DEPARTMENT OF ECE, UCLA
ECE 188: THE FOUNDATIONS OF COMPUTER VISION

INSTRUCTOR: Prof. Achuta Kadambi
TA: Howard Zhang, Shijie Zhou

NAME: Yaman Yucel
UID: 605704529

HOMEWORK 3

PROBLEM	TYPE	TOPIC	MAX. POINTS
1	Analytical	Difference of Gaussians	5
2	Analytical	Keypoint Localization for SIFT	10
3	Coding	Image Stitching	15
4	Coding	Olympic Champion using Homography	5
5	Coding	Eight-Point Algorithm	10

Motivation

In the previous homework and in lecture, we have seen how to extract useful features such as corners using the Harris corner detector and keypoints and feature descriptors using SIFT. These features can then be used to compute correspondences between multiple images, which are useful for a variety of tasks such as image stitching and 3D reconstruction. In this homework, we will focus on SIFT and some applications of correspondences. First, we will examine some analytical aspects of SIFT. We will then transition to various applications of correspondences in 2D: two applications of image stitching, which combines correspondences (extracted via SIFT + RANSAC or manually defined) and homographies. Finally, we will use correspondences and the eight-point algorithm to reconstruct 3D points given correspondences.

The problem set consists of:

- analytical questions to solidify the concepts covered in the class; and
- coding questions to implement some of the algorithms described in class using Python.

Homework Layout

The homework consists of 5 problems in total, with subparts for each problem. There are 2 types of problems in this homework - analytical and coding. We encourage you to answer all the problems using the Overleaf document; however, handwritten solutions will also be accepted.

Submission

You will need to make two submissions: (1) Gradescope: You will submit a PDF with all the answers on Gradescope. (2) BruinLearn: You will submit your Jupyter notebook (.ipynb file) with filename {your UID}.ipynb with all the cells executed on BruinLearn.

1 Difference of Gaussians (5.0 points)

In class, you were taught that the SIFT (scale-invariant feature transform) detector and descriptor uses Difference of Gaussians (DoG) as a computationally efficient approximation to Laplacian of Gaussians (LoG). In this question, you will derive that the Difference of Gaussians approximates the Laplacian of Gaussians. Let $G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-(\frac{x^2+y^2}{2\sigma^2})}$ be the 2D Gaussian.

1.1 Compute $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$ (1.0 points)

Write the expression for $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$.

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{1}{2\pi\sigma^2}\frac{1}{\sigma^3}(x^2+y^2-2\sigma^2)e^{-(\frac{x^2+y^2}{2\sigma^2})}$$

1.2 Laplacian of a 2D Gaussian (1.0 points)

Write the expression for the Laplacian of a 2D Gaussian, $L(x,y)$. *Hint*: this expression was computed in Homework 2.

$$L(x,y) = \frac{1}{2\pi\sigma^2}\frac{x^2+y^2-2\sigma^2}{\sigma^4}e^{-(\frac{x^2+y^2}{2\sigma^2})}$$

$$L(x,y) = -\frac{1}{\pi\sigma^4}(1-\frac{x^2+y^2}{2\sigma^2})e^{-(\frac{x^2+y^2}{2\sigma^2})}$$

1.3 Relationship of $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$ to Laplacian of Gaussian (1.0 points)

Using the expressions you obtained in the previous two parts, express $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$ in terms of the Laplacian of Gaussian $L(x,y)$.

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \sigma L(x,y)$$

1.4 Approximating $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$ (1.0 points)

Write an expression approximating $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$ in terms of $G(x,y,k\sigma)$ and $G(x,y,\sigma)$ for $k \approx 1$.

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma-\sigma}$$

1.5 Approximating Laplacian of Gaussian Using Difference of Gaussians (1.0 points)

Write an expression approximating the Laplacian of Gaussian, $L(x,y)$, in terms of the Difference of Gaussians, $D(x,y,\sigma) = G(x,y,k\sigma) - G(x,y,\sigma)$, for $k \approx 1$.

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{D(x,y,\sigma)}{k\sigma^2-\sigma^2}$$

2 Keypoint Localization for SIFT (10.0 points)

In class, you were taught that SIFT first finds the extrema of the Difference of Gaussians (DoG) and then localizes the keypoints using a Taylor series approximation of the DoG. In this question, you will derive the keypoint localization formula and explain why it is used in SIFT. Let $f(\mathbf{x})$ be the Difference of Gaussians, where $\mathbf{x} = (x, y, \sigma)$ represents the location and scale.

2.1 Taylor Series Approximation for DoG (1.0 points)

Write the second order Taylor series approximation of $f(\mathbf{x}+\Delta\mathbf{x})$ centered around $f(\mathbf{x})$. You do not need to compute the derivatives.

$$f(x+\Delta\mathbf{x}) = f(\mathbf{x}) + \frac{\partial f^T}{\partial \mathbf{x}}\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \frac{\partial^2 f}{\partial \mathbf{x}^2}\Delta\mathbf{x} + \text{higher order terms}$$

2.2 Derivative of Taylor Series Approximation (1.0 points)

Using the Taylor series approximation of $f(\mathbf{x}+\Delta\mathbf{x})$, write the expression for $\dfrac{\partial f(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}}$.

$$\frac{\partial f(x+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = 0 + \frac{\partial f}{\partial \mathbf{x}} + \frac{1}{2}\left(\frac{\partial^2 f}{\partial \mathbf{x}^2} + \frac{\partial^2 f^T}{\partial \mathbf{x}^2}\right)\Delta\mathbf{x} = \frac{\partial f}{\partial \mathbf{x}} + \frac{1}{2}\left(2\frac{\partial^2 f}{\partial \mathbf{x}^2}\right)\Delta\mathbf{x} = \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial^2 f}{\partial \mathbf{x}^2}\Delta\mathbf{x}$$

2.3 Extrema of Taylor Series Approximation (1.0 points)

Using the results from the previous part, write the expression for the extrema $\Delta\mathbf{x}$ of the Taylor series approximation.

$$\frac{\partial f(x+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial^2 f}{\partial \mathbf{x}^2}\Delta\mathbf{x} = 0 \longrightarrow \Delta\mathbf{x} = -\frac{\partial^2 f^{-1}}{\partial \mathbf{x}^2}\frac{\partial f}{\partial \mathbf{x}}$$

2.4 Keypoint Localization (1.0 points)

Given a keypoint $\mathbf{x} = (x, y, \sigma)$ obtained via the scale-space extrema step of SIFT (lecture 7 slide 40), what is the new keypoint obtained via the Taylor series approximation? Write the expression for the new keypoint.

New keypoint is $\mathbf{x}+\Delta\mathbf{x}$, therefore $\mathbf{x}+\Delta\mathbf{x} = x - \dfrac{\partial^2 f^{-1}}{\partial \mathbf{x}^2}\dfrac{\partial f}{\partial \mathbf{x}}$

2.5 Purpose of Keypoint Localization (3.0 points)

What is the purpose of the keypoint localization step in SIFT? Please explain.

The purpose of the keypoint localization step in SIFT is to accurately determine the location and scale of keypoints or interest points in an image. Keypoints are distinctive image features that can be robustly matched across different images.

Scale-space extrema detection is prone to find misleading unstable poorly localized keypoints. Keypoint localization step improves the matching and stability of scale-space extrema detection by discarding poorly localized keypoints.

By localizing keypoints accurately, SIFT ensures that these keypoints are stable and repeatable across different images, even under variations in scale, rotation, and affine transformations. This allows for reliable matching and robust keypoint extraction.

2.6 Inaccuracy of Original Keypoint (3.0 points)

Assume that the new keypoint from part 2.4 is closer to a different pixel than it is to the original keypoint $\mathbf{x}$. Then, the original keypoint was not completely accurate. Propose a method to obtain a more accurate estimate of the keypoint. *Note:* A similar method can be applied if the scale of the keypoint is inaccurate (i.e. the keypoint's scale is closer to the scale of a different Gaussian kernel used in computing the Difference of Gaussians).

If the new keypoint is not the close to that original keypoint, then the new keypoint is not accurately localized and extremum lies closer to a different keypoint. Iterative refinement can fix the mislocalized keypoint. First, using the SIFT localization method, we should obtain the estimate of the keypoint's location and scale. Second, we can perform interpolation to get subpixel localization of the keypoint with the interpolation,then we refine its location. Third, we need to compare distances between the new obtained keypoint and the original keypoint as well as neighbors of original keypoint. If the distance to neighbors are smaller than the distance to original keypoint, it shows that the original keypoint is mislocalized or inaccurate. Fourth, there are two ways we need to discuss in this step. You can move the new keypoint towards the pixel it is closer to or you can use interpolation to find the new location of the keypoint which falls between the original keypoint and the closer pixel. Fifth, similar method to adjust scale factor can be used rather than using locations to refine localization of the keypoint. The process from second to fifth can be repeated as long as keypoint parameters converge to a stable estimate.

3 Image Stitching (15.0 points)

In this question, you will be implementing the image stitching pipeline used to create image panoramas. This pipeline combines SIFT, RANSAC, and homographies to find the homography between a pair of images. After finding the homography between the pair of images, you can use it to stitch the two images together.

Note: For extracting SIFT keypoints and features, you should install OpenCV version 4.5.1.48, which can be installed either by running the top cell of the Jupyter notebook or by using the following command:

```
pip install opencv-contrib-python==4.5.1.48
```

3.1 Obtaining SIFT Keypoints and Descriptors (1.0 points)

(See the Jupyter notebook). In this sub-part, you will write a function to obtain SIFT keypoints and descriptors for an image. Make sure that your code is within the bounding box.

```
def run_sift(image, num_features):
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    sift = cv2.SIFT_create(nfeatures = num_features)
    kp, des = sift.detectAndCompute(gray,None)
    return kp,des
```

