به نام خدا

## گزارش تمرین سری سوم

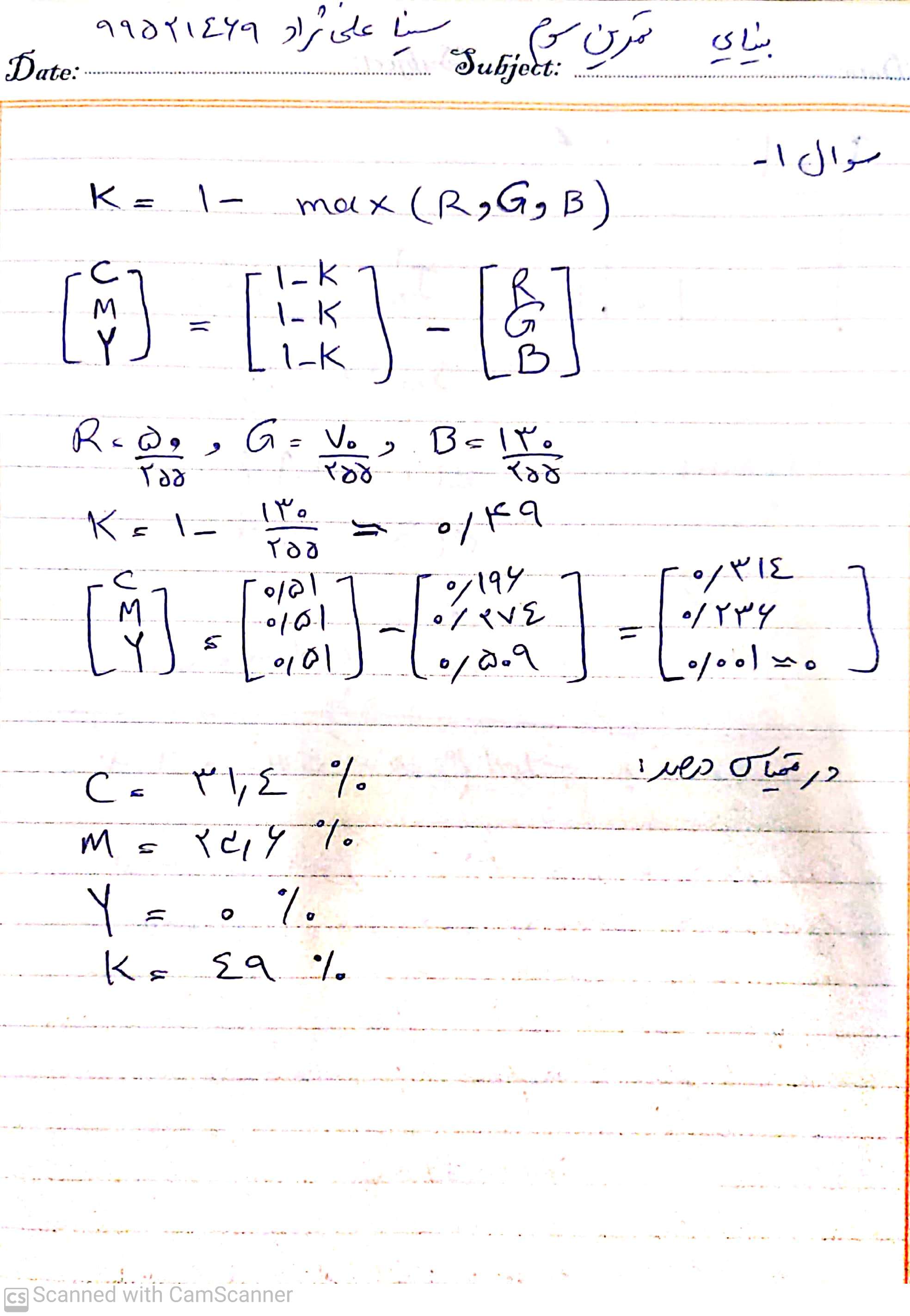
## درس مبانی بینایی کامپیوتر

## سینا علی‌نژاد

## 99521469

# سوال 1-

### الف)



### ب)

برای انتقال از فضای BGR به YCrCb از تابع آماده OpenCV استفاده کردم.



برای انتقال به فضای HSV از همان تابع به شکل زیر استفاده میکنیم.

