Computer Vision Assignment 2

Student Name: Uğur Ali Kaplan
Student No: 150170042

Part 1

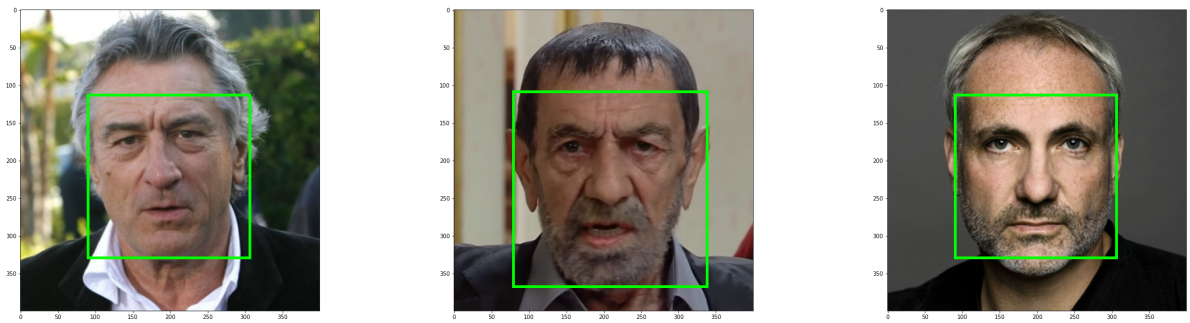
In this part, I have used the skeleton code provided in the homework. To use a trained predictor, I have downloaded the `"shape_predictor_68_face_landmarks.dat"` file.

I have used matplotlib to get the desired outputs.

Code:

```
data_path = os.path.join(os.curdir, "images")
predictor_path = os.path.join(os.curdir,
"shape_predictor_68_face_landmarks.dat")

detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor(predictor_path)

images = ["deniro.jpg", "aydemirakbas.png", "kimbodnia.png"]
fig, ax = plt.subplots(1, 3, figsize = (40,10))

for i in range(3):
    image = cv2.imread(os.path.join(data_path, images[i]))
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rectangles = detector(gray)
    print(f"Number of rectangles for image {i + 1}: {len(rectangles)}")

    ax[i].imshow(image[:,:,[2, 1, 0]])
    tl = rectangles[0].tl_corner()
    rect = patches.Rectangle((tl.x, tl.y),
                             rectangles[0].width(),
                             rectangles[0].height(),
                             linewidth=5, edgecolor='lime',facecolor='none')
    ax[i].add_patch(rect)

plt.show()
```

Output:

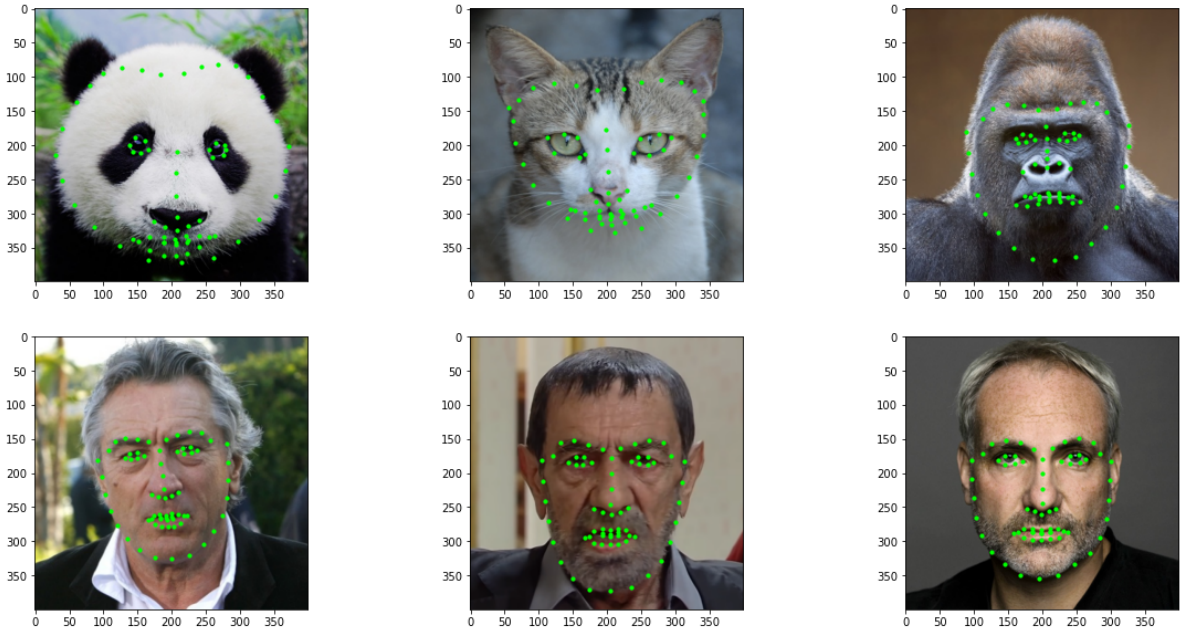


Then, making sure only one rectangle was identified, I have predicted 68 landmark points for people with *dlib*.

Code:

```
human_images = ["deniro.jpg", "aydemirakbas.png", "kimbodnia.png"]
animal_images = ["panda.jpg", "cat.jpg", "gorilla.jpg"]
animal_data = [np.load(os.path.join(data_path, "panda_landmarks.npy")),
               np.load(os.path.join(data_path, "cat_landmarks.npy")),
               np.load(os.path.join(data_path, "gorilla_landmarks.npy"))]

fig, ax = plt.subplots(2, 3, figsize = (20, 10))

human_landmarks_x = []
human_landmarks_y = []

animal_landmarks_x = []
animal_landmarks_y = []

for i in range(3):
    image = cv2.imread(os.path.join(data_path, animal_images[i]))
    points_x = [animal_data[i][k, 0] for k in range(68)]
    points_y = [animal_data[i][k, 1] for k in range(68)]
    animal_landmarks_x.append(points_x)
    animal_landmarks_y.append(points_y)
    ax[0, i].imshow(image[:,:,[2, 1, 0]])
    ax[0, i].scatter([points_x], [points_y], s=10, color="lime")

for i in range(3):
    image = cv2.imread(os.path.join(data_path, human_images[i]))
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rectangles = detector(gray)
    points = predictor(gray, rectangles[0])
    points_x = [points.part(k).x for k in range(68)]
    points_y = [points.part(k).y for k in range(68)]
    human_landmarks_x.append(points_x)
    human_landmarks_y.append(points_y)
    ax[1, i].imshow(image[:,:,[2, 1, 0]])
    ax[1, i].scatter([points_x], [points_y], s=10, color="lime")

plt.show()
```

Output:



Part 2

I will morph 3 images into each other.

1. De Niro $\Longrightarrow$ Kimbodnia
2. Aydemir Akbas $\Longrightarrow$ Gorilla
3. Panda $\Longrightarrow$ Cat

I need to triangulate De Niro, Aydemir Akbas, and Panda and use the ID correspondences to triangulate Kimbodnia, Gorilla, and Cat.

In order to do that, after creating the triangles, I have looked up which vertex represents which landmark. Then, without creating new triangles, I have changed the vertex coordinates to the positions of the landmarks at the target image.

For example, if a triangle created for source image is `[feature_4, feature_45, feature_27]`, I find the coordinates of `feature_4, feature_45` and `feature_27` in the target image and update the points of the triangle.