3.2 Finding Initial Correspondences (1.0 points)

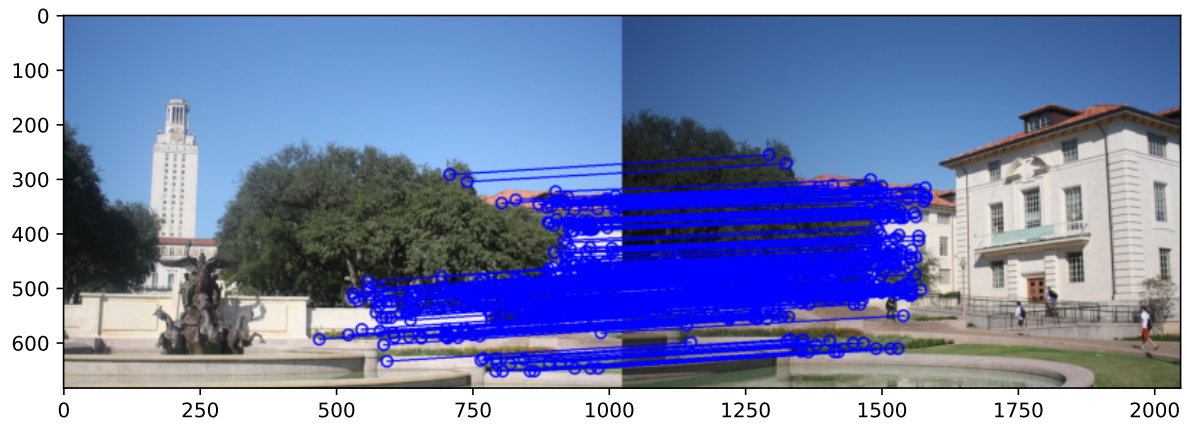
(See the Jupyter notebook). In this sub-part, you will write a function to obtain an initial set of correspondences by matching SIFT descriptors. Make sure that your code is within the bounding box.

```
def find_sift_correspondences(kp1, des1, kp2, des2, ratio):
    correspondences = []
    for idx, kp in enumerate(kp1):
        des = des1[idx]
        all_distances = []
        for i in range(len(kp2)):
            all_distances.append(np.linalg.norm(des-des2[i]))
        idx_closest = np.argsort(all_distances)
        idx_kp_1 = idx_closest[0]
        idx_kp_2 = idx_closest[1]
        if(np.linalg.norm(des-des2[idx_kp_1]) < ratio *  \
        np.linalg.norm(des-des2[idx_kp_2])):
```

```
            correspondences.append((kp.pt,kp2[idx_kp_1].pt))
    return correspondences
```

3.3 Visualizing Initial Correspondences (1.0 points)

(See the Jupyter notebook). In this sub-part, you will visualize the initial correspondences obtained by matching SIFT descriptors. Copy the saved image from the Jupyter notebook here.



3.4 Computing Homography Using DLT (1.0 points)

(See the Jupyter notebook). In this sub-part, you will start implementing the RANSAC loop in parts. Write a function to compute a homography between two images given a set of correspondences using direct linear transform (DLT). Make sure that your code is within the bounding box.

```
def compute_homography(correspondences):
    A = np.array([])
    for i in range(len(correspondences)): #should be more than 4
        pair = correspondences[i]
        x,y = pair[0][0],pair[0][1]
        x_t,y_t = pair[1][0],pair[1][1]
        row1 = np.array([-x,-y,-1,
                         0,0,0,
                         x*x_t,y*x_t,x_t])
        row2 = np.array([0,0,0,
                         -x,-y,-1,
                         x*y_t,y*y_t,y_t])
```

```
        A_i = np.vstack((row1,row2))
        if(A.shape[0] == 0):
            A = A_i
        else:
            A = np.vstack((A,A_i))

    u, s, vh = np.linalg.svd(A, full_matrices=True)
    homography = vh[-1].reshape((3,3))
    return homography/homography[2,2]
```

3.5 Applying a Homography (1.0 points)

(See the Jupyter notebook). In this sub-part, you will write a function that applies a homography to warp a set of 2D points. Make sure that your code is within the bounding box.

```
def apply_homography(points, homography):
    output_points = []
    for point in points:
        homogeneous_point = np.array([point[0],point[1],1]).reshape(-1,1)
        homogeneous_mapped_point = np.matmul(homography,homogeneous_point)
        output_point = (homogeneous_mapped_point[0]/homogeneous_mapped_point[2],\
                        homogeneous_mapped_point[1]/homogeneous_mapped_point[2])
        output_points.append(output_point)
```

3.6 Computing Inliers (1.0 points)

(See the Jupyter notebook). In this sub-part, you will write a function that computes the inlier correspondences given a homography, a list of possible correspondences, and a distance threshold. Make sure that your code is within the bounding box.

```
def compute_inliers(homography, correspondences, threshold):
    inliers = []
    outliers = []
    for correspondence in correspondences:
        p_i  = correspondence[0]
        p_i_prime = np.array(correspondence[1]).reshape(-1,1)
        homogeneous_p_i = np.array([p_i[0],p_i[1],1]).reshape(-1,1)
        homogeneous_mapped_p_i = np.matmul(homography,homogeneous_p_i)
        p_i_mapped = np.array((homogeneous_mapped_p_i[0]/homogeneous_mapped_p_i[2],\
                        homogeneous_mapped_p_i[1]/homogeneous_mapped_p_i[2]))
```

```
        if(np.linalg.norm(p_i_prime-p_i_mapped) <= threshold):
            inliers.append(correspondence)
        else:
            outliers.append(correspondence)
    return inliers, outliers
```

3.7 RANSAC Loop (2.0 points)

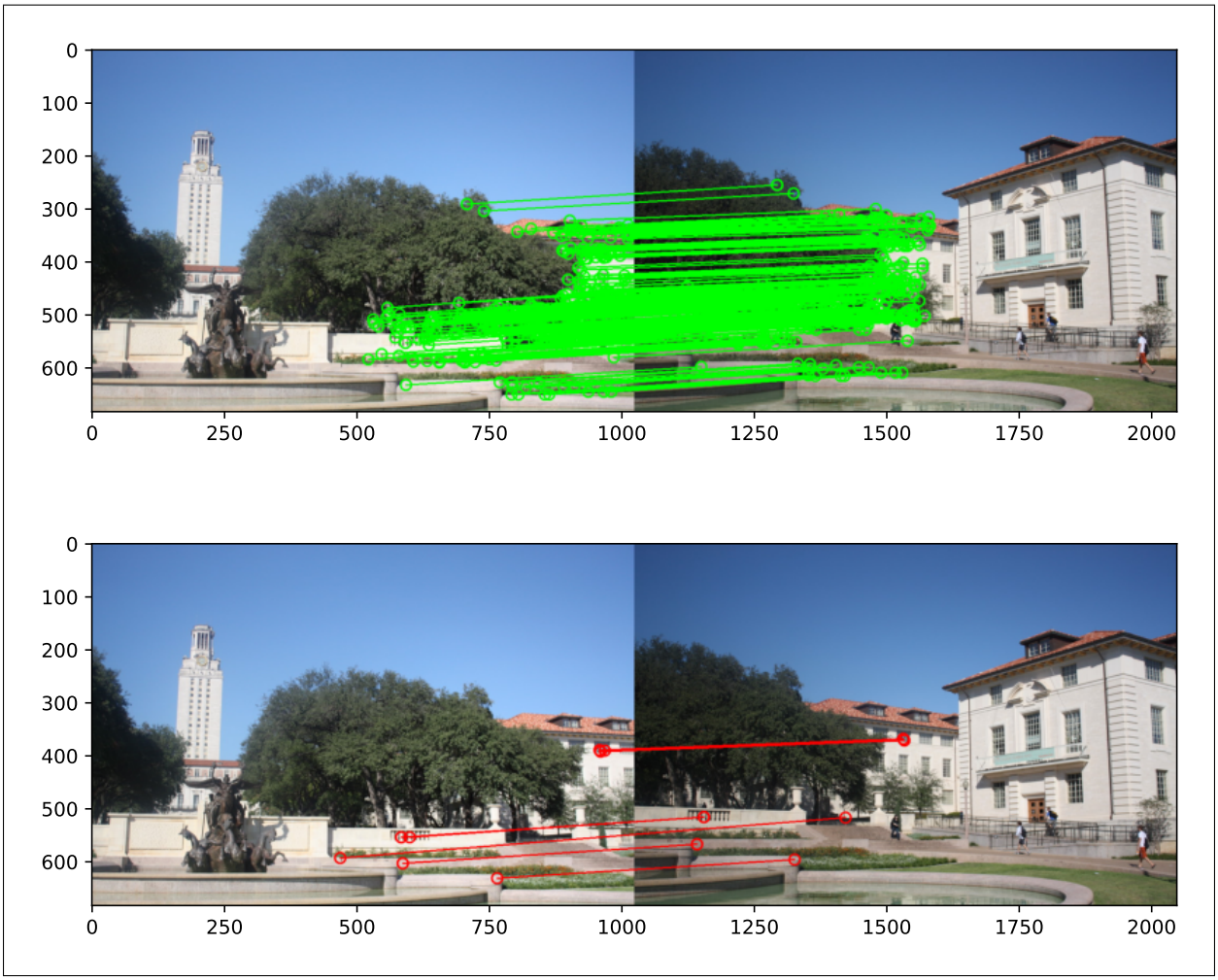
(See the Jupyter notebook). In this sub-part, you will write a function that implements the RANSAC loop to compute a homography matrix and its inlier and outlier correspondences. This part uses some of the earlier parts such as computing a homography and inliers. Make sure that your code is within the bounding box.

```
def ransac(correspondences, num_iterations, num_sampled_points, threshold):
    count_iter = 0
    num_max_inliers = 0
    while(count_iter < num_iterations):
        sample_correspondences = random.sample(correspondences, num_sampled_points)
        homography = compute_homography(sample_correspondences)
        inliers, outliers = compute_inliers(homography, correspondences, threshold)
        if(len(inliers)> num_max_inliers):
            best_inliers = inliers
            best_outliers = outliers
            num_max_inliers = len(inliers)
        count_iter = count_iter + 1

    homography = compute_homography(best_inliers)
    return homography, best_inliers,best_outliers
```

3.8 Visualizing RANSAC Inliers and Outliers (1.0 points)

(See the Jupyter notebook). In this sub-part, you will visualize the inlier and outlier correspondences obtained from running the RANSAC loop on the initial correspondences obtained from matching SIFT features. Copy the saved images from the Jupyter notebook here.



3.9 Bilinear Interpolation (1.0 points)

(See the Jupyter notebook). In this sub-part, you will start implementing the actual image stitching in parts. As the image stitching relies on inverse warping and hence, interpolation, write a function that implements bilinear interpolation. Make sure that your code is within the bounding box.

```
def interpolate(image, loc):
    x0 = int(loc[0])
    y0 = int(loc[1])
    loc_x = loc[0]
    loc_y = loc[1]
    if x0 < image.shape[1] - 1 and y0 < image.shape[0] - 1\
                                    and y0 >= 0 and x0 >= 0:
        x1 = x0 + 1
```

```
        y1 = y0 + 1

        dx = loc_x - x0
        dy = loc_y - y0
        pixel_value = (image[y0, x0] * (1 - dx) * (1 - dy) +
                       image[y0, x1] * dx * (1 - dy) +
                       image[y1, x0] * (1 - dx) * dy +
                       image[y1, x1] * dx * dy)
    else:
        return 0
    return pixel_value
```

3.10 Image Stitching Given Homography (2.0 points)

(See the Jupyter notebook). In this sub-part, you will write a function to implement the actual image stitching using inverse warping given two images and the homography between them. Make sure that your code is within the bounding box.

IN THIS PART, I HAVE ATTACHED IMAGE 1 TO IMAGE 2 using stitch_image_given_H
I HAVE ATTACHED IMAGE 2 TO IMAGE 1 using stitch_image_given_H2

```
def stitch_image_given_H(image1, image2,homography):
    #this stitches 1 to 2
    image_height, image_width = image1.shape[:2]
    output_height, output_width = image2.shape[:2]

    warped_image = np.zeros((output_height, 2*output_width, image1.shape[2]))
    zeros_array = np.zeros((output_height, output_width, image1.shape[2]))

    homography_inv = np.linalg.inv(homography)
    image2_new = np.hstack((zeros_array,image2))
    for y in range(output_height):
        for x in range(-output_width,output_width):
            loc_homogeneous = np.dot(homography_inv, np.array([x, y, 1]))

            loc_x = loc_homogeneous[0] / loc_homogeneous[2]
            loc_y = loc_homogeneous[1] / loc_homogeneous[2]
            loc = (loc_x,loc_y)

            if loc_x >= 0 and loc_x < image_width and loc_y >= 0 \
                                        and loc_y < image_height:
```

```
                    pixel_value = interpolate(image1, loc)
                    warped_image[y, x + output_width] = pixel_value
    stitched_image = np.hstack((zeros_array,zeros_array))

    for i in range(stitched_image.shape[0]):
        for j in range(stitched_image.shape[1]):
             for k in range(stitched_image.shape[2]):
                    if (image2_new[i,j,k] != 0 and warped_image[i,j,k] != 0):
                        stitched_image[i,j,k] = (image2_new[i,j,k] + \
                                                 warped_image[i,j,k])/2
                    elif(image2_new[i,j,k] == 0):
                        stitched_image[i,j,k] = warped_image[i,j,k]
                    elif(warped_image[i,j,k] == 0):
                        stitched_image[i,j,k] = image2_new[i,j,k]

    return stitched_image

def stitch_image_given_H_2(image1, image2,homography):
    #this stitches 2 to 1
    image_height, image_width = image1.shape[:2]
    output_height, output_width = image2.shape[:2]

    warped_image = np.zeros((output_height, 2*output_width, image1.shape[2]))
    zeros_array = np.zeros((output_height, output_width, image1.shape[2]))

    image1_new = np.hstack((image1,zeros_array))
    for y in range(output_height):
        for x in range(0,2*output_width):
            loc_homogeneous = np.dot(homography, np.array([x, y, 1]))

            loc_x = loc_homogeneous[0] / loc_homogeneous[2]
            loc_y = loc_homogeneous[1] / loc_homogeneous[2]
            loc = (loc_x,loc_y)

            if loc_x >= 0 and loc_x < image_width and \
                    loc_y >= 0 and loc_y < image_height:

                    pixel_value = interpolate(image2, loc)
                    warped_image[y, x] = pixel_value
    stitched_image = np.hstack((zeros_array,zeros_array))

    for i in range(stitched_image.shape[0]):
        for j in range(stitched_image.shape[1]):
```

```
            for k in range(stitched_image.shape[2]):
                if (image1_new[i,j,k] != 0 and warped_image[i,j,k] != 0):
                    stitched_image[i,j,k] = (image1_new[i,j,k] + \
                                             warped_image[i,j,k])/2
                elif(image1_new[i,j,k] == 0):
                    stitched_image[i,j,k] = warped_image[i,j,k]
                elif(warped_image[i,j,k] == 0):
                    stitched_image[i,j,k] = image1_new[i,j,k]

    return stitched_image
```

3.11 Image Stitching: Putting It All Together (1.0 points)

(See the Jupyter notebook). In this sub-part, you will put everything together and write a function that implements the whole image stitching pipeline. Make sure that your code is within the bounding box.

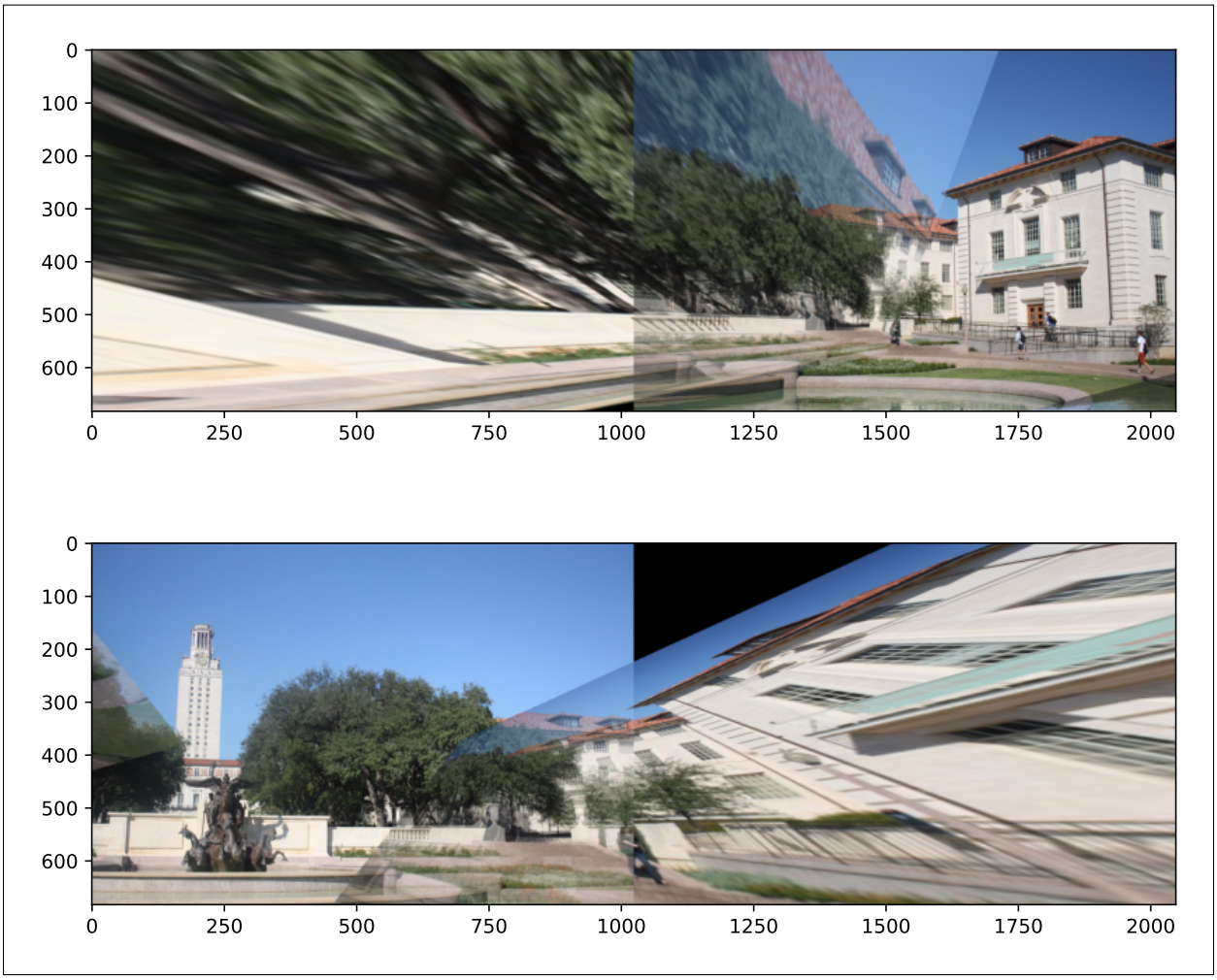
```
def stitch_image(image1, image2, num_features, sift_ratio, ransac_iter, ransac_sampled_
    kp1, des1 = run_sift(image1, num_features)
    kp2, des2 = run_sift(image2, num_features)
    correspondences = find_sift_correspondences(kp1, des1, kp2, des2, sift_ratio)
    if use_ransac:
        homography, inliers, outliers = ransac(correspondences, ransac_iter, \
        ransac_sampled_points, inlier_threshold)
    else:
        homography = compute_homography(correspondences)
    if stitch_mode == 1:
        stichted_image = stitch_image_given_H(image1, image2, homography)
    else:
        stichted_image = stitch_image_given_H_2(image1, image2, homography)
    return stichted_image
```

3.12 Visualizing the Stitched Image (1.0 points)

(See the Jupyter notebook). In this sub-part, you will visualize the stitched image. Copy the saved image from the Jupyter notebook here.

3.13 Visualizing the Stitched Image Without RANSAC (1.0 points)

(See the Jupyter notebook). In this sub-part, you will visualize the stitched image if you do not use RANSAC. The result here should look much worse than the previous stitched image that uses RANSAC. Copy the saved image from the Jupyter notebook here.

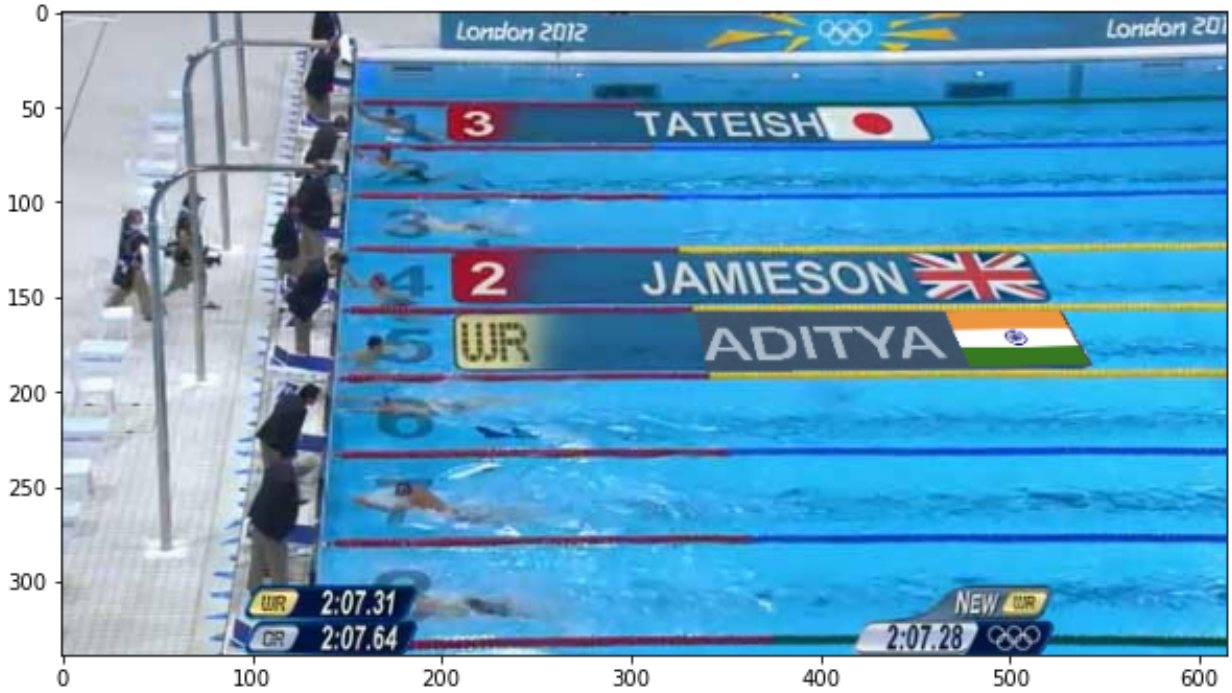

0
100
200
300
400
500
600
0
250
500
750
1000
1250
1500
1750
2000
0
100
200
300
400
500
600
0
250
500
750
1000
1250
1500
1750
2000

4 Olympic Champion using Homography (5.0 points)

In this question you will make yourself an Olympic swimming champion using homography.

