«*In The Name Of GOD*»

[HW-01-Report]

[DIGITAL IMAGE PROCESSING]

Hasan Masroor | [403131030] | March 16, 2025

"فهرست مطالب تمرين 01"

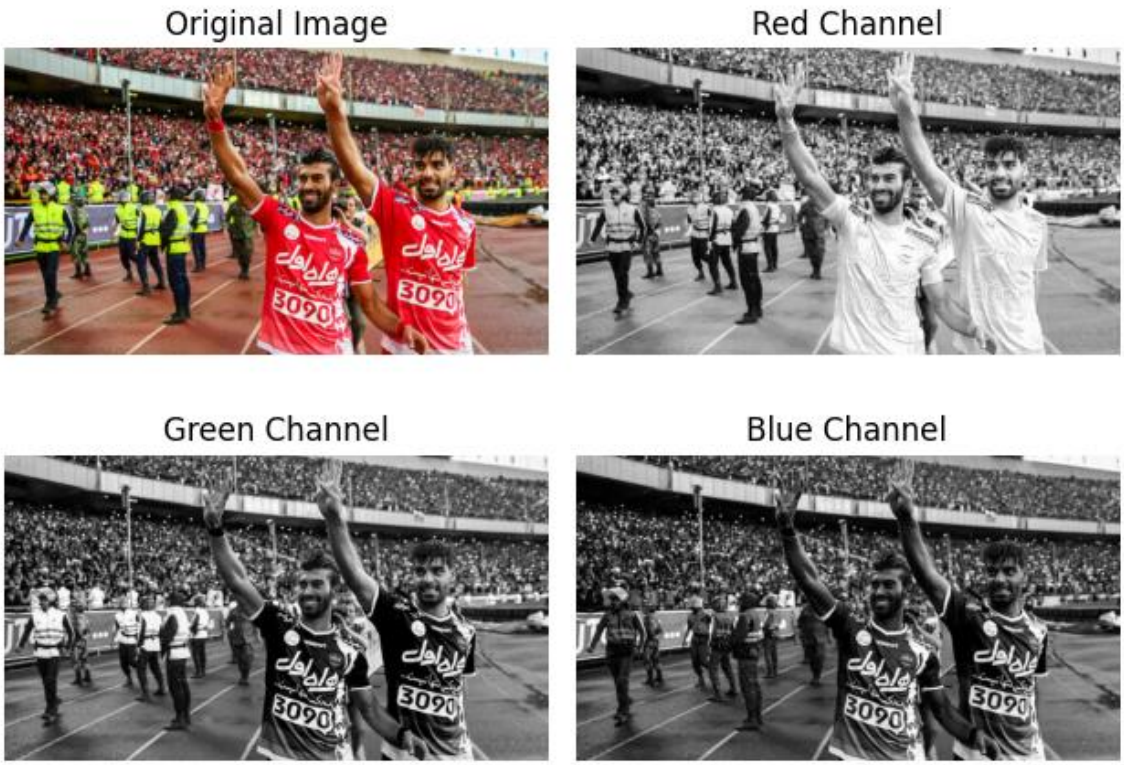
Problem 1: Exploring Color Spaces and Image Transformation

A.

در این بخش باید تصویر داده شده را آپلود کنیم و بعد از آن هر یک از کانال‌های رنگی آن شامل قرمز، سبز و آبی را نمایش دهیم. در ابتدا عکس را با کتابخانه OpenCV باز می‌کنیم، از آن‌جایی که فرمت اولیه عکس به صورت BGR است پس آن را به فرمت RGB تبدیل می‌کنیم. بعد از این مرحله با استفاده از image[:, :,]به استخراج کانال‌های رنگی می‌پردازیم (دو بعد اول ارتفاع و عرض را مشخص می‌کند و بعد سوم نیز کانال رنگی را مشخص می‌کند) و می‌دانیم در RGB کانال قرمز مقدار 0، کانال سبز مقدار 1 و کانال آبی هم مقدار 2 را برای بعد سوم می‌گیرد.

در این بخش از روی تصویر ورودی سه کانال رنگی را استخراج کردیم تا تأثیر هر کانال در تشکیل تصویر اصلی بررسی کنیم؛ در نهایت نیز خروجی را که شامل تصویر اصلی و هر یک از کانال‌های رنگی است نمایش می‌دهیم:

حالا برای تجزیه و تحلیل بهتر اگر مجدد تصویر بالایی رو نمایش دهیم اما این بار cmap این سه کانال اصلی RGB را با رنگ gray چاپ کنیم به تصویر زیر می‌رسیم که طبق مثال‌های کلاس درس برای این نمایش می‌توانیم به خوبی تاثیر این کانال‌ها روی تصویر اصلی را مشاهده کنیم؛ در هر تصویر مربوط به کانال‌های رنگی هر قسمتی از تصویر که رنگ روشن‌تری نسبت به سایر قسمت‌های تصویر داشته باشد به رنگ همان کانال در می‌آید به عنوان در تصویر مربوط به کانال قرمز مشاهده می‌کنیم که لباس بازیکن‌ها رنگ روشن‌تری نسبت به قسمت‌های دیگر دارند پس به رنگ همین کانال یعنی قرمز در می‌آیند و یا در کانال سبز لباس‌های کادر انتظامی روشن‌تر هست و به رنگ سبز در می‌آیند و به همین منوال می‌توانیم موارد دیگر را در تصویر زیر ببینیم:



B.

برای جابجایی دو کانال قرمز و آبی می‌توانیم از cv2 کمک بگیریم، راه دوم هم این است که مستقیما و با روش دستی جای این دو کانال رو عوض کنیم و مثلا با image_B2R[:, :, 2] = image[:, :, 0] بخش‌های قرمز در تصویر اصلی، در تصویر جدید آبی خواهند شد.

در نهایت نیز تصویر اصلی و تصویری که کانال قرمز و آبی با هم جابجا شده‌اند را در خروجی نشان می‌دهیم:

همان‌طور که در تصویر بالا مشاهده می‌کنیم پس از اینکه کانال آبی را با کانال قرمز و کانال قرمز را با کانال آبی تعویض کردیم تأثیرات بصری قابل توجهی در تصویر داشتیم، در حالت معمول در دنیای واقعی هرکدام از اشیاء و بخش‌های مختلف یک تصویر رنگ‌های خاصی را به تناسب نور، شرایط محیطی و ... به خود اختصاص می‌دهند اما با این تغییری که در این پارت ایجاد کردیم؛ با توجه به به جابجایی رنگ‌ها تصویر از ظاهر طبیعی‌اش خارج می‌شود. همچنین مشاهده می‌کنیم که قسمت‌های قرمز در تصویر اصلی به رنگ آبی دیده می‌شوند و مثلا رنگ پیراهن دو بازیکن تیم پرسپولیس و ... به رنگ آبی در آمده است؛ همچنین این جابجایی باعث می‌شود که قسمت‌های آبی‌رنگ در تصویر اصلی به رنگ قرمز در بیایند. کاری که در این پارت انجام دادیم باعث می‌شود که تصویر از حالت طبیعی خود خارج شود و حالت غیرواقعی و غیرطبیعی پیدا کند

C.

تصاویر را می‌توان با نمایش‌های مختلفی از جمله RGB که در بخش‌های قبل نشان دادیم نمایش داد. نمایش‌های دیگری مثل HSV، Grayscale و ... نیز وجود دارند که هریک از آن ها به روش متفاوتی به نمایش رنگ‌ها می‌پردازند و در ادامه این موارد را هم پیاده‌سازی می‌کنیم.

در بیشتر اوقات برای تجزیه و تحلیل تصاویر در کاربردهای مختلف نیازی به یک تصویر رنگی نداریم و تنها داشتن یک تصویر خاکستری هم کافی است. همان‌طور که قبلا هم اشاره کردیم یک تصویر RGB شامل سه ماتریس برای کانال‌های رنگی قرمز، سبز و آبی است و حال می‌توانیم یک تصویر RGB برای تبدیل به Grayscale استفاده کنیم و یک روش ساده و کوتاه استفاده از دستور cv2.COLOR_RGB2GRAY است که به طور خودکار این تبدیل انجام می‌شود. روش دوم استفاده از فرمول **I=0.299R+0.587G+0.114B** است و کانال قرمز را در ضریب 0.299، کانال سبز در ضریب 0.587 و کانال آبی را در ضریب 0.114 ضرب می‌کنیم و در نهایت هم تصویر اصلی و تصویر خاکستری را در خروجی نمایش می‌دهیم:

برای تبدیل تصویر به خاکستری روش‌های دیگری مثل موارد زیر نیز وجود دارند:

I. **I = (Red + Green + Blue) / 3**
II. **I = (Red + Green + Blue) * 0.33**

در دو فرمول بالا ما از میانگین ساده استفاده کردیم به این صورت که سه کانال قرمز، سبز و آبی را با هم جمع کردیم و بعد یا حاصل جمع را یا بر 3 تقسیم کردیم و یا در عدد 0.33 که همان 1/3 می‌باشد ضرب کنیم و به همه کانال‌ها وزن برابر و یکسانی دادیم و هیچ تفاوت و تمایزی بین کانال‌ها قائل نشدیم، اما در فرمول قبلی که در کد برای پیاده‌سازی هم از آن استفاده کردیم از میانگین وزن‌دار استفاده کردیم و به توجه به حساسیت چشم انسان به رنگ‌ها، به هر کانال ضریب یا وزن متفاوتی دادیم و مثلا چشم انسان حساسیت بیشتری به رنگ سبز دارد و از همین رو کانال سبز وزن 0.587 را گرفت و به همین منوال برای دو کانال دیگر هم وزن مربوطه را اختصاص دادیم که تصویر خاکستری شکل طبیعی‌تری داشته باشد.

استفاده از فرمول میانگین وزن‌دار برخلاف میانگین ساده که برای هر کانال وزن یکسانی را در نظر می‌گیرد، با اختصاص وزن‌های درست به هر کانال باتوجه به حساسیت چشم انسان باعث می‌شود این تبدیل از یک تصویر رنگی به تصویر خاکستری به طور صحیح‌تر انجام شود و یکسری جزئیات رنگی مهم نیز نادیده گرفته نشود و به طور کلی تبدیل بهتری انجام شود.

D.

در این پارت از ما سوال شده است که آیا تبدیل RGB به خاکستری همیشه ممکن است. نکاتی را در مورد پیامدهای این تبدیل و هرگونه از دست دادن اطلاعات ارائه دهیم.

پاسخ به این سوال بله است و تبدیل تصویر از RGB به تصویر Grayscale همیشه ممکن است و با استفاده از همان فرمول میانگین وزن‌داری که بالاتر به آن اشاره کردیم می‌توان به سادگی هر تصویر رنگی RGB را به Grayscale تبدیل کرد اما این تبدیل عواقبی دارد و می‌تواند به از دست دادن یکسری اطلاعات هم منجر شود.

