تمرین سری دوم – بخش عملی

سوال سوم – گزارش کار

مهرشاد فلاح اسطلخزیر – 401521462

منطق کد:

لبه‌یاب Canny:

تابع Gaussian_kernel:

```python
def gaussian_kernel(size, sigma=1):

    k = size // 2
    y, x = np.meshgrid(np.arange(-k, k+1), np.arange(-k, k+1), indexing='ij')

    kernel = np.exp(-(x ** 2 + y ** 2) / (2 * sigma ** 2)) / (2 * np.pi * sigma**2)

    return kernel / np.sum(kernel)
```

وظیفه این تابع ساخت کرنل گاوسی با هر اندازه‌ای است. برای پیاده سازی این تابع ابتدا اندازه را تقسیم صحیح بر 2 می‌گیریم سپس با استفاده از np.meshgrid دو ماتریس در جهت x و y می‌سازیم که از مقدار k- شروع می‌شوند و تا مقدار k می‌روند و به مانند تابع range در لیست عمل می‌کند. کرنل را با استفاده از فرمول گاوسی تشکیل می‌دهیم و در نهایت با تقسیم بر جمع تمام مقادیر کرنل مقادیر موجود در کرنل را نرمالیزه می‌کنیم.

تابع convolve:

```python
def convolve(image, kernel):

    h, w = image.shape
    kernel_size = kernel.shape[0]
    pad = kernel_size // 2

    padded_img = cv2.copyMakeBorder(image, pad, pad, pad, pad, cv2.BORDER_CONSTANT, value=0)
                                    (parameter) image: Any

    output = np.zeros_like(image, dtype = np.float32)
    for i in range(h):
        for j in range(w):
            region = padded_img[i:i+ kernel_size, j: j + kernel_size]
            # print(region)
            output[i, j] = np.sum(region * kernel)
            # print(output[i,j])

    return output
```

وظیفه این تابع کانوالو کردن عکس و کرنل است برای این موضوع ابتدا با درست کردن بردر با مقدار ثابت به اندازه نصف سایز کرنل (به همین مقدار به بیرون می‌رود موقع کانوالو کردن). یک حلقه تو در تو به ازای هر پیکسل می‌زنیم و پنجره دور و برش را تشکیل می‌دهیم و در نهایت مقدار خروجی را با استفاده از جمع حاصلضرب مقدار کرنل در پیکسل مقابل بدست می‌آوریم.

تابع sobel_gradients:

```python
def sobel_gradients(image):

    horizontal = np.array([-1, 0, 1, -2, 0, 2, -1, 0, 1]).reshape(3, 3)
    vertical = np.array([-1, -2, -1, 0, 0, 0, 1, 2, 1]).reshape(3, 3)

    sobel_x = convolve(image, horizontal)
    sobel_y = convolve(image, vertical)

    direction = np.arctan2(sobel_y, sobel_x)
    magnitude = np.sqrt(sobel_x ** 2 + sobel_y ** 2)

    return magnitude, direction, sobel_x, sobel_y
```

وظیفه این تابع محاسبه کانولوشن کرنل sobel و فیلترهای افقی و عمودی آن و محاسبه جهت و اندازه گرادیان در این نقاط است. که برای همین موضوع از توابع قبلی و همچنین تابع arctan2 و همچنین فرمول اندازه استفاده می‌کنیم.

