

هنگامی که از تصویر تبدیل فوریه میگیریم و آن را بر تابع تخریب تقسیم میکنیم، مولفه نویز تصویر تقویت میشود.

اگر تصویر را به صورت زیر مدل کنیم که با تابع تخریب کانوالو شده و سپس با نویز جمع شونده، جمع شده،

$$g(x, y) = (h \star f)(x, y) + \eta(x, y)$$

میتوانیم آن را به فضای فرکانس ببریم و با تقسیم بر تابع تخریب اثر تخریب را از بین ببریم،

$$G(u, v) = H(u, v)F(u, v) + N(u, v)$$

$$\hat{F}(u, v) = \frac{G(u, v)}{H(u, v)}$$

$$\hat{F}(u, v) = F(u, v) + \frac{N(u, v)}{H(u, v)}$$

ولی با توجه به این که تابع تخریب معمولا در فرکانس های بالا مقدار بسیار کمی دارد و نویز تصویر هم فرکانس بالاست، پس از تقسیم مولفه نویز به تابع تخریب مقدار نویز به شدت تقویت میشود.

برای جلوگیری از این اتفاق، میتوانیم از فیلتر پایین گذر و یا فیلتر ویز استفاده کنیم.

با توجه به این که نویز معمولا در فرکانس های بالا قرار دارد، میتوان با استفاده از فیلتر پایین گذر مقدار آن را کاهش داد ولی طبیعتا بخشی از سیگنال اصلی تصویر نیز تخریب خواهد شد.

```
def deconvolution_wiener(src: np.ndarray, psf: np.ndarray, K: np.float64) -> np.ndarray:
    # ...
    psf = np.float64(psf)
    psf /= np.sum(psf)
    dummy = np.copy(src)
    dummy = fft2(dummy)
    psf = fft2(psf, s = src.shape)
    psf = np.conj(psf) / (np.abs(psf)**2 + K)
    deconvolved = dummy * psf
    deconvolved = np.abs(ifft2(deconvolved))
    return deconvolved
```

ابتدا psf را به float تبدیل میکنیم و آن را نرمال میکنیم، از تصویر وتابع تحریب تبدیل فوریه میگیریم سپس طبق رابطه زیر فیلر وینر را اعمال میکنیم:

$$\hat{F}(u, v) = \left[\frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + K} \right] G(u, v)$$

$$|H(u, v)|^2 = H(u, v)H^*(u, v)$$

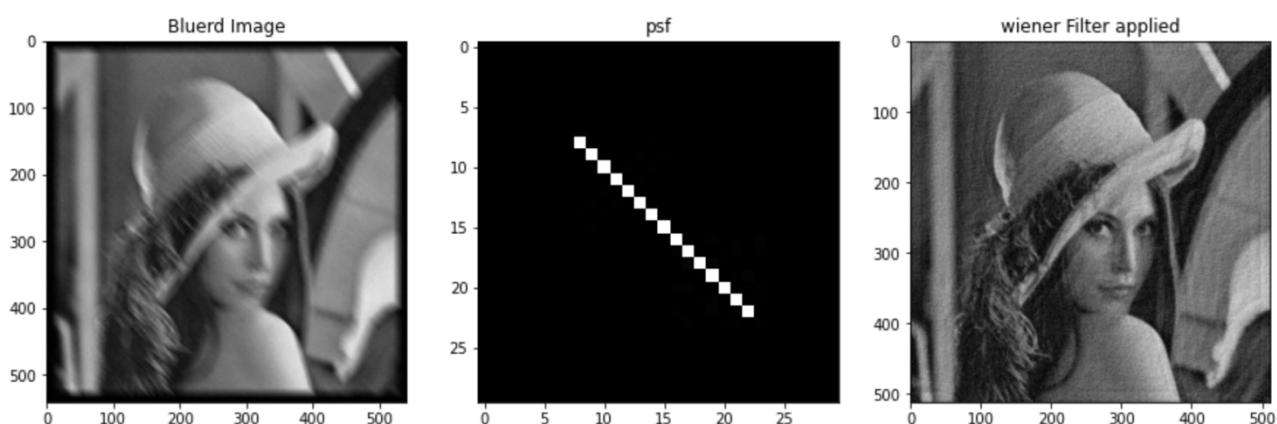
از دو رابطه بالا، رابطه زیر نتیجه میشود که در کد پیاده سازی شده است

$$\hat{F}(u, v) = \left[\frac{H^*(u, v)}{|H(u, v)|^2 + K} \right] G(u, v)$$

در پایان، تبدیل فوریه معکوس میگیریم.

```
deblurred_img = deconvolution_wiener(img_1, psf_1, K=0.01)
deblurred_img = deblurred_img[:512,:512]
```

