

(الف)

پارامتر های زیر را در نظر میگیریم:

w : نسبت تعداد نقاط inlier به تمام نقاط بدون inlier بودن یک نقطه است.

p : احتمال یافتن یک مجموعه از نقاط بدون outlier در k تکرار به دلیل اینکه مدل ما دایره است برای تخمین آن به سه نقطه نیاز داریم احتمال اینکه یک مجموعه کاملاً از نقاط inlier تشکیل شده باشد: w^3

اگر k تعداد تکرار باشد، احتمال اینکه هیچ مجموعه درستی انتخاب نشده باشد برابر است با:

$$1 - p = (1 - w^3)^k$$

$$k = \frac{\lg(1 - p)}{\lg(1 - w^3)}$$

در سوال w برابر با ۰.۵ و p برابر با ۰.۹۹۹۹۹ در نظر گرفته شده است

$$k = \frac{\lg(1 - 0.999999)}{\lg(1 - 0.5^3)} = \frac{\lg(0.000001)}{\lg(0.875)} \approx 103$$

برای دست یابی به این دقت، 10^3 بار تکرار الگوریتم RANSAC کافی است

(ب)

```
def __init__(self,data, k , n , delta , threshold):
    self.data = data # all points coordinates on image # [x,y]
    self.k = k # iterations allowed to run algorithm #
    self.n = n # number of sample points to estimate model parameters #
    self.delta = delta # allowed distance between candidate model and inliers #
    self.threshold = threshold # this ratio is terminating algorithm #
    self.bestModel = None
    self.maxInliers = [] #we have to save best model by inliers#
```

در این قسمت پارامتر های الگوریتم RANSAC تعریف شده اند:

k : تعداد تکرار الگوریتم

n : تعداد نقاط لازم برای ساخت یک مدل (دایره)

δ : حداقل فاصله مجاز یک نقطه تا مرز دایره، برای انتخاب شدن آن نقطه به عنوان inlier آن دایره

Threshold : حداقل نسبت inlier های یک مدل به کل نقاط دیتا برای انتخاب شدن آن مدل به عنوان مدل مطلوب و اتمام الگوریتم

Bestmodel : متغیری که در آن بهترین مدلی که تا آن لحظه پیدا شده را نگه داری میکنیم

maxInliers : نقاط inlier متناظر با بهترین مدل

```

def random_sampling(self):
    ## Start Your Code ##
    for iterate in range(self.k):
        model = self.calculateModel(random.sample(self.data, self.n))
        inliers = []
        for point in self.data:
            d = self.evaluate_model(model, point)
            if d < self.delta:
                inliers.append(point)
        if len(inliers) > len(self.maxInliers):
            self.maxInliers = inliers
            self.bestModel = model
        if len(self.maxInliers)/len(self.data) > self.threshold:
            break
    ## End ##

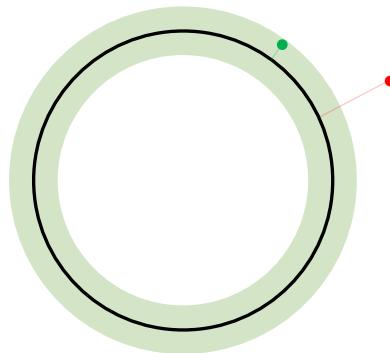
    return self.bestModel, self.maxInliers # after this function we have to draw model #

```

عملکرد الگوریتم به صورت زیر است:

به تعداد k بار مراحل زیر را انجام میدهیم:

به صورت تصادفی تعداد n نقطه از نقاط دیتا را انتخاب میکنیم و به وسیله تابع `calculateModel` (که در ادامه شرح داده خواهد شد) با آن ها یک مدل میسازیم، این مدل درواقع دایره ای است که از آن سه نقطه میگذرد. حال باید برای این مدل رای گیری کنیم تا ببینیم چه تعدادی از نقاط دیتا با این مدل موافق هستند، پس فاصله هر نقطه دیتا را تا مرز دایره مدل محاسبه میکنیم و با مقدار حد آستانه `delta` مقایسه میکنیم تا ببینیم در محدوده مشخص شده توسط حد آستانه قرار دارد یا نه، اگر در آن محدوده قرار داشت آن نقطه را به مجموعه `inlier` های مدل اضافه میکنیم.



در پایان تعداد نقاط `inlier` یافت شده برای آن مدل را با طول آرایه `maxInliers` مقایسه میکنیم و اگر بزرگتر بود آن را جایگزین `maxInliers` میکنیم و مدل متناظر را نیز به عنوان بهترین مدلی که تا به حال یافت شده در متغیر `bestModel` ذخیره میکنیم.

آرایه `bestModel` بیشترین نقاط `inlier` یافت شده برای یک مدل و متغیر `bestModel`، مدل متناظر با آن نقاط `inlier` را نگه داری میکند که درواقع مدلی است که بیشترین رای را آورده است.

