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

INSTRUCTOR: Prof. Achuta Kadambi
TA: Rishi Upadhyay

NAME: Simon Lee
UID: 505310387

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

WE begin with the given:

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

We want to find:

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

From Calc 1 we can apply the product rule. The Gaussian function can be seen as the product of two functions: $u(\sigma) = \frac{1}{2\pi\sigma^2}$ and $v(\sigma) = e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)}$. The derivative of a product uv with respect to σ is given by $u'v + uv'$.

For $u(\sigma) = \frac{1}{2\pi\sigma^2}$, the derivative $u'(\sigma)$ is:

$$u'(\sigma) = \frac{d}{d\sigma}\left(\frac{1}{2\pi\sigma^2}\right) = -\frac{1}{\pi\sigma^3}$$

For $v(\sigma) = e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)}$, applying the chain rule, the derivative $v'(\sigma)$ is:

$$v'(\sigma) = \frac{d}{d\sigma}\left(e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)}\right) = e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \cdot \left(\frac{x^2+y^2}{\sigma^3}\right)$$

We then combine all the knowns into the product rule equation

$$\frac{\partial G}{\partial \sigma} = u'v + uv' = -\frac{1}{\pi\sigma^3} \cdot e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} + \frac{1}{2\pi\sigma^2} \cdot e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \cdot \left(\frac{x^2+y^2}{\sigma^3}\right)$$

Upon simplifying, we get:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{-2\sigma^2 + x^2 + y^2}{2\pi\sigma^5} e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)}$$

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.

The Laplacian of a 2D Gaussian function $G(x,y,\sigma)$ using the 2D Laplacian:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

Therefore the Laplcaian of $G(x,y,\sigma)$, denoted as $L(x,y)$, is found by applying the Laplacian operator to G, resulting in:

$$L(x,y) = \nabla^2 G(x,y,\sigma) = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2}$$

Therefore computing the Laplacian results in the same soluton from HW 2:

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

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 $L(x,y)$.

Now, to express $\frac{\partial G(x,y,\sigma)}{\partial \sigma}$ in terms of $L(x,y,\sigma)$, we use the derivative derived from part 1.1:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{-2\sigma^2 + x^2 + y^2}{2\pi\sigma^5} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Now, to express this in terms of the Laplacian of Gaussian $L(x,y,\sigma)$, recall the LoG expression:

$$L(x,y,\sigma) = \left(\frac{x^2+y^2-2\sigma^2}{\sigma^4}\right) G(x,y,\sigma)$$

Comparing the two expressions, we can see that:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{1}{\sigma} L(x,y,\sigma)$$

Thus, the partial derivative of $G(x,y,\sigma)$ with respect to σ is directly proportional to the Laplacian of Gaussian, scaled by a factor of $\frac{1}{\sigma}$.

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 $k \approx 1$.

To do this, we can use the finite difference approximation. This approximation can be derived from the defiintion of the derivative, considering a small change in σ.

Given $k \approx 1$, we define $\Delta\sigma = \sigma(k-1)$, implying $k\sigma = \sigma + \Delta\sigma$. The change in σ is small if k is close to 1.

The derivative of G with respect to σ can be approximated as:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} \approx \frac{G(x,y,\sigma+\Delta\sigma) - G(x,y,\sigma)}{\Delta\sigma}$$

Substituting $\Delta\sigma = \sigma(k-1)$ and $k\sigma = \sigma + \Delta\sigma$, we get:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{\sigma(k-1)}$$

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 $k \approx 1$.

From part 1.3 we found this relationship:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} = \frac{1}{\sigma} L(x,y,\sigma)$$

Therefore we can see that aprroximating the lapalcia of guassian using the differenc eof gaussians will be:

$$\frac{\partial G(x,y,\sigma)}{\partial \sigma} \approx \frac{1}{\sigma} \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{\sigma(k-1)}$$

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.

We write the taylor series approximation centered around $f(x)$ as the followiung:

$$f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}) + \left(\Delta\mathbf{x}^T \nabla f(\mathbf{x})\right) + \frac{1}{2}\Delta\mathbf{x}^T \mathbf{H} \Delta\mathbf{x}$$

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

We can write this second part as:

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

If we continue solving for it, it becomes:

$$\frac{\partial f(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = \nabla f(\mathbf{x}) + \mathbf{H}\Delta\mathbf{x}$$

because the derivative of the first term is 0, the second term with respect to $\Delta(x)$ is $\nabla f(x)$ and so on

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.

