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

INSTRUCTOR: Prof. Achuta Kadambi
TA: Rishi Upadhyay

NAME: Krish Patel
UID: 605796227

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. The overleaf is here `https://www.overleaf.com/read/gqyffskvzxww#9fafcf`. The code is here: `https://colab.research.google.com/drive/1WgPQdJCqsqEpcWuLOyWh5QdkvXVg6zvq?usp=sharing`

Submission

Submission will be done via Gradescope. You will need to submit three things: (1) this PDF with answers filled out, (2) Your .ipynb file, (3) a PDF printout of your .ipynb file with all cells executed. You do not need to create a folder, you will be able to upload all three files directly to gradescope.

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

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

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.

Laplacian of a 2D Gaussian would be the sum of both the partial derivatives which would be equal to:

$$\frac{1}{2\pi\sigma^2}\left(-\frac{1}{\sigma^2}+\frac{x^2}{\sigma^4}\right)e^{-\frac{x^2+y^2}{2\sigma^2}} + \frac{1}{2\pi\sigma^2}\left(-\frac{1}{\sigma^2}+\frac{y^2}{\sigma^4}\right)e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Simplifying this, we should get: $\frac{1}{2\pi\sigma^2}\left(-\frac{2}{\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} + \frac{x^2+y^2}{\sigma^4}e^{-\frac{x^2+y^2}{2\sigma^2}}\right)$

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 $L(x,y)$.

From the equations above, the relationship can be defined as: $\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \sigma L(x,y)$

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 $k \approx 1$.

The expression for this would be: $\frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma-\sigma}$

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 $k \approx 1$.

Multiplying the above equation with sigma would give the equation: $\frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma^2-\sigma^2}$

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.

Taylor Series approximation:

$$f(\mathbf{x}+\Delta\mathbf{x}) = f(x) + \Delta x \cdot f'(x) + \frac{\Delta x^2}{2!} f''(x)$$

This excludes the terms after the second derivative as they might be too small to be considered thus can be neglected.

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 $\frac{\partial f(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}}$.

Derivative of the Taylor Series Approximation with respect to Δx:

$$\frac{d}{d(\Delta x)}(f(\mathbf{x}+\Delta\mathbf{x})) = f'(x) + \Delta x f''(x) + \frac{\Delta x^2}{2} f'''(x) + \frac{\Delta x^3}{6} f''''(x) + \text{derivative}(O(\Delta x^5))$$

Excluding the negligible-ish terms:

$$\frac{d}{d(\Delta x)}(f(\mathbf{x}+\Delta\mathbf{x})) = f'(x) + \Delta x f''(x)$$

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.

For the extrema calculation, we find the values for which the derivative is 0. Considering the derivative of $O(\Delta x^5)$ is going to be considerably small and negligible, we don't consider this term. Thus, we set the derivative to zero and solve for $\Delta\mathbf{x}$:

$$f'(\mathbf{x}) + \Delta\mathbf{x} f''(\mathbf{x}) + \frac{\Delta\mathbf{x}^2}{2} f'''(\mathbf{x}) = 0$$

Solving this equation for $\Delta\mathbf{x}$ gives the extrema of the Taylor series approximation.

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.

To find the new keypoint via the Taylor series approximation, we need to compute the displacement $\Delta\mathbf{x}$ such that the gradient at the keypoint $\mathbf{x}$ is maximized. This displacement can be found by solving the equation:

$$\mathbf{0} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} + \Delta\mathbf{x} \frac{\partial^2 \mathbf{f}}{\partial \mathbf{x}^2}$$

Changing the subject:

$$\Delta\mathbf{x} = -\frac{\frac{\partial \mathbf{f}}{\partial \mathbf{x}}}{\frac{\partial^2 \mathbf{f}}{\partial \mathbf{x}^2}}$$

where $\mathbf{f}$ is the gradient magnitude computed from the difference of Gaussians (DoG) pyramid. Solving this equation for $\Delta\mathbf{x}$ gives us the displacement needed to maximize the gradient magnitude, resulting in the new keypoint.

