به نام خدا

مبانی بینایی کامپیوتر

دکتر محمدی

تمرین سه

علی عطاریان – ۹۹۵۲۱۴۵۱

سوال ۱) الف)

ابتدا مقادیر را نرمالایز کرده و سپس با توجه به فرمول روبرو پیاده‌سازی می‌کنیم:

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

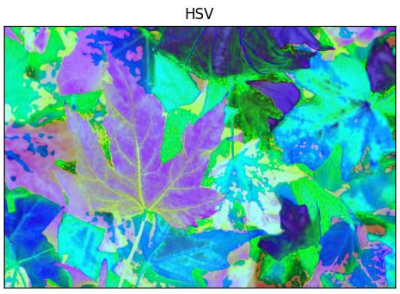
$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1-K \\ 1-K \\ 1-K \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

نتیجه:

```
[0.31372549 0.23529412 0.         0.49019608]
```

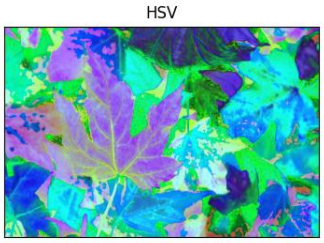
سوال ۱) ب)

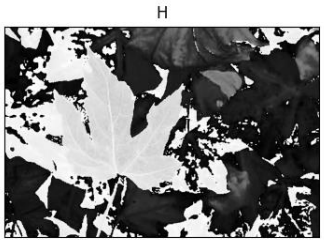
در تابع cv2.cvtcolor از فلگ های cv2.COLOR_BGR2YCR_CB و cv2.COLOR_BGR2HSV استفاده می‌کنیم

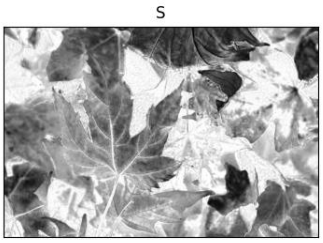


سوال ۱) ج)

ابتدا تصویر را به فضای HSV می‌بریم و حال کانال‌های این تصویر به ترتیب مقادیر H,S,V را نشان می‌دهند.



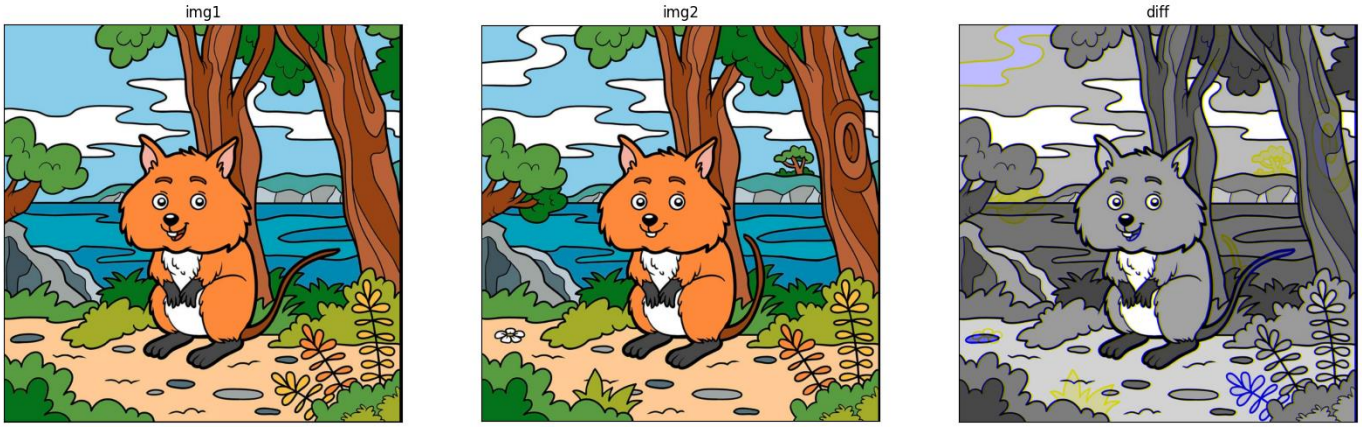




سوال ۱) د)

حالت خاکستری هر تصویر را به عنوان یکی از کانالهای یک تصویر RGB قرار می‌دهیم. تصویر۱ را در کانالهای آبی و سبز و تصویر۲ را در کانال قرمز قرار دادیم پس رنگ زرد مربوط به نقاط متمایز تصویر۲ و رنگ آبی مربوط به نقاط متمایز تصویر۱ می‌باشد.

در پیاده‌سازی به این مشکل برمی‌خوریم که دو تصویر ابعاد متفاوتی دارند. به همین دلیل باید از مینیمم طول و عرض استفاده کرد.



سوال ۱) ه)

۱. ادراک: برخی فضاهای رنگی سعی می‌کنند درک انسان از رنگها را بازسازی کنند مانند فضای HSI. مثلا انسان هنگام توصیف رنگ از عبارت (ترکیب قرمز و سبز) استفاده نمی‌کند بلکه به صورت (زرد روشن) بیان می‌کند که به مدل رنگ Hue، اشباع Saturation، شدت روشنایی Intensity نزدیکتر است

۲. پایداری نسبت به روشنایی: برخی مدلها مانند HSV نسبت به تغییرات شرایط نوری نامتغیر هستند پس برای کاربردهایی مثل یافتن اشیاء مناسب هستند.

۳. جدایی نور و روشنایی: فضاهایی مانند YCbCr طراحی شده‌اند که کانالهای نور و روشنایی در آنها جدا باشند که مناسب کاربردهایی مثل فشرده‌سازی ویدیو می‌باشد.

۴. بهینگی محاسباتی: فضاهایی مانند gray-scale یا binary color باعث افزایش سرعت محاسبات کامپیوتری در برخی کاربردها می‌شوند.

۵. کاهش هزینه‌ها: فضای رنگی‌ای مثل CMYK طراحی شده است که در هنگام چاپ تصاویر با رنگهای کمتر بتوان ترکیب‌های متنوع‌تری ایجاد کرد تا در هزینه رنگ صرفه‌جویی شود.

سوال ۲)

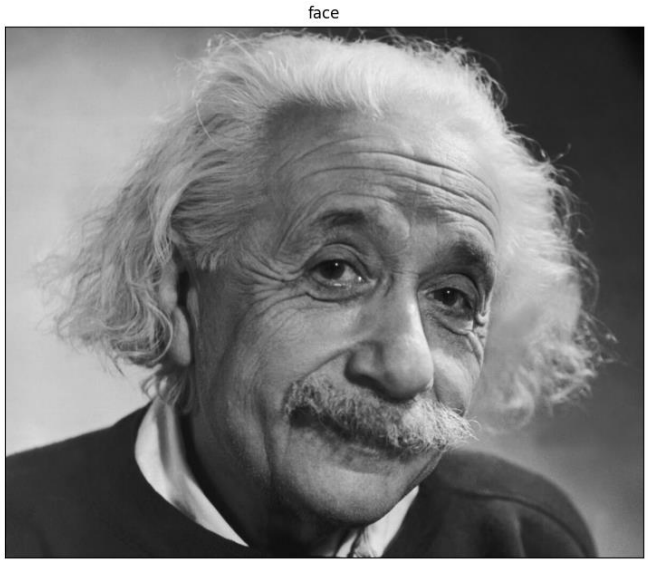
ابتدا تصاویر را یکی‌یکی می‌خوانیم و در یک ردیف نشان می‌دهیم.