تابع nms:

```python
def non_maximum_suppression(magnitude, direction):

    rows, cols = magnitude.shape
    suppressed = np.zeros((rows, cols), dtype=np.float32)
    direction = np.rad2deg(direction + 180) % 180
    for i in range(1, rows -1):
        for j in range(1, cols - 1):

            if (0 <= direction[i, j] < 22.5) or (157.5 <= direction[i, j] < 180): # 0 Are
                q = magnitude[i, j + 1]
                r = magnitude[i, j - 1]
            elif 22.5 <= direction[i, j] < 67.5: # 1 Area
                q = magnitude[i + 1, j + 1]
                r = magnitude[i - 1, j - 1]
            elif 67.5 <= direction[i, j] < 112.5: # 2 Area
                q = magnitude[i - 1, j]
                r = magnitude[i + 1, j]
            elif 112.5 <= direction[i, j] < 157.5:  # 3 Area
                q = magnitude[i - 1, j + 1]
                r = magnitude[i + 1, j - 1]

            if magnitude[i, j] >= q and magnitude[i, j] >= r:
                suppressed[i, j] = magnitude[i, j + 1]



    return suppressed.astype(np.uint8)
```

✓ 0.0s

برای پیاده‌سازی این تابع از پیاده‌سازی تمرین دو استفاده شده و تمام مراحل مانند همان است. (به همان داک مراجعه کنید).

تابع hysteresis_thresholding:

```python
def hysteresis_thresholding(image, low_threshold, high_threshold):

    strong = (image >= high_threshold) * 255
    weak = ((image >= low_threshold) & (image < high_threshold)) * 75

    rows, cols = strong.shape

    directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

    queue = deque(np.argwhere(strong == 255))
    edges = strong.copy()
    while queue:
        y, x = queue.popleft()

        for dy, dx in directions:
            ny, nx = y + dy, x + dx

            if 0 <= ny < rows and 0 <= nx < cols and weak[ny, nx] == 75:
                weak[ny, nx] = 255
                edges[ny, nx] = 255
                queue.append((ny, nx))

    return edges
```

این تابع وظیفه آستانه‌گذاری دو مرحله‌ای را دارد. نکات پیاده‌سازی آن در سوال دوم گفته شده و برای اطلاعات بیشتر می‌توانید به داک مربوط به همان گزارش مراجعه کنید.

Ransac_Hough:

تابع initialize_accumulator:

وظیفه این تابع مقداردهی اولیه به جمع‌کننده در الگوریتم تبدیل هاف است. این تابع اندازه تصویر ورودی را بررسی می‌کند و اندازه مناسب برای accumulator را بدست می‌آورد. Rho_max بیشترین فاصله از مرکز تصویر است که می‌تواند وجود داشته باشد. بعد از این میزان rhoها محاسبه شده و اندازه accumulator با (num_rhos, thetha_resolution) مشخص می‌شود.

تابع edge_direction:

این تابع گرادیان x, y را گرفته و با استفاده از atan2 جهت گرادیان را محاسبه می‌کند.

تابع hough_transform:

این تابع تبدیل هاف را انجام می‌دهد. ابتدا مقداردهی اولیه به accumulator را با کمک تابعی که بالاتر تعریف کردیم انجام می‌دهیم. و مقادیر thetas و rhos را محاسبه می‌کنیم. هرجا که لبه داشته باشیم (مقدار ماتریس لبه در آن نقطه برابر با 255 باشد)و ابتدا شرط اینکه کسینوس تفاضل جهت ما از مقدار زاویه در حال بررسی بیشتر باشد را چک می‌کنیم (مثل شبه‌کد اسلاید هشت) در صورت برقرار بودن rho را حساب می‌کنیم و در نهایت ایندکس مربوط به آن را بدست می‌آوریم و یک رای به آن اضافه می‌کنیم.

تابع find_local_maxima:

این تابع برای شناسایی ماکسیمم‌های محلی کاربرد دارد. یک حلقه تو در تو می‌زنیم و برای هر عنصر accumulator آن را با همسایه‌هایش مقایسه می‌کنیم و مقدار ماکسیمم را در آن همسایگی با پیکسل مقایسه می‌کنیم و در صورتی که این مقدار با ماکسیمم محلی برابر بود و همچنین از آستانه‌گذاری هم بالاتر بود در لیست ماکسیمم‌های محلی اضافه می‌کنیم.