2.5 Purpose of Keypoint Localization (3.0 points)

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

The primary goal of SIFT—to detect and describe local features in images in a way that is invariant to image scale and rotation, and robust to changes in illumination, noise, and small changes in viewpoint. There are many reasons why keypoint localization is done for sift. Keypoint localization helps with increasing the efficiency of the sift algorithm, making it more robust to scale changes, rotation, etc. It also helps detecting scale-variant features, as it allows for scale-invariant features to be detected, i.e it is invariant to the scale of the features(unlike corner detection algorithms). The scale space extrema detection, a crucial step in keypoint

localization, involves identifying keypoints at local extrema in the scale-space representation of the image. This process allows SIFT to capture features at multiple scales, which ensure that important details are not overlooked and provides an accurate representation of the image content. Keypoint localization improves the precision of feature points within an image by adjusting their positions to the exact spot, even between pixels, which makes matching these features between different images more reliable. By determining the main direction of each keypoint based on its surroundings, the algorithm ensures that features can be recognized no matter how the image is rotated, making it versatile in handling images from various angles.

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 closer to a different pixel than the original keypoint, it means that the original keypoint was not entirely accurate. To obtain a more precise estimate of the keypoint, we can use interpolation techniques. Interpolation helps us estimate the value of a point based on the values of nearby points. For example, we can use bilinear interpolation, which considers the values of the four nearest pixels to the original keypoint and calculates a weighted average to determine the new keypoint's position more accurately. Similarly, if the scale of the keypoint is inaccurate, we can adjust it using interpolation methods to better match the scale of the surrounding Gaussian kernels. This way, we can improve the accuracy of the keypoint localization process.

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):
    grayimg = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    sift = cv2.SIFT_create(nfeatures=num_features)
    keypoints, descriptors = sift.detectAndCompute(grayimg, None)
    return keypoints, descriptors
```

3.2 Finding Initial Correspondences (1.0 points)

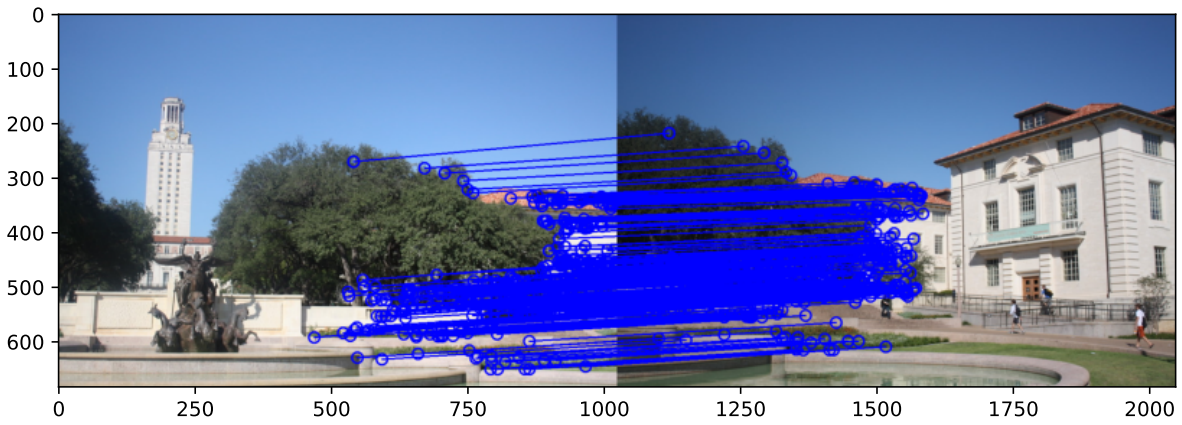
(See the Jupyter notebook). In this sub-part, you will write a function to obtain an initial set of correspondences by matching SIFT descriptors. Make sure that your code is within the bounding box.

```
def find_sift_correspondences(kp1, des1, kp2, des2, ratio):
    correspondences = []
    for idx, kp in enumerate(kp1):
        des = des1[idx]
        distances=[np.linalg.norm(des-des2[i]) for i in range(len(kp2))]
        idx_closest = np.argsort(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 = []
    for i in range(len(correspondences)):
        x1, y1 = correspondences[i][0]
        x2, y2 = correspondences[i][1]
        A.append([x1, y1, 1, 0, 0, 0, -x2 * x1, -x2 * y1, -x2])
        A.append([0, 0, 0, x1, y1, 1, -y2 * x1, -y2 * y1, -y2])
    A = np.array(A)
    U, S, V = np.linalg.svd(A)
    H = V[-1].reshape(3, 3)
    return H
```

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):
    out = []
    for out_point in points:
        out_point = np.array([out_point[0], out_point[1], 1])
        out_point = np.dot(homography, out_point)
        out_point = out_point / out_point[2]
        out.append(out_point[:2])
    return np.array(out)
```