در پایان اگر نسبت تعداد نقاط در `maxInliers` به تعداد کل نقاط دیتا، از پارامتر `threshold` بیشتر بود به این معناست که نسبت خوبی از نقاط به یک مدل رای داده اند و مدل مطلوب پیدا شده است پس از حلقه خارج میشویم و مدل برگزیده و نقاط آن را برミگردانیم.

```
def evaluate_model(self , model , point):
    ## Start your code ##
    model = list(model)
    dist = sqrt(np.sum(np.subtract(model[0:2],point)**2))
    d = np.abs(dist-model[2])

    ## End ##
    return d # the distance of point to center of candidate model compared to radius of model
```

تابع `evaluate_model` فاصله نقطه ورودی از مرز دایره مدل را محاسبه میکند، روش کار آن به این صورت است که فاصله نقطه از مرکز دایره کم میکند و قطر مطلق نتیجه را برمیگرداند.

```
# process image #
pointsImage = cv2.imread('points.jpg')
pointsImage_gray = cv2.cvtColor(pointsImage ,cv2.COLOR_BGR2GRAY)

ret,thresholded_image = cv2.threshold(pointsImage_gray,10,255, cv2.THRESH_BINARY)

cv2.imwrite('points.jpg' , thresholded_image)

height , width = thresholded_image.shape
data_points = []
# data_points = np.where(thresholded_image == 255)
for i in range(height):
    for j in range(width):
        if thresholded_image[i][j] == 255:
            data_points.append([i,j]) # I save row and column of pixel here as points to be optimized algorithm #
        else:
            continue

print(len(data_points))
print(thresholded_image.shape)
plt.imshow(thresholded_image , cmap='gray')
```

در این قسمت از کد، تصویر ورودی را تحلیل و نقاط آن را استخراج میکنیم به این صورت که ابتدا آن را به مقیاس خاکستری تبدیل میکند، سپس آن را به تابع `cv2.threshold` میدهد تا به صورت باینری سیاه و سفید شود، یعنی نقاطی که از مقدار حد آستانه بیشتر باشند به ۲۵۵ (سفید) تبدیل میشوند و نقاط کمتر از حد آستانه به ۰ (سیاه) تبدیل میشوند. سپس تصویر را پیکسل به پیکسل پایش میکنیم و مختصات نقاط سفید را در آرایه `data_points` ذخیره میکنیم.

```

# set K and just run this cell #
# you can change or add to this cell if needed #
if __name__ == "__main__":
    # make your ransac object here#
    ransac = RANSAC(data_points , k =50 , n = 3 , delta = 5 , threshold = 0.50)
    # find first model#
    final_model , inliers = ransac.random_sampling()
    center = [final_model[1],final_model[0]] #change order of x , y here
    print("The final model is centered at {} , radius is {} with {} inliers".format(center, final_model[2], len(inliers)))
    drawModel(final_model)
    print(len(inliers))
    data_points2 = [p for p in data_points if p not in inliers]
    ransac2 = RANSAC(data_points2 , k =50 , n = 3 , delta = 5 , threshold = 0.50)
    final_model2 , inliers = ransac2.random_sampling()
    drawModel(final_model2)
    print(len(inliers))

```

حال الگوریتم RANSAC را با این پارامتر ها اجرا میکنیم:

تعداد تکرار حلقه: ۵۰ بار

تعداد نقاط برای ساخت مدل: ۳ نقطه

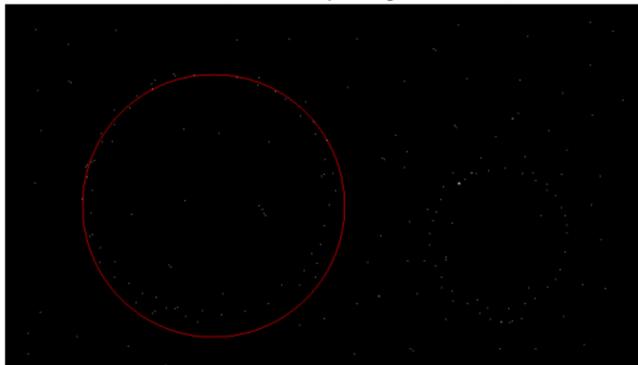
ماکریم فاصله مطلوب از مرز دایره برای انتخاب نقطه به عنوان `inlier` ۵

اگر نسبت `inlier` های یک مدل به کل نقاط داده از ۰.۵ بیشتر بود، آن مدل را به عنوان مدل برگزیده بر میگردانیم و الگوریتم را پایان میدهیم.

الگوریتم را اجرا میکنیم و اولین دایره پیدا میشود، با توجه به اینکه در تصویر دو دایره وجود دارد نقاط `inlier` دایره اول را از نقاط داده حذف میکنیم و دوباره الگوریتم را اجرا میکنیم تا دایره دوم نیز شناسایی شود.

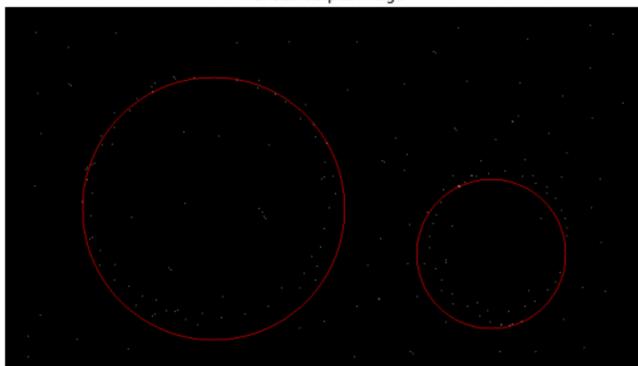
نتیجه به صورت زیر است:

The final model is centered at [239, 232] , radius is 151.21177 with 24 inliers
Ransac output image