تابع apply_threshold:

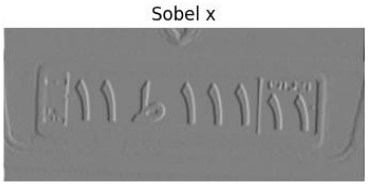
این تابع وظیفه آستانه‌گذاری و باینری کردن تصویر را دارد و هر چیزی که از آستانه‌گذاری ما بیشتر باشد را به max_val ما تبدیل می‌کند (فقط در نوع THRESH_BINARY). مقدار ret برای ما کاربردی ندارد و فقط برای THRESH_OTSU کاربرد دارد.

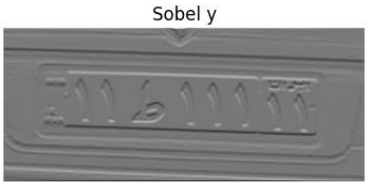
تابع detect_circles:

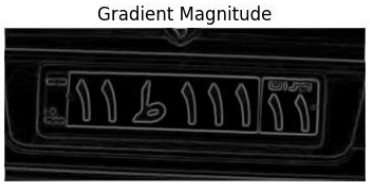
این تابع وظیفه شناسایی دایره‌ها را دارد و برای همین منظور از تابع HoughCircles که به همین منظور وجود دارد استفاده می‌کنیم. خروجی این تابع دایره‌های شناسایی شده در تصویر است.

نتایج:

لبه‌یاب Canny:

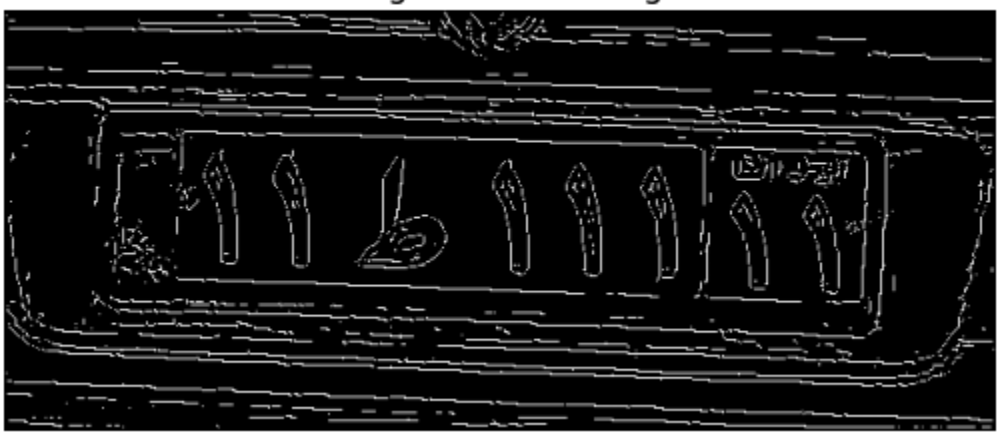






Non-Maximum Suppression

Single Thresholding

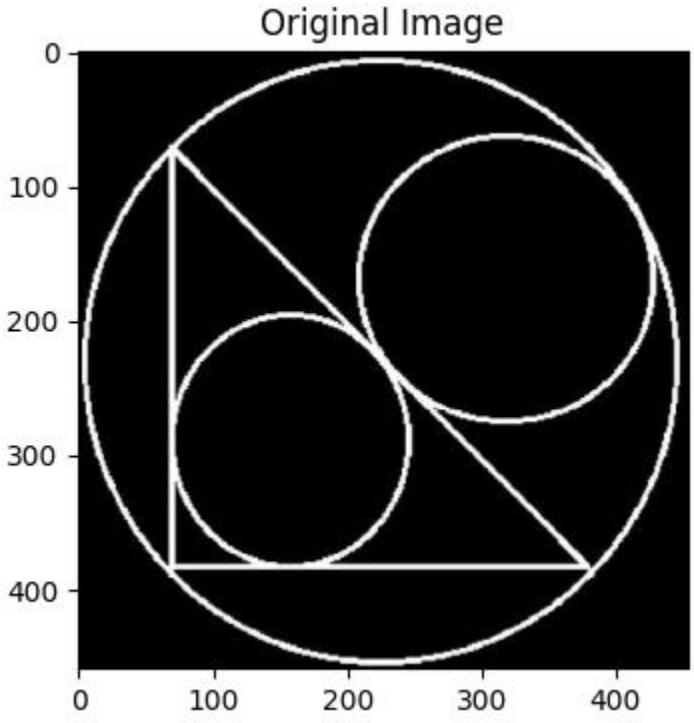


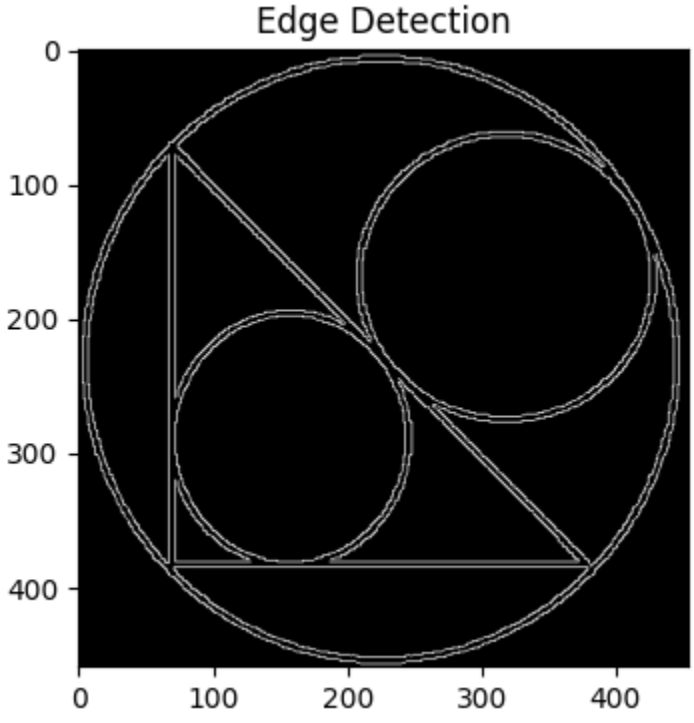
همانطور که مشاهده می‌شود اگر یک آستانه‌گذاری داشته باشیم لبه‌های خوبی شناسایی نمی‌شوند.

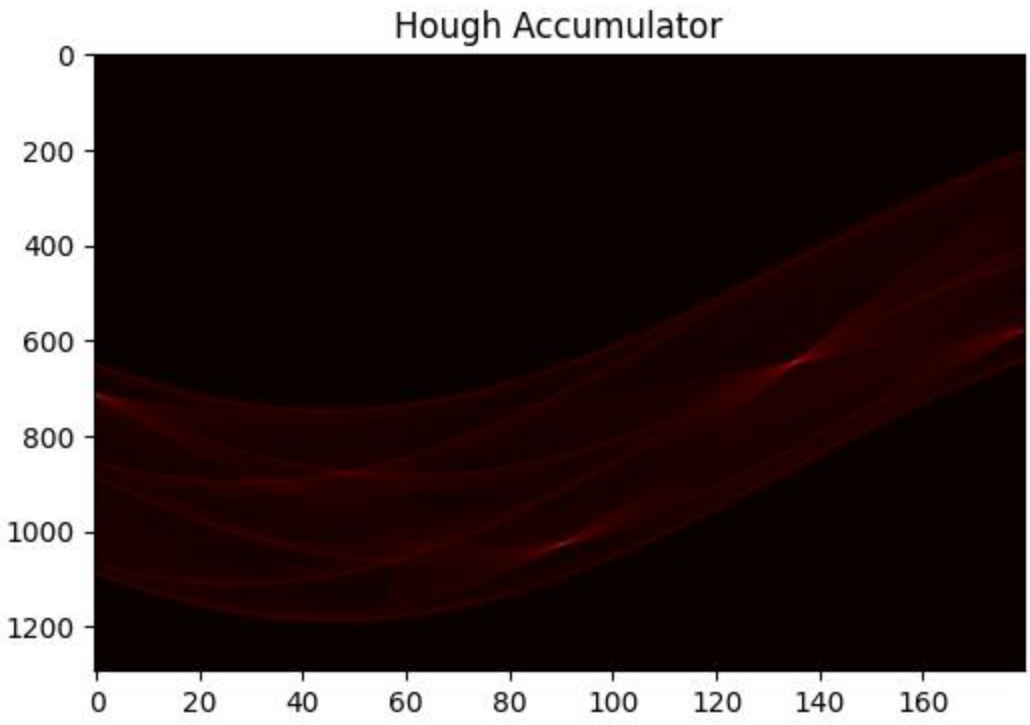
Hysteresis Thresholding (Canny Edges)