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 i in range(len(correspondences)):
        point1 = np.array(correspondences[i][0])
        point2 = np.array(correspondences[i][1])
        point1_transformed = apply_homography(point2[None, :],
                np.linalg.inv(homography)).squeeze()
        if np.linalg.norm(point1-point1_transformed)<threshold:
            inliers.append(correspondences[i])
        else:
            outliers.append(correspondences[i])
    return inliers, outliers
```

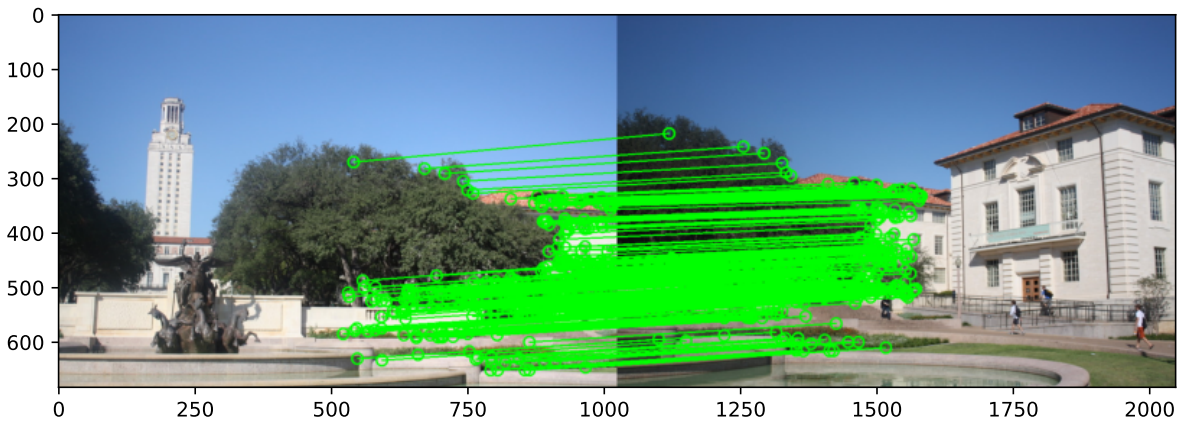
3.7 RANSAC Loop (2.0 points)

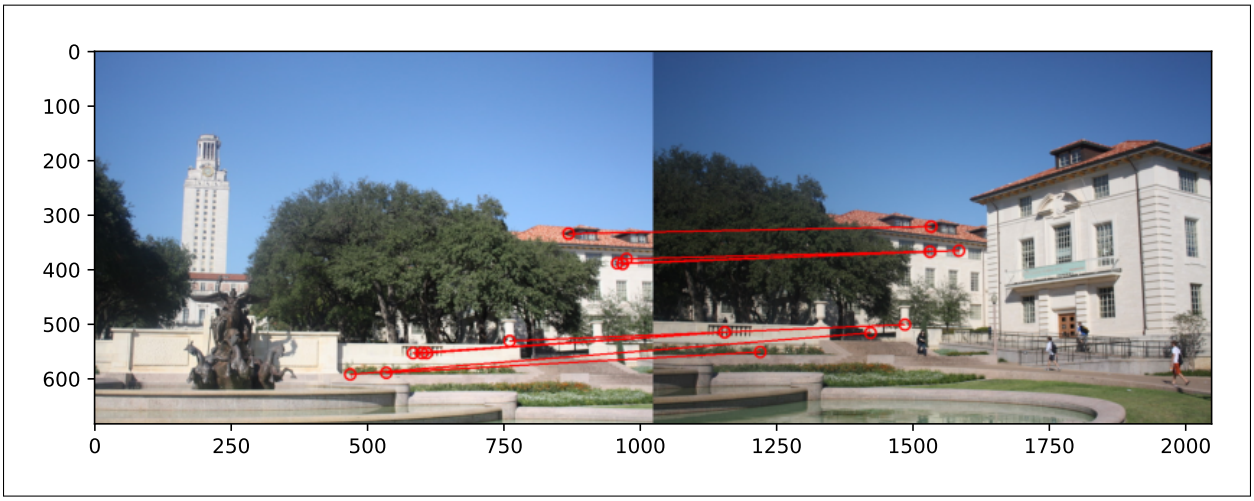
(See the Jupyter notebook). In this sub-part, you will write a function that implements the RANSAC loop to compute a homography matrix and its inlier and outlier correspondences. This part uses some of the earlier parts such as computing a homography and inliers. Make sure that your code is within the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