در این تبدیل اطلاعات مربوط به رنگ‌های سه کانال قرمز ، سبز و آبی به یک شدت روشنایی تبدیل می‌شوند و در نتیجه اطلاعات مربوط به رنگ‌ها که برای تجزیه و تحلیل دقیق‌تر و استخراج ویژگی‌های خاص تصویر مورد

استفاده قرار می‌گیرد را از دست می‌دهیم، مثلا در کاربردهایی مانند تشخیص اشیا بر اساس رنگ، این تبدیل ممکن است مناسب نباشد؛ اما از آن طرف چون تصویر خاکستری تنها یک کانال دارد در حالی که تصویر RGB سه کانال دارد و این باعث کاهش حجم داده‌ها و ساده‌تر شدن پردازش می‌شود، مثلا برای تشخیص لبه‌ها، فیلتر کردن یا پردازش‌های مبتنی روی شدت روشنایی، این تبدیل بسیار مفید است.

E.

حالا باید تصویر RGB را به HSV تبدیل کنیم و هر کانال مربوط به HSV را نمایش دهیم. HSV شامل سه کانال Hue، Saturation و Value می‌باشد که در ادامه به توضیح هر یک می‌پردازیم:

- **Hue**: این کانال رنگ اصلی تصویر را نشان می‌دهد که به صورت زاویه‌ای در دایره رنگی بین 0 تا 360 قرار می‌گیرد. به عنوان مثال قرمز 0 درجه، سبز 120 درجه و آبی 240 درجه است و بین این درجات هم سایر رنگ‌های دیگر قرار می‌گیرند.
- **Saturation**: در مورد شدت و وضوح رنگ به ما اطلاعات می‌دهد و بین 0 تا 100 قرار دارد و بخش‌هایی که مقدار اشباع بالاتر باشد، رنگ‌ها به وضوح و زنده قابل مشاهده هستند و بخش‌هایی که اشباع کمتری داشته باشند کمرنگ و خاکستری دیده می‌شوند.
- **Value**: میزان روشنایی رنگ را نشان می‌دهد و مشخص می‌کند که رنگ چه‌قدر روشن یا تاریک است. هرچه مقدارش بالاتر باشد تصویر روشن‌تر خواهد بود و به تبع هرچه مقدارش کم باشد رنگ تاریک تر خواهد بود.

همان‌طور که توضیح دادیم RGB از سه کانال رنگی قرمز، سبز و آبی تشکیل شده است اما HSV اطلاعات بهتری راجع به رنگ اصلی تصویر، شدت رنگ‌ها و میزان روشنی به ما می‌دهد و از طرفی می‌توانیم به طور مستقل این سه کانال را برررسی کنیم و در پردازش تصویر، تنظیم رنگ‌ها و ویژگی‌های مختلف آنها کمک زیادی می‌کند. به عنوان مثال کانال Hue مستقل از شدت رنگ و روشنایی به تحلیل تصاویر بدون تاثیر نور یا ... کمک می‌کند و یا اینکه برای شناسایی یک رنگ در تصویر بدون اینکه تغییرات روشنایی روی نتیجه تاثیر بگذارد استفاده کنیم. همچنین برای شناسایی اشیاء در تصاویر کابردهای گسترده‌ای دارد.

با تفکیک دقیق رنگ‌ها و ویژگی‌های آن‌ها درHSV می‌توانیم تغییرات دقیق‌تری در تصاویر اعمال کنیم:

تنظیمات رنگ و کنتراست: با تغییر هرکدام از کانال‌های HSV به‌طور مستقل می‌توانیم تغییرات دقیق و غیر مخربی در تصویر اعمال کنیم؛ مثلا در صورتی که تصویر خیلی روشن باشد می‌توانیم فقط اشباع رنگ‌ها را کاهش دهیم تا شدت رنگ‌ها کمتر شود، در حالی که روشنایی تصویر ثابت بماند.

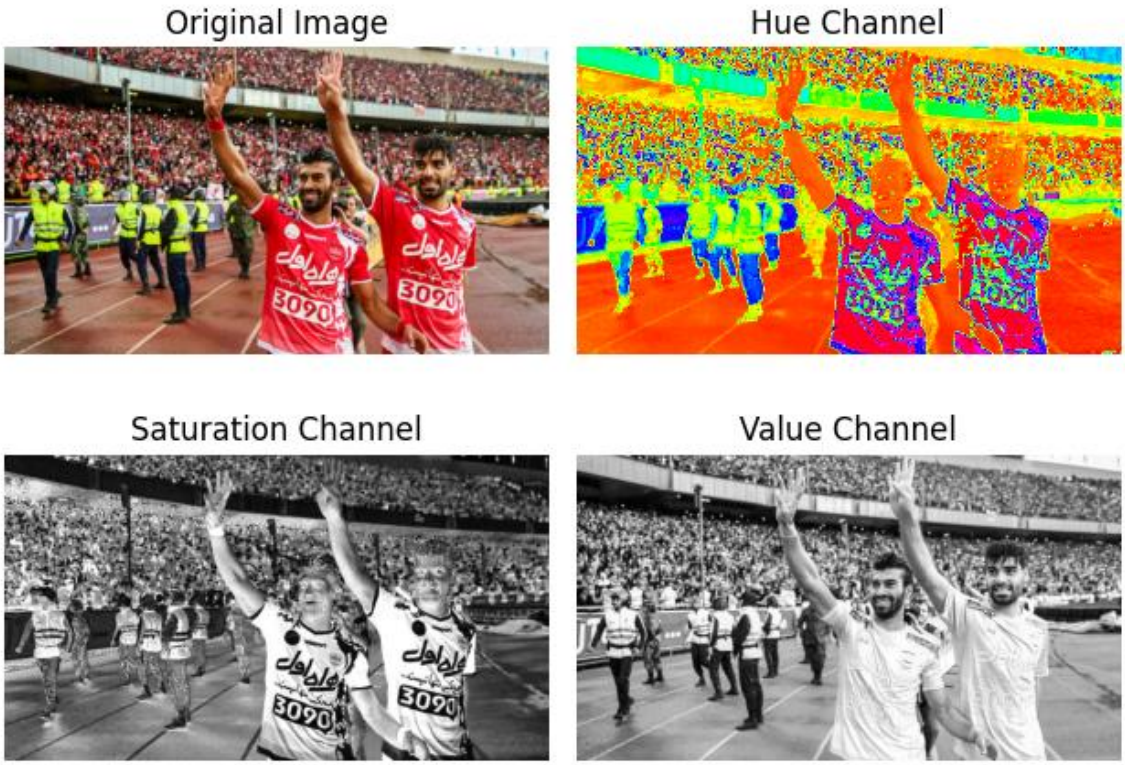
تغییر رنگ: برای تغییر رنگ‌ها بدون تأثیر بر روشنایی یا شدت رنگ‌ها تنها کافی است که کانال Hue را تغییر دهیم؛ به این ترتیب می‌توانیم رنگ‌های یک تصویر را به دقت تغییر دهیم .

حذف رنگ‌های ناخواسته: در صورت نیاز به حذف رنگ‌های خاص از تصویر می‌توان از Hue برای شناسایی رنگ مورد نظر و از Saturation برای افزایش یا کاهش شدت آن استفاده کرد. این کار به ما این امکان را می‌دهد که به‌طور دقیق بخش‌های دلخواه را نگه داریم و یا کارهای دیگری که می‌خواهیم روی تصویر اعمال کنیم.

نمایش HSV در پردازش تصویر به ما این امکان را می‌دهد که رنگ‌ها، اشباع و روشنایی را به‌طور مستقل از یکدیگر بررسی کنیم و یا تغییر دهیم. این ویژگی‌ها برای کاربردهایی مثل تحلیل رنگ‌ها، ویرایش دقیق تصاویر، تنظیم کنتراست و روشنایی و حذف یا تغییر رنگ‌ها بسیار مفید است و به تحلیل دقیق‌تر و دستکاری ساده‌تر تصاویر کمک می‌کند و در پردازش‌های گرافیکی و تصاویر اهمیت زیادی دارد.

برای پیاده‌سازی این بخش از cv2.COLOR_RGB2HSV استفاده می‌کنیم تا تصویر را به HSV تبدیل کنیم؛ حال برای آنکه هر کانال را جدا کنیم (مثل نمایش کانال‌های RGB در پارت‌های قبل) به عنوان راه اول می‌دانیم در HSV، Hue مقدار 0 دارد و با image_hsv[:, :, 0] می‌توانیم به آن دسترسی داشته باشیم و همینطور Saturation مقدار 1 و Value هم مقدار 2 دارد و این‌گونه هر سه کانال را جدا می‌کنیم. راه دوم به جای این کار می‌توانستیم به طور مستقیم از Hue, Saturation, Value = cv2.split(image_hsv) استفاده کنیم که این کار هم می‌آید سه کانال را مثل روش قبلی جداجدا بدست می‌آورد.

در نهایت نیز تصویر اصلی را به همراه تصاویر این سه کانال HSV در خروجی نمایش می‌دهیم:



همانطور که گفتیم Hue اطلاعات مربوط به رنگ‌ها را نشان می‌دهد و نمایانگر موقعیت رنگ‌ها در دایره رنگی می‌باشد؛ پس رنگ‌هایی که اثر قوی‌تری دارند برجسته‌تر می‌شوند. این کانال برای شناسایی و تمایز رنگ‌های مختلف کمک خوبی می‌کند. در تصویر بالا کانال Saturation اطلاعاتی در مورد شدت و وضوح رنگ‌ها به ما می‌دهد و مثلا رنگ پیراهن بازیکن‌ها و ... به وضوح قابل مشاهده هستند و بخش‌هایی که اشباع کمتری دارند هم کمرنگ دیده می‌شوند. کانال Value هم میزان روشنایی تصویر را نشان می‌دهد و اینکه چه قسمت‌هایی روشن و چه قسمت‌هایی تاریک‌تر اند، مثلا پوست بازیکن‌ها یا پیراهن و ... روشن‌تر به نظر می‌رسند.

