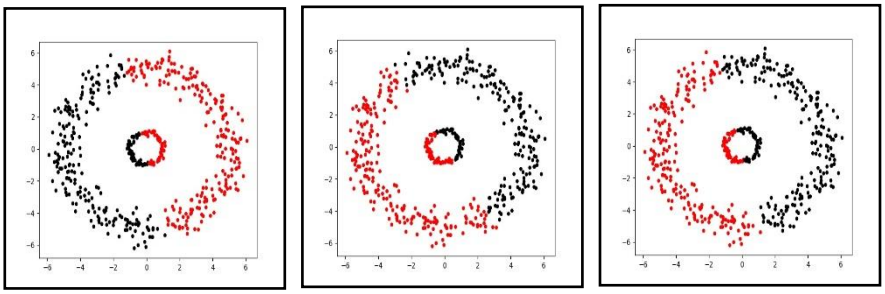
q1

ابتدا فایل Points.txt را باز کرده، خط اول که تعداد نقاط است را در متغیر `n` ذخیره میکنیم. حال به تعداد `n` بار خط های بعدی را از فایل میخوانیم. هر سطر شامل دو عدد `x` و `y` است که با فاصله از هم جدا شده اند که میتوانیم با دستور `split` این دو عدد را از یکدیگر جدا کرده و به عدد اعشاری `cast` کرده و در متغیر متناظرشان ذخیره کنیم. اکنون زوج مرتب `[x,y]` را به آرایه‌ی `points` اضافه میکنیم. اکنون با استفاده از تابع `plot` از کتابخانه‌ی `matplotlib.pyplot` نمودار نقاط را رسم میکنیم. ورودی های این تابع به ترتیب عبارتند از `x` ها، `y` ها، نوع نمودار که در اینجا `'o'` یا نقطه‌ای است، اندازه‌ی نقاط `4` و رنگ سیاه برای نقاط. نمودار حاصل را بعنوان خواسته‌ی اول سوال ذخیره کرده و فیگورهای داخل `plt` را با دستور `clf` حذف میکنیم تا بتوانیم مجددا نمودارهای بعدی را رسم کنیم. اکنون با استفاده از تابع `KMeans` از کتابخانه‌ی `sklearn.cluster` تقسیم بندی دوتایی را بر روی نقاط `fit` میکنیم. خروجی این تابع دارای پارامتری به نام `labels_` است که برچسب هر نقطه را تعیین میکند. دو رنگ سیاه و قرمز را تعریف میکنیم که برای مشخص کردن دسته بندی نقاط به کار میایند. این بار از آنجا که نقاط برچسب دار شده اند، با استفاده از تابع `scatter` از کتابخانه‌ی `matplotlib.pyplot` نمودار نقاط دسته بندی شده را رسم میکنیم. ورودی های این تابع به ترتیب عبارتند از `x` ها، `y` ها، اندازه‌ی نقاط `16` و رنگ سیاه و قرمز برای رنگ بندی هر دسته از نقاط. نمودار حاصل را بعنوان خواسته‌ی دوم سوال ذخیره کرده و فیگورهای داخل `plt` را با دستور `clf` حذف میکنیم. این کار را چند بار انجام میدهیم و مشاهده میکنیم که اگر مجموعه نقاط را بصورت دایره ای فرض کنیم، هر بار دو نیم دایره نا همرنگ مطابق نتیجه‌ی ذخیره شده بدست میاید و با هر بار اجرا کردن برجه برچسب برخی نقاط تغییر میکند و در دسته‌ی دیگری قرار میگیرند. به صورت:



اکنون آرایه‌ی `dists` را میسازیم که هر درایه‌ی آن برابر با فاصله‌ی نقطه‌ی متناظر از مرکز `[0,0]` است. حال برای اینکه این فضای ساخته شده را به تابع `scatter` داد، این آرایه را `reshape` میکنیم تا درایه‌های آن بصورت ستونی درآیند. بار دیگر با تابع `KMeans` فاصله ها را به دو دسته

تقسیم میکنیم و سپس با استفاده از تابع scatter نقاط را برحسب برچسب فاصله‌ی آنها با دو رنگ رسم میکنیم. نمودار حاصل را بعنوان خواسته‌ی سوم سوال ذخیره میکنیم.

q2

در ابتدا تصویر اصلی را 1/4 کرده و سپس آن را reshape میکنیم زیرا تابعی که از آن استفاده خواهیم کرد یک آرایه‌ی دو بعدی میگیرد ولی عکس ما سه بعدی است. سپس با استفاده از تابع آماده‌ی MeanShift از کتابخانه‌ی sklearn.cluster عکس reshape شده را قطعه بندی میکنیم. یکی از پارامتر های خروجی این تابع labels_ میباشد که آن را دوباره با دستور reshape به ابعاد عکس اصلی میبریم و آن را labels مینامیم. اکنون با یک for برای هر قطعه‌ی موجود در ماتریس labels، با استفاده از تابع cv2.morphologyEx هر قطعه یک mask در نظر گرفته و آن را MORPH_CLOSE میکنیم تا نویز های درون هر قطعه کمتر شوند. سپس با استفاده از تابع آماده‌ی label2rgb از کتابخانه‌ی skimage.color، بجای هر قطعه، میانگین رنگ های آن قطعه را قرار میدهیم. در نهایت عکس را به کیفیت اصلی خودش برمیگردانیم و از آنجا که عکس حالت کارتونی دارد، کیفیت آن تغییر چندانی نخواهد کرد.

q3

در این بخش یک کلاس پیاده سازی شده که ابتدا توضیحشان میدهیم.

کلاس SuperPixel :

ویژگی ها:
1) زوج مرتبی که نشان دهنده‌ی مختصات مرکز آن سوپر پیکسل است centerLoc
2) سه تایی مرتبی که نشان دهنده‌ی LAB مرکز آن سوپر پیکسل است centerLab

استفاده: برای نظم بخشیدن به کد این مفهوم تعریف شده است و همانطور که میبینیم، در هر SuperPixel تنها اطلاعات مرکز آن سوپر پیکسل نگه داشته میشود و به اطلاعات دیگری نیاز نداریم. یک تابع نیز در این کلاس بنام moveCenterTo تعریف شده است که با صدا زدن آن برای هر سوپرپیکسل، با ورودی یک تصویر img و دو عدد صحیح i و j، ابتدا مرکز این سوپر پیکسل را به نقطه‌ی [i,j] میبرد تا centerLoc به روز رسانی شود. سپس این مختصات را در تصویر img قرار میدهد تا LAB این نقطه در تصویر را بدست آورد تا centerLab به روز رسانی شود.

در این بخش چند تابع پیاده سازی شده که ابتدا توضیحشان میدهیم.

تابع calError:

ورودی ها:
1) دو لیست از سوپر پیکسل ها که یکی برای مرحله‌ی فعلی و دیگری برای مرحله‌ی قبلی است

خروجی ها:
1) مقدار خطا یا جابجایی مرکز سوپر پیکسل ها

عملکرد: در ابتدا متغیر error را برابر با 0 در نظر میگیریم و سپس با یک for فاصله‌ی مرکز سوپر پیکسل s ام از لیست اول و مرکز سوپر پیکسل s ام از لیست دوم را محاسبه کرده و به error اضافه میکنیم. در نهایت error را تقسیم بر تعداد سوپر پیکسل ها میکنیم تا میانگین جابجایی نقاط بدست آید. این مقدار را خطا نامیده و بعنوان خروجی بازگشت میدهیم.