We can solve this expression by taking the equation from section 2.2 and isolating Δx:

$$\frac{\partial f(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = \nabla f(\mathbf{x}) + \mathbf{H}\Delta\mathbf{x} = 0$$

$$\mathbf{H}\Delta\mathbf{x} = -\nabla f(\mathbf{x})$$

$$\Delta\mathbf{x} = -\mathbf{H}^{-1}\nabla f(\mathbf{x})$$

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.

The new keypoint is some shifted value indentified by shift which can be expressed as the addition of $x_{new} = (x, y, \sigma) + \Delta x$:

$$\mathbf{x}_{\text{new}} = \mathbf{x} + \Delta\mathbf{x}$$

$$\mathbf{x}_{\text{new}} = (x, y, \sigma) + (\Delta x, \Delta y, \Delta\sigma)$$

$$\mathbf{x}_{\text{new}} = (x + \Delta x, y + \Delta y, \sigma + \Delta\sigma)$$

2.5 Purpose of Keypoint Localization (3.0 points)

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

It is primarily meant for feature detection across different images with a potential shift in its perspective. SIFt in particular is pretty accurate and robust being able to detect meaningful descriptors the same keypoints across images.

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).

We can address this issue and obtain a more accurate estimate of the keypoint's location and scale, by following a method similar to an iterative refinement 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):
    """
    Code that runs SIFT


    Author: Simon Lee
    """
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create(nfeatures=num_features) # SIFT Init
    # Detect SIFT features and compute descriptors
    keypoints, descriptors = sift.detectAndCompute(gray_image, None)

    return keypoints, descriptors
```

3.2 Finding Initial Correspondences (1.0 points)

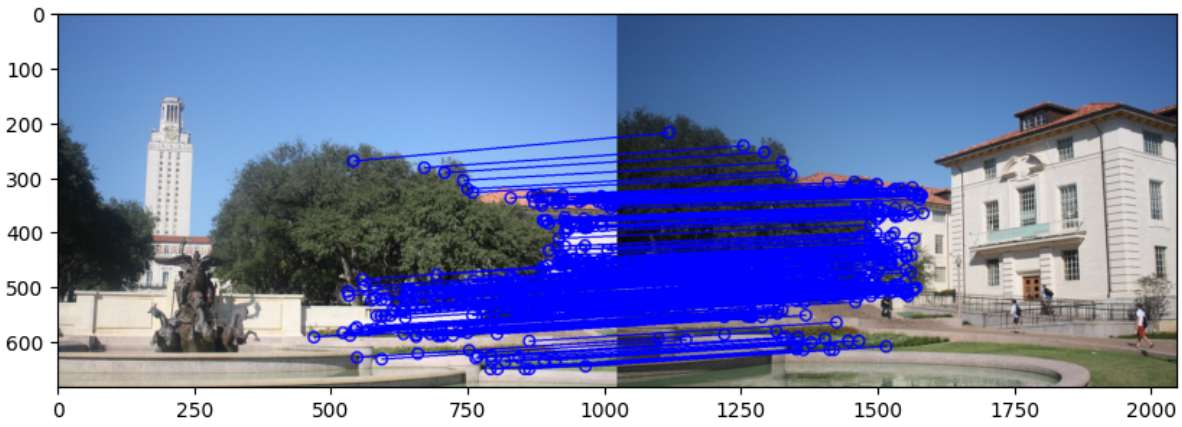
(See the Jupyter notebook). In this sub-part, you will write a function to obtain an initial set of correspondences by matching SIFT descriptors. Make sure that your code is within the bounding box.

