Wensong Hu
EECS
University of Michigan

March 3, 2024

HW3 — Fitting Models and Image Wraping

1 RANSAC and Fitting Models

1.1 Task1: RANSAC Theory

1.1.1 Minimum # of points

To compute a putative model, the minimum number of 3D points needed to sample in an iteration is three.

1.1.2 Probability single iteration fials

The probability P that the data picked for the putative model in a single iteration fails, assuming an outlier ratio $e = 0.5$, is calculated as:

$$P = 1 - (1 - e)^s$$

Where $s = 3$ for the case of 3D planes. Plugging the values we get:

$$P = 1 - (0.5)^3 = 0.875$$

Thus, the probability of failure in a single iteration is 87.5

1.1.3 Minimum # of RANSAC trials

The minimum number of RANSAC trials n needed to achieve at least a 98% chance of success P, with an outlier ratio $e = 0.5$, is given by the formula:

$$1 - (1 - (1 - e)^s)^n \geq P$$

Rearranging for n, we get:

$$n \geq \frac{\log(1 - P)}{\log(1 - (1 - e)^s)}$$

Plugging in the values for $P = 0.98$ and $s = 3$, we find:

$$n \geq \frac{\log(1 - 0.98)}{\log(1 - (0.5)^3)}$$

Upon calculation, we find:

$$n \geq 30$$

Therefore, at least 30 trials are needed to have a 98% chance of success.

1.2 Task2: Fitting Linear Transformations

1.2.1 Degrees of freedom, Minimum # of correspondences

The matrix M representing a linear transformation in $\mathbb{R}^{2\times 2}$ has four degrees of freedom since it can be parameterized as $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$. Therefore, to fully constrain or estimate M, we need a minimum of two 2D correspondences.

1.2.2 Form of A, m, and b

Given 2D correspondences $(x_i', y_i')^T \leftrightarrow (x_i, y_i)^T$, we formulate the fitting problem as a least-squares problem:

$$\arg\min_{m\in\mathbb{R}^4} \|Am - b\|^2$$

where $m = [a, b, c, d]^T$ contains all the parameters of M, A is a $2N \times 4$ matrix dependent on the points (x_i, y_i), and b is a $2N \times 1$ vector containing the coordinates (x_i', y_i'). The matrices are defined as:

$$A = \begin{bmatrix} x_1 & y_1 & 0 & 0 \\ 0 & 0 & x_1 & y_1 \\ x_2 & y_2 & 0 & 0 \\ 0 & 0 & x_2 & y_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_N & y_N & 0 & 0 \\ 0 & 0 & x_N & y_N \end{bmatrix}, \quad b = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ \vdots \\ x_N' \\ y_N' \end{bmatrix}$$

The solution m that minimizes the sum of squared differences is found by solving the normal equations:

$$m = (A^T A)^{-1} A^T b$$

This provides the parameters of the linear transformation that best fits the correspondences in the least-squares sense.

1.3 Task3: Fitting Affine Transformations

1.3.1 Code implementation and Report (S, t)

For points_case_1.npy:

$$S = \begin{bmatrix} 1.41444296 & -1.41424374 \\ -0.70762108 & -0.70690933 \end{bmatrix} \tag{1-1}$$

$$t = \begin{bmatrix} 0.09998617 \\ 0.20014656 \end{bmatrix} \tag{1-2}$$

```python
def p3(filename: str):
    # code for Task 3
    # 1. load points X from task3/
    X = np.load(filename)
    N, D = X.shape

    # 2. fit a transformation y=Sx+t
    X_train = X[:, :2]
    y_train = X[:, 2:]

    A = np.zeros((2*N, D+2))
    A[:N, :2] = np.copy(X_train)
    A[N:, 2:4] = np.copy(X_train)
    A[:N, 4] = 1.0
    A[N: , 5] = 1.0

    b = np.zeros((2*N, 1))
    b[:N, 0] = y_train[:, 0]
    b[N:, 0] = y_train[:, 1]

    Result = np.linalg.lstsq(A, b)
    S = Result[0][:4].reshape(2, 2)
    t = Result[0][4:]
    # print(S, t)

    # 3. transform the points
    X_transformed = (S @ X_train.T + t).T

    # 4. plot the original points and transformed points
    plt.scatter(X[:, 0], X[:, 1], label='Origianl points', c='blue', s=1)
    plt.scatter(X[:, 2], X[:, 3], label='Transformed GT', c='red', s=1)
    plt.scatter(X_transformed[:, 0], X_transformed[:, 1], label='Transformed points', c='green', s=1.5)
    plt.legend()
    plt.show()

    return S, t
```

1.3.2 Plots

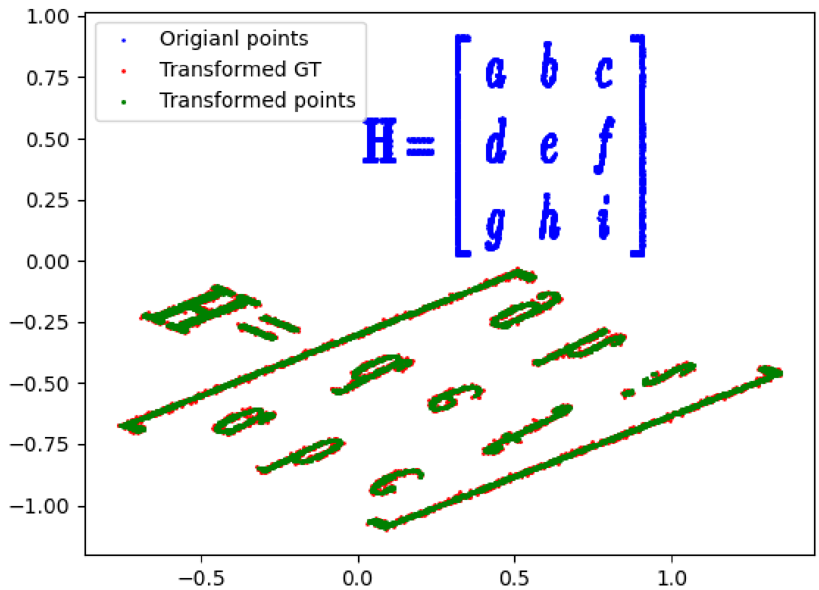

Figure 1: Plot for points_case_1.npy

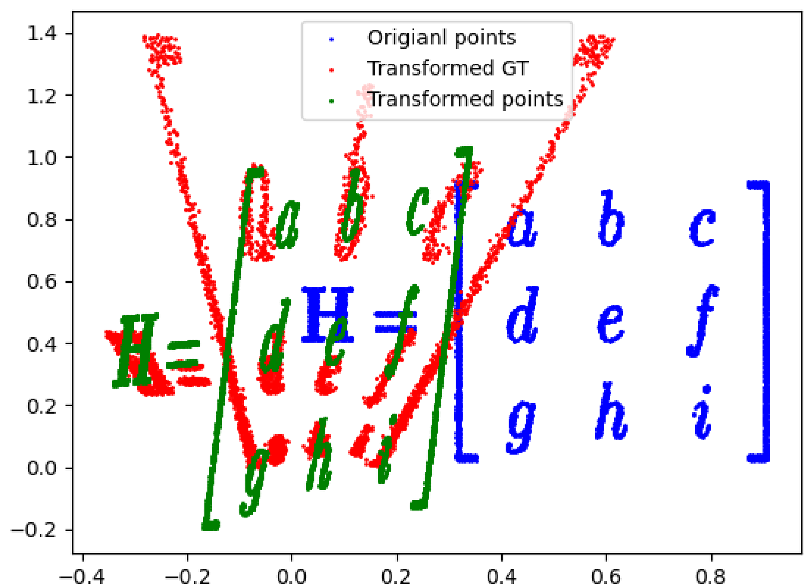

Figure 2: Plot for points_case_2.npy

1.3.3 Discussion

Affine transformation describe the data transformation in points_cast_1.npy quite good, but performs bad for points_cast_2.npy. This is because points_cast_1.npy is affine transformation, which can be described with 6 degree of freedom. But points_cast_2.npy is projection transformation, or homography, this transformation need higher dimension to describe.

1.4 Task4: Fitting Homographies

1.4.1 Implementation for fit_homography() in homography.py

Submitted to Canvas.

```python
"""
Homography fitting functions
You should write these
"""
import numpy as np
from common import homography_transform

def fit_homography(XY):
    '''
    Given a set of N correspondences XY of the form [x,y,x',y'],
    fit a homography from [x,y,1] to [x',y',1].

    Input - XY: an array with size(N,4), each row contains two
            points in the form [x_i, y_i, x'_i, y'_i] (1,4)
    Output -H: a (3,3) homography matrix that (if the correspondences can be
            described by a homography) satisfies [x',y',1]^T === H [x,y,1]^T

    '''
    N = XY.shape[0]
    A = np.zeros((2*N, 9))
    for i in range(N):
        x, y, xp, yp = XY[i]
        # A[2*i] = [-x, -y, -1, 0, 0, 0, x*xp, y*xp, xp]
        # A[2*i+1] = [0, 0, 0, -x, -y, -1, x*yp, y*yp, yp]
        A[2*i] = [0, 0, 0, -x, -y, -1, x*yp, y*yp, yp]
        A[2*i+1] = [x, y, 1, 0, 0, 0, -x*xp, -y*xp, -xp]

    # Perform Singular Value Decomposition (SVD)
    U, S, Vt = np.linalg.svd(A)

    # The solution is the last column of V (or the last row of V transpose)
    h = Vt[-1]
    # Normalize h
    h /= np.linalg.norm(h)
    # Reshape h to get the homography matrix H
    H = h.reshape(3, 3)
    return H
```

1.4.2 Report H

H for case 1 and case 4:

$$H1 = \begin{bmatrix} 1.00555949e+00 & 1.61370672e-03 & -1.35143989e-01 \\ 2.56045861e-03 & 6.22536404e-01 & -7.35872070e-01 \\ 4.51704286e-05 & 3.59823762e-05 & 1.00000000e+00 \end{bmatrix} \tag{1-3}$$

$$H4 = \begin{bmatrix} 1.63877010e-14 & 1.00000000e+00 & 1.04256051e-13 \\ 1.00000000e+00 & 1.03129045e-16 & 6.32521301e-14 \\ 3.68894767e-17 & 9.67901093e-17 & 1.00000000e+00 \end{bmatrix} \tag{1-4}$$

1.4.3 Plots

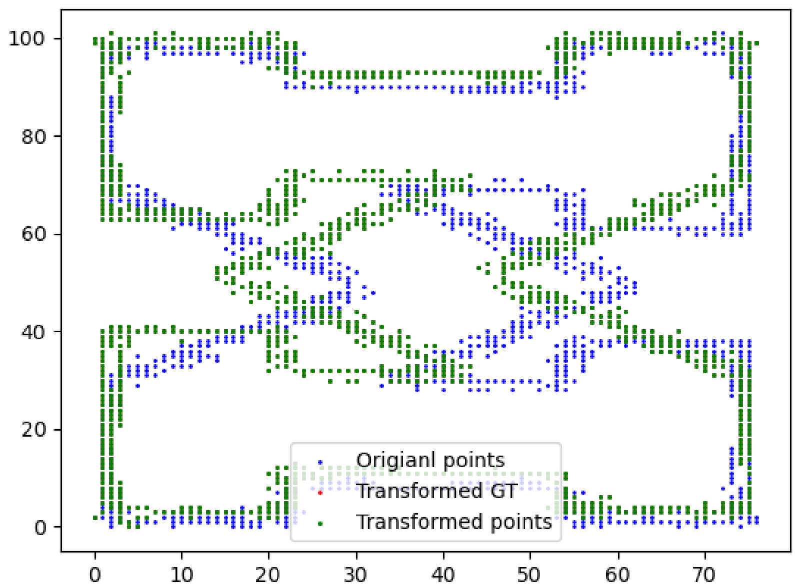

Figure 3: Plot for points_case_5.npy

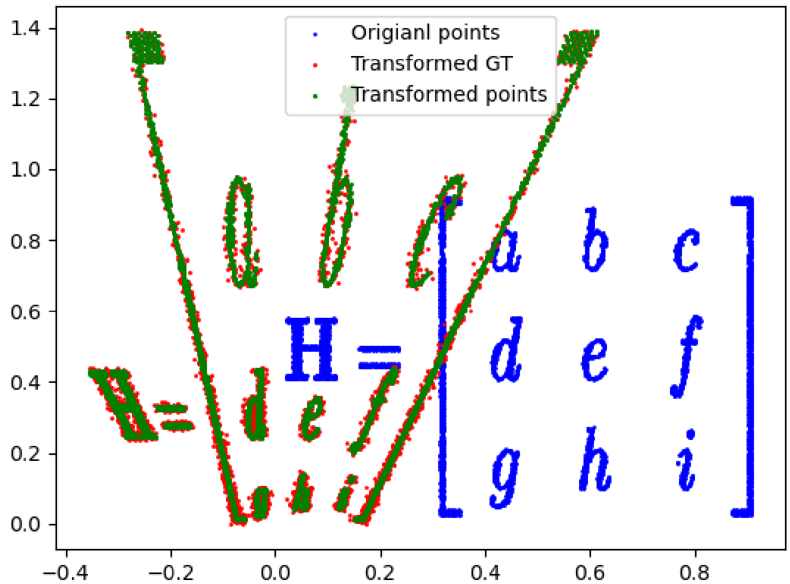

Figure 4: points_case_9.npy

2 Image Warping and Homographies

2.1 Task5: Synthetic Views - Name that Book!

2.1.1 Code implement of make_synthetic_view(sceneImage,corners,size) in task5.py

Code submitted to Canvas.

```
"""
Task 5 Code
"""
import numpy as np
from matplotlib import pyplot as plt
from common import save_img, read_img
from homography import fit_homography, homography_transform
import os
import cv2


def make_synthetic_view(img, corners, size):
    '''
    Creates an image with a synthetic view of selected region in the image
    from the front. The region is bounded by a quadrilateral denoted by the
    corners array. The size array defines the size of the final image.

    Input - img: image file of shape (H,W,3)
            corner: array containing corners of the book cover in
```

```
            the order [top-left, top-right, bottom-right, bottom-left]
    (4,2)
            size: array containing size of book cover in inches [height,
    width] (1,2)

    Output - A fronto-parallel view of selected pixels (the book as if the
    cover is
            parallel to the image plane), using 100 pixels per inch.
    '''
    # The desired coordinates for the book corners
    h, w = size
    # Convert from inches to pixels: 1 inch is 100 pixels
    h, w = h * 100, w * 100
    dst_points = np.array([[0, 0], [w - 1, 0], [w - 1, h - 1], [0, h -
    1]], dtype='float32')
    XY = np.hstack((corners, dst_points))

    # Compute the homography matrix
    h_matrix = fit_homography(XY)

    # Perform the warp perspective
    warped_image = cv2.warpPerspective(img, h_matrix, (int(w), int(h)))
    return warped_image

if __name__ == "__main__":
    # Task 5

    case_name = "threebody"

    I = read_img(os.path.join("task5",case_name,"book.jpg"))
    corners = np.load(os.path.join("task5",case_name,"corners.npy"))
    size = np.load(os.path.join("task5",case_name,"size.npy"))
#     import pdb; pdb.set_trace()

    result = make_synthetic_view(I, corners, tuple(size[0]))
    save_img(result, case_name+"_frontoparallel.jpg")


    case_name = "palmer"

    I = read_img(os.path.join("task5",case_name,"book.jpg"))
    corners = np.load(os.path.join("task5",case_name,"corners.npy"))
    size = np.load(os.path.join("task5",case_name,"size.npy"))
#     import pdb; pdb.set_trace()

    result = make_synthetic_view(I, corners, tuple(size[0]))
    save_img(result, case_name+"_frontoparallel.jpg")
```

2.1.2 Result

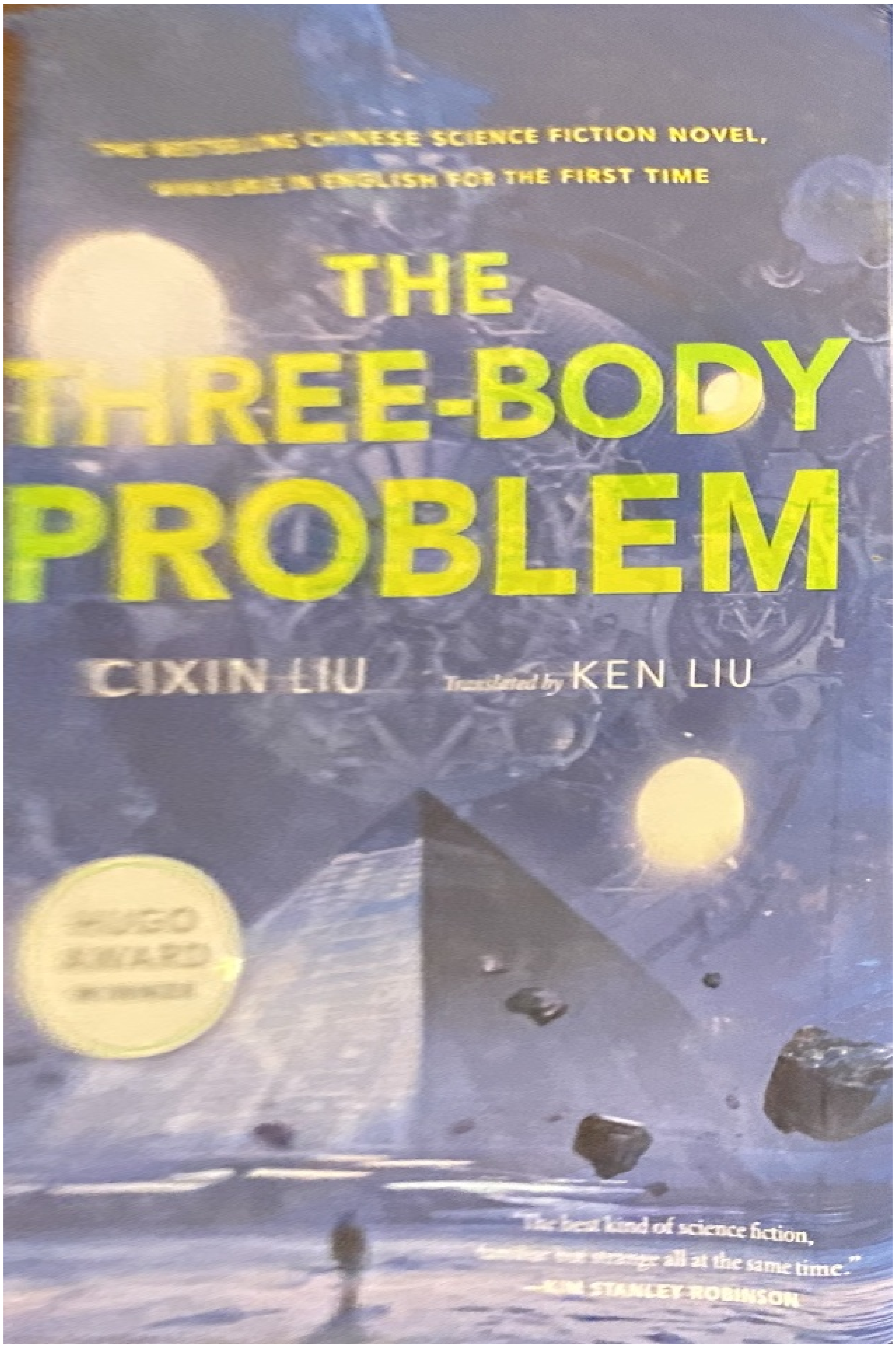

Figure 5: Warped threebody

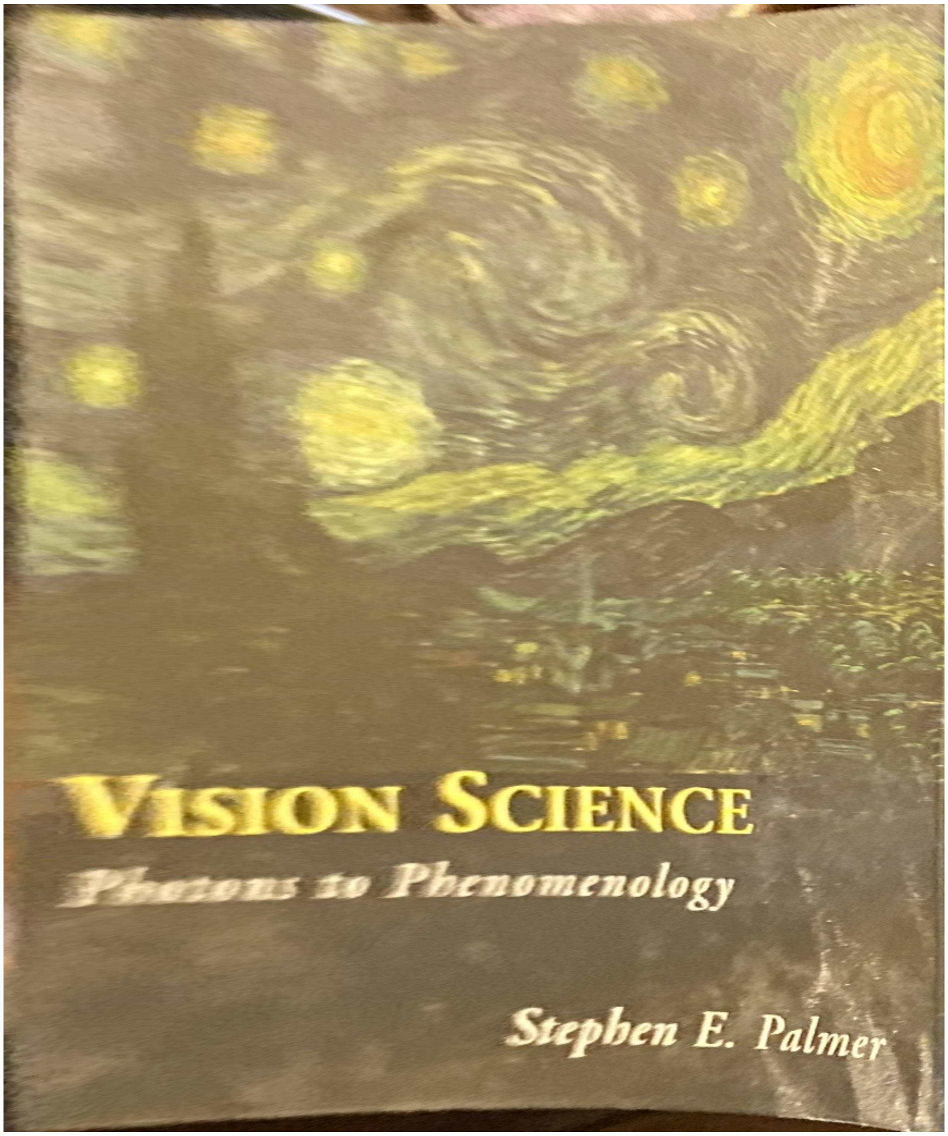

Figure 6: Warped palmer

2.1.3 Discussion

The Vision Science (palmer) book's top can bottom edge is curved after the warp action. This is because the surface of this book is slightly concave, therefore the photo toke from side view will cause the edge between the corners are curved as well. The homography (projection transformation) preserve lines inherently, so these curves are kept.

2.1.4 Optional

If the synthetic cover contains only ones, it would act as a white mask. When we apply the inverse warp using the homography that maps the synthetic cover onto the scene image, this white mask would be positioned over the area where the book cover appears in the scene image, making it look like the book cover is entirely white.

2.2 Task6: Stitching Stuff Together

2.2.1 Fill in compute_distance in task6.py

```python
"""
Task6 Code
"""
import numpy as np
import common
from common import save_img, read_img
from homography import fit_homography, homography_transform,
    RANSAC_fit_homography
import os
import cv2

def compute_distance(desc1, desc2):
    '''
    Calculates L2 distance between 2 binary descriptor vectors.

    Input - desc1: Descriptor vector of shape (N,F)
            desc2: Descriptor vector of shape (M,F)

    Output - dist: a (N,M) L2 distance matrix where dist(i,j)
             is the squared Euclidean distance between row i of
             desc1 and desc2. You may want to use the distance
             calculation trick
             ||x - y||^2 = ||x||^2 + ||y||^2 - 2x^T y
    '''
    X = desc1
    Y = desc2

    X_norm_sq = np.linalg.norm(X, axis=1, keepdims=True) ** 2
    Y_norm_sq = np.linalg.norm(Y, axis=1, keepdims=True) ** 2
    dist = np.sqrt(np.maximum(0, (X_norm_sq + Y_norm_sq.T - 2 * (X @ Y.T))
    ))
    return dist
```

2.2.2 Fill in find_matches in task6.py

```python
def find_matches(desc1, desc2, ratioThreshold):
    '''
    Calculates the matches between the two sets of keypoint
    descriptors based on distance and ratio test.

    Input - desc1: Descriptor vector of shape (N,F)
            desc2: Descriptor vector of shape (M,F)
            ratioThreshhold : maximum acceptable distance ratio between 2
                              nearest matches

    Output - matches: a list of indices (i,j) 1 <= i <= N, 1 <= j <= M
    giving
             the matches between desc1 and desc2.

```

```
            This should be of size (K,2) where K is the number of
            matches and the row [ii,jj] should appear if desc1[ii,:] and
            desc2[jj,:] match.
    '''
    matches = []

    dist = compute_distance(desc1, desc2)
    idx_smallest_two = np.argsort(dist, axis=1)[:, :2]
    ratio = np.take_along_axis(dist, idx_smallest_two, axis=1)[:, 0] / np.take_along_axis(dist, idx_smallest_two, axis=1)[:, 1]

    idx_ii = np.where((ratio < ratioThreshold))[0]
    idx_jj = idx_smallest_two[idx_ii, 0]
    matches = np.hstack((idx_ii[:, np.newaxis], idx_jj[:, np.newaxis]))
    # import pdb; pdb.set_trace()
    return matches
```

2.2.3 Fill in draw_matches in task6.py

```
def draw_matches(img1, img2, kp1, kp2, matches):
    '''
    Creates an output image where the two source images stacked vertically
    connecting matching keypoints with a line.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 2 of shape (H2,W2,3)
            kp1: Keypoint matrix for image 1 of shape (N,4)
            kp2: Keypoint matrix for image 2 of shape (M,4)
            matches: List of matching pairs indices between the 2 sets of
                     keypoints (K,2)

    Output - Image where 2 input images stacked vertically with lines joining
             the matched keypoints
    Hint: see cv2.line
    '''
    #Hint:
    #Use common.get_match_points() to extract keypoint locations
    output = np.vstack((img1, img2))
    H1, W1, _ = img1.shape
    kps = common.get_match_points(kp1, kp2, matches)
    for i in range(kps.shape[0]):
        p1 = kps[i, :2].astype(int)
        p2 = (kps[i, 2:] + np.array([0, H1])).astype(int)
        # print(p1, p2)
        cv2.line(output, (p1), (p2), (0, 0, 255), 4 )
    return output
```

2.2.4 Picture of matches

Figure 7: Matches of Lowetag

2.2.5 Fill in RANSAC_fit_homography in homography.py

```python
def RANSAC_fit_homography(XY, eps=1, nIters=1000):
    '''
```

```
    Perform RANSAC to find the homography transformation
    matrix which has the most inliers

    Input - XY: an array with size(N,4), each row contains two
            points in the form [x_i, y_i, x'_i, y'_i] (1,4)
            eps: threshold distance for inlier calculation
            nIters: number of iteration for running RANSAC
    Output - bestH: a (3,3) homography matrix fit to the
                    inliers from the best model.

    Hints:
    a) Sample without replacement. Otherwise you risk picking a set of points
       that have a duplicate.
    b) *Re-fit* the homography after you have found the best inliers
    '''
#     bestH, bestCount, bestInliers = np.eye(3), -1, np.zeros((XY.shape[0],))
#     bestRefit = np.eye(3)

    # Initialize the best homography matrix, inlier count and inlier set
    bestH = None
    bestCount = -1
    bestInliers = None

    for _ in range(nIters):
        # Step 1: Randomly select 4 pairs of points without replacement
        indices = np.random.choice(XY.shape[0], 4, replace=False)
        sample = XY[indices]

        # Step 2: Compute the homography matrix using the provided utility function
        H = fit_homography(sample)

        # Step 3: Apply homography and determine inliers
        # Transform source points to destination plane
        homogenized_src_pts = np.concatenate((XY[:, :2], np.ones((XY.shape[0], 1))), axis=1)
        transformed_pts = np.dot(H, homogenized_src_pts.T).T
        transformed_pts /= transformed_pts[:, 2][:, np.newaxis]  # Normalize

        # Calculate distances from actual to projected points
        homogenized_dst_pts = np.concatenate((XY[:, 2:], np.ones((XY.shape[0], 1))), axis=1)
        distances = np.linalg.norm(homogenized_dst_pts[:, :2] - transformed_pts[:, :2], axis=1)

        # Inliers are points with distance less than epsilon
        inliers = distances < eps
        inlier_count = np.sum(inliers)

        # Step 4: Keep track of the best homography with the most inliers
        if inlier_count > bestCount:
```

```
            bestCount = inlier_count
            bestH = H
            bestInliers = inliers

    # Step 5: Re-fit the homography using all inliers from the best model found
    if bestInliers is not None and bestCount > 4:  # More than the minimal sample size
        all_inliers = XY[bestInliers]
        bestH = fit_homography(all_inliers)
    else:
        bestH = np.eye(3)  # Fallback to identity matrix if no good model is found

    return bestH
```

2.2.6 Fill in make_warped and warp_and_combine in task6.py

```
def warp_and_combine(img1, img2, H):
    '''
    You may want to write a function that merges the two images together given
    the two images and a homography: once you have the homography you do not
    need the correspondences; you just need the homography.
    Writing a function like this is entirely optional, but may reduce the chance
    of having a bug where your homography estimation and warping code have odd
    interactions.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 2 of shape (H2,W2,3)
            H: homography mapping betwen them
    Output - V: stitched image of size (?,?,3); unknown since it depends on H
    '''
    # Get dimensions of input images
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    # Corners of img1
    corners_img1 = np.array([[0, 0], [0, h1], [w1, h1], [w1, 0]], dtype=np.float32).reshape(-1, 1, 2)

    # Corners of img2 transformed by H
    corners_img2 = np.array([[0, 0], [0, h2], [w2, h2], [w2, 0]], dtype=np.float32).reshape(-1, 1, 2)
    corners_img2_transformed = cv2.perspectiveTransform(corners_img2, H)

    # Combine the corners
    all_corners = np.concatenate((corners_img1, corners_img2_transformed), axis=0)
```

```python

    # Find the bounding rectangle
    x_min, y_min = np.intp(np.min(all_corners, axis=0).ravel() - 0.5)
    x_max, y_max = np.intp(np.max(all_corners, axis=0).ravel() + 0.5)

    # Translation homography
    translation_dist = [-x_min, -y_min]
    H_translation = np.array([[1, 0, translation_dist[0]],
                              [0, 1, translation_dist[1]],
                              [0, 0, 1]], dtype=np.float32)

    # Warp both images
    warp_img1 = cv2.warpPerspective(img1, H_translation, (x_max - x_min,
y_max - y_min))
    warp_img2 = cv2.warpPerspective(img2, H_translation.dot(H.astype(np.
float32)), (x_max - x_min, y_max - y_min))

    # Create a mask of the combined size for where img1 and warped img2
are not zero
    mask_img1 = np.sum(warp_img1, axis=2) > 0
    mask_img2 = np.sum(warp_img2, axis=2) > 0
    mask_overlap = mask_img1 & mask_img2
    mask_img1_only = mask_img1 & ~mask_overlap
    mask_img2_only = mask_img2 & ~mask_overlap

    # Initialize the stitched image canvas
    stitched_img = np.zeros_like(warp_img1)

    # Place each image on the canvas according to the masks
    stitched_img[mask_img1_only] = warp_img1[mask_img1_only]
    stitched_img[mask_img2_only] = warp_img2[mask_img2_only]

    # Handle overlapping areas
    stitched_img[mask_overlap] = warp_img1[mask_overlap] // 2 + warp_img2[
mask_overlap] // 2

    return stitched_img
```

```python
def make_warped(img1, img2):
    '''
    Take two images and return an image, putting together the full
pipeline.
    You should return an image of the panorama put together.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 1 of shape (H2,W2,3)

    Output - Final stitched image
    Be careful about:
    a) The final image size
    b) Writing code so that you first estimate H and then merge images
with H.
    The system can fail to work due to either failing to find the
homography or
```

```
    failing to merge things correctly.
    '''

    kp1, desc1 = common.get_AKAZE(I1)
    kp2, desc2 = common.get_AKAZE(I2)

    ratio = 0.7
    matches = find_matches(desc1, desc2, ratio)
    kps = common.get_match_points(kp1, kp2, matches)

    H = RANSAC_fit_homography(kps, eps= 4, nIters=2000)
    print(H)

    stitched = warp_and_combine(img2, img1, H)

    return stitched
```

2.2.7 Two panorama figures

Figure 8: Lowetag Panorama

Figure 9: Eynsham Panorama

2.2.8 Include Figures in .zip

Above two figures are submitted in Canvas

3 Augmented Reality on a Budget

3.1 Task7: Augmented Reality on a Budget

3.1.1 Fill in the function improve_image(scene,template,transfer) in task7.py

```
"""
Task 7 Code
"""
import numpy as np
import common
from common import save_img, read_img
from homography import homography_transform, RANSAC_fit_homography
import cv2
import os

from task6 import *

def task7_warp_and_combine(img1, img2, H):
    '''
    You may want to write a function that merges the two images together given
    the two images and a homography: once you have the homography you do not
    need the correspondences; you just need the homography.
```

```
    Writing a function like this is entirely optional, but may reduce the chance
    of having a bug where your homography estimation and warping code have odd
    interactions.

    Input - img1: Input image 1 of shape (H1,W1,3)
            img2: Input image 2 of shape (H2,W2,3)
            H: homography mapping betwen them
    Output - V: stitched image of size (?,?,3); unknown since it depends on H
                but make sure in V, for pixels covered by both img1 and warped img2,
                you see only img2
    '''
    # Warp img2 onto img1's plane
    warp_img2 = cv2.warpPerspective(img2, H, (img1.shape[1], img1.shape[0]))
    # Create mask of where the warped image is non-zero
    mask = (warp_img2.sum(-1) > 0)
    # Initialize output image
    V = img1.copy()
    # Place img2 on the masked regions of img1
    V[mask] = warp_img2[mask]

    return V

def improve_image(scene, template, transfer):
    '''
    Detect template image in the scene image and replace it with transfer image.

    Input - scene: image (H,W,3)
            template: image (K,K,3)
            transfer: image (L,L,3)
    Output - augment: the image with

    Hints:
    a) You may assume that the template and transfer are both squares.
    b) This will work better if you find a nearest neighbor for every template
       keypoint as opposed to the opposite, but be careful about directions of the
       estimated homography and warping!
    '''
    # augment = None
    # Resize transfer image to the template's size
    transfer = cv2.resize(transfer, (template.shape[1], template.shape[0]))

    kp1, desc1 = common.get_AKAZE(template)
    kp2, desc2 = common.get_AKAZE(scene)

    ratio = 0.7
```

```
    matches = find_matches(desc1, desc2, ratio)
    kps = common.get_match_points(kp1, kp2, matches)

    H = RANSAC_fit_homography(kps, eps= 4, nIters=2000)

    augment = task7_warp_and_combine(scene, transfer, H)

    return augment

if __name__ == "__main__":
    # Task 7
    scene_img_path = 'task7/scenes/lacroix/scene.jpg'
    template_img_path = 'task7/scenes/lacroix/template.png'
    transfer_img_path = 'task7/seals/monk.png'
    # scene_img_path = 'task7/scenes/bbb/scene.jpg'
    # template_img_path = 'task7/scenes/bbb/template.png'
    # transfer_img_path = 'task7/seals/um.png'

    scene = read_img(scene_img_path)
    template = read_img(template_img_path)
    transfer = read_img(transfer_img_path)

    improved_image = improve_image(scene, template, transfer)

    save_img(improved_image, f'improved_lacroix.jpg' )
```

3.1.2 Result

Figure 10: BBB Scene

Figure 11: BBB Template

Figure 12: UM logo To Transfer

Figure 13: Augmented BBB

Submitted by Wensong Hu on March 3, 2024.