تابع getGradient :

ورودی ها:
1) تصویر اصلی‌ای که گرادیان هر نقطه روی آن را محاسبه میکنیم
2) مختصات نقطه‌ای که میخواهیم گرادیانش را محاسبه کنیم

خروجی ها:
1) گرادیان آن نقطه در تصویر

عملکرد: با استفاده از فرمول زیر برای نقطه‌ی داده شده محاسبه میشود و بعنوان خروجی بازگشت داده میشود.

$$gradient(i,j) = ||img(i+1,j) - img(i-1,j)||^2 + ||img(i,j+1) - img(i,j-1)||^2$$

تابع moveCenters :

ورودی ها:
1) تصویر اصلی
2) لیست تمام سوپر پیکسل ها
3) اندازه‌ی پنجره که در این سوال باید 5 باشد

خروجی ها:
1) لیست سوپر پیکسل ها پس از انتقال

عملکرد: هدف این است که مرکز سوپر پیکسل ها را به گونه ای در پنجره‌ی 5×5 حرکت دهیم که مرکز هر سوپر پیکسل کمترین گرادیان را به نسبت همسایه‌هایش دارا شود. برای پیاده سازی میدانیم که پنجره‌ی 5×5 به این معنیست که به اندازه‌ی a= (5-1) /2=2 پیکسل از هر طرف را برای مرکز هر

سوپر پیکسل چک کنیم. پس یک for بر روی تعداد سوپر پیکسل ها زده و هر بار مختصات مرکز سوپر پیکسل را در متغیر centerLoc نگه میداریم. متغیر gradientMin را با مقدار اولیه‌ی بی نهایت میسازیم. سپس با دو for تو در تو، همسایگی 2 پیکسل از طرفین این مرکز را چک کرده و با تابع getGradient که در بالا آمده، گرادیان هر حالت را چک میکند و اگر کمتر از گرادیان مینیمم بدست آمده برای این سوپر پیکسل باشد، مقدار گرادیان مینیمم را بروزرسانی کرده و مختصات چک شده را بعنوان بهترین مختصات besti و bestj در نظر میگیرد. پس از اتمام این دو for تو در تو، مرکز این سوپر پیکسل با تابع moveCenterTo که در کلاس SuperPixel تعریف شد، به مختصات besti و bestj انتقال میابد. پس از انتقال هر سوپر پیکسل، اکنون لیست سوپر پیکسل ها بروزرسانی شده و مرکز های جدید دارند. پس لیست جدید سوپر پیکسل ها را بعنوان خروجی بازگشت میدهیم.

تابع initiate :

ورودی ها:
1) تصویر اصلی
2) تعداد سوپر پیکسل ها یا K

خروجی ها:
1) متغیر S ذکر شده در صورت سوال
2) لیستی از سوپر پیکسل ها

عملکرد: هدف این است که به تعداد K تا سوپر پیکسل در ابتدای کار بسازیم به صورتی که مرکز آنها با فاصله‌ی یکسان از هم قرار داشته باشند. برای این کار ابعاد عکس اصلی را در متغیر height و width نگه میداریم. سپس فاصله‌ی مرکز هر دو سوپر پیکسل از هم را در راستای عمودی Si و در راستای افقی Sj مینامیم. برای آنکه به تعداد K تا سوپر پیکسل داشته باشیم مقدار Si باید برابر با height تقسیم بر جذر K و مقدار Sj باید برابر با width تقسیم بر جذر K باشد. متغیر S که طبق صورت سوال، فاصله‌ی مرکز دو خوشه همسایه در ابتدای کار تعریف شده، را میانگین Si و Sj در نظر میگیریم زیرا فاصله‌ی مرکز دو سوپر پیکسل همسایه در راستای افقی و عمودی به دلیل ابعاد عکس، متفاوت است. اکنون یک لیست خالی برای سوپر پیکسل ها میسازیم و آن را superPixels مینامیم. سپس با دو for تو در تو، هر بار سوپر پیکسلی به مرکز [i,j] میسازیم. [i,j] را از مختصات [Si/2,Sj/2] شروع کرده و در هر گام i را بعلاوه‌ی Si و j را بعلاوه‌ی Sj میکنیم و در هر گام یک SuperPixel جدید به مرکز [i,j] و مقدار LAB برابر با img[i,j,:] میسازیم و به لیست superPixels اضافه میکنیم. این گونه K تا سوپر پیکسل منظم در ابتدای

کار بدست میاوریم. در نهایت متغیر S را که بالا محاسبه کردیم و همچنین لیست superPixels را بعنوان خروجی بازگشت میدهیم.

تابع iterate :

ورودی ها:
1) تصویر اصلی
2) ماتریس اندیس ها
3) لیستی از سوپر پیکسل ها
4) فاصله‌ی بین مرکز دو خوشه‌ی همسایه در ابتدای کار S
5) ضریب ثابت آلفا

خروجی ها:
1) ماتریس برچسب های هر پیکسل از تصویر labels
2) لیستی از سوپر پیکسل ها پس از یک مرحله تکرار

عملکرد: هدف این است که به هر پیکسلی که فاصله‌ی مکانی و رنگی آن از مرکز سوپر پیکسل s ام کمتر از فاصله‌ی آن از مرکز بقیه‌ی سوپر پیکسل ها بود، برچسب s زده شود تا بدانیم در هر مرحله هر پیکسل به کدام سوپر پیکسل تعلق دارد. هدف دیگر این است که از مختصات پیکسل هایی که متعلق به یک سوپر پیکسل تشخیص داده شدند، میانگین گیری شود و مختصات حاصل بعنوان مکان جدید مرکز آن سوپر پیکسل در نظر گرفته شود. به با هر بار میانگین گیری از پیکسل های یک خوشه، مرکز آن خوشه بروزرسانی میشود و اگر این کار برای تمام سوپر پیکسل ها انجام شود، در نهایت لیستی از سوپر پیکسل ها داریم که میتوانیم از آنها بعنوان خوشه‌های جدید برای مرحله‌ی بعدی استفاده کنیم. اکنون برای پیاده سازی این دو هدف، ابتدا ماتریسی با ابعاد تصویر اصلی ساخته و آن را labels مینامیم که در ابتدا تمام درایه های این ماتریس -1 اند. درایه‌ی [i,j] این ماتریس برچسب پیکسل [i,j] از تصویر اصلی است. یعنی اگر labels[i,j]=s باشد، به این معنیست که پیکسل [i,j] تصویر اصلی متعلق به خوشه‌ی s ام میباشد. پس درایه‌های این ماتریس پس از مقدار دهی باید عدد صحیحی بین 0 تا $K-1$ باشند. با فرض آنکه K تا سوپر پیکسل داریم. در ابتدای کار تمام درایه های این ماتریس با عدد -1 پر شده است و به این معناست که پیکسل ها در ابتدای کار به هیچ خوشه ای تعلق ندارند. پس برای برآورده شدن هدف اول باید این ماتریس مقدار دهی شود و بعنوان خروجی بازگشت داده شود. در ابتدا یک ماتریس minDists برای نگهداری نزدیک ترین فاصله‌ی هر پیکسل تا یکی از خوشه ها میسازیم. پس درایه‌ی [i,j] این ماتریس نشان میدهد که فاصله‌ی پیکسل [i,j] تصویر اصلی از مرکز نزدیکترین خوشه چقدر است و در ابتدا تمام درایه های این ماتریس بی نهایت در نظر گرفته

میشود. تاکید میشود که در اینجا منظور از فاصله تنها فاصله‌ی مکانی نیست، بلکه $d = d_{lab} + \alpha d_{xy}$ است که ضریب آلفا نیز بعنوان ورودی به تابع داده خواهد شد. اکنون باید برای همسایگی 2S×2S مرکز هر سوپر پیکسل، بررسی کنیم که پیکسل های داخل این محدوده چه فاصله ای از مرکز این سوپر پیکسل دارند. اگر این مقدار کمتر از مقدار فاصله‌ی مینیمم آن پیکسل بود، برچسب آن پیکسل را از آن خود کند. مقدار فاصله‌ی مینیمم هر پیکسل هم طبق توضیحات بالا در ماتریس minDists نگهداری میشود. پس ما باید با استفاده از یک for بر روی تعداد سوپر پیکسل ها این کار را انجام دهیم. فرض کنید در حال بررسی سوپر پیکسل s ام باشیم. ابتدا مختصات مرکز این خوشه را در متغیر centerLoc و LAB مرکز این خوشه را در centerLab نگهداری میکنیم. اکنون دو برش یا slice میسازیم. برشی که در راستای عمودی میسازیم را iSlice و برشی که در راستای افقی میسازیم را jSlice مینامیم. این برش ها در واقع یک پنجره به طول 2S میسازند. ولی به گونه ای نوشته میشوند که پنجره هرگز از عکس بیرون نزند. برای این کار آنها را با فرمول های زیر تعریف میکنیم.

$$iSlice: from \ \max(centerLoc_x - S, 0) \ \ to \ \ \min(centerLoc_x + S, height)$$

$$jSlice: from \ \max(centerLoc_y - S, 0) \ \ to \ \ \min(centerLoc_y + S, width)$$

