به نام خدا

گزارش تمرین سری صفر درس بینایی کامپیوتر
نام مدرس: دکتر محمدرضا محمدی

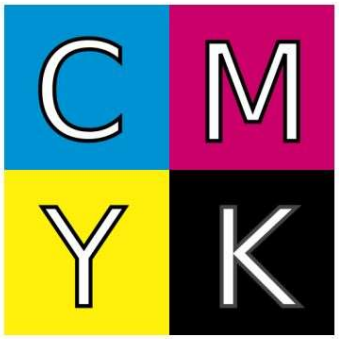
فرزان رحمانی ۹۹۵۲۱۲۷۱

سوال ۱

الف)

مدل رنگ CMYK

- در اغلب پرینترها از ۴ جوهر با رنگ‌های فیروزه‌ای، بنفش روشن، زرد و سیاه استفاده می‌شود
- علت استفاده از جوهر سیاه آن است که چاپ کردن رنگ سیاه با استفاده از ۳ جوهر هزینه‌بر است



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

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

با استفاده از فرمول های اسلاید ها به پیاده سازی پرداختیم. در دو تابع جداگانه هر تبدیل را پیاده سازی کرده ایم و همان طور که میبینید سختی زیادی نداشت.

```
def RGB_to_CMYK(RGB_array):
    K = 1 - np.round((max(RGB_array) / 255), 2) # scale is percentage(0 to 1) of
black ink
    CMYK_array = [0, 0, 0, K] # in CMYK format
    CMYK_array[0:3] = ([1 - K] * 3) - np.round((np.array(RGB_array) / 255), 2) #
scale is percentage(0 to 1) of cyan, magenta and yellow ink
    return CMYK_array
```

```
def CMYK_to_RGB(CMYK_array):
    K = CMYK_array[3]
    RGB_array = np.round(np.array(([1 - K] * 3) - (np.array(color_CMYK[0:3]))) *
255)
    # change to int
    RGB_array = RGB_array.astype(int)
    return RGB_array

color_RGB = [50, 70, 130] # in RGB format
print("RGB: ", color_RGB)

# change to CMYK format
color_CMYK = RGB_to_CMYK(color_RGB)
print("CMYK: ", color_CMYK)

# back to RGB format from CMYK format
color_RGB_from_CMYK = CMYK_to_RGB(color_CMYK)
print("RGB from CMYK: ", color_RGB_from_CMYK)
```

ب) با توجه به دو لینک زیر و استفاده از تابع convert color کتابخانه opencv به پیاده سازی پرداختیم و در وردی اول خود عکس و در ورودی دوم تبدیل مطلوب را دادیم.

```
result = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)
result = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
# https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html
# https://www.geeksforgeeks.org/python-opencv-cv2-cvtcolor-method/
```

ج) بعد از تبدیل به فضای HSV با تابع split یا با استفاده از عملگر slicing کانال های تصویر را جدا کردیم و مانند سوال الف و ب آن را نشان دادیم.

```
# https://www.codespeedy.com/splitting-rgb-and-hsv-values-in-an-image-using-
opencv-python/

hsv_img = convert_to_hsv(image)
print(hsv_img.shape)
# h_channel, s_channel, v_channel = cv2.split(hsv_img)
h_channel = hsv_img[:, :, 0]
s_channel = hsv_img[:, :, 1]
v_channel = hsv_img[:, :, 2]
# show histogram of each channel
image_list = []
# image_list.append([h_channel.ravel(), 'H channel', 'hist'])
image_list.append([h_channel, 'H channel', 'img'])
```

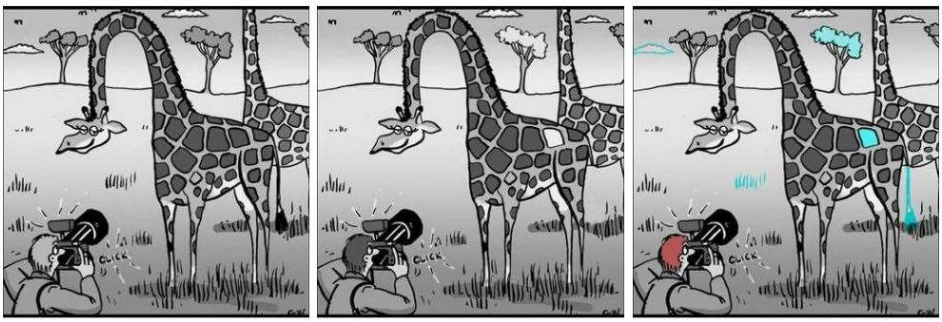
```
image_list.append([s_channel, 'S channel', 'img'])
image_list.append([v_channel, 'V channel', 'img'])
plotter(image_list, 1, 3, True, 20, 10, '2C')
```

د)

با توجه به اسلاید های درس که در زیر آمده است هر تصویر را بعد از خاکستری کردن در روی یک کانال قرار دادیم و کانال سوم را به دلخواه یکی از آن ها گذاشتیم و نتیجه حاصل شد. البته چون ابعاد عکس ها یکی نبود min طول و عرض را گرفته و برش دادیم.

ترکیب دو تصویر خاکستری

- چطور می‌شود از دو تصویر خاکستری یک تصویر رنگی ساخت که تغییرات را مشخص کند؟
- می‌توان هر کانال رنگی از یک تصویر RGB را برابر با یکی از تصاویر قرار داد



ه)

Using different color spaces in computer vision and image processing can provide different benefits, depending on the specific task at hand. Here are some reasons why different color spaces are used:

1. Perception-based color models: The RGB color model is based on the sensitivity of the human eye to red, green, and blue light, but it does not reflect the way humans perceive color. Other color models, such as the HSV and HIS models, are designed to more closely match the way humans perceive color, which can be useful in tasks such as image segmentation and object recognition.

2. Separation of color and intensity information: Some color spaces, such as YCbCr and YIQ, separate color information from intensity information, which can make it easier to process and manipulate color information independently of the image's brightness.
3. Compression and storage: Different color spaces may be more efficient for compressing and storing images than others. For example, the YCbCr color space is commonly used in JPEG compression because it separates color and intensity information and can therefore achieve higher compression rates without significant loss of image quality.
4. Illumination invariance: Some color spaces, such as the HSV and HSI models, are designed to be invariant to changes in illumination, which can be useful in tasks such as object recognition in different lighting conditions.

In summary, different color spaces are used in computer vision and image processing to better reflect human perception, separate color and intensity information, achieve efficient compression and storage, and provide illumination invariance, among other reasons.

هر فضای رنگی ممکن است برای کاربردهای خاص مفید باشد. به طور کلی، هر فضای رنگی دارای مزایا و معایب خود است و بسته به نوع کاربرد و نوع تصاویری که مورد بررسی قرار می‌گیرند، فضای رنگی مناسبی برای استفاده انتخاب می‌شود.

برای مثال، فضای رنگی RGB برای نمایش رنگ‌های دقیق در تصاویر دیجیتالی و برای کاربردهایی مانند عکاسی دیجیتال و گیمینگ مناسب است. اما برای بسیاری از کاربردهای پردازش تصویر، این فضای رنگی دارای مشکلاتی است که می‌توان با استفاده از فضاهای رنگی دیگر بهبود بخشید.

به طور مثال، فضای رنگی HSV برای تغییرات رنگی مانند تنظیم روشنایی و سایه‌ها برای یک تصویر مفید است. همچنین، فضای رنگی YCbCr برای کاربردهایی مانند فشرده‌سازی تصویر و کاهش نویز به خصوص در ویدئو مناسب است.

به طور خلاصه، استفاده از چندین فضای رنگی به ما کمک می‌کند تا برای هر کاربرد، فضای رنگی مناسب را انتخاب کرده و عملکرد بهتری را در پردازش تصاویر داشته باشیم.

استفاده از چندین فضای رنگی به دلایل مختلفی انجام می شود:

۱. نمایش بهتر رنگ‌ها: هر فضای رنگی ویژگی‌های مختلفی را در نمایش رنگ دارد. به عنوان مثال، فضای رنگی HSV برای نمایش رنگ‌ها از شباهت با تجربه‌ی انسانی در توصیف رنگ‌ها بهره می‌برد، در حالی که فضای رنگی CMYK به عنوان فضای رنگی استفاده شده در چاپ، رنگ‌هایی را که در RGB نمی‌توان نمایش داد، از جمله رنگ‌های سفید و صورتی، می‌تواند نمایش دهد.

۲. پردازش سریع‌تر تصاویر: برخی فضاهای رنگی برای پردازش سریع تصاویر بهتر عمل می‌کنند. به عنوان مثال، در فضای رنگیYCbCr ، کانال Y معادل با شدت نور تصویر است که بسیار مهم برای پردازش سریع تصاویر است.

۳. استفاده در کاربردهای مختلف: برخی فضاهای رنگی مخصوص کاربردهای خاصی هستند. به عنوان مثال، فضای رنگی CMYK برای چاپ کاربرد دارد و فضای رنگی YCbCr برای فشرده‌سازی تصاویر به کار می‌رود.

۴. تبدیل مابین فضاهای رنگی: استفاده از چندین فضای رنگی امکان تبدیل مابین فضاهای رنگی را فراهم می‌کند. این قابلیت به ما اجازه می‌دهد تا با توجه به نیاز خود، تصاویر را به فضای رنگی مناسب تبدیل کنیم و از این طریق، پردازش را سریع‌تر و دقیق‌تر انجام دهیم.

از چندین فضای رنگی مانندRGB ، CMYK، HSV، YCbCrو غیره برای نشان دادن رنگ‌ها در مختلف برنامه‌های کاربردی استفاده می‌شود. هر فضای رنگی دارای ویژگی‌های منحصر به فردی است که آن را برای کاربردهای خاص مناسب می‌کند. به عنوان مثال، فضای رنگی RGB برای نمایش رنگ‌ها در صفحات وب و دیگر برنامه‌های مرتبط با تصویر مناسب است، در حالی که فضای رنگی CMYK برای چاپ و محصولات چاپی استفاده می‌شود. همچنین، فضای رنگی HSV برای تصویرسازی و تشخیص اشیاء در تصاویر و فیلم‌ها استفاده می‌شود. به طور کلی، استفاده از چندین فضای رنگی به ما اجازه می‌دهد تا رنگ‌ها را به شیوه‌های مختلف نمایش دهیم و از آنها در برنامه‌های مختلف بهره ببریم.

There are a number of reasons why there are so many color spaces:

1. Color resolution - sRGB and similar color spaces only cover a portion of the visible spectrum. This means that they can be represented with fewer bits per color channel. (8 in the case of sRGB.) For something like Rec. 2020 or ACEScg, you need 16 or more bits per channel. If you don't have enough bits for each color channel, you'll start to notice banding in displays of gradations of color in the color space. If your hardware is limited, you'll use one that requires fewer resources, whereas if output quality is of the utmost importance, you'll use one that can represent more colors.
2. Output Devices - televisions and until recently computer monitors are only able to display a portion of the visible spectrum. (Likewise with printers.) If the output device can't display it, what's the point of being able to store and transmit it? Color spaces used for televisions and earlier computers were chosen based on what they

could produce. Recently we've started to see wide gamut and HDR displays that can display a much larger portion of the visible spectrum (but still not all of it), so we're starting to see new color spaces for these devices.

3. Proprietary Needs - various vendors have created color spaces that match the devices and software they sell. The vast majority of photographers and videographers are home users doing it for fun. They don't need high bit depths and wide color gamuts. But professionals do!
4. Color spaces can also be designed to have certain properties that others don't. For example, some color spaces separate the colors based on perceptual uniformity (colors that appear similar to humans are grouped together), whereas other color spaces separate the color based on display hardware (RGB color spaces are generally used for displays that have red, green, and blue sub-pixels). Depending on what you're using the colors for, you choose a color space that makes working with colors in that way easy.

Sources:
https://video.stackexchange.com/questions/22419/why-are-there-so-many-color-spaces-available
https://chat.openai.com/

سوال ۲

این کد برای ادغام چند تصویر با هم و ساخت یک تصویر بزرگتر (panorama) استفاده می‌شود .ابتدا، تمام تصاویر در پوشه "images/Q2/" با استفاده از تابع glob.glob در یک لیست قرار می‌گیرند . سپس با استفاده از تابعcv2.imread ، تصاویر از طریق شیء imgs بازیابی می‌شوند .سپس با استفاده از تابع cv2.Stitcher.create() یک شی Stitcher ایجاد می‌شود. سپس با استفاده از تابع stitch() در این شیStitcher ، تصاویر درون لیست imgs با هم ادغام می‌شوند تا یک تصویر بزرگتر به نام resultایجاد شود. در صورت موفقیت‌آمیز بودن ادغام، تصویر result با استفاده از تابع cv2.imwriteدر مسیر "images/Q2/output/output.jpg" ذخیره می‌شود. در صورتی که ادغام انجام نشد، پیام "Error during stitching" چاپ می‌شود. در نهایت تصویر ساخته شده با استفاده از تابع cv2.imshow در پنجره نمایش داده می‌شود.
توضیحات کامل تر در لینکی که صورت سوال داده است آمده است صرفا در اینجا خلاصه ای از آن را بیان کرده ایم.

سوال ۳

الف)

این کد تابعی به نام **put_mask** دارد که تصویر صورت و تصویر ماسک را به عنوان ورودی دریافت کرده و ماسک را روی تصویر صورت قرار می دهد.

