Funiba Neba CV HW 3 Q1 #1-3

1 20 Transformations

i) $(a,b) \rightarrow$ homogeneous $\rightarrow \begin{bmatrix} a \\ b \\ 1 \end{bmatrix}$

$\rightarrow$ translation to origin $\rightarrow \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix} \rightarrow T_o$

$\rightarrow$ rotation of angle θ about origin $\rightarrow \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow R$

$\rightarrow$ translation back to $(a,b) \rightarrow \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \rightarrow T_p$

Full transformation $= T_o R T_p = \begin{bmatrix} \cos\theta & -\sin\theta & -a\cos\theta + a + b\sin\theta \\ \sin\theta & \cos\theta & -a\sin\theta - b\cos\theta + b \\ 0 & 0 & 1 \end{bmatrix}$

ii) $p_1 = (1,1) \sim \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

$p_2 = (2,1) \rightarrow$ stays same since its about p_2, w/ $a = 2$, $b = 1$

$p_3 = (2,2) \rightarrow \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$

$p_4 = (1,2) \longrightarrow \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$

$T_o R T_p p_1 = \begin{bmatrix} \frac{4-\sqrt{2}}{2} \\ \frac{2-\sqrt{2}}{2} \\ 1 \end{bmatrix}$, $T_o R T_p p_3 = \begin{bmatrix} \frac{4-\sqrt{2}}{2} \\ \frac{2+\sqrt{2}}{2} \\ 1 \end{bmatrix}$

$T_o R T_p p_4 = \begin{bmatrix} 2-\sqrt{2} \\ 1 \\ 1 \end{bmatrix}$

iii) $x' = ax + by + t_x + \alpha x^2 + \beta y^2$

$y' = cx + dy + t_y + \gamma x^2 + \theta y^2$

$\Rightarrow$ for (x_i, y_i) to (x_i', y_i') you have equations

$x_i' = ax_i + by_i + t_x + \alpha x_i^2 + \beta y_i^2$

$y_i' = cx_i + dy_i + t_y + \gamma x_i^2 + \theta y_i^2$, now set up linear system $Ax = b$ where A :

You have parameters $(a, b, c, d, t_x, t_y, \alpha, \beta, \gamma, \theta)$. If you group by x' & $y' \Rightarrow (a, b, t_x, c, d, t_y)$, then you have the quadratic contributions after that.
Construct A:

$$A = \begin{bmatrix} a & b & t_x & c & d & t_y & x^2 & y^2 \\ x_i & y_i & 1 & 0 & 0 & 0 & x_i^2 & y_i^2 \\ 0 & 0 & 0 & x_i & y_i & 1 & x_i^2 & y_i^2 \\ & & & \vdots & & & & \end{bmatrix}$$

rows of form similar to row 1 correspond to x' equation & rows of form similar to row 2 correspond to y' equation. And obviously b in this system is

$$b = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ \vdots \\ x_n' \\ y_n' \end{bmatrix}$$

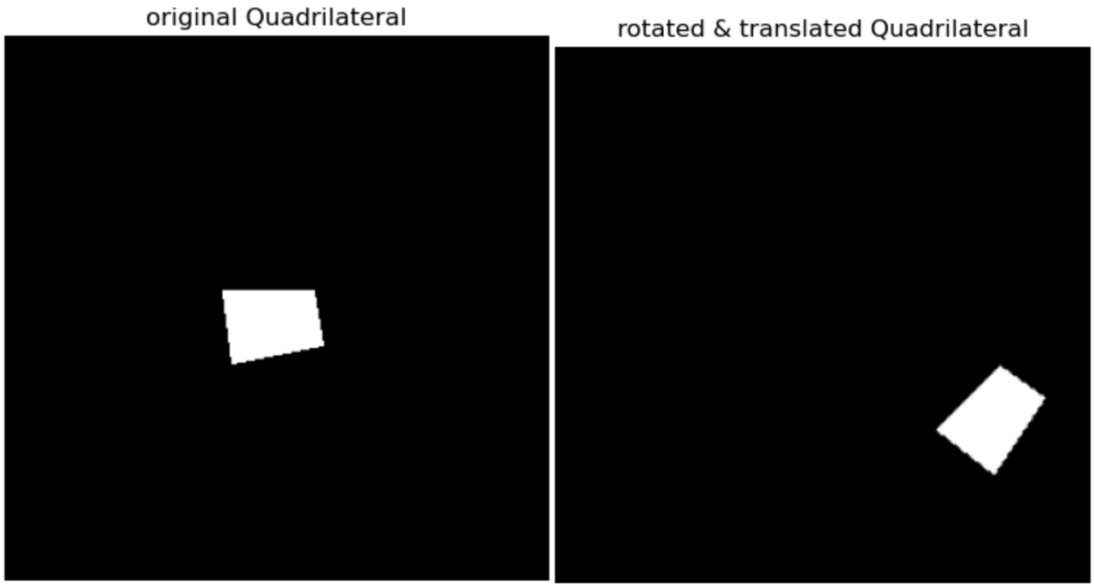
then x here is the collection of points. You'd need at least 10 equations to solve for each of the 10 parameters & since each point gives 2 equations, you'd need at minimum 5 points.