این برش ها به این منظور ساخته شدند که بتوانیم از numpy استفاده کنیم و مجبور نباشیم هر پیکسل درون این همسایگی 2S×2S را تک به تک چک کنیم. مقدار LAB پیکسل های این همسایگی را با استفاده از برش های ساخته شده، از تصویر اصلی که بعنوان ورودی به این تابع داده شده، برش میدهیم و در متغیر neighborLab نگه میداریم. همچنین مختصات پیکسل های این همسایگی را با استفاده از برش های ساخته شده، از ماتریس اندیس ها که بعنوان ورودی به تابع داده شده، برش میدهیم و در متغیر neighborLoc نگه میداریم. سپس برای محاسبه‌ی فاصله‌ی رنگی بین پیکسل های این همسایگی و مرکز خوشه‌ی s ام، تمام درایه های ماتریس neighborLab را منهای مقدار centerLab میکنیم و سپس مجذور هر سه تایی مرتب که مقادیر LAB اند را با یکدیگر جمع میکنیم تا $d_{lab} = (l - l_c)^2 + (a - a_c)^2 + (b - b_c)^2$ بدست آید. همچنین برای محاسبه‌ی فاصله‌ی مکانی بین پیکسل های این همسایگی و مرکز خوشه‌ی s ام، تمام درایه های ماتریس neighborLoc را منهای مقدار centerLoc میکنیم و سپس مجذور هر زوج مرتب که مختصات نقطه اند را با یکدیگر جمع میکنیم تا $d_{xy} = (x - x_c)^2 + (y - y_c)^2$ بدست آید. اکنون دو ماتریس را با یکدیگر طبق فرمول $d = d_{lab} + \alpha d_{xy}$ جمع میکنیم و ماتریس حاصل را dist مینامیم. هر درایه از این ماتریس، فاصله‌ی پیکسل متناظر را از مرکز خوشه‌ی s ام نشان میدهد. پس مقادیر این ماتریس را درایه به درایه با برشی از ماتریس minDists مقایسه میکنیم. و هرجا که

درایه‌ی ماتریس dist کمتر از مقدار درایه‌ی ماتریس minDists بود، مقدار فاصله‌ی کمینه‌ی جدید را برای آن پیکسل جایگزین میکنیم و سپس برچسب آن پیکسل را در برش مناسب از ماتریس labels را برابر s قرار میدهیم. این قسمت برای هر خوشه انجام میشود و هر جا که خوشه‌ی جدیدی یافت شود که مرکزش به یک پیکسلی نزدیک تر باشد، خود به خود با مقایسه‌ی فاصله و برچسب دهی مجدد، برچسب قبلی از ماتریس برچسب ها labels حذف شده و برچسب جدید جایگزین میشود و میتوان گفت که این پیکسل از این به بعد متعلق به خوشه‌ی جدید است و از خوشه‌ی قبلی حذف شده. پس اکنون هدف اول این تابع یعنی برچسب گذاری انجام شده. هدف دوم این است که مختصات مرکز خوشه ها را با توجه به میانگین مختصات پیکسل های هر یک بروزرسانی کنیم تا اگر مرحله‌ی بعدی ای در کار بود، خوشه ها مرکز بهتری داشته باشند و انگار مرکز خوشه به میانگین پیکسل هایش نزدیک تر میشود و سوپر پیکسل بهتری میسازد. برای محاسبه‌ی مختصات جدید مرکز هر خوشه، میدانیم تمام پیکسل هایی که در این خوشه قرار دارند حتما در بازه‌ی 2S×2S مرکز خوشه قرار دارند. پس نیاز نیست تمام درایه‌های ماتریس labels را بررسی کنیم. پس با یک for برش iSlice و jSlice را مشابه قبل برای خوشه s ام ساخته و در این برش از ماتریس labels بررسی میکنیم که کدام پیکسل ها برچسب s دارند. میانگین آنها را محاسبه میکنیم و با دستور moveCenterTo که در کلاس SuperPixels تعریف شد، مرکز خوشه‌ی s ام را به مکان جدیدش میبریم. پس از اتمام اجرای این for، یک لیست از خوشه ها بنام superPixels داریم که مرکز خوشه ها را برای مرحله‌ی بعد تعیین میکند و هم چنین یک ماتریس labels داریم که اگر این مرحله، مرحله‌ی آخر باشد این ماتریس بکار میاید. زیرا باید بتوانیم مرز خوشه ها را با استفاده از مرکز ماتریس labels رسم کنیم. پس دو متغیر ذکر شده را بعنوان خروجی بازگشت میدهیم.

تابع smooth :

ورودی ها:
1) لیستی از سوپر پیکسل ها
2) ماتریس برچسب ها
3) فاصله‌ی بین مرکز دو خوشه‌ی همسایه در ابتدای کار S

خروجی ها:
1) ماتریس برچسب ها که smooth شده است

عملکرد: میدانیم با روشی که تا کنون در تابع iterate شرح دادیم، همواره ماتریس labels به گونه ای است که اگر مرز بین برچسب‌هایش رسم شود دندانه ها و ناصافی های شدیدی ملاحظه میکنیم.

برای آنکه اینگونه ناصافی ها را از بین ببریم، از این تابع استفاده میکنیم. برای انجام این کار کرنل مربعی با ابعاد [S/5] میسازیم. یعنی هر چه فاصله‌ی بین مرکز دو خوشه‌ی همسایه در ابتدای کار بیشتر باشد، کرنل ما بزرگتر خواهد بود و هر چه کرنل بزرگتر باشد، خواهیم دید که جزئیات بیشتری smooth میشوند. اکنون باید لبه های هر برچسب smooth شوند. پس یک for بر روی تعداد برچسب های درون ماتریس labels میزنیم و سپس برای برچسب s ام برش های iSlice و jSlice را مشابه آنچه در تابع iterate توضیح داده شد، حول مرکز سوپرپیکسل s ام میسازیم. زیرا میدانیم تنها پیکسل‌هایی که برچسب s دارند، حتما در بازه‌ی 2S×2S مرکز خوشه‌ی s ام قرار دارند و نیازی نیست که کل ماتریس labels را بگردیم تا درایه‌های s را بیابیم. پس ماتریس labels را با برش های iSlice و jSlice برش میدهیم و در labelsCropped قرار میدهیم. اگر مختصات تمام درایه‌هایی از ماتریس labelsCropped که مقدار s دارند را در بردارهای i و j نگه داریم، آنگاه یک mask سیاه به ابعاد labelsCropped ساخته و تمام درایه‌های i و j آن را برابر 1 قرار میدهیم. اکنون در واقع یک ماسک ساختیم که تمام پیکسل‌های با برچسب s را سفید و بقیه را سیاه نشان میدهد. اکنون اگر این mask را با دستور dilate و کرنل ذکر شده ابتدا منبسط کرده و سپس با دستور erode منقبض کنیم، دندانه ها از روی mask حذف میگردند. حال مجددا مختصات تمام درایه‌هایی از mask که مقدار 1 دارند را در بردارهای i و j نگه میداریم و اکنون تمام درایه‌های i و j ماتریس labelsCropped را برابر s قرار میدهیم و ماتریس labelsCropped جدید را به مکان ابتدایی خود در ماتریس labels بازمیگردانیم. یعنی برش iSlice و jSlice از ماتریس labels را برابر labelsCropped قرار میدهیم. پس از اجرای کامل for، میبینیم که لبه‌های تمام برچسب ها با این روش smooth شده اند پس ماتریس labels حاصل را بعنوان خروجی بازگشت میدهیم.

تابع slic :

ورودی ها:
1) تصویر اصلی
2) تعداد سوپر پیکسل ها K
3) ضریب آلفا

خروجی ها:
1) عکس قطعه بندی شده که مرز هر خوشه در آن رسم شده است

عملکرد: هدف این است که با استفاده از توابع پیاده سازی شده در بالا، هر عکس داده شده به K قطعه تقسیم بندی شود. برای این کار ابتدا عکس را به فضای LAB میبریم، سپس ماتریس اندیس ها indices را با ابعاد عکس اصلی میسازیم که در واقع در درایه‌ی [i,j] آن، مقدار [i,j] ذخیره شده است. یعنی هر درایه، مقدار مختصات خود را دارد. این ماتریس در ادامه نیاز خواهد شد. در ابتدا با استفاده از تابع initiate به تعداد K تا سوپر پیکسل اولیه با مرکزهایی هم فاصله میسازیم. همچنین این تابع مقدار S را نیز برایمان محاسبه میکند. اکنون با تابع moveCenters مرکز هر خوشه را در همسایگی 5×5 به گونه ای جایجا میکنیم که در جایی قرار گیرد که کمترین گرادیان را دارد. سپس سوپر پیکسل های حاصل را در متغیر superPixelsOld کپی میکنیم و با یک for با ماکزیمم تکرار 20 در نظر میگیریم و هر بار superPixels جدید را با تابع iterate بدست میاورد و همچنین ماتریس labels را از این تابع میگیرد. و مقدار جابجایی مرکز خوشه ها را با استفاده از تابع calError محاسبه میکند و اگر این مقدار کمتر از آستانه بود، تکرار را ادامه نمیدهد. ضمنا در هر مرحله اجرای این حلقه، superPixelsOld نیز بروزرسانی میشود تا در هر مرحله، این متغیر، لیستی از خوشه های دقیقا مرحله‌ی قبل باشد. پس از اجرای کامل این حلقه، ما یک ماتریس labels داریم که برچسب پیکسل ها در آخرین مرحله است. حال با استفاده از تابع آماده‌ی martk_boundaries از کتابخانه‌ی skimage.segmentation مرز ها را بر روی تصویر اصلی رسم کرده و از آنجا که خروجی این تابع، ماتریسی با درایه های اعشاری بسیار کوچک است، آن را normalize میکنیم و نتیجه‌ی حاصل را بعنوان خروجی بازگشت میدهیم.