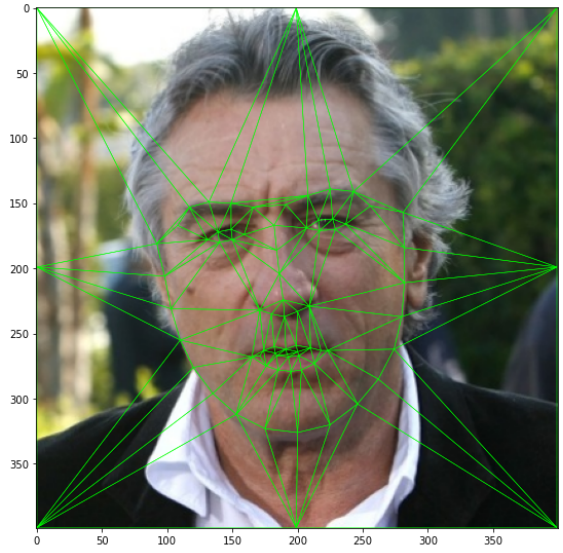
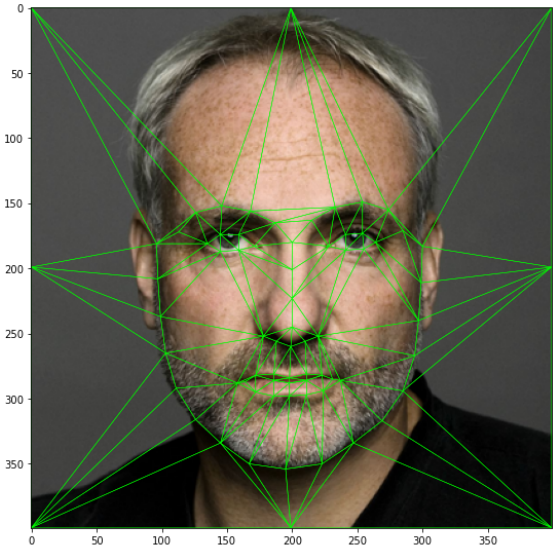
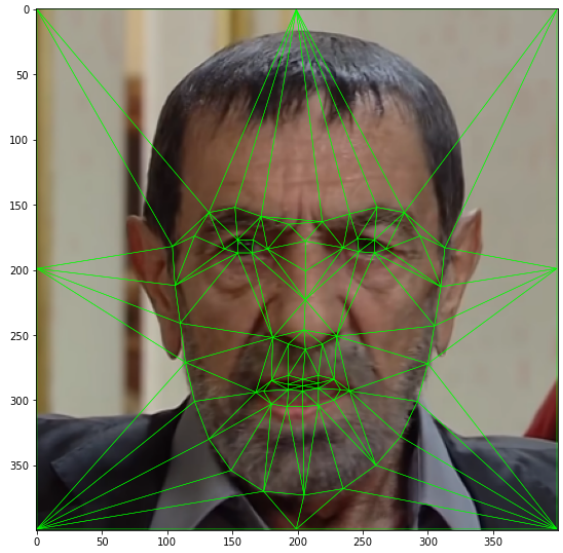
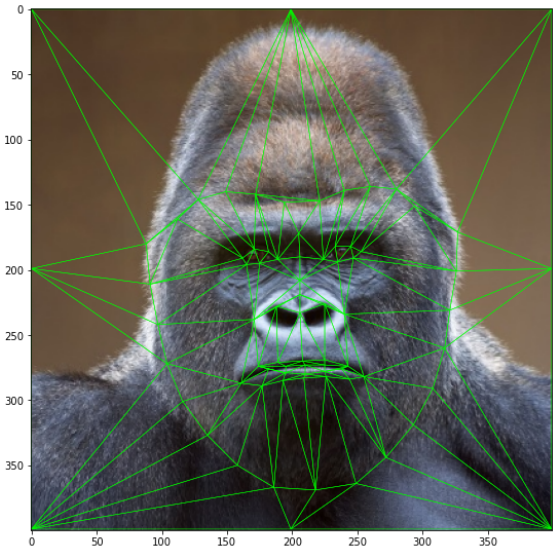
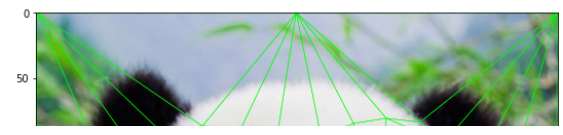
Code:

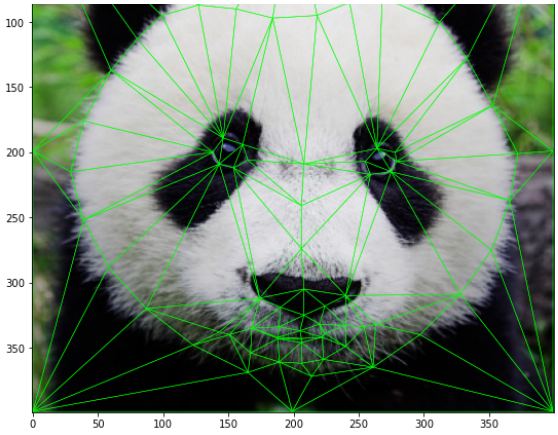
```
pairings = [("deniro.jpg", "kimbodnia.png", 0, 2),
            ("aydemirakbas.png", "gorilla.jpg", 1, 5),
            ("panda.jpg", "cat.jpg", 3, 4)]

morph_pairs = list()

landmarks_x = human_landmarks_x + animal_landmarks_x
landmarks_y = human_landmarks_y + animal_landmarks_y

fig, ax = plt.subplots(3, 2, figsize = (20,50))

for i in range(3):
    src_image = cv2.imread(os.path.join(data_path, pairings[i][0]))
    subdiv = cv2.Subdiv2D((0, 0, src_image.shape[0], src_image.shape[1]))
    landmark_points = list()

    for j in range(68):
```

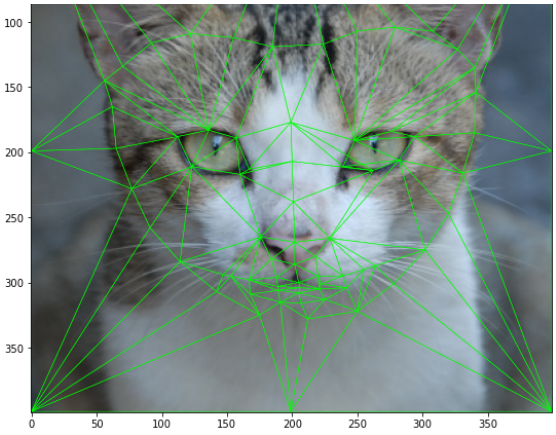
```
        subdiv.insert((landmarks_x[pairings[i][2]][j], landmarks_y[pairings[i]
[2]][j]))
        landmark_points.append([landmarks_x[pairings[i][2]][j],
landmarks_y[pairings[i][2]][j]])

    subdiv.insert((0,0))
    subdiv.insert((0, src_image.shape[1] - 1))
    subdiv.insert((src_image.shape[0] - 1, 0))
    subdiv.insert((src_image.shape[0] - 1, src_image.shape[1] - 1))
    subdiv.insert((0, (src_image.shape[1] - 1)//2))
    subdiv.insert((src_image.shape[0] - 1, (src_image.shape[1] - 1)//2))
    subdiv.insert(((src_image.shape[0] - 1)//2, 0))
    subdiv.insert(((src_image.shape[0] - 1)//2, src_image.shape[1] - 1))

    landmark_points.append([0,0])
    landmark_points.append([0, src_image.shape[1] - 1])
    landmark_points.append([src_image.shape[0] - 1, 0])
    landmark_points.append([src_image.shape[0] - 1, src_image.shape[1] - 1])
    landmark_points.append([0, (src_image.shape[1] - 1)//2])
    landmark_points.append([src_image.shape[0] - 1, (src_image.shape[1] -
1)//2])
    landmark_points.append([(src_image.shape[0] - 1)//2, 0])
    landmark_points.append([(src_image.shape[0] - 1)//2, src_image.shape[1] -
1])

    img1_triangles = subdiv.getTriangleList()


    a = subdiv.getTriangleList().reshape(-1, 3, 2)

    ax[i, 0].imshow(src_image[:,:,[2,1,0]])
    for j in range(a.shape[0]):
        polygon = patches.Polygon(a[j], fill=False, closed=True,
edgecolor="lime", linewidth=0.5)
        ax[i, 0].add_patch(polygon)

    tar_image = cv2.imread(os.path.join(data_path, pairings[i][1]))
    trig_pts = list()

    for j in range(a.shape[0]):
        pt1, pt2, pt3 = a[j, 0, :].astype(np.uint64).tolist(), a[j, 1,
:].astype(np.uint64).tolist(), a[j, 2, :].astype(np.uint64).tolist()
        trig_pts.append([landmark_points.index(pt1),
landmark_points.index(pt2), landmark_points.index(pt3)])

    tar_landmark_points = [[landmarks_x[pairings[i][3]][j],
landmarks_y[pairings[i][3]][j]] for j in range(len(landmarks_x[i]))]
    tar_landmark_points.append([0,0])
    tar_landmark_points.append([0, src_image.shape[1] - 1])
    tar_landmark_points.append([src_image.shape[0] - 1, 0])
    tar_landmark_points.append([src_image.shape[0] - 1, src_image.shape[1] -
1])
    tar_landmark_points.append([0, (src_image.shape[1] - 1)//2])
    tar_landmark_points.append([src_image.shape[0] - 1, (src_image.shape[1] -
1)//2])
    tar_landmark_points.append([(src_image.shape[0] - 1)//2, 0])
    tar_landmark_points.append([(src_image.shape[0] - 1)//2, src_image.shape[1]
- 1])
```

```
    img2_triangles = list()

    for j in range(a.shape[0]):
        co1, co2, co3 = tar_landmark_points[trig_pts[j][0]],
tar_landmark_points[trig_pts[j][1]], tar_landmark_points[trig_pts[j][2]]
        img2_triangles.append([co1, co2, co3])

    img2_triangles = np.array(img2_triangles)

    ax[i, 1].imshow(tar_image[:,:,[2,1,0]])
    for j in range(img2_triangles.shape[0]):
        polygon = patches.Polygon(img2_triangles[j], fill=False, closed=True,
edgecolor="lime", linewidth=0.5)
        ax[i, 1].add_patch(polygon)

    morph_pairs.append([img1_triangles, img2_triangles.reshape(-1, 6),
src_image, tar_image])

plt.show()
```

