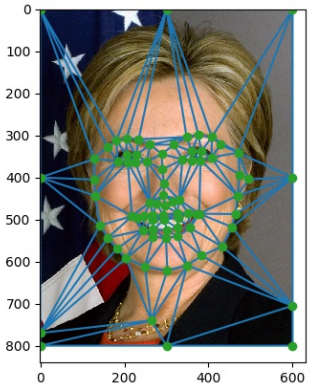
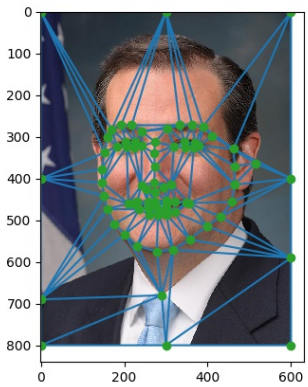
فایل helper.py

همراه تمامی کدها، یک فایل helper.py قرار دارد که به منظور خوانایی کدهاست. تابع هایی که در فایل اصلی کدها هستند (Qi.py) در فایل helper پیاده سازی میشوند تا کد اصلی کوتاه تر و خواناتر باشد. در توضیحات سوال ها، هم کدهای اصلی هم تابع های پیاده سازی شده را تا جای ممکن توضیح میدهم.

ابتدا نقطه ها رو روی تصویر اولی بدست آوردم و سپس متناظر همان نقاط رو در تصویر آخر هم بدست آوردم.
بعد از اون همانطور که در کلاس و صورت سوال گفته شد، از لایبرری Delaunay برای مثلث بندی شکل استفاده کردم. دقت شود که روی یکی از شکل ها(تصویر اول) مثلث بندی را انجام میدهیم و سپس همان را برای هر دو تصویر استفاده میکنم.





مشاهده می شود که چطور مثلث ها در دو شکل متناظر یکدیگر هستند.
خروجی مثلث بندی، به ما میگوید که کدام نقطه ها(با استفاده از اندیس نقاط) با یکدیگر تشکیل مثلث میدهند. به صورت آرایه ای از tuple های سه تایی.

حال نیاز داریم که در بین تصویر اول و آخر تعدادی تصویر میانی بدست آوریم. بدین منظور از یک for استفاده میکنم که در هر مرحله باتوجه به مقدار k تصویر میانی را بدست می آورد و در یک آرایه میریزد. در نهایت با استفاده از این آرایه یک تصویر gif میسازم.

دقت شود که k در واقع همان نسبت k به m است که در توضیحات صورت سوال آمده است.

حال توضیح میدهم که پیداکردن تصویر میانی چگونه انجام می شود. این کار را تابع

git_middle_morphing_image انجام میدهد که در helper.py پیاده سازی شده است.

ابتدا فقط مثلث ها را وارپ کردم که در ریزالت قسمت هایی خالی (سیاه) وجود داشت. بعد از سرچ کردن در اینترنت، فهمیدم که باید با مستطیل ها کار بکنم و در نهایت یک مثلث رو نگه دارم. بدین صورت:

ابتدا مختصات نقاط مستطیل ها را بدست می آورم. (در قسمت قبل بدست آوردیم که هرکدام از مثلث ها از کدامین نقاط هستند(مثلا نقطه اول و بیستم و سی ام) حال میخواهیم مختصات آن ها را بدست آوریم:

```
first_image_triangle = [first_image_points[triangle[0]], first_image_points[triangle[1]],
                        first_image_points[triangle[2]]]
# for a triangle, first_image_triangle is something like: [array([300,   0]), array([181, 312]), array([0, 0])]
last_image_triangle = [last_image_points[triangle[0]], last_image_points[triangle[1]],
                       last_image_points[triangle[2]]]

# get middle image correspond triangle
middle_image_triangle = get_middle_triangle(k, first_image_triangle, last_image_triangle)
```

برای نقاط مثلث عکس میانی، یک تابع جدا نوشته ام:

```
def get_middle_triangle(k, first_image_triangle, last_image_triangle):
    """
    According to first and last triangles, calculates the coordinates of triangle in middle image

    :param k: ratio step / number of middle images
    :param first_image_triangle: triangle in first image
    :param last_image_triangle:  correspond triangle in last image
    :return: triangle in middle image
    """
    middle_triangle = []
    for vertex in range(3):
        middle_vertex = [k * last_image_triangle[vertex][0] + (1 - k) * first_image_triangle[vertex][0],
                         k * last_image_triangle[vertex][1] + (1 - k) * first_image_triangle[vertex][1]]
        middle_triangle.append(middle_vertex)

    return middle_triangle
```

که عملکرد آن همان گونه است که در توضیحات سوال گفته شده است.

حال به ازای هر مثلث، یک مستطیل که محاط به آن است پیدا میکنم. یعنی کوچکترین مستطیلی که مثلث را کاملا در برگیرد:

```
# get min boundary rectangle for first, last and middle triangles
first_image_rectangle = cv2.boundingRect(np.float32([first_image_triangle]))
last_image_rectangle = cv2.boundingRect(np.float32([last_image_triangle]))
middle_image_rectangle = cv2.boundingRect(np.float32([middle_image_triangle]))
```

مسئله بعدی این است که مستطیل هایی که در بالا بدست آوردم مختصاتشان مختصات تصویر است. اما برای آنکه راحت تر کار کنیم(برای وارپ کردن نیاز به همان مستطیل به تنهایی داریم و مهم نیست که کجای عکس ما قرار دارد) نقاط مثلث را طوری تغییر می دهیم که نقطه بالا سمت چپ مستطیل (0,0) باشد:

```
# change dimension of rectangles: change top-left corner coordinate of rectangle to (0,0)
first_image_rectangle_new_dimension = change_rectangle_dimension(first_image_triangle, first_image_rectangle)
last_image_rectangle_new_dimension = change_rectangle_dimension(last_image_triangle, last_image_rectangle)
middle_image_rectangle_new_dimension = change_rectangle_dimension(middle_image_triangle, middle_image_rectangle)
```

که تابع استفاده شده پایین تر پیاده سازی شده است:

```
def change_rectangle_dimension(triangle, rectangle):
    """
    Change dimension of rectangles to top-left corner coordinate of rectangle be (0,0)

    :param triangle:
    :param rectangle:
    :return:
    """
    rectangle_new_dimension = []
    for i in range(0, 3):
        rectangle_new_dimension.append(((triangle[i][0] - rectangle[0]), (triangle[i][1] - rectangle[1])))

    return rectangle_new_dimension
```

مسئله بعدی کراپ کردن تصویرها برای بدست آوردن مستطیل ها است. (قسمتی از تصویر که مستطیل ما قرار دارد را فقط نیاز داریم):

```
# get rectangle part of images
x_first_image_rectangle = first_image_rectangle[0]
y_first_image_rectangle = first_image_rectangle[1]
width_first_image_rectangle = first_image_rectangle[2]
height_first_image_rectangle = first_image_rectangle[3]
first_image_rectangle_part = get_rectangle_part_of_image(first_image,
                                                         x_first_image_rectangle,
                                                         y_first_image_rectangle,
                                                         width_first_image_rectangle,
                                                         height_first_image_rectangle)
x_last_image_rectangle = last_image_rectangle[0]
y_last_image_rectangle = last_image_rectangle[1]
width_last_image_rectangle = last_image_rectangle[2]
height_last_image_rectangle = last_image_rectangle[3]
last_image_rectangle_part = get_rectangle_part_of_image(last_image,
                                                        x_last_image_rectangle,
                                                        y_last_image_rectangle,
                                                        width_last_image_rectangle,
                                                        height_last_image_rectangle)
```

که از این تابع استفاده میکند:

```
def get_rectangle_part_of_image(image, x, y, width, height):
    return image[y:y + height, x:x + width]
```

حال میخواهیم تصویر را وارپ کنیم. برای سایز مستطیلی که تصویر وارپ شده به ما می دهد، در ابتدا از سایز خود مستطیل استفاده میکردم. اما در بعضی از مستطیل ها این موضوع باعث ارور میشد. بعد از بررسی فهمیدم که این موضوع وقتی اتفاق می افتد که طول مستطیل بعلاوه x نقطه بالا چپ آن بیشتر از طول عکس اصلی شود(به همین ترتیب برای عرض مستطیل و ارتفاع عکس). پس ابتدا با استفاده از تابع زیر این موضوع رو برطرف کردم:

```python
def get_output_rectangle_size(main_images_width, main_images_height, middle_image_rectangle):
    width = middle_image_rectangle[2]
    if middle_image_rectangle[0] + middle_image_rectangle[2] > main_images_width:
        width = main_images_width - middle_image_rectangle[0]

    height = middle_image_rectangle[3]
    if middle_image_rectangle[1] + middle_image_rectangle[3] > main_images_height:
        height = main_images_height - middle_image_rectangle[1]

    return width, height
```