2D Transformations (part 4)

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Source for translation and rotation commands
(https://learnopencv.com/image-rotation-and-translation-using-opencv/)
def transform(image):

    # translate the image by (30, 100)
    M_1 = np.float32([[1, 0, 30], [0, 1, 100]])
    translated = cv2.warpAffine(image, M_1, (300, 300))

    # rotate the image by 45 degrees about the center
    M_2 = cv2.getRotationMatrix2D((150, 150), 45, 1)
    rotated = cv2.warpAffine(translated, M_2, (300, 300))

    plt.imshow(cv2.cvtColor(rotated, cv2.COLOR_BGR2RGB))
    plt.title('rotated & translated Quadrilateral')
    plt.axis('off')
    plt.show()

# Make the 300x300 image
image = np.zeros((300,300,3), dtype=np.uint8)

# Create a white irregular quadrilateral of 50x50 in size
corners = np.array([[120, 140], [170, 140], [175, 170], [125, 180]] ,
np.int32)

# fill the quadrilateral white
cv2.fillPoly(image, [corners], (255, 255, 255))

plt.imshow(cv2.fillPoly(image, [corners], (255, 255, 255)))
plt.title('original Quadrilateral')
plt.axis('off')
plt.show()

transform(image)
```

Output:



Writing your Own Image Stitching Algorithm

---------- ***computeH() function code:***

```
def computeH(im1_pts, im2_pts):
    M = []
    for i in range(len(im1_pts)):
        x, y = im1_pts[i][0], im1_pts[i][1]
        xp, yp = im2_pts[i][0], im2_pts[i][1]
        M.append([-x, -y, -1, 0, 0, 0, x*xp, y*xp, xp])
        M.append([0, 0, 0, -x, -y, -1, x*yp, y*yp, yp])

    M = np.array(M)
    U, s, Vt = np.linalg.svd(M)
    H = Vt[-1].reshape(3, 3)
    H = H / H[2, 2]
    return H
```

---------- ***ransac() function code:***

```
def ransac(corr, thresh, itera):
    max_inliers = []
    best_H = None
    for i in range(itera):
        idx = np.random.choice(len(corr), 4, replace=False)
        pic1_pts = np.float32([corr[i][0] for i in idx])
```

```
        pic2_pts = np.float32([corr[i][1] for i in idx])
        H = computeH(pic1_pts, pic2_pts)
        inliers = []
        for (pt1, pt2) in corr:
            homog_pt1 = np.append(pt1, 1)
            estimated_pt2 = np.dot(H, homog_pt1)
            estimated_pt2 /= estimated_pt2[2]
            dist = np.linalg.norm(estimated_pt2[:2] - pt2)
            if dist < thresh:
                inliers.append((pt1, pt2))
        if len(inliers) > len(max_inliers):
            max_inliers = inliers
            best_H = H
    return best_H
```

---------- estimateH() function code for estimating homography:

```
def estimateH(image1_path, image2_path):
    images = [cv2.imread(image1_path), cv2.imread(image2_path)]
    sift = cv2.SIFT_create()

    # Detect keypoints and compute descriptors for each image
    kp1, des1 = sift.detectAndCompute(images[0], None)
    kp2, des2 = sift.detectAndCompute(images[1], None)

    # Draw keypoints on images &  Display images with keypoints
    img1_kp = cv2.drawKeypoints(images[0], kp1, None)
    img2_kp = cv2.drawKeypoints(images[1], kp2, None)

    # match and sort descriptors
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)

    # Draw the top N matches
    N = 100
    img_matches_bf = cv2.drawMatches(images[0], kp1, images[1], kp2,
matches[:N], None)


    good_matches = [m for m in matches if m.distance < 300]
    src_points = np.float32([kp1[m.queryIdx].pt for m in
matches]).reshape(-1, 1, 2)
    dst_points = np.float32([kp2[m.trainIdx].pt for m in
matches]).reshape(-1, 1, 2)
    corr = np.concatenate((src_points, dst_points), axis=1)
```

```
    H = ransac(corr, thresh=5, itera=1000)

    return H, img1_kp, img2_kp, img_matches_bf
```

---------- ***stitching_2() code:***