حال پس از توضیحات مربوط به توابع پیاده سازی شده به توضیحات روند استفاده از آنها و رسیدن به خروجی مطلوب میپردازیم.

حال کافی است برای K های 64، 256، 1024 و 2048 عکس اصلی را بخوانیم، و با صدا زدن تابع slic تصویر قطعه بندی شده را بگیریم. آلفا های در نظر گرفته شده برای هر عکس به ترتیب 0.005 و 0.01 و 0.05 و 0.1 اند.

K	64	256	1024	2048
alpha	0.005	0.01	0.05	0.1

q4

روش اول برای حل این سوال این است که در ابتدا با استفاده از تابع felzenszwalb از کتابخانه‌ی skimage.segmentation و پارامتر های scale=200 که مانند τ عمل میکند و هر

بیشتر باشد، سگمنت های بیشتری با یکدیگر ادغام میشوند. sigma=4 که مربوط به فیلتر گوسی برای بلور کردن عکس قبل از قطعه بندی میباشد. min_size=100000 هم کمترین تعداد پیکسل لازم برای تشکیل سگمنتی مجزا را تعیین میکند. پس با این پارامتر ها عکس را قطعه بندی میکنیم. خروجی این تابع label خواهد بود. که برچسب 1 ام مربوط به قطعه‌ی پرنده ها میباشد. پس بقیه‌ی label ها را سیاه کرده و قطعه‌ی باقی مانده را نمایش میدهیم.

از آنجایی که روش دقیق عمل نمیکند، کد آن در ابتدای فایل کد این سوال به صورت کامنت درآمده و نتیجه‌ی آن با نام res09-.jpg ذخیره شده و کد اصلی محسوب نمیشود. روش اصلی من برای حل این سوال در ادامه آمده است.

در این بخش چند تابع پیاده سازی شده که ابتدا توضیحشان میدهیم.

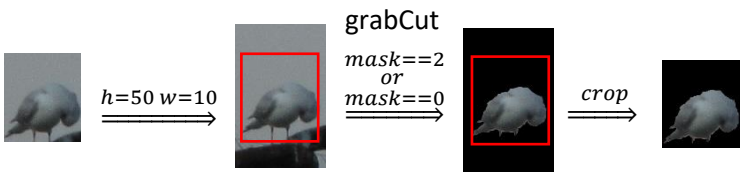
تابع cut :

ورودی ها:
1) تصویر اصلی
2) مختصات نقطه‌ی بالا سمت چپ و پایین سمت راست کادری محدود به فقط یک پرنده
3) ارتفاع و پهنای اضافی برای بک گراند grabcut

خروجی ها:
1) تصویر فقط پرنده

عملکرد: هدف این است پرنده‌ی داخل کادر داده شده را جدا کنیم. روش بهره برده شده در اینجا روش grabcut از کتابخانه‌ی cv2 میباشد که یک عکس به آن میدهیم، همچنین یک کادر بسیار محدود مستطیلی به آن میدهیم که فقط پرنده کاملا داخل این کادر باشد، سپس این روش تمام بخش های خارج از این کادر را بعنوان background حتمی تشخیص میدهد ولی قسمت‌های داخل کادر مجهول به حساب میایند. تابع cv2.grabCut با بهره گرفتن از Gaussian mixture که یک مدل سازی احتمالاتی است، سعی میکند توزیع احتمال پیکسل‌های مجهول را بدست آورد و سپس برچسب بک‌گراند یا فورگراند را به آن پیکسل اختصاص دهد که در جلسه‌ی هجدهم مباحث مربوط به آن درس داده شد. در واقع ناحیه‌ی خارج از کادر به مدل آموزش میدهد که چه پیکسل‌هایی را بک‌گراند و چه پیکسل‌هایی را فورگراند برچسب‌زنی کند. میدانیم اگر کل تصویر اصلی را به این تابع بدهیم، این تابع پرنده‌ی ما را تشخیص نمیدهد. زیرا برخی پرنده ها خارج کادر محدود به پرنده قرار میگیرند و بعنوان بک گراند تشخیص داده میشوند. پس باید تصویری که به این تابع میدهیم تصویری باشد که اولا فقط یک پرنده در آن باشد، دوما بخشی از تصویر خارج از کادر نیز وجود داشته باشد تا

بتواند از آن را بعنوان بک گراند در نظر بگیرد. دو ورودی ارتفاع و پهنای اضافی برای بک‌گراند grabcut با نام‌های h و w به همین منظور داده شده تا علاوه بر کادر داده شده در ورودی که یک کادر بسیار محدود است و فقط یک پرنده در آن جا شده، بتوان به اندازه‌ی h پیکسل به بالا و پایین کادر پرنده، و w پیکسل به چپ و راست کادر پرنده اضافه کنیم تا مدل ما طبق آن learn شود و تصویر حاصل را imgGround مینامیم. دقت کنید که این دو عدد h و w دستی به این تابع داده میشوند و باید به گونه ای باشند که به اندازه‌ی کافی فضا برای تشخیص بک گراند فراهم کنند و همچنین پرنده‌ی دیگری را در بر نگیرند تا پرنده‌ی ما بعنوان بک‌گراند تشخیص داده نشود. حال یک mask سیاه به ارتفاع و پهنای imgGround میسازیم. سپس دو متغیر bgdMode و fgdMode را بصورت np.zeros((1,65),np.float64) تعریف میکنیم و متغیر rect را نیز طوری در نظر میگیریم که همان کادر اولیه‌ی محدود به پرنده را نشان دهد. حال متغیر های ذکر شده را به تابع cv2.grabCut میدهیم که بعنوان نتیجه، mask را تغییر میدهد. در جاهایی که پرنده حضور دارد، مقدار mask را برابر 2 قرار میدهد. که ما ماسک جدیدی میسازیم که در جاهای که mask برابر با 2 یا 0 است را 1 و بقیه‌ی جاها را برابر 0 قرار دهد. اکنون اگر ماسک جدیدمان را در تصویر imgGround ضرب درایه به درایه کنیم میبینیم که فقط پرنده در این تصویر باقی میماند. حال برای آنکه ابعاد عکس را به حالت اولیه باز گردانیم به اندازه‌ی h از بالا و پایین تصویر حاصل و به اندازه‌ی w از چپ و راست تصویر برش میدهیم و نتیجه را بعنوان خروجی بازگشت میدهیم. اعمال انجام شده به صورت زیر اند.



تابع cleanList :

ورودی ها: 1) دو لیست که یکی مختصات افقی و دیگری مختصات های عمودی را نشان میدهد

خروجی ها: 1) دو لیست تمیز شده که یکی مختصات افقی و دیگری مختصات های عمودی را نشان میدهد

عملکرد: در اینجا برای ساده شدن فرض کنید که به جای دو لیست تک نقطه‌ای، یک لیست از نقاط بعنوان ورودی داده شده که زوج مرتب اند و مختصات نقاط را نشان میدهند. میخواهیم نقاطی از این لیست که به یکدیگر نزدیک اند را حذف کنیم. برای این کار ابتدا یک لیست خالی میسازیم و آن را لیست تمیز شده مینامیم. سپس نقطه‌ی اول لیست داده شده در ورودی را به لیست تمیز شده اضافه میکنیم. حال با یک for بر روی تعداد نقاط داخل لیست اولیه، بررسی میکنیم که اگر فاصله‌ی این نقطه تا یکی از نقاط موجود در داخل لیست تمیز، کمتر از یک threshold ای بود، آن نقطه را صرف نظر کن و

بررسی آن نقطه را ادامه نده. زیرا در واقع برای بررسی کردن هر نقطه از لیست کثیف باید فاصله‌ی آن را تا تمام نقاط داخل لیست تمیز چک کنیم. ولی خب اگر به یکی از این نقاط داخل لیست تمیز نزدیک باشد، ادامه‌ی بررسی این نقطه بی فایده است زیرا میدانیم این نقطه قرار نیست به لیست تمیز اضافه شود. حال اگر پس از بررسی کامل فاصله‌ی این نقطه از تمام نقاط لیست تمیز بیشتر از آستانه بود، آن را به لیست تمیز اضافه میکنیم. پس از بررسی کامل، لیستی تمیز از نقاط داریم که از بین هر دسته از نقاط نزدیک به هم، یکی از آنها بعنوان کاندید در آن باقی مانده، که آنها را بعنوان خروجی بازگشت میدهیم.

