اصول پردازش تصاویر

نیم‌سال اول ۱۴۰۰-۱۴۰۱

مدرس: مصطفی کمالی

دانشکده ریاضی

تمرین سری اول

نام و نام‌خانوادگی: -

شماره دانشجویی:-

پرسش ۱

در این سوال سعی شده است با استفاده از تابعی به نام change_value افزایش نسبی روشنایی در تمام پیکسل ها و نیز افزایش کنتراست را انجام بدهیم.

در ابتدای کد، تصویر داده شده بصورت rgb وارد برنامه می‌شود. سپس با استفاده از تابع cv۲.cvtColor تصویر به فرمت hsv تبدیل می‌شود. در ادامه دو عدد حلقه تودرتو آمده است. یک حلقه برای پیمایش افقی و یک حلقه برای پیمایش عمودی عکس. سقف این دو حلقه به ترتیب عرض و طول تصاویر هستند. به اینصورت مقدار vlue تک تک پیکسل های تصویر به تابع change_value داده می‌شود و مقدار اصلاح شده آن به عنوان مقدار vlue جدید همان پیکسل در آن اعمال می‌شود. سپس در ادامه فرمت رنگی عکس را با cv۲.cvtColor به rgb تغییر می‌دهیم و آنرا با استفاده از imwrite ذخیره می‌کنیم. سپس به BGR تبدیل کرده و با استفاده از plt.imshow() نمایش می‌دهیم. خروجی این برنامه حدود ۲۴ ثانیه می‌باشد.

حال توضیحی در مورد تابع change_value و نحوه کارکرد آن می‌دهیم. این تابع را برای جمع کردن کل تغییرات روی پیکسل ها در یک تابع ساخته‌ایم. این تابع یک آرگومان value دریافت می‌کند و مقدار حاصل تقسیم آن بر عدد ۲۵۶ در متغیر value ذخیره می‌گردد. سپس با استفاده از تابع نمایی با نمای ۰.۳ روشنایی همه نقاط را زیاد می‌کنیم اما بنابر ذات این تابع نقاط تاریک تر بیشتر روشن می‌شوند. سپس مقدار بدست آمده را در عدد ثابت a که در اینجا مقدار بهینه ی ۱.۹ قرار داده شده است ضرب می‌کنیم. به این صورت پیکسل ها به ۲ دسته تقسیم می‌گردند، دسته‌ای روی ۱ و دسته‌ای زیر ۱ هستند. حال همین اعداد به دست آمده را اگر بتوان عددی مانند ۱.۹ برسانیم، اعداد روی ۱ بزرگ تر و اعداد زیر ۱ کوچکتر می‌شوند، و هرچقدر از ۱ فاصله بیشتری داشته باشند نیز اختلاف آنها بیشتر خواهد شد و این باعث افزایش کنتراست بصورت دستی و خروجی تصویر می‌شود. در انتهای تابع نیز مقدار value ابتدا بر a تقسیم و در ۲۵۶ ضرب می‌شود که به فرمت value به همان صورتی که وارد شده بود، تبدیل شود و خروجی داده شود.

پرسش ۲

در این سوال دقیقا مشابه الگوریتم سوال ۱ عمل شده است، با این تفاوت که کمی شرایط تغییر کنتراست را تغییر داده‌ایم. به دلیل وجود منطقه ای با روشنایی بسیار بالا (آسمان)، پس از تغییر روشنایی و کنتراست، منطقه ای تیره در بالای عکس بوجود می آید. که حاصل از بیشتر از ۲۵۶ شدن مقدار $value$ خروجی تابع است. بنابر این نیاز به الگوریتمی داشتیم که بتواند این مناطق را بصورت خودکار جدا کند و اجازه ندهد مقدارشان از ۲۵۶ بیشتر شود.

پس عبارت شرطی در انتهای تابع change_value را تعریف کردیم. در اینجا میگوییم هرگاه مقدار تغییر روشنایی پیکسل از حدی بیشتر شد، پیکسل به مقدار اولیه اش برگردد. اما در این حالت بعضی از نقاط تیره عکس که می‌خواهیم شامل افزایش کنتراست شود را نیز در بر می‌گیرد و آنها نیز ثابت می‌ماند. لذا ضریب m که نشانگر روشنایی تصاویر است نیز در آن ضرب می‌شود تا بین نقاط تاریک و روشن تفاوت زیادی بوجود بیاورد. در نهایت با استفاده از ضریب مناسب ۰.۲ و مقایسه با مقدار مقیاس ما، می‌توانیم تنها نقاط روشن آسمان را تشخیص دهیم و مانع آن شویم که در این نقاط افزایش کنتراست بیش از حد و تاریکی تصاویر رخ بدهد، از طرفی تصویر نیز طبیعی بماند و آسمان روشنایی مناسب خود را حفظ کند.