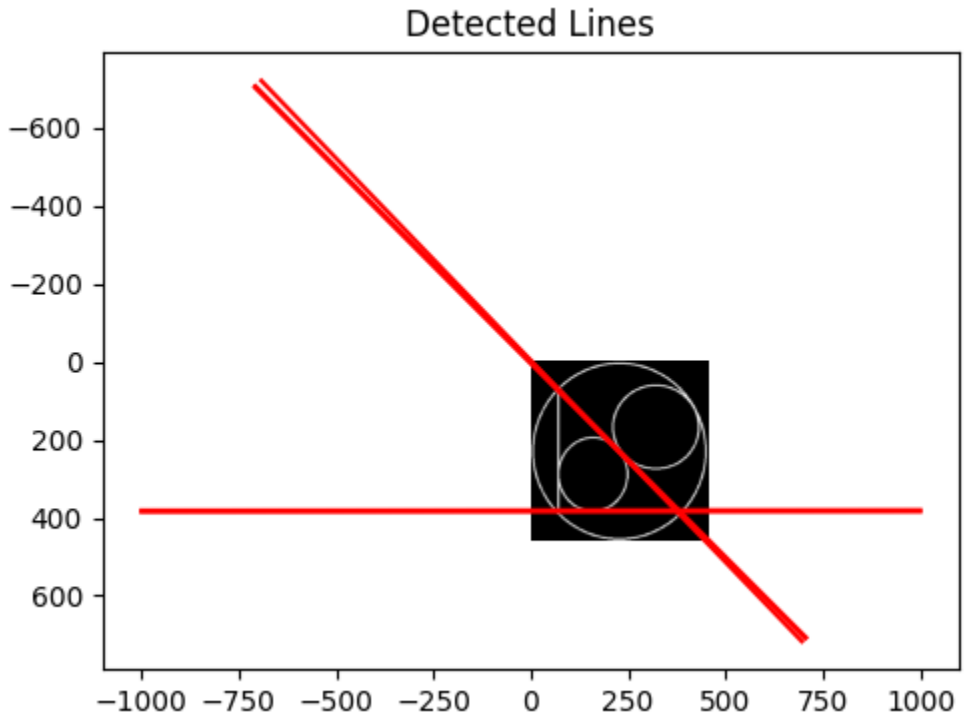
با آستانه‌گذاری دو مرحله‌ای به نتایج بهتری رسیدیم و لبه‌های خوبی شناسایی شدند.