در این تابع، با استفاده از کتابخانه dlib ، ابتدا موقعیت چهره در تصویر پیدا می شود. سپس با استفاده از پیش بینی کننده شکل dlib ، نقاط مهم صورت مانند نقطه بینی ، چانه ، چپ و راست صورت شناسایی می شوند.

در مرحله بعد، با استفاده از نقاط شناسایی شده، تبدیل Perspective بین تصویر ماسک و تصویر صورت بدست می آید و سپس تصویر ماسک روی تصویر صورت اعمال می شود.

در نهایت، دو تصویر (تصویر اصلی صورت و تصویر صورت با ماسک) به همراه عنوان مناسب با استفاده از تابع **plotter** به نمایش در می آیند.
برای پیدا کردن نقاط ماسک از برنامه paint و با استفاده از سعی و خطا استفاده کردیم. برای نقاط صورت هم از روی نمودار ۶۸ نقططه مدلی که استفاده کردیم و سعی و خطا به نقاط مطلوب رسیدیم.

- در مرحله اول، تصویر چهره ورودی کپی شده و در متغیر result ذخیره می شود.
- در مرحله دوم، با استفاده از کتابخانهdlib ، چهره در تصویر محلی سفید پیدا می شود. برای این کار از تابع get_frontal_face_detector استفاده می شود و نتیجه آن در متغیر faces ذخیره می شود.
- در مرحله سوم، با استفاده از کتابخانهdlib ، نقاط کلیدی ساختار صورت بر روی چهره پیدا می شود. برای این کار از تابع shape_predictor استفاده می شود و نتیجه آن در متغیر landmarksذخیره می شود.
- سپس در این قسمت از کد، نقاط چپ چشم (left_point) ، نقطه بینی (nose_point) ، نقطه چانه (chin_point) و نقطه راست چشم (right_point) در نظر گرفته شده و مختصات آنها به ترتیب در متغیر هایleft ، nose، rightو chin ذخیره می شود.
- در مرحله چهارم، نقاط src_points و dest_points به ترتیب نقاط گوشه‌های مربعی هستند که تصویر ماسک باید روی آن قرار گیرد و نقاط متناظر با این گوشه‌ها در تصویر چهره پیدا می‌شود.
- در مرحله پنجم، با استفاده از تابعgetPerspectiveTransform ، ماتریس تبدیل بین نقاط src_pointsو dest_points پیدا شده و در متغیر transformation_matrix ذخیره می شود.

- در مرحله ششم، تصویر ماسک به اندازه تصویر چهره با استفاده از تابع warpPerspective با استفاده از ماتریس تبدیلtransformation_matrix ، تغییر شکل داده می شود و نتیجه در متغیر mask_warped ذخیره می شود.
- در مرحله آخر، تصاویر چهره و ماسک با هم ترکیب می‌شوند. برای این کار از تابع **()np.where**استفاده می‌شود. این تابع بررسی می‌کند که هر پیکسل در تصویر ماسک مقدار صفر دارد یا نه؛ اگر مقدار آن صفر باشد، به جای آن پیکسل، پیکسل متناظر در تصویر چهره قرار می‌گیرد؛ در غیر این صورت، پیکسل متناظر در تصویر ماسک استفاده می‌شود. این کار با استفاده از عبارت **mask_warped == [0, 0, 0]**انجام می‌شود که بررسی می‌کند که مقدار هر پیکسل در تصویر ماسک برابر با [۰, ۰, ۰] است یا نه. سپس با استفاده از تابع **()np.where** مقدار پیکسل‌ها جایگزین شده و تصویر نهایی به عنوان خروجی تابع بازگردانده می‌شود.

ب)

تشخیص نقطه عطف چهره یک تسک بینایی کامپیوتری است که در آن می‌خواهیم نقاط کلیدی صورت انسان را شناسایی و ردیابی کنیم. این وظیفه برای بسیاری از مشکلات کاربرد دارد.

به عنوان مثال، ما می توانیم از نقاط کلیدی برای تشخیص موقعیت و چرخش سر انسان استفاده کنیم. با آن، می‌توانیم ردیابی کنیم که آیا راننده توجه دارد یا خیر. همچنین، می‌توانیم از نکات کلیدی برای استفاده آسان‌تر از واقعیت افزوده استفاده کنیم. و راه حل های زیادی وجود دارد که می توانیم بر اساس این کار ایجاد کنیم.

خوشبختانه، لازم نیست مفاهیم تشخیص نقطه عطف چهره را با جزئیات درک کنیم. ما می توانیم از کتابخانه های از پیش ساخته شده مانند dlib، OpenCV و mediapipe استفاده کنیم.

Dlib کتابخانه ای برای استفاده از راه حل های یادگیری ماشین و بینایی کامپیوتر است. این کتابخانه بر اساس زبان C++ است، اما می توانیم از زبانی مانند پایتون برای استفاده از کتابخانه استفاده کنیم. یکی از راه حل هایی که با استفاده از این کتابخانه می توانیم اعمال کنیم، تشخیص چهره نقطه عطف است.

Mediapipe ابزاری برای پیاده سازی راه حل های بینایی کامپیوتری مبتنی بر ML است. این ابزار توسط گوگل ساخته شده است. این ابزار شامل انواع راه حل های بینایی کامپیوتری مانند تشخیص چهره، تخمین وضعیت، تشخیص اشیا و بسیاری موارد دیگر است. مزیت این کتابخانه این است که می توانید راه حل ها را در بسیاری از پلتفرم ها مانند وب، موبایل، رایانه شخصی و بسیاری دیگر اعمال کنید.

Face landmark detectionیکی از فعالیت‌های پرکاربرد در حوزه بینایی کامپیوتر است که در آن به دنبال یافتن نقاط کلیدی در چهره افراد می‌گردیم. این نقاط شامل بخش‌های مختلف چهره مانند چشم، بینی، دهان و... می‌باشند و می‌تواند به عنوان یک ورودی مهم برای بسیاری از برنامه‌های کاربردی مانند

تشخیص چهره، تشخیص احساسات و ارائه تغییرات جالب در تصویر (مانند تغییر اندازه، ستون‌کردن و جابجایی) استفاده شود.

برای دستیابی به نقاط کلیدی چهره، الگوریتم‌های متفاوتی وجود دارند. یکی از معروف‌ترین روش‌های استفاده شده در این حوزه، استفاده از شبکه‌های عصبی عمیق و یادگیری عمیق (Deep Learning) است که در آن با آموزش مدل‌های عمیق بر روی داده‌های مجموعه‌های بزرگ چهره، می‌توان به دقت و صحت بالاتری در تشخیص نقاط کلیدی چهره دست یافت.

در این روش‌ها، شبکه‌های عصبی عمیق به عنوان یک تابع تصویری کار می‌کنند و در ورودی شبکه، تصویر چهره به عنوان ورودی قرار می‌گیرد. سپس در لایه‌های شبکه‌های عمیق، فیلترهای مختلفی برای شناسایی ویژگی‌های مختلف چهره از جمله چشم، بینی، دهان و ... قرار داده می‌شود. پس از آن، با استفاده از شبکه‌های عصبی، نقاط کلیدی چهره استخراج شده و در خروجی الگوریتم قرار می‌گیرد.