با مقدار $K=0.01$ نتایج زیر حاصل شده است. همانگونه که مشخص است کیفیت تصویر بهبود یافته است.



```
def ssim(image_1: np.ndarray, image2: np.ndarray):
    return skimage.metrics.structural_similarity(image_1, image2, data_range=256);
```

برای ارزیابی نتایج از معیار ssim به صورت بالا استفاده میکنیم.

```
print("ssim for k=0.01: {}".format(ssim(img_base,deblurred_img)))
```

```
ssim for k=0.01: 0.31120744589694177
```

معیار ssim را برای تصویر حاصل از $k=0.01$ محاسبه میکنیم.

```
max_value = 0
max_k = 0
for K in range(100):
    K /= 100
    ssim_value = ssim(img_base, deconvolution_wiener(img_1, psf_1, K)[:512,:512])
    if ssim_value > max_value:
        max_value = ssim_value
        max_k = K
print("max ssim for diffrent Ks: {}".format(max_value))
print("corresponding K: {}".format(max_k))
```

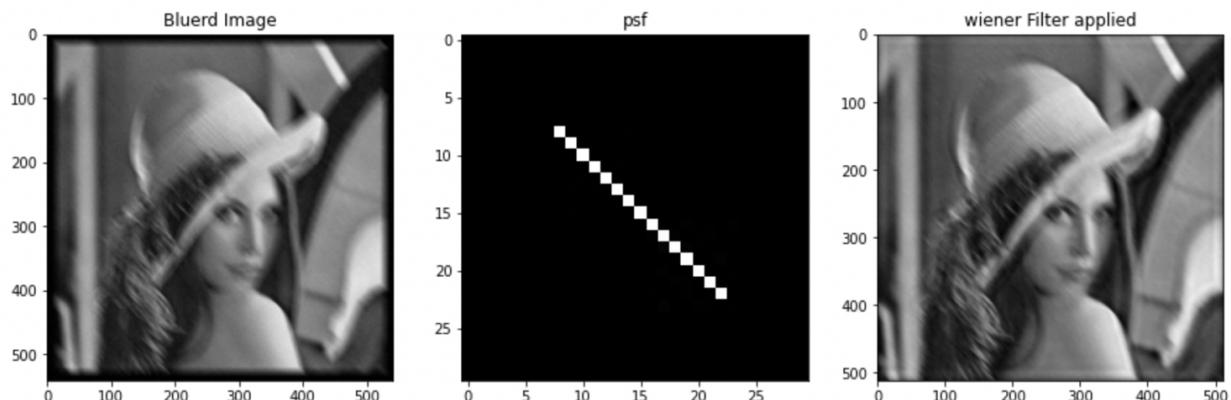
```
max ssim for diffrent Ks: 0.6139065355450042
corresponding K: 0.16
```

برای یافتن بیشترین ssim، با استفاده از یک حلقه for مقدار k را از ۰ تا ۱ با گام ۰.۰۱ حرکت می دهیم و به ازای هر k مقدار ssim را محاسبه میکنیم، که نتایج بالا حاصل شد.

اگر با $k=0.16$ تصویر را بهبود دهیم نتایج زیر حاصل میشود:

```
deblurred_max_ssim = deconvolution_wiener(img_1, psf_1, K=0.16)
deblurred_max_ssim = deblurred_max_ssim[:512,:512]

display = [img_1, psf_1, deblurred_max_ssim]
label = ['Bluerd Image', 'psf', 'wiener Filter applied']
fig = plt.figure(figsize=(15, 10))
for i in range(len(display)):
    fig.add_subplot(1, 3, i+1)
    plt.imshow(display[i], cmap = 'gray')
    plt.title(label[i])
plt.show()
```



قسمت II مقاله در واقع توضیح الگوریتم کنی است که به صورت خلاصه به شرح زیر است:
لبه یاب canny طبق مراحل زیر عمل میکند:

1 - ابتدا تصویر را کمی هموار میکنیم تا اثر نویز در خروجی مشتق کمتر شود.
این هموار سازی معمولاً به وسیله کانوالو کردن فیلتر گاوی دو بعدی با تصویر انجام میشود که رابطه آن به صورت زیر است:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

2 - گرادیان تصویر را محاسبه میکنیم،تابع محاسبه گرادیان، کرنل های مربوط به عملگر سوبل را در تصویر کانوالو میکند و از این طریق گرادیان افقی و عمودی تصویر را برمی گرداند. این کرنل ها به صورت زیر هستند:

-1	0	+1
-2	0	+2
-1	0	+1

-1	-2	-1
0	0	0
+1	+2	+1

اندازه و جهت گرادیان نیز از روابط زیر محاسبه میشوند:

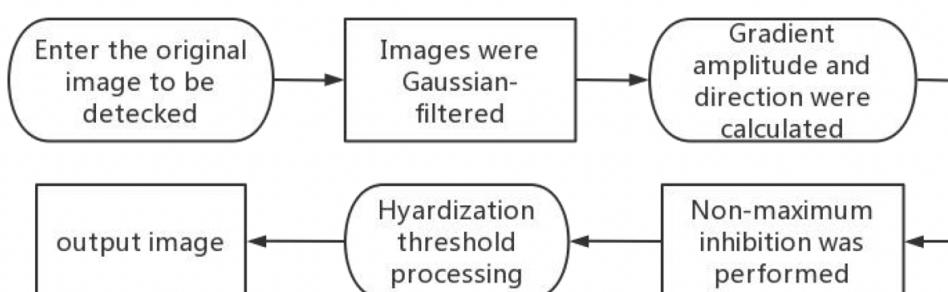
$$O = \sqrt{I_x^2 + I_y^2}$$

$$\theta = \arctan(I_y / I_x)$$

3 - با توجه به جهت گرادیان، مقادیر غیر بیشینه را در جهت عمود بر لبه حذف میکنیم تا دقیق تر افزایش یابد همچنین ضخامت لبه در همه جا یک پیکسل شود. روش کار به این صورت است که پیکسل های تصویر را پیمایش میکنیم و اگر اندازه گرادیان آن پیکسل از اندازه گرادیان دو پیکسل مجاور که در جهت گرادیان قرار دارند بیشتر نبود، آن پیکسل به عنوان غیر لبه نشانه گذاری میشود.

4 - با استفاده از آستانه گذاری دو مرحله ای، نقاط را به سه دسته تقسیم میکنیم، نقاطی که از آستانه بالایی بیشتر باشند حتماً لبه هستند، نقاطی که پایین تر از آستانه پایینی باشند حتماً لبه نیستند، نقاط بین دو آستانه به شرط آنکه به یک ناحیه قوی متصل باشند (یعنی در همسایگی 3×3 آنها حداقل یک پیکسل قوی وجود داشته باشد)، لبه هستند و در غیر اینصورت لبه نیستند.

این مراحل را میتوان به صورت زیر نمایش داد:



الگوریتم کنی اصلی برای تصاویر بسیار نویزی مناسب نیست، از این رو روش جدیدی برای فیلتر تصاویری که نویز pretzel دارد طراحی شد، استفاده از گرادیان در چهار جهت و تعیین حد آستانه ها با استفاده از الگوریتم، به جای تعیین دستی آن ها، میتواند منجر به تشخیص لبه بهتر و دقیق تری شود.

اصول طراحی فیلتر جدید:

تمایز بین نویز نقاط لبه، نویز pretzel در عکس بسیار مشخص است، و مشخصه آن این است که پیکسل نویزی با ۸ پیکسل همسایه اش خیلی متفاوت است، از این جهت میتوان با بررسی دامنه تغییرات پیکسل نسبت به پیکسل های همسایه اش، آن را شناسایی کرد. برای بررسی دامنه تغییرات یک نقطه باید به مقدار مشتق آن نگاه کرد. به دلیل آن که پیکسل های لبه نیز رفتاری مشابه دارند و نسبت به پیکسل های همسایه تغییر زیادی دارند، بررسی های بیشتری لازم است.

یک پیکسل تصویر خاکستری از ۰ تا ۲۵۵ میتواند مقدار داشته باشد، از این رو پیکسل های با مقدار ۰ یا میتواند نویز باشد، مشتق $f(q,r)$ از رابطه زیر تعیین میشود:

$$f(q, r) = \frac{F(q, r) - F(x, y)}{1}$$

شرح الگوریتم:

- ۱ - یک پنجره ۳*۳ و یک حد آستانه در نظر میگیریم و پنجره را حرکت میدهیم.
- ۲ - اگر مرکز پنجره مقدار ۰ یا ۲۵۵ نداشت، پنجره به حرکت ادامه میدهد و گرنه به مرحله ۳ میرویم
- ۳ - مقدار پیکسل مرکزی و مشتق $f(q,r)$ را در هشت جهت تعیین میکنیم. اگر قدر مطلق ماکریم این هشت مشتق از حد آستانه کمتر بود، پنجره به حرکت ادامه میدهد، اگر قدر مطلق مینیم این هشت مشتق از حد آستانه بیشتر بود به مرحله ۴ میرویم، اگر هیچکدام از شرایط بالا اتفاق نیافتد به مرحله ۵ میرویم
- ۴ - $F(x,y)$ را به عنوان مرکز پنجره در نظر میگیریم و مقدار میانه پنجره را با آن جایگزین میکنیم
- ۵ - مرکز پنجره را به نقطه ای با $T^{f(q,r)}$ منتقل میکنیم. مرکز این پنجره (i,j) است که که مقادیر مشتق در ۸ جهت اطراف همسایگی را مجدداً محاسبه میکند. اگر بیش از سه عدد از این مشتق ها کمتر از T بودند، این نقطه یک نقطه لبه است. در غیر این صورت به مرحله ۴ میرویم.
- ۶ - پنجره تا انتهای حرکت میکند و الگوریتم پایان میابد

