = np.nonzero(kernel)
    matrix_padded = np.pad(image, (size//2, size//2), 'edge')
    m ,n = matrix_padded.shape
    result = np.zeros((w,h))
    for i in range( w):
        for j in range(h):
            if(i+size <= m and j+size <= n):
                window = matrix_padded[i:i+size, j:j+size]
                #return min value
                prdct = window * kernel
                min_value = np.min(prdct[ones])
                result[i,j] = min_value
    return result
```



نتیجه سایش

3- عملگر open

این عملگر در ابتدا سایش را اعمال می کند و سپس افرایش را. و برای از بین بردن خطوط اضافه تصویر استفاده می شود.

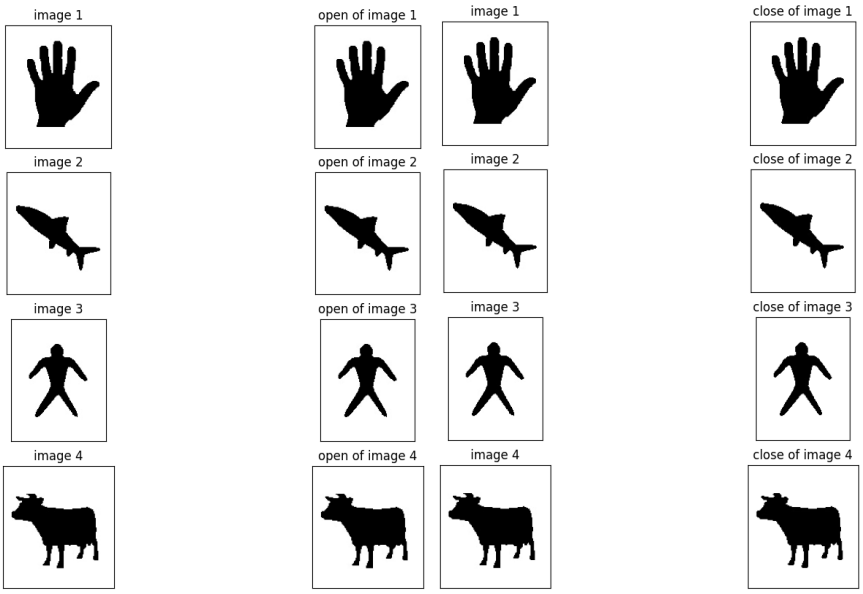
```
img_opened = dilate(erode(img, kernel), kernel)
```

4- عملگر close

این عملگر در ابتدا تصویر را گسترش می دهد و در مرحله بعد آن را سایش می دهد.

فایده استفاده از آن پر کردن تکه های توخالی و نزدیک به هم است.

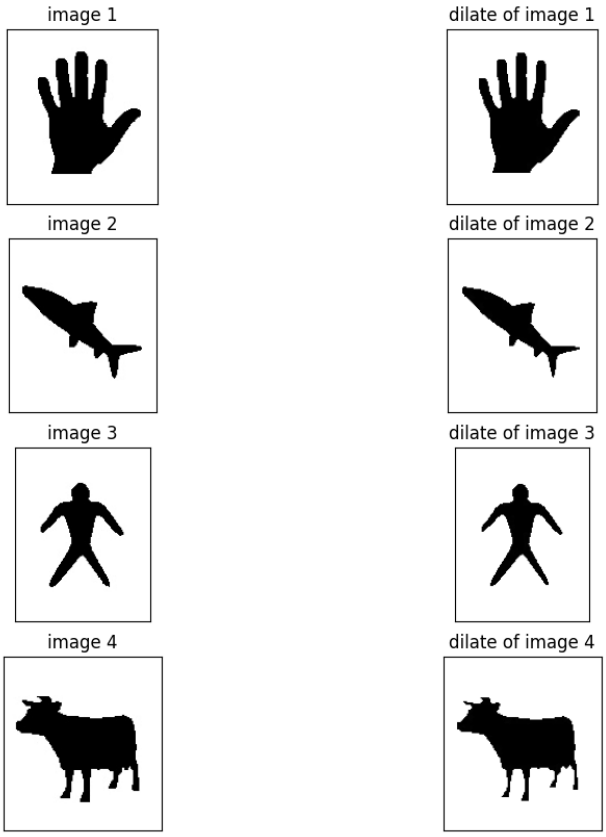
```
img_closed = erode(dilate(img, kernel) , kernel)
```



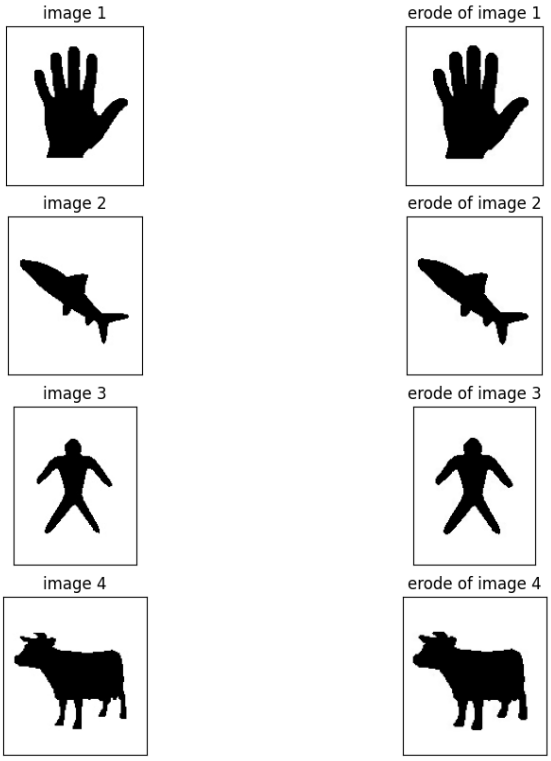
نتیجه عملگر های باز و بسته

ب) پیاده سازی با کمک توابع opencv