24

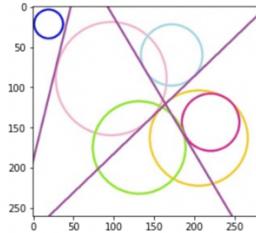
Ransac output image



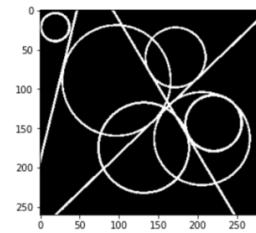
17

الگوریتم Hough میخواهیم دایره های موجود در شکل زیر را با الگوریتم هاف بیابیم. ابتدا آن را باینری میکنیم:

```
shapeImage = cv2.imread("ContourCircles.jpg")
shapeImage = cv2.cvtColor(shapeImage ,cv2.COLOR_BGR2RGB)
plt.imshow(shapeImage)
plt.show()
```



```
grayImage = cv2.cvtColor(shapeImage ,cv2.COLOR_RGB2GRAY)
ret,thresholded_image = cv2.threshold(grayImage,230,255, cv2.THRESH_BINARY_INV)
plt.imshow(thresholded_image , cmap="gray")
plt.show()
```



```
h , w , ch= shapeImage.shape
print("height and width are :" , h , w)
edgePointsX , edgePointsY = np.where(thresholded_image == 255)
print(len(edgePointsX) , edgePointsX , edgePointsY)

height and width are : 260 280
6929 [ 0 0 0 ... 259 259 259] [ 45 46 47 ... 245 246 247]
```

نقاط سفید را به عنوان لبه ذخیره میکنیم.

```
def CircleHough(x,y,radiusList):
    ## Start Your Code ##
    circleParams=[]
    circles = np.zeros((len(radiusList),int(h),int(w)))
    for p in zip(x ,y):
        edgePx , edgePy = p[0], p[1]
        for radius in range(len(radiusList)):
            for theta in range(360):
                angle = theta * np.pi / 180
                x0 = int(edgePx - radiusList[radius] * np.cos(angle))
                y0 = int(edgePy - radiusList[radius] * np.sin(angle))
                if 0<=x0<h and 0<=y0<w:
                    circles[radius,x0,y0] += 1

    for i in range(len(radiusList)):
        circleParams.append(np.append(np.unravel_index([np.argmax(circles[i,:,:])],circles[i].shape),i))
    print(circleParams)
    circleParams = np.array(circleParams).T
    # ## End ##

    return circleParams
```

در این مثال به دلیل اینکه میخواهیم دایره ها را به دست بیاوریم از الگوریتم هاف دایره ای استفاده میکنیم. ابتدا ماتریسی سه بعدی با مقادیر صفر ایجاد میکنیم، ابعاد این ماتریس را به صورت زیر در نظر میگیریم:

تعداد پیکسل های عرض تصویر * تعداد پیکسل های طول تصویر * تعداد شعاع های ممکن

این ماتریس درواقع تمام دایره های ممکن که مرکزشان میتواند یکی از پیکسل های تصویر و شعاعشان میتواند یکی از شعاع های مشخص شده باشد را نشان می دهد و برای رای گیری استفاده خواهد شد.

به ازای هر کدام از پیکسل های لبه، مرکز دایره هایی که با زوایا و شعاع های مختلف از آن پیکسل لبه میگذرند را محاسبه و به المان مربوطه در ماتریس رای گیری یک رای میدهیم.
معادله دایره به صورت زیر است:

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

پس از مراحلی به روابط زیر میرسیم که مرکز دایره ای که از نقطه (X, Y) میگذرد و شعاع r دارد را محاسبه میکند:
 θ زاویه خط واصل بین مرکز دایره و نقطه (X, Y) را نشان میدهد.

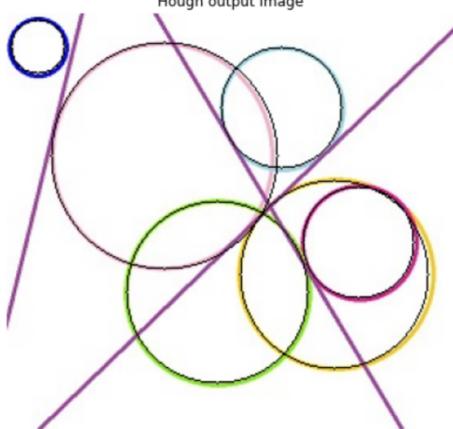
$$x_0 = x - r \cos \theta$$

$$y_0 = y - r \sin \theta$$

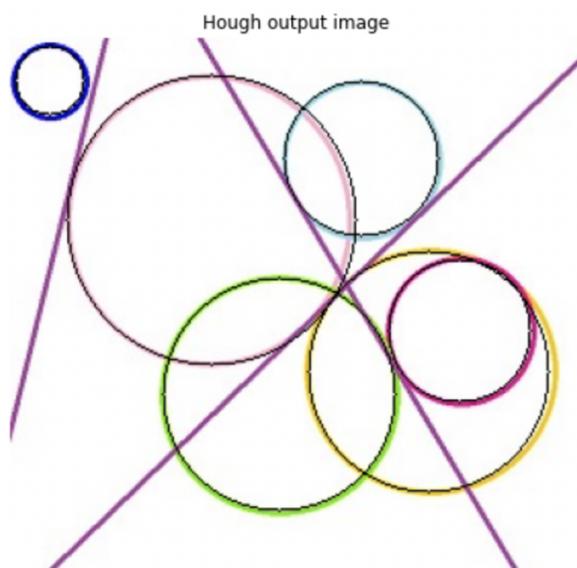
پس از پایان رای گیری، به ازای هر کدام از شعاع های ممکن، دایره ای که بیشترین رای را آورده انتخاب میکنیم و مشخصات آن را به آرایه `circleParams` اضافه میکنیم. حال به ازای هر کدام از شعاع های ممکن، دایره ای داریم که بیشترین رای را آورده است.

```
radiusList = [57, 71, 38, 17, 35, 59] # we use fixed radiiuses #
circleParams = CircleHough(edgePointsX, edgePointsY, radiusList)
drawCircle(circleParams)
showHoughResult(shapeImage)

[array([174, 131, 0]), array([89, 98, 1]), array([59, 171, 2]), array([21, 19, 3]), array([143, 219, 4]), array([163, 204, 5])]
```