با استفاده از تشخیص لندمارک(نطقه عطف و برجسته) چهره، می‌توان بسیاری از کاربردهای مختلفی را در بینایی کامپیوتر داشته باشیم، از جمله:

۱. تشخیص عیوب چهره: با استفاده از لندمارک چهره می‌توان عیوب چهره مانند تراشیدگی‌های صورت، چین و چروک‌ها و ... را شناسایی کرد و از طریق پردازش تصویر برای رفع این عیوب استفاده کرد.

۲. تشخیص احساسات: احساسات افراد را می‌توان با تحلیل لندمارک چهره تشخیص داد. به عنوان مثال، با تشخیص موقعیت لب‌ها و چشم‌ها، می‌توان از روی چهره فرد، احساسات مختلفی مانند خوشحالی، ناراحتی و ... را تشخیص داد.

۳. تشخیص جنسیت: با استفاده از لندمارک چهره می‌توان جنسیت فرد را تشخیص داد.

۴. تشخیص سن: می‌توان با استفاده از تحلیل لندمارک چهره، سن فرد را تخمین زد.

۵. تشخیص هویت: لندمارک چهره به عنوان اطلاعاتی که می‌توان از یک چهره بدست آورد، می‌تواند در تشخیص هویت فرد نقش داشته باشد.

۶. تشخیص تبعیت از قانون: با استفاده از لندمارک چهره، می‌توان تبعیت از قانون را نیز تشخیص داد.

با توجه به کاربردهای مختلفی که تشخیص لندمارک چهره دارد، این تکنیک در بسیاری از برنامه‌های بینایی کامپیوتر مانند تشخیص چهره، رباتیک، تشخیص حرکت، بازیابی اطلاعات، مدیریت

Face landmark detection is a computer vision task that involves detecting and localizing key points on a face, such as the eyes, nose, mouth, and chin. It is a fundamental task in many applications, including facial expression analysis, face recognition, and virtual reality.

One of the popular solutions for face landmark detection is the Dlib library, which provides an implementation of the Constrained Local Model (CLM) algorithm. The CLM algorithm is a model-based approach that uses a combination of local feature extraction and machine learning techniques to detect facial landmarks. It works by constructing a statistical model of the face, which includes a set of landmarks and their corresponding appearance variations. During inference, the algorithm searches for the best fit of the model to the input image by iteratively optimizing the landmark positions.

The Dlib library provides an easy-to-use API for performing face landmark detection, which includes a pre-trained facial landmark detector that can detect 68 facial landmarks. The detected landmarks can be used for various applications, such as face alignment, face recognition, and facial expression analysis.

In summary, face landmark detection is a crucial task in computer vision, and the Dlib library is one of the available solutions that provide an efficient and accurate implementation of the CLM algorithm for detecting facial landmarks.

Sources:
https://chat.openai.com/
https://towardsdatascience.com/face-landmark-detection-using-python-1964cb620837#:~:text=Face%20landmark%20detection%20is%20a,is%20paying%20attention%20or%20not.

سوال ۴

الف)

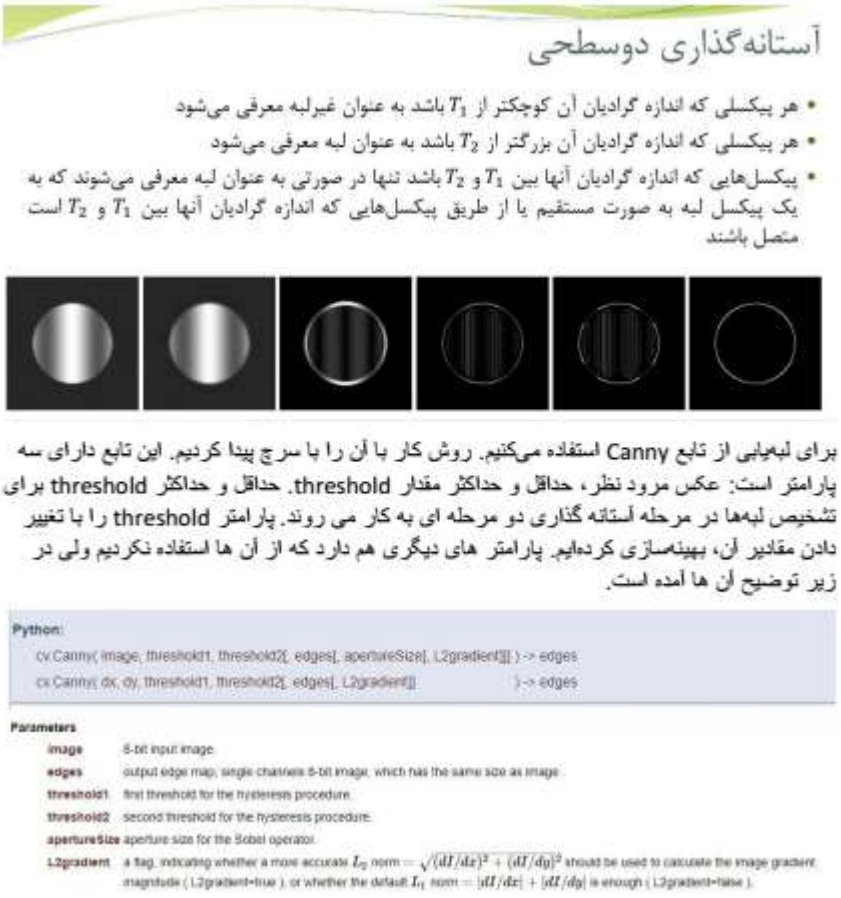
بخش الف :ابتدا باید تصویر را به گری‌اسکیل تبدیل کرد تا پردازش روی آن سریع‌تر و با دقت بیشتری صورت بگیرد. برای نویزگیری تصویر از فیلتر گوسی با اندازه کرنل ۵ استفاده می‌کنیم. سپس با استفاده از الگوریتمCanny ، لبه‌های تصویر را پیدا می‌کنیم. پارامترهای این الگوریتم به شکل زیر هستند:

```
im_blurred = cv2.GaussianBlur(im_grayscale, (5, 5), 0)

upper_trheshold = 100
lower_threshold = 200
edges = cv2.Canny(im_blurred, lower_threshold, upper_trheshold)
```

```
imshow(edges)
```

با انتخاب این پارامترها، می‌توانیم لبه‌های تصویر را با دقت و کیفیت بالا پیدا کنیم.