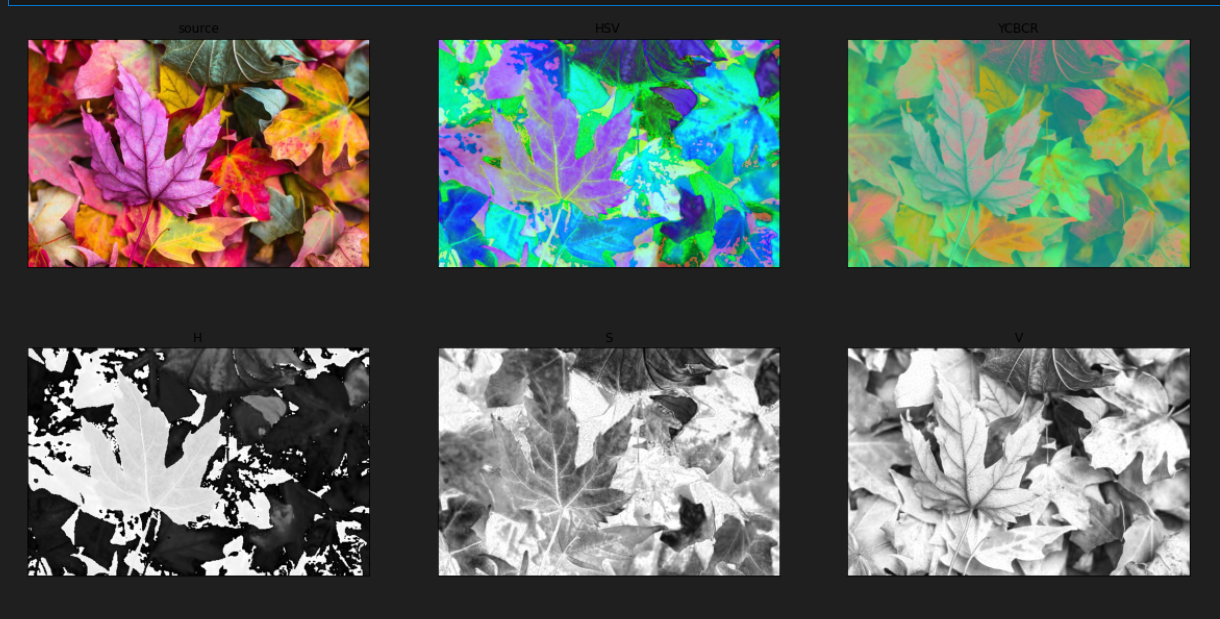


### ج)

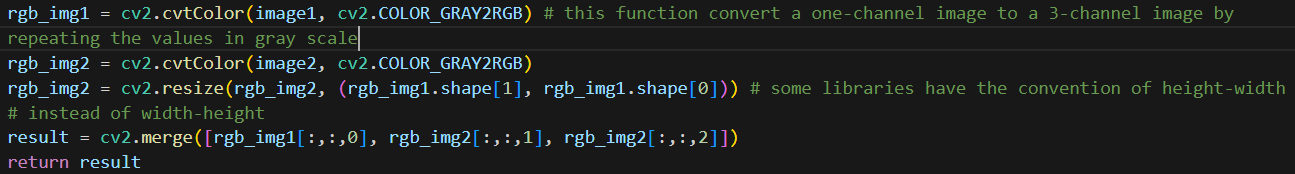
برای جدا کردن کانال‌های مختلف می‌توان از تابع cv2.split به شکل زیر استفاده کرد.



نتیجه نهایی:



د) برای بدست آوردن تفاوت دو تصویر میتوان ابتدا دو تصویر را به فضای رنگی خاکستری منتقل و سپس یک تصویر رنگی ساخت که دو کانال آن از یکی از عکسها و کانال دیگر از عکس دیگر باشد.



برای ترکیب این دو تصویر در یک تصویر رنگی از تابع cv2.merge استفاده کردم که برای هر کانال یک آرایه به عنوان ورودی می‌گیرد. چندبار به ارور خوردم و نهایتا با استفاده از تابع cv2.cvtColor اومدم عکس خاکستری رو به فضای رنگی بردم و این به این شکل کار میکنه که میاد همون مقادیر خاکستری رو تو هر سه کانال میذاره. حالا میتونستم توی تابع merge ازشون استفاده کنم. مشکل دیگری که پیش اومد این بود که تابع merge آرایه های هم‌سایز به عنوان ورودی میگیره و خب عکس اول و دوم یکم در سایز متفاوت بودن پس اومدم از تابع cv2.resize استفاده کردم و هم‌سایزشون کردم.

خروجی نهایی:



قسمت‌هایی که در عکس چپ موجود بودند ولی در عکس راست نبودند، به رنگ آبی و برای قسمتهای بالعکس این قضیه با رنگ قرمز مشخص شدند.

### ه)

دلایل مختلفی برای وجود چندین فضای رنگی وجود دارد.

1. ابزارهای مختلف ممکن است از فضاهای رنگی متفاوتی برای نشان دادن رنگ استفاده کنند.

برای مثال یک دوربین ممکن است از فضای رنگ RGB استفاده کند یا یک پرینتر از CMYK بهره ببرد.

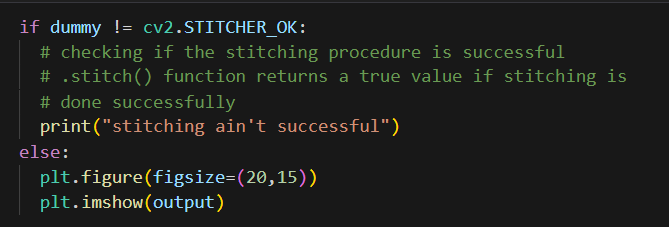
1. بعضی از فضاهای رنگی برای قابل فهم بودن برای انسان بوجود آمدند. مثالی از اونها HSL , HSV هستند. انسان یک رنگ اصلی را میفهمد که این رنگ شدت روشنایی متفاوتی میتواند داشته باشد. کاربرد این فضاهای رنگی در مواردی مانند image segmentation است که میخواهیم رنگهای مشابه را در یک گروه ببریم.
2. بهینگی در محاسبات یکی دیگر از دلایل وجود چندین فضای رنگی است. برای بعضی کاربردها کار کردن با برخی فضاهای رنگی بهینه‌تر است.

# سوال 2-

برای این سوال از پکیج Stitcher در کتابخانه OpenCV استفاده کردم. تابع stitch در این پکیج لیستی از عکسها را ورودی گرفته و یک عکس پانوراما خروجی می‌دهد.