تابع `findAll` :

ورودی ها:
1) تصویر اصلی
2) تصویر هدف یا همان تصویری که قرار است فقط شامل پرنده باشد
3) یک تمپلیت که تصویر کوچکی از پرنده بعنوان نمونه است
4) ضریب آلفا
5) ارتفاع و پهنای اضافی برای بک گراند `grabcut`

خروجی ها:
1) تصویر هدف که پرنده‌های شبیه به تمپلیت به آن اضافه شده است.

عملکرد: هدف این است که تمام پرنده‌های شبیه به پرنده‌ی نمونه یا تمپلیت، به تصویر هدف اضافه شوند. برای این کار ابتدا با استفاده از تابع `cv2.matchTemplate` شباهت تمام قسمت‌های تصویر اصلی را به پرنده‌ی نمونه میابیم. سپس چون روش ما `cv2.CCOEFF_NORMED` بوده پس تمام درایه‌های ماتریس حاصل عددی بین `0` تا `1` اند و ضریب `alpha` داده شده بعنوان ورودی نیز عددی بین `0` تا `1` است. اکنون مختصات تمام قسمت هایی از تصویر که شباهت آنها در ماتریس حاصل از تمپلیت مچینگ بیشتر از `alpha` باشد را در بردارهای `imask` و `jmask` نگه میداریم. میدانیم برخی پرنده‌ها چند بار تشخیص داده میشوند. یعنی تمام قاب های حول یک پرنده که به پرنده شبیه است در این بردار ها اضافه میشود. پس با تابع `cleanList` که بالاتر تعریف شد، مختصات های تشخیص داده شده‌ی شبیه به هم را از `imask` و `jmask` حذف میکنیم. اکنون به ازای هر قاب پیدا شده که شبیه به تمپلیت اولیه بوده، با استفاده از دستور `cut` که بالاتر تعریف شد، تنهای پرنده‌ی آن قاب را بدست آورده و در متغیر `templateCut` قرار میدهیم. اگه دستور `cut` نتواند جسمی داخل این قاب را تشخیص دهد، تصویری سیاه بازمیگرداند که در این صورت آن را جایگذاری نخواهیم کرد.

حال پس از توضیحات مربوط به توابع پیاده سازی شده به توضیحات روند استفاده از آنها و رسیدن به خروجی مطلوب میپردازیم.

مختصات پنج قاب حاوی در بر دارنده‌ی پرنده از تصویر اصلی به صورت دستی انتخاب شده است

q5

در این بخش چند تابع پیاده سازی شده که ابتدا توضیحشان میدهیم.

تابع `drawContour` :

ورودی ها:
1) یک تصویر که میخواهیم کانتور را بر روی آن رسم کنیم
2) لیست مختصات نقاط روی کانتور
3) ضخامت قلم برای رسم کانتور
4) نوع کانتور که میتواند خطی `l` یا نقطه‌ای `p` یا خطی‌نقطه‌ای `p&l` باشد

خروجی ها:
1) تصویری که کانتور بر روی آن رسم شده است

عملکرد: اگر `type` یا نوع رسم `l` یا `p&l` بود، بین هر دو نقطه‌ی متوالی با استفاده از تابع آماده‌ی `line` یک خط به ضخامت قلم داده شده بعنوان ورودی، بر روی عکس داده شده، رسم میکنیم. دو نقطه‌ی متوالی را بصورت `points[p]` و `points[(p-1)%len(points)]` محاسبه میکنیم. زیرا نقاط روی کانتور به صورت دوری به یکدیگر متصل اند و باید نقطه‌ی ابتدایی به نقطه‌ی انتهایی متصل باشد. `p` نیز در `range(len(points))` است. سپس اگر نوع رسم `p` یا `p&l` بود، به مرکز مختصات هر نقطه با استفاده از تابع آماده‌ی `circle` یک دایره به شعاع ضخامت قلم داده شده بر روی عکس داده شده، رسم میکنیم.

تابع `calAvgDist` :

ورودی ها:
1) لیست مختصات نقاط روی کانتور

خروجی ها:
1) میانگین فاصله‌ی بین هر دو نقطه‌ی متوالی روی کانتور یا $\bar{d}$

عملکرد: ابتدا یک آرایه به نام `dist` میسازیم و به آن فاصله‌ی بین هر دو نقطه‌ی متوالی را اضافه میکنیم و در نهایت میانگین تمام عناصر داخل این آرایه برابر خواهد بود با میانگین فواصل بین هر دو نقطه‌ی متوالی روی کانتور که آن را بعنوان خروجی بازگشت میدهیم.

دو نقطه‌ی متوالی را بصورت `points[p]` و `points[(p-1)%len(points)]` محاسبه میکنیم. زیرا نقاط روی کانتور به صورت دوری به یکدیگر متصل اند و باید نقطه‌ی ابتدایی به نقطه‌ی انتهایی متصل باشد. `p` نیز در `range(len(points))` است.

تابع `externalE` :

ورودی ها:
1) مختصات یک نقطه

خروجی ها:
1) انرژی external نقطه‌ی داده شده

متغیر های `global`:
1) تصویر گرادیان `gradient`
2) ضریب گاما `gamma`

عملکرد: هدف این است که انرژی نقطه در صورت قرار گرفتن بر روی مرزهای تصویر، کمتر شود. برای این کار مختصات نقطه‌ی داده شده را درون تصویر گرادیان `gradient` قرار میدهد و intensity مربوط به مختصات آن نقطه را در ضریب `gamma`- ضرب کرده و عدد حاصل را بعنوان خروجی بازگشت میدهد.

تابع `internalE` :

ورودی ها:
1) مختصات دو نقطه‌ی متوالی
2) میانگین فاصله‌ی بین هر دو نقطه‌ی متوالی روی کانتور یا $\bar{d}$

خروجی ها:
1) انرژی internal نقطه‌ی داده شده

متغیر های `global`:
1) ضریب آلفا `alpha`

عملکرد: هدف این است که انرژی نقطه در صورت دور بودن از فاصله‌ی میانگین، بیشتر شود. یعنی نقاط تمایل داشته باشند که فاصله‌ی متناسبی از یکدیگر داشته باشند و این گونه نباشد که دو نقطه‌ی متوالی بسیار دور از هم و دو نقطه‌ی متوالی دیگری بسیار نزدیک بهم باشند. برای این کار مجذور فاصله‌ی دو نقطه‌ی داده شده را منهای مجذور میانگین فاصله‌ی بین نقاط متوالی روی کانتور میکند و حاصل را به توان دو رسانده و در ضریب `alpha` ضرب کرده و عدد حاصل را بعنوان خروجی بازگشت میدهد. فرمول آن بصورت زیر است:

$$\left(||v_i - v_{i-1}||^2 - \bar{d}^2\right)^2$$

تابع `centralE` :

ورودی ها:
1) مختصات یک نقطه

خروجی ها:
1) انرژی central نقطه‌ی داده شده

متغیر های `global`:
1) مختصات مرکز کانتور `center`
2) ضریب بتا `beta`

عملکرد: هدف این است انرژی نقطه در صورت دور شدن از مرکز، بیشتر شود. یعنی نقاط تمایل داشته باشند که به مرکز کانتور نزدیک شوند تا کانتور متراکم تر شود. برای این کار فاصله‌ی مختصات نقطه‌ی داده شده از مرکز را محاسبه میکند و در ضریب `beta` ضرب کرده و عدد حاصل را بعنوان خروجی بازگشت میدهد.

تابع `energy` :

ورودی ها:
1) مختصات دو نقطه‌ی متوالی
2) میانگین فاصله‌ی بین هر دو نقطه‌ی متوالی روی کانتور یا $\bar{d}$

خروجی ها:
1) انرژی نقطه‌ی داده شده

عملکرد: مجموع انرژی های external، internal و central را با استفاده از تابع های پیاده سازی شده در بالا بدست آورده و عدد حاصل را بعنوان خروجی بازگشت میدهد.