ب)

برای پیدا کردن کاغذ، از تابع **findContours** در OpenCV استفاده می‌کنیم. این تابع با گرفتن تصویری به عنوان ورودی، لیستی از کانتورها (یا خطوطی که مرز شیء را تشکیل می‌دهند) را برمی‌گرداند. در این حالت، کاغذ ما یک کانتور بسته است که در لیست برگشتی تابع **findContours** قرار دارد. برای پیدا کردن مرز کاغذ، از تابع **drawContours** استفاده می‌کنیم.

برای پیدا کردن مرز کاغذ و مشخص کردن آن در تصویر، ابتدا باید کانتورهای موجود در تصویر را با استفاده از تابع **findContours** پیدا کنیم. سپس بین تمام کانتورها، کانتوری را که بیشترین مساحت(تصویر مطلوب) را به خود اختصاص می‌دهد را انتخاب کرده و مرز کاغذ را با استفاده از آن مشخص کنیم.

توضیحات کد:

- ابتدا با استفاده از تابع **findContours**کانتورهای موجود در تصویر **edges**پیدا می‌شوند و در متغیر **contours**ذخیره می‌شوند.
- با استفاده از تابع **max**، کانتوری که بیشترین مساحت را به خود اختصاص می‌دهد را از بین تمام کانتورهای موجود انتخاب می‌کنیم و در متغیر **max_contour**ذخیره می‌کنیم.
- سپس با استفاده از تابع **drawContours**، کانتور **max_contour**را روی تصویر **image** رسم می‌کنیم.

توضیحات پارامترهای تابع :**findContours**

- :**cv2.RETR_EXTERNAL**نوع روشی که برای پیدا کردن کانتورها استفاده می‌شود. در این حالت، فقط کانتورهایی که به بیرون ترسیم شده‌اند (مثل مرز کاغذ) شناسایی می‌شوند.
- :**cv2.CHAIN_APPROX_SIMPLE**روشی که برای تقریب خطوط کانتور استفاده می‌شود. در این حالت، تنها نقطه اول و آخر هر خط باقی می‌مانند و نقاط وسط خطها حذف می‌شوند.

ج)

نقاط مبدا را از گوشه های بزرگترین کانتور(بعد از اعمال approxPolyDP) پیدا میکنیم و نقاط مقصد را با فرض اینکه کاغذ A4 است قرار می دهیم.

در این بخش با استفاده از تابع **warpPerspective**تصویر را به صورت خطی در بیاوریم تا زمینه‌ی اضافی حذف شود.

ابتدا نیاز است که ماتریس تبدیلی را به دست آوریم. برای این کار، از توابع **getPerspectiveTransform**یا **findHomography**در OpenCV استفاده می‌کنیم که با استفاده از چهار نقطه‌ی ورودی (با فرض این که متون کاغذ مستطیل هستند) و چهار نقطه‌ی خروجی، ماتریس تبدیلی را به دست می‌آورند. برای محاسبه‌ی نقاط ورودی و خروجی می‌توانید از تصویر اولیه و تصویر خروجی یا اندازه‌ی مورد نظر برای کاغذ استفاده کنید.

سپس با استفاده از تابع **warpPerspective**تصویر را به صورت خطی در بیاوریم.

د)

روش هایی که تا کنکون آموخته ایم در زیر آمده اند میتوانیم هر یک را پیاده سازی یا از کتابخانه opencv استفاده و اجرا کنیم و نتیجه را مقایسه کنیم.

- histogram clipping and stretching
- histogram equalization
- histogram matching
- ACE (Adaptive contrast enhancement)
- AHE (Adaptive histogram equalization)

- CLAHE (contrast limited AHE)
- smoothing filters (AVG, gaussian, median, biliteral)
- LPF, HPF
- Sharpening filters (Gradient, Laplacian) (Prewitt, Sobel)
- LPF, HPF in frequency domain (Fourier Transform)
- filter magnitude spectrum in frequency domain

روش های زیر را پیاده سازی کردیم:

```
# using otsu thresholding
ret, im_gray_result_1 = cv2.threshold(im_gray_result, 0, 255,
cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# smooth the image with bilateralFilter
im_gray_result_2 = cv2.bilateralFilter(im_gray_result, 31, 40, 40)

# # smooth the image with GaussianBlur
im_gray_result_3 = cv2.GaussianBlur(im_gray_result, (5, 5), 0)

# # smooth the image with medianBlur
im_gray_result_4 = cv2.medianBlur(im_gray_result, 101)

# using histogram stretching
im_gray_result_5 = cv2.normalize(im_gray_result, None, 0, 255, cv2.NORM_MINMAX)
# imshow(im_gray_result_5)

# using histogram equalization
im_gray_result_6 = cv2.equalizeHist(im_gray_result)
# imshow(im_gray_result_6)

# using adaptiveThreshold
im_gray_result_7 = cv2.adaptiveThreshold(im_gray_result, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 31, 6)
```

از بین روش های زیر که آن ها را اعمال کردیم ترکیب CLAHE,adaptiveThreshold,medianBlur عملکرد بهتری را در نمونه ما داشت.

```
# better the quality of im_gray_result with CLAHE and adaptiveThreshold (otsu)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))

im_gray_result3 = cv2.adaptiveThreshold(im_gray_result1, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 31, 6)
```

```
im_gray_result4 = cv2.medianBlur(im_gray_result3, 11)
```

برای تصاویر رنگی نیز ابتدا به فضای HSV برده و سپس CLAHE, AdaptiveThreshold زدیم و بعد از آن biliteral.

سوال ۵

الف) با توجه به اسلاید های CV_13 (صفحه ۳۱ به بعد) و CV_14 (صفحه ۴ تا ۷) به توضیح آن می پردازیم.و همچنین از https://en.wikipedia.org/wiki/Harris_corner_detector کمک میگیریم.

الگوریتم هریس (Harris) یکی از الگوریتم‌های آشکارسازی نقاط کلیدی (Key Points) در تصاویر است. این الگوریتم در سال ۱۹۸۸ توسط کریس هریس و مایکل استفن (Chris Harris and Mike Stephens) ارائه شده است. هدف این الگوریتم پیدا کردن نقاطی در تصویر است که تغییر شدیدی در آنها به وجود می‌آید. این نقاط به عنوان نقاط کلیدی شناخته شده و در الگوریتم‌هایی مانند SIFT، SURF و ORB برای تشخیص و تطبیق تصاویر استفاده می‌شوند.

الگوریتم هریس (Harris)، یک الگوریتم آشکار ساز گوشه (corner detector) برای تشخیص گوشه ها در تصاویر دیجیتالی است. این الگوریتم برای بسیاری از کاربردها از جمله پردازش تصویر، بینایی ماشین، رباتیک و ... مورد استفاده قرار می گیرد.