در الگوریتم کنی اصلی فقط از مشتق عمودی و افقی برای محاسبه گرادیان استفاده میشود که باعث میشود الگوریتم مستعد نویز خارجی باشد، از این رو ما جهت های ۴۵ و ۱۳۵ درجه را به عملگر سوبل اضافه میکنیم تا گرادیان در این جهت ها نیز محاسبه شود. در زیر میتوانید عملگر های گرادیان برای محاسبه گرادیان در چهار جهت با ماتریس های ۳ در ۳ را مشاهده کنید.

$$H_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad H_y = \begin{bmatrix} -1 & -2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$H_{45^\circ} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad H_{135^\circ} = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

این عملگرها با تصویر کانوالو میشوند و مولفه های M_x , M_y , M_{45° , M_{135° در چهار جهت به دست می آیند. در نهایت برای محاسبه اندازه گرادیان با استفاده از این مولفه ها از رابطه زیر استفاده میکنیم:

$$M = \sqrt{M_x^2 + M_y^2 + M_{45^\circ}^2 + M_{135^\circ}^2}$$

با استفاده از روابط زیر مولفه های 45° و 135° در محاسبه مولفه های افقی و عمودی تاثیر میدهیم:

$$M_x = M_x + \frac{\sqrt{2}}{2} M_{45^\circ} - \frac{\sqrt{2}}{2} M_{135^\circ}$$

$$M_y = M_y + \frac{\sqrt{2}}{2} M_{45^\circ} - \frac{\sqrt{2}}{2} M_{135^\circ}$$

الگوریتم *otsu* برای تعیین حد آستانه بالا و پایین

در الگوریتم اصلی کنی حد آستانه به صورت دستی تعیین میشود در حالی که استفاده از الگوریتم *otsu* برای تعیین این حد آستانه ها میتواند با تعیین مقادیر بهینه، باعث سازگاری بیشتر الگوریتم شود و لبه ها با دقت بیشتری تعیین شوند.

الگوریتم *otsu* برای آستانه گذاری تصاویر گرادیان بعد از مرحله حذف مقادیر بیشینه توسط یک محقق ژاپنی معرفی شده است. این الگوریتم براساس مقیاس خاکستری تصویر را به پس زمینه و پیش زمینه تقسیم میکند.

اگر تصویر هدف سایز $M * N$ داشته باشد و دارای L سطح متفاوت خاکستری باشد، الگوریتم واریانس مربوطه را در مقیاس خاکستری پیکسل از 0 تا $L-1$ محاسبه میکند. از طریق مقایسه مداوم، در نهایت بیشینه واریانس کلاس بین پیش زمینه و پس زمینه به دست می آید. هر چه واریانس بیشتر باشد، طبقه بندی صحیح تر است.

در تصاویر با سایز $M * N$ که دارای L سطح خاکستری متفاوت باشند، مقدار پیکسل متناظر با هر سطح خاکستری n_i است. احتمال یک پیکسل خاص P_i است که از رابطه زیر به دست می آید:

$$P_i = \frac{n_i}{MN}$$

حد آستانه k , $k \in [0, L-1]$ را در نظر بگیرید که پیکسل های تصویر را به دو دسته w_0 , w_1 طبقه بندی میکند، که بازه های آن ها به ترتیب $[0, k]$, $w_1 \in [k+1, L-1]$ است. در نتیجه احتمال اینکه یک پیکسل در w_0 یا w_1 بیافتد $P_{w_0}(k), P_{w_1}(k)$ است. به فرمول های متناظر نگاه کنید:

$$P_{w_0}(K) = \sum_{i=0}^k p_i$$

$$P_{w_1}(k) = \sum_{i=k+1}^{L-1} p_i = 1 - P_{w_0}(k)$$

مقادیر متوسط برای پیش زمینه w_0 و پس زمینه w_1 $h_0(k)$ و $h_1(k)$ هستند. همانگونه که در روابط زیر نشان داده شده است.

$$h_0(k) = \sum_{i=0}^k ip_i (i/W_0) = \frac{1}{p_{w0}(k)} \sum_{i=0}^k ip_i$$

$$h_1(k) = \sum_{i=k+1}^{L-1} ip_i (i/W_1) = \frac{1}{p_{w1}(k)} \sum_{i=k+1}^{L-1} ip_i$$

مقدار متوسط کل تصویر برابر است با $h(k)$

$$h(k) = \sum_{i=0}^{L-1} ip_i$$

حداکثر واریانس بین کلاسی بین پیش زمینه W_0 و پس زمینه W_1 برابر است با δ^2

$$\delta^2 = P_{w0}(k) [h_0(k) - h(k)]^2 + P_{w1}(k) [h_1(k) - h(k)]^2$$

مقدار واریانس بین کلاسی را برای k های مختلف میابد، هنگامی که ب به بیشترین مقدار خود برسد k را به عنوان آستانه بالایی و $\frac{k}{2}$ را به عنوان آستانه پایینی در نظر میگیریم.

