پاسخ تمرین ششم **مهدیه نادری: ۹۸۵۲۲۰۷۶**

سوال اول) تعداد تکرار در الگوریتم RANSAC:

منبع: اسلاید FCV_09 صفحه ی ۱۶

فرض های زیر برای حل مسئله لازم است:

۱-w = نسبت تعداد نقاط inlier (داده های غیر پرت) به کل نقاطی که داریم.

۲-p = احتمال پیدا کردن یک مجموعه از نقاط بدون outlier

۳-دو نقطه برای تخمین یک خط کافی است.

۴-k = تعداد تکرار ها ی اجرای الگوریتم

بر اساس تئوری احتمال چون هر بار اجرای الگوریتم مستقل از دفعه ی قبلی است احتمال اینکه هیچ مجموعه ی درستی در هیچ تکراری انتخاب نشود یعنی هر بار تمام نقاط انتخاب شده outlier باشند از رابطه ی زیر بدست می آید:

$$1 - p = (1 - w^2)^k$$

برای این که k را به دست بیاوریم باید از طرفین معادله ی بالا لگاریتم بگیریم چون k در توان قرار دارد:

$$log^{(1-p)} = log^{\left((1-w^2)^k\right)} \rightarrow log^{(1-p)} = k(log^{(1-w^2)})$$

$$\rightarrow k = \frac{log^{(1-p)}}{log^{(1-w^2)}}$$

طبق صورت سوال باید پارامتر ها لازم جهت حل معادله را بدست آوریم.

۱-w:

تعداد کل نقاط(نقاط نویزی هم حساب هستند):

$$60 + 80 + 120 + 100 = 360$$

تعداد نقاط inlier در ضلع وتر: 120

$$w = \frac{120}{360} = \frac{1}{3}$$

الف) p = 0.9

$$k = \frac{log^{(1-p)}}{log^{(1-w^2)}} = \frac{log(1-0.9)}{log(1-(\frac{1}{3})^2)} = \frac{log(0.1)}{log(\frac{8}{9})} \approx \frac{-1}{-0.0511} \approx 19.54937 \approx 20$$

ب) p = 0.99

$$k = \frac{log^{(1-p)}}{log^{(1-w^2)}} = \frac{log(1-0.99)}{log(1-(\frac{1}{3})^2)} = \frac{log(0.01)}{log(\frac{8}{9})} \approx \frac{-2}{-0.0511} \approx 39.09875 \approx 40$$

با ۲۰ بار تکرار الگوریتم با احتمال ۹۰ درصد وتر درست پیدا میشود و با ۴۰ بار تکرار اجرای الگوریتم با احتمال ۹۹ درصد وتر درست پیدا میشود.

سوال دوم) الف: الگوریتم Hough:

منبع: صفحات ۱۷ اتا ۲۹ اسلاید FCV_09

Hough Transform using OpenCV | LearnOpenCV

https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html

تبدیل هاف:

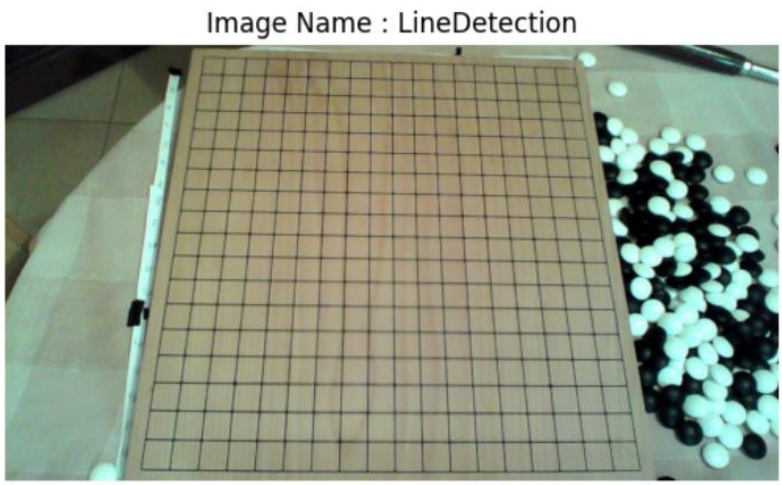
قبل از استفاده از این تبدیل باید مجموعه نقاطی که روی لبه های تصویر قرار دارند را پیدا کنیم. برای اینکار معمولا از Canny استفاده میشود. سپس در تبدیل هاف از نوع فضای (ρ,Θ) نقاطی که تعداد منحنی بیشتری از آن ها عبور میکند را پیدا میکنیم. این نقطه ها هر کدام متناظر با یک خط در فضای (x,y) هستند.