.F

فضای YCbCr یکی از فضاهای رنگی است که عمدتاً برای کاربردهایی مثل فشرده‌سازی تصویر و پخش تلویزیونی استفاده می‌شود و متشکل از سه کانال است که در ادامه هر کدام را توضیح می‌دهیم:

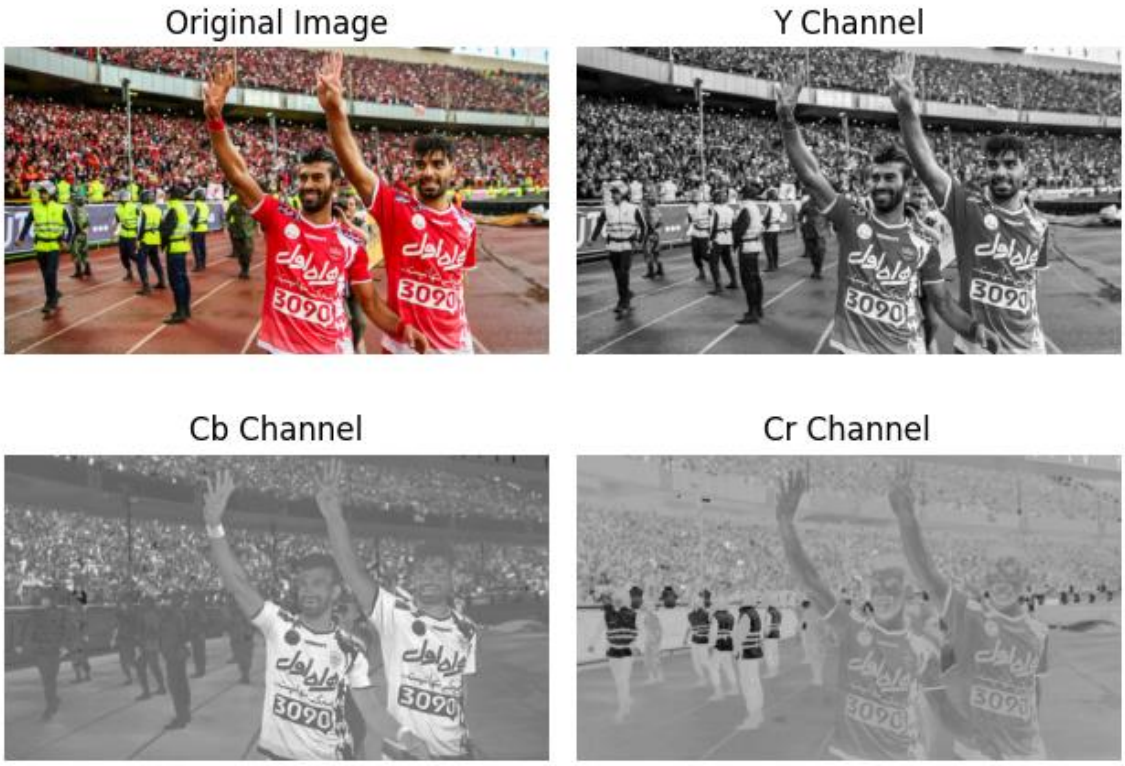
- **Y**: کانال روشنایی یا Luminance که اطلاعات مربوط به روشنایی پیکسل را نشان می‌دهد و همیشه بدون فشرده‌سازی حفظ می‌شود اما Cb و Cr را می‌توان فشرده کرد.
- **Cb**: کانال Chrominance Blue که تفاوت رنگ آبی نسبت به روشنایی را نشان می‌دهد و نشان می‌دهد تصویر چه‌قدر آبی‌تر از مقدار عادی روشنایی است، هرچه مقدار Cb بیشتر باشد نشان‌دهنده آبی‌تر بودن رنگ‌هاست.
- **Cr:** کانال تفاوت Chrominance Red که تفاوت رنگ قرمز نسبت به روشنایی را نشان می‌دهد و هرچه مقدار این Cr بیشتر باشد یعنی رنگ‌های قرمز بیشتری در تصویر وجود دارند.

جداسازی YCbCr به سه کانال Y، Cb و Cr این امکان را به ما می‌دهد که پردازش تصویر بهینه‌تری داشته باشیم و مثلا برای فشرده‌سازی می‌توان دقت رنگ‌ها را با تغییر کرومیننس آبی و قرمز کمتر کرد بدون اینکه روی کاهش کیفیت تصویر تاثیر زیادی بگذارد (به این خاطر که چشم انسان نسبت به تغییرات روشنایی تا تغییرات رنگ بیشتر حساس است). مدل YCbCr به ویژه در پخش تلویزیونی و فشرده‌سازی تصاویر بسیار مهم است. دلیل اصلی این است که این مدل به ما اجازه می‌دهد که روشنایی و رنگ‌ها را به صورت جداگانه ذخیره و پردازش کنیم. به عبارت دیگر با استفاده از این مدل می‌توانیم کیفیت تصویر را بدون تاثیر زیاد بر روشنایی کاهش دهیم که در مواردی مثل پخش تلویزیونی و فشرده‌سازی تصویر به‌ویژه در باندهای محدود بسیار کارآمد است و در فشرده‌سازی‌های ویدیویی و تصویری مثل JPEG و MPEG استفاده می‌شود.

در RGB که متشکل از سه کانال رنگی قرمز، سبز آبی می‌باشد، همه اطلاعات مربوط به رنگ‌ها و روشنایی در هر سه کانال ذخیره می‌شوند و این باعث می‌شود که در فشرده‌سازی یا انتقال داده‌های تصویری، ذخیره‌سازی اطلاعات روشنایی و رنگ به طور یکسان انجام شود که بهینه نیست اما در YCbCr، به خاطر اینکه اطلاعات روشنایی از رنگ جدا هستند می‌توان به‌طور جداگانه بر روی هرکدام از این ویژگی‌ها تمرکز و پردازش کرد. (این جدا کردن ویژگی‌ها برای فشرده‌سازی و کاهش حجم داده‌ها بسیار مفید است و همچنین چشم انسان نسبت به تغییرات در رنگ‌ها نسبت به تغییرات روشنایی حساسیت کمتری دارد و می‌توان اطلاعات رنگ‌ها را با دقت کمتری ذخیره کرد.)

در این پارت می‌خواهیم که تصویر RGB را به فضای YCbCr ببریم و هر کانال را جداگانه نمایش دهیم. مجدد از دستور cv2.COLOR_RGB2YcrCb برای تبدیل تصویر رنگی استفاده می‌کنیم و مثل پارت قبلی با کمک cv2.split() هر کانال را جدا می‌کنیم، البته به عنوان راه دوم هم می‌توانیم برای هر کدام از کانال‌های Y، Cb و Cr از فرمولی که داخل کد کامنت شده است برای این تبدیل استفاده کنیم.

در نهایت تصویر اصلی را به همراه تصویر هر کدام از این سه کانال به طور جداگانه در خروجی نمایش می‌دهیم:



G.

❖ RGB

- **مزایا:**

فضای رنگی RGB یکی از ساده‌ترین و پرکاربردترین مدل‌های رنگی است که به راحتی قابل درک و استفاده است. این فضا بر اساس سه رنگ اصلی قرمز، سبز و آبی ساخته شده و به خوبی با نمایشگرهای دیجیتال مانند کامپیوترها، تلویزیون‌ها و دوربین‌ها هماهنگ است. یکی از مزایای اصلی این فضا هم‌راستایی آن با سخت‌افزارهای دیجیتال است، زیرا بیشتر صفحه‌نمایش‌ها و دوربین‌ها به طور بومی از این مدل رنگی برای نمایش و ضبط تصاویر استفاده می‌کنند. (این ویژگی باعث می‌شود RGB برای کاربردهای روزمره مانند نمایش تصاویر و ویدیوها بسیار مناسب باشد.)

- **معایب:**

با این حال فضای RGB معایبی نیز دارد. یکی از مهم‌ترین معایب آن تداخل رنگ و روشنایی است. در این مدل رنگ‌ها به‌طور مستقیم با میزان روشنایی در تعامل هستند، بنابراین تغییر روشنایی تصویر ممکن است باعث تغییر در رنگ‌ها نیز شود (این موضوع می‌تواند در پردازش تصویر مشکل‌ساز باشد، به‌ویژه زمانی که نیاز به جداسازی رنگ و روشنایی وجود دارد). علاوه بر این، در برخی کاربردها مانند تشخیص اشیاء یا تجزیه و تحلیل رنگ‌ها مدل RGB ممکن است محدودیت‌هایی ایجاد کند، زیرا نمی‌توان به‌طور مستقل رنگ‌ها و روشنایی را دستکاری کرد؛ این محدودیت‌ها باعث می‌شود در برخی موارد فضاهای رنگی دیگر مانند HSV یا YCbCr ترجیح داده شوند.

- **مثال‌های کاربرد:**

همانطور که گفتیم فضای رنگی RGB به‌طور گسترده‌ای در نمایش تصاویر دیجیتال استفاده می‌شود و در نمایشگرهای کامپیوتری، تلویزیون‌ها و ... کاربرد دارد. این فضا به دلیل سادگی و سازگاری با سخت‌افزارهای دیجیتال، برای نمایش رنگ‌ها و جزئیات تصاویر ایده‌آل است. همچنین در پردازش تصویر، بیشتر ابزارها از RGB برای تبدیل تصاویر از فرمت‌های مختلف استفاده می‌کنند، زیرا این مدل رنگی با اکثر الگوریتم‌های پردازش تصویر سازگار است؛ این ویژگی‌ها باعث می‌شود RGB برای کاربردهای عمومی و روزمره بسیار مناسب باشد.

❖ HSV

- **مزایا:**

فضای رنگی HSV به دلیل تفکیک به سه کانال مجزای رنگ، روشنایی و اشباع، مزایای قابل توجهی دارد. این ویژگی اجازه می‌دهد تا رنگ‌ها را بدون تأثیرپذیری از تغییرات روشنایی یا سایه‌ها تحلیل کنیم که برای کاربردهایی مانند تشخیص رنگ‌های خاص بسیار مفید است، مثلا در تشخیص اشیاء بر اساس رنگ (مانند میوه‌های رسیده)، فضای HSV به دلیل جداسازی واضح رنگ‌ها در کانال Hue عملکرد بهتری نسبت به RGB دارد. همچنین این فضا برای پردازش‌های هنری که نیاز به تغییر رنگ‌ها بدون تأثیر بر روشنایی یا اشباع دارند نیز مفید است.

- **معایب:**

در کنار مزایایی که گفتیم فضای HSV معایبی نیز دارد. تبدیل تصاویر از RGB به HSV معمولاً زمان‌برتر است و ممکن است برای پردازش‌های پیچیده‌تر که نیاز به ترکیب ویژگی‌ها دارند چالش‌برانگیز باشد. علاوه بر این بیشتر نمایشگرها و دستگاه‌های دیجیتال از فضای RGB استفاده می‌کنند، بنابراین برای نمایش تصاویرHSV تبدیل آن‌ها به RGB ضروری است. این فرآیند گاهی می‌تواند باعث افت کیفیت یا از دست رفتن جزئیات شود؛ با این وجود برای کاربردهایی که نیاز به تحلیل دقیق رنگ‌ها دارند HSV همچنان یک انتخاب بهتر و هوشمندانه است.

