ID:80585887
CS 5363
Dr. Olac Fuentes

I. Problem Description

For Lab 4 our task was to implement solutions to two different problems. The first problem required to develop warping based on the automatic detection of points-of-interest between any “best-pair” of images. In addition, we were required to implement the warping of the two images once the points were obtained to a medium point between the correspondences of every pair of image. The previously described process was applied iteratively until the entire list of images that is provided to the algorithm as input became a fully stitched result. In addition, we also had a second problem for which the task was to use or obtain a set of points which represented correspondences between two images. Then, by using the correspondences we were tasked with linearly combining such points and the images to observe one image morphing into the other.

II. Algorithms Implemented

Problem 1

For this problem I first construct a list of the images I would like to stitch. Then for every pair of images in the list of images I find the pair of indices that indicate which pair of images in the list are the most similar. Similarity is computed by computing distances between the descriptors of every pair of images. Descriptors are composed of the histogram of gradients of an image, color histograms of every color channel, and histogram of gradients of the quadrants of every image.This process of selecting the most similar images in terms of distance is complemented by appending to the list of images the resulting stitching of the previous iteration. Also, the process is repeated until all images in a list are stitched together.

In order to “automatically” and correctly stitch the images I make use of the ORB package which generates descriptors for points of interest. Then the descriptors that are found on one image are then matched to the descriptors of the other image via the use of hamming distance. Hamming distance guarantees that any two matches are indeed close in terms of their corresponding descriptors. Finally, I applied RANSAC as means to select the best points among those that were closely matched through Hamming Distance selection/ORB matcher. The correspondences that remain after applying the RANSAC algorithm on the list of points are then used to create the homography used for the warping of an image to the other.

When a pair of images are warped to visually become one, the points obtained from the RANSAC algorithm are then used to calculate a medium point to warp the images. The difference between this method and the previous manner in which an image is warped is that one image is not forced to match the other but both are forced to match a certain shape to look better.

Problem 2

For this problem I first started by interpolating the points of the images along the edges. By making the points around the edges and the corners of the images to be the same between images the geometric transformation is forced to produce 0 displacement around the edges of the image that way the image does not fold to the inside or towards a specific direction. In addition, I choose by hand the points that matched between one dog and the other and then I used the points I choose as hardcoded points that could be used if the images of the dogs were used.

After the corresponding points by the user are selected such points are then used to perform a PiecewiseAffineTransformation to both images. As a target every image gets a weighted average between the points of every image controlled by a variable alpha, so the target points are at any given iteration :

transform_points = p0*(alpha) + (1-alpha)*p1

After obtaining the warped images that use as target the “transformation_points” we combine the resulting warped images linearly as follows:

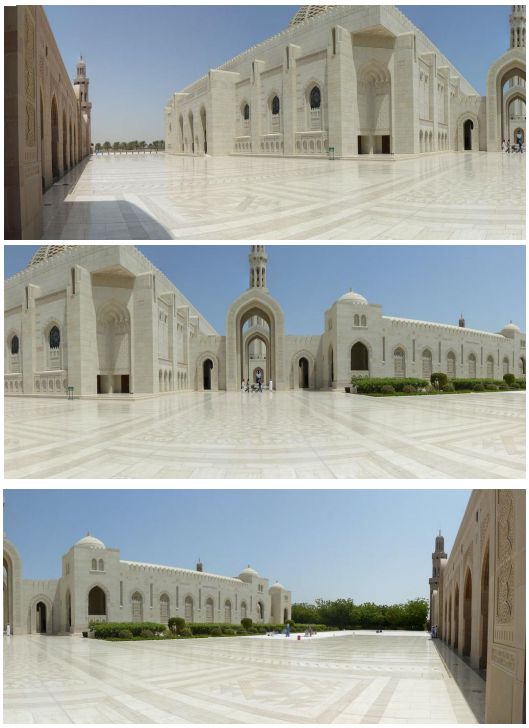
combined = ((alpha)*tf.warp(I0, t0.inverse)+(1-alpha)*tf.warp(I1,t1.inverse))

And finally we output the “combined” image as part of a video.

III. Experimental Results

Problem 1:

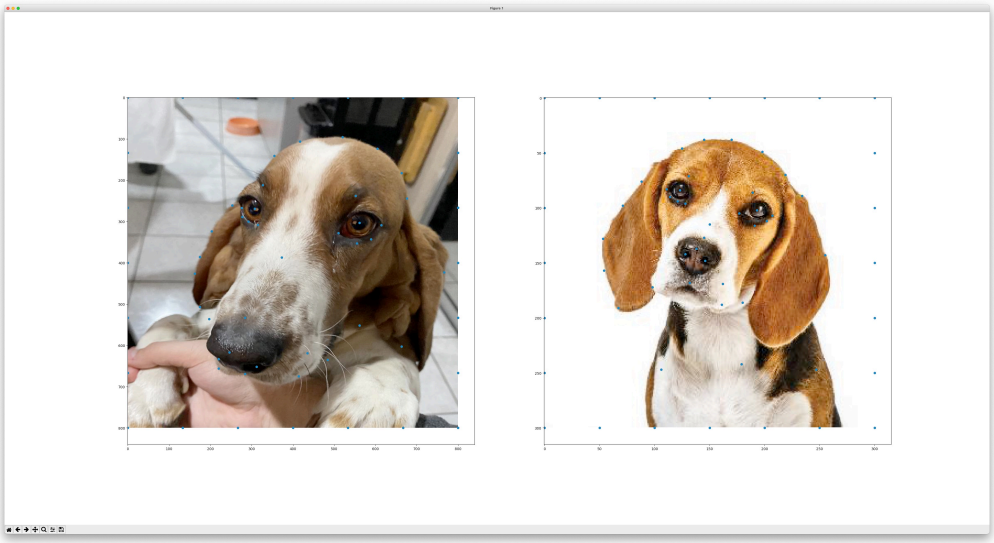
- Original Pieces



- Result

Problem 2

- Please observe the videos “dog.mp4” and “morphing.mp4”
- Points



IV. Discussion of Results

By observing the results of both algorithms it was clear that the algorithms work pretty well. In the case of problem 1 it is clear that the results are good because the salient points between images are easy to detect and hence easy to be matched with ORB. In addition, I think the process of using the histogram of gradients and the color histogram to produce descriptors of the images could be a robust method to detect similarities between images, specially if we know in advance that the images must match in some way or another. The reason for such robustness is due to the fact that one side of a big image is probably very different than the other side, so, it “easy” to detect what images are close to each other. Hence, the images are relatively easy to stitch, this, according to me might not be the case if the say one of the sides was shown at night and the other two during the day.

For problem 2 I observed that points are fundamental to observe a good morphing progression. To me it is obvious that points of interest should be found by an algorithm instead of being selected by the user. In addition there is something to be said about the velocity of change. In my result it is kind of good that the video takes 2 seconds to complete the morphing because there are some morphed images that actually look “weird”.

V. Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from skimage import transform as tf
from scipy import signal
from scipy.interpolate import interp1d as ip1d
import cv2

def correlate2d(image,filt):
    if len(image.shape) == 2:
        r,c = filt.shape
        result = np.zeros((image.shape[0]-r + 1, image.shape[1]-c + 1))
        for i in range(result.shape[0]):
            for j in range(result.shape[1]):
                result[i,j] = np.sum(image[i:i+r,j:j+c]*filt) # Dot product of image region and
filter
        return result
    else:
        r,c = filt.shape
        result = np.zeros((image.shape[0]-r + 1, image.shape[1]-c + 1, 3))
        for i in range(result.shape[0]):
            for j in range(result.shape[1]):
                for channel in range(image.shape[2]):
```

```
                result[i,j,channel] = np.sum(image[i:i+r,j:j+c,channel]*filt) # Dot product of
image region and filter
        result = result/255
        return result.astype(np.float32)

def correlate2d_scipy(image,filt):
    if len(image.shape) == 2:
        return signal.correlate2d(image, filt,mode='valid')
    else:
        r,c = filt.shape
        img = np.zeros((image.shape[0]-r + 1, image.shape[1]-c + 1, 3))
        for channel in range(image.shape[2]):
            img[...,channel] = signal.correlate2d(image[...,channel], filt,mode='valid')
        return (img/255).astype(np.float32)

def color_histograms(img,display=True):
    r,_ = np.histogram(img[:,:,0], bins=255)
    g,_ = np.histogram(img[:,:,1], bins=255)
    b,_ = np.histogram(img[:,:,2], bins=255)
    if display:
        fig, ((i,r_x),(g_x,b_x)) = plt.subplots(2, 2)
        fig.suptitle('Histograms by Color')
        i.imshow(img)
        r_x.hist(img[:,:,0].reshape(img.shape[0]*img.shape[1]),bins=256,color='red')
        g_x.hist(img[:,:,1].reshape(img.shape[0]*img.shape[1]),bins=256,color='green')
        b_x.hist(img[:,:,2].reshape(img.shape[0]*img.shape[1]),bins=256,color='blue')
        fig.subplots_adjust(wspace=0.5,hspace=0.5)
        r_x.set_xlabel("Intensities")
        g_x.set_xlabel("Intensities")
        b_x.set_xlabel("Intensities")
        r_x.set_ylabel("Counts")
        g_x.set_ylabel("Counts")
        b_x.set_ylabel("Counts")
        plt.show()
    #Normalize to be able to compare
    r = r.astype(np.float32)/(img.shape[0]*img.shape[1])
    g = g.astype(np.float32)/(img.shape[0]*img.shape[1])
    b = b.astype(np.float32)/(img.shape[0]*img.shape[1])
    return r,g,b

def histogram_gradients(img):
    filt = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    b_w = np.average(img,axis=2)
    g_v = correlate2d_scipy(b_w,filt)
    g_h = correlate2d_scipy(b_w, filt.T)
    g_m = np.sqrt((np.power(g_v,2)+np.power(g_h,2)))
```

```
    g_d = np.arctan2(g_v, g_h)
    normalized = (((g_d/np.amax(g_d))*180))
    normalized = normalized.reshape(normalized.shape[0]*normalized.shape[1])
    normalized+=180
    g_m = g_m.reshape(g_m.shape[0]*g_m.shape[1])
    hist = np.zeros(360).astype(np.float32)
    for i in range(normalized.shape[0]):
        j = min(359,int(normalized[i]))
        hist[j] += g_m[i]
    return hist

def construct_descriptor(h_r,h_g,h_b,q1_hog,q2_hog,q3_hog,q4_hog,g_d):
    return [h_r,h_g,h_b,q1_hog,q2_hog,q3_hog,q4_hog,g_d]

def compute_distance(orig_descriptor,target_descriptor):
    s = 0
    for arr in range(len(orig_descriptor)):
        s+= np.sum(np.power(orig_descriptor[arr]-target_descriptor[arr],2),axis=0)
    return np.sqrt(s)

def display_control_lines(im0,im1,pts0,pts1,clr_str = 'rgbycmwk'):
    canvas_shape = (max(im0.shape[0],im1.shape[0]),im0.shape[1]+im1.shape[1],3)
    canvas = np.zeros(canvas_shape,dtype=type(im0[0,0,0]))
    canvas[:im0.shape[0],:im0.shape[1]] = im0
    canvas[:im1.shape[0],im0.shape[1]:canvas.shape[1]]= im1
    fig, ax = plt.subplots(figsize=(8, 4))
    ax.imshow(canvas)
    ax.axis('off')
    pts2 = pts1+np.array([im0.shape[1],0])
    for i in range(pts0.shape[0]):
        ax.plot([pts0[i,0],pts2[i,0]],
[pts0[i,1],pts2[i,1]],color=clr_str[i%len(clr_str)],linewidth=1.0)
    fig.suptitle('Point correpondences', fontsize=16)

def cond_num_and_det(H):
    # Very large condition numbers usually indicate a bad homography
    # Negative determinants and those with low absolute values usually indicate a bad
homography
    # Large determinant also usually indicate a bad homography
    w,v = np.linalg.eig(np.array(H.params))
    w = np.sort(np.abs(w))
    cn = w[2]/w[0]
    d = np.linalg.det(H.params)
    return cn, d
    print('condition number {:7.3f}, determinant {:7.3f}'.format(cn,d))
```

```
def homography_error(H,pts1,pts2):
    pts1 =  np.hstack((pts1,np.ones((pts1.shape[0],1))))
    pts2 =  np.hstack((pts2,np.ones((pts2.shape[0],1))))
    proj = np.matmul(H,pts1.T).T
    proj = proj/proj[:,2].reshape(-1,1)
    err = proj[:,:2] - pts2[:,:2]
    return np.mean(np.sqrt(np.sum(err**2,axis=1)))

def select_matches_ransac(pts0, pts1):
    H, mask = cv2.findHomography(pts0.reshape(-1,1,2), pts1.reshape(-1,1,2),
cv2.RANSAC,5.0)
    choice = np.where(mask.reshape(-1) ==1)[0]
    return pts0[choice], pts1[choice]

def get_points(img1,img2):
    orb = cv2.ORB_create()

    # fig, ax = plt.subplots(ncols=2)
    keypoints1, descriptors1 = orb.detectAndCompute(img1,mask=None)
    # ax[0].imshow(cv2.drawKeypoints(img1, keypoints1, None, color=(0,255,0),
flags=0))

    keypoints2, descriptors2 = orb.detectAndCompute(img2,mask=None)
    # ax[1].imshow(cv2.drawKeypoints(img2, keypoints2, None, color=(0,255,0),
flags=0))

    # Create BFMatcher object
    matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = matcher.match(descriptors1,descriptors2)

    # Extract data from orb objects and matcher
    ind1 = np.array([m.queryIdx for m in matches])
    ind2 = np.array([m.trainIdx for m in matches])
    keypoints1 = np.array([p.pt for p in keypoints1])
    keypoints2 = np.array([p.pt for p in keypoints2])
    keypoints1 = keypoints1[ind1]
    keypoints2 = keypoints2[ind2]
    # keypoints1[i] and keypoints2[i] are a match
    # display_control_lines(img1,img2, keypoints1, keypoints2)

    print('Original number of matches',keypoints1.shape[0])
    keypoints1, keypoints2 = select_matches_ransac(keypoints1, keypoints2)
    # display_control_lines(img1,img2, keypoints1, keypoints2)
    print('Number of matches after performing RANSAC',keypoints1.shape[0])
    return keypoints1,keypoints2
```

```
def warp(img,src,dst,shape = None):
    H = tf.ProjectiveTransform()
    H.estimate(src,dst)
    result = None
    if not shape:
        result = tf.warp(img, H.inverse)
    else:
        result = tf.warp(img, H.inverse,output_shape=shape)
    return result,H

def get_descriptors(images):
    descriptors = []
    for image in images:
        img = image
        g_d = histogram_gradients(img).astype(np.float32)
        h_r,h_g,h_b = color_histograms(img,display=False)
        h_w,h_l = int(img.shape[0]/2),int(img.shape[1]/2)
        pixel_cout = img.shape[0]*img.shape[1]
        q1_hog = histogram_gradients(img[:h_w,:h_l])/pixel_cout
        q2_hog = histogram_gradients(img[:h_w,h_l:])/pixel_cout
        q3_hog = histogram_gradients(img[h_w:,:h_l])/pixel_cout
        q4_hog = histogram_gradients(img[h_w:,h_l:])/pixel_cout
        descriptors.append(construct_descriptor(h_r, h_g, h_b, q1_hog, q2_hog, q3_hog,
q4_hog,g_d))
    return descriptors

def find_best_match(descriptors):
    best = float("inf")
    best_i, best_j = 0,0
    for i in range(0,len(descriptors)):
        for j in range(0,len(descriptors)):
            if j==i: continue
            d = compute_distance(descriptors[i], descriptors[j])
            if best > d:
                best = d
                best_i = i
                best_j = j
    return best_i,best_j

def stitch(img1,img2):
    img_p,img2_p = get_points(img1, img2)
    offset = max(img1.shape[0],img1.shape[1])
    offset2 = max(img2.shape[0],img2.shape[1])
    offset = max(offset,offset2)
    space =
np.zeros((offset*2+img2.shape[0],offset*2+img2.shape[1],3)).astype(np.float32)
```

```
    space2 =
np.zeros((offset*2+img2.shape[0],offset*2+img2.shape[1],3)).astype(np.float32)
    space[offset:offset+img1.shape[0],offset:offset+img1.shape[1],:] = img1
    space2[offset:offset+img2.shape[0],offset:offset+img2.shape[1],:] = img2
    img_p += offset
    img2_p += offset
    dst_avg = (img_p+img2_p)/2
    ##################################

    #Apply Homography and calculate limits of resulting image
    result1,H1 = warp(space,img_p,dst_avg)
    # plt.imshow(result1.astype(np.uint8))
    # plt.show()
    result,H2 = warp(space2,img2_p,dst_avg)
    # plt.imshow(result.astype(np.uint8))
    # plt.show()
    plt.imshow(np.maximum(result1,result).astype(np.uint8))
    coords = tf.warp_coords(H1, space.shape[:-1])
    coords2 = tf.warp_coords(H2, space2.shape[:-1])

    corners_img_1 = np.array([
        [int(coords[0,offset,offset]),int(coords[1,offset,offset])],

[int(coords[0,offset+img1.shape[0],offset]),int(coords[1,offset+img1.shape[0],offset])],

[int(coords[0,offset,offset+img1.shape[1]]),int(coords[1,offset,offset+img1.shape[1]])],

[int(coords[0,offset+img1.shape[0],offset+img1.shape[1]]),int(coords[1,offset+img1.shap
e[0],offset+img1.shape[1]])]
        ])
    corners_img_2 = np.array([
        [int(coords2[0,offset,offset]),int(coords2[1,offset,offset])],

[int(coords2[0,offset+img2.shape[0],offset]),int(coords2[1,offset+img2.shape[0],offset])],

[int(coords2[0,offset,offset+img2.shape[1]]),int(coords2[1,offset,offset+img2.shape[1]])],

[int(coords2[0,offset+img2.shape[0],offset+img2.shape[1]]),int(coords2[1,offset+img2.sh
ape[0],offset+img2.shape[1]])]
        ])

    min_r = min(min(corners_img_1[:,0])-1,min(corners_img_2[:,0])-1)
    max_r =max(max(corners_img_1[:,0])+1,max(corners_img_2[:,0])+1)
    min_c = min(min(corners_img_1[:,1])-1,min(corners_img_2[:,1])-1)
    max_c =max(max(corners_img_1[:,1])+1,max(corners_img_2[:,1])+1)
```

```
    result = np.maximum(result1,result).astype(np.uint8)
    stitch_result = result[min_r:max_r,min_c:max_c]
    return stitch_result

def merge_images(image_list):
    plt.close('all')
    while len(image_list) > 1:
        i,j = find_best_match(get_descriptors(image_list))
        img1,img2 = image_list[i],image_list[j]
        image_list.remove(img1)
        image_list.remove(img2)
        stitched = stitch(img1,img2).astype(np.uint8)
        plt.imshow(stitched)
        plt.show()
        image_list.append(stitched)
    return image_list

def morph(I0,I1,p0,p1,steps,display=False,name='morphing'):
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(f'{name}.mp4',fourcc, 50, (I0.shape[0],I0.shape[1]))
    for i in range(steps+1):
        alpha = i/(steps)
        transform_points = p0*(alpha) + (1-alpha)*p1
        t0 = tf.PiecewiseAffineTransform()
        t1 = tf.PiecewiseAffineTransform()
        t0.estimate(p0,transform_points)
        t1.estimate(p1,transform_points)
        combined = ((alpha)*tf.warp(I0, t0.inverse)+(1-alpha)*tf.warp(I1,t1.inverse))
        combined_write = (combined*255).astype(np.uint8)
        out.write(combined_write[:,:,::-1])
        if display:
            f,a = plt.subplots(nrows=1,ncols=3)
            f.suptitle(f"Alpha at {alpha}")
            a[0].imshow(tf.warp(I0, t0.inverse))
            a[1].imshow(tf.warp(I1,t1.inverse))
            a[2].imshow(combined)
            plt.show()

def distance(p0,p1):
    return np.sqrt(np.power(p0[0]-p1[0],2) + np.power(p0[1]-p1[1],2))

def read_points(points):
    while True:
        p = np.asarray(plt.ginput(n=2), dtype=np.int)
        if distance(p[0],p[1]) < 5:
            break
```

```
        points.append(p)
        # plt.plot(p[:,0],p[:,1],marker = '.',color='cyan')
        plt.pause(0.01)
        plt.show()


def interpolate(corner1,corner2,steps=5):
    if corner1[0] == corner2[0]: #rows are the same
        r = corner1[0]
        c_pred = np.arange(1/(steps+1),1,step=1/(steps+1))
        interpolator_c =
ip1d(np.array([0,1]),np.array([min(corner1[1],corner2[1]),max(corner1[1],corner2[1])]))
        prediction_c = interpolator_c(c_pred[:-1])
        r = np.array([r]*steps)
        points = np.stack((r,prediction_c),axis=-1).reshape(-1,2)
        return points

    else: #cols are the same
        c = corner1[1]
        r_pred = np.arange(1/(steps+1),1,step=1/(steps+1))
        interpolator_r =
ip1d(np.array([0,1]),np.array([min(corner1[0],corner2[0]),max(corner1[0],corner2[0])]))
        prediction_r = interpolator_r(r_pred[:-1])
        c = np.array([c]*steps)
        points = np.stack((prediction_r,c),axis=-1).reshape(-1,2)
        return points



def morph_new(I0,I1,automatic=True,graph=True):
    #Compute corner coordinates
    corners_0 = np.array([[0,0],[0,1],[1,1],[1,0]]) * I0.shape[:2] #r,c
    corners_1 = np.array([[0,0],[0,1],[1,1],[1,0]]) * I1.shape[:2]
    ###############################################
    #Interpolate edge points
    upper = interpolate(corners_0[0],corners_0[1])
    below = interpolate(corners_0[2],corners_0[3])
    right = interpolate(corners_0[1],corners_0[2])
    left  = interpolate(corners_0[0],corners_0[3])
    points_0 = np.concatenate((corners_0,upper,below,right,left))
    ###############################################
    #Interpolate edge points
    upper = interpolate(corners_1[0],corners_1[1])
    below = interpolate(corners_1[2],corners_1[3])
    right = interpolate(corners_1[1],corners_1[2])
    left  = interpolate(corners_1[0],corners_1[3])
```

```
    points_1 = np.concatenate((corners_1,upper,below,right,left))
    left = [[251., 610.],[220., 635.],[306., 655.],[313., 308.],[514., 332.],[275., 269.],[615.,
309.],
            [539., 343.],[588., 337.],[281., 289.],[288., 300.],[318., 210.],[539., 244.],[597.,
238.],
            [513.,  95.],[525., 585.],[560., 564.],[174., 507.],[298., 495.],[363., 403.],[400.,
296.],
            [442., 181.],[192., 362.],[253., 262.],[642., 281.],[693., 437.],[564., 598.]]
    right = [[377., 358.],[342., 384.],[392., 399.],[350., 251.],[487., 282.],[298., 224.],[552.,
285.],
             [499., 298.],[536., 299.],[308., 243.],[326., 249.],[352., 183.],[504., 230.],[543.,
236.],
             [468., 105.],[468., 515.],[491., 503.],[180., 513.],[388., 326.],[396., 306.],[414.,
248.],
             [434., 175.],[180., 283.],[237., 205.],[714., 324.],[758., 472.],[703., 650.]]
    f,a = plt.subplots(nrows=1,ncols=2)
    a[0].imshow(I0)
    a[1].imshow(I1)
    if not automatic:
        points = []
        print("Click Points")
        read_points(points)
        points = np.array(points)
        left,right = points[:,0],points[:,1]
    left = np.concatenate((points_0[:,::-1],left))
    right = np.concatenate((points_1[:,::-1],right))
    if graph:
        a[0].scatter(left[:,0],left[:,1])
        a[1].scatter(right[:,0],right[:,1])
        print(points_0.shape)
        print(right)
        plt.show()
    morph(I0,I1,left,right,100,display=False,name='dogs')
    return np.array([left,right])

if __name__ == "__main__":
    # An ORB feature consists of a keypoint (the coordinates of the region's center) and
a descriptor (a binary vector of length 256 that characterizes the region)
    plt.close('all')

    # img1 = mpimg.imread('Part0.jpg')
    # img2 = mpimg.imread('Part1.jpg')
    # img3 = mpimg.imread('Part2.jpg')
    # merge_images([img1,img2,img3])
```

```
    # img1 = mpimg.imread('/Users/oscargalindo/Desktop/Classes/CS 5363/Lab 4/
gorilla.jpg')[:,:,:3]
    # img2 = mpimg.imread('/Users/oscargalindo/Desktop/Classes/CS 5363/Lab 4/
mona_lisa.jpg')[:,:,:3]
    # pts_img1 = np.load('/Users/oscargalindo/Desktop/Classes/CS 5363/Lab 4/
gorilla_pts_72.npy')
    # pts_img2 = np.load('/Users/oscargalindo/Desktop/Classes/CS 5363/Lab 4/
monalisa_pts_72.npy')
    # morph(img1, img2, pts_img1, pts_img2, 100)

    img1_new = mpimg.imread('/Users/oscargalindo/Desktop/Classes/CS 5363/Lab 4/
rubi.jpg')[:,:,:3]
    img2_new = mpimg.imread('/Users/oscargalindo/Desktop/Classes/CS 5363/Lab 4/
beagle.jpg')[:,:,:3]
    used_points = morph_new(img1_new,img2_new)
```

VI. Conclusion

I think for this lab two interesting things that I learned were first to do warping on in both directions to a median point and, second, it is possible to do morphing from one object to the other as long as the points are correctly chosen and there is some reasonable differences in the geometry of every image. I believe the lab was as “short” as it was very interesting and it has taught me a lot about how to combine transformations.

Statement of Academic Honesty:

“I certify that this project is entirely my own work. I
wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”