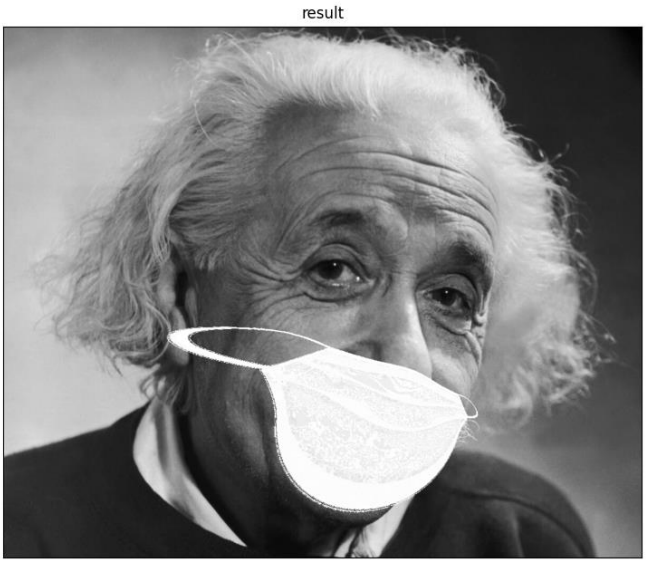
با دستور cv2.Stitcher.create() یک instance از آبجکت موردنظر را می‌سازیم. با استفاده از متد stitch تصاویر را به یکدیگر وصل می‌کنیم. این متد دو خروجی می‌دهد که یکی از آنها نشان می‌دهد آیا متد موفق به تشکیل تصویر شده است یا خیر و دومی تصویر خروجی است.

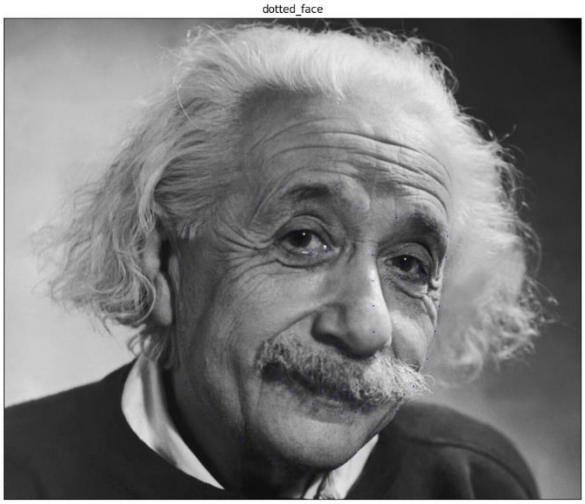
سوال ۳)

توضیح تابع put_mask: ابتدا آبجکت detector را می‌سازیم که وظیفه‌اش تشخیص چهره‌ها در تصویر می‌باشد. سپس مدل یادگیری dlib.shape_predictor را با دیتاستی که دانلود کرده‌ایم آموزش می‌دهیم. این مدل وظیفه‌اش تشخیص نقاط کلیدی چهره است. در اینجا face_rects = detector(gray_face, 1) مستطیل‌هایی که دور چهره‌ها کشیده شده‌اند (در این مسئله یک مستطیل) را دریافت می‌کنیم. پارامتر دوم ورودی نشان‌دهنده تعداد لایه‌هایی است

که برای upscale کردن تصویر استفاده می‌شود. هرچه این عدد بیشتر باشد یعنی امکان یافتن تصاویر بیشتر بیشتر است اما هزینه محاسباتی بالا می‌رود. با توجه به مسئله فعلی ۱ لایه کافی است. حال نقاط کلیدی هر چهره را دریافت می‌کنیم و آن را برای آسانی از تایپ آبجکت به np.array تبدیل می‌کنیم. فور لوپ منطقا در اینجا لازم نبود اما بدون آن درست کار نمی‌کرد! ۴ لندمارک مربوط به چهره و عکس را با آزمون و خطای بسیار زیاد به دست می‌آوریم. باید توجه داشت که از np.float32 حتما استفاده شود. تابع cv2.getPerspectiveTransform با نقاط کلیدی منبع و مقصد ماتریس تبدیل را می‌سازد و سپس با تابع cv2.warpPerspective تبدیل را روی منبع انجام می‌دهیم. به پارامتر size دقت شود که به ترتیب (عرض، طول) باشد. سپس برای قرار دادن ماسک روی صورت، از آنجایی که نقاط بدون ماسک در تصویر خود ماسک سیاه هستند از cv2.bitwise_or استفاده می‌کنیم.