همانگونه که در تصویر مشخص است، الگوریتم توانسته به خوبی دایره های تصویر را شناسایی کند.



```

def global_thresholding(image , k):

    ## your Code here ##
    ret,thresh = cv2.threshold(image,k,255, cv2.THRESH_BINARY)
    return thresh

def otsu_thresholding(image):

    ## your Code here ##
    ret2,threshold = cv2.threshold(image,0,255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)

    return threshold

def adaptive_thresholding(image , C , blocksize):
    threshold = cv2.adaptiveThreshold(image,255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,blocksize,C)

    ## your Code here ##

    return threshold

```

با این مقادیر نتایج زیر حاصل شده است:

```

k = 122
image_global = global_thresholding(img , k)
image_otsu = otsu_thresholding(img)
image_adaptive = adaptive_thresholding(img , C = 2 , blocksize = 11)

```



در روش آستانه گذاری وفقی، پیرامون هر پیکسل تصویر، پنجره‌ای در نظر می‌گیرد و میانگین مقدار پیکسل‌های موجود در آن پنجره را محاسبه می‌کند، سپس آن میانگین را با مقدار ثابت C جمع می‌کند و به عنوان حد آستانه مربوط به آن پیکسل در نظر می‌گیرد، براساس کوچکتر یا بزرگتر بودن مقدار آن پیکسل نسبت به حد آستانه متناظر با خودش، آن را در یکی از دو کلاس دسته بندی می‌کند.

برای بررسی تاثیر مقدار این پارامتر ها، میتوانیم مقادیر خیلی زیاد و یا خیلی کم آن ها را اعمال کنیم و نتایج را تحلیل کنیم، تا ببینیم به صورت حدی چه اتفاقی می افتد.

```
max_blocksize = adaptive_thresholding(img , C = 2 , blocksize = 485)
min_blocksize = adaptive_thresholding(img , C = 2 , blocksize = 3)
high_C = adaptive_thresholding(img , C = 30 , blocksize = 21)
low_C = adaptive_thresholding(img , C = -30 , blocksize = 21)
zero_C = adaptive_thresholding(img , C = 0 , blocksize = 21)
# show results here
plt.gray()
fig = plt.figure(figsize=(30,30))
cols = 5
rows = 1

fig.add_subplot(rows , cols , 1)
plt.imshow(max_blocksize)
plt.axis('off')
plt.title("max blocksize")

fig.add_subplot(rows , cols , 2)
plt.imshow(min_blocksize)
plt.axis('off')
plt.title("min blocksize")

fig.add_subplot(rows , cols , 3)
plt.imshow(high_C)
plt.axis('off')
plt.title("high C")

fig.add_subplot(rows , cols , 4)
plt.imshow(low_C)
plt.axis('off')
plt.title("low C")

fig.add_subplot(rows , cols , 5)
plt.imshow(zero_C)
plt.axis('off')
plt.title("Zero C")

plt.show()
```



هر چه *blocksize* را بزرگ تر کنیم و در حالت حدی آن را برابر کل ابعاد تصویر قرار دهیم، در واقع آستانه گذاری وفقی تبدیل به آستانه گذاری سراسری میشود زیرا کل تصویر را برای تعیین آستانه درنظر میگیرد، و همانگونه که مشاهده میشود نتیجه حاصل شده نیز مشابه آستانه گذاری سراسری است.

اگر $blocksize$ را کوچک تر کنیم و در حالت حدی آن را ۳ در نظر بگیریم باعث میشود نگاه کاملا محلی داشته باشیم و پیکسل هایی که در واقع متعلق به یک کلاس هستند و اختلاف رنگ ناچیزی دارند به دلیل نگاه محلی در دو کلاس مختلف دسته بندی شوند؛ در تصویر نتیجه هم مشاهده میشود که پیکسل هایی که در واقع متعلق به یک کلاس بوده اند به دلیل نگاه کاملا محلی در دو کلاس مختلف دسته بندی شده اند و طبقه بندی نویزی شده است.

اگر پارامتر C را خیلی بزرگ در نظر بگیریم باعث میشود بیشتر پیکسل های تصویر در کلاس اول دسته بندی شوند زیرا در این حالت کلاس اول بازه بیشتری از شدت روشانی ها را شامل میشود و اگر پارامتر C را خیلی کوچک در نظر بگیریم باعث میشود بیشتر پیکسل های تصویر در کلاس دوم دسته بندی شوند. این نتایج در تصاویر هم قابل مشاهده است.

اگر پارامتر C را نزدیک صفر و در حالت حدی خود صفر در نظر بگیریم، در این حالت الگوریتم بسیار به نویز حساس است و پیکسل هایی که مقدارشان کمی از میانگین بیشتر است در کلاس اول و پیکسل هایی که مقدارشان کمی از میانگین کمتر است در کلاس دوم دسته بندی می شوند، به همین دلیل طبقه بندی نویزی میشود و در تصویر هم مشخص است.



برای انتخاب مقادیر مناسب C و $blocksize$ ، تعدادی از مقادیر را تست کرده ایم.
به نظر میرسد مقادیر $c=5$ و $blocksize=19$ مقادیر مناسبی هستند زیرا ناحیه ها به خوبی از هم تفکیک شده اند.

(ب) آستانه گذاری سراسری:

روش عملکرد این روش آستانه گذاری ساده است و به این صورت است که مقدار پیکسل را با یک حد آستانه مقایسه میکند و بر اساس کوچکتر یا بزرگتر بودن، به دو کلاس دسته بندی میکنیم که با سفید و سیاه نشان میدهیم.