حالا مراحل خواسته شده را به ترتیب انجام میدهیم.

۱- خواندن تصویر و نمایش آن:

چون ترتیب رنگ ها در ماتریس تصویر با کتابخانه ی cv2 فرق میکنه در کانال باید از BGR به RGB تبدیل بشه.

```
#TODO
im = cv2.imread('LineDetection.jpg')
im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
plt.imshow(im)
plt.title('Image Name : LineDetection')
plt.axis('off')
plt.show()
```



۲- اعمال تبدیل Hough با تابع های آماده ی cv2:

باید از تابع cv2.HoughLines برای پیدا کردن خطوط استفاده کنیم. ابتدا کمی با این تابع از کتابخانه ی OpenCV آشنا میشویم.

آرگومان های این تابع به شرح زیر است:

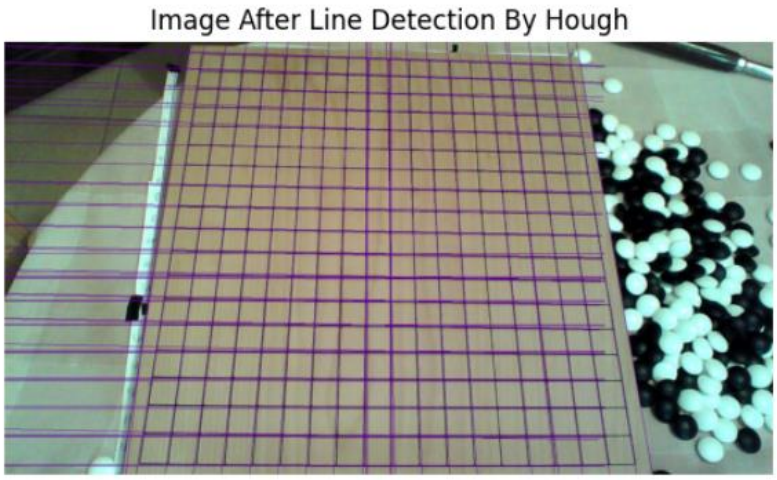
- *dst*: Output of the edge detector. It should be a grayscale image (although in fact it is a binary one)
- *lines*: A vector that will store the parameters (r,θ) of the detected lines
- *rho* : The resolution of the parameter r in pixels. We use **1** pixel.
- *theta*: The resolution of the parameter θ in radians. We use **1 degree** (CV_PI/180)
- *threshold*: The minimum number of intersections to "*detect*" a line

برای فراخوانی تابع اول فایل تصویر خوانده شده را به کانال grayscale میبریم. بعد به عملگر Canny میدهیم تا لبه ها(edges) پیدا شوند. بعد از آن خطوط را با تابع cv.HoughLines بدست می آوریم. طبق گفته ی سوال threshold را ۲۵۰ درنظر میگیریم. Dst همان خروجی canny است و rho را ۱ درنظر میگیریم تا تمام اعداد صحیح برای ρ آزمایش شوند و رزولشن آن ۱ واحد باشد. Theta را هم ۱ درجه درنظر گرفتیم اما باید یک درجه را برحسب رادیان بنویسیم.

این تابع به ما ρ,Θ های پیدا شده را برمیگرداند. برای پیدا کردن خطوط در تصویر باید خط متناظر با این نقاط را در مختصات دکارتی از روی فرمول ها پیدا کنیم و سپس با پیدا کردن خطوط آن ها را روی تصویر بکشیم و نمایش دهیم.