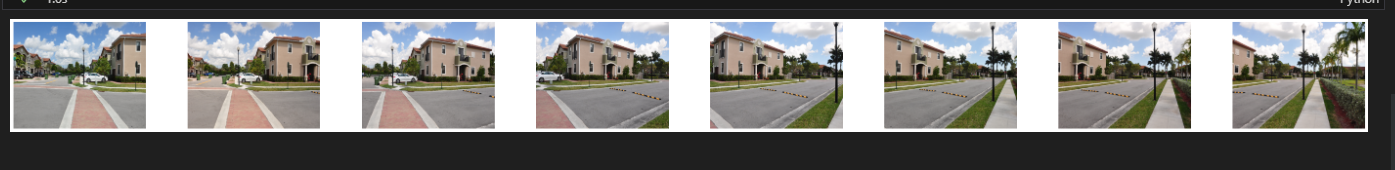
مراحل کار این تابع به شکل زیر است:

1. استخراج نقاط کلیدی و همچنین استخراج descriptor برای این نقاط. برای descriptor از همون SIFT Feature استفاده میکنه.
2. از توصیفگرها برای match کردن نقاط کلیدی استفاده میکنه.
3. با استفاده از الگوریتم Ransac ماتریس تبدیل رو تخمین میزنه.
4. از ماتریس بدست آمده برای تطبیق و چسباندن تصاویر به هم استفاده میکنه.



شرطی که گذاشتم برای اینه که اگر عکسهای ورودی نقاط کلیدی شون به هم match نشن، این تابع نمیتونه تبدیل رو انجام بده.

عکسهای اصلی:

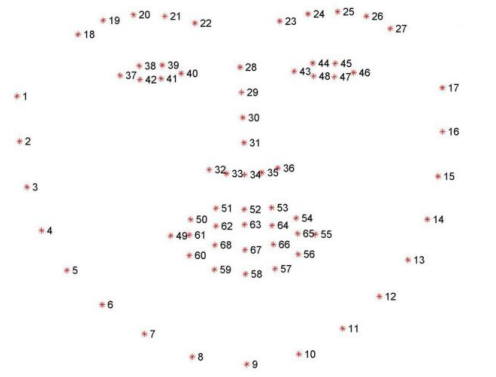


### عکس پانورامای نهایی:

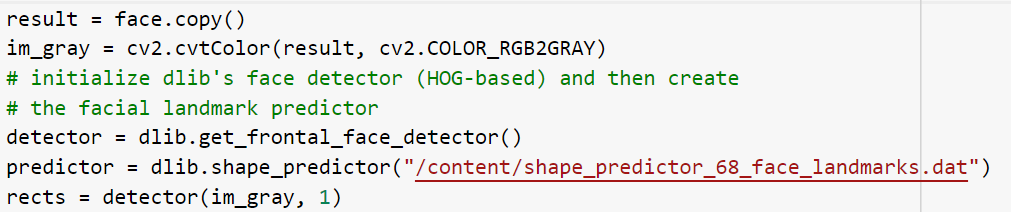


# سوال 3-

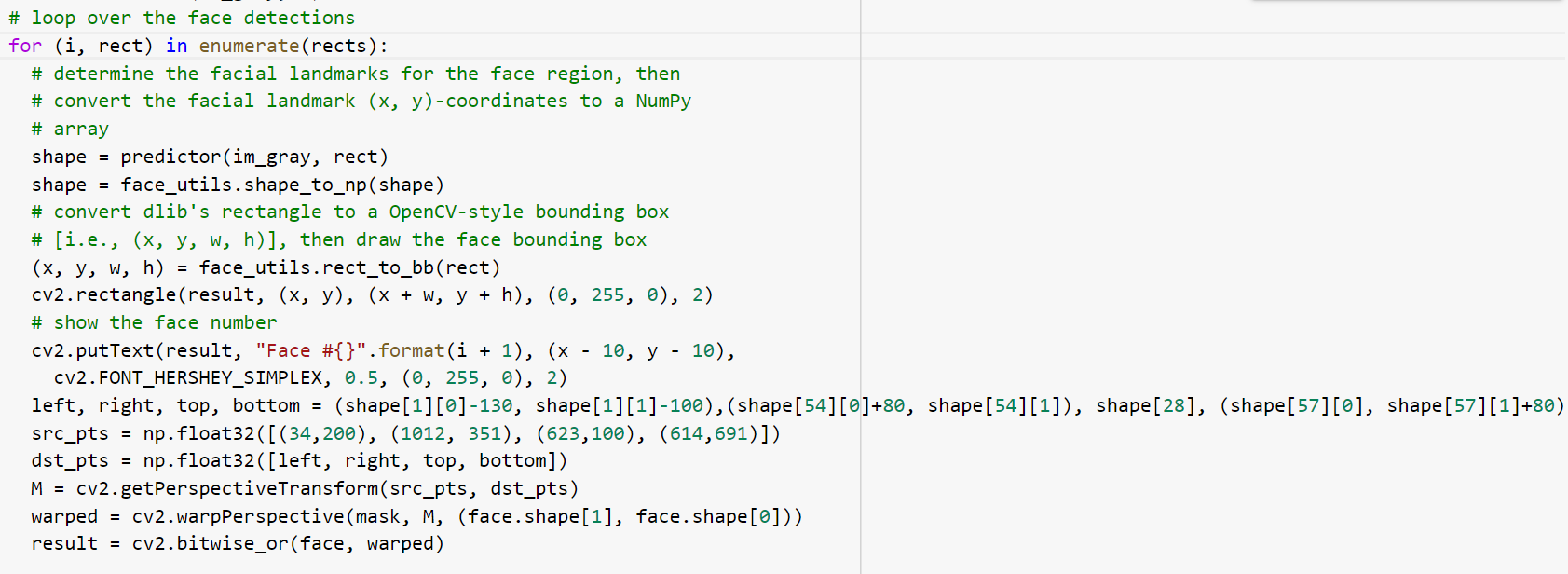
برای این سوال از کتابخانه dlib که نقاط کلیدی چهره رو مشخص میکنه استفاده کردم. ابتدا یک دیتاستی به این مدل باید داد تا train شود و سپس تصویر چهره انیشتین را به عنوان ورودی داده و 68 نقطه را طبق pattern زیر به ما می‌دهد.



