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

تابع `findTriangles` :

ورودی ها:
1) لیست مثلث‌های تصویر اول
2) لیست مختصات نقاط landmark تصویر اول
3) لیست مختصات نقاط landmark تصویر دوم

خروجی ها:
1) لیست مثلث‌های تصویر دوم

عملکرد: در اینجا منظور از تصویر اول، تصویریست که مثلث بندی شده است و منظور از تصویر دوم، تصویریست که میخواهیم آن را متناظراً مثلث بندی کنیم. همچنین منظور از مثلث، شش‌تایی مرتبی است که مختصات سه رأس آن را نمایش میدهد. هدف این است که تصویر دوم را با الگو برداری از مثلث‌های تصویر اول متناظراً مثلث بندی کنیم. یعنی به ازای هر مثلث از تصویر اول ، با حفظ ترتیب، نظیرش یک مثلث در تصویر دوم ایجاد کنیم. میدانیم ترتیب نقاط در لیست نقاط landmark هر دو تصویر یکسان میباشد که این خاصیت تابع `shape_predictor` است که جلوتر شرح میدهیم. پس یعنی اگر مثلث `i` ام در تصویر اول توسط نقاط `n1` ام، `n2` ام و `n3` ام از لیست نقاط تصویر اول تشکیل شده باشد، باید با نقاط `n1` ام، `n2` ام و `n3` ام از لیست نقاط تصویر دوم، مثلث `i` ام را در لیست مثلث‌های تصویر دوم ایجاد کنیم. پس برای این کار یک `for` روی تعداد مثلث‌های تصویر اول میزنیم و هر بار مختصات سه راس مثلث `i` ام آن را در سه متغیر `p1`، `p2` و `p3` نگه میداریم. حال باید ببینیم که هر یک از این سه مختصات مربوط به چندمین نقطه از لیست نقاط تصویر اول میباشند. پس شماره‌ی اندیس هر یک از سه نقطه‌ی `p1`، `p2` و `p3` در لیست نقاط تصویر اول را با استفاده از دستور `np.where` میابیم و به ترتیب در سه متغیر `n1`، `n2` و `n3` نگه میداریم. اکنون میتوانیم مختصات نقاط `n1` ام، `n2` ام و `n3` ام از لیست نقاط تصویر دوم را بعنوان یک مثلث به لیست مثلث‌های تصویر دوم اضافه کنیم و پس از اجرای کامل حلقه، این لیست مثلث ها را بعنوان خروجی بازگشت دهیم. این روش، ترتیب مثلث های داخل لیست را حفظ میکند. یعنی مثلث `i` ام از لیست مثلث‌های تصویر اول، با مثلث `i` ام از لیست مثلث‌های تصویر دوم متناظر است.

تابع `iterate` :

ورودی ها:
1) تصویر مبدا و لیست مثلث‌هایش
2) تصویر مقصد و لیست مثلث‌هایش
3) لیست مثلث‌های تصویر میانه
4) یک عدد `a` که وزن ترکیب دو تصویر را نشان میدهد

خروجی ها: 1) تصویر میانه

عملکرد: ما در اینجا تصویر میانه را نداریم ولی مثلث بندی آن را داریم و باید به کمک آن و همچنین به کمک تصویر مبدا و تصویر مقصد، تصویر میانه را به گونه ای بسازیم که اجزای صورت تصویر میانه به درستی حاصل شود. اجزای هر صورت در هر تصویر، توسط مثلث بندی آن تصویر مشخص میشود و از آنجا که مثلث بندی هر سه تصویر مبدا، مقصد و میانه را داریم، کافی است هر مثلث i ام از تصویر مبدا و مقصد را به گونه ای warp کنیم که بر روی مثلث i ام در مثلث بندی تصویر میانه قرار گیرند. لازم به ذکر است که لیست مثلث‌های داده شده در ورودی، همگی مرتب اند. یعنی مثلث i ام در هر سه لیست مثلث‌های تصویر مبدا، مقصد و میانه متناظر اند یک جزء مشترکی از چهره را نشان میدهد. پس با یک for روی تعداد مثلث‌های لیست، مختصات سه راس مثلث i ام از لیست مثلث‌های تصویر مبدا را در متغیر t1، مختصات سه راس مثلث i ام از لیست مثلث‌های تصویر مقصد را در متغیر t2 و مختصات سه راس مثلث i ام از لیست مثلث‌های تصویر میانه را در متغیر t3 نگه میداریم و باید پیکسل‌های داخل مثلث t3 را با مقادیر مناسب پر کنیم. پس با استفاده از تابع آماده‌ی getAffineTransform ماتریسی که میتواند مثلث t1 را به t3 نگاشت دهد را بدست میاوریم و در متغیر matrix1 نگه میداریم و ماتریسی که میتواند مثلث t2 را به t3 نگاشت دهد را بدست میاوریم و در متغیر matrix2 نگه میداریم. اکنون تصویر مبدا را تحت matrix1 و تصویر مقصد را تحت matrix2 نگاشت میدهیم و به ترتیب در دو متغیر imgWarped1 و imgWarped2 نگه میداریم. ولی ما تنها به قسمتی از این دو تصویر نگاشت یافته نیاز داریم که در داخل مثلث t3 باشند. زیرا در پی پر کردن پیکسل‌های داخل مثلث t3 هستیم. پس یک mask تعریف میکنیم. این mask یک تصویر سیاه به ابعاد تصویر مبدا یا مقصد است که در آن یک مثلث سفید به مختصات راس‌های مثلث t3 با استفاده از تابع آماده‌ی fillPoly رسم شده است. اکنون با دستور np.where تمام محدوده‌ای که mask در آن سفید هست را انتخاب میکنیم و فقط همین محدوده‌ی مثلثی شکل از دو تصویر نگاشت یافته را برش میدهیم و بصورت میانگین وزن دار، با وزن a برای تصویر مقصد و وزن 1-a برای تصویر مبدا، آن دو را جمع میکنیم و در همان محدوده از تصویر میانی قرار میدهیم. عمل میانگین وزن دار برای آن است که رنگ پیکسل‌های درون مثلث t3 در تصویر میانی ترکیبی وزن دار از رنگ پیکسل‌های همین ناحیه از تصویر مبدا و مقصد باشد. از آنجایی که کل تصویرها مثلث بندی شده اند و هر پیکسل از تصاویر حتما درون یکی از مثلث ها قرار دارند و هیچ پیکسلی از این تصاویر نیست که مثلث بندی نشده باشد یا خارج ناحیه مثلث بندی شده قرار گرفته باشد،

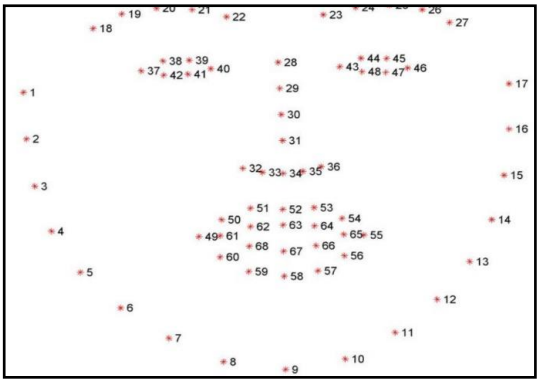
پس کل تصویر میانی پس از پر شدن تمام مثلث هایش، مقدار دهی میشود و میتوان آن را بعنوان خروجی بازگشت داد.

تابع `morph` :