روی خروجی تابع for زدیم و به ازای هر کدام بعد از پیدا کردن خط متناظر با تابع ()cv2.line آن را روی تصویر رسم کردیم و نمایش دادیم.

```
#TODO
grayim = cv2.cvtColor(im, cv2.IMREAD_GRAYSCALE)

dst = cv2.Canny(grayim, 30, 300)

lines = cv2.HoughLines(dst, 1, np.pi/180, 250)


im2 = im.copy()
for i in lines:
    for ro, theta in i:
        x1 = int(ro * np.cos(theta) + 1000 * (-np.sin(theta)))
        y1 = int(ro * np.sin(theta) + 1000 * (np.cos(theta)))
        x2 = int(ro * np.cos(theta) - 1000 * (-np.sin(theta)))
        y2 = int(np.sin(theta) * ro - 1000 * (np.cos(theta)))
        cv2.line(im2, (x1,y1), (x2,y2), (100, 0, 150), 1)

plt.imshow(im2)
plt.title('Image After Line Detection By Hough')
plt.axis('off')
plt.show()
```



سوال دوم) ب: الگوریتم Probabilistic Hough Transform:

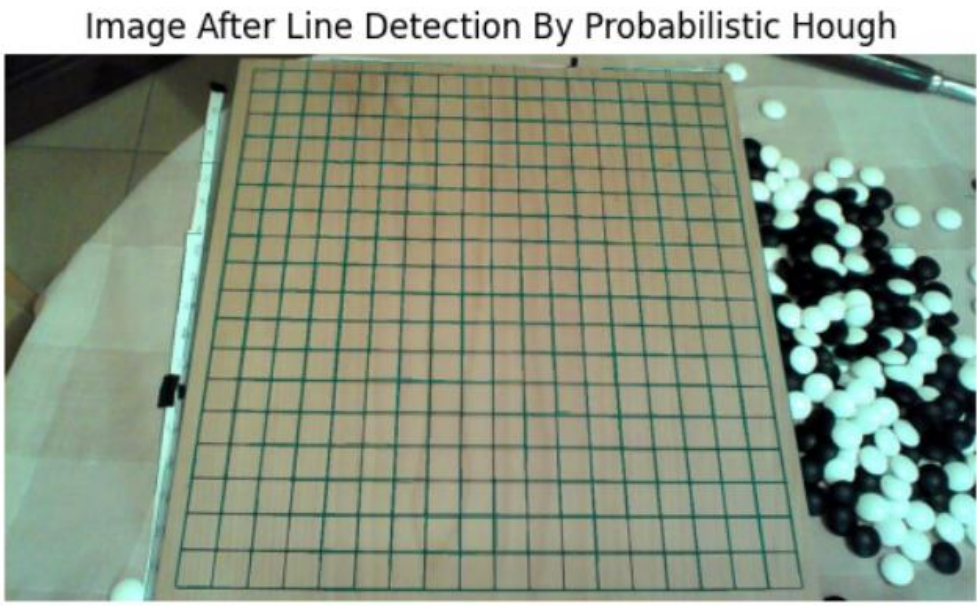
این الگوریتم طول خطوط را هم در خطوط پیدا شده دخیل میکند. باید از تابع HoughLinesP در OpenCV استفاده کنیم. تفاوتش با تابع قبلی این است که دو آرگومان اضافه تر زیر را هم دارد:

- **minLineLength** - Minimum length of line. Line segments shorter than this are rejected.
- **maxLineGap** - Maximum allowed gap between line segments to treat them as a single line.

که اولی به معنی کمترین طول ممکن برای خط است و خط هایی که طول آن ها کمتر از این مقدار باشد پذیرفته نمیشوند و دومی بیشترین فاصله ی مجاز بین خطوط است که به عنوان یک خط به هم پیوست شوند. اولی را ۱۰ و دومی را ۱۵ درنظر گرفتم. Threshold را هم از ۲۵۰ به ۱۰۰ تغییر دادم.