Harris corner detector

The **Harris corner detector** is a corner detection operator that is commonly used in computer vision algorithms to extract corners and infer features of an image. It was first introduced by Chris Harris and Mike Stephens in 1988 upon the improvement of Moravec's corner detector.[1] Compared to its predecessor, Harris' corner detector takes the differential of the corner score into account with reference to direction directly, instead of using shifting patches for every 45 degree angles, and has been proved to be more accurate in distinguishing between edges and corners.[2] Since then, it has been improved and adopted in many algorithms to preprocess images for subsequent applications.

Introduction

A corner is a point whose local neighborhood stands in two dominant and different edge directions. In other words, a corner can be interpreted as the junction of two edges, where an edge is a sudden change in image brightness.[3] Corners are the important features in the image, and they are generally termed as interest points which are invariant to translation, rotation and illumination. Although corners are only a small percentage of the image, they contain the most important features in restoring image information, and they can be used to minimize the amount of processed data for motion tracking, image stitching, building 2D mosaics, stereo vision, image representation and other related computer vision areas.

In order to capture the corners from the image, researchers have proposed many different corner detectors including the Kanade-Lucas-Tomasi (KLT) operator and the Harris operator which are most simple, efficient and reliable for use in corner detection. These two popular methodologies are both closely associated with and based on the local structure matrix. Compared to the Kanade-Lucas-Tomasi corner detector, the Harris corner detector provides good repeatability under changing illumination and rotation, and therefore, it is more often used in stereo matching and image

database retrieval. Although there still exists drawbacks and limitations, the Harris corner detector is still an important and fundamental technique for many computer vision applications.

Process of Harris corner detection algorithm

Commonly, Harris corner detector algorithm can be divided into five steps.

1. Color to grayscale
2. Spatial derivative calculation
3. Structure tensor setup
4. Harris response calculation
5. Non-maximum suppression

Color to grayscale

If we use Harris corner detector in a color image, the first step is to convert it into a grayscale image, which will enhance the processing speed.

The value of the gray scale pixel can be computed as a weighted sums of the values R, B and G of the color image.

Spatial derivative calculation

Next, we are going to find the derivative with respect to x and the derivative with respect to y, Ix(x,y) and Iy(x,y).

Structure tensor setup

With Ix(x,y) and Iy(x,y), we can construct the structure tensor M.

Harris response calculation

For $x \ll y$, one has $\frac{x \cdot y}{x+y} = x\frac{1}{1+x/y} \approx x$. In this step, we compute the smallest eigenvalue of the structure tensor using that approximation:

$$\lambda_{\min} \approx \frac{\lambda_1 \lambda_2}{(\lambda_1 + \lambda_2)} = \frac{\det(M)}{\operatorname{tr}(M)}$$

with the trace $\operatorname{tr}(M) = m_{11} + m_{22}$.

Another commonly used Harris response calculation is shown as below,

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 = \det(M) - k\operatorname{tr}(M)^2$$

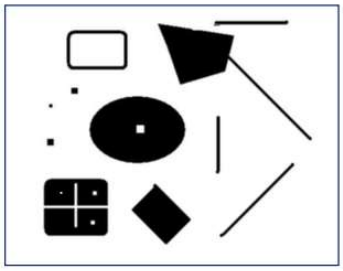
where k is an empirically determined constant; $k \in [0.04, 0.06]$.

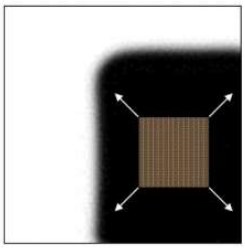
Non-maximum suppression

In order to pick up the optimal values to indicate corners, we find the local maxima as corners within the window which is a 3 by 3 filter.

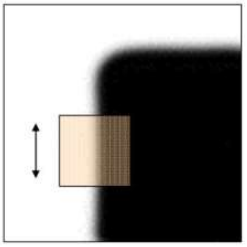
نقاط کلیدی

- چه نقاطی در تصویر زیر به سادگی قابل تشخیص و انطباق هستند؟
- گوشه‌ها تکرارپذیر و متمایز هستند

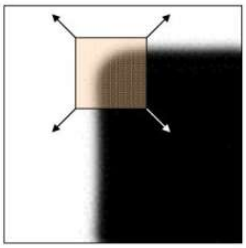




“flat” region:
no change in all directions



“edge”:
no change along the edge direction



“corner”:
significant change in all directions

آشکارساز Harris

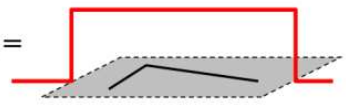
پنجره برای جستجوی محلی

- میزان اختلاف سطح روشنایی تصویر به ازای جابجایی (u,v)

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2$$

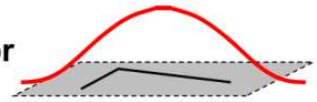
- که w یک پنجره برای تاثیر همسایه‌های محلّی است که می‌تواند مستطیلی یا گاوسی باشد
- گوشه نقطه‌ای است که تابع فوق در آن بزرگ باشد

$w(x,y) =$



1 in window, 0 outside

or



Gaussian

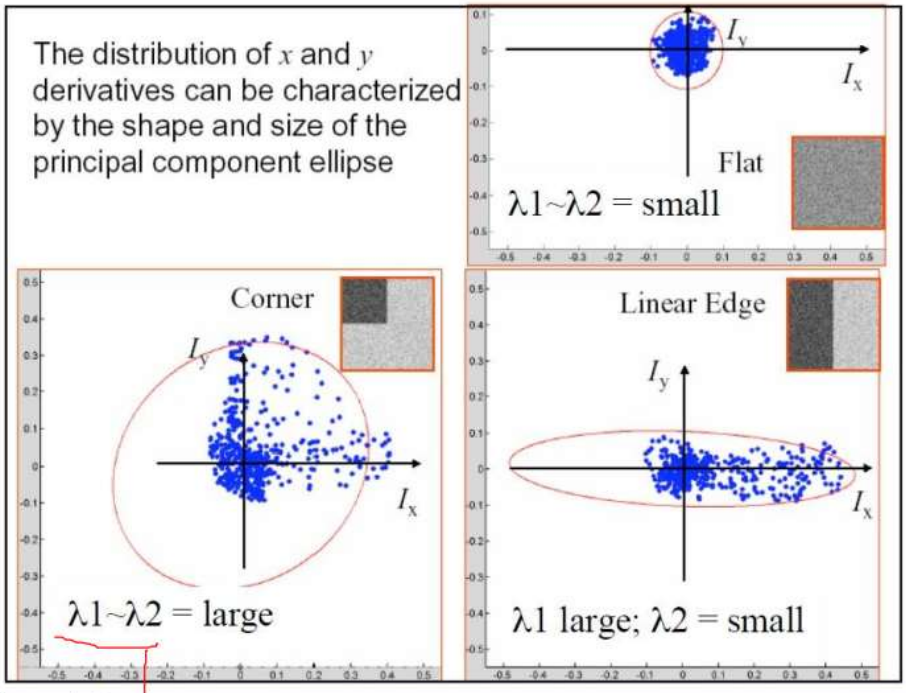
آشکارساز Harris

$$E(u,v) \approx [u \quad v]\left(\sum_{x,y} w(x,y)\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}\right)\begin{bmatrix} u \\ v \end{bmatrix}$$

$$M \triangleq \sum_{x,y} w(x,y)\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