Output:



Part 3

In this part, I have used the helper functions given in the homework.

1. `make_homogeneous(triangle)`

   This function takes a triangle created by OpenCV and turns it into a matrix.

$$[X_1 \quad Y_1 \quad X_2 \quad Y_2 \quad X_3 \quad Y_3] \Longrightarrow \begin{bmatrix} X_1 & X_2 & X_3 \\ Y_1 & Y_2 & Y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

```
def make_homogeneous(triangle):
    triangle = triangle.reshape(6) # Otherwise it does not change
    homogeneous = np.array([triangle[::2], triangle[1::2], [1, 1, 1]]) # We
create matrices for use in transformation
    # in the form of [[X1, X2, X3], [Y1, Y2, Y3], [1, 1, 1]]
    return homogeneous
```

2. `calc_transform(triangle1, triangle2)`

   This function takes two triangles, and finds the coefficients required to transform first triangle into the second one.

   It creates an M matrix.

$$M = \begin{bmatrix} X_1 & Y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & X_1 & Y_1 & 1 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 \\ X_3 & Y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & X_3 & Y_3 & 1 \end{bmatrix}$$

   Then, finds the transformation coefficient:

$$
\begin{aligned}
Ma &= b \\
M^{-1}Ma &= M^{-1}b \\
a &= M^{-1}b
\end{aligned}
$$

Instead of using the inverse directly, it uses pseudo-inverse and this makes it suitable to use it with non-square matrices. In this example, a is the coefficients and b is the target coordinates.

Last of all, it returns a `Transform` matrix.

$$
\begin{bmatrix}
a_1 & a_2 & a_3 \\
a_4 & a_5 & a_6 \\
0 & 0 & 1
\end{bmatrix}
$$

In this matrix, first row is taking care of the transform of the x coordinates, and second row takes care of the transform of the y coordinates. If it is multiplied with the output of the `make_homogeneous`, it will produce the transformed version of them.

```
def calc_transform(triangle1, triangle2):
    source = make_homogeneous(triangle1).T
    target = triangle2.reshape(6)

    Mtx = np.array([
        np.concatenate((source[0], np.zeros(3))),
        np.concatenate((np.zeros(3), source[0])),
        np.concatenate((source[1], np.zeros(3))),
        np.concatenate((np.zeros(3), source[1])),
        np.concatenate((source[2], np.zeros(3))),
        np.concatenate((np.zeros(3), source[2])),
                    ]) # Create the M matrix for Affine Transformation
Estimation
    # [[X1, Y1, 1, 0, 0, 0],
    #  [0, 0, 0, X1, Y1, 1],
    #  [X2, Y2, 1, 0, 0, 0],
    #  [0, 0, 0, X2, Y2, 1],
    #  [X3, Y3, 1, 0, 0, 0],
    #  [0, 0, 0, X3, Y3, 1]]


    coefs = np.matmul(np.linalg.pinv(Mtx), target) # Ma = b -> M^{-}Ma =
M^{-}b -> We have found the transformatino coefficients

    Transform = np.array([coefs[:3], coefs[3:], [0, 0, 1]]) # Reshape the
coefficients matrix
    # Now first row is only for the transformation of x coordinates
    # Second row is only for the transformation of y coordinates
    # Third row is not used, it repeats the third row of the matrix to its
right

    return Transform
```

3. `vectorised_Bilinear(coordinates, target_img, size)`

   This will be used to find out the pixel intensities in the transform. First, it clips the values between the possible coordinates.

   Transformed coordinates are not guaranteed to be integers. Therefore, we have to interpolate the intensity of the pixels.

   Using the `error` and the `resindual` variables, we measure the distance to the neighbouring pixels and weight their intensity depending on the distance of our transformed coordinates.

   Finally, we sum weighted intensities. By doing this, we have interpolated the intensity for our transformed coordinate.

```
def vectorised_Bilinear(coordinates, target_img, size):
    # c[0] -> x
    # c[1] -> y
    coordinates[0] = np.clip(coordinates[0], 0, size[0] - 1) # Points will
lie between the range of the coordinate system
    coordinates[1] = np.clip(coordinates[1], 0, size[1] - 1)
    lower = np.floor(coordinates).astype(np.uint32) # Points rounded down
    upper = np.ceil(coordinates).astype(np.uint32) # Points rounded up

    error = coordinates - lower # Difference between coordinates and the
rounded down coordinates (different for each point)
    resindual = 1 - error # Difference between coordinates and the rounded
up coordinates

    # We find the pixel intensities in the image, and give a weight to
found intensities depending on the error rate
    top_left = np.multiply(np.multiply(resindual[0],
resindual[1]).reshape(coordinates.shape[1], 1), target_img[lower[0],
lower[1], :])
    top_right = np.multiply(np.multiply(resindual[0],
error[1]).reshape(coordinates.shape[1], 1), target_img[lower[0], upper[1],
:])
    bot_left = np.multiply(np.multiply(error[0],
resindual[1]).reshape(coordinates.shape[1], 1), target_img[upper[0],
lower[1], :])
    bot_right = np.multiply(np.multiply(error[0],
error[1]).reshape(coordinates.shape[1], 1), target_img[upper[0], upper[1],
:])

    return np.uint8(np.round(top_left + top_right + bot_left + bot_right))
# We add the weighted intensities to find the color
```

4. `image_morph(image1, image2, triangles1, triangles2, transforms, t)`

   This function is where the magic happens.
   `t` controls the weights and it also represents the time step. When `t=0`, meaning we are just starting, every information comes from the image 1. When `t=1`, meaning we are finally at our target, everything belongs to the image 2.

   For every triangle we have created, we have to find the transformation.
   First, we calculate the intermediate triangle points at store this information with variable `homo_inter_tri`.

Then, since we want to morph the area inside the triangle, we create a mask and fill the contents of the triangle with black. We store the mask with `polygon_mask`.

We find out the points painted black and create a matrix with the coordinates of the black points. Pixel coordinates inside the triangle is now stored in `mask_points`.

Since we want to use our previous helper functions, we transform the shape of the coordinates of the intermediate triangle. Now, it has the same format returned by the `subdiv.getTriangleList()` function.

We calculate the transformation coefficients from intermediate points to the image points and transform the points inside the triangle.

Now, we have mappings from intermediate points to the original points. With this information, we interpolate the intensities of the pixels.

Finally, we combine the found pixel intensities with regards to `t`.