You are given the following image from Olympics 2012, where Gyurta is the new world champion.

You are supposed to use homography and make yourself the new world champion.



To make yourself the world champion, you will need two images: (1) the Olympic pool, which is given to you and (2) an image with your name and flag besides it. We will provide you with 4 points on the pool image, which will be the corresponding points for the 4 corners of the image (top left, top right, bottom left, bottom right). Using these corresponding points, you will construct the homography matrix and stitch your name on the Olympic record. To get an image with your name and flag, you can use any method of your choice. We used Keynote + Screenshot (for Mac).

4.1 Correspondence (1.0 points)

(See the Jupyter notebook). Copy the correspondence list from the Jupyter notebook here.

```
A_1 = [3,3]
B_1 = [5,154]
C_1 = [478,4]
D_1 = [480,152]
correspondence = [
                  ([334,158], A_1),
                  ([340,190], B_1),
                  ([528,157], C_1),
                  ([545,187], D_1),
]
```

4.2 Stitching (1.0 points)

(See the Jupyter notebook). In the previous question, you stitched two images side-by-side. In this sub-part, you will stitch one image inside the other. Make sure that your code is within the bounding box. *Hint*: You will need to use code from the previous question and delete/modify a few lines from it.

```
def stitch_image_given_H_new(image1, image2,homography):

    image_height, image_width = image1.shape[:2]
    output_height, output_width = image2.shape[:2]

    warped_image = np.zeros((output_height, output_width, image1.shape[2]))
    zeros_array = np.zeros((output_height, output_width, image1.shape[2]))

    for y in range(output_height):
        for x in range(output_width):
            loc_homogeneous = np.dot(homography, np.array([x, y, 1]))

            loc_x = loc_homogeneous[0] / loc_homogeneous[2]
```

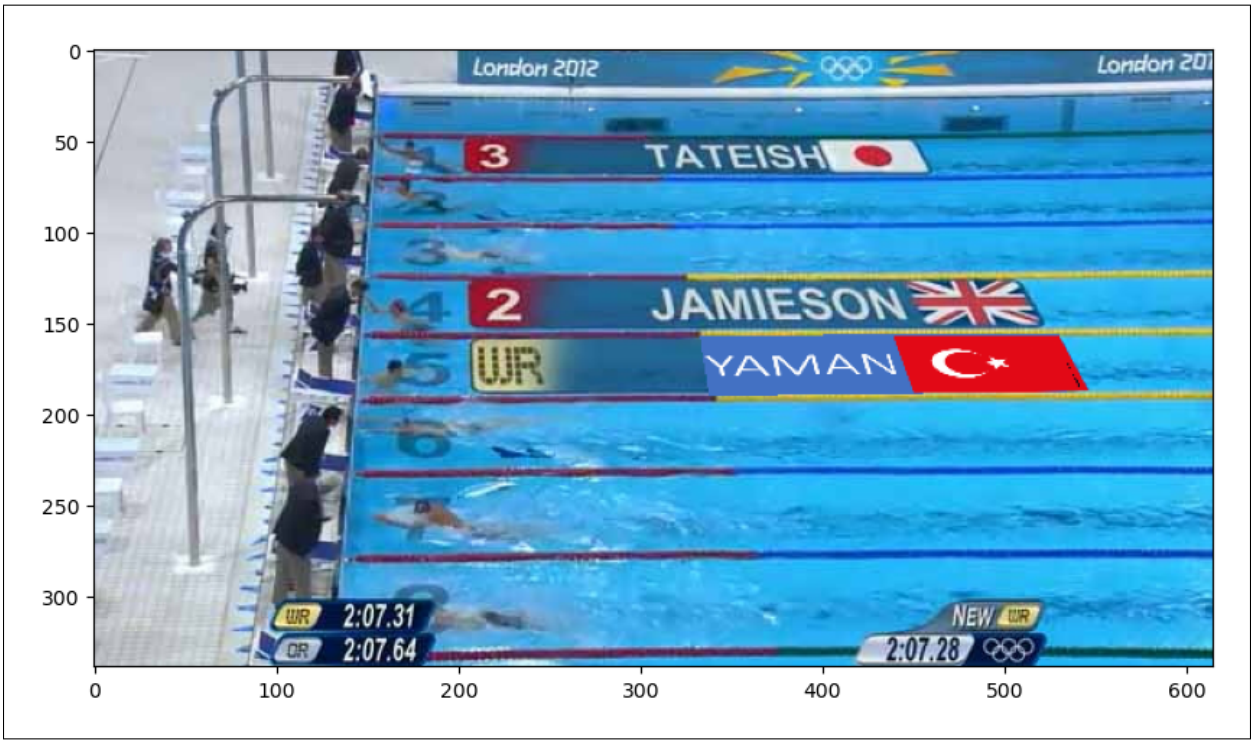
```
            loc_y = loc_homogeneous[1] / loc_homogeneous[2]
            loc = (loc_x,loc_y)

            if loc_x >= 0 and loc_x < image_width and \
                    loc_y >= 0 and loc_y < image_height:

                    pixel_value = interpolate(image1, loc)
                    warped_image[y, x] = pixel_value
    stitched_image = zeros_array
    #display_color(warped_image,True)
    for i in range(stitched_image.shape[0]):
        for j in range(stitched_image.shape[1]):
             for k in range(stitched_image.shape[2]):
                    if (image2[i,j,k] != 0 and warped_image[i,j,k] != 0):
                        stitched_image[i,j,k] = warped_image[i,j,k]
                    else:
                        stitched_image[i,j,k] = image2[i,j,k]
    return stitched_image
```

4.3 Visualize (3.0 points)

(See the Jupyter notebook.) Display the final image with you as the world champion.