بقیه ی مراحل این سوال عینا مانند سوال ۱ است و بطور کامل در صفحه قبل توضیح داده شد. زمان اجرای این برنامه حدود ۲۵ ثانیه می‌باشد.

پرسش ۳

در سوال ۳ ابتدا تابعی به نام resize تعریف کرده‌ایم که با گرفتن یک عکس و یک ratio ، به عنوان خروجی عکس ورودی را می‌دهد که به اندازه ratio کوچک یا بزرگتر شده است. در کد ابتدا ۴ متغیر $a2$ تا $b3$ تعریف شده و ۰ به عنوان مقدار اولیه آنها در نظر گرفته شده است. سپس تصویر ورودی به عنوان یک تصویر تک رنگ سیاه و سفید دریافت می‌شود.

سپس یک حلقه فور با دامنه ۰ تا ۴ تعریف می‌کنیم. این حلقه شامل اکثر دستورات ما می‌شود. ابتدا متغیر های $a2$ ، $a3$ ، $b2$ ، $b3$ تعریف شده در اول کد، ۲ برابر می‌شوند، که در اینجا ۰ می‌مانند. سپس یک w و h بعنوان طول و عرض تصویر تعریف می‌شوند. این مقادیر برابر می‌شوند با طول و عرض عکس اولیه ضربدر ضریب $(0.0625 * (2^{count}))$. این ضریب در حلقه ی اول $\frac{1}{16}$ است، یعنی ابعاد عکس را $\frac{1}{16}$ می‌کنیم و از آن استفاده می‌کنیم و هر بار ۲ برابر شده تا اینکه در حلقه ی نهایی یعنی حلقه چهارم، این مقدار به ۱ می‌رسد و این یعنی با عکس اصلی اولیه سر و کار خواهیم داشت.

سپس متغیر dif تعریف می‌شود که رند شده ی یک سوم ارتفاع h است. این به این دلیل است که چون احتمال دارد h بر ۳ بخش پذیر نباشد، ابعاد عکس ها ما هم تفاوت کند. به همین دلیل برای ارتفاع همواره با متغیر dif سر و کار خواهیم داشت. بعد از آن نیز با دستور cv۲.resize ، بنابر متغیر های dif ضربدر ۳ و w ، سایز عکس اصلی را تغییر می‌دهیم و به مقدار مناسب آن شماره حلقه می‌رسیم و در gray_image ذخیره می‌کنیم. بعد از آن یک دستور شرطی تعریف می‌شود که در صورتی که بار اولمان بود وارد این حلقه می‌شویم، gray_۱ ، gray_۲ و gray_۳ را با بریدن های مناسب gray_image به سه قسمت مساوی تقسیم می‌کنیم و سه عکس خام با ابعاد یک شانزدهم در بار اول را تولید می‌کنیم.

در ادامه برای افزایش سرعت و دقت اجرا، الگوریتمی پیاده سازی شده است و ضرایب ثابتی در اینجا تعریف شده است. به ترتیب توضیح می‌دهیم. q یک ضریب است که نشان می‌دهد یک چندم از هر طرف عکس را قرار است ببریم و با با کانال های دیگر مقایسه کنیم. این ضریب عدد صحیح است اما در هر حلقه فرق می‌کند. بنا بر فرمولی که نوشته شده است در شماره ی اول حلقه برابر با ۱۶ و در شماره پایانی حلقه برابر با ۵ است. یعنی یک پنجم هر طرف تصویر قطع می‌شود که به ما تصویر کوچکتری بدهد تا سرعت محاسبه بالا برود. سپس یک عدد به نام ضریب داریم، که این بیانگر این است که چند درصد از عکسی که بالاتر بدست آمد باید برای جابجایی و پیدا کردن بهترین مقدار مورد استفاده قرار بگیرد. این zarib_start عدد شروع ضریب است. در هر مرحله نیز عدد خط بعدی آن یعنی p بتوان آن در $1 - zarib$ مورد استفاده قرار می‌گیرد. برای مثال در حلقه نهایی عدد ضریب به ۰.۴۹ می‌رسد و ۹۴ درصد از تصویر اصلی با اطرافش مقایسه می‌شود که سرعت بسیار خوبی با روند اجرای برنامه می‌دهد.

سپس اعداد a و b بنابر q تعریف می‌شود و با استفاده از آنها اطراف تصاویر بطور موقت قطع می‌شوند و در متغیر های cut_۱ ، cut_۲ و cut_۳ ریخته می‌شوند. اینها تصاویر آزمایشی ما هستند و کل کار ما در همین بازه هاست. بعد از آن zarib به شیوه ای که گفته شد درست می‌شود و طول و عرض آن در tool و arz وارد شده و درصد لازم بنابر zarib کم شده و در متغیر check_area قرار می‌گیرد. این عکس، عکس ثابت از کانال آبی ماست که قرار است دو کانال دیگر روی آن بلغزند. در نهایت پس از نوشتن دو مقدار اولیه برای حداقل مقدار اختلاف روشنایی ها و مقدار جابجایی ها، دو عدد حلقه فور می‌نویسیم و همان ناحیه یعنی tool و arz ضربدر مقدار کاستی از هر طرف را بعنوان رنج آن در نظر می‌گیریم.

حال با روشی همانند درست کردن check_area ، از کانال سبز عکسی به همان ابعاد می‌سازیم و بصورت ماتریسی آن دو را از هم کم و قدر مطلق می‌گیریم و همه درایه ها را باهم جمع می‌کنیم و چک می‌کنیم که آیا این کمترین مقدار ممکن است یا خیر، اگر بود، در متغیر minimum ریخته شده و اختلاف این i و j از گوشه های عکس بریده شده ی ثابت کم شده و مقدار جابجایی را به ما می‌دهد و در متغیر best_۲ ذخیره می‌گردد. سپس کار مشابه با عکس کانال قرمز و آبی انجام شده و این نتایج هم در best_۳ ذخیره می‌گردد. در نهایت این اعداد با ۴ عدد $a2$ ، $a3$ ، $b2$ و $b3$ که از قبل تعریف کرده بودیم جمع می‌شوند و به مرحله ی بعد برده می شوند.

در اینجا به طور خلاصه عکس های خام اولیه از gray_image گرفته می‌شوند و بر اساس $a2$ تا $b3$ بر روی هم می‌لغزند. یعنی از یک طرف قطع و به طرف دیگر اضافه می‌شوند. منتهی مشکلی که با آن برخورد می‌کنیم، محدودیت حاشیه های gray_image است که این با الگوریتم های ماتریسی numpy که در آن ستون و سطر های مورد نیاز با مقدار پیش‌فرض ۰ اضافه می‌شوند، حل شده است. عکس ثابت gray_۱ که همان می‌ماند؛ اما gray_۲ و gray_۳ مقادیر جدیدی می‌گیرند.

شرط اول حلقه ی اصلی‌مان هم دیگر نمی‌گذارد مقادیر اولیه را به بگیرند. پس عکس های gray_ و gray_۳ جدید دقیقا همین تصویر لغزش یافته می‌ماند و در حلقه ی بعدی نیاز نیست مقادیر زیادی را جست و جو کنیم، زیرا به هم بسیار نزدیک شده‌اند.

حال فلسفه ی دو برابر کردن مقدار قبلی و اضافه کردن مقدار جدید به $a2$ ، $a3$ ، $b2$ و $b3$ این است که وقتی ابعاد تصویرمان را داریم دو برابر می‌کنیم، لزوما نیاز است که مقادیر جابجا شده را نیز دوبرابر کنیم. در پایان که تصویر نهایی‌مان است هم نیاز نیست ۲ برابری ایجاد شود، زیرا پس از ۳ بار پیدا کردن بهترین تطابق و دوبرابر کردن مقادیر تصویر در بار آخر با کمی لغزش تصویر بزرگ اولیه، مناسب‌ترین مقدار پیدا شده و با مقادیر جابجا شده ی قبلی جمع شده و پس از اعمال دوباره ی آنها روی ۲ تصویر آبی و قرمز بسیار بزرگ نهایی، بر هم منطبق می‌شوند و از حلقه خارج می‌شویم و توسط تابع cv۲.merge باهم ادغام شده و تصویر نهایی را خروجی می‌دهد.

برای برش حواشی نیز با الگوریتم ساده و استفاده است تابع cv۲.canny ، گوشه ها با مقادیر ترش‌هولد ۱۰ و ۲۰۰ بدست می آید. حال ما باید ۴ تا حلقه به ترتیب برای راست، چپ، بالا و پایین تصویر تعریف کنیم. از آنجا که تصویر پس از اعمال فیلتر cv۲.canny به ۰ و ۲۵۵ می‌رسد، می‌آییم از بازه یک شانزدهم تا یک هشتادم هر طرف تصویر به پیش می‌رویم. در این بین یک ماتریس به ارتفاع تصویر و به عرض ۵ و ۱۰ پیکسل تعریف می‌کنیم و کل درایه ها را باهم جمع و تقسیم بر ۲۵۶ می‌کنیم. در نهایت برنامه را اینطور کامل می‌کنیم که هرگاه از تعداد کل درایه های ماتریسی که ساخته شده است بیشتر از یک پنجم آنها سفید است، برنامه را متوقف کن و نقطه بدست آمده را با اختلاف ۱۰ پیکسل به سمت داخل تصویر به دلیل احتمال بروز خطا، به عنوان حاشیه ذخیره کن.

برای ۴ طرف تصویر همین کار را تکرار می‌کنیم و پس از بریدن بر اساس ۴ عدد به دست آمده، تصویر نهایی را در result ریخته و آنرا ابتدا ذخیره سپس با دستور plt.imshow() نمایش می‌دهیم. زمان تقریبی اجرای کل برنامه، ۳۵ ثانیه می‌باشد.

حال تصاویر و مقادیر بدست آمده در هر مرحله را در اینجا ذکر می‌کنیم. دقت کنید که کانال آبی ثابت قرار داده شده است.

تصویر اول: **امیر بخارا**

مرحله $\frac{1}{16}$:

مرحله $\frac{1}{8}$

مرحله $\frac{1}{4}$:

مرحله $\frac{1}{2}$:

مرحله نهایی :

جابجایی لازم نهایی برای ساخت تصویر امیر بخارا:

برای لایه سبز:

$$x_1 = 25 \ , \ y_1 = 49$$

برای لایه قرمز:

$$x_2 = 42 \ , \ y_2 = 104$$

تصویر دوم: **مسجد سمرقند**

مرحله $\frac{1}{16}$:

مرحله $\frac{1}{8}$

مرحله $\frac{1}{4}$:

مرحله $\frac{1}{2}$:

مرحله نهایی :

جابجایی لازم نهایی برای ساخت تصویر مسجد بخارا:

برای لایه سبز:

$$x_1 = 0 \ , \ y_1 = 56$$

برای لایه قرمز:

$$x_2 = 0 \ , \ y_2 = 126$$

تصویر سوم: **قطار بخار**

مرحله $\frac{1}{16}$:

مرحله $\frac{1}{8}$

مرحله $\frac{1}{4}$:

مرحله $\frac{1}{2}$:

مرحله نهایی :

جابجایی لازم نهایی برای ساخت تصویر قطار بخار:

برای لایه سبز:

$$x_1 = 6\ ,\ y_1 = 51$$

برای لایه قرمز:

$$x_2 = 33\ ,\ y_2 = 94$$

پرسش ۴

در سوال ۴ یک تابع به نام change_hue تعریف کرده‌ایم. این تابع پیکسل های ورودی را تنظیم می‌کنیم و به ۲ روش با آنها بنابر مقدار hue شان رفتار می‌کند. در ادامه بیشتر توضیح خواهد داده شد.

در کد این سوال ابتدا تصویر وارد می‌شود و با تابع img.shape مقادیر طول و عرض آن در w و h ذخیره می‌شود. سپس تصویر با تابع cv۲.cvtColor به فرمت رنگی hsv تغیر داده می‌شود و در hsv_image ریخته می‌شود. حال وارد حلقه یکی برای پیمایش در تصویر در جهت افقی و دیگری عمودی می‌شویم. در این دو حلقه فور، ۲ متغیر count و mid تعریف می‌کنیم و مقدار را ۰ قرار می‌دهیم. سپس ۲ عدد حلقه فور دیگر برای بدست آوردن مقدار میانگین value ماتریس ۵ در ۵ اطراف هر پیکسل قرار می‌دهیم و مقدار هر کدام را با قبلی ها جمع می‌کنیم. برای گوشه های تصویر نیز از try و except استفاده می‌کنیم که هرجا پیکسلی که می‌خواهیم خانه های اطرافش را میانگین بگیریم، در یکی از گوشه های تصویر قرار داشت و نتوانستیم ۲۵ پیکسل اطرافش را حساب کنیم، به ارور بر نخوریم و تنها آن پیکسل در محاسبه لحاظ نشود.

حال که با تقسیم مجموع پیکسل های اطراف بر تعداد آنها مقدار لازم برای جایگذاری در صورتی که بخواهیم از فیلتر blur ۵ در ۵ استفاده کنیم بدست آمد، تنها لازم است هر پیکسل را به تابع change_hue بدهیم و نتیجه آن را بجای پیکسل یاد شده جایگذاری کنیم. در ادامه با تبدیل تصویر به rgb آن را ذخیره کرده و با تبدیل به brg ، با استفاده از plt.imshow() آنرا نمایش می‌دهیم. زمان تقریبی برای اجرای برنامه به دلیل ۵ در ۵ بودن و دستی بودن فیلتر، حدود ۶۰ ثانیه است.

تابع مذکور change_hue یک پیکسل را دریافت می‌کند. آرگومان اول پیکسل یعنی hue را درون متغیر hue قرار می‌دهد. حال ۲ اتفاق اینجا امکان دارد رخ بدهد. یا hue این پیکسل بین دو عدد ۱۳۵ و ۱۷۵ است، یا نیست. اگر بین این دو عدد باشد، به این معنی است این پیکسل یکی از پیکسل های صورتی است. پس از مقدار آن ۵۶ واحد کم می‌شود که در محدوده ی تقریبی رنگ زرد قرار بگیرد. اگر بین ۱۳۵ و ۱۷۵ نباشد، طبق خواسته سوال، باید blur انجام شود. پس در بخش else ، می‌نویسیم مقدار value این پیکسل را برابر با مقدار میانگین value های کناره های پیکسل قرار بده که توسط آرگومان blue_value وارد این تابع شده است. پس آنرا جایگذاری می‌کنیم. سپس بعد از اتمام دستور شرطی، پیکسل بدست آمده را به عنوان خروجی قرار می دهیم که جایگزین پیکسل اولیه در تصویر شود.

پرسش ۵

در این سوال، همانطور که در صورت سوال گفته شده بود، به ۳ روش فیلتر جعبه ای را اعمال کردیم. ابتدا با روش تابع آماده cv۲.blur کتابخانه cv۲ این فیلتر را اعمال کردیم، سپس با استفاده از دو عدد حلقه که در هر کدام تک تک پیکسل ها بررسی می‌شوند و در مرکز یک ماتریس مربعی با اندازه فرد قرار می‌گیرند، سپس با آنها جمع شده و تقسیم بر تعداد کل همسایه‌هایش می‌شود و مقدار حاصل را به خود می‌گیرد. در روش دیگر با استفاده از کانولوشن ماتریسی این عمل را انجام داده‌ایم. یعنی اگر اندازه ماتریس مربعی را size بنامیم، با استفاده از الگوریتمی $size^2$ تصویر از تصویر اصلی ساخته می شود که به حالت های مختلفی کناره هایش حذف شده است. این حالت ها بسته به جایی است که داریم عمل جمع یکی از خانه های فیلتر را بر عکس اعمال می‌کنیم. برای مثال برای خانه ی گوشه ی سمت چپ بالا $size - 1$ ستون سمت راست و $size - 1$ سطر پایین را حذف می‌کنیم. سپس تک تک اینها را باهم جمع می‌کنیم و در نهایت تصویری بدست می آید که کاملا با تصویر دوم یکسان است و هیچ تفاوتی در آنها وجود ندارد.

حال به سراغ توضیح کد می‌رویم. ابتدا یک عکس به عنوان ورودی با استفاده از دستور cv۲.imread دریافت می‌شود. سپس مقدار size تعریف می‌شود که سایز فیلتر را مشخص می‌کند. در ادامه w و h تعریف می‌شوند که بیانگر عرض و ارتفاع عکس ورودی‌اند. (در انتهای و ابتدای هر روش یک خط کد برای گرفتن زمان اجرا نوشته شده است که به آن اشاره نمی‌کنیم.) حال می‌توانیم توضیحات را به توجه به ۳ الگوریتم مورد استفاده به ۳ بخش تقسیم کنیم.

۱. در این روش تنها به cv۲.blur عکس اولیه، اندازه فیلتر که همان size، (size) هست را می‌دهیم و خروجی ماتریسی حاصل را نیز بصورت دیتاتایپ $'uint8'$ دریافت می‌کنیم. سپس همانطور که گفته شد، با توجه به سایر روش ها از هر سمت به تعداد مقدار صحیح عدد $\frac{size}{2}$ ستون و سطر حذف می‌کنیم. در ادامه با استفاده از plt.imshow() و همزمان تبدیل فرمت رنگی به BGR آنرا نمایش می‌دهیم و با cv۲.imwrite آنرا ذخیره می‌کنیم.

۲. برای این روش ابتدا یک ماتریس خالی به نام blur_۲ با درایه های ۰ می‌سازیم. سپس دقیقا با همان روشی که در سوال ۴ مورد استفاده واقع شد، اما اینبار کل w و h پیمایش نمی‌شود و از هر طرف به مقدار $\frac{size}{2}$ کمتر پیش می‌رویم. برای همین نیازی به استفاده از try و except نیست و همواره همسایه ها کامل هستند؛ همچنین بجای مقدار value سه کانال r،g و b بطور جدا جمع و تقسیم می‌شوند. نتیجه نیز در blur_۲ در ۳ بعد که حاصل جمع این اعداد و تقسیم آنها در توان دو اندازه فیلتر است نگهداری می‌گردد. بعد از اینکه کار های گفته شده انجام شد و همه ی پیکسل های blur_۲ با مقدار دلخواه جایگزین شدند، مانند قبل از هر طرف $\frac{size}{2}$ به عنوان پیکسل های ۰ بریده می‌شوند و با دیتاتایپ $'uint8'$ در r_method۳ قرار می‌گیرند. سپس همزمان با تبدیل به BGR ، مانند قبل به نمایش در می‌آید که ذخیره می‌شود. با توجه به اینکه ۲ حلقه تمام پیکسل ها را باید بررسی و جایگذاری کند، انتظار می‌رود این روش بسیار وقت گیر باشد.

۳. برای روش سوم، ابتدا ۲ ماتریس تک بعدی با اندازه $size - 1$ به عنوان لیست سطر ها و لیست ستون ها در اختیار داریم که همه درایه های آن ها ۰ است. سپس یک تصویر خالی با درایه های ۰ می‌سازیم که از هر راستا مقدار $size - 1$ از آن کم شده است. حالا سایز این عکس سیاه جدید دقیقا برابر ۲ عکس قبلی است. دقت کنید که در اینجا از دیتا تایپ $float64$ استفاده شده است. این به دلیل هماهنگی با سایز عکس ها و عملیات غیر صحیح است. هرچند در انتها تصویر را پس از صحیح کردن آرایه هایش به $'uint8'$ بر می‌گردانیم.
در ادامه وارد ۲ حلقه تو در تو می‌شویم که به مقدار $size$ درون حلقه می‌چرخند. این یعنی $size^2$ بار دستورات درون حلقه داخلی را اجرا می‌کنیم که این دقیقا دلالت بر ساختن ۹ تصویر و جمع آنها با هم دارد. درون حلقه داخلی حلقه ای دیگر قرار دارد که به مقدار $size - 1$ در آن می‌گردیم. در هر کدام از این چرخش ها، به طرق مختلف آرایه های rows و columns پر می‌شوند. اینها حالات مختلف برای حذف ستون و سطر های بالا، پایین، چپ و راست تصویر هستند. این حالات توسط دو شرط و اعداد شمارش حلقه های اول و دوم و سوم و جمع و منهای آنها با ترتیب مناسب است که باعث می‌شود به سادگی این حالات در $size^2$ ساخته شوند. سپس rows و columns را که داریم، و می‌توانیم با دستور np.delete به سادگی سطر و ستون های مورد نظر را از جهات مختلف حذف کنیم. بعد از حذف آنها در دو دستور delete متوالی، حاصل col_deleted که تصویر نهایی ساخته شده است، به r_method۳ اضافه می‌گردد. پس از اینکه ۹ بار این چرخه تکرار شد و درای های همه ی $size^2$ تصویر یک به یک باهم جمع شدند، نتیجه را ابتدا با np.multiply بر $size^2$ تقسیم می‌کنیم، سپس با $np.floor$ تبدیل به

اعداد صحیح می‌شوند. در نهایت با np.array بصورت ماتریسی با فرمت ۸ بیتی $'uint8'$ ذخیره می‌گردد. در آخر هم طبق معمول به BGR تبدیل شده و نشان داده می‌شود و ذخیره می‌گردد.

در ادامه ی کد هم، ۴ زمان به دست آمده، بنابر ترتیبی مناسب از هم کم شده که زمان اجرای هر روش را بدست بیاوریم و نشان بدهیم. توجه شود که هر ۳ عکس دقیقا مشابه هم بوده و از لحاظ ظاهری هیچ تفاوتی ندارند، یعنی تصویر دوم سوم تک به تک درایه های برابری نیز دارند، اما تصویر اول به دلیل دارا بودن الگوریتمی خاص برای گوشه ها، چند پیکسل در گوشه ها با دو عکس دیگر متفاوت است که با تابع تفریق تصاویر نشان داده می‌شود، اما از لحاظ ظاهری، باز هم هیچ تفاوتی ندارند. ارقام بدست آمده برای زمان اجرا در حالت $size = 5$ (نتایج با این سایز فیلتر هم بوجود آمده است)، نیز به صورت زیر هستند:

$$First\ method : 0.326\ sec$$

$$Second\ method : 150.414\ sec$$

$$Third\ method : 2.858\ sec$$

پرسش ۶

برای این سوال، راه حل های متفاوتی در کتاب و ویدیو ها و همچنین داک openCv مشاهده کردم. اما اکثر آنها علیرغم سرعت بالا، نتیجه ی دقیقی را نمی‌دادند و به همین دلیل این مسئله به فکرم رسید که الگوریتم جدیدی هرچند طولانی و سنگین طراحی کنم که نتیجه آن دقیقا با خواسته مسئله یکی باشد. یعنی هیستوگرام های دو تصویر کاملا بر هم منطبق شوند و هیچ کم و کاستی در آن وجود نداشته باشد. سپس پیاده سازی و عیب یابی شد و زمان اجرا کاهش شدیدی یافت، چرا که با اعداد بزرگی سر و کار داریم که همین مسئله با جابجایی ماتریس ها سرعت اجرا را کم می‌کند. اما سعی شد تا جای ممکن از استفاده از حلقه ها و سایر مواردی که باعث کم شدن سرعت می‌شود اجتناب شود. در نهایت زمان اجرا به زیر ۲ دقیقه رسید و البته با استفاده از الگوریتم های میانبر می‌توان آنرا کمتر هم کرد.

تابع plot_histogram در ابتدای برنامه، یک تصویر را دریافت می‌کند و با دستورات معمولی پلات موجود در کتاب و داک، ۳ کانال رنگی آنرا پلات می‌کند. در ادامه نتایج حاصل از پلات هیستوگرام های مربوط آورده می‌شود. اما این تابع نیاز به بررسی زیادی ندارد. پس به سراغ کد اصلی می‌رویم. در ابتدا یک تایمر گذاشته ایم که زمان اجرا را اندازه بگیریم. سپس تصویر Dark را وارد برنامه کرده و طول و عرض آنرا در متغیر های w_dark و h_dark قرار می‌دهیم. سپس عکس دوم یعنی Pink را وارد برنامه کرده و اینجا با دستور cv۲.resize ، طول و عرض عکس دوم را به طول و عرض عکس اول تغییر می‌دهیم. با اینکار هیستوگرام هیچ تغییری نمی‌کند اما کارمان را برای اجرای الگوریتم مورد نظر بسیار ساده می‌کند. حال که متغیر های شروع و تصاویر را تعریف کردیم، وارد بخش اصلی کد می‌شویم.

بخش اصلی کد از اینجا شروع می‌شود که یک حلقه با دامنه ۰ تا ۲ تعریف کردیم و وارد این حلقه می‌شویم. در اینجا ۳ بار می‌چرخیم که ۳ کانال رنگی عکس اول را به ترتیب با ۳ کانال رنگی عکس دوم یکسان سازی کنیم. ابتدا یک آرایه تک بعدی difference با طول ۲۵۶ تعریف می‌کنیم. در این آرایه قرار است اختلاف پیکسل های تصویر اول با تصویر دوم را در خود نگهدارد. سپس آرایه ای به نام Mapper تعریف می‌کنیم که متشکل از ۲۵۶ لیست است. در هرکدام از این لیست ها قرار است شماره های پیکسل هایی با روشنایی متناظر عدد لیست را در خود نگهدارد.

سپس یک دو حلقه ی فور تو در تو تشکیل می‌شود که در رنج ۰ تا طول و ۰ تا عرض هستند. یعنی تمام پیکسل های تصویر پیمایش می‌شوند. در باری که حلقه ی داخلی تر اجرا می‌شود ۳ تا اتفاق رخ می‌دهد.

۱. ابتدا در صورتی که یک پیکسل مورد پیمایش در تصویر Dark مقدار x داشته باشد، به درایه ی شماره x آرایه ی difference یک واحد افزوده می‌گردد. زیرا یک واحد روشنایی عملا بین تصویر اول و دوم اختلاف افتاده است.

۲. سپس در صورتی که یک پیکسل مورد پیمایش در تصویر Pink مقدار x داشته باشد، از درایه ی شماره x آرایه ی difference یک واحد کم می‌گردد. زیرا یک اختلاف روشنایی بین تصویر اول و دوم یک واحد کم شده است.

۳. هر پیکسلی که در تصویر Dark پیمایش می‌شود، مقدار $(j * w_dark) + i$ که یک نقشه از پیکسل ورودی با روشنایی خاص است.

پس از مقدار دهی به دو آرایه ی مذکور، وارد حلقه ی عملیاتی اصلی می‌شویم. این حلقه از ۰ تا ۲۵۴ پیمایش می‌کند و عمل آن تنظیم پیکسل هایی با روشنایی ۰ تا ۲۵۵ در جای درست است که دو تصویر هماهنگ باشند. ابتدا برای افزایش سرعت، لیست شماره i آرایه Mapper ، بصورت عدد صحیح در ماتریس mat قرار می‌گیرد. در اینجا ۲ اتفاق دارد رخ بدهد که با شرط مدیریت شده‌اند.

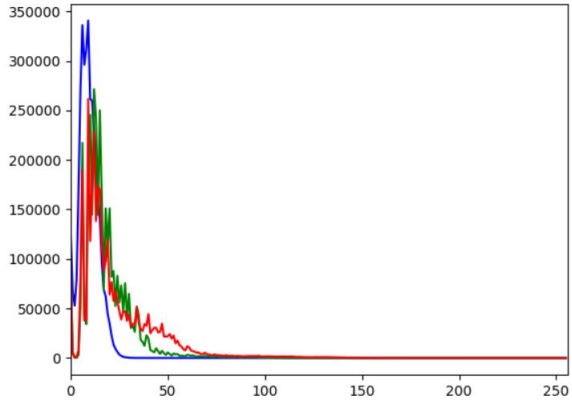
۱. مقدار اختلاف کمتر از طول ماتریس باشد. در این حالت نباید کل پیکسل های لیست را در لیست بعدی قرار داد. بلکه از ۰ تا اندازه اختلاف از لیستی که داریم را می‌بریم و با دستور extend در لیست بعدی می‌گذاریم. در مرحله بعدی بطور ماتریسی جای کل اعضای لیستی که مانده را به استفاده از خارج قسمت و باقی مانده، جایگاه اعدادی که به لیست بعد نرفته‌اند را پیدا می‌کنیم و مقدار پیکسل پیدا شده در کانال مورد نظر را برابر با عدد i+۱ می‌کنیم، زیرا از آنجا که تا ۲۵۴ پیشروی می‌کنیم، لازم است ۲۵۵ نیز در انتهای حلقه با مقدار باقی مانده پر شود. سپس مقدار لیست را خالی می‌کنیم که محاسبات بعدی با سرعت بیشتری انجام شود. سپس از آنجا که به مقدار درایه ی i ام difference به روشنایی بعدی انتقال یافته است، همان مقدار با به درایه ی بعدی difference اضافه

می‌کنیم. چراکه اختلاف روشنایی را بیشتر کرده‌ایم. و در نهایت نیز درایه i ام difference هم برابر با ۰ قرار می‌گیرد.

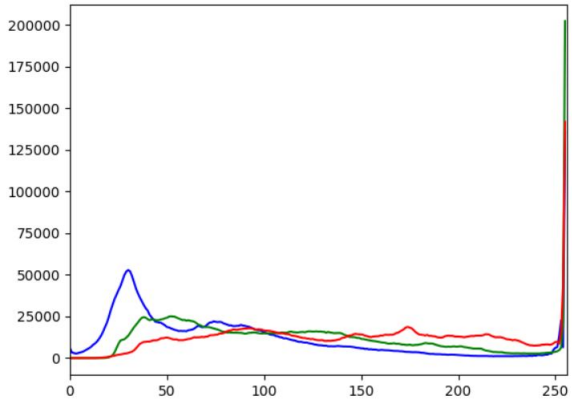
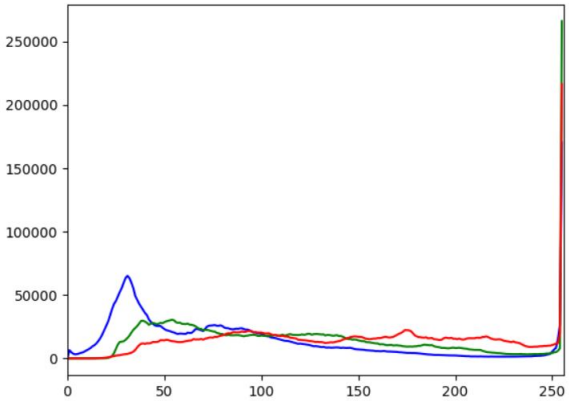
۲. در حالت دیگر هم mat از اختلاف در روشنایی i کوچکتر یا مساوی با آن است. در این حالت کل mat وارد لیست بعدی می‌شود، بازهم جای پیکسل مورد نظر مشخص می‌شود و مقدار دهی می‌گردد. سپس طول mat به درایه ی بعدی در آرایه جمع می‌شود. چون اختلاف در روشنایی بعدی به همان مقدار بیشتر شده است. لیست خالی شده و difference در درایه i هم ۰ می‌شود. یک دستور print برای فهمیدن مرحله ی پیشروی هم نوشته شده است.

کل مراحل بالا ۳ بار برای هر کانال تکرار می‌شود و در نهایت هیستوگرام تصاویر Dark و Pink اصلی رسم می‌شود و تصویر Dark ذخیره می‌شود. تصاویر گفته شده در فایل ها قرار دارند اما تصاویر زیر هم دید کلی نسبت به نتیجه ی این برنامه به ما خواهد داد:

هیستوگرام تصویر اولی Dark



هیستوگرام تصویر Pink



هیستوگرام تصویر Dark نهایی