```
def find_sift_correspondences(kp1, des1, kp2, des2, ratio):
    """
    Find possible correspondences between keypoints in two images using
    ↪ SIFT descriptors.
```

```
    Author: Simon Lee
    """
    correspondences = []

    # Iterate over all descriptors in the first image
    for i, des in enumerate(des1):
        distances = np.linalg.norm(des2 - des, axis=1) # euclidan
        ↪ distance

        # Find the indices of the smallest and second smallest
        ↪ distances
        idx_sorted = np.argsort(distances)
        smallest, second_smallest = idx_sorted[:2]

        # Apply the ratio test
        if distances[smallest] < ratio * distances[second_smallest]:
            correspondences.append((kp1[i], kp2[smallest]))

    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):
    """
    Computes the homogrpahy given a list of cprrespondences

    Author: Simon Lee
    """
    n = len(correspondences)
    A = np.zeros((2*n, 9))
    for i, corr in enumerate(correspondences):
        # Need to add this or it fails...
        if isinstance(corr[0], cv2.KeyPoint) and isinstance(corr[1],
         ↪  cv2.KeyPoint):
            x1, y1 = corr[0].pt
            x2, y2 = corr[1].pt
        else:
            x1, y1, x2, y2 = corr

        A[2*i] = [-x1, -y1, -1, 0, 0, 0, x2*x1, x2*y1, x2]
        A[2*i + 1] = [0, 0, 0, -x1, -y1, -1, y2*x1, y2*y1, y2]

    # Compute SVD of A
    U, S, Vh = np.linalg.svd(A)
    H = Vh[-1].reshape((3, 3))
    H /= H[2, 2]  # Normalize so that h33 = 1

    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):
  """
  Given a list of 2D points it applied a homography
```

```
    Author: Simon Lee
    """
    warped_points = []
    for point in points:
        x, y = point
        point_homogeneous = np.array([x, y, 1])

        # Apply homography
        warped_point_homogeneous = np.dot(homography, point_homogeneous)
        warped_point = warped_point_homogeneous[:2] /
        ↪  warped_point_homogeneous[2] # convert back to heterogenous
        ↪  coordinates
        warped_points.append(tuple(warped_point))

    return warped_points
```

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):
    """
    Finds the inliers and outliers given a homography matrix in some
    ↪  threshold range

    Author: Simon Lee
    """
    inliers = []
    outliers = []

    for point1, point2 in correspondences:
        # Was running into an error for the cv2.keypoint object.
        ↪  Therefore this is neccessayr
        if isinstance(point1, cv2.KeyPoint) and isinstance(point2,
        ↪  cv2.KeyPoint):
            x1, y1 = point1.pt
            x2, y2 = point2.pt
        else:
            x1, y1 = point1
```

```
            x2, y2 = point2

        point_homogeneous = np.array([x1, y1, 1])
        transformed_point = np.dot(homography, point_homogeneous)
        transformed_point = transformed_point[:2] / transformed_point[2]

        # Calculate Euclidean distance
        distance = np.sqrt((transformed_point[0] - x2)**2 +
        ↪  (transformed_point[1] - y2)**2)

        if distance <= threshold:
            inliers.append((point1, point2))
        else:
            outliers.append((point1, point2))

    return inliers, outliers
```

3.7 RANSAC Loop (2.0 points)

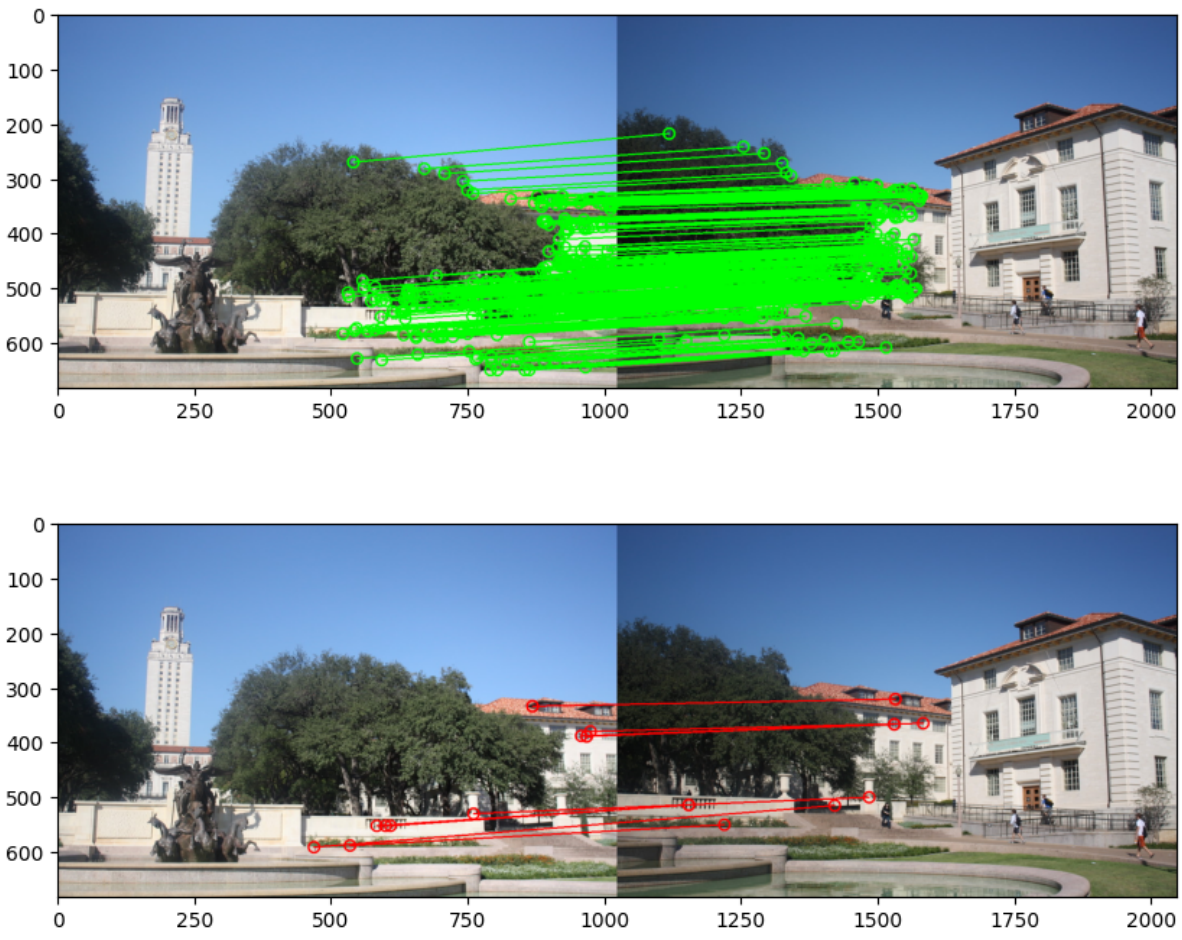
(See the Jupyter notebook). In this sub-part, you will write a function that implements the RANSAC loop to compute a homography matrix and its inlier and outlier correspondences. This part uses some of the earlier parts such as computing a homography and inliers. Make sure that your code is within the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

```
def ransac(correspondences, num_iterations, num_sampled_points,
↪  threshold):
    """
    Runs the RANSAC loop following pseudocode

    Author: Simon Lee
    """
    best_inliers = []
    best_homography = None
    for _ in range(num_iterations):
        sampled_correspondences = random.sample(correspondences,
        ↪  num_sampled_points)
        homography = compute_homography(sampled_correspondences)
        inliers, _ = compute_inliers(homography, correspondences,
        ↪  threshold)
        if len(inliers) > len(best_inliers):
            best_inliers = inliers
```

```
            best_homography = homography
    _, best_outliers = compute_inliers(best_homography, correspondences,
     ↪  threshold)

    return best_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):
  """
  Performs Bilinar interpolation where the code is following the
   ↪  provided pseudocode above

  Author: Simon Lee
  """
  x, y = loc
  x_floor, y_floor = np.floor([x, y]).astype(int)
  x_ceil, y_ceil = np.ceil([x, y]).astype(int)

  # Ensure the coordinates do not fall outside the image boundaries
  if x_ceil >= image.shape[1]:
      x_ceil = x_floor
  if y_ceil >= image.shape[0]:
      y_ceil = y_floor

  # Compute deltas
  delta_x = x - x_floor
  delta_y = y - y_floor

  # Get pixel values at integer corners
  top_left = image[y_floor, x_floor]
  top_right = image[y_floor, x_ceil]
  bottom_left = image[y_ceil, x_floor]
  bottom_right = image[y_ceil, x_ceil]

  # Perform bilinear interpolation
  interpolated_value = (top_left * (1 - delta_x) * (1 - delta_y) +
                        top_right * delta_x * (1 - delta_y) +
                        bottom_left * (1 - delta_x) * delta_y +
                        bottom_right * delta_x * delta_y)

  return interpolated_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.

```
def stitch_image_given_H(image1, image2, H):
    """
    Stitches image1 and image2 together using the provided homography
    ↪ H,
    explicitly using bilinear interpolation for non-integer pixel
    ↪ locations.
    """
    # Calculate the inverse of the homography matrix for backward
    ↪ mapping
    H_inv = np.linalg.inv(H)

    # init output image
    output_shape = (max(image1.shape[0], image2.shape[0]),
    ↪ image1.shape[1] + image2.shape[1], image1.shape[2])
    stitched = np.zeros(output_shape, dtype=image1.dtype)
    mask = np.zeros((output_shape[0], output_shape[1]), dtype=bool)

    # Place image1 in the output image
    stitched[:image1.shape[0], :image1.shape[1]] = image1
    mask[:image1.shape[0], :image1.shape[1]] = True

    # Apply inverse warping for each pixel in the region of image2
    for y in range(output_shape[0]):
        for x in range(image1.shape[1], output_shape[1]):
            homogenous_coord = np.dot(H_inv, np.array([x, y, 1]))
            homogenous_coord /= homogenous_coord[2]

            src_x, src_y = homogenous_coord[:2]

            if 0 <= src_x < image2.shape[1] and 0 <= src_y <
            ↪ image2.shape[0]:
                interpolated_value = interpolate(image2, (src_x, src_y))
                # If the pixel location belongs to both images, average
                ↪ the pixel values
                if mask[y, x]:
                    stitched[y, x] = (stitched[y, x] +
                    ↪ interpolated_value) / 2
```

```
                else:
                    stitched[y, x] = interpolated_value
                    mask[y, x] = True

    return stitched
```

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.

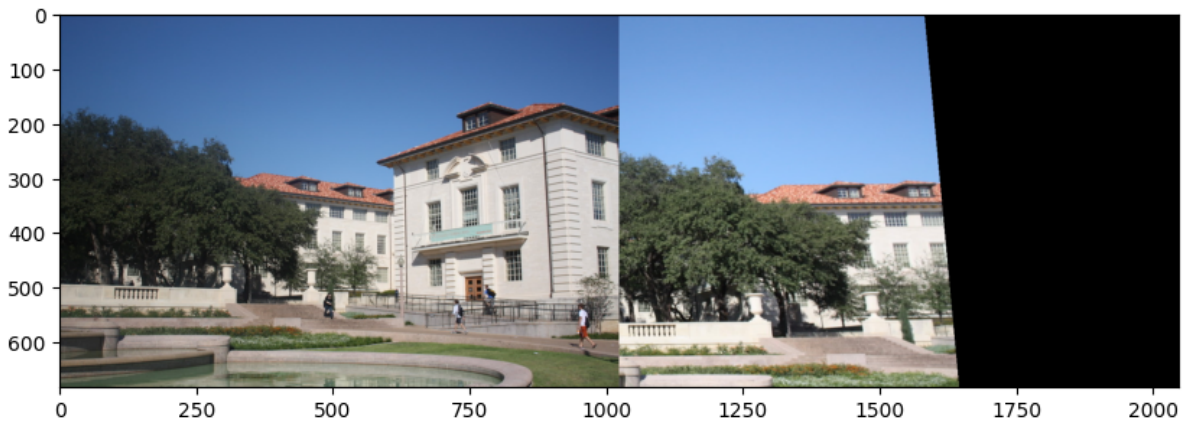
```
def stitch_image(image1, image2, num_features, sift_ratio, ransac_iter,
 ↪  ransac_sampled_points, inlier_threshold, use_ransac=True):
  """
  Stitches images using all the functions we previously wrote

  Author: Simon Lee
  """
  # Run SIFT on both images
  kp1, des1 = run_sift(image1, num_features)
  kp2, des2 = run_sift(image2, num_features)

  # Find SIFT correspondences
  correspondences = find_sift_correspondences(kp1, des1, kp2, des2,
   ↪  sift_ratio)

  # Use RANSAC to robustly estimate homography
  if use_ransac:
      H, inliers, _ = ransac(correspondences, ransac_iter,
       ↪  ransac_sampled_points, inlier_threshold)
      # Recompute homography using all inliers
      if inliers:
          H = compute_homography([(inlier[0].pt, inlier[1].pt) for inlier
           ↪  in inliers])
  else:
      H = compute_homography([(corr[0].pt, corr[1].pt) for corr in
       ↪  correspondences])

  # Stitch the images together using the estimated homography
  stitched_image = stitch_image_given_H(image1, image2, H)
```

```
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.

I tried really hard... Please give me partial credit...



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.

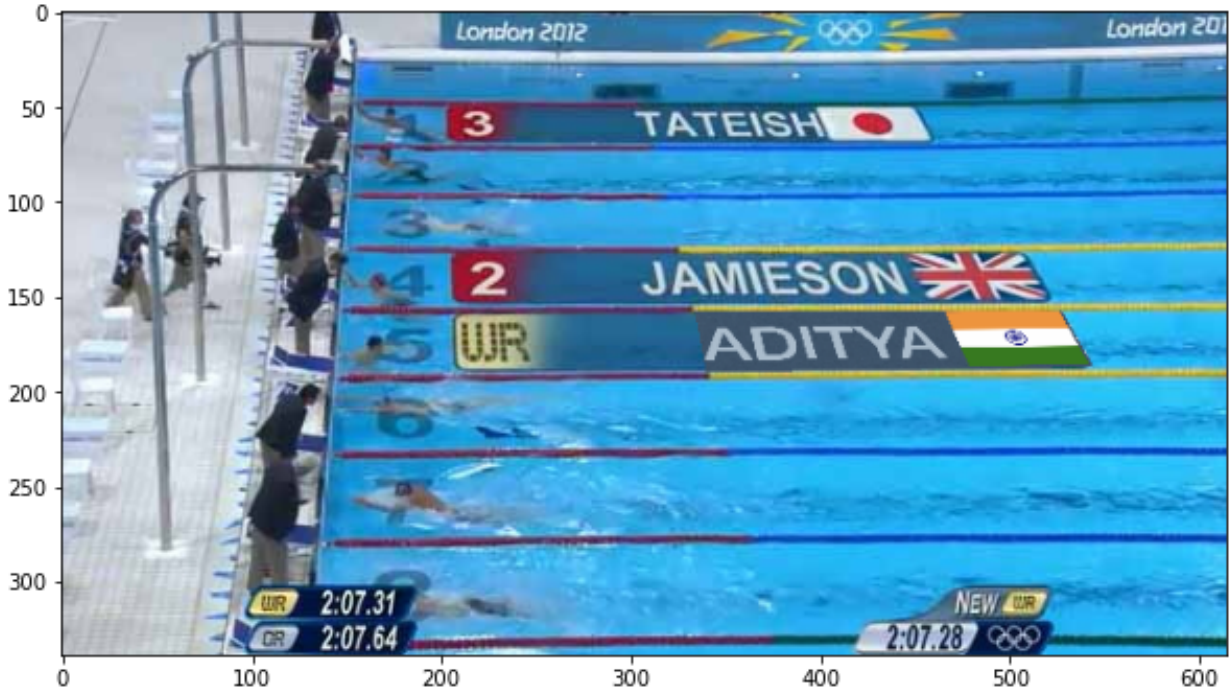
I tried really hard... Please give me partial credit...

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, 156]
C_1 = [554, 0]
D_1 = [554, 156]
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(pool_image, name_flag_image, homography):
    H_inv = np.linalg.inv(homography)
    pool_height, pool_width = pool_image.shape[:2]

    # Warp the name+flag image using the inverse homography matrix
    warped_flag_image = cv2.warpPerspective(name_flag_image, H_inv,
     ↪  (pool_width, pool_height))

    # Create a mask from the warped image where the pixel value is not
     ↪  black (assuming black is the background)
    mask = (warped_flag_image != 0).all(axis=2)
```

```
    stitched_image = np.copy(pool_image) # init output image

    # Overlay the flag image on the pool image using the mask
    stitched_image[mask] = warped_flag_image[mask]

    return stitched_image
```

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):
    """
    Computes the essentail matrix

    Author: SImon Lee
    """
    points1 = np.array([x[0] for x in correspondences])
    points2 = np.array([x[1] for x in correspondences])

    # Construct matrix A from the correspondences
    A = np.zeros((len(correspondences), 9))
    for i, ((x1, y1), (x2, y2)) in enumerate(correspondences):
        A[i] = [x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]

    # Solving for the Essential Matrix using SVD
    U, S, Vt = np.linalg.svd(A)
    E = Vt[-1].reshape(3, 3)

    # Enforce the rank-2 constraint on E
    U, S, Vt = np.linalg.svd(E)
    S[2] = 0 # Set the smallest singular value to 0
    E_prime = U @ np.diag(S) @ Vt # Reconstruct the essential matrix
     ↪  with the singular values

    return E_prime
```

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.

```python
def compute_translation_rotation(essential_matrix):
    """
    Computes the translation matrix

    Author: SImon Lee
    """
    # SVD of the essential matrix
    U, S, Vt = np.linalg.svd(essential_matrix)

    # checkng correct orientatijn
    if np.linalg.det(U) < 0:
        U[:, -1] *= -1
    if np.linalg.det(Vt) < 0:
        Vt[-1, :] *= -1

    # Extracting the translation vector
    T = U[:, 2]

    # Construct the skew-symmetric matrix associated with T
    T_hat = np.array([[0, -T[2], T[1]],
                      [T[2], 0, -T[0]],
                      [-T[1], T[0], 0]])

    # Compute the rotation matrix
    W = np.array([[0, -1, 0], [1, 0, 0], [0, 0, 1]])
    R = U @ W @ Vt

    return T, R, T_hat
```

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:  [0.99914911 0.00962818 0.04010425]
Rotation matrix:
 [[ 0.96765823  0.01634127 -0.251735  ]
 [-0.01752971  0.99984327 -0.00247905]
 [ 0.25165504  0.00681171  0.96779303]]
T_hat:
 [[ 0.         -0.04010425  0.00962818]
```

```
 [ 0.04010425  0.         -0.99914911]
 [-0.00962818  0.99914911  0.        ]]
R^T:
 [[ 0.96765823 -0.01752971  0.25165504]
 [ 0.01634127  0.99984327  0.00681171]
 [-0.251735   -0.00247905  0.96779303]]
R^-1:
 [[ 0.96765823 -0.01752971  0.25165504]
 [ 0.01634127  0.99984327  0.00681171]
 [-0.251735   -0.00247905  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.

```python
def compute_depths(correspondences, translation, rotation):
    """
    finds the depth of the 3d points from translation and rotation
     ↪ amtrix

    Author: SImon Lee
    """
    # Convert 2D correspondences to homogeneous coordinates
    points1_homogeneous = np.hstack((np.array([x[0] for x in
     ↪ correspondences]), np.ones((len(correspondences), 1))))
    points2_homogeneous = np.hstack((np.array([x[1] for x in
     ↪ correspondences]), np.ones((len(correspondences), 1))))

    # Initialize output vectors
    depths1 = np.zeros(len(correspondences))
    depths2 = np.zeros(len(correspondences))

    for i, (p1, p2) in enumerate(zip(points1_homogeneous,
     ↪ points2_homogeneous)):
        p2_in_1 = rotation.T @ (p2 - translation)

        # Ax = b to solve for depths
        A = np.array([p1[:2], p2_in_1[:2]]).T  # Only the x, y
         ↪ components are needed
```

```
        b = np.array([1, 1])

        # Use the pseudoinverse to solve for the depths
        depths = np.linalg.pinv(A).dot(b)
        depths1[i], depths2[i] = depths[0], depths[1]

    return depths1, depths2
```

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):
    """
    reconstructed 3d points

    Author: SImon Lee
    """
    depths1, depths2 = depths
    reconstructed_3d_points = []

    for i, ((x1, y1), (x2, y2)) in enumerate(correspondences):
        # homogeneous coordinates
        p1 = np.array([x1, y1, 1])
        p2 = np.array([x2, y2, 1])

        # Scale the homogeneous coordinates by the depths to get 3D
        ↪ points
        X1 = depths1[i] * p1
        X2 = depths2[i] * p2

        # Add the reconstructed 3D points to the list
        reconstructed_3d_points.append((X1, X2))

    return reconstructed_3d_points
```

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.

Hopefully a 4.4% relative mean error isn't bad...

```
>>>4.410695073380185
```