تابع `iterate` :

ورودی ها:
1) مختصات نقاط روی کانتور

خروجی ها:
1) مختصات نقاط روی کانتور در مرحله‌ی بعد

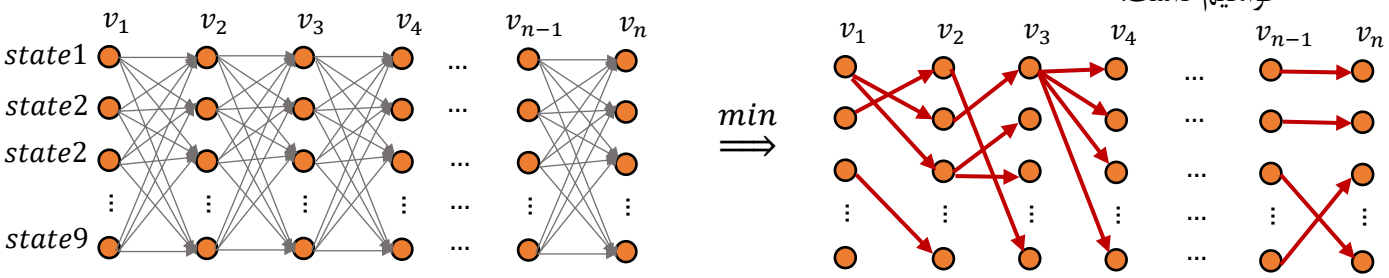
عملکرد: هدف این است نقاط داده شده بعنوان ورودی، که همان مختصات نقاط روی کانتور اند، به گونه ای جابجا شوند که انرژی کل کانتور کمترین مقدار ممکن شود. میدانیم انرژی کل کانتور را میتوانیم به صورت زیر بنویسیم:

$$E_{total}(v_1, \dots, v_n) = \sum_{p=1}^{n} E\left(v_{p-1}, v_p\right)$$

نقاط کانتور در هر بار اجرای این تابع، تنها در یک همسایگی 3×3 توانایی حرکت دارند. قبل از هر کاری متغیر `dbar` را که همان میانگین فاصله‌ی بین هر دو نقطه‌ی متوالی کانتور در این مرحله است را با استفاده از تابع `calAvgDist` که در بالا آمده، محاسبه میکنیم که در ادامه نیازمان میشود. در ابتدا یک آرایه ای از مختصات پوزیشن‌ها به نام `neighbors` میسازیم که اعضای آن بصورت زیر اند:

اندیس پوزیشن	0	1	2	3	4	5	6	7	8
مختصات پوزیشن	`[0,0]`	`[0,-1]`	`[0,1]`	`[1,0]`	`[1,-1]`	`[1,1]`	`[-1,0]`	`[-1,-1]`	`[-1,1]`

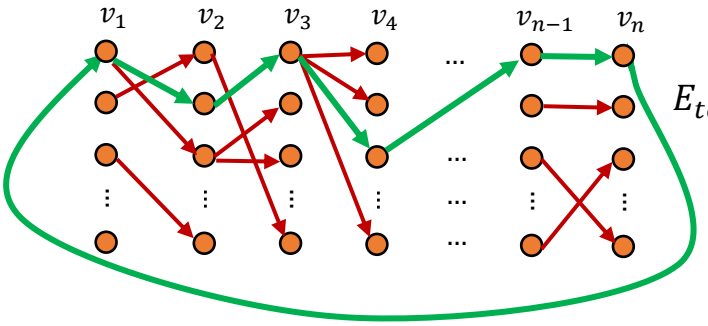
که در واقع تمام پوزیشن های ممکن در همسایگی یک نقطه است و به هر یک اندیسی نسبت داده شده. مثلا پوزیشن `0` ام مربوط به حرکت نقطه به اندازه‌ی `[0,0]` و پوزیشن `1` ام مربوط به حرکت نقطه به اندازه‌ی `[0,-1]` و ... میباشد. در این گام ما باید ببینیم که اگر نقطه‌ی `p` ام به پوزیشن `i` ام خود برود، نقطه‌ی `p-1` ام به کدام پوزیشن خود برود تا انرژی آن دو نقطه‌ی کمینه شود. پس ماتریس سه بعدی انرژی‌ها یا `energies` را با ابعاد تعداد نقطه‌ها × تعداد همسایگی × تعداد همسایگی، میسازیم که در درایه‌ی `[p,i,j]` این ماتریس، مقدار انرژی بین پوزیشن `i` ام نقطه‌ی `p` و پوزیشن `j` ام نقطه‌ی `p-1` ذخیره خواهد شد. در ابتدا تمام درایه‌های این ماتریس را بی نهایت قرار میدهیم. پس از آن ماتریس دو بعدی پوزیشن ها یا `positions` را با ابعاد تعداد نقطه‌ها × تعداد همسایگی، میسازیم که در درایه‌ی `[p,i]` این ماتریس، اندیس پوزیشنی از نقطه‌ی `p-1` ام که انرژی بین آن و پوزیشن `i` ام نقطه‌ی `p` کمینه است، ذخیره خواهد شد. پس درایه‌های این ماتریس طبق جدول اندیس پوزیشن ها که در بالا آمد، عدد صحیحی بین `0` تا `8` میباشد. در ابتدا تمام درایه های این ماتریس را `0` قرار میدهیم. برای مثال اگر داشته باشیم `positions[p,i]=j` به این معنیست که اگر نقطه‌ی v_p به پوزیشن `i` ام خود برود، نقطه‌ی قبلی یعنی v_{p-1}، باید به پوزیشن `j` ام خود برود تا $E(v_{p-1}, v_p)$ کمینه شود. پس قبل از هر چیز باید این دو ماتریس کاملا مقداردهی شوند تا بدانیم از هر پوزیشن از هر نقطه، به کدام پوزیشن از نقطه‌ی قبلی برویم تا انرژی بین آنها کمینه شود. اگر انرژی بین هر پوزیشن از نقطه‌ای و پوزیشنی از نقطه‌ی ماقبلش را با یک فلش نشان دهیم، شکلی مانند زیر خواهیم داشت:



در شکل سمت چپ تمام انرژی های ممکن بین هر دو پوزیشن ممکن از هر دو نقطه‌ی متوالی رسم شده و در شکل سمت راست تنها انرژی‌های کمینه از میان تمام انرژی‌های وارد بر یک پوزیشن، برگزیده شده است. همانطور که میبینیم فلش ها پوشا اند ولی یک به یک نیستند. یعنی به هر راس یک فلش وارد شده ولی لزوما از هر راس، یک فلش خارج نشده است که این یعنی برای رسیدن به پوزیشن i ام نقطه‌ی v_p حتما و فقط یک پوزیشن از نقطه‌ی v_{p-1} وجود دارد که $E(v_{p-1}, v_p)$ کمینه شود. ما در ماتریس energies که بالاتر تعریف کردیم، مقدار انرژی‌های بین هر دو پوزیشن از دو گره متوالی را نگهداری میکنیم. که به تعداد فلش‌های شکل سمت چپ خواهند بود. البته از آنجا که نقاط روی کانتور به صورت دوری به هم متصل اند، در شکل چپ، پوزیشن‌های نقطه‌ی v_n نیز باید به پوزیشن‌های نقطه‌ی v_1 متصل باشند ولی برای بهتر دیده شدن شکل از رسم آنها خودداری شده است. پس در مجموع به اندازه‌ی تعداد نقطه‌ها × تعداد همسایگی × تعداد همسایگی بار انرژی محاسبه میشود و کل ماتریس energies پر میشود. طبق مطالب گفته شده، برای پیاده سازی نیاز به سه for تو در تو داریم. for اول بر روی تعداد نقاط زده میشود. فرض میکنیم در گام p ام از for اول هستیم. یعنی نقطه‌ی p ام را بررسی میکنیم. نقطه‌ی فعلی را pointCur و نقطه‌ی ماقبل آن را pointPrv مینامیم که با فرمول‌های pointCur=points[p] و pointPrv=[(p-1)%len(points)] محاسبه میشوند تا به صورت دوری، نقطه‌ی انتهایی به نقطه‌ی ابتدایی آرایه متصل در نظر گرفته شود. اکنون با استفاده از for دوم، نقطه‌ی pointCur را به پوزیشن پوزیشن i ام $(0 \leq i \leq 8)$ میبریم و آن را pointCurNew مینامیم و باید ببینیم که به ازای انتخاب هر i ، کدام پوزیشن از نقطه‌ی قبلی انتخاب شود تا $E(v_{p-1}, v_p)$ کمینه شود. پس به ازای هر یک از پوزیشن‌های ممکن نقطه‌ی pointCur، متغیر minE را با مقدار اولیه‌ی بی‌نهایت میسازیم که کمترین انرژی بین پوزیشن i ام نقطه‌ی pointCur و یکی از پوزیشن‌های pointPrv است. سپس با for سوم تمام پوزیشن‌های ممکن pointPrv را بررسی میکنیم تا ببینیم کدام پوزیشن این نقطه، بهترین پوزیشن ممکن برای کمینه شدن $E(v_{p-1}, v_p)$ است. با بردن نقطه‌ی pointPrv به پوزیشن j ام $(0 \leq j \leq 8)$ نقطه‌ی حاصل را pointPrvNew مینامیم. حال انرژی بین دو نقطه‌ی pointCurNew و pointPrvNew را با استفاده از dbar که در بالا محاسبه شد و تابع energy که در بالا پیاده سازی شد، محاسبه کرده و در درایه‌ی [p,i,j] ماتریس energies نگهداری میکنیم و اگر این انرژی از انرژی کمینه یا minE کمتر بود، مقدار آن را درون minE جایگزین کرده و اندیس j را که همان اندیس پوزیشن انتخاب شده برای pointPrv است، در درایه‌ی [p,i] ماتریس positions نگهداری میکنیم. پس از اجرای کامل for ها، اکنون هر دو ماتریس energies و positions با همان تعریفی که میخواستیم کاملا مقداردهی

شدند. کد مربوط به محاسبه‌ی انرژی را درون `try` قرار میدهیم زیرا اگه همسایگی نقطه‌ای بیرون از تصویر قرار گیرد، محاسبه‌ی $E_{external}$ آن نقطه، غیرممکن میشود. پس اگر نقطه‌ای بعد از حرکت خارج از محدوده قرار گیرد باید `except: continue` اجرا شود تا از این همسایگی، صرف نظر شود. حال که هر دو ماتریس `energies` و `positions` مقداردهی شدند به گام بعدی میرویم.