در لبه ها معمولاً تغییر رنگ داریم، می توانیم از همین موضوع برای تشخیص لبه استفاده کنیم. مشتق تصویر تغییرات رنگ را نشان می دهد، مشتق افقی تغییرات افقی و مشتق عمودی تغییرات عمودی را نشان میدهد.

رابطه مشتق دو طرفه به صورت زیر است:

$$\hat{f}_x = \frac{f(x-1, y) - f(x+1, y)}{2}$$

بر اساس همین رابطه، برای محاسبه مشتق افقی هر پیکسل میتوانیم ماتریس زیر را با آن پیکسل و همسایه هایش کانوالو کنیم.

1	.	1
---	---	---

به همین ترتیب، برای محاسبه مشتق عمودی هر پیکسل میتوانیم ماتریس زیر را با آن پیکسل و همسایه هایش کانوالو کنیم.

1
.
1

با ترکیب مولفه های عمودی و افقی مشتق، بردار گرادیان به دست می آید،

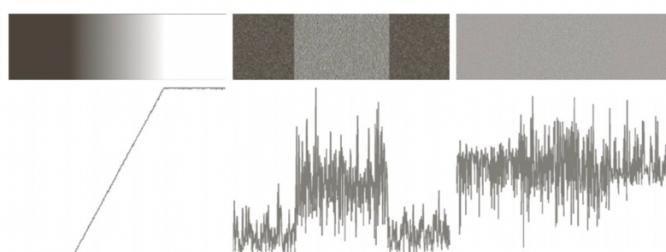
$$\nabla f(x, y) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

اندازه و جهت بردار گرادیان از روابط زیر محاسبه می شوند:

$$M(x, y) = \|\nabla f\| = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2}$$

$$\alpha(x, y) = \text{dir}(\nabla f) = \text{atan2}(g_y, g_x)$$

با توجه به این که مقادیر نویز در پیکسل های همسایه میتوانند مقادیر کاملاً متفاوتی داشته باشند و مشتق هم تغییرات را نشان می دهد، نویز ها در خروجی مشتق به شدت تقویت میشوند.



به همین دلیل قبل از مشتق گرفتن، کمی تصویر را هموار میکنیم، اما هموار کردن تصویر باعث ضعیف شدن لبه ها نیز می شود، برای جلوگیری از ضعیف شدن لبه ها در جهت مشتق، تصویر را فقط در خلاف جهتی که میخواهیم مشتق بگیریم هموار میکنیم.

به این ترتیب هم از ضعیف شدن لبه ها در جهت مشتق جلوگیری کرده ایم و هم تا حدی نویز را کاهش داده ایم. با ضرب آرایه هموارساز و آرایه مشتق گیر در هم، عملگر *sobel* ساخته میشود. در عملگر سوبل آرایه هموارساز وزن دار است.

$$\begin{array}{c|c} \begin{array}{c|c|c} +1 & & \\ +2 & -1 & 0 & +1 \\ +1 & & \end{array} & = \end{array} \quad \begin{array}{c|c|c} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{array} \quad \begin{array}{c|c|c} -1 & & \\ 0 & +1 & +2 & +1 \\ +1 & & \end{array} = \quad \begin{array}{c|c|c} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{array}$$

```
def modifiedSobelDetector(image):
    #define Sobel filters
    Gx = np.array([[1,0,-1], [2,0,-2], [1,0,-1]])
    Gy = np.array([[1,2,1], [0,0,0], [-1,-2,-1]])
```

در این قسمت از کد، عملگر های سوبل عمودی و افقی تعریف شده اند.

```
h , w = image.shape
kernel_h = 3
kernel_w = 3
kernel_half = 1
```