- **مثال‌های کاربرد:**

فضای رنگی HSV برای پردازش و ویرایش رنگ‌ها بسیار کاربردی است، به عنوان مثال در تغییر رنگ یک شی یا فیلتر کردن رنگ‌های خاص این فضا به دلیل تفکیک واضح رنگ‌ها در کانال Hue عملکرد بهتری

دارد. همچنین در شناسایی اشیاء بر اساس رنگ (مانند تشخیص میوه‌های رسیده یا اشیاء رنگی در محیط) فضای HSV به دلیل جداسازی بهتر رنگ‌ها از روشنایی و اشباع بسیار مفید است و این ویژگی‌ها باعث می‌شود HSV برای کاربردهای مبتنی بر تحلیل رنگ انتخاب خیلی خوبی باشد.

❖ YCbCr

- **مزایا:**

فضای رنگی YCbCr مزایای زیادی دارد به‌ویژه در فشرده‌سازی تصاویر و ویدیوها. این فضا به دلیل تفکیک روشنایی (Y) از اطلاعات رنگ‌ها (Cb) و(Cr) امکان کاهش حجم داده‌ها را بدون تأثیر چشمگیر بر کیفیت تصویر فراهم می‌کند؛ به همین دلیل YCbCr در فرمت‌های فشرده‌سازی مانند JPEG و MPEG به‌طور گسترده استفاده می‌شود. همچنین این تفکیک روشنایی از رنگ‌ها، پردازش و ویرایش تصاویر را بدون تأثیر بر روشنایی تسهیل می‌کند.

- **معایب:**

فضای YCbCr علاوه بر مزایایی که بالاتر به آن اشاره کردیم معایبی نیز دارد. هنگام فشرده‌سازی تصاویر ممکن است دقت رنگ‌ها کاهش یابد به‌ویژه اگر از فشرده‌سازی‌های شدید استفاده شود، علاوه بر این این فضا در مقایسه با RGB پیچیده‌تر است و برای افرادی که با پردازش تصویر آشنایی کمتری دارند، ممکن است چالش‌برانگیز به نظر برسد. به همین دلیل، YCbCr معمولاً برای کاربردهای ساده‌تر کمتر استفاده می‌شود اما برای کاربردهای حرفه‌ای مانند فشرده‌سازی و پردازش ویدیو انتخاب مفیدی است.

- **مثال‌های کاربرد:**

فضای رنگی YCbCr همان‌طور بالا به آن اشاره کردیم کاربردهای گسترده‌ای در فشرده‌سازی تصاویر و ویدیوها دارد. به عنوان مثال فرمت‌های رایج مانند JPEG برای فشرده‌سازی تصاویر و MPEG برای فشرده‌سازی ویدیو از این مدل رنگی استفاده می‌کنند. این فضا به دلیل تفکیک روشنایی از رنگ‌ها امکان کاهش حجم داده‌ها را بدون افت چشمگیر کیفیت فراهم می‌کند. علاوه بر این YCbCr در استانداردهای تلویزیونی دیجیتال و آنالوگ مانند NTSC و PAL نیز استفاده می‌شود؛ زیرا انتقال رنگ‌ها و تصاویر را به‌طور کارآمد و با کیفیت بالا امکان‌پذیر می‌سازد. این ویژگی‌ها باعث می‌شود YCbCr برای کاربردهای حرفه‌ای مانند فشرده‌سازی و پخش ویدیو کاربردهای گسترده و مفیدی داشته باشد.

پس فضای رنگی RGB برای نمایش مستقیم تصاویر و پردازش‌های ساده بسیار کاربردی است، اما برای تجزیه و تحلیل دقیق رنگ‌ها و روشنایی مناسب نیست. از طرف دیگر فضای HSV به دلیل توانایی پردازش رنگ‌ها به‌صورت مستقل از روشنایی، برای کاربردهای هنری و شناسایی رنگ‌ها بسیار مفید است. در نهایت نیز فضای YCbCrبرای فشرده‌سازی تصاویر، ویدیوها و همچنین تفکیک روشنایی از رنگ‌ها کاربرد دارد. (هر یک از این فضاهای رنگی مزایا و معایب خود را دارند و بسته به کاربرد و کاری که می‌خواهیم انجام دهیم انتخاب درست فضا می‌تواند برای ما بسیار کمک کننده و مفید باشد.)

.H

ابتدا سه تصویری که در سوال گفته شد را آپلود می‌کنیم و سپس مطمئن می‌شویم که این سه تصویر هم اندازه باشند و در غیر اینصورت ارور نمایش دهد. در ادامه تابع find_peak_intensity() را برای پیدا کردن پیک شدت روشنایی هر تصویر تعریف می‌کنیم و با کمک کتابخانه OpenCv هیستوگرام برای نمایش توزیع شدت روشنایی رنگ‌ها ایجاد می‌کنیم؛ [0] مشخص می‌کند کدام کانال رنگی باید انتخاب شود که اینجا از تنها کانال‌مان استفاده کردیم، چون از ماسکی برای تصویر استفاده نکردیم None گذاشتیم، تعداد باین‌ها را هم 256 انتخاب می‌کنیم و شدت‌ها در 255 بخش مختلف تقسیم می‌شوند و در نهایت هم این مقادیر را در محدوده 0 تا 255 محاسبه می‌کنیم. پس از اینکه این تابع را تعریف کردیم پیک شدت روشنایی سه تصویر ورودی را بدست بدست می‌آوریم.

بعد از اینکه تصاویر را آپلود کردیم و پیک شدت روشنایی هر تصویر را پیدا کردیم، از آنجایی که در اکثر تصاویر کانال قرمز معمولا بالاترین شدت را دارد ابتدا سعی می‌کنیم که از روی پیک شدت که در مرحله قبل بدست آوردیم تشخیص بدهیم کدام تصویر شدت بیشتری دارد و آن را به عنوان کانال قرمز در نظر می‌گیریم. وقتی کانال قرمز را پیدا کردیم آن را از لیست کانال‌های باقی‌مانده حذف می‌کنیم و سراغ پیدا کردن دیگر کانال‌های سبز و آبی را می‌رویم. برای پیدا کردن دو کانال دیگر میانگین شدت رنگ را حساب می‌کنیم و در بیشتر تصاویر طبیعی چون چشم انسان حساسیت بیشتری به رنگ قرمز دارد معمولا میانگین شدت بیشتری دارد و از این رو هر کدام که میانگین بیشتری داشت را به کانال سبز اختصاص می‌دهیم و دیگری را به کانال آبی می‌دهیم.

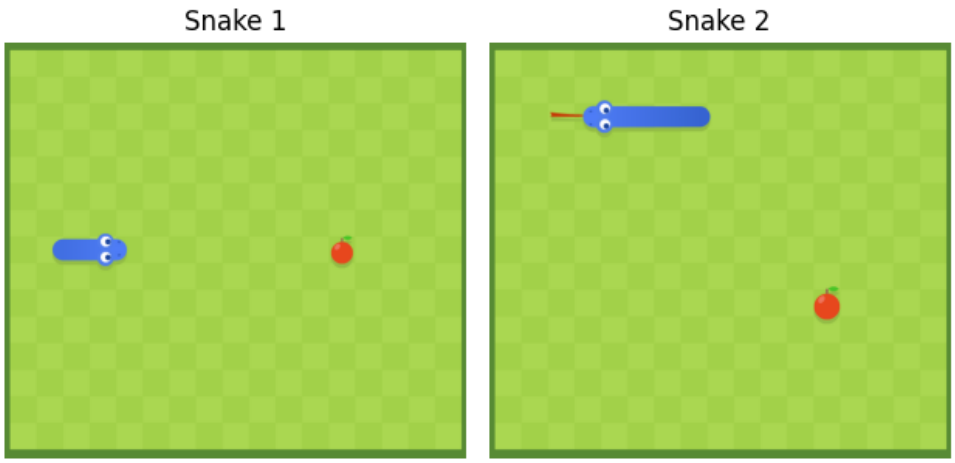
حالا که کانال‌ها را شناسایی کردیم آنها را به متغیر هایی که تعریف کردیم می‌دهیم و کانال‌ها را به ترتیب BGR با هم ترکیب کرده و در متغیر Reconstructed_image ذخیره می‌کنیم، سپس به فرم RGB تبدیل کرده و در نهایت نیز تصویری که بازسازی کردیم را در خروجی نمایش می‌دهیم:

Problem 2: The Snake Game is Here!

A.

این تمرین مربوط به همان ماربازی است که اکثر افراد این بازی را انجام داده اند و حالا در این پارت ما می‌خواهیم با استفاده از دو تصویر بازی "Q2_snake1.png" و"Q2_snake2.png" اجزای اصلی بازی شامل صفحه خالی، سیب، مار قبل از خوردن سیب و مار پس از خوردن سیب را استخراج و شناسایی کنیم.

طبق روال اول کتابخانه‌های مورد نیاز را می‌آوریم و سپس بعد از اینکه دو تصویر را خواندیم، آنها را به RGB تبدیل می‌کنیم و دو تصویر اولیه را در خروجی نمایش می‌دهیم:



- **نمایش سیب**

در این بخش، ابتدا تصویر از فضای رنگی BGR به HSV تبدیل می‌شود تا بتوان محدوده رنگ سبز پس‌زمینه را به‌راحتی تشخیص داد. با استفاده از یک ماسک، ناحیه‌های سبز (پس‌زمینه) از تصویر جدا شده و سپس با معکوس کردن این ماسک، نواحی مربوط به اشیاء (مانند سیب و مار) استخراج می‌شوند. در ادامه، با استفاده از روش تشخیص کانتورها، موقعیت و محدوده اشیاء در تصویر شناسایی شده و کوچک‌ترین کانتور (که مربوط به سیب است) جدا می‌شود. در نهایت، سیب از تصویر اصلی برش خورده و به‌عنوان خروجی ذخیره می‌شود:

نمایش مار قبل و بعد از خوردن سیب

برای نمایش مار قبل از خوردن سیب و بعد از خوردن سیب هم از روش قسمت قبل می‌توانیم استفاده کنیم و تصویر ورودی از فضای رنگی BGR به HSV تبدیل می‌شود تا محدوده رنگ سبز پس‌زمینه به‌راحتی تشخیص داده شود. با ایجاد یک ماسک، نواحی سبز (پس‌زمینه) از تصویر جدا شده و سپس با معکوس کردن این ماسک، نواحی مربوط به اشیاء استخراج می‌شوند. در ادامه، با استفاده از روش تشخیص کانتورها، موقعیت و محدوده اشیاء در تصویر شناسایی شده و بزرگ‌ترین کانتور (که مربوط به مار است) جدا می‌شود؛ سپس مار از تصویر اصلی برش خورده و به‌عنوان خروجی ذخیره می‌شود.