آستانه گذاری otsu:

تفاوت این روش با روش سراسری این است که حد آستانه را به صورت خودکار محاسبه میکند، روش محاسبه براساس کمینه کردن واریانس درون کلاسی است.

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

w_i درصد پیکسل های کلاس i نسبت به کل پیکسل ها و σ_i واریانس پیکسل های آن کلاس است.

حد آستانه ای که در آن σ_w کمینه شود، حد آستانه مناسب است. به ازای تمام مقادیر ممکن حد آستانه که از ۰ تا ۲۵۵ است، این واریانس را محاسبه میکند و مقدار کمینه آن ها را میابد، حد آستانه متناظر با واریانس کمینه، حد آستانه مطلوب است.

آستانه گذاری وفقی:

این روش پیرامون هر پیکسل تصویر، پنجره ای در نظر میگیرد و میانگین مقادیر پیکسل های موجود در آن پنجره را محاسبه میکند، سپس آن میانگین را با یک مقدار ثابت جمع میکند و به عنوان حد آستانه مربوط به آن پیکسل در نظر میگیرد، براساس کوچکتر یا بزرگتر بودن مقدار آن پیکسل نسبت به حد آستانه متناظر با خودش، آن را در یکی از دو کلاس دسته بندی میکند.

(ج) احتمالاً از روش رشد محلی استفاده شده که در آن مقایسه پیکسل های جدید صرفاً با پیکسل های همسایه انجام میشود، در نتیجه در مرز های ضعیف دچار نشتی میشود.

در این روش در هر مرحله از رشد، پیکسل های جدید به شرطی به ناحیه قبلی اضافه میشوند که رنگ آن ها با پیکسل های مجاور مرحله قبل اختلاف کمی داشته باشد؛ بنابراین اگر تغییر رنگ تدریجی رخ دهد، در هر مرحله از رشد، اختلاف رنگ پیکسل های جدید با پیکسل های مجاور مرحله قبلی کم است و این پیکسل ها حتی اگر اختلاف رنگ زیادی با پیکسل بذر اصلی داشته باشند، به ناحیه اضافه خواهند شد.

برای حل این مشکل میتوانیم مقایسه با پیکسل بذر را نیز سهیم کنیم تا در صورتی که پیکسل های جدید اختلاف رنگ زیادی با پیکسل بذر اصلی داشتند، به ناحیه اضافه نشوند و از نشت ناحیه جلوگیری شود.

```

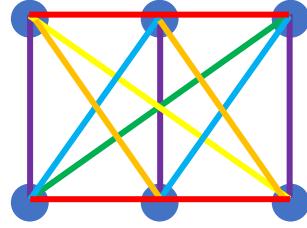
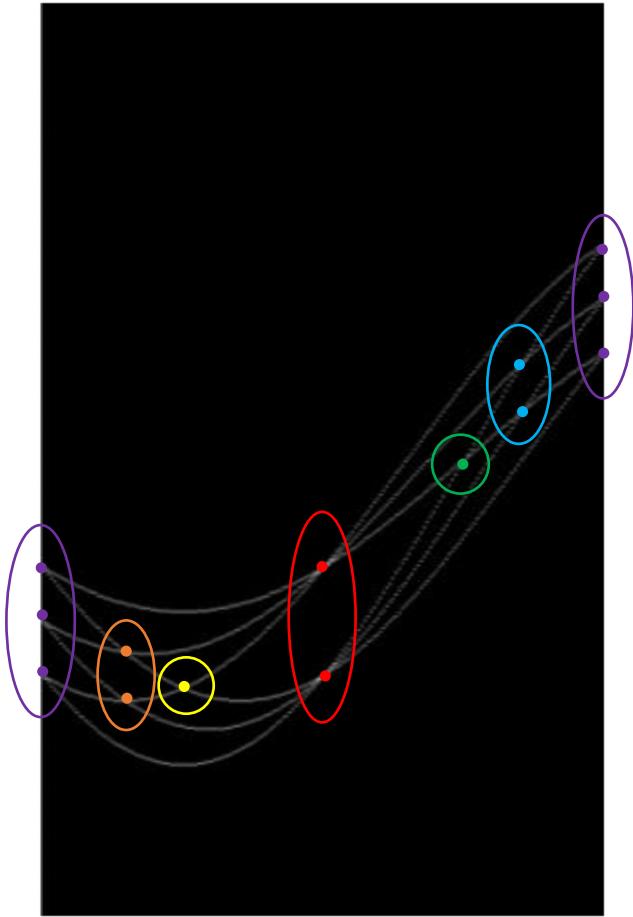
## Your code here ##
#Start#
brain = cv2.imread("brain.jpg")
# brain = cv2.cvtColor(brain ,cv2.COLOR_BGR2RGB)
gray_brain = cv2.cvtColor(brain, cv2.COLOR_RGB2GRAY)
brain_otsu = otsu_thresholding(gray_brain)
brain_adaptive = adaptive_thresholding(gray_brain , c = 5 , blocksize = 21)
kernel = np.ones((21,21))
tumor = cv2.morphologyEx(brain_otsu, cv2.MORPH_OPEN, kernel)
painted_MRI = brain.copy()
painted_MRI[tumor==255] = (255,0,0)
#End#

```



عملگر *opening* میتواند ناحیه های سفید کوچک را در تصویر حذف کند، با استفاده از همین خاصیت پس از آستانه گذاری، خطوط نازک سفید و ناحیه های کوچک سفید دیگر تصویر را حذف کرده ایم و به ناحیه تومور رسیده ایم که به دلیل اینکه ناحیه سفید بزرگی است در *opening* حذف نمیشود.