ورودی ها:
1) تصویر مبدا
2) تصویر مقصد

خروجی ها:
1) لیست تمام تصویرهای میانی از مبدا تا مقصد

عملکرد: قبل از هر چیز باید دیتاستی از این لینک دانلود شود و کنار فایل کد قرار گیرد تا بتوان نقاط روی صورت را توسط آن پیدا کرد. برای اجرای این برنامه به دو تصویر با ابعاد یکسان نیاز داریم که یکی تصویر مبدا `img1` و دیگری تصویر مقصد `img2` میباشد که طی انجام عمل مورفینگ، در ویدیوی نهایی، تصویر مبدا باید به تصویر مقصد تبدیل شود. در ابتدا با استفاده از تابع آماده‌ی `dlib.get_frontal_face_detector` یک `detector` میسازیم که کار تشخیص دادن چهره‌های عکس را انجام میدهد. سپس با استفاده از تابع آماده‌ی `dlib.shape_predictor` یک `predictor` میسازیم. نام دیتاست ذکر شده را باید بعنوان ورودی به این تابع بدهیم تا با آن روش اجزای صورت را نقطه دهی کند که در دیتاست آورده شده، روش 68 نقطه‌ای بکار گرفته میشود. اکنون هر یک از دو تصویر مبدا و مقصد را به `detector` ساخته شده پاس میدهیم و بعنوان خروجی لیستی از مستطیل هایی به ما میدهد که مشخص کننده‌ی تمام چهره‌های درون هر تصویر میباشد ولی از آنجا که غالبا تصاویر مبدا و مقصد انتخاب شده در این سوال، هر یک حاوی فقط یک چهره میباشند، پس مولفه‌ی صفرم این لیست‌ها را به ترتیب در متغیرهای `rect1` و `rect2` نگه میداریم. پس `rect1` مستطیل حاوی چهره در تصویر مبدا و `rect2` مستطیل حاوی چهره در تصویر مقصد است. اکنون تصویر مبدا `img1` و مستطیل `rect1` را بعنوان ورودی به `predictor` میدهیم و لیست مرتبی از نقاط لندمارک landmark چهره را بعنوان خروجی میگیریم و در لیست `points1` ذخیره میکنیم. همین کار را برای تصویر مقصد نیز انجام داده و نقاط لندمارک landmark چهره آن را نیز در لیست `points2` ذخیره میکنیم.



ترتیب نقاط داخل لیست points1 و points2 یکسان است و نقطه‌ی i ام از هر دو لیست با یکدیگر متناظر اند زیرا هنگام predict کردن با دیتاست ذکر شده، نقاط لندمارک همواره با ترتیبی که در تصویر روبرو آمده است در لیست نقاط قرار میگیرند.

درایه‌های دو لیست points1 و points2 نقاطی از جنس object اند. برای آنکه مختصات آنها را داشته باشیم از تابع آماده‌ی face_untils.shape_to_np از کتابخانه‌ی imutils استفاده شده. سپس برای سهولت استفاده از این لیست‌ها جای مختصات x و y این نقاط را تعویض میکنیم. در گام بعدی میخواهیم مثلث بندی‌ای روی این نقاط انجام دهیم ولی میدانیم نقاط درون این لیست‌ها تنها بر روی چهره قرار دارند و پس از مثلث بندی کردن آنها، مثلث‌ها کل عکس را پوشش نخواهد داد بلکه فقط چهره مثلث بندی میشود. پس مختصات چهار نقطه‌ی گوشه‌های عکس و چهار نقطه‌ی وسط اضلاع عکس را نیز به لیست points1 و points2 اضافه میکنیم. همچنین برای افزایش دقت مورفینگ یک سری نقاط به طور دستی نیز برای تعیین گوش ها و موها در دو متغیر morePoints1 و morePoints2 قرار گرفته است که میتوانید برای اتوماتیک شدن کد آن خط را کامنت کنید. همچنین نمایش تصویری این نقاط دستی اضافه شده در دو فایل 1ExtraPoints.jpg و 2ExtraPoints.jpg در فایل zip آورده شده است.
اکنون یک مستطیل به ابعاد تصویرهایمان ساخته و بعنوان ورودی به تابع cv2.Subdiv2D میدهیم و خروجی را در متغیر subdiv ذخیره میکنیم. از این دستور برای مثلث بندی استفاده میشود. حال باید تصویر مبدا را مثلث بندی کنیم. برای این کار نقاط لندمارک درون لیست points1 را با دستور insert یکی یکی به subdiv اضافه میکنیم و سپس با دستور getTriangleList لیست مثلث بندی حاصل برای تصویر مبدا را در لیست triangles1 ذخیره میکنیم. هر مثلث درون این لیست در واقع با یک شش‌تایی مرتب تعریف شده است که مختصات سه راس مثلثی را نشان میدهد. معیار subdiv برای مثلث بندی این است که مثلثی با زاویه‌ی بسیار بزرگ ساخته نشود و از روش Delaunay استفاده میکند. پس نمیتوان تصویر مقصد را هم با همین روش مثلث بندی کرد زیرا این معیار در دو تصویر ممکن است متفاوت باشد و باعث شود مثلث‌های دو تصویر مبدا و مقصد متناظر یکدیگر نشوند. پس برای آنکه تصویر مقصد را