تصویر مار قبل از خوردن سیب:

تصویر مار بعد از خوردن سیب:

نمایش صفحه خالی بازی

برای نمایش صفحه خالی با استفاده از دو تصویر ورودی اولیه که از حالت‌های مختلف بازی هستند، تفاوت‌های بین آن‌ها محاسبه می‌شود. این تفاوت‌ها شامل نواحی‌ای است که در آن‌ها اشیاء (مانند مار یا سیب) قرار دارند. سپس با استفاده از یک حد آستانه این نواحی به صورت باینری مشخص می‌شوند. در ادامه سوراخ‌های ایجاد شده توسط اشیاء (همان قسمت های مشکی که اثر جای مار سیب بوده‌اند) در تصویر شناسایی و برچسب‌گذاری می‌شوند. برای پر کردن این سوراخ‌ها از رنگ‌های اطراف هر سوراخ نمونه‌برداری شده و میانگین رنگ آن‌ها محاسبه می‌شود. در نهایت سوراخ‌ها با این میانگین رنگ پر شده و صفحه خالی بازی بدون اشیاء (مار و سیب) ذخیره می‌شود و در خروجی داریم:

.B

در این بخش باید به کمک بخش قبل یک انیمیشن از فرآیند خوردن سیب توسط مار ایجاد کنیم. ابتدا اجزای اصلی بازی شامل تصویر اولیه، صفحه خالی، سیب و تصاویر مار در دو حالت قبل و بعد از خوردن سیب را آپلود می‌کنیم، سپس موقعیت اولیه مار و سیب در صفحه تعیین می‌شود و با استفاده از روش اینترپولاسیون خطی مسیر حرکت مار از نقطه شروع به سمت سیب محاسبه شده و 20 فریم میانی برای نمایش حرکت روان مار تولید می‌شود. در هر فریم صفحه بازی خالی به عنوان پایه استفاده شده، سیب در موقعیت ثابت خود قرار می‌گیرد و مار با توجه به موقعیت محاسبه‌شده در آن مرحله به تصویر اضافه می‌شود. در فریم‌های نهایی نیز هنگامی که مار به سیب نزدیک می‌شود تصویر سیب حذف شده و مار به حالت تغییر شکل‌یافته پس از خوردن سیب نمایش داده می‌شود و در نهایت به صورت یک فایل gif ذخیره می‌شود.

Problem 3: Digital Image Acquisition

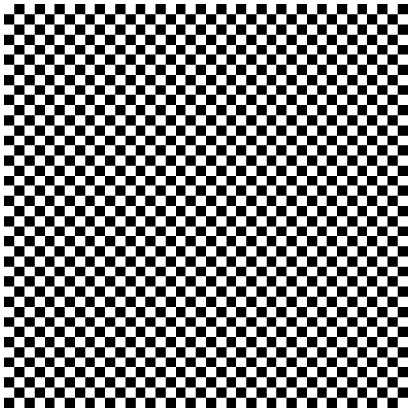
A.

در این بخش و بخش بعدی باید توابع ریاضی گفته‌شده در سوال را پیاده‌سازی کنیم و تابع generate_image() را تعریف می‌کنیم که یک تابع ریاضی و اندازه تصویر را گرفته و تصویر معادل را ایجاد می‌کند و سپس در مسیری که تعیین می‌کنیم آن را ذخیره می‌کند. در ادامه می‌آییم دو آرایه x و y را ایجاد می‌کنیم که یکسری مقادیر از 1- تا 1 برای هر یک از ابعاد تصویر (عرض و ارتفاع) بسازد. x و y دو آرایه یک بعدی هستند و با استفاده از np.meshgrid() به دو آرایه دو بعدی تبدیل می‌کنیم. (X مقادیر x را در امتداد ردیف‌ها تکرار می‌کند و Y نیز مقادیر y را در امتداد ستون‌ها تکرار می‌کند.)

از np.clip() استفاده می‌کنیم تا داده‌ها در محدوده 0 تا 255 قرار گیرند و شامل مقادیر دیگر نباشند و بعد از Image.fromarray() برای تبدیل آرایه به شی تصویر استفاده می‌کنیم و سپس تصویر ایجادشده در مسیر ذخیره می‌شود. حالا func_1 را برای تابع ریاضیاتی اولی تعریف می‌کنیم که این تابع یک الگوی دایره‌ای ایجاد می‌کند و اینکه در 255 هم ضرب شره به خاطر این است که مقیاس 0 تا 255 شود. سپس تابع دوم را تعریف می‌کنیم و خروجی یک صفحه شطرنجی سیاه و سفید می‌شود که اگر شرط اول برقرار باشد آن پیکسل سفید و در غیر اینصورت سیاه خواهد شد.

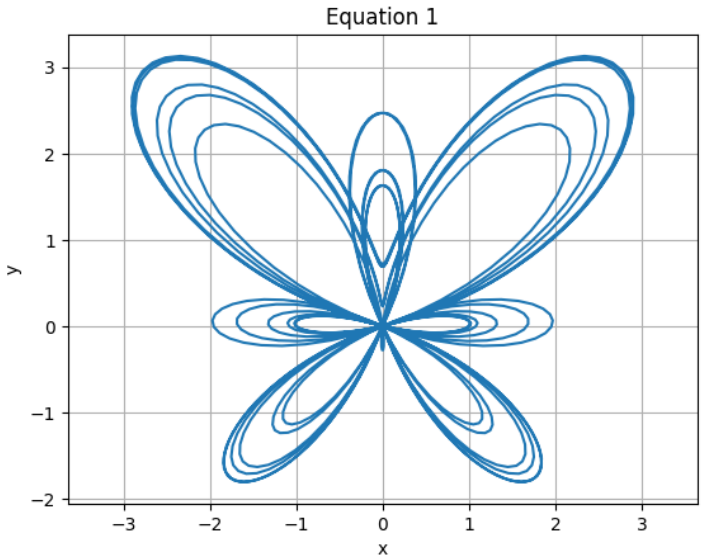
B.

بعد از اینکه در قسمت قبلی توابع مدنظر را ایجاد کردیم، در اینجا باید تصاویر خروجی دو تابع را با اندازه 512x512 تولید کنیم. سایز تصویر را همانطور که گفته شد 512x512 در نظر می‌گیریم و بعد تابع generate_image() را برای دو تابع بخش قبل فراخوانی می‌کنیم و تصاویر تولیدی را ذخیره می‌کنیم که خروجی در نهایت به این صورت در می‌آیند:

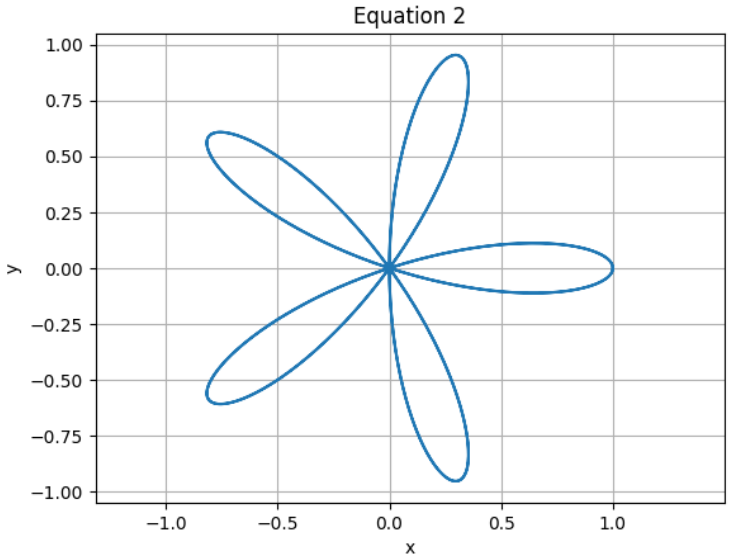


C.

در این پارت هم مثل قسمت‌های قبل دو تابع ریاضیاتی دیگر که گفته شده است را باید پیاده‌سازی و تصاویر مربوط به آنها را تولید کنیم. معادله اول را با تابع generate_eq1(*t*) تعریف می‌کنیم و محدوده مجاز t هم به کد می‌دهیم و در نهایت تصویر تولیدی را در خروجی نمایش می‌دهیم:



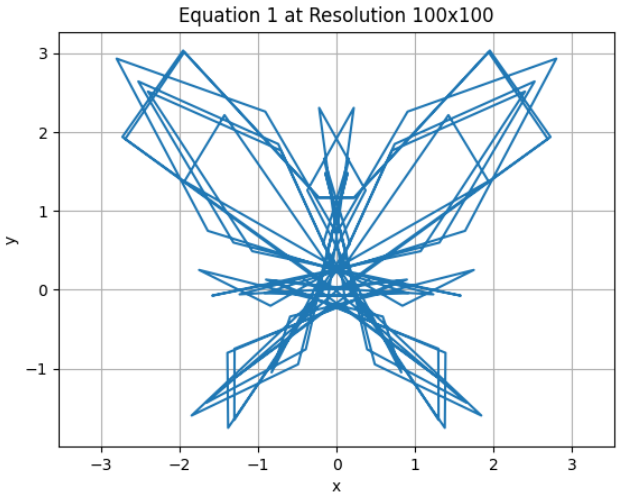
معادله دوم هم با تابع generate_eq2() تعریف می‌کنیم و بعد محدوده مجاز تتا را مشخص می‌کنیم و در نهایت هم تصویر تولیدی معادله دوم را در خروجی نمایش می‌دهیم:

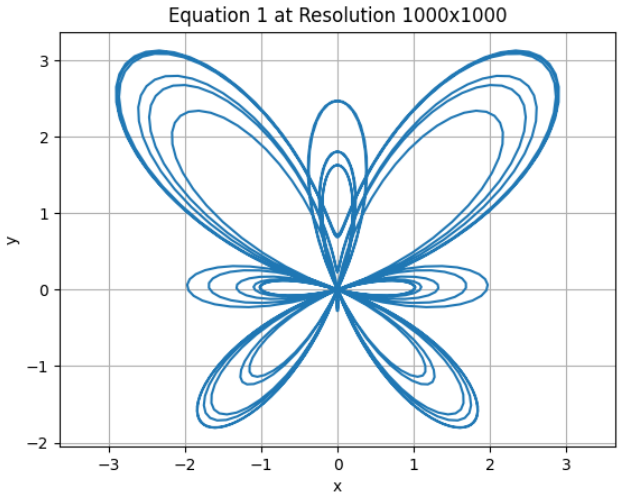


D.

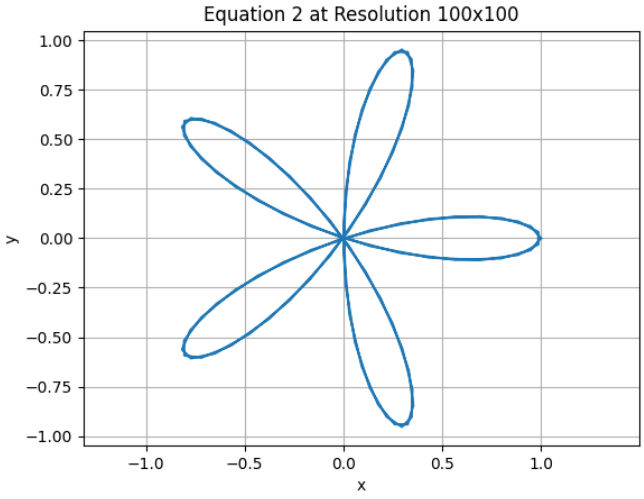
حالا باید معادله‌های بخش قبل را با رزولوشن‌های مختلف مثل 100x100، 500x500 و 1000x1000 تولید کنیم و یک تابع generate_image_resolution() تعریف می‌کنیم که تابع مربوطه و رزولوشن را می‌گیرد و بعد برای متغیر resolutions هم سه مقدار 100، 500 و 1000 می‌گذاریم یک حلقه for تعریف می‌کنیم که تصاویر تولیدی با رزولوشن‌های rxr تولید کند و در نهایت هم این تصاویر را برای هر معادله ذخیره کند.

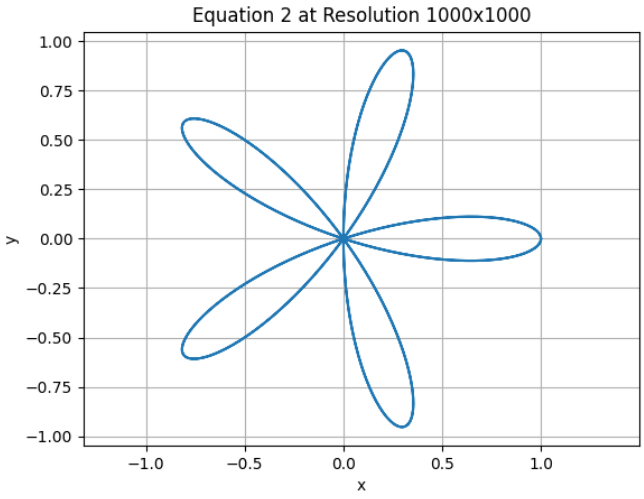
برای معادله اول برای آنکه تغییرات را نشان دهیم به عنوان مثال دو تصویر 100x100 و 1000x1000 را نمایش می‌دهیم:





برای معادله دوم هم به همین صورت تصاویر تولیدی را نمایش می‌دهیم:





.E

در کدهای پارت قبل ما از سه رزولوشن مختلف برای نمایش تصاویر تولیدی خروجی استفاده کردیم، رزولوشن در تصویر به تعداد پیکسل‌های موجود در تصویر اشاره دارد و هر چه رزولوشن بالاتر باشد تعداد پیکسل‌ها بیشتر خواهد بود و به تبع آن دقت و جزئیات تصویر هم بیشتر می‌شود.

همانطور که در بخش قبل تصاویر خروجی رزولوشن‌های مختلف را نشان دادیم. وقتی که تصویر با رزولوشن‌های پایین مثل همان 100x100 تولید می‌شود، چون تعداد پیکسل‌ها کمتر است و اطلاعات کمتری در تصویر وجود دارد ممکن است منحنی‌ها با خطوط کج و ناصاف دیده شوند (مثل تصویر 100x100 معادله اول که این موضوع را به خوبی نشان می‌دهد)؛ پس برای توابعی که پیچیدگی دارند وقتی از رزولوشن پایین استفاده کنیم ممکن است تصویر مثل شکل اصلی آن تولید نشود. حالا وقتی که تعداد پیکسل‌ها بیشتر شود و از 500x500 استفاده کنیم منحنی‌ها نسبت به رزولوشن قبلی شروع به صاف‌شدن می‌کنند و جزئیات بیشتری در تصویر می‌بینیم. با افزایش مجدد تعداد پیکسل‌ها تا 1000x1000 تصویر بهتر و دقیق‌تر تولید می‌شود و جزئیات به‌طور واضح قابل مشاهده هستند و منحنی‌ها و خطوط به خوبی نمایان می‌شوند (این تفاوت رزولوشن به وضوح و کیفیت تصویر مربوط می‌شود).

پس رزولوشن‌های بالا باعث می‌شود که تصویر با وضوح بهتر و دقیق‌تر باشد اما در رزولوشن‌های پایین جزئیات تصویر ممکن است از دست بروند و در نتیجه اگر بخواهیم کیفیت تصویر را به بهترین شکل ببینیم باید از رزولوشن‌های بالاتر استفاده کنیم (البته در اکثر موارد نیازی به نمایش با بالاترین کیفیت نیست و اگر نیازی به نمایش جزئیات بیشتری در توابع پیچیده‌تر و ... داشتیم استفاده می‌کنیم). با افزایش رزولوشن، کیفیت و وضوح تصویر بهبود می‌یابد و مثلاً در رزولوشن 100x100 شکل کلی تابع ممکن است دیده شود ولی جزئیات بیشتر، انحناها و ... مشخص نباشد؛ در حالی که در رزولوشن 1000x1000 جزئیات دقیق تابع قابل مشاهده است و تصویر بهتر و صاف‌تر به نظر می‌رسد.

Problem 4: Extracting Relevant Colors from an Image

A.

در تمرین چهارم قرار هست که به فرایند تقسیم‌بندی تصویر یا Image Segmentation بپردازیم. (تقسیم‌بندی تصویر یعنی تقسیم یک تصویر به بخش‌های مختلف بر اساس یکسری معیار، هدف اصلی این کار این است که قسمت‌های مهم تصویر را جدا کنیم و مثلاً اشیا یا نواحی خاصی که می‌خواهیم روی آنها کار کنیم را جدا می‌کنیم.) در این تمرین یک فایل با نام Q4_us_president.txt داریم که شامل 210012 بردار سه‌تایی است که هر بردار مقادیر رنگ‌های قرمز، سبز و آبی پیکسل‌های تصویر موجود در تمرین را نشان می‌دهندو همچنین این تصویر دارای 516 ردیف و 407 ستون می‌باشد (پیکسل‌های در فایل به صورت ردیفی از بالا به پایین و در هر ردیف از چپ به راست قرار دارند). این پارت از ما می‌خواهد ابتدا تصویر را بخوانیم و بعد dominant color و مجموعه پالت را با سایز 5، 7 و 9 نشان دهیم.

برای آنکه داده‌های پیکسل‌ها را از فایل txt آپلود کنیم ابتدا یک تابع تعریف می‌کنیم و هر موقع که خواستیم با استفاده از مسیر فایل و دستور np.loadtxt() که برای خواندن از یک فایل txt است استفاده می‌کنیم. برای پیدا کردن dominant color یا رنگ‌های غالب تابع find_dominant_colors() را تعریف می‌کنیم و از الگوریتم خوشه‌بندی Kmeans استفاده می‌کنیم. (الگوریتم K-means یک روش بدون نظارت است که هدف آن تقسیم یک مجموعه داده به K خوشه مجزا است. این الگوریتم نقاط داده را به خوشه‌هایی گروه‌بندی می‌کند که ویژگی‌های مشابهی دارند و در هر تکرار، مرکز هر خوشه به‌روزرسانی می‌شود تا به بهترین ترتیب ممکن نقاط داده به خوشه‌ها اختصاص یابند.)

تعداد خوشه‌ها در Kmeans را برابر k قرار می‌دهیم که همان مقادیر 5، 7 و 9 می‌باشد؛ سپس با fit الگوریتم Kmeans روی داده‌های ورودی اعمال می‌شود و در نهایت نیز رنگ‌های غالب را به عنوان مراکز خوشه‌ها بر می‌گردانیم. تا به اینجای کار دو تابع برای خواندن تصویر و پیدا کردن رنگ‌های غالب تصویر با کمک خوشه‌بندی Kmeans تعریف کردیم و حالا برای نمایش این رنگ‌های غالب تابع display_palette() را تعریف می‌کنیم؛ اول یک آرایه تماما صفر ایجاد می‌کنیم و ارتفاع پالت را برابر مقدار ثابت 100 قرار می‌دهیم، عرض پالت به تعداد رنگ‌ها بستگی دارد و برای آن از len(colors) * 100 استفاده می‌کنیم و مثلا اگر 4 رنگ داشته باشیم عرض 400 پیکسل می‌شود. تعداد کانال‌های رنگی هم برابر 3 قرار می‌دهیم، پس تا به اینجا ما یک آرایه‌ای با ابعاد گفته‌شده ایجاد کردیم که برای نمایش رنگ‌های پالت به کار گرفته می‌شود.

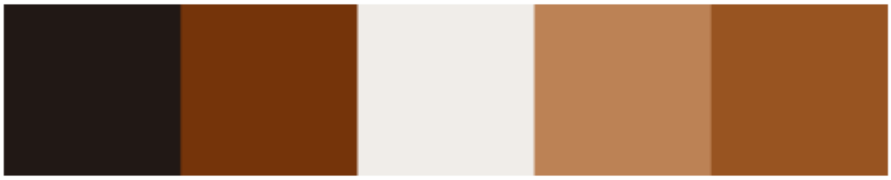
حالا با استفاده از یک حلقه for بخش‌های مختلف پالت را با رنگ‌های مربوطه پر می‌کنیم و با کمک enumerate() هر رنگ و اندیس مربوط به آن را پیدا می‌کنیم و در جای خود قرار می‌دهیم. (100 * i اولین ستون از بخش مربوط به رنگ i می‌باشد و 100 * (i + 1) هم آخرین ستون مربوط به آن رنگ را نشان می‌دهد) و در نهایت پالت رنگی را با رنگ‌هایی که در colors اند پر می‌کنیم و سپس پالت را نمایش می‌دهیم.

بعد از اینکه سه تابع بالا را تعریف کردیم حالا فایل txt را می‌خوانیم و بعد با reshape شکل آرایه را تغییر می‌دهیم و آن را به یک آرایه دوبعدی تبدیل می‌کنیم که هر ردیف آرایه نشان‌دهنده یک پیکسل است و سه مقدار

مربوط به رنگ‌های قرمز، سبز و آبی را دارد؛ سپس اندازه پالت را با توجه به خواسته سوال برابر 5، 7 و 9 قرار می‌دهیم و رنگ‌های غالب را پیاده می‌کنیم و مقادیر آنها را چاپ می‌کنیم و در نهایت به صورت یک پالت رنگی با اندازه‌های خواسته شده نمایش می‌دهیم:

- **K = 5**

```
Dominant colors for palette size 5:
[[ 33.90614455  24.56074213  21.20899311]
 [117.9844066   52.56739592  10.40872094]
 [240.75265584 237.99418721 233.06414111]
 [188.82344849 130.96632019  85.14124121]
 [152.57443997  84.60196616  33.36219178]]
```



- **K = 7**

```
Dominant colors for palette size 7:
[[ 23.59039065  23.30117329  25.1705801 ]
 [158.93756643  99.48075235  53.26928231]
 [101.39514202  48.2061322   14.57127688]
 [241.33577385 238.75899939 234.03721782]
 [195.30974071 137.41506657  92.668185  ]
 [142.19518336  63.87121442  10.01979344]
 [ 60.57512971  27.76997579  10.69830508]]
```

- **K = 9**

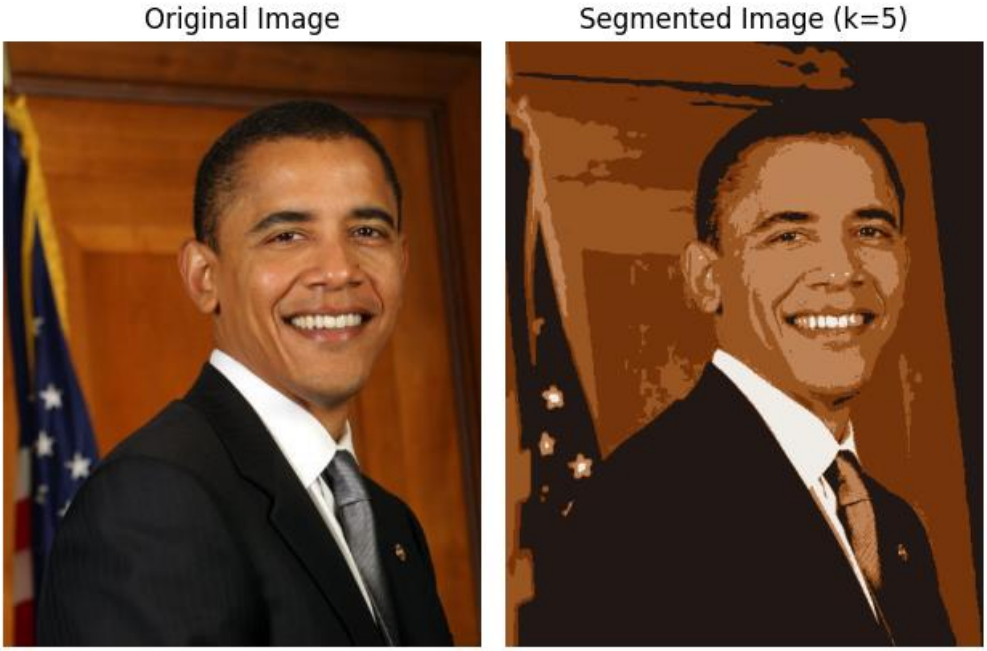
```
Dominant colors for palette size 9:
[[ 20.86679086  17.88013771  16.81151697]
 [159.12687733  91.39817374  40.65535264]
 [243.53708495 241.51530832 237.54312204]
 [135.19180597  60.09352054   9.50874863]
 [188.88724751 123.10823033  68.11147121]
 [ 78.33288043  35.52884615  11.01682692]
 [201.8865061  152.58099637 116.99736061]
 [108.9661752   94.35212489  83.70793582]
 [ 33.09368066  30.98437642  33.36578619]]
```

.B

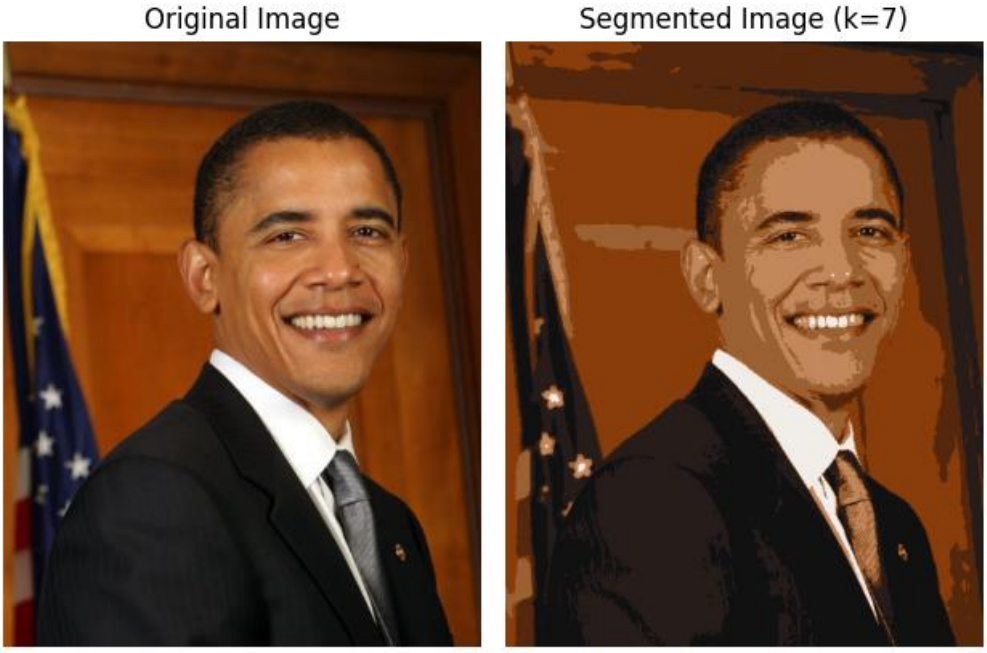
این پارت از ما می‌خواهد از بخش قبل برای تقسیم‌بندی تصویر استفاده کنیم و ابتدا تابع reconstruct_image را تعریف می‌کنیم که ورودی‌های آن داده‌های فایل txt و تعداد ردیف‌ها و ستون‌ها که در صورت تمرین اشاره شده به ترتیب 516 و 407 هستند را داریم؛ سپس با استفاده از reshape داده‌های تصویری را از حالت تک بعدی به سه بعدی تبدیل می‌کنیم و ابعاد آن به صورت (rows, cols, 3) قرار می‌گیرد. تابع get_dominant_colors را هم برای شناسایی رنگ‌های غالب در تصویر تعریف می‌کنیم و مثل پارت قبل از Kmeans برای این کار استفاده کردیم.

برای تقسیم‌بندی تصویر هم تابع segment_image را تعریف می‌کنیم که پیکسل‌ها را بر اساس رنگ‌های غالبی که شناسایی کرده بودیم تقسیم می‌کند و بعد فاصله اقلیدسی بین هر پیکسل و رنگ غالب را محاسبه می‌کنیم و سپس با np.argmin(distances) نزدیک‌ترین رنگ غالب برای هر پیکسل را پیدا می‌کنیم و سپس با رنگ‌های غالب تصویر را بازسازی می‌کنیم و reshape(image.shape) باعث می‌شود که تصویر جدید به همان ابعاد اولیه شکل اصلی برگردد و در نهایت با مقادیر k برابر 5، 7 و 9 خروجی این سگمنتیشن را نمایش می‌دهیم:

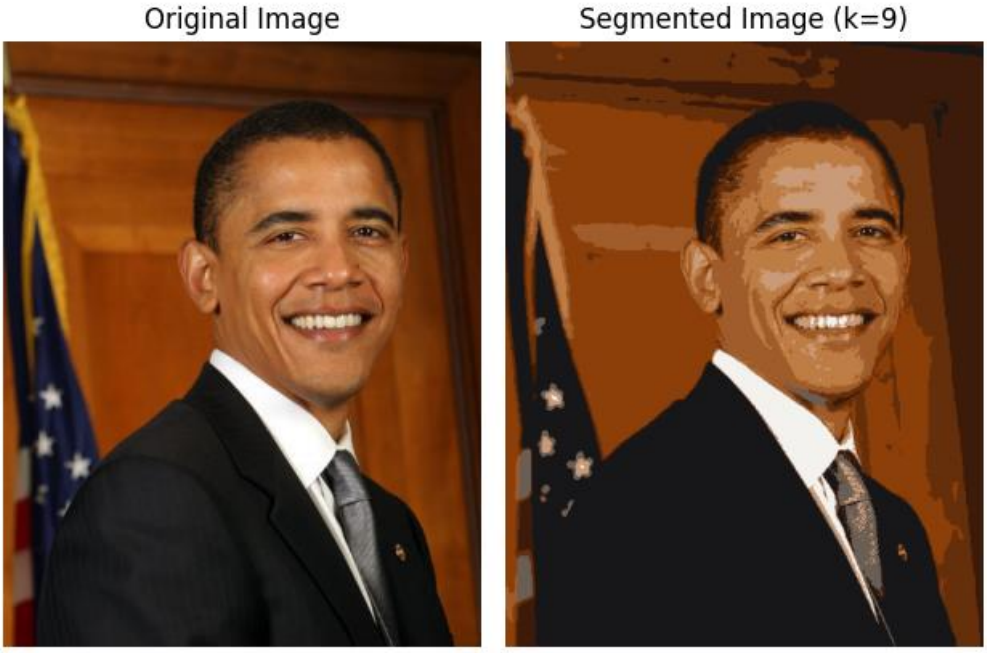
- **K = 5**


Original Image
Segmented Image (k=5)

- **K = 7**


Original Image
Segmented Image (k=7)

- **K = 9**



اگر به تصویرهای بالا کمی دقت کنیم می‌بینیم که با افزایش مقدار K که همان سایز پالت رنگی هست، از رنگ‌های بیشتری برای توصیف تصویر استفاده می‌شود و در نتیجه خروجی به تصویر اصلی نزدیک‌تر است و باعث افزایش دقت در تصویر می‌شود و جزئیات بیشتری را می‌تواند حفظ کند (البته افزایش بیش از حد مقدار K و سایز پالت ممکن است به خاطر محاسبات و پردازش‌های بیشتر باعث پیچیدگی بشود).

Problem 5: Creating an Image Mosaic using OpenCV

A.

در این تمرین هدف ما ایجاد یک تصویر موزائیک است که در آن یک تصویر هدف با استفاده از تصاویر کوچک‌تر از یک پوشه مشخص بازسازی می‌شود. ابتدا تصویر هدف به یک شبکه از کاشی‌های کوچک تقسیم می‌شود که اندازه هر کاشی بر اساس ابعاد تصویر، تعداد سطرها و ستون‌های شبکه تعیین می‌شود. سپس برای هر کاشی، تصویری از پوشه تصاویر کوچک‌تر انتخاب می‌شود که بیشترین شباهت رنگی را با کاشی داشته باشد و بعد از یافتن بهترین تصویر تطبیق‌یافته، هر کاشی در تصویر هدف با آن تصویر جایگزین می‌شود و در نهایت تصویر موزائیک ایجاد می‌شود.

اول کتابخانه‌های مورد نیاز را می‌آوریم و برای ایجاد تصاویر ورودی از یک شخص، کوئری مشخصی را اجرا می‌کنیم و از اینترنت تصاویر مربوط به شخص (به عنوان مثال لئوناردو دیکاپریو) را به تعداد گفته شده در کد دانلود می‌کنیم و آن‌ها را در یک پوشه موقت ذخیره و سپس تصاویر دانلود شده را به پوشه نهایی images منتقل می‌کنیم. بعد از اینکه تصاویر ورودی را دانلود کردیم و در فولدر images قرار دادیم، حالا تابع select_random_target() را تعریف می‌کنیم و به صورت رندوم از بین تصاویر ورودی یک تصویر را به عنوان هدف انتخاب می‌کنیم و با نام target ذخیره می‌کنیم.