```
def stitching2(img1, img2, H):
    img1 = cv2.imread(img1)
    img2 = cv2.imread(img2)
    # Note img_2 is the reference image here

    offset = 0.6
    extension = 200
    shift_y = 100

    # Max height and total width
    canvasH = max(img1.shape[0], img2.shape[0]) + extension
    canvasW = img1.shape[1] + img2.shape[1] + extension

    # Translation
    shift = np.array([[1, 0, offset*img2.shape[1]], [0, 1, shift_y],
[0, 0, 1]])
    adjusted_H = shift @ H

    # Warp img_1
    warped_img = cv2.warpPerspective(img1, adjusted_H, (canvasW,
canvasH)).astype(np.float32)

    # Create the blank canvas and put the reference image on it
    canvas = np.zeros((canvasH, canvasW, 3), dtype=np.float32)
    canvas[shift_y:img2.shape[0]+shift_y,
int(offset*img2.shape[1]):int((1+offset)*img2.shape[1])] = img2

    # Form final stitch by adding the warped image to the canvas
    final_stitch = canvas + warped_img

    # Normalize pixel values to the range [0, 255]
    final_stitch = np.clip(final_stitch, 0, 255).astype(np.uint8)

    return final_stitch
```

---------- ***stitching_3 code:***

```
def stitching3(img1, img2, img3, H1, H2):
    img1 = cv2.imread(img1)
    img2 = cv2.imread(img2)
    img3 = cv2.imread(img3)
    # Note img_2 is the reference image here

    offset = 0.6
    extension = 200
    shift_y = 100

    # Max height and total width
    canvasH = max(img1.shape[0], img2.shape[0]) + extension
    canvasW = img1.shape[1] + img2.shape[1] + extension

    # Translation for img 1
    shift = np.array([[1, 0, offset*img2.shape[1]], [0, 1, shift_y],
[0, 0, 1]])
    adjusted_H = shift @ H1

    # Warp img_1
    warped_img = cv2.warpPerspective(img1, adjusted_H, (canvasW,
canvasH)).astype(np.float32)

    # Translation for img3
    shift2 = np.array([[1, 0, offset*img2.shape[1]], [0, 1, shift_y],
[0, 0, 1]])
    adjusted_H3 = shift2 @ H2

    # Warp img_3
    warped_img2 = cv2.warpPerspective(img3, adjusted_H3, (canvasW,
canvasH)).astype(np.float32)

    # Create the blank canvas and put the reference image on it
    canvas = np.zeros((canvasH, canvasW, 3), dtype=np.float32)
    canvas[shift_y:img2.shape[0]+shift_y,
int(offset*img2.shape[1]):int((1+offset)*img2.shape[1])] = img2