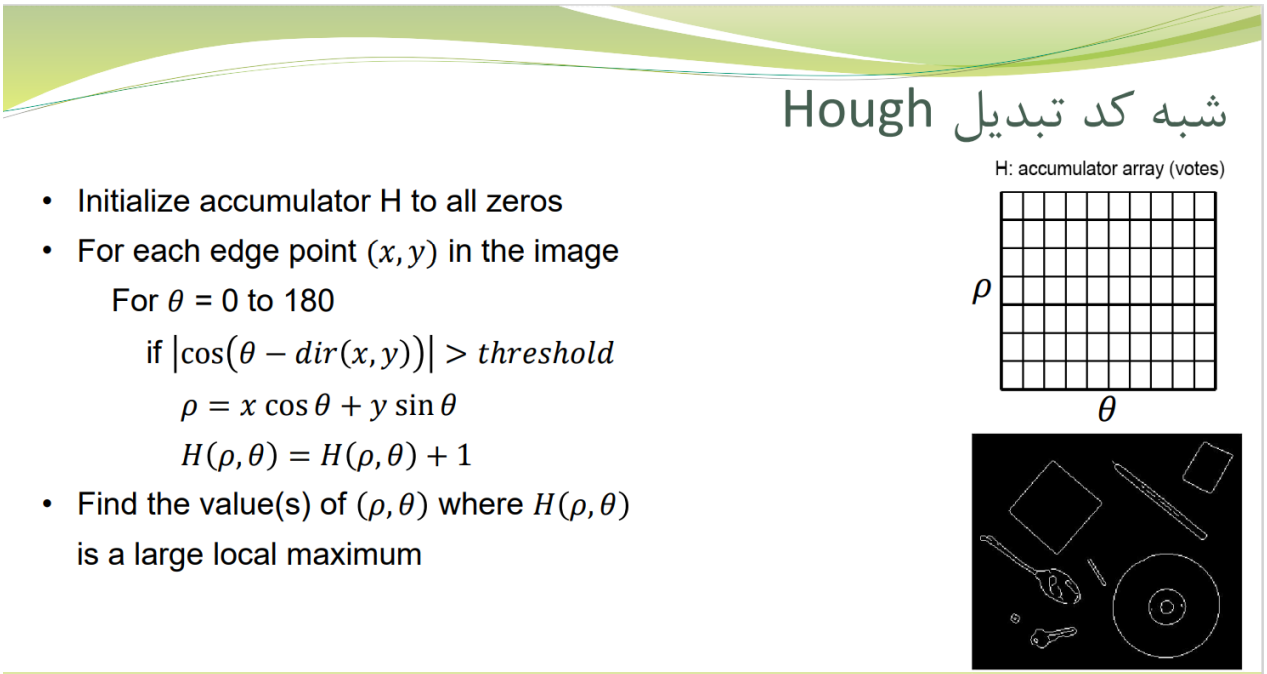
خروجی این تابع خطوط پیدا شده هستند و دیگر نیازی نیست با استفاده از فرمول ها خروجی ها را از نقطه در فضای قطبی به خط در فضای دکارتی تبدیل کنیم.

```
#TODO
lines = cv2.HoughLinesP(dst, 1, np.pi/180, 100, minLineLength=10, maxLineGap=15)

im3 = im.copy()
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(im3, (x1, y1), (x2, y2), (0, 100, 80), 1)

plt.imshow(im3)
plt.title('Image After Line Detection By Probabilistic Hough')
plt.axis('off')
plt.show()
```



سوال دوم) ج: پیاده سازی تبدیل Hough:

شبه کد تبدیل Hough در صفحه ی ۲۲ اسلاید FCV_09 موجود است. طبق همین شبه کد آن را در تابع my_houghدر پایتون پیاده سازی کردیم. این تابع برای انجام کار خود به مجموعه نقاط که همان آرایه ی dst است نیاز دارد که همان نقاط روی لبه ها هستند. همچنین برای درست کار کردن باید یک پارامتر threshold هم داشته باشیم علاوه بر آن باید تقسیمات (ρ,Θ) هم مشخص باشد پس دو پارامتر rho و thetaرا هم به عنوان ورودی در نظر میگیریم.