در این قسمت ابعاد تصویر استخراج و ابعاد کرنل سوبل تعیین شده است.

```
sobel_filtered_image = np.zeros(shape=(h,w))
image = np.pad(array= image, pad_width=(kernel_h//2 , kernel_w//2), mode='constant', constant_values=(0,0))
# Now sweep the image in both x and y directions and compute the output#
for i in range(kernel_half , h-1):
    for j in range(kernel_half , w-1):
        cover_area = image[i - kernel_half: i + kernel_half + 1, j - kernel_half: j + kernel_half + 1]
        gx = np.sum(np.multiply(Gx , cover_area))
        gy = np.sum(np.multiply(Gy , cover_area))
        sobel_filtered_image[i-kernel_half , j-kernel_half] = np.sqrt(gx ** 2 + gy ** 2)
```

حال باید عملگر سوبل را روی تصویر اعمال کنیم، برای این منظور ابتدا ماتریسی با ابعاد تصویر اولیه و درایه های صفر ساخته شده، در مرحله بعد باید کرنل سوبل را بر روی پیکسل های تصویر کانوالو کنیم، ولی برای پیکسل های مرزی به مشکل بروخواهیم خورد پس ابتدا تصویر را پد میکنیم.

به وسیله دو حلقه *for* روی پیکسل های تصویر حرکت میکنیم و در هر مرحله، کرنل سوبل را روی آن پیکسل و همسایه هایش کانوالو میکنیم، بازه همسایگی براساس ابعاد کرنل سوبل تعیین میشود.

پس از محاسبه مولفه های عمودی و افقی گرادیان، اندازه آن را محاسبه و در پیکسل متناظر از ماتریس اولیه ای که ساختیم ذخیره میکنیم.

به این ترتیب در پایان عملیات، ماتریسی متناظر با تصویر داریم که اندازه گرادیان در هر نقطه از تصویر را نشان میدهد.

همانگونه که قبلا ذکر شد، در لبه ها به دلیل تغییر رنگ، گرادیان افزایش می یابد، بر همین اساس پیکسل هایی که مقدار گرادیان در آن ها از یک حد آستانه مشخص بیشتر است را به عنوان لبه در نظر میگیریم.

این حد آستانه را به صورت نسبی نسبت به بیشترین مقدار گرادیان تعیین میکنیم.

```
maximum_gradient = np.max(sobel_filtered_image)
threshold = 0.3 * maximum_gradient
sobel_filtered_image[sobel_filtered_image < threshold] = 0
sobel_filtered_image[sobel_filtered_image >= threshold] = 1
```

در اینجا حد آستانه ۰.۳ بیشینه گرادیان در نظر گرفته شده و براساس آن تصویر به دو ناحیه ۱ (لبه) و ۰ (غیر لبه) تقسیم شده است.

از جمله مشکلاتی که لبه یاب سوبل دارد میتوان به موارد زیر اشاره کرد:

- ۱ - لبه های پیدا شده قطر متفاوتی دارند
- ۲ - انتخاب پیکسل ها بر اساس حد آستانه باعث میشود لبه ها دچار گسستگی زیادی شوند

برای حل مشکلات ذکر شده لبه یاب *canny* معرفی شده است که طبق مراحل زیر عمل میکند:

- ۵ - ابتدا تصویر را کمی هموار میکنیم تا اثر نویز در خروجی مشتق کمتر شود
- ۶ - گرادیان تصویر را محاسبه میکنیم
- ۷ - با توجه به جهت گرادیان، مقادیر غیر بیشینه را در جهت عمود بر لبه حذف میکنیم
- ۸ - با استفاده از آستانه گذاری دو مرحله ای، نقاط را به سه دسته تقسیم میکنیم، نقاطی که از آستانه بالایی بیشتر باشند حتما لبه هستند، نقاطی که پایین تر از آستانه پایینی باشند حتما لبه نیستند، نقاط بین دو آستانه به شرط آنکه به یک ناحیه قوی متصل باشند لبه هستند و در غیر اینصورت لبه نیستند.

حال به بررسی کد میپردازیم

```
def smoothing(img):
    output = img.copy()

    output = cv2.GaussianBlur(img, (5, 5), 1.3)

    return output
```

تابع هموارسازی، این تابع با استفاده از فیلتر گوسی تصویر را هموار میکند.

```

def comput_gradient(img):

    mag = np.zeros_like(img)
    angle = np.zeros_like(img)

    # we used library here#
    gx = cv2.Sobel(np.float32(img), cv2.CV_64F, 1, 0, 3)
    gy = cv2.Sobel(np.float32(img), cv2.CV_64F, 0, 1, 3)

    mag = np.sqrt(gx **2 + gy**2)
    angle = np.arctan2(gy,gx) * 180 / np.pi

    return mag,angle