روش آستانه گذاری وفقی برای این کار مناسب نیست زیرا به دلیل نگاه محلی، تومور را هم به ناحیه های مختلف تقسیم کرده است؛ اما آستانه گذاری *otsu* به خوبی کل تومور را از سایر نقاط مغز تفکیک کرده است.



هر خط در فضای تبدیل Hough نشان دهنده مجموعه خطوطی است که از یک نقطه میگذرند و نقاط برخورد خطوط فضای تبدیل هاف، خط هایی را در تصویر نشان میدهد که از دو یا چند نقطه میگذرند.
محور افقی زاویه این خطوط و محور عمودی فاصله آن‌ها را نشان می‌دهد.

در این تبدیل ۶ خط دیده میشود که نشان دهنده ۶ نقطه در تصویر اصلی است.

نقاط قرمز: این نقاط نشان دهنده دو خط با زاویه 0° درجه است که هر کدام از سه نقطه میگذرند

نقاط سبز: این نقطه نشان دهنده خطی با زاویه نزدیک به 45° درجه است که از دو نقطه میگذرد

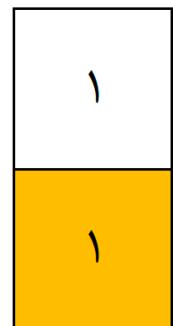
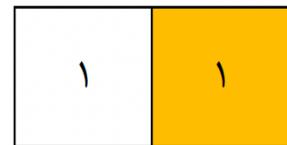
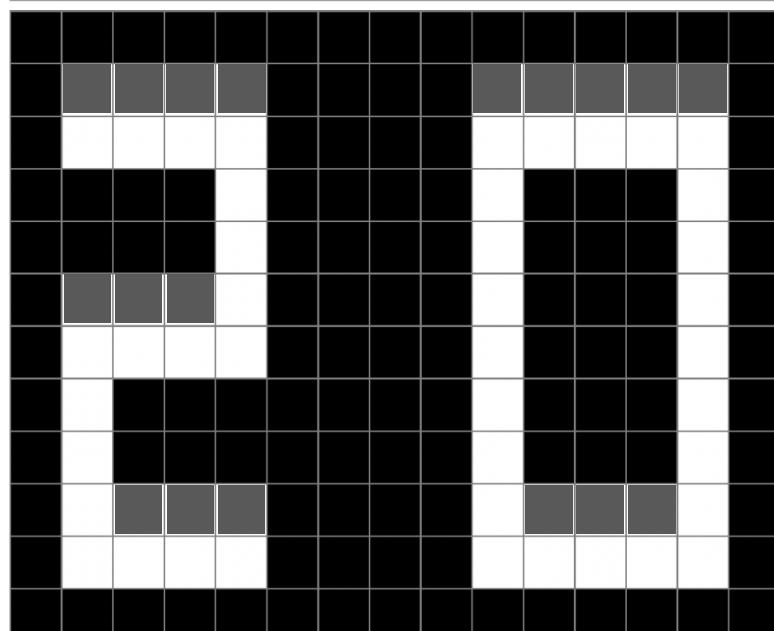
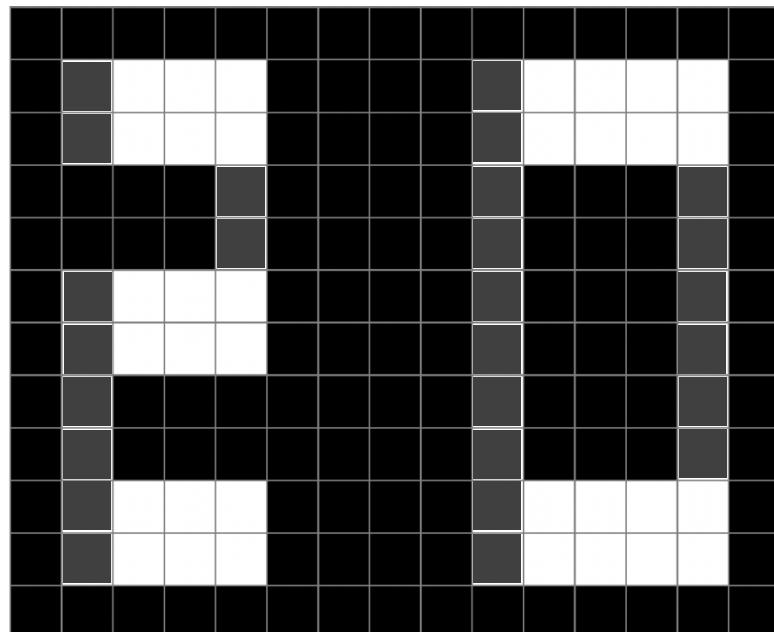
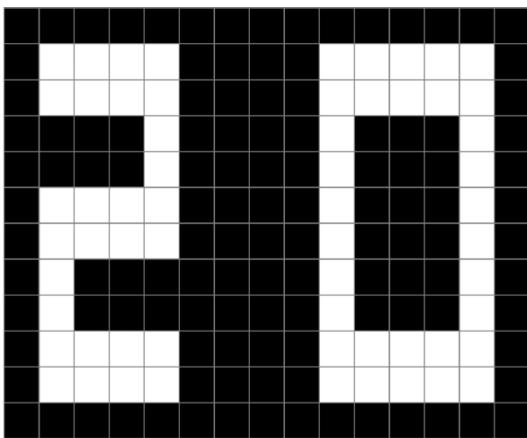
نقاط زرد: این نقطه نشان دهنده خطی با زاویه نزدیک به -45° درجه است که از دو نقطه میگذرد