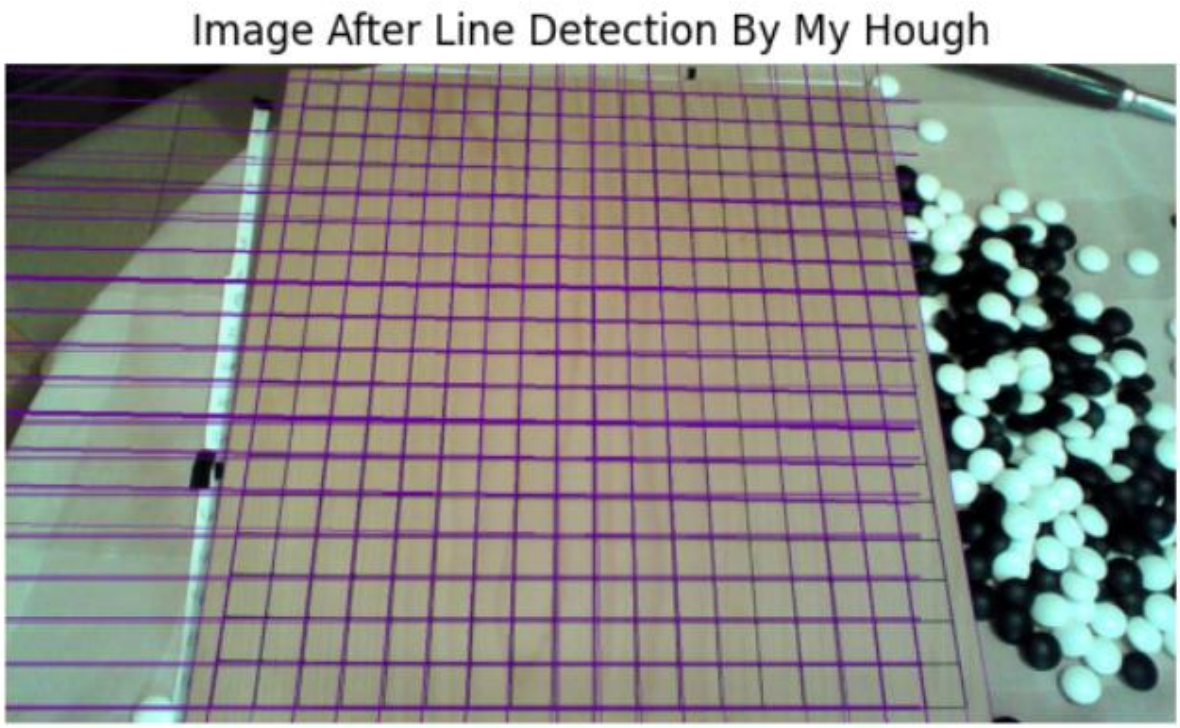



```
def my_hough(dst, treshhold, rho, theta):
    xs, ys = dst.shape
    z = np.sqrt(xs**2 + ys**2)
    max_tresh = int(z)

    thetas = np.deg2rad(np.arange(-90, 90, theta))
    rhos = np.linspace(-max_tresh, max_tresh, max_tresh * 2)


    hough = np.zeros((max_tresh * 2, len(thetas)))
    points = np.argwhere(dst)

    ro = 0
    for y, x in points:
        for thta in range(len(thetas)):
            ro = x * np.cos(thetas[thta]) + y * np.sin(thetas[thta]) + max_tresh
            hough[int(ro), thta] += 1

    less_than_treshold = np.argwhere(hough >= treshhold)
    return [[rhos[i], thetas[j]] for x, y in less_than_treshold]
```

مقایسه خروجی تابع پیاده سازی شده با خروجی تابع آماده:

خروجی تقریبا مشابه است چون threshold را همان ۲۵۰ پارامتر ها را مثل قبلی برای تقسیمات (ρ,Θ) درنظر گرفتم اما زمان اجرای کامل تابع خیلی بیشتر از تابع آماده شد.

```
im4 = im.copy()
for ro, theta in lines:
    x1 = int(ro * np.cos(theta) + 1000 * (-np.sin(theta)))
    y1 = int(ro * np.sin(theta) + 1000 * (np.cos(theta)))
    x2 = int(ro * np.cos(theta) - 1000 * (-np.sin(theta)))
    y2 = int(np.sin(theta) * ro - 1000 * (np.cos(theta)))
    cv2.line(im4, (x1,y1), (x2,y2), (100, 0, 150), 1)

plt.imshow(im4)
plt.title('Image After Line Detection By My Hough')
plt.axis('off')
plt.show()
```



پاسخ تمرین ششم **مهدیه نادری: ۹۸۵۲۲۰۷۶**

سوال سوم)نتیجه ی اجرای کد و مقایسه با سوال قبل:

منبع: ویدیو ۹ و صفحات ۳۰ تا ۳۳ اسلاید FCV_09

https://www.programcreek.com/python/example/110661/cv2.createLineSegmentDetector

https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html

در خط اول همان تصویر را با کمک تابع cv2.imread() خوانده است که آرگومان اول آدرس تصویر و نام آن و آرگومان دوم که صفر داده به معنی این است که تصویر در کانال سیاه و سفید خوانده شود.

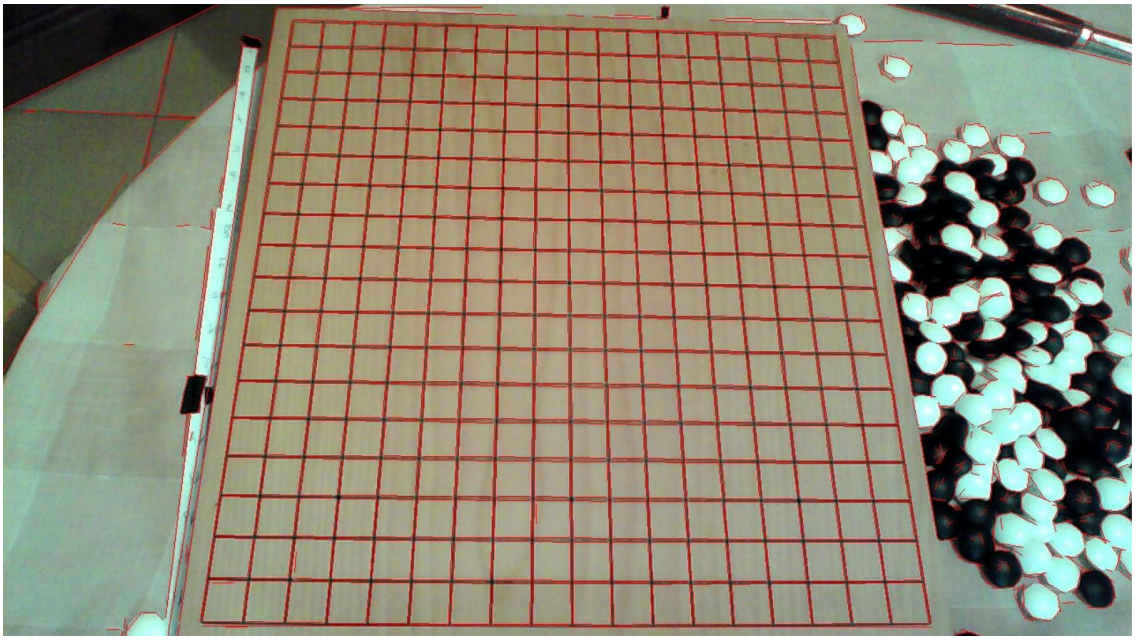
در خط بعدی همان تصویر را با کمک همان تابع در کانال رنگی خوانده است.

در خط سوم یک نمونه از کلاس cv2. LineSegmentDetector ایجاد کرده است.