```
def ransac(correspondences,num_iterations,num_sampled_points, threshold):
    count = 0
    max_inlier_count = 0
    while(count < num_iterations):
        sample = random.sample(correspondences, num_sampled_points)
        inliers, outliers = compute_inliers(compute_homography(sample),
                correspondences, threshold)
        if len(inliers) > max_inlier_count:
            max_inlier_count = len(inliers)
            best_inliers = inliers
            best_outliers = outliers
        count += 1
    return compute_homography(sample), 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:
        dx = loc_x - x0
        dy = loc_y - y0
        pixel = (image[y0,x0]*(1-dx)*(1-dy)+image[y0,x0+1]*dx*(1-dy)
            +image[y0+1,x0]*(1-dx)*dy+image[y0+1,x0+1]*dx*dy)
    else:
        return 0 #out of bounds
    return pixel
```

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.

```
def stitch_image_given_H(front_img, back_img, homography_matrix):
    front_height, front_width = front_img.shape[:2]
    back_height, back_width = back_img.shape[:2]
    warp_canvas=np.zeros((back_height,2*back_width, front_img.shape[2]))
    empty_space = np.zeros((back_height, back_width, front_img.shape[2]))

    inverse_homography = np.linalg.inv(homography_matrix)
    expanded_back_img = np.hstack((empty_space, back_img))
    for vert_idx in range(back_height):
        for horiz_idx in range(-back_width, back_width):
            transformed_coords = np.dot(inverse_homography,
                    np.array([horiz_idx, vert_idx, 1]))
            mapped_col = transformed_coords[0] / transformed_coords[2]
            mapped_row = transformed_coords[1] / transformed_coords[2]
            if 0<=mapped_col<front_width and 0<=mapped_row< front_height:
                new_pixel=interpolate(front_img,
                            (mapped_col, mapped_row))
                warp_canvas[vert_idx,horiz_idx+back_width]=new_pixel

    stitched_result = np.hstack((empty_space, empty_space))
    for i in range(stitched_result.shape[0]):
        for j in range(stitched_result.shape[1]):
            for channel in range(stitched_result.shape[2]):
                if expanded_back_img[i, j, channel] != 0
                        and warp_canvas[i, j, channel] != 0:
                    stitched_result[i, j, channel] =
                        (expanded_back_img[i,j,channel]
                        warp_canvas[i, j,channel])/2
                elif expanded_back_img[i, j, channel] == 0:
                    stitched_result[i,j,channel]=warp_canvas[i,j,channel]
                elif warp_canvas[i, j, channel] == 0:
                    stitched_result[i, j, channel] =
                        expanded_back_img[i, j, channel]
    return stitched_result
```

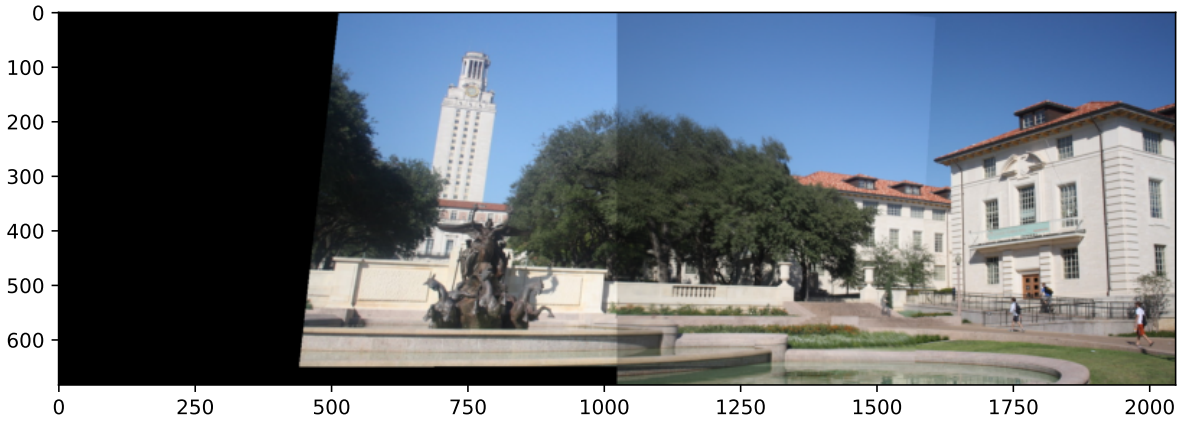
3.11 Image Stitching: Putting It All Together (1.0 points)

(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_points, inlier_threshold, use_ransac=True):
    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, _, __ = ransac(correspondences,ransac_iter,
            ransac_sampled_points,inlier_threshold)
    else:
        homography = compute_homography(correspondences)
    stitched_image = stitch_image_given_H(image1, image2, homography)
    return stitched_image
```

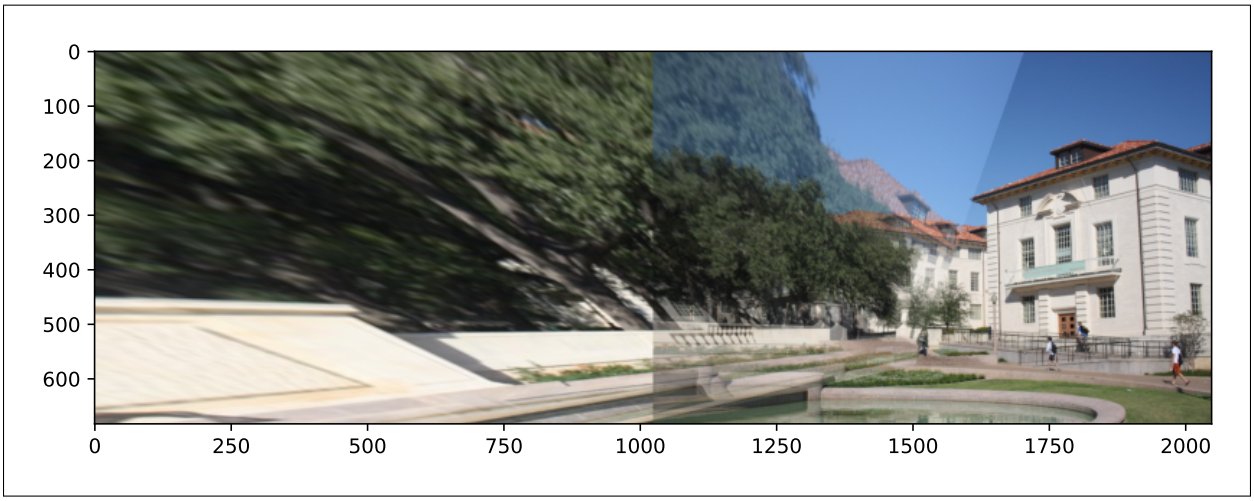
3.12 Visualizing the Stitched Image (1.0 points)

(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.

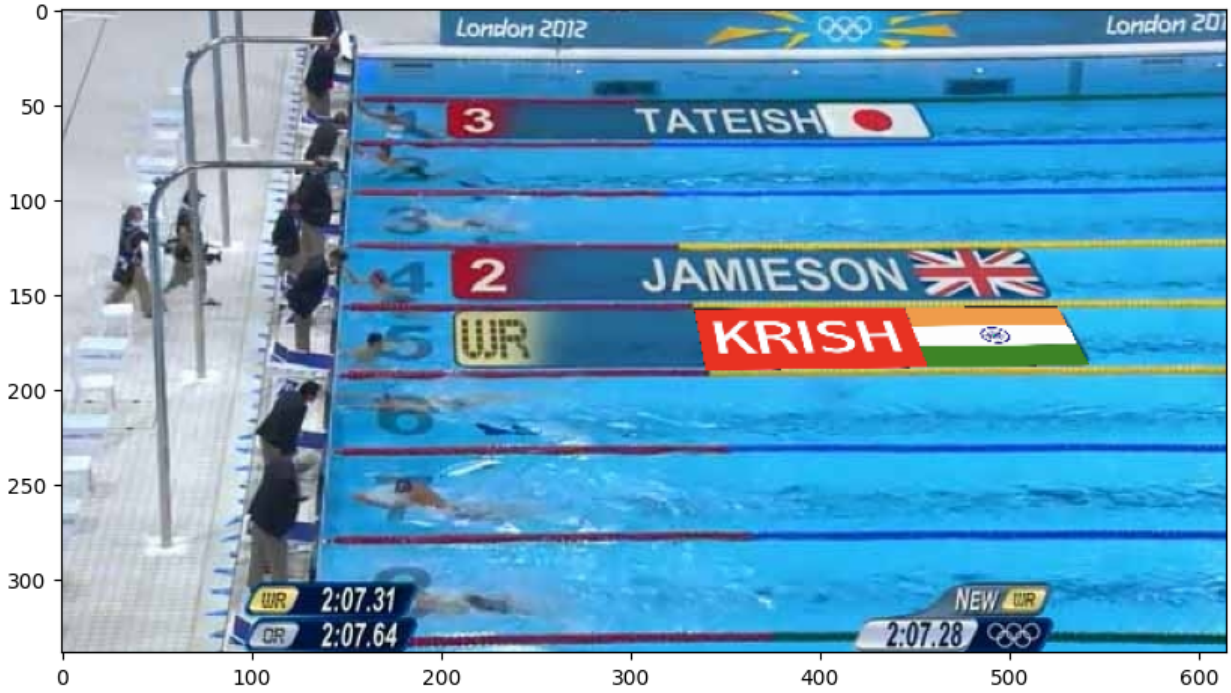

0
100
200
300
400
500
600
0
250
500
750
1000
1250
1500
1750
2000

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 = [0,0]
B_1 = [0,410]
C_1 = [1450,0]
D_1 = [1450,410]
correspondence = [
                  ([334,158], A_1),
                  ([340,190], B_1),
                  ([528,157], C_1),
                  ([545,187], D_1),
]
```

4.2 Stitching (1.0 points)

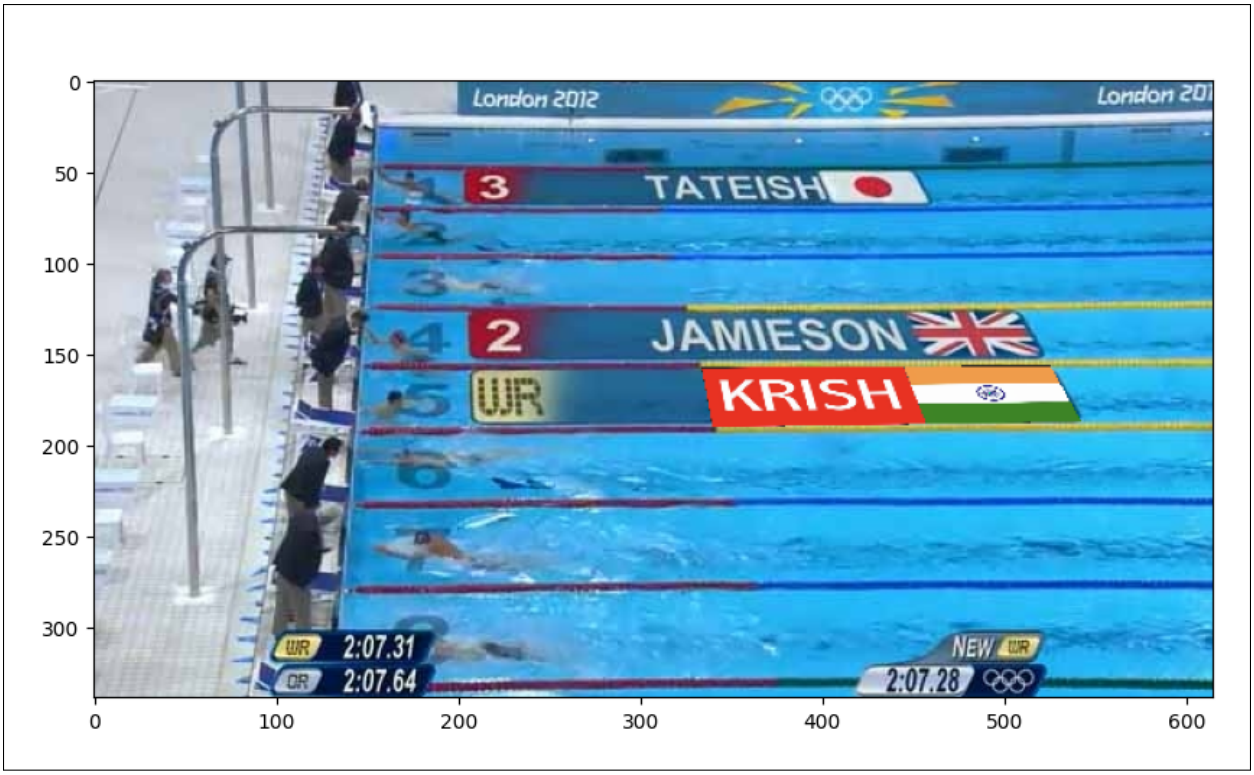
(See the Jupyter notebook). In the previous question, you stitched two images side-by-side. In this sub-part, you will stitch one image inside the other. Make sure that your code is within the bounding box. *Hint*: You will need to use code from the previous question and delete/modify a few lines from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