ما تا ایجا کمینه انرژی مابین هر نقطه و نقطه‌ی ماقبلش را داریم و باید با استفاده از فرمول زیر، کمینه انرژی E_{total} را بیابیم. میدانیم کمینه شدن E_{total} معادل است با یافتن مسیری بر روی تصویر پایین که این مسیر باید عقبگرد باشد. یعنی از نقطه‌ی آخر به اول بازمیگردیم تا مسیر را کامل کنیم.



$$E_{total}(v_1, \dots, v_n) = \sum_{p=1}^{n} E(v_{p-1}, v_p)$$

تمام فلش‌های بالا انرژی های کمینه بین دو نقطه متوالی اند و تنها کافی است یکی از پوزیشن‌های نقطه‌ی v_n را بعنوان پوزیشن شروع در نظر بگیریم و رو به عقب حرکت کنیم. در شکل بالا پوزیشن اول نقطه‌ی v_n بعنوان نقطه‌ی شروع در نظر گرفته شده. میدانیم طبق تعاریفی که در قبل‌تر شد، فقط یک فلش به این راس حتما وارد شده است. پس با آن فلش به یکی از پوزیشن‌های نقطه‌ی v_{n-1} میرویم و از آنجا که به هر راس یک فلش وارد شده میتوانیم این کار را عقبگرد ادامه دهیم تا به نقطه‌ی v_1 برسیم و از نقطه‌ی v_1 به نقطه‌ی v_n بازگردیم که مسیری مانند مسیر بسته‌ی سبز رنگ در بالا شکل میگیرد. مجموع انرژی های روی مسیر را محاسبه میکنیم و انرژی بین نقطه‌ی ابتدا و انتها را نیز به آن اضافه میکنیم. عدد حاصل E_{total} خواهد بود. مسئله مهم در این گام، پوزیشن شروع است. یعنی از کدام پوزیشن نقطه‌ی v_n شروع کنیم تا E_{total} کمینه شود. . بدون کاسته شدن از کلیت مسئله، فرض شده که نقطه‌ی شروع، نقطه‌ی v_n است. زیرا بهرحال مسیر ما باید از یکی از پوزیشن‌های این نقطه بگذرد و چون مسیرمان حالت دوری دارد فرقی نمی‌کند کدام نقطه را بعنوان نقطه‌ی شروع برگزینیم ولی مهم است که کدام پوزیشن از این نقطه را بعنوان پوزیشن شروع برگزینیم. در کل 9 حالت برای تعیین مسیر داریم. زیرا 9 پوزیشن مختلف برای شروع از نقطه‌ی v_n وجود دارد. پس هر یک از این 9 حالت را چک میکنیم و به ازای هر حالت، E_{total} را محاسبه میکنیم و کمترین مقدار آن را می‌یابیم. برای این کار متغیری به نام `minTotalE` با مقدار اولیه‌ی بی نهایت میسازیم که همان کمترین مقدار E_{total}

است. همچنین متغیر bestStartPose با مقدار اولیه‌ی صفر را میسازیم که بهترین پوزیشن v_n برای شروع را نشان میدهد. اکنون یک for روی تعداد حالت‌های ممکن برای انتخاب پوزیشن شروع میزنیم. فرض میکنیم از پوزیشن i ام نقطه‌ی v_n شروع کرده ایم. متغیری به نام totalE با مقدار اولیه‌ی صفر میسازیم که انرژی کل مسیر طی شده را با فرض شروع از این پوزیشن، در نهایت به ما خواهد داد. همچنین متغیری به نام poseCur با مقدار اولیه‌ی i میسازیم که اندیس پوزیشن فعلی که در حین طی کردن مسیر روی آن قرار داریم را نشان میدهد. چون فرض کردیم از پوزیشن i ام v_n شروع کردیم و هنوز مسیری نرفته ایم. پس پوزیشن فعلی باید مقدار اولیه‌ی i را داشته باشد. اکنون یک for دیگر بر روی تعداد نقاط منهای یک میزنیم تا بتوانیم مسیری به طول len (points) -1 را بین نقاط مطابق مسیر سبز رنگ شکل بالا تا رسیدن به v_1 طی کنیم. همانطور که میدانیم این مسیر عقبگرد است. برای اینکه بدانیم از پوزیشن فعلی نقطه‌ی v_p که در آن قرار داریم به کدام پوزیشن نقطه‌ی v_{p-1} برویم تا کمترین انرژی صرف شود، از ماتریس positions استفاده میکنیم آن اندیس را posePrv مینامیم. سپس انرژی بین پوزیشن poseCur نقطه‌ی p ام و پوزیشن posePrv نقطه‌ی p-1 را با استفاده از ماتریس energies بدست آورده و عدد حاصل را به انرژی کل مسیر طی شده تا اینجا، یعنی totalE اضافه میکنیم و به پوزیشن نقطه‌ی قبلی حرکت میکنیم. یعنی پوزیشن فعلی یا poseCur، به posePrv تغییر میابد. تاکید میشود که چون مسیر عقبگرد است هر بار به پوزیشن نقطه‌ی قبلی میرویم. پس از اجرای کامل این for میبینیم که به نقطه‌ی v_1 رسیده ایم. اکنون پوزیشن متعلق به v_1 که در آن قرار داریم در متغیر poseCur ذخیره شده است. این مقدار را به پوزیشن شروع v_n وصل میکنیم و انرژی بین آن دو، با توجه به این که فرض کردیم از پوزیشن i ام v_n شروع کردیم، که برابر با energies[0,poseCur,i] است را به totalE اضافه میکنیم. انرژی totalE حاصل، انرژی یک کانتور بسته خواهد بود حال آن را با مینیمم انرژی بدست آمده تا اینجای کار مقایسه میکنیم. اگر کمتر از minTotalE بود، مقدار totalE بدست آمده را در متغیر minTotalE جایگزین کرده و بهترین پوزیشن شروع bestStartPose را برابر i قرار میدهیم. پس در این گام مشخص کردیم که بهترین پوزیشن شروع از نقطه‌ی v_n کدام پوزیشن است. مسیری که در امتداد این پوزیشن است و یکتا است، بهترین مسیر برای جایجایی نقاط خواهد بود.

با توجه به اینکه ما بهترین پوزیشن شروع از v_n، که منجر به بهترین مسیر میشود را داریم، در این گام که گام آخر است، نقاط روی کانتور را با توجه به بهترین مسیر یافت شده جابجا میکنیم. پس در ابتدا نقطه‌ی v_n را به پوزیشن bestStartPose میبریم و باز هم به صورت عقبگرد عمل میکنیم.

حال میدانیم مقدار درایه‌ی `[p,i]` از ماتریس `positions` به ما میگوید که اگر نقطه‌ی `p` ام به پوزیشن `i` ام خود برود، نقطه‌ی `p-1` ام باید چه پوزیشنی برود تا $E(v_{p-1}, v_p)$ مینیمم شود. پس با این استدلال متغیری به نام `poseCur` با مقدار اولیه‌ی `bestStartPose` میسازیم و با یک `for` روی تعداد نقاط کانتورمان، در گام `p` ام، نقطه‌ی `p` را با توجه به `poseCur` جابجا میکنیم. مقدار `poseCur` هم در هر گام با مقدار `positions[p,poseCure]` جایگزین میشود تا بهترین پوزیشن نقطه‌ی قبلی هر مرحله در متغیر `poseCur` ذخیره شود تا در گام بعد هم بتوان نقطه را با توجه به `poseCur` جابجا کرد. پس از اجرای کامل این `for` تمام نقاط روی کانتور جابجا شده اند و میتوانیم این نقاط با مختصات جدیدشان را بعنوان خروجی بازگشت دهیم.