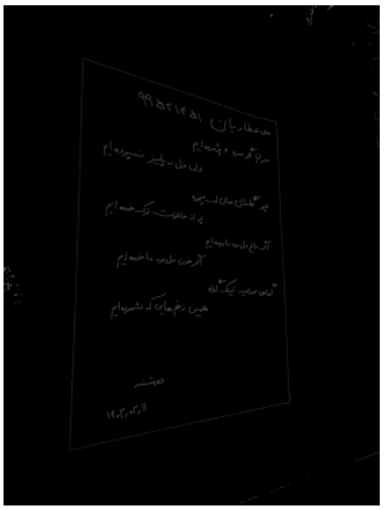


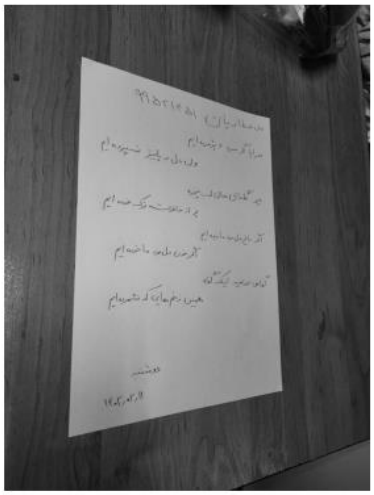
قسمت show landmarks on face فقط برای visualize کردن بهتر پیاده‌سازی شده است که ۶۸ نقطه کلیدی چهره را نشان می‌دهد.

سوال ۴) الف)

برای نویزگیری از bilateralfiltering استفاده می‌کنیم چون در عین کم‌کردن نویز تصویر را محو نمی‌کند (برعکس فیلترهایی مانند فیلترگوسی که ممکن است باعث شود جزییاتی را از دست بدهیم مخصوصا در متن). برای لبه یابی از canny استفاده می‌کنیم چون لبه‌های دقیق‌تر و نازکتر می‌دهد و برای ادامه کار مثل پیدا کردن کانتور راحت‌تر هستیم. همچنین لبه‌یاب sobel جواب خاکستری برمی‌گرداند و binary-color نیست که برای ورودی findContours نامناسب است. پارامترهای cv2.bilateralfilter به ترتیب تصویر ورودی، سایز فیلتر، سیگما در فضای رنگی، سیگما در فضای مختصاتی می‌باشد. هر چه سیگمای رنگی بیشتر باشد، رنگهایی دورتر از رنگ پیکسل موردنظر با آن ترکیب می‌شوند (در صورتی که همسایه پیکسل باشند) و هر چه سیگمای مختصاتی بیشتر باشد، پیکسلهایی دورتر همدیگر را تحت تاثیر قرار می‌دهند (به شرطی که رنگشان اندازه کافی شبیه باشد). در کل با زیاد شدن هرکدام ۳ پارامتر عددی وارد شده، خروجی smooth تر می‌شود. مقداری را انتخاب کردیم که بین حذف نویز و حذف جزییات تعادل برقرار شود. پارامترهای ورودی cv2.canny را در داک تمرین ۲ توضیح داده‌ایم:

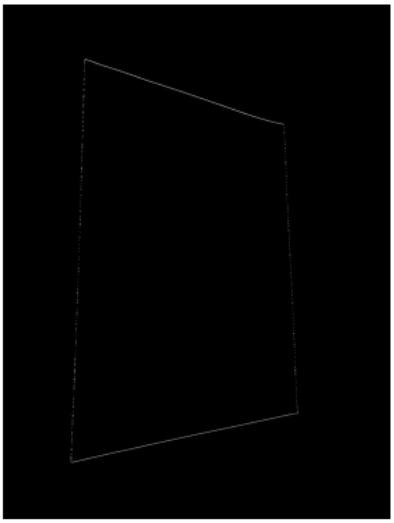
"پارامترهای تابع cv2.canny به ترتیب تصویر ورودی، minValue و maxValue هستند. مقادیر بالاتر از maxValue حتما لبه هستند و مقادیر کمتر از minValue حتما لبه نیستند اما مقادیر بین این دو اگر به نقاط لبه(نقاط با مقدار بیش از maxValue) وصل باشند آنگاه لبه حساب می‌شوند." با آزمون و خطا متوجه می‌شویم مقادیر ۶۰ و ۹۰ لبه‌ها را به خوبی می‌یابند.





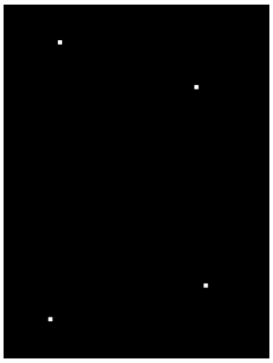
سوال ۴) ب)

کانتورها را پیدا می‌کنیم، با توجه به وجود متن در تصویر کانتورهای زیادی یافت می‌شوند اما بزرگترین کانتور نشان‌دهنده کاغذ است، آن را پیدا کرده و در پس‌زمینه سیاه نشان می‌دهیم.

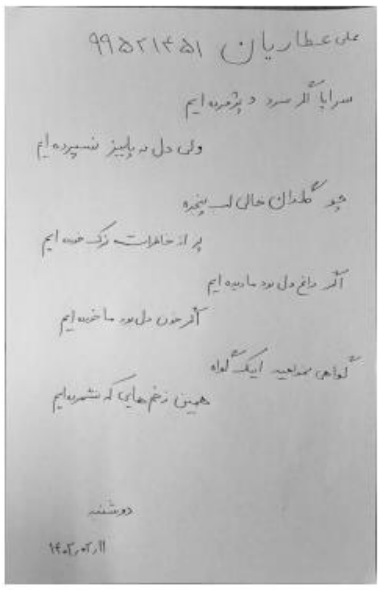


سوال ۴) ج)

ابتدا باید چهار نقطه گوشه کاغذ را بیابیم تا آنها را به چهار گوشه تصویر نهایی مپ کنیم. ابتدا کانتور کاغذ را به یک چندضلعی تخمین می‌زنیم. در ابتدای پیاده‌سازی به علت اپسیلون بسیار کم یکی از گوشه ها پیدا نمی‌شد اما با آزمون و خطا متوجه می‌شویم مقدار ۰.۰۵ کمترین اپسیلونی است که گوشه سمت راست بالا را برمی‌گرداند. اما حال cv2.approxpolyDP بیشتر از ۴ نقطه برمی‌گرداند. می‌دانیم نقاط با بالاترین و پایین‌ترین ارتفاع به ترتیب نقاط سمت چپ بالا و سمت چپ پایین هستند. همچنین نقطه با ارتفاع نزدیک به نقطه سمت چپ بالا، نقطه سمت راست بالا می‌باشد و نقطه با ارتفاع نزدیک به نقطه سمت چپ پایین، نقطه سمت راست پایین است. بر این اساس تابع get_corners را می‌نویسیم که ۴ نقطه را برگرداند.

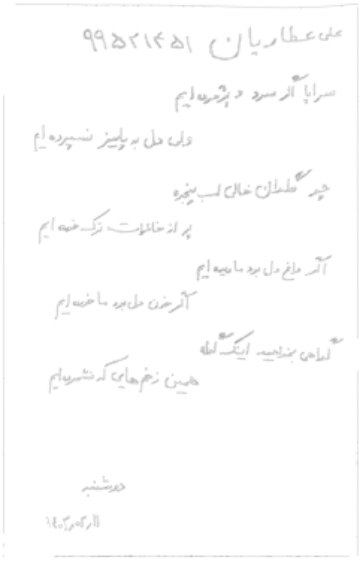


با داشتن این نقاط می‌توان تابع تبدیل را گرفت و تبدیل را انجام داد.



سوال ۴) د)

برای بهبود کیفیت تصویر باید کنتراست بین متن و کاغذ را زیاد کنیم تا متن بهتر جلوه کند پس ابتدا از کلاهه استفاده می‌کنیم. سپس به علت نویز زیاد یکبار نویزگیری می‌کنیم. حال تصویر را به فضای باینری می‌بریم تا فقط متن دیده شود. از کنی استفاده می‌کنیم اما چون پس‌زمینه سیاه است به کمک cv2.threshhold و لیبل cv2.THRESH_BINARY_INV پس زمینه را سفید می‌کنیم.



سوال ۵) الف)

الگوریتم هریس برای شناسایی گوشه‌ها در تصویر است که برپایه شناسایی ناحیه‌هایی است که در آنها تغییر در تمام جهت‌ها رخ می‌دهد. ابتدا با عملگر سوبل مشتق‌های افقی و عمودی را در فضای grayscale به دست می‌آوریم. سپس توان دو مشتق ها و حاصلضرب آنها را محاسبه کرده و چنین ماتریسی تشکیل می‌دهیم.

$$\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

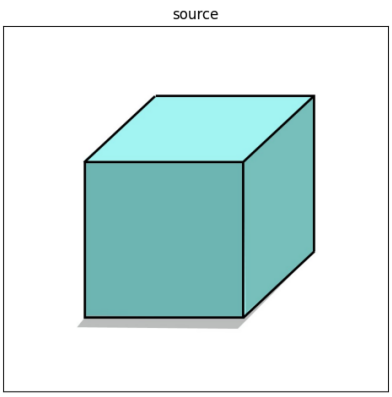
سپس یک پنجره برای تاثیر همسایه‌های محلی در نظر می‌گیریم که می‌تواند گوسی یا مستطیلی باشد (معمولا مستطیلی است). در نهایت با اعمال چنین فرمولی روی تک‌تک پیکسل‌ها و حذف مقادیر غیربیشینه (non-maximum suppression) گوشه ها با مقادیر روشن (سفید) یافت می‌شوند.

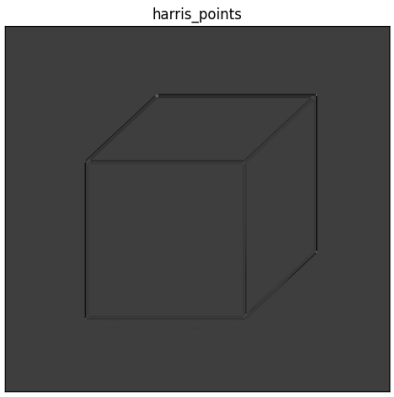
$$R = det(M) - k\big(trace(M)\big)^2$$

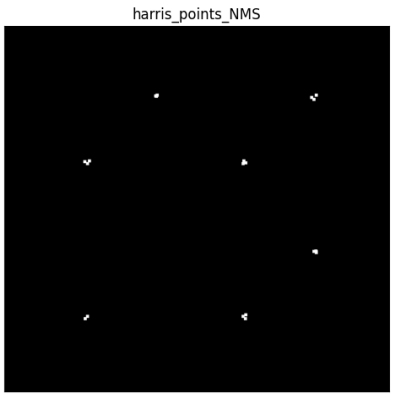
مقدار k در حساسیت الگوریتم تاثیرگذار است به طوری که هرچه بیشتر باشد، حساسیت بالاتر است. معمولا مقداری است بین ۴ تا ۶ درصد.