```

تابع محاسبه گرادیان، این تابع با استفاده از کرنل های مربوط به عملگر سوبل، گرادیان تصویر را محاسبه و اندازه و جهت آن را برمی گرداند.

```

def NMS(mag,angle):

    ang = angle
    height, width = mag.shape

    for i_x in range(width):
        for i_y in range(height):
            grad_ang = abs(ang[i_y, i_x])
            if grad_ang<= 22.5:
                neighb_indexes = [[i_x-1, i_y],[i_x + 1, i_y]]

            elif grad_ang>22.5 and grad_ang<=(22.5 + 45):
                neighb_indexes = [[i_x-1, i_y-1],[i_x + 1, i_y + 1]]

            elif grad_ang>(22.5 + 45) and grad_ang<=(22.5 + 90):
                neighb_indexes = [[i_x, i_y-1],[i_x, i_y + 1]]

            elif grad_ang>(22.5 + 90) and grad_ang<=(22.5 + 135):
                neighb_indexes = [[i_x-1, i_y + 1],[i_x + 1, i_y-1]]

            elif grad_ang>(22.5 + 135) and grad_ang<=180:
                neighb_indexes = [[i_x-1, i_y],[i_x + 1, i_y]]

            # Non-maximum suppression step
            if width>neighb_indexes[0][0]>= 0 and height>neighb_indexes[0][1]>= 0:
                if mag[i_y, i_x]<mag[neighb_indexes[0][1], neighb_indexes[0][0]]:
                    mag[i_y, i_x]= 0
                continue

            if width>neighb_indexes[1][0]>= 0 and height>neighb_indexes[1][1]>= 0:
                if mag[i_y, i_x]<mag[neighb_indexes[1][1], neighb_indexes[1][0]]:
                    mag[i_y, i_x]= 0

    output = mag

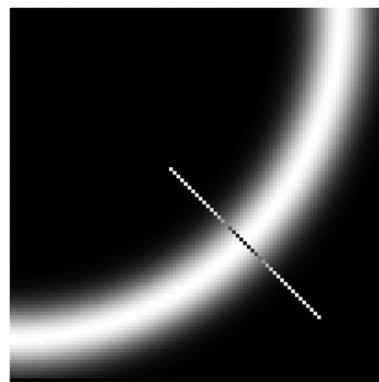
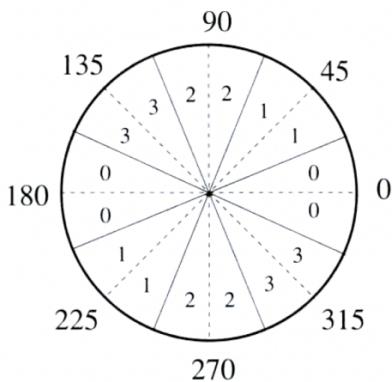
    return output

```

تابع حذف مقادیر غیر بیشینه، به وسیله دو حلقه *for* روی مولفه های گرادیان حرکت میکنیم، در هر مرحله براساس بازه زاویه گرادیان، پیکسل های مورد نظر جهت بررسی را تعیین میکنیم و سپس مقادیر غیر بیشینه را در بین آن پیکسل ها صفر میکنیم.

جهت گرادیان را میتوان به چهار گروه دسته بندی کرد که در شکل زیر نشان داده شده اند.

از بین پیکسل هایی که در همسایگی 3×3 پیکسل مورد بررسی قرار دارند، حذف مقادیر غیر بیشینه بین آن هایی انجام می شود که در جهت گرادیان قرار دارند.



0	0	0	0	1	2	1	3
0	0	0	1	2	1	3	1
0	0	2	1	2	1	1	0
0	1	3	2	1	1	0	0
0	3	2	1	0	0	0	0
2	3	2	0	0	1	0	1
2	3	2	0	1	0	2	0

```
def CheckConnection(img,x,y):
    for i in range(-1,2):
        for j in range(-1,2):
            if img.shape[1]>x + j>=0 and img.shape[0]>y + i>=0:
                if img[y + i,x + j] == 125:
                    img[y + i,x + j] = 255
                    CheckConnection(img,x+j,y+i)
```

این تابع به صورت بازگشتی اتصال بین پیکسل ها را بررسی می کند و قرار است روی مولفه های قوی اعمال شود به ازای هر مولفه قوی اگر در همسایگی 3×3 آن، مولفه ضعیفی وجود داشت، آن را به مولفه قوی تبدیل می کند و این فرایند به صورت بازگشتی تکرار می شود.

```
def hysteresis_threshold(edges,min_th,max_th):
    weak_ids = np.zeros_like(edges)
    strong_ids = np.zeros_like(edges)
    ids = np.zeros_like(edges)

    for i_x in range(edges.shape[1]):
        for i_y in range(edges.shape[0]):

            grad_mag = edges[i_y, i_x]

            if grad_mag<min_th:
                ids[i_y, i_x]= 0
            elif max_th>grad_mag>= min_th:
                ids[i_y, i_x]= 125
            else:
                ids[i_y, i_x]= 255

    Ys,Xs = np.where(ids==255)
    for i in range(len(Ys)):
        CheckConnection(ids,Xs[i],Ys[i])
    Ys,Xs = np.where(ids==125)
    ids[Ys,Xs] = 0
    output = ids

    return output
```

این تابع دو حد آستانه پایین و بالا دریافت می کند، پیکسل هایی که کمتر از حد آستانه پایین باشند حذف، پیکسل هایی که بین دو حد آستانه باشند تبدیل به مولفه ضعیف و پیکسل هایی که بیشتر از حد آستانه بالایی باشند تبدیل به مولفه قوی می شوند.