حال باید تصویر ماسک را بر روی صورت انیشتین به درستی قرار دهیم، برای این کار از تبدیل perspective برای ماسک استفاده میکنیم و چهار نقطه بالا، پایین، چپ و راست ماسک را به نقاط مناسب که از face landmarks داریم منتقل میکنیم. پس ماتریس تبدیل از این چهار نقطه بوجود می‌آید.



این تکه از کد مربوط به ایجاد یک آبجکت face\_detector و train کردن آن به کمک یک دیتاست که در سلول قبلتر دانلود کردیم، است. در خط آخر از تابع detector برای تشخیص چهره‌های درون عکس استفاده می‌شود که اینجا ما تنها یک چهره داریم.

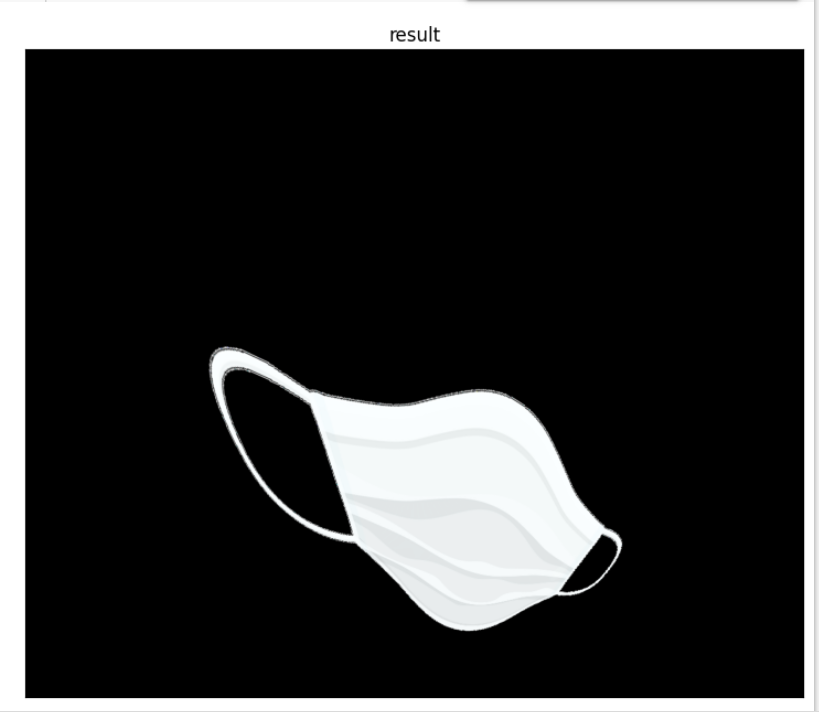


در قدم بعدی باید روی چهره‌های درون تصویر حلقه for زده و هر کدام از این چهره‌ها را به تابع predictor بدهیم تا face landmarks مربوط به آن چهره را به ما خروجی بدهد. سپس با استفاده از تابع shape\_to\_np آن را تبدیل به یک آرایه نامپای میکنیم. از تابع putText که بگذریم، حال نوبت آن است که نقاط متناظر را به تابع آماده getPerspectiveTransform بدهیم تا ماتریس تبدیل را برای ما بسازد.

نکته: برای نقاط متناظر خیلی به مشکل خوردم و جور در نمیومد و خب متوجه شدم اون نقاطی که face landmark تشخیص داده شده بود از اون نقاطی که من میخواستم بعضا فاصله داشت و مجبور شدم اول با استفاده از تابع cv2.circle بیام ببینم اون نقطه کجا میفته دقیقا و بقیش رو با جمع و کم کردن یه عددی درست کردم.

پس از بدست آوردن ماتریس تبدیل، این ماتریس را به تابع warpPerspective داده تا خروجی تطابق یافته را به بدهد.

عکسی که بهش رسیدم این بود:



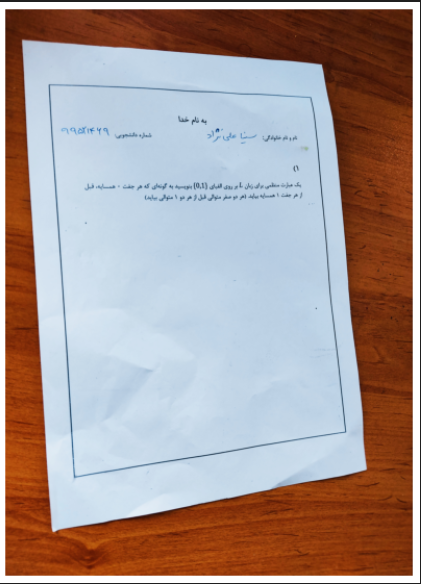
حال باید به گونه‌ای این ماسک را روی صورت انیشتن قرار میدادم، برای این کار از تابع cv2.bitwise\_or طبق راهنمایی اساتید حل‌تمرین استفاده کردم.

و نتیجه نهایی به صورت زیر بدست آمد:



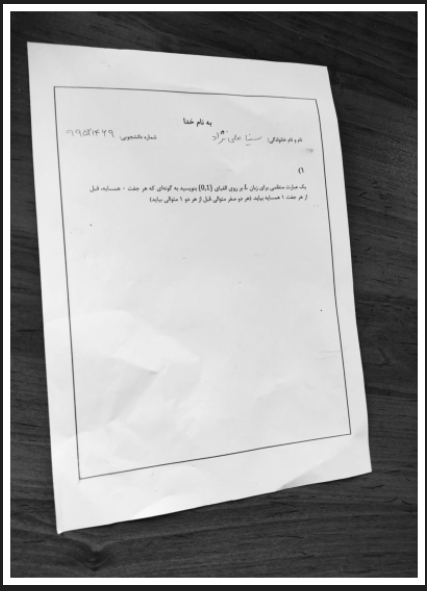
# سوال 4-

تصویر اولیه من برای این سوال به شکل زیر بود.