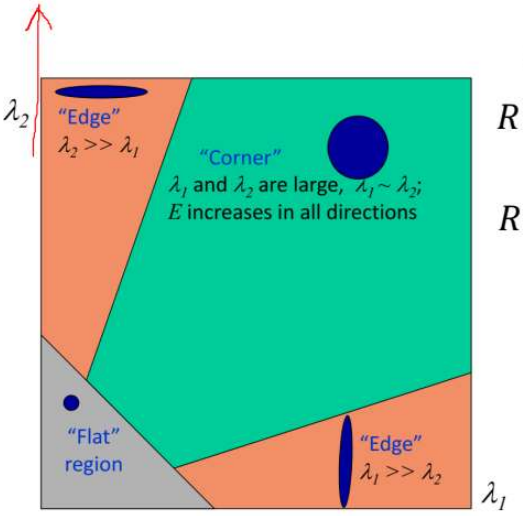
جبر خطی

- مقادیر ویژه یک ماتریس نشان می‌دهند که در یک راستا چه مقدار انرژی وجود دارد و بردارهای ویژه جهت آنها را مشخص می‌کنند



شعاع بزرگ و کوچک بیضی

آشکارساز Harris



- برای داشتن گوشه، نیاز است تا هر دو مقدار ویژه بزرگ باشند

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

هم ارز

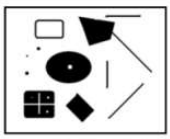
$$R = det(M) - k\big(trace(M)\big)^2$$

دترمینان — جمع قطر اصلی

$$M \triangleq \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

آشکارساز Harris

window: همسایگی محلی

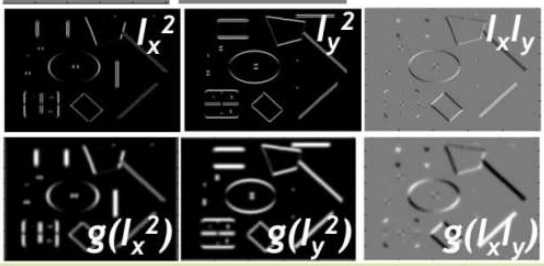


$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

$$M = g(\sigma_I) * \begin{bmatrix} I_x^2(\sigma_D) & I_xI_y(\sigma_D) \\ I_xI_y(\sigma_D) & I_y^2(\sigma_D) \end{bmatrix}$$

گاوسی

$$R = det(M) - k\big(trace(M)\big)^2$$

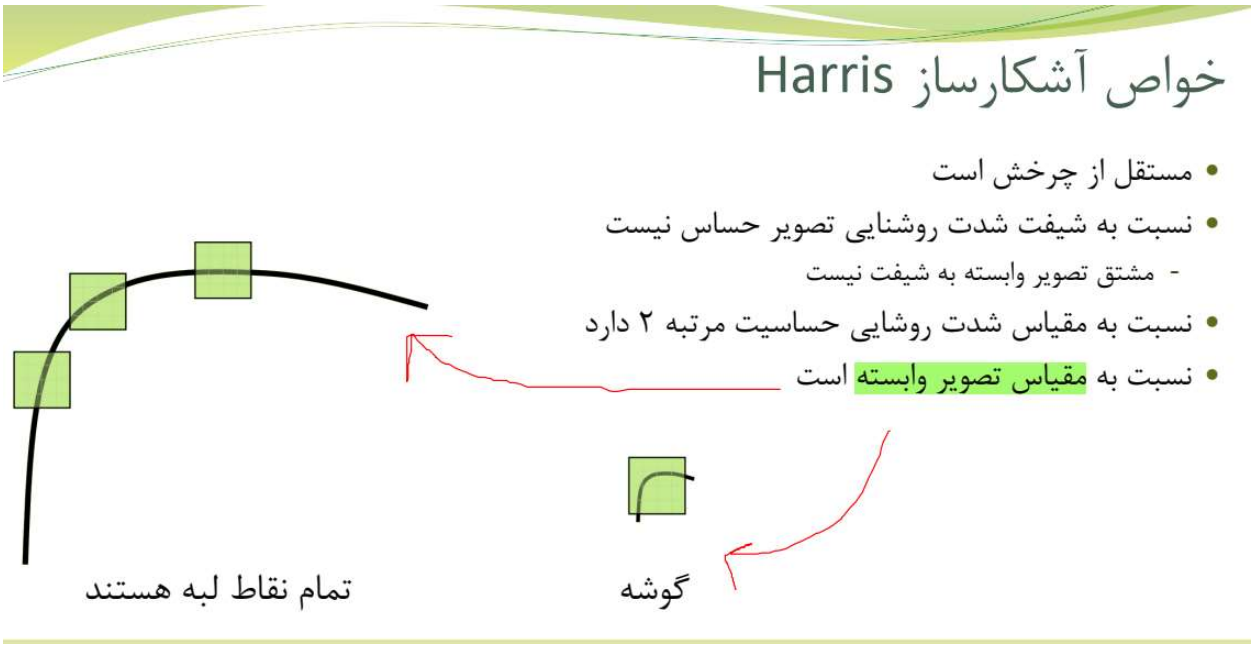


Iy Ix

1. محاسبه مشتق افقی و عمودی
- محاسبه مربع مشتق‌ها
- اعمال اثر پنجره w
- محاسبه مقادیر R
- حذف مقادیر غیر بیشینه

non-max suppression

اگر در یک همسایگی چندین نقطه کنار هم بودند بهترین و بزرگترین را انتخاب کنیم.



ب)

مطالب بالا را گام به گام پیاده سازی کرده ایم.
توضیحات کامل در کامنت های کد آمده است.این تابع با استفاده از الگوریتم Harris detection از نقاط گوشه‌ای تصویر ورودی استخراج می‌کند. این تابع ابتدا تصویر را به رنگ خاکستری تبدیل می‌کند، سپس با استفاده از فیلترSobel ، ماتریس‌های Ix و Iy را محاسبه کرده و سپس با استفاده از این دو ماتریس، ماتریس‌هایIx2 ، Iy2و Ixy را محاسبه می‌کند. در ادامه با استفاده از فیلتر Gaussian برای هرکدام از ماتریس‌هایIx2 ، Iy2وIxy ، یک نمایش دو بعدی از آن‌ها را بدست می‌آورد و با استفاده از آن‌ها ماتریس R را محاسبه می‌کند. در نهایت، با اعمال تکنیک non-maximum suppression و با تعیین یک آستانه threshold برای مقادیر ماتریسR ، از بین نقاط گوشه‌ای استخراج شده، نقاطی که دارای مقدار بیشتر از آستانه هستند، انتخاب و به عنوان نقاط گوشه‌ای بر روی تصویر ورودی نمایش داده می‌شوند.
در پیاده سازی با کتابخانه از لینک زیر استفاده کرده ایم و توضیحات آن در کامنت ها آمده است.

src: https://docs.opencv.org/3.4/dc/d0d/tutorial_py_features_harris.html

مقایسه نتایج: همان طور که دیده می شود خروجی هر دو تابع یکسان است و نتیجه مشابهی دارند.