بصورت متناظر با مثلث بندی تصویر مبدا مثلث بندی کنیم، از تابع findTriangles که بالاتر تعریف شد استفاده میکنیم و بعنوان ورودی لیست مثلث‌های تصویر مبدا triangles1، لیست نقاط لندمارک تصویر مبدا points1 و لیست نقاط لندمارک تصویر مقصد points2 را به این تابع میدهیم تا تصویر مقصد را با الگو برداری از تصویر مبدا، مثلث بندی کند. مثلث بندی حاصل برای تصویر مقصد را در لیست triangles2 ذخیره میکنیم. حال لیست خالی‌ای به نام imagesList میسازیم که تصویر هر frame از ویدیو در آن ذخیره خواهد شد. تصویر مبدا را بعنوان اولین عکس به آن اضافه میکنیم. میخواهیم k=50 تا عکس میانه از تصویر مبدا تا مقصد بسازیم. میدانیم نقاط درون لیست points1 و points2 دارای ترتیب مشخصی اند. پس اگر لیست نقاط لندمارک تصویر مقصد را منهای لیست نقاط تصویر مبدا کنیم، لیستی از بردارها بدست میاید که مولفه‌ی i ام آن در واقع بردار واصل نقطه‌ی لندمارک i ام از تصویر مبدا به نقطه‌ی لندمارک i ام از تصویر مقصد میباشد. آن را در لیست v ذخیره میکنیم. در واقع برای آنکه مکان نقطه‌ی لندمارک i ام از تصویر مبدا پس از 50 تکرار به مکان نقطه‌ی لندمارک i ام از تصویر مقصد برود، در هر گام باید به اندازه‌ی 1/50 بردار i ام از لیست v، به مختصاتش اضافه شود تا پس از 50 تکرار، به مقصد منتقل شده باشد. چیزی که برای ساختن عکس میانه نیاز داریم مثلث بندی نقاط لندمارک روی تصویر میانه است. برای بدست آوردن نقاط لندمارک روی تصویر میانه‌ی i ام، عدد a=i/k را به ازای $0 \leq i \leq k$ در بردار v ضرب میکنیم و بعلاوه‌ی points1 میکنیم و حاصل را، که همان لیست نقاط لندمارک روی تصویر میانه‌ی i ام میباشد، در لیست points3 ذخیره میکنیم. اکنون مثلث بندی را نیز برای این تصویر میانه انجام میدهیم که برای این کار از تابع findTriangles که بالاتر تعریف کردیم استفاده میکنیم و باز هم مثلث بندی را با الگو برداری از مثلث بندی تصویر مبدا انجام میدهیم تا مثلث هایشان متناظر باشند. اکنون با استفاده از تابع iterate که بالاتر تعریف شد با ورودی‌های تصویر مبدا و مقصد و لیست مثلث‌هایشان و همچنین لیست مثلث‌های تصویر میانه و همچنین عدد a=i/k بعنوان وزن ترکیب، تصویر میانه‌ی i ام را میسازیم و آن را به لیست imagesList اضافه میکنیم و این کار را 50 بار انجام میدهیم تا 50 تصویر میانی همگی ساخته شوند. از آنجایی که مثلث بندی‌های تصویر میانه و تصویر مقصد با الگو برداری از مثلث بندی تصویر مبدا انجام شده اند، ترتیب مثلث‌های داخل لیست triangles1، triangles2 و triangles3 یکسان است و مثلث i ام از هر سه لیست با یکدیگر متناظر اند. اکنون لیست تصاویر ساخته شده یعنی imagesList 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.

ابتدا دو تصویر با ابعاد برابر را انتخاب کرده و به تابع `morph` که تعریف کردیم، پاس میدهیم و بعنوان خروجی، لیست تمام تصاویر میانی برای تبدیل تصویر مبدا به مقصد را بدست میاوریم و در متغیر `imagesList` نگه میداریم. حال با تابع آماده‌ی `cv2.videoWriter` تمام تصاویر درون لیست `imagesList` را به صورت پیشرو `write` میکنیم و یکبار هم تمام تصاویر درون همین لیست را به صورت پسرو `write` میکنیم تا به شکل ویدیوی بومرنگی، تصویر مبدا ابتدا به تصویر مقصد مورف شود و سپس دوباره به حالت اولیه‌ی خود بازگردد.