نقاط آبی: این نقاط نشان دهنده دو خط با زاویه نزدیک به 60° درجه است که هر کدام از دو نقطه میگذرند

نقاط نارنجی: این نقاط نشان دهنده دو خط با زاویه نزدیک به -60° درجه است که هر کدام از دو نقطه میگذرند

نقاط بنفش: این نقاط نشان دهنده سه خط با زاویه نزدیک به 90° درجه است که هر کدام از دو نقطه میگذرند

(ب)

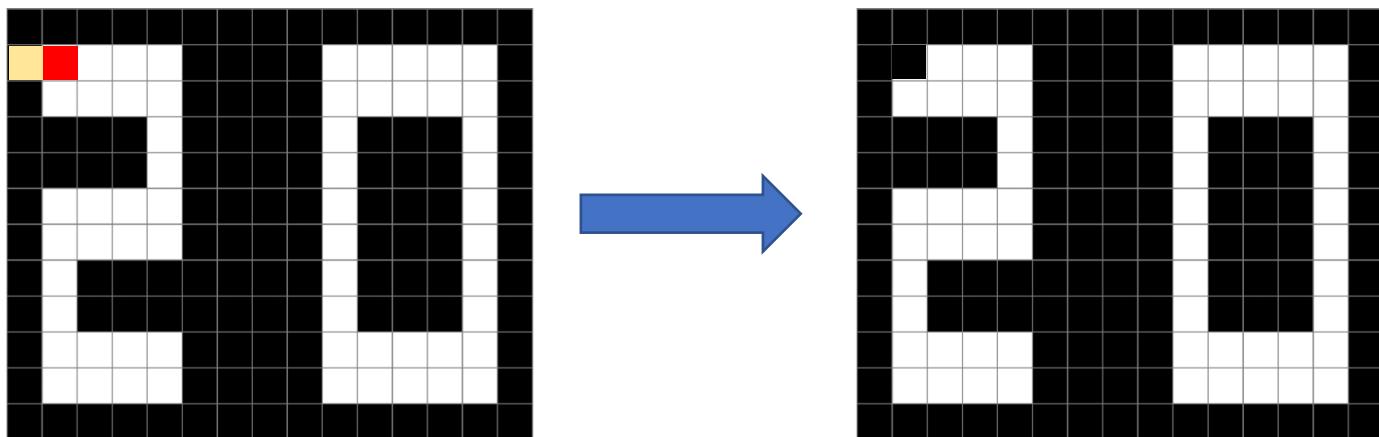


نحوه عملکرد عملگر سایش:

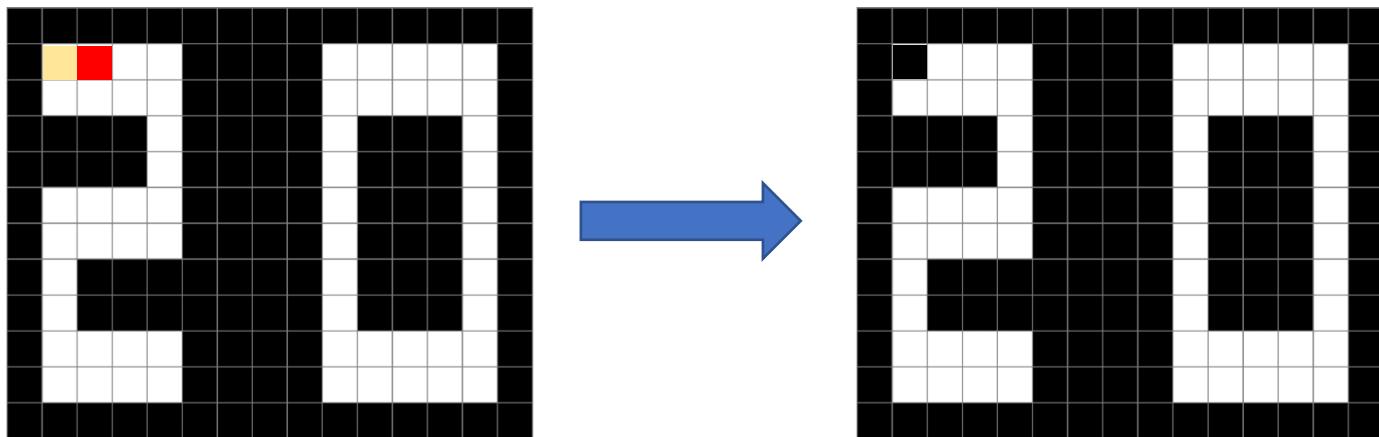
هر بار مرکز عنصر ساختاری را روی یکی از پیکسل های سفید زیر مجموعه عنصر ساختاری بودند، پیکسلی که زیر نقطه مرکزی عنصر ساختاری قرار دارد به عنوان نقطه سفید در نظر گرفته می شود و در غیر این صورت آن پیکسل سیاه در نظر گرفته میشود.

$$A \ominus B = \{z | (B)_z \subseteq A\}$$

برای مثال چند مرحله را انجام میدهیم:



در این حالت نقاط سفید زیر مجموعه عنصر ساختاری نیستند در نتیجه پیکسلی که زیر مرکز عنصر ساختاری قرار دارد سیاه میشود.