London 2012
3
TATEISH
2
JAMIESON
WR
YAMAN
WR 2:07.31
OR 2:07.64
NEW WR
2:07.28
0
50
100
150
200
250
300
0
100
200
300
400
500
600

5 Eight-Point Algorithm (10.0 points)

In this question, you will implement the eight-point algorithm to reconstruct 3D points associated with 2D correspondences of an image pair.

5.1 Compute the Essential Matrix (2.0 points)

(See the Jupyter notebook). In this sub-part, you will compute the essential matrix using the eight-point algorithm. Make sure that your code is within the bounding box.

```
def compute_essential_matrix(correspondences):

    A = np.array([])
    for correspondence in correspondences:
        x,y = correspondence[0][0],correspondence[0][1]
        x_t,y_t = correspondence[1][0],correspondence[1][1]
        row = np.array([x_t*x,x_t*y,x_t,
                        y_t*x,y_t*y,y_t,
                        x,y,1])
        if(A.shape[0] == 0):
            A = row
        else:
            A = np.vstack((A,row))
    u, s, vh = np.linalg.svd(A, full_matrices=True)
    E = vh[-1,:].reshape((3,3))
    u_f, s_f, vh_f = np.linalg.svd(E, full_matrices=True)
    s_f[-1] = 0

    E = np.matmul(np.matmul(u_f,np.diag(s_f)),vh_f)
    return E
```

5.2 Compute the Translation and Rotation (2.0 points)

(See the Jupyter notebook). In this sub-part, you will compute the translation and rotation between the two images' cameras given the essential matrix. Make sure that your code is within the bounding box.

```
def compute_translation_rotation(essential_matrix):

    W = np.array([[0,-1,0],[1,0,0],[0,0,1]])
    u,s,vh = np.linalg.svd(essential_matrix, full_matrices=True)
    R = np.matmul(u,np.matmul(W , vh))
```

```
    T = np.matmul(u,np.matmul(np.diag(s),np.matmul(W,u.T)))
    t = np.array([T[2,1],T[0,2],T[1,0]])

    return t,R,T
```

5.3 Sanity Check Translation and Rotation (3.0 points)

(See the Jupyter notebook). In this sub-part, you will perform some sanity-checks on the translation and rotation obtained previously. Copy the output from the Jupyter notebook here.

Translation vector: [-0.70244976 -0.01835019 0.14775649]

Rotation matrix:

[[0.96765823 -0.01752971 0.25165504]

[0.01634127 0.99984327 0.00681171]

[-0.251735 -0.00247905 0.96779303]]

That:

[[1.11753402e-04 -1.53002559e-01 -1.83501945e-02]

[1.47756490e-01 4.94428979e-04 6.78484370e-01]

[1.87296502e-02 -7.02449764e-01 -6.06182381e-04]]

RT:

[[0.96765823 0.01634127 -0.251735]

[-0.01752971 0.99984327 -0.00247905]

[0.25165504 0.00681171 0.96779303]]

R-1:

[[0.96765823 0.01634127 -0.251735]

[-0.01752971 0.99984327 -0.00247905]

[0.25165504 0.00681171 0.96779303]]

5.4 Compute Depths (1.0 points)

(See the Jupyter notebook). In this sub-part, you will compute the depths of the 3D points corresponding to the given 2D correspondences. Make sure that your code is within the bounding box.

```
def compute_depths(correspondences, translation, rotation):
    depths = []
    for correspondence in correspondences:
        y = np.array([correspondence[0][0],correspondence[0][1],1]).reshape(-1,1)
        y_t = np.array([correspondence[1][0],correspondence[1][1],1]).reshape(-1,1)
        A = np.hstack((-np.matmul(rotation, y),y_t))
        depth = np.matmul(np.linalg.pinv(), translation)
        depths.append(np.array([depth[0],depth[1]]))
    return depths
```

5.5 Reconstruct 3D Points (1.0 points)

(See the Jupyter notebook). In this sub-part, you will reconstruct the 3D points given the 2D correspondences and the associated depths. Make sure that your code is within the bounding box.

```
def reconstruct_3d(correspondences, depths):
    corr_3d = []
    for idx,correspondence in enumerate(correspondences):
        y,Z = correspondence[0],depths[idx][0]
        y_t,Z_t = correspondence[1],depths[idx][1]
        X1 = Z * np.array([y[0],y[1],1])
        X2 = Z_t * np.array([y_t[0],y[1],1])
        corr_3d.append((X1,X2))
    return corr_3d
```

5.6 Check the 3D Points (1.0 points)

(See the Jupyter notebook). In this sub-part, you will check the reconstructed 3D points from the first image by reprojecting them into the second image. Copy the output from the Jupyter notebook here.

0.024302595899410355