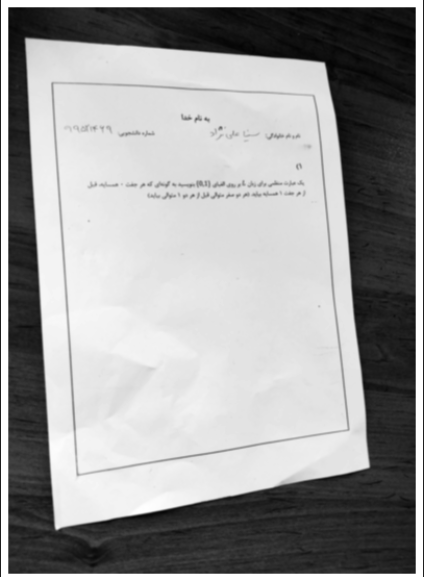


### الف)

ابتدا اومدم به فضای خاکستری منتقلش کردم.



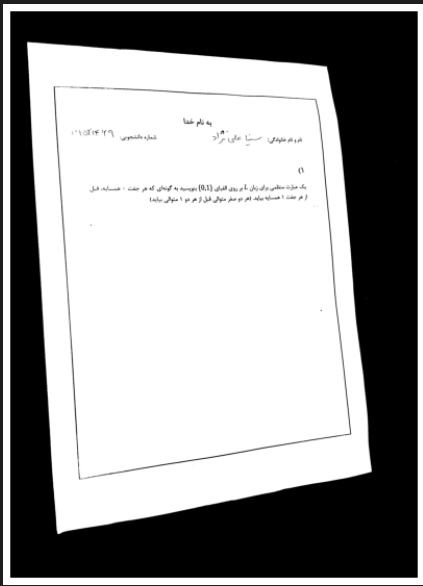
سپس فیلتر گاوسی رو روش اعمال کردم. این کار رو از این جهت انجام دادم که لبه‌های نویزی از بین برن و لبه‌های اصلی مثل خود برگه باقی بماند.



مشکلی که برام پیش میومد این بود که الگوریتم Canny لبه های زیادی رو از پس‌زمینه که میز بود پیدا میکرد و در نتیجه تابع findContours تعداد کانتورهای زیادی رو پیدا می‌کرد. پس اومدم کل پس‌زمینه رو سیاه کردم و این کار رو با استفاده از تابع cv2.threshold انجام دادم. همانند زیر:

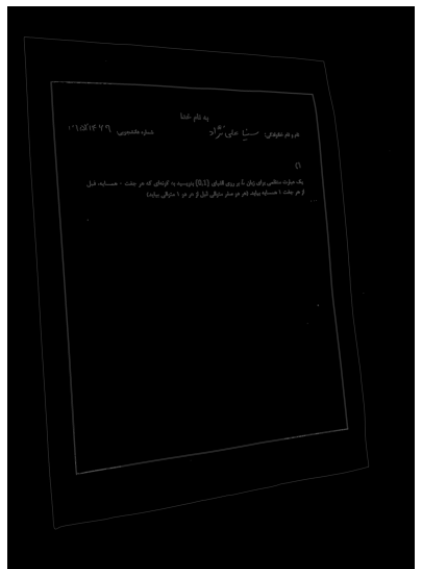


و نتیجه حاصل به شکل زیر شد:



یکی از مشکلات دیگر که بهش برخورد کرده بودم این بود که تابع findContours برای خود برگه کاغذ چندین کانتور خروجی میداد به جای اینکه یکی خروجی بده، برای این روشهای زیادی مثال عملگر بسته زدن یا عملگر گسترش و سپس سایش رو امتحان کردم ولی مشکلم حل نشد. در انتها با استفاده از همین thresholding مشکلم حل شد.

سپس خروجی thresholding رو به لبه یاب canny دادم. خروجی canny به صورت زیر شد:



علت انتخاب canny به عنوان لبه یاب این بود که این الگوریتم تا جایی که بتواند لبه‌های پیوسته می‌سازد و همچنین تا جای ممکن لبه‌های باریک ایجاد میکند.

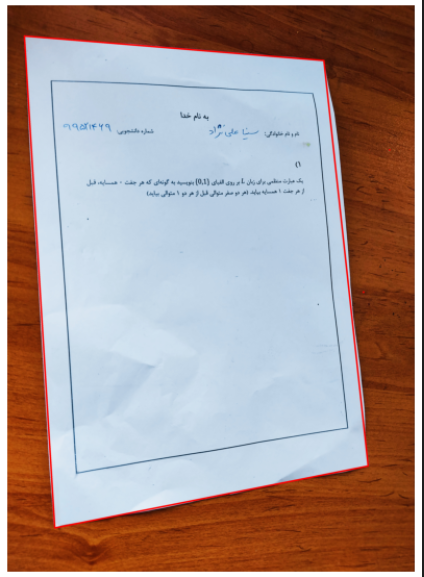


پارامترهایی که استفاده کردم برای آستانه گذاری دو مرحله ای 5 و 30 بود و این رو با آزمون و خطا بدست آوردم به طوری که تعداد لبه های نویزی به حداقل برسه و همزمان لبه های اصلی پیوسته و کامل باشن.

### ب)

در نهایت با استفاده از توابع findContours و approxPolyDP به چهار نقطه گوشه برگه رسیدم.

شکل زیر با استفاده از تابع cv2.drawContours بدست آمد:



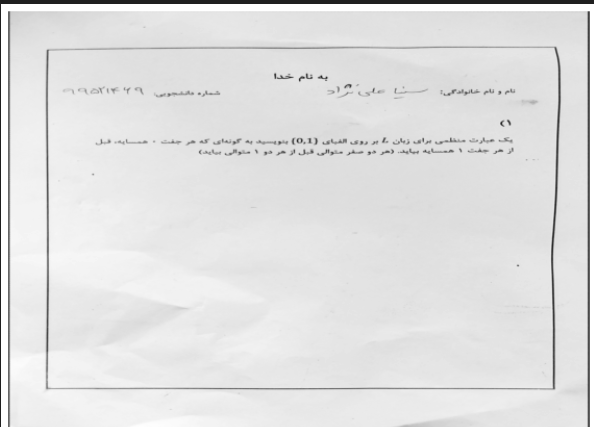
همانطور که میبینید چهار نقطه گوشه برگه مشخص شده و به هم وصل شدند.

### ج)



بقیه مراحل همانند کاری بود که با عکس ماسک و انیشتن انجام دادیم، تعیین نقاط مبدا و مقصد متناظر و بدست آوردن ماتریس تبدیل و انطباق عکس.

نتیجه نهایی به صورت زیر بدست آمد:

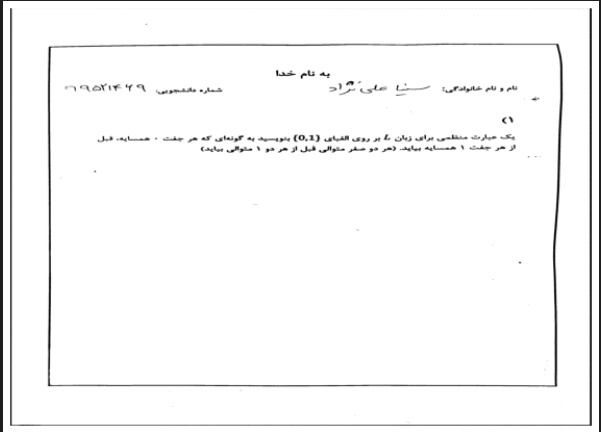


### د) امتیازی:

برای بهبود کیفیت این عکس کارهای متفاوتی از جمله اعمال clahe یا sharp کردن برای دیده شدن بهتر کلمات انجام داد. همچنین می‌توان از فیلترهایی مثل bilateral filtering برای کاهش نویز و همزمان حفظ حداکثری کیفیت استفاده کرد.

همونطور که میبینید تو عکس نهایی من به دلیل شکستگی هایی که برگه داشت، بعضی قسمت ها سایه دار دیده میشه. کاری که به من کمک کرد این مشکل رو حل کنم و کیفیتش رو بهبود بدم، اعمال threshold به کمک تابع cv2.threshold بود.

نتیجه نهایی به صورت زیر بود:



# سوال 5-

### الف) مراحل الگوریتم گوشه یاب Harris :

1. ابتدا باید مشتقهای افقی و عمودی تصویر را بدست آوریم.
2. در این مرحله مربع مشتق‌ها را محاسبه میکنیم.
3. حال باید به کمک این مقادیر، ماتریس M که به آن ماتریس تنسور نیز گفته میشود را بسازیم. پس از ساخت ماتریس، یک پنجره w را بر روی آن اعمال میکنیم، این پنجره معمولا کلا از 1 تشکیل شده است ولی میتواند پنجره گاوسی باشد. این پنجره باعث میشود گوشه های نویزی تا حدودی از بین بروند.
4. حال که ماتریس M بدست آمد، میتوان مقادیر ویژه‌ی آن را بدست آورد. اگر ماتریس، دو مقدار ویژه بزرگ داشت، یعنی گوشه داریم و اگر فقط یک مقدار ویژه بزرگ داشت، نشان دهنده لبه و اگر هر دو مقدار کوچک باشند، نشان دهنده ناحیه flat است. برای این موضوع یک متغیر جدید به اسم R تعریف میشود که برابر است با:

R = y1\*y2 – k(y1 + y2)^2

که y1,y2 مقادیر ویژه ماتریس ما هستند. اگر هر دوی این مقدار بزرگ باشند در نتیجه R مقدار بزرگی خواهد بود. فرمول بالا را میتوان به صورت زیر بازنویسی کرد:

R = det(M) – k(trace(M))^2

1. مشکلی که پیش می‌آید، وقتی در یک نقطه ای گوشه داریم، نقاط اطراف آن گوشه هم مقادیر R بزرگ پیدا میکنند و به عنوان گوشه معرفی میشوند، ولی ما میخواهیم این گوشه های اضافی حذف شوند، برای این کار از همان ایده Non-maximum suppression استفاده می‌کنیم.

### ب)

برای پیاده‌سازی این الگوریتم به طور کلی بخوام بگم، اومدم یه پنجره فرضی گرفتم و روی تصویرم لغزوندم. روی هر پیکسل مرکزی که این پنجره روش افتاد، اومدم ماتریس M رو برای اون پنجره محاسبه کردم و مقدار R برای اون ماتریس بدست آوردم و اون مقدار R رو برای اون پیکسل مرکزی در نظر گرفتم. در نهایت به یه ماتریس از پیکسلها رسیدم که توش مقادیر R مربوط به هر پیکسل ذخیره شده. بعدش اومدم اونایی که local maximum نبودن مقدار R رو براشون صفر کردم که این همون مرحله NMS است.