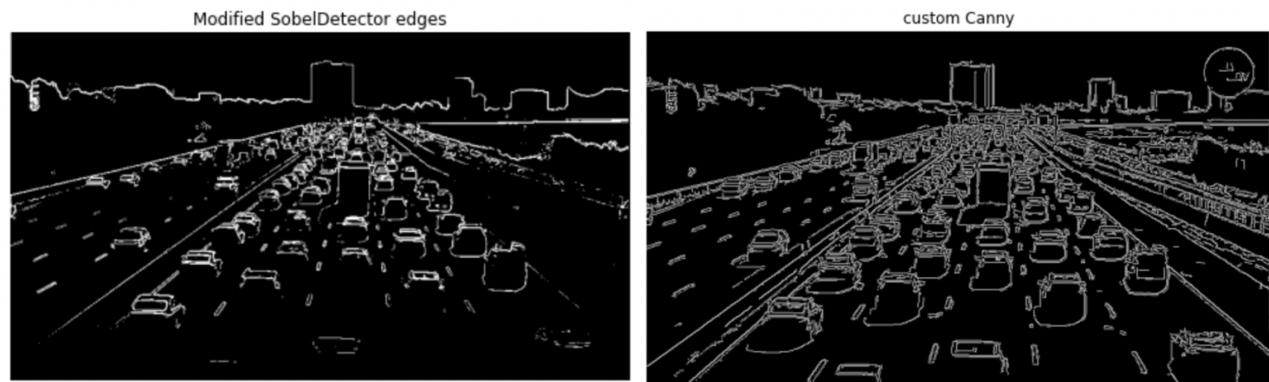
سپس تابع *CheckConnection* را روی مولفه های قوی اجرا می کند.

```

def canny(img,min_th,max_th):
    smooth_img = smoothing(img)
    mag,angle = comput_gradient(smooth_img)
    edges = NMS(mag,angle)
    final_edges = hysteresis_threshold(edges,min_th,max_th)
    return final_edges

```

در این قسمت مراحل شرح داده شده قبلی در تابع *canny* اجرا شده اند.



با مقایسه نتایج این دو لبه یاب متوجه میشویم لبه یاب *canny* بسیار بهتر عمل کرده است، در لبه یاب *canny* لبه ها دارای ضخامت یکسان و پیوسته هستند و لبه یابی با دقت بیشتری انجام شده است.

(ج)

kernel_w و *Kernel_h*

این دو پارامتر اندازه کرنل *sobel* را مشخص میکنند و هر چه بزرگتر باشند لبه های بزرگتر و کلی تر را میابیم و به دلیل اینکه داریم از ناحیه بزرگتری مشتق میگیریم تصویر خروجی تار تر میشود.

حد آستانه لبه یاب *sobel* پیکسل هایی که مقدار گرادیان آنها بیشتر از این حد آستانه باشد به عنوان لبه شناخته میشوند، با افزایش این پارامتر انتخاب لبه سخت گیرانه تر خواهد شد و با اینکه لبه های دقیق تری پیدا میشوند ولی لبه ها دچار گسترش بیشتری خواهند شد.

اندازه کرنل و سیگما در کرنل گاوی
هر چه این دو پارامتر را افزایش دهیم تصویر بیشتر نرم خواهد شد و نویز بیشتری حذف میشود ولی در عین حال جزییات بیشتری از تصویر را از دست میدهیم

حد آستانه های بالا و پایین در آستانه گذاری دو مرحله ای
اگر حد آستانه بالا را افزایش دهیم باعث میشود با وسوسات بیشتری لبه های قوی انتخاب شوند و بیشتر لبه ها به صورت ضعیف دسته بندی شوند از این رو ممکن است یک ناحیه بسته از لبه ها به صورت ضعیف دسته بندی شوند و به دلیل اتصال نداشتن با ناحیه قوی، به عنوان لبه شناخته نشوند.
اگر حد آستانه پایین را بیش از حد کاهش دهیم ممکن است نواحی که لبه نیستند به عنوان لبه شناخته شوند و در صورتی که آن را بیش از حد افزایش دهیم ممکن است در انتخاب لبه ها بیش از حد سختگیری کنیم.