:Hough

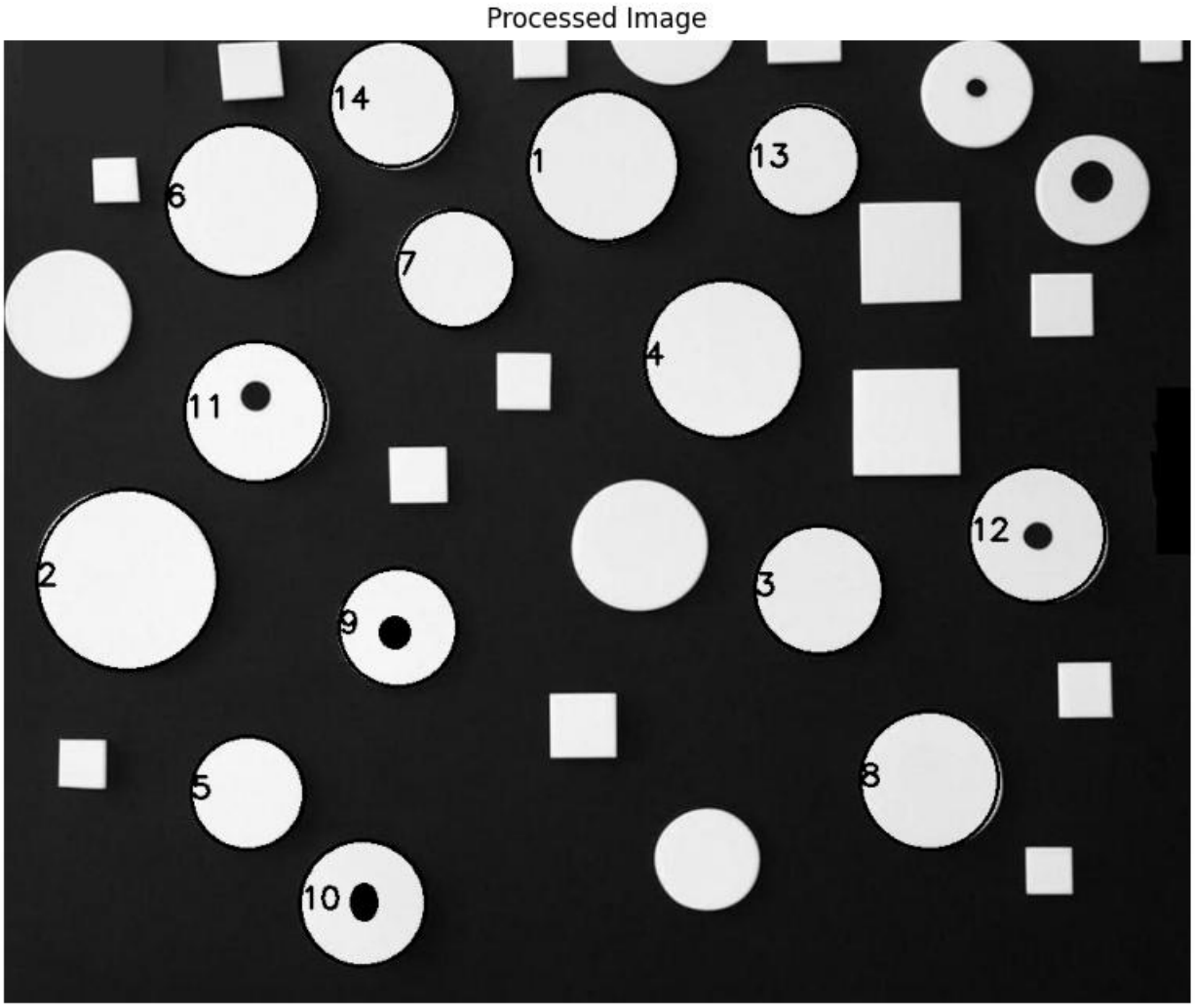

Original Image
0
100
200
300
400
0
100
200
300
400


Edge Detection
0
100
200
300
400
0
100
200
300
400





در تصویر مشاهده می‌شود که دو خط مربوط به دو ضلع از سه ضلع مثلث به خوبی یافت شدن اما ضلع سوم آن یافت نشد که فک کنم به دلیل این است که تعداد رای‌هایش مناسب نبود.



این هم نتیجه شناسایی دایره با الگوریتم Hough است که به خوبی دایره‌ها و همچنین حفره‌ها را تشخیص داده.