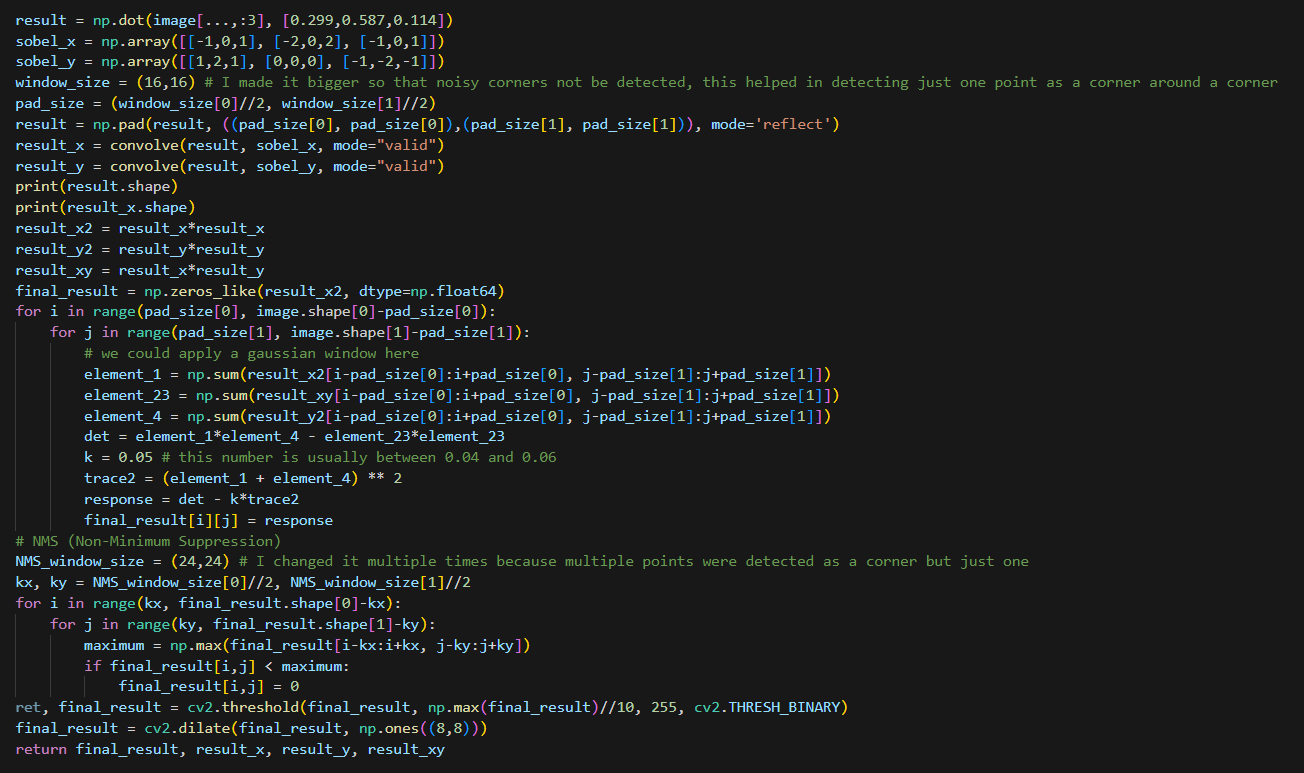
بعدش اومدم یک threshold متناسب برای این مقادیر R در نظر گرفتم و مقادیر بیشتر از اون رو گوشه گرفتم و کمتر از اون رو غیر گوشه.

در نهایت برای اینکه گوشه ها توی تصویر بهتر مشخص بشن، اومدم از عملگر مورفولوژی گسترش استفاده کردم.

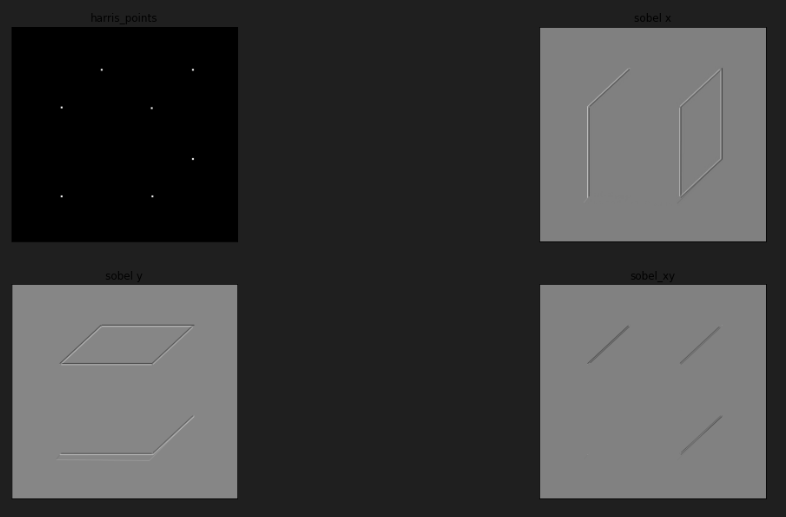
کد همه مواردی که عرض کردم در پایین آمده است:

یک سری اعمال padding هم انجام دادم که در نقاط گوشه مشکلی پیش نیاد، البته برای padding باید روش مناسب رو انتخاب میکردم، وقتی از zero-padding استفاده میکردم، نقاط انتهایی تصویر رو گوشه تشخیص میداد. بهترین نوع padding همون reflect بود.

برای محاسبه مشتقها هم از کرنل سوبل استفاده کردم و با استفاده از تابع آماده convolve در کتابخانه scipy.signal با عکس کانوالو کردم.



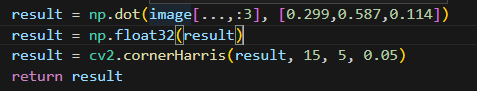
نتیجه نهایی:



چالشهایی که خوردم: چالشی که وجود داشت این بود که با وجود اعمال nms باز هم چندین نقطه در نزدیکی یه نقطه گوشه، گوشه تشخیص داده میشد، برای این اومدم سایز پنجره‌ای که روش nms زدم رو زیاد کردم، البته بازم حل نشد، این بار اومدم سایز پنجره‌ای که براش هر بار ماتریس M رو بدست میاوردم رو زیاد کردم، که نتیجه داد.