تابع `click`:

ورودی ها:
1) دکمه‌ای که با فشردن آن رخدادی رخ میدهد
2) مختصات نقطه‌ی کلیک شده `x,y`
3) متغیر های `p1` و `p2` که بطور پیش فرض باید در این تابع قرار گیرنده ولی نیاز نمیشوند

متغیر های `global`:
1) مختصات نقاط `points` روی کانتور
2) تصویر که بر روی آن کلیک میکنیم
3) متغیر وضعیت `status` که دو مقدار `"first click"` یا `"not first click"` را دارد
4) مختصات مرکز تسبیح `center`

عملکرد: اولین کلیک برای تعیین مرکز تسبیح است. یعنی کاربر باید یک نقطه داخل تسبیح که فکر میکند به طور تقریبی در وسط تسبیح قرار داد را انتخاب کند. کلیک های بعدی برای تعیین نقاط روی کانتور اند که ترتیبشان باید حفظ شود. پس اگر دکمه‌ی فشرده شده کلیک چپ ماوس بود، و اگر `status` مقدار `"first click"` را داشت، به این معنیست که اولین کلیک انجام شده و باید با این کلیک مرکز یا `center` تسبیح تعیین گردد مرکز انتخاب شده را با دایره‌ای سیاه با شعاع 4 با تابع `circle` رسم میکنیم و متغیر `status` را به مقدار `"not first click"` تغییر میدهیم تا کلیک های بعدی برای کانتور در نظر گرفته شوند.

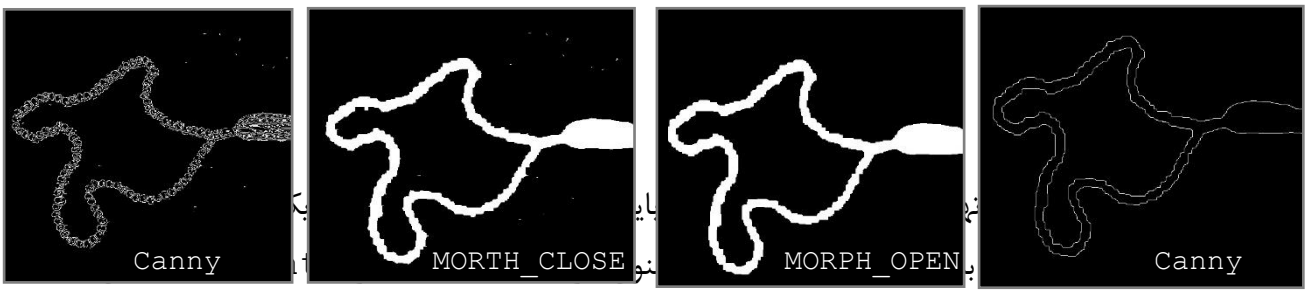
مرکز تسبیح را چشمی و به صورت تقریبی انتخاب کنید. مثل:

پس از آن با هر بار کلیک، مختصات نقطه‌ی انتخاب شده را به آرایه‌ی points اضافه یا append میکنیم و با استفاده از تابع آماده‌ی circle یک دایره به شعاع 2 و مرکز مختصات کلیک شده بر روی عکس، رسم میکنیم تا نقطه‌ی کلیک شده نمایان شود. سپس تصویر "tasbih" حاصل را با دستور cv2.imshow بروزرسانی و نمایش میدهیم.

حال پس از توضیحات مربوط به توابع پیاده سازی شده به توضیحات روند استفاده از آنها و رسیدن به خروجی مطلوب میپردازیم.

قبل از هر چیز ضرایب انرژی‌ها را تعیین میکنیم که بصورت global در توابع مربوط به انرژی مورد استفاده قرار میگیرند. مقدار gamma=100 و alpha=0.04 و beta=1 خواهد بود. سپس تصویر تسبیح را در متغیر Iclick نگه میداریم و یک لیست خالی برای مختصات نقاط کانتور به نام points میسازیم. این دو پارامتر، متغیرهای global تابع click خواهند بود. متغیر status را در وضعیت "first click" قرار میدهیم. زیرا اولین کلیک که برای تعیین مرکز هست، هنوز رخ نداده. متغیر center را نیز با مقدار اولیه‌ی [0,0] میسازیم. دو متغیر status و center نیز از متغیر های global تابع click اند. پس از آن تصویر خوانده شده را نمایش داده و سپس با استفاده از تابع cv2.setMouseCallBack با هر بار کلیک بر روی این تصویر تابع click که بالاتر پیاده سازی شد، اجرا میشود و با اولین کلیک مرکز تقریبی تسبیح توسط کاربر انتخاب شده و پس از آن با هر بار کلیک، یک نقطه‌ی جدید برای کانتور به آرایه‌ی points اضافه میشود. نقاط باید به ترتیب و دنباله وار اضافه شوند و نباید به صورت پراکنده انتخاب شود. پس از انتخاب تمام نقاط دلخواه برای کانتور، پنجره‌ی نمایش داده شده‌ی تصویر را ببندید و منتظر باشید. اکنون پورسه اصلی آغاز میشود. در ابتدا تصویر تسبیح را در متغیر I ذخیره میکنیم. و سپس ابعاد آن را در متغیر های height، width و channels نگهداری میکنیم. سپس گرادیان تصویرمان را با استفاده از تابع cv2.Canny محاسبه میکنیم. این تابع گرادیان را بصورت باینری محاسبه میکند. یعنی هر نقطه‌ای یا بر روی لبه‌های تصویر قرار دارد یا ندارد. پس خروجی یک عکس سیاه است که لبه‌های تسبیح با خطوط سفید تعیین شده اند. پارامتر های تابع Canny را با آزمون و خطا به گونه ای تعیین میکنیم که نویز های اطراف به کمترین حد خود برسند به طوری که لبه‌های تسبیح به خوبی تشخیص داده شوند. تصویر گرادیان حاصل را در متغیر gradient نگه میداریم. اما همچنان همانطور که در شکل پایین سمت چپ میبینیم این تصویر نویز دارد و تسبیح با خطوطی غیر بسته تشخیص داده شده. برای بسته شدن تسبیح با استفاده از تابع cv2.morphologyEx و ماتریس kernel ای با ابعاد 10×10 که تمام درایه‌های آن یک است، تصویر تسبیح را MORPH_CLOSE میکنیم. سپس برای

از بین بردن نویز های اطراف تسبیح بار دیگر با همان kernel و همان تابع تصویر تسبیح را MORPH_OPEN میکنیم. اکنون کل تسبیح سفید شده است ولی ما مرزهای آن را نیاز داریم. پس بار دیگر از تصویر حاصل با تابع Canny گرادیان گرفته تا مرزهای تسبیح به ما داده شوند. مراحل طی شده، به ترتیب نتایج زیر را بدست میدهند که در نهایت تصویر gradient تصویر نهایی سمت راست بوده و متغیر global برای تابع externalE خواهد بود.



مختصات نقاط روی کانتور اند را با تابع iterate که بالاتر تعریف شد، بروزرسانی میکنیم. در هر بار بروز رسانی، نقاط در یک همسایگی 3×3 در اطراف خود جابجا میشوند به طوری که انرژی E_{total} کمترین مقدار در آن مرحله شوند. هر بار که نقاط را بروزرسانی کردیم، با استفاده از تابع drawContour نقاط را بر روی تصویر رسم کرده و به لیست تصاویر imagesList اضافه میکنیم تا هر مرحله در ویدیوی نهایی نمایان باشد. پس از اجرای کامل این حلقه، به تعداد لازم تکرار انجام شده، و تمام تصاویر در imagesList ذخیره شده اند. اکنون با تابع cv2.VideoWriter با فرمت mp4 و سرعت 60 fps و به ابعاد height و width یک خروجی ویدیو میسازیم و هر تصویر از لیست imagesList را در آن write میکنیم و در نهایت ویدیو را ذخیره میکنیم و همچنین اکنون آرایه‌ی points پس از اجرای for اخیر، به آخرین مقدارش بروزرسانی شده و کافی است آن را با تابع drawContour بر روی تصویرمان رسم کنیم و خروجی را بعنوان عکس نهایی سگمنت شده، ذخیره کنیم. عکسی دلخواهی از تسبیح که باید از کاربر گرفته شود را با نام "tasbih.jpg"در کنار فایل این کد قرار دهید تا تمام اتفاقات بالا برای آن تصویر رخ دهد. همچنین برای اجرا و نتایج بهتر لطفا تعداد نقاط زیادی انتخای کنید. مثلا حداقل 60 نقطه که با 60 کلیک میتوان تعیین کرد. همچنین اگر هر جفت نقطه‌ی متوالی شما تقریبا هم فاصله باشند، نتیجه بهتر هم خواهد شد.