```
def dilate_2(image , kernel):
    kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
    return cv2.dilate(image, kernel)
```



```python
def erode_2(image , kernel):
    kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
    return cv2.erode(image, kernel)
```



متوجه میشویم که پیاده سازی ما تفاوتی با نتیجه ی opencvندارد.

**قسمت امتیازی : پیاده سازی الگوریتم اسکلت تصویر:

برای به دستن آوردن اسکلت از فرمول زیر استفاده کردیم.

1-تصویر را کامل باینری می کنیم. (با کمک cv2.threshold)

2-تا زمانی که با سایش تصویر هیچ پیکسلی باقی نمانده باشد، تصویر را سایش ، عملگر openingرا روی تصویر اعمال میکند و از هم کم می کند.

3- این مقدار به تصویر نهایی اضافه می کنیم.

```
    res = np.zeros(image.shape, dtype = 'uint8')
    #Write your code here
    params = []
    element = np.ones((3,3) , dtype = 'uint8')
    image = cv2.threshold(image, 180, 255, cv2.THRESH_BINARY_INV)[1]
    while cv2.countNonZero(image)!=0:
        #open operation
        eroded = cv2.erode(image, element)
        open_ = cv2.dilate(eroded, element)
        params.append(image - open_)
        res = res +  (image - open_)
        image = eroded
#inverse
    res = 255 - res
```

اسکلت یک ناحیه

- رابطه اسکلت ناحیه A:

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

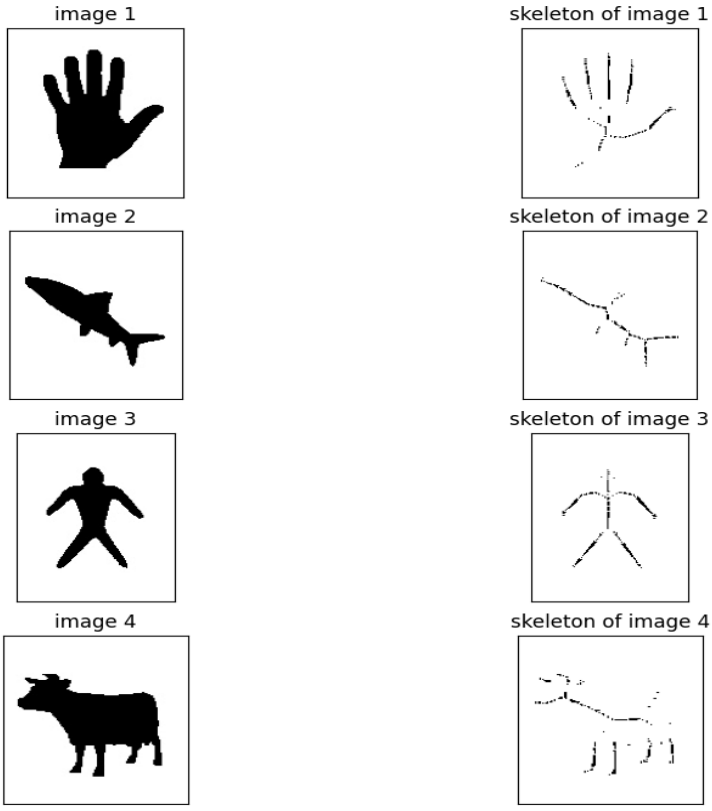
$$S_k(A) = (A \ominus kB) - (A \ominus kB) \circ B$$

$$A \ominus kB = \big((A \ominus B) \ominus B) \ominus \cdots\big)$$

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

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

نتیجه نهایی :



اسکلت تصویر