چالش دیگه این بود که در تصویر حاصل نقاط گوشه مشخص نبودن چون فقط یک پیکسل بودن، برای این از عملگر گسترش استفاده کردم که مشخص بشن.

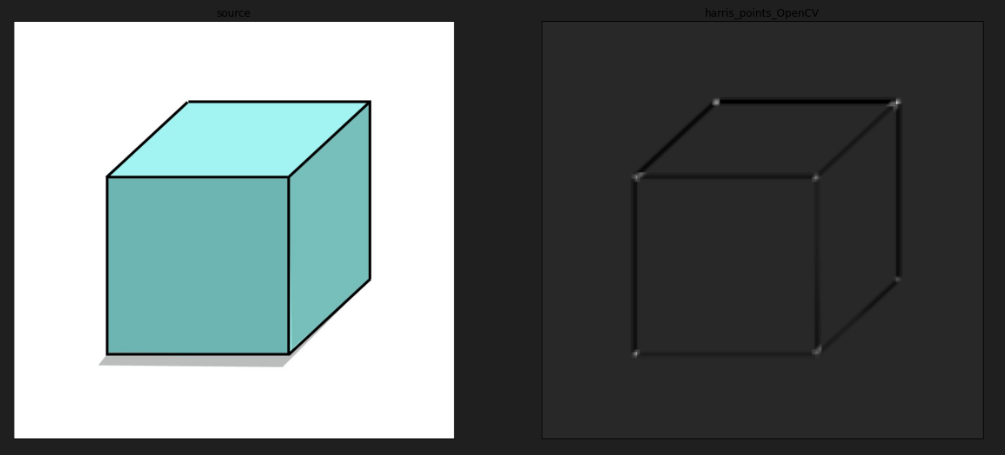
# پیاده سازی الگوریتم harris به کمک توابع آماده OpenCV :



به کمک تابع cv2.cornerHarris میتوان گوشه های تصویر را استخراج کرد.

پارامترها: پارامتر اول عکس سورس است. پارامتر دوم سایز پنجره‌ای است که ماتریس رو هربار براش محاسبه میکردیم. پارامتر سوم سایز کرنل سوبلی هست که برای بدست آوردن مشتق ازش استفاده شده و پارامتر چهارم مقدار در نظر گرفته شده برای متغیر k در فرمول مربوط به R است.

خروجی کد بالا:



خروجی هردومون گوشه ها رو به درستی تشخیص داده، خروجی مربوط به کتابخانه آماده نقاط لبه را هم نمایش میدهد زیرا نقاط لبه کمترین مقدار R دارند، در این تصویر سیاه دیده میشوند و ناحیه flat به رنگ خاکستری است. اما کدی که من زدم اومدم تصویر رو باینری کردم، یا گوشه هست که میشه 255 یا گوشه نیست که میشه صفر.

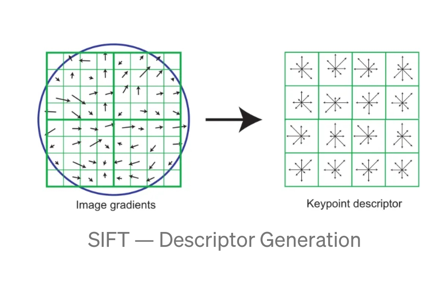
# سوال 6-

هر کدام از الگوریتم های استخراج ویژگی که در ادامه می‌آیند، مزایا و معایب خودشان را دارند و با توجه کاربرد ما انتخاب میشوند.

SIFT (Scale-Invariant Feature Transform):

این الگوریتم با این هدف توسعه پیدا کرد که نسبت به تغییر مقیاس و چرخش شدید مقاوم باشد و همینطور هم شد. هرچند از لحاظ محاسباتی هزینه بر است و برای کاربردهای real-time مناسب نیست. این الگوریتم شامل دو بخش است: 1- تشخیص نقاط کلیدی 2- استخراج توصیفگر برای نقاط کلیدی

برای تشخیص نقاط کلیدی از Laplacian of Gaussian و برای استخراج توصیفگر از orientation histogram استفاده میشود بدین صورت که در یک همسایگی جهت گرادیان ها مورد استفاده قرار میگیرد. شکل زیر نشان دهنده نحوه استخراج توصیفگرها است.



SURF (Speeded Up Robust Features):

یک جایگزین سریعتر برای SIFT هست ولی مقداری از صحت و مقاوم بودن را فدای سرعت میکند. همچنین برای شناسایی نقاط کلیدی در تصویرهایی که الگوهای تکراری دارند بسیار مناسب هست.

ORB (Oriented FAST and Rotated BRIEF):

ترکیبی از دو الگوریتم قبل است و سریع ترینشان است. برای کاربردهای real-time بسیار مناسب است مانند علم رباتیک و رباتها که نیاز به تصمیم آنی دارند. هرچند این الگوریتم ممکن است به اندازه SIFT , SURF در مواقع تغییر زیاد مقیاس و جهت خوب کار نکند.



این شکل مقدار زمان لازم برای استخراج 300 توصیفگر نقطه کلیدی را نشان میدهد، این مقدار برای الگوریتم ORB کمترین مقدار را دارد و نسبت به دو الگوریتم دیگر فاصله بسیاری دارد.

به طور خلاصه برای خیلی از کاربردها الگوریتم ORB جوابگو است و نسبت به تغییر شدت نور و زاویه مقاوم است و در بهترین زمان نیاز ما را حل میکند ولی مواقعی نیز است که ترجیح میدهیم از دو روش دیگر استفاده کنیم زیرا به دقت بالاتری نیاز داریم.

پایان