سوال ۶

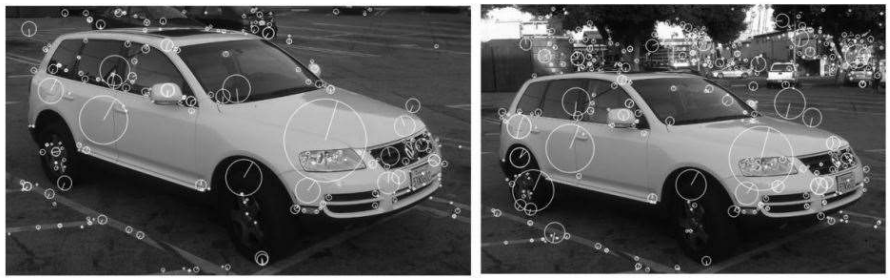
- **Scale-Invariant Feature Transform (SIFT)**
- **Speeded Up Robust Feature (SURF)**
- **Orientated FAST and Robust BRIEF (ORB)**.

SIFT was created in 2004 by D. Lowe in the University of British Columbia to solve the problem of scale variance for feature extraction.
SURF was created as an improvement on SIFT in 2006, aimed at **increasing the speed** of the algorithm.
ORB, which as the name suggests is the combination of two algorithms **FAST and BRIEF** and was created as an alternative to both SIFT and SURF in 2011. (FAST[key-point detector]: Features from Accelerated Segment Test, BRIEF[key-point descriptor]: Binary Robust Independent Elementary Features)

بر اساس آزمایش‌ها، ORB بهترین عملکرد را در سراسر صفحه با محاسبات سریع دارد و نشان داده شده است که در برابر روشنایی و تغییرات چرخشی قوی است. ممکن است شرایط دیگری وجود داشته باشد که در آن SIFT یا SURF مورد نیاز باشد، اما برای اکثر موارد استفاده به نظر می رسد که ORB بهترین روش استخراج ویژگی از این سه است.

نقاط کلیدی

- روش‌های پرکاربرد برای اسخراج نقاط کلیدی و توصیفگرهای آنها عبارتند از:
  - SIFT
  - SURF
  - ORB



https://mikhail-kennerley.medium.com/a-comparison-of-sift-surf-and-orb-on-opencv-59119b9ec3d0

SIFT، SURF و ORB همه ی سه روش برای استخراج نقاط کلیدی از تصاویر هستند. این روش‌ها از الگوریتم‌های تشدید شدگی، تبدیل ویژگی مشتق شده، تحلیل تکرار شوندگی پراکنده، معیار ارزیابی همپوشانی و تفاوت مقیاس استفاده می‌کنند. در زیر خلاصه‌ای از مقایسه‌ی این سه روش ارائه شده است:

- SIFT: این روش به روشی از تشدید شدگی مبتنی است. یکی از مزیت‌های این روش این است که نقاط کلیدی مستحکم و قابل اعتمادی تولید می‌کند. علاوه بر این، SIFT توانایی در تشخیص تصاویر با چرخش و تغییر مقیاس دارد. اما این روش برای تصاویر با نویز بالا کارایی پایینی دارد.
- SURF: مانند SIFT از تشدید شدگی استفاده می‌کند، اما با استفاده از تبدیل موجک برای مقیاس بندی و یافتن نقاط کلیدی، سرعت بیشتری دارد. SURF قابلیت شناسایی نقاط کلیدی در تصاویر با نویز بالا و تصاویر متحرک را دارد. این روش مقاومت به تغییرات شدت روشنایی را نیز دارد.

- ORB: یکی از مزایای این روش سرعت بالا و کارایی خوب در تصاویر با نویز بالا است. ORB از تحلیل تکرار شوندگی پراکنده برای یافتن نقاط کلیدی استفاده می‌کند. در این روش، از بیت‌های رندوم برای توصیف نقاط کلیدی استفاده می‌شود که به کاربر اجازه می‌دهد که در دو تصویر تطبیق‌پذیری را برای نقاط کلیدی پیدا کند.

نتایج آزمایش روی عکس های مذکور با توجه به نمودار های داده شده بر روی ۴ فاکتور:

۱. سرعت محاسبه
از جدول بالا می بینیم که SURF در واقع سریعتر از SIFT عمل می کند، هرچند با مقدار کمی (به ترتیب ۱۱۲٫۸ میلی ثانیه در مقابل ۱۱۶٫۲ میلی ثانیه)، در حالی که ORB یک مرتبه قدر سریعتر از هر دو در ۱۱٫۵ میلی ثانیه محاسبه می شود.

۲. تعداد کل نقاط کلیدی شناسایی شده
در اینجا می بینیم که ORB قادر است بیشترین تعداد نقاط کلیدی را در هر تصویر استخراج کند، تقریباً سه برابر SURF. این احتمالاً به این دلیل است که ORB به نقطه کلیدی نیاز ندارد که حداکثر مطلق محلی باشد، بلکه ماکزیمم/حداقل یک سری n پیوسته باشد.

۳. درصد نقاط کلیدی منطبق
برای تصویر روشن‌شده، می‌توانیم ببینیم که عملکرد SIFT و SURF با اختلاف کمتر از ۱٪ مشابه هستند و ORB در عملکرد در ۹۶٪ نقاط کلیدی مطابقت دارد.
برای تصویر چرخانده شده، ORB و SURF هر دو ۱۰۰٪ از نقاط کلیدی مطابقت دارند. SIFT توانست ۹۳٪ از نقاط کلیدی خود را مطابقت دهد که مشابه عملکرد آن برای تصویر روشن شده است.

۴. دریفت از نقاط کلیدی همسان
می‌توانیم ببینیم که SIFT در این تست‌ها در مقایسه با SURF و ORB ضعیف عمل می‌کند و میانگین دریفت ۲۰ و ۹۱ پیکسل برای تصاویر روشن‌تر و چرخانده شده است. SURF عملکرد خوبی دارد، ORB را برای تصویر چرخانده شده و با دریفت کمتر از ۱ پیکسل برای تصویر روشن شده مطابقت دارد. ORB در تصویر روشن شده ۰ دریفت و در تصویر چرخانده کمتر از ۲ پیکسل دریفت دارد.
عملکرد در همه روش ها با تغییر روشنایی در مقایسه با چرخش بهتر است.

پایان