سپس تابع detect این نمونه را با ورودی تصویر سیاه و سفید فراخوانی کرده است که خروجی درایه ی ۰ ام آن شامل خطوط تشخیص داده شده میباشد برای همین آن را در متغیر lines ذخیره کرده است.

بعد از آن با کمک متود drawSegments(imgcolorous,lines) خطوط پیدا شده را روی تصویر رنگی رسم کرده است. این تابع تصویر و خطوط را به عنوان پارامتر ورودی میگیرد که برای تصویر ورودی تصویر رنگی به آن داده شده است.

سپس با کمک cv2.imwrite('UNKNOWN.jpg',drawn_img) تصویر ایجاد شده را ذخیره کرده است که به صورت زیر میباشد.



کلاس cv2. LineSegmentDetector برای اجرای الگوریتم LSD برای تشخیص پاره خط ها به کار میرود. تفاوت پاره خط با خط این است که برای تشخیص خط داشتن دو نقطه کافی است اما برای تشخیص پاره خط باید دو سر بسته ی آن را هم پیدا کنیم برای همین تبدیل هاف برای اینکار کافی نیست.

پاسخ تمرین ششم **مهدیه نادری: ۹۸۵۲۲۰۷۶**

همچنین تبدیل هاف قبل از اجرای خود آن نیاز داشت که با استفاده از یک الگوریتم دیگر به نام Canny لبه ها را پیدا کنیم و پارامتر های استفاده شده در آن برای پیدا کردن نقاط لبه و ضخامت لبه ی پیدا شده در دقت الگوریتم هاف تاثیر داشت. همچنین خود الگوریتم هاف پارامتر هایی مثل ترشولد یا تقسیمات رو و تتا و همینطور ماکسیمم فاصله ی بین خطوط و مینیمم طول خط برای تشخیص پاره خط نیاز داشت که زمان اجرا را بیشتر و پیاده سازی را سخت میکرد همچنین طول میکشید تا بتوانیم بهترین مقادیر برای این پارامتر ها را پیدا کنیم. یا حتی نویز هایی که تصویر داشت روی لبه یابی ها موثر بود.

برای اینکه Hough بتواند نقاط انتهایی یک بخش را شناسایی کند ، باید روش های تقسیم بندی نیز اعمال شود که همین اتلاف زمانی دارد.

از طرفی تبدیل هاف ماهیت تصادفی داشت و هر بار که اجرا میشد خطوط متفاوتی را میتوانست ارائه دهد.

LSD معمولا نتایج دقیق تری از Hough ارائه میدهد چون از جهت گرادیان نیز استفاده می کند. همانطور که در خروجی LSD مشاهده می شود، خیلی تمیزتر و دقیقتر از Hough توانسته پاره خط ها را پیدا کند.

الگوریتم LSD ۲ سر پاره خط را پیدا میکند و کل خط را روی تصویر رسم نمیکند و از این جهت خیلی بهتر است. برای اینکه بتواند خط را محدود کند و از Canny هم استفاده نکند جهت گرادیان هر پیکسل را حساب میکند. راستایی که تعداد قابل توجهی نقطه با جهت گرادیان مشابه داریم میتوانند مشخص کننده ی یک خط باشند.

علاوه بر این در LSD تصویر ورودی از کانال خاکستری است،در صورتی که در تبدیل Hough طیف تصویر ورودی باید باینری باشد. الگوریتم LSD نیازی به پارامتر های اضافه تری که Hough میخواست (بالاتر توضیح داده شد) ندارد.

الگوریتمHough برای تشخیص خط های خاص تر به دلیل داشتن پارامترهای مختلف، کاربردی تر است زیرا با تنظیم و ترکیب هر یک از این پارامترها باهم (مثلا حداکثر گپ خط و حداقل طول خط) می توان خط های خاص مورد نظر خود را بدست آورد.

پیچیدگی زمانی الگوریتم LSD خطی است اما پیچیدگی زمانی الگوریتم تبدیل Hough به پارامتر های آن بستگی دارد.