```
def stitch_image_given_H_new(image1, image2, homography):
    primary_height, primary_width = image1.shape[:2]
    secondary_height, secondary_width = image2.shape[:2]
    transformed_canvas = np.zeros((secondary_height,
            secondary_width, image1.shape[2]))
    for row in range(secondary_height):
        for col in range(secondary_width):
            homogeneous_coord = np.dot(homography, np.array([col,row, 1]))
            trans_col = homogeneous_coord[0] / homogeneous_coord[2]
            trans_row = homogeneous_coord[1] / homogeneous_coord[2]
            if 0<=trans_col<primary_width and 0<=trans_row<primary_height:
                pixel_val = interpolate(image1, (trans_col, trans_row))
```

```
                transformed_canvas[row, col] = pixel_val
    combined=np.where(transformed_canvas!=0,transformed_canvas,image2)
    return combined
```

4.3 Visualize (3.0 points)

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



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):

    matrix_A = np.array([])
    for pair in correspondences:
        point1_x, point1_y = pair[0][0], pair[0][1]
        point2_x, point2_y = pair[1][0], pair[1][1]
        new_row = np.array([point2_x*point1_x,point2_x*point1_y, point2_x,
                            point2_y*point1_x,point2_y*point1_y, point2_y,
                            point1_x,point1_y,1])
        if(matrix_A.shape[0] == 0):
            matrix_A = new_row
        else:
            matrix_A = np.vstack((matrix_A, new_row))
    u,s,singular_vh=np.linalg.svd(matrix_A,full_matrices=True)
    essential_matrix = singular_vh[-1, :].reshape((3, 3))
    u_final,s_final,vh_final=np.linalg.svd(essential_matrix,
            full_matrices=True)
    s_final[-1] = 0
    essential_matrix = np.matmul(np.matmul(u_final,
            np.diag(s_final)), vh_final)
    return essential_matrix
```

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_matrix = np.array([[0, -1, 0], [1, 0, 0], [0, 0, 1]])
    U, singular_values, V_transpose = np.linalg.svd(essential_matrix,
            full_matrices=True)
```

```
    rotation_matrix = U @ W_matrix @ V_transpose
    translation_matrix_form = U @ np.diag(singular_values)
            @ W_matrix @ U.T
    translation_vector = np.array([translation_matrix_form[2, 1],
        translation_matrix_form[0, 2], translation_matrix_form[1, 0]])
    return translation_vector, rotation_matrix, translation_matrix_form
```

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.

Output:

```
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]]
T_hat:
 [[ 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]]
R^T:
 [[ 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:
        yt=np.array([correspondence[1][0],correspondence[1][1],1])
                .reshape(-1,1)
        y = np.array([correspondence[0][0], correspondence[0][1], 1]).
                reshape(-1, 1)
        A = np.hstack((-np.matmul(rotation, y), yt))
        depth_solution = np.matmul(np.linalg.pinv(A), translation)
        depths.append(np.array([depth_solution[0], depth_solution[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):
    points_3d = []
    for i, pair in enumerate(correspondences):
        pt1, depth1 = pair[0], depths[i][0]
        pt2, depth2 = pair[1], depths[i][1]
        point_3d_1 = depth1 * np.array([pt1[0], pt1[1], 1])
        point_3d_2 = depth2 * np.array([pt2[0], pt2[1], 1])
        points_3d.append((point_3d_1, point_3d_2))
    return points_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.

Output

```
zip  <zip object at 0x15c596bc0>
0.024302595899405602
```