سوال ۵) ب)

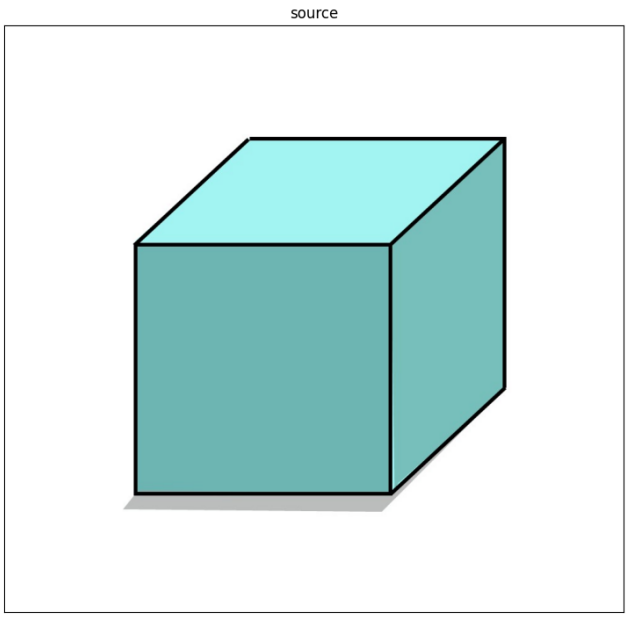
مطابق الگوریتم ابتدا مشتق افقی و عمودی را با سوبل به دست می‌آوریم. سپس توان دو و حاصلضرب مشتقات را حساب می‌کنیم. حال پنجره مستطیلی را روی این سه مقدار(که در واقع ماتریسی به اندازه ابعاد تصویر هستند) اعمال می‌کنیم. مقدار R را می‌یابیم. حال مقادیر غیربیشینه را حذف می‌کنیم. در این فرایند هر پیکسلی که مقدارش از آستانه تعیین شده (یک درصد ماکسیمم) و ۸ همسایه اش بیشتر باشد ۱ می‌شود و باقی پیکسل‌ها صفر. در نهایت صرفا جهت اطمینان جواب پیش و پس از حذف مقادیر غیربیشینه را برمی‌گردانیم تا نمایش دهیم. برای نمایش بهتر نقاط سفید که بسیار کوچک هستند از cv2.dilate که نقاط روشن را گسترش می‌دهد.

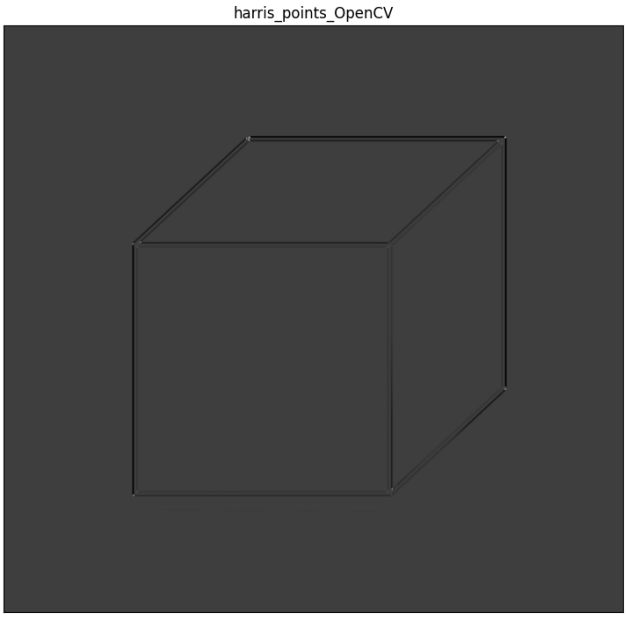






برای استفاده از تابع آماده opencv ابتدا تصویر را خاکستری کرده و سپس پارامترهای اندازه پنجره هریس (در نظر گرفتن همسایه‌ها)، سایز فیلتر سوبل و k را به عنوان ورودی به cv2.cornerharris می‌دهیم. مشاهده می‌کنیم خروجی مشابهی تولید شد.