بعد از مشخص شدن تصویر هدف؛ تابع split_target_image() را تعریف می‌کنیم و تصویر هدف را به تعدادی کاشی کوچک تقسیم می‌کنیم. اندازه هر کاشی بر اساس ابعاد تصویر، تعداد سطرها و ستون‌های شبکه تعیین می‌شود. در نهایت تعداد کاشی‌ها و ابعاد آن‌ها چاپ می‌شود و به عنوان مثال خروجی این قسمت برای تصویر هدف بالا به این صورت در می‌آید:

```
Generated 625 tiles of size 120x80
```

بعد از مراحل بالا اول تابع average_color() را تعریف می‌کنیم که میانگین رنگ یک تصویر را محاسبه می‌کند که نشان‌دهنده رنگ غالب آن تصویر است؛ سپس تابع دوم یعنی find_best_match() را تعریف می‌کنیم که بهترین تصویر تطبیق‌یافته را پیدا می‌کند و این تطابق بر اساس شباهت رنگ میانگین تصاویر انجام می‌شود، به این صورت که تصویری انتخاب می‌شود که کمترین فاصله رنگی (فاصله اقلیدسی) را با کاشی مورد نظر داشته باشد. در نهایت نیز بهترین تصویر تطبیق‌یافته برگردانده می‌شود تا در ساخت تصویر موزائیک در بخش بعدی استفاده

شود. در ادامه نیز تابع create_mosaic() را تعریف کردیم و در انتهای این بخشکد یک تصویر موزائیک از تصاویر کوچک‌تر ساخته می‌شود و تلاش می‌کنیم که تصویر موزائیک نهایی شبیه به تصویر هدف باشد.

ابتدا ما الگوریتم را با دانلود 100 تصویر از لئوناردو دیکاپریو شروع کردیم و grid_size را هم برابر (20, 20) در نظر گرفتیم، تصویر زیر را به صورت تصادفی به عنوان هدف در نطر گرفت و در نهایت تصویر راست ایجاد شد که اگر یکم چشمای خود را ریز کنیم می‌توانیم ببنیم که تا حدودی به هدف نزدیک شده است:


برای بهبود، بار دوم تعداد تصاویر بیشتری را یعنی 150 تا به الگوریتم دادیم تا دانلود کند و همچنین grid_size را برابر (25, 25) قرار دادیم تا دقت و شباهت تصویر موزائیکی به هدف ما نزدیک‌تر بشود و تصویر هدف (که چون تصادفی انتخاب می‌شود این بار این عکس از بین تصاویر دانلودشده انتخاب شد) و تصویر موزائیکی نهایی به صورت زیر به نمایش در آمدند:


می‌بینیم که تصویر موزائیکی نسبت به بار قبل دقت و بهبود قابل توجهی پیدا کرد و از نظر شباهت توانستیم تصویر هدف را بسازیم. مطمئنا با تنظیم مجدد پارامترها و ... می‌توانیم باز هم دقت را افزایش بدهیم و خیلی بیشتر تصویر موزائیکی به تصویر هدف نزدیک شود اما برای تصویر بالایی که نمایش دادیم و موارد مختلف را برای بهبود دقت دادیم؛ به دلیل محدودیت سیستم و توان سخت‌افزاری، برای تولید تصویر موزائیکی بالایی مدت زمان طولانی برای اجرا روی سیستم سپری شد و نزدیک به ۹۸ دقیقه طول کشید تا خروجی را نشان دهد و مطمئنا در صورت عدم مشکل سیستم و تغییراتی در کد مثل grid_size و ... می‌توان تصویر موزائیکی نهایی خیلی بهتری ایجاد کرد که شباهت بیشتری به تصویر هدف ما خواهد داشت.

Problem 6: Solving a Jigsaw Puzzle Using Basic Image Operations

A.

در این تمرین قصد داریم که یک پازل تصویری را با استفاده از تکنیک‌های پردازش تصویر حل کنیم و تصویر ورودی شامل 15 قطعه از یک پازل 5x3 است و ما باید این قطعات را از تصویر اصلی استخراج کنیم و سپس آن‌ها را به درستی کنار هم قرار دهیم تا تصویر کامل با ابعاد 954x750 پیکسل بازسازی شود. تصویر ورودی این تمرین به این صورت است:

در این پارت باید این 15 قطعه را به درستی استخراج کنیم و تصاویر را با نام [piece_XX.jpg] برای استفاده در مراحل بعد ذخیره کنیم. ابتدا بررسی کردیم که آیا پوشه‌ای به نام pieces وجود دارد یا خیر و اگر وجود نداشته باشد این پوشه ایجاد می‌شود تا قطعات استخراج‌شده در آن ذخیره شوند. سپس تصویر اصلی پازل آپلود کردیم و آن را به تصویر خاکستری تبدیل کردیم تا پردازش ساده‌تر شود. در ادامه با استفاده از حد آستانه قطعات پازل از پس‌زمینه سفید جدا شدند و نقاط مرزی قطعات شناسایی می‌شوند. قطعاتی که مساحت آن‌ها از 1000 پیکسل بیشتر است، به عنوان قطعات معتبر انتخاب می‌شوند و این قطعات از بالا به پایین و از چپ به راست مرتب شدند (چون قطعه‌ای که در گوشه بالا سمت چپ قرار دارد و در جای درست خود قرار دارد به عنوان تصویر اول ذخیره شود) و سپس هر قطعه از تصویر اصلی برش داده شد و به عنوان یک تصویر جداگانه در پوشه pieces با نام [piece_XX.jpg] ذخیره شدند.

.B

این بخش مهم‌ترین و اصلی‌ترین قسمت این سوال است و فرآیند حل پازل را با چیدمان خودکار قطعات انجام می‌دهد. ابتدا قطعات پازل از فولدر مربوطه آپلود می‌شوند و به ابعاد ثابت تغییر اندازه داده می‌شوند؛ سپس یک ماتریس 5x3 برای نگهداری موقعیت قطعات ایجاد می‌شود و قطعه اول در گوشه بالا سمت چپ قرار می‌گیرد و معیار اصلی ما برای شروع حل پازل و پیدا کردن همسایه‌های هر قطعه از همین قطعه شماره 1 شروع می‌شود. برای هر موقعیت در ماتریس قطعات باقی‌مانده بررسی می‌شوند و بهترین تطابق با همسایه‌های آن قطعه پیدا می‌شود. برای این کار شباهت لبه‌ها و گوشه‌های قطعات با استفاده از تفاوت پیکسلی محاسبه می‌شود. هر قطعه در زوایای 0، 90، 180 و 270 درجه چرخانده می‌شود تا بهترین تطابق پیدا شود. پس از یافتن بهترین قطعه، آن قطعه در موقعیت مناسب قرار می‌گیرد و ماتریس پازل به‌روزرسانی می‌شود و به همین منوال پیش می‌رویم تا همه قطعات پازل یعنی هر 15 قطعه در جای صحیح خود قرار گیرند و در نهایت ماتریس نهایی که موقعیت درست هر قطعه را نشان می‌دهد و برای پارت بعدی سوال و باز گرداندن تصویر نهایی حل پازل اهیمت دارد را در خروجی نمایش می‌دهیم:

```
Jigsaw Puzzle Matrix:
[[ 1  7  5]
 [14 12  4]
 [13  2 11]
 [ 8  3 15]
 [ 9 10  6]]
```

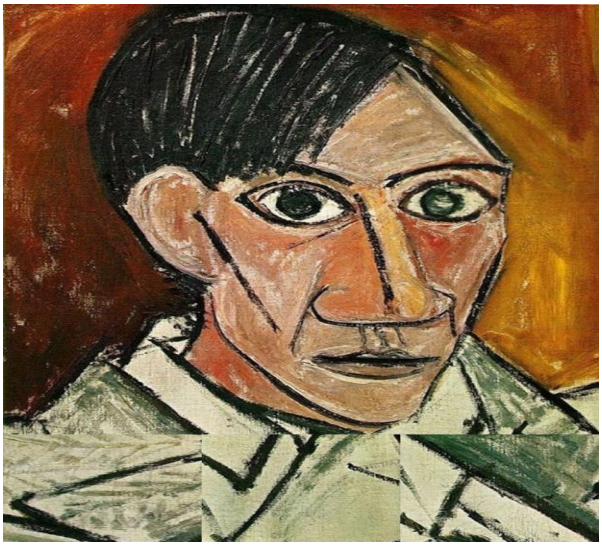
خروجی بالا چهار سطر اول ماتریس یعنی 12 قطعه کاملا به درستی در جای صحیح مربوط به خودشان قرار گرفتند اما سه قطعه ردیف آخر ماتریس یعنی 9، 10 و 6 به ترتیب درستی جای خود قرار نگرفتند و ساعت‌ها تغییراتی مختلفی را روی قسمت‌های گوناگون کدهای این پارت مثل تابع calculate_edge_similarity() یا calculate_diagonal_similarity() برای تشخیص درست شباهت لبه‌ها، قطر و ... انجام دادم اما متاسفانه نتیجه نهایی را نتوانستم بهبود بدم و نه تنها بهتر نمی‌شد بلکه قطعه‌های درست قبلی که سر جای صحیح خودشان بودند را هم دچار غلط می‌کرد و به دلیل اتمام مهلت تحویل تمرین، همین کد را قرار دادم که اگر خروجی بخش بعد را هم ببینیم همه قطعات به جز ردیف آخر که چون شباهت خیلی زیادی به هم داشتند و تشخیص آنها دشوارتر بود به درستی جای خود قرار گرفته‌اند.

.C

و در نهایت در این بخش تصویر نهایی پازل را با چیدمان صحیح قطعات ایجاد می‌کنیم. ابتدا یک تصویر خالی با ابعاد نهایی پازل یعنی 954x750 پیکسل ایجاد می‌کنیم. سپس ابعاد هر قطعه محاسبه می‌شود که با تقسیم ارتفاع و عرض تصویر نهایی بر تعداد ردیف‌ها و ستون‌های پازل به دست می‌آید. در ادامه هر قطعه از ماتریس پازل گرفته

می‌شود، به اندازه مناسب تغییر اندازه داده می‌شود و در موقعیت صحیح خود در تصویر نهایی قرار می‌گیرد. این کار با استفاده از مختصات شروع و پایان هر قطعه در تصویر نهایی انجام می‌شود.

پس از قرار دادن تمام قطعات، تصویر نهایی با نام solved_puzzle.jpg ذخیره می‌شود و در پایان این پارت نتیجه نهایی حل پازل را داریم که در خروجی آن را نمایش می‌دهیم:



«. . . اسفندماه ۱۴۰۳ . . .»