در این حالت نقاط سفید زیر مجموعه عنصر ساختاری هستند در نتیجه پیکسلی که زیر مرکز عنصر ساختاری قرار دارد سفید باقی می ماند.

```

def solidity(contour):
    """
    You should implement one of the descriptors in this method (compactness, ...)
    You can change name of this method.
    You can copy this cell and implement another descriptor in next cell.
    You can create as many descriptor as you want.
    For more information, refer to https://docs.opencv.org/4.x/d1/d32/tutorial\_py\_contour\_properties.html
    input(s):
        contour (ndarray): contour of the shape
    output(s):
        output (float): computed feature value by applying the descriptor on the contour
    """
    area = cv2.contourArea(contour)
    hull = cv2.convexHull(contour)
    hull_area = cv2.contourArea(hull)
    solidity = float(area)/hull_area
    return solidity

def compactness(contour):
    area = cv2.contourArea(contour)
    perimeter = cv2.arcLength(contour, True)
    compactness = np.pi*4*float(area)/(perimeter**2)
    return compactness

```

دو ویژگی *solidity* و *compactness* را برای کانتور ها محاسبه میکنیم.

میزان چگال بودن شکل را نشان میدهد: *Solidity*

میزان فشردگی یک شکل را نشان می دهد: *Compactness*

نحوه محاسبه آن ها از روابط زیر است:

$$\text{Compactness} = \frac{4\pi \text{ Area}}{\text{Perimeter}^2}$$

$$\text{Solidity} = \frac{\text{Area}}{\text{ConvexArea}}$$

```

# In this cell, you should use contours that we calculated in the previous cell and
# your defined descriptors to extract features for each shape.
# These features should be a numpy array with the shape of (m,n), where m is the number of shapes and n is the number of features
features = np.zeros((len(contours),2))
for i,contour in enumerate(contours):
    features[i,0] = solidity(contour)
    features[i,1] = compactness(contour)

```

این دو ویژگی را برای هر شکل در یک ماتریس ذخیره میکنیم، این ماتریس ۱۲*۲ خواهد بود زیرا ۱۲ شکل و ۲ ویژگی داریم.

```

def distance_criteria(x,y):
    """
    You should implement your distance criteria here.
    This method is used for comparing features of shapes.
    input(s):
    x (ndarray): feature vector of first shape with the shape of (n,). (n is number of features)
    y (ndarray): feature vector of second shape with the shape of (n,). (n is number of features)
    output(s):
    output (float): Distance between features of two shapes
    """
    output = np.linalg.norm(x-y)
    return output

```

تابع *distance_criteria* اختلاف دو شکل را از دید ویژگی های آن ها محاسبه میکند.

```

def grouping(features, threshold):
    """
    split feature between multiple groups based on their distance :
    input(s):
    features (ndarray): a numpy array with the shape of (m,n), where m is number of shapes and n is number of features
    threshold (float): maximum distance between two feature vector
    output(s):
    output (list): group of each shape. For example, [[1,2],[3,4,5]]
    """
    groups = []
    groups.append([features[0], [0]])
    for i, feature in enumerate(features[1:]):
        new_group_flag = True
        for group in groups:
            if distance_criteria(group[0], feature) < threshold:
                group[1].append(i+1)
                new_group_flag = False
                break
        if new_group_flag:
            groups.append([feature, [i+1]])
    return [[idx for idx in group[1]] for group in groups]

```

این تابع اشکال را براساس شباهت ویژگی های آن ها گروه بندی میکند. روش کار آن به این صورت است که ابتدا با شکل اول یک گروه می سازد، سپس شکل دوم را از نظر ویژگی ها با آن مقایسه میکند، اگر شبیه بودن آن ها را در یک گروه قرار میدهد ولی اگر شبیه نبودن برای شکل دوم یک گروه ایجاد میکند، در مرحله بعد شکل سوم را با گروه های موجود مقایسه میکند، اگر به یکی از گروه ها شباهت داشت، شکل را به آن گروه اضافه میکند و در غیر این صورت برای شکل سوم نیز یک گروه جدید ایجاد میکند. به همین ترتیب تمام اشکال با هم مقایسه میشوند و بر اساس شباهت ویژگی ها گروه بندی میشوند.

```
contours, _ = cv2.findContours(img, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

این تابع اشکال مختلف را از تصویر استخراج میکند

```

def painting(groups, contours, colors):
    """
    paint shapes base on their groups.
    input(s):
    groups (list): group of each shape. The format is as same as the output of grouping method.
    contour (ndarray): contour of the shape
    colors (list): color of each group
    """
    result = I.copy()
    for group, color in zip(groups,colors):
        for idx in group:
            plt.imshow(cv2.drawContours(result,contours,idx,color,4))
    plt.show()

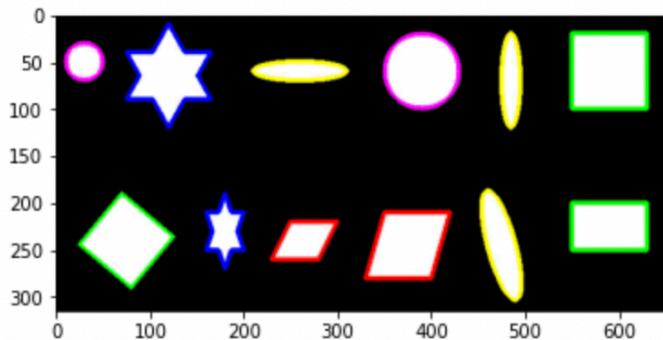
```

تابع *painting* هر گروه را با رنگ مشخص شده رنگ آمیزی میکند.

```

# You should replace your features array with None
colors = [(255,0,0),(0,255,0),(0,0,255),(255,255,0),(255,0,255)]
groups = grouping(features,0.08)
painting(groups, contours, colors)

```



در نتیجه به دست آمده اشکال مشابه به خوبی در یک گروه دسته بندی شده اند.