سوال ۶)

الگوریتم Scale-invariant Feature Transform ابتدا با استفاده از Difference of Gaussian اکسترمم های محلی را شناسایی می‌کند. DoG بدین صورت عمل می‌کند که یک ورژن از عکس که توسط فیلتر گوسی بلار شده است را از یک ورژن کمتر بلار شده منها می‌کند. سپس به هر کدام از این لوکال اکسترمم ها یک جهت‌گیری اختصاص داده می‌شود که با استفاده از گرادیانت خود و همسایه‌هایش به دست می‌آید. سپس یک همسایگی ۱۶*۱۶ از هر نقطه کلیدی به ۴ بلاک ۴*۴ تقسیم می‌شود که هر بلاک ۴ جهت‌گیری در ۸ جهت را نشان می‌دهد یعنی جمعا ۱۲۸ ویژگی. پس هر نقطه کلیدی با ۱۲۸ ویژگی توصیف می‌شود.

الگوریتم Speeded-Up Robust Feature نسخه سریع‌تر SIFT است. این الگوریتم به جای DoG از فیلترهای جعبه‌ای استفاده می‌کند که بسیار سریع تر می‌باشند زیرا فیلترهای کوچکتری از فیلتر گاوسی هستند و از مقادیر ساده‌ای مثل -۱ و +۱ استفاده می‌کنند. برای تشخیص جهت‌گیری نیز از Haar-Wavelet استفاده می‌کند که در واقع روشی بر اساس مشتق گیری است. سپس یک همسایگی ۲۰*۲۰ از هر نقطه به ۱۶ ناحیه ۴*۴ تقسیم می‌شود که هر ناحیه ۴ مقدار(مشتق افقی و عمودی و قدرمطلق آنها) را نشان‌می‌دهد. پس جمعا ۶۴ ویژگی برای هر نقطه کلیدی تولید می‌کند.

الگوریتم Orientated FAST and Robust BRIEF جدیدترین الگوریتم از میان این ۳ می‌باشد. این الگوریتم ابتدا با استفاده از FAST نقاط کلیدی را شناسایی می‌کند. FAST نقاط کلیدی را بر اساس مقایسه شدت روشنایی یک پیکسل با همسایه‌اش در یک ناحیه دایروی پیدا می‌کند (گوشه ها را تشخیص می‌دهد). سپس جهت‌گیری را با استفاده از intensity centroid method شناسایی می‌کند که از یک ناحیه دایروی استفاده می‌کند تا جهت کلی را به دست آورد. سپس از BRIEF استفاده می‌کند تا یک توصیف binary برای هر نقطه‌کلیدی به دست آورد. این الگوریتم جفت پیکسل‌ها را مقایسه می‌کند و بر اساس نسبت مقادیر آنها یک رشته دوتایی تولید می‌کند. ORB از آنجایی که جهت‌گیری را شناسایی کرده، می‌داند کدام جفت پیکسل‌ها را به BRIEF و این باعث بهبود performance می‌شود.

الگوریتم ORB از دو الگوریتم دیگر سریعتر است اما نسبت به تغییر اندازه (scale) پایدار نیست و به محو بودن و نویز بیشتر حساس است. این الگوریتم در محاسبه تقریبا ده برابر از دوتای دیگر سریع‌تر است و بیش از ۳ برابر نقاط کلیدی استخراج می‌کند. در شناسایی نقاط کلیدی یکسان در تصاویری که چرخیده‌اند یا تغییر در روشنایی داشته اند ۳ الگوریتم عملکرد نسبتا مشابهی داشته‌اند. در مقایسه مختصات نقاط کلیدی تصویر ترنسفورم شده و تصویر اصلی، SIFT به‌طور متوسط ۲۰ پیکسل خطا می‌کند. SURF و ORB در حدود صفر. نتیجه‌گیری اینکه سریع‌ترین الگوریتم ORB است با اینکه دو الگوریتم دیگر کاربردهای خود را دارند.