سوال چهارم)RGB و CMYK:

منبع: اسلاید FCV_10

- اگر هر یک از مولفه‌های R، G و B توسط ۸ بیت نشان داده شوند، هر پیکسل رنگی دارای عمق ۲۴ بیت خواهد بود
- تعداد کل رنگ‌های متمایز برای هر پیکسل: $2^{24} = 16{,}777{,}216$

فضای CMY و RGB

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \qquad \begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$K = 1 - max(R, G, B)$$

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1-K \\ 1-K \\ 1-K \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

```
def rgb_to_cmyk(r, g, b, RGB_SCALE = 255, CMYK_SCALE = 100):

  #TODO
    if [r, g, b] == (0, 0, 0):
        return 0, 0, 0, CMYK_SCALE
    r /= RGB_SCALE
    g /= RGB_SCALE
    b /= RGB_SCALE

    max_scale = max(r, g, b)
    k = (1 - max_scale)

    return round(CMYK_SCALE * (1 - k - r) / (1 - k)), round(CMYK_SCALE * (1 - k - g) / (1 - k)), round(CMYK_SCALE * (1 - k - b) /
```

```
rgb_to_cmyk(25, 56, 25)
```

```
(55, 0, 55, 78)
```

```
def cmyk_to_rgb(c, m, y, k, CMYK_SCALE = 100, RGB_SCALE = 255):

  #TODO

    c /= CMYK_SCALE
    m /= CMYK_SCALE
    y /= CMYK_SCALE
    k /= CMYK_SCALE

    return round(RGB_SCALE * (1 - c) * (1 - k)), round(RGB_SCALE * (1 - m) * (1 - k)), round(RGB_SCALE * (1 - y) * (1 - k))
```

```
cmyk_to_rgb(55, 0, 55, 78)
```

```
(25, 56, 25)
```

سوال پنجم)پارمتر های HSIو VوLو I:

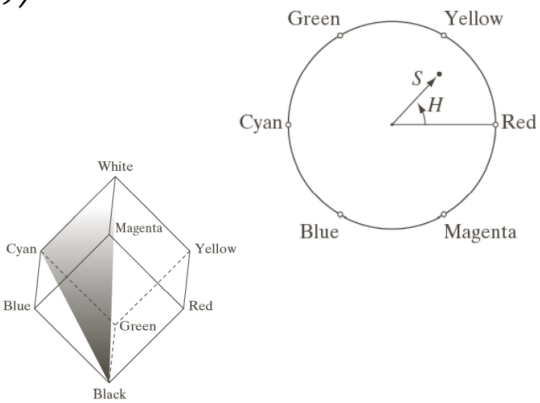
تبدیل RGB به HSI

$$\theta = cos^{-1}\left(\frac{(R-G)+(R-B)}{2\sqrt{(R-G)^2+(R-B)(G-B)}}\right)$$

$$H = \begin{cases} \theta, & if\ B \le G \\ 360-\theta & if\ B > G \end{cases}$$

$$S = 1 - 3\frac{min(R,G,B)}{R+G+B}$$

$$I = \frac{R+G+B}{3}$$



تبدیل RGB به Gray

- بسیاری از الگوریتم‌های پردازش بر روی تصاویر ۱ کاناله عمل می‌کنند
- در OpenCV می‌توان با دستور cvtColor و حالت RGB2Gray تصویر را به سطح خاکستری تبدیل کرد

$$Y = 0.299R + 0.587G + 0.114B$$

- میزان شدت روشنایی با وزن یکسان برای هر سه رنگ اصلی چندان مناسب نیست

$$V = max(R,G,B)$$

$$L = \frac{max(R,G,B)+min(R,G,B)}{2}$$

خروجی کد:

```
H = 278.5130242830111
S = (0.5301204819277108, 3)
I = (0.5424836601307189, 3)
HSI: (278.5130242830111, (0.5301204819277108, 3), (0.5424836601307189, 3))
V = (0.7843137254901961, 3)
L = 0.5196078431372549
Y = 105.80499999999999
```