حال باید تابع Affine که بین تصویر اول و تصویر وسط و همینطور بین تصویر آخر و تصویر وسط وجود دارد را پیدا کنیم. تابع Affine درجه آزادی ۶ دارد. و ما هم ۶ مختصات داریم(۳ راس مثلث). بنابراین میتوان پیدا کرد که چه تابعی بوده که باعث شده که مثلث در شکل اول به مثلث در شکل میانی تبدیل شود و سپس این تابع را روی کل مستطیل اول اعمال کرد تا یک مستطیل در شکل میانی به ما بدهد که شامل مثلثی است که به دنبال آن هستیم:

```python
def find_and_warp_affine_transform(initial_pic, initial_points, final_points, width, height):
    """
    According to points of rectangle in first_image and points of rectangle in middle_image, find the Affine transform.
    Then, transform first_image with that func to get middle image.
    :param initial_pic: source pic
    :param initial_points: source points
    :param final_points:  destination points
    :param width: output width
    :param height: output height
    :return: pic transformed
    """
    affine_transform = cv2.getAffineTransform(np.float32(initial_points), np.float32(final_points))
    final_pic = cv2.warpAffine(initial_pic, affine_transform, (width, height), borderMode=cv2.BORDER_REFLECT_101)
    return final_pic
```

بدین ترتیب ما دو تصویر وارپ شده بدست می آوریم:

```python
# warp rectangle part of first and last images
warped_rectangle_part_of_first_image = find_and_warp_affine_transform(first_image_rectangle_part,
                                                                      first_image_rectangle_new_dimension,
                                                                      middle_image_rectangle_new_dimension,
                                                                      output_width, output_height)
warped_rectangle_part_of_last_image = find_and_warp_affine_transform(last_image_rectangle_part,
                                                                     last_image_rectangle_new_dimension,
                                                                     middle_image_rectangle_new_dimension,
                                                                     output_width, output_height)
```

یکی تصویر میانی حاصل اعمال تابع Affine که از تصویر اول به تصویر میانی بدست آوردیم، و دیگری تصویر میانی ای که از اعمال تابع Affine که از تصویر آخر به تصویر میانی بدست آوردیم. حال با استفاده از نسبت K ، این دو تصویر را ادغام میکنم.

```
# get middle image rectangle part using warped first and last images and k ratio
middle_rectangle_part = (1.0 - k) * warped_rectangle_part_of_first_image + \
                        k * warped_rectangle_part_of_last_image
```

حال برای اینکه میخواهیم تنها مثلث را در تصویر میانی جایگذاری کنیم نه کل مستطیل را باید یک ماسک بسازیم که داخل مثلث ۱ باشد و خارج آن ۰. بدین ترتیب میتوانیم بدون تاثیرگذاشتن روی قسمت های خارج مثلث، قسمت مورد نظر را پر کنیم:

```
def fill_rectangle_in_middle_image(middle_image, middle_image_rectangle, middle_rectangle_part, mask):
    """
    Filling triangle in middle image
    Have a rectangle and a triangle in that rectangle!
    Outside of triangle must don't change, so we multiple middle_image to (1-mask)

    :param middle_image: the result pic
    :param middle_image_rectangle: the rectangle correspond to this middle image
    :param middle_rectangle_part: part of middle image that rectangle is there
    :param mask: a mask that inside of triangle is 1 and outside of it is 0
    :return: the middle image (result pic)
    """
    x = middle_image_rectangle[0]  # x_middle_rect
    y = middle_image_rectangle[1]  # y_middle_rect
    width = middle_image_rectangle[2]  # width_middle_rect
    height = middle_image_rectangle[3]  # height_middle_rect

    middle_image[y:y + height, x:x + width] = \
        middle_rectangle_part * mask + middle_image[y:y + height, x:x + width] * (1 - mask)

    return middle_image
```

برای ساختن ماسک، بهترین روش کشیدن مثلث با رنگ سفید داخل مستطیل است. که بدین منظور کافی است از تابع fillConvexPoly استفاده کنم:

```
# build a mask with output size: inside of triangle = 1, outside of triangle = 0
mask = np.zeros((output_height, output_width, 3), dtype=np.float32)
cv2.fillConvexPoly(mask, np.int32(middle_image_rectangle_new_dimension), (255, 255, 255), 16, 0)
mask = np.vectorize(lambda x: 1 if x == 255 else 0)(mask)
```

بدین ترتیب با وارپ کردن تک تک مثلث ها، یک تصویر میانی ساخته میشود.

گیفی از تشکیل این مثلث ها با نام triangle_completion.gif در پوشه out قرار دارد که روند ساخته شدن مثلث ها را در میانی ترین تصویر (k=0.5) نشان می دهد.