We do this for every triangle provided in the arguments.

```
def image_morph(image1, image2, triangles1, triangles2, transforms, t):

    inter_image_1 = np.zeros(image1.shape).astype(np.uint8)
    inter_image_2 = np.zeros(image2.shape).astype(np.uint8)

    for i in range(len(transforms)):
        homo_inter_tri = (1-t)*make_homogeneous(triangles1[i]) +
t*make_homogeneous(triangles2[i]) # We are calculating a middle point
between the vertices in source and target triangle depending on the value
of t

        polygon_mask = np.zeros(image1.shape[:2], dtype=np.uint8) # Zeros
matrix in the shape of the image (height, width)
        cv2.fillPoly(polygon_mask, [np.int32(np.round(homo_inter_tri[1::-1,
:].T))], color=255) # # Inside of the triangle in the polygon_mask is
colored black
        seg = np.where(polygon_mask == 255) # Points in which mask is
colored black (seg[0] -> x coordinates, seg[1] -> y coordinates)
        mask_points = np.vstack((seg[0], seg[1], np.ones(len(seg[0]))))
        # We have created a matrix in the form of
        # X X X X X ...
        # Y Y Y Y Y ...
        # 1 1 1 1 1 ...
        # and it has the coordinates of the points we want to morph for the
current triangle
        # (coordinates of the pixels in the region bound by the triangle)

        inter_tri = homo_inter_tri[:2].flatten(order="F") # Intermediate
points (pixel location in the morphed image)
        # In the format of [[X1, Y1, X2, Y2, X3, Y3]]

        inter_to_img1 = calc_transform(inter_tri, triangles1[i]) #
Calculate transformation from intermediate points to the points in the
image 1
        inter_to_img2 = calc_transform(inter_tri, triangles2[i]) #
Calculate transformation from intermediate points to the points in the
image 2
```

```
        mapped_to_img1 = np.matmul(inter_to_img1, mask_points)[:-1] #
Transform the mask points using the intermediate->image map
        mapped_to_img2 = np.matmul(inter_to_img2, mask_points)[:-1] #
Transform the mask points using the intermediate->image map

        # We find the pixel intensities in the intermediate image
        # depending on the current transformation and the error rates
        inter_image_1[seg[0], seg[1], :] =
vectorised_Bilinear(mapped_to_img1, image1, inter_image_1.shape)
        inter_image_2[seg[0], seg[1], :] =
vectorised_Bilinear(mapped_to_img2, image2, inter_image_2.shape)

    result = (1-t)*inter_image_1 + t*inter_image_2 # Combine the
intermediate images with weights depending on the timestep

    return result.astype(np.uint8)
```

5. **Generating the Videos**

   We use the triangles we have created in part 2. For each triangle, we have to find coefficients. Therefore, we calculate the transformation from source to target image for each triangle.

   Each element of `Transforms` is a transformation matrix.

   Then, we produce the frames using the helper functions I explained above. We are changing the weight of source and target images in each step.

   When we add the frames into the morphs array, we change its channel order as well.

```
sequences = []

for it in range(3):
    img1_triangles, img2_triangles, src_image, tar_image = morph_pairs[it]
    img1_triangles = img1_triangles[:, [1, 0, 3, 2, 5, 4]]
    img2_triangles = img2_triangles[:, [1, 0, 3, 2, 5, 4]]

    Transforms = np.zeros((len(img1_triangles), 3, 3))

    for i in range(Transforms.shape[0]):
        source = img1_triangles[i].reshape(6)
        target = img2_triangles[i].reshape(6)
        Transforms[i] = calc_transform(source, target) # We will transform
triangles corresponding to same regions (We have changed the feature
landmarks of the second image without creating a new one after all)
        # Here, each element of the Transforms is a coefficient matrix that
can be multiplied with the points in the source triangle
        # So, we will be multiplying the outputs of the make_homogeneous
with coefficients we have here

    morphs = []
    for t in np.arange(0, 1.0001, 0.02): # # We have to calculate multiple
frames for a smooth transition, each t represents a timestep in the
transition
        print("Processing:\t", t*100, "%")
        morphs.append(image_morph(src_image, tar_image, img1_triangles,
img2_triangles, Transforms, t)[:, :, ::-1])

    sequences.append(morphs)
```

Finally, we can create the videos with:

```
for it in range(3):
    clip = mpy.ImageSequenceClip(sequences[it], fps=25)
    clip.write_videofile("video_" + str(it) + ".mp4", codec="libx264")
```

Outputs are: `video_0`, `video_1`, and `video_2`.