    # Form final stitch by adding the warped image to the canvas
    final_stitch = canvas + warped_img + warped_img2

    # Normalize pixel values to the range [0, 255]
    final_stitch = np.clip(final_stitch, 0, 255).astype(np.uint8)
```

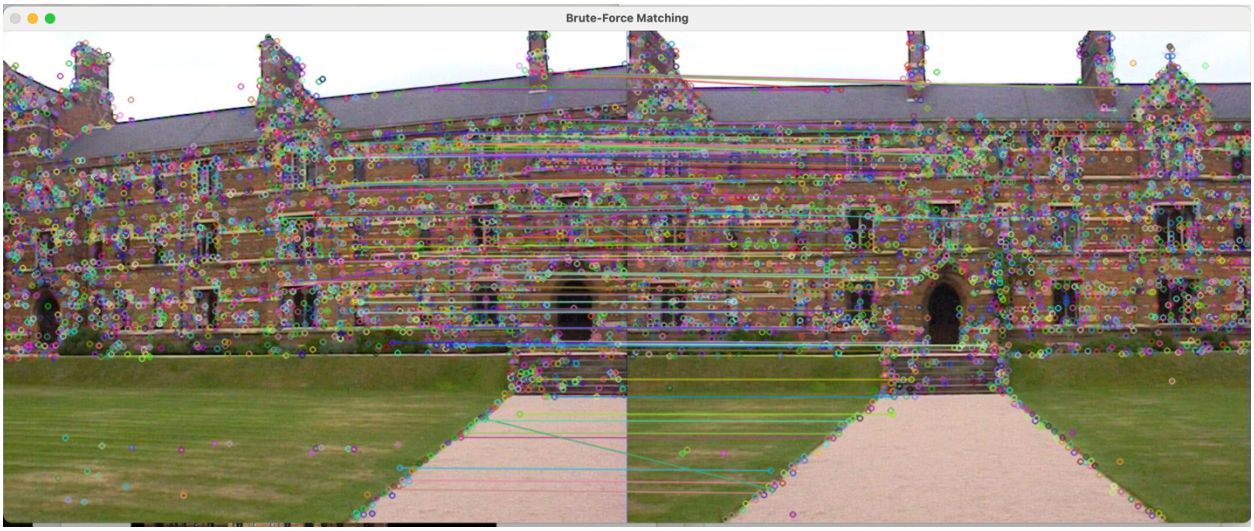
```
    return final_stitch
```

---------- Code utilizing all these functions for submission outputs:

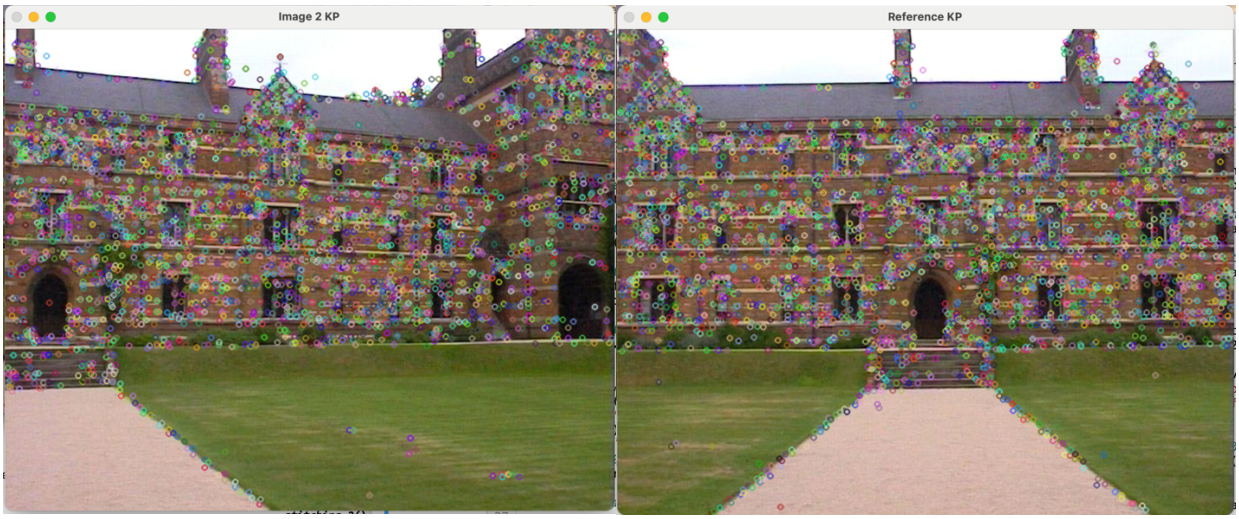
```
image1 = "/Users/fneba/Desktop/Hw3/keble_a.jpg"
image2 = "/Users/fneba/Desktop/Hw3/keble_c.jpg"
reference = "/Users/fneba/Desktop/Hw3/keble_b.jpg"

Homography, img1_kp, img2_kp, img_matches_bf = estimateH(image1,
reference)

Homography2, img1_kp2, img2_kp2, img_matches_bf2 = estimateH(image2,
reference)

# From image 1 and reference
cv2.imshow('Image 1 KP', img1_kp)
cv2.imshow('Reference KP', img2_kp)
cv2.imshow('Brute-Force Matching', img_matches_bf)

# From image 2 and reference
cv2.imshow('Image 2 KP', img1_kp2)
cv2.imshow('Reference KP', img2_kp2)
cv2.imshow('Brute-Force Matching', img_matches_bf2)

# Create the mosaic from two images and their homography
mosaic1 = stitching2(image1, reference, Homography)
mosaic2 = stitching2(image2, reference, Homography2)

# Show the resulting 2 image mosaic (image 1 and ref)
cv2.imshow("Mosaic1", mosaic1)
# Show the resulting 2 image mosaic (image 2 and ref)
cv2.imshow("Mosaic2", mosaic1)

# Create the panorama from 3 images and their homographies
panorama = stitching3(image1, reference, image2, Homography,
Homography2)

# Show the resulting 3 image mosaic
cv2.imshow("Panorama", panorama)
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.waitKey()
```

Outputs:

Raw matches & visualization of tentative matches & inliers for each pair (image 1 and reference):



Raw matches & visualization of tentative matches & inliers for each pair (image 2 and reference):



Output of stitching 2 images:

Final mosaic (or in this case, I called it panorama):

---------- Extra Credit Part 1 function for "cropping":

```
def crop_panorama(panorama):
    grayscale = cv2.cvtColor(panorama, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(grayscale, 1, 255, cv2.THRESH_BINARY)

    # Find spots where there are non-black pixels
    NB = np.argwhere(thresh > 0)
```

```
    # Get bounding box for this non-black area
    y1, x1 = NB.min(axis=0)
    y2, x2 = NB.max(axis=0) + 1

    # The actual cropping
    cropping = panorama[y1:y2, x1:x2]

    return cropping


panorama = crop_panorama(panorama)
cv2.imshow("Panorama", panorama)
```

Output from using code on the original panorama: