3DCV Hw1

R11944014 戴靖婷

tags: `3DCV` `Python` `Report`

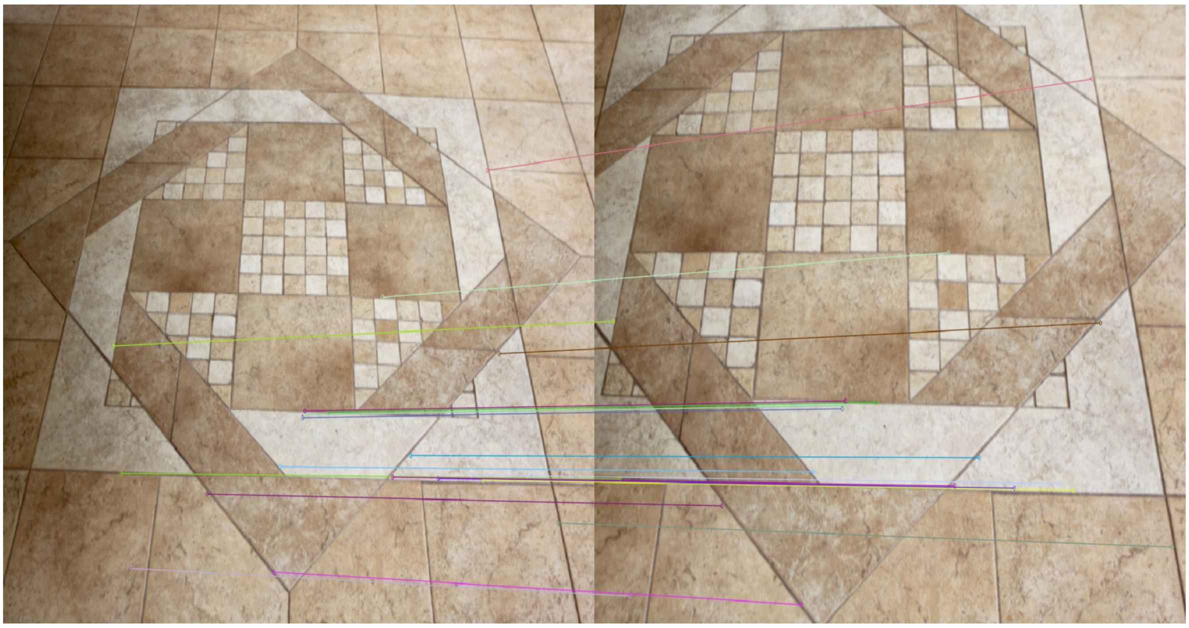
Problem 1: Homography estimation

- Settings:
  - Execute codes:
    `python .\1.py .\images\1-0.png .\images\1-[1or2].png .\groundtruth_correspondences\correspondence_0[1or2].npy [k_toppairs]`
    - ex: `python .\1.py .\images\1-0.png .\images\1-2.png .\groundtruth_correspondences\correspondence_02.npy 20`
  - Package:
    - sys, numpy, cv2, math
  - Environment:
    - python: 3.7.9
    - opencv-python == 4.5.1.48

- Screenshots of sampling k correspondences
  - img 1-0 and 1-1
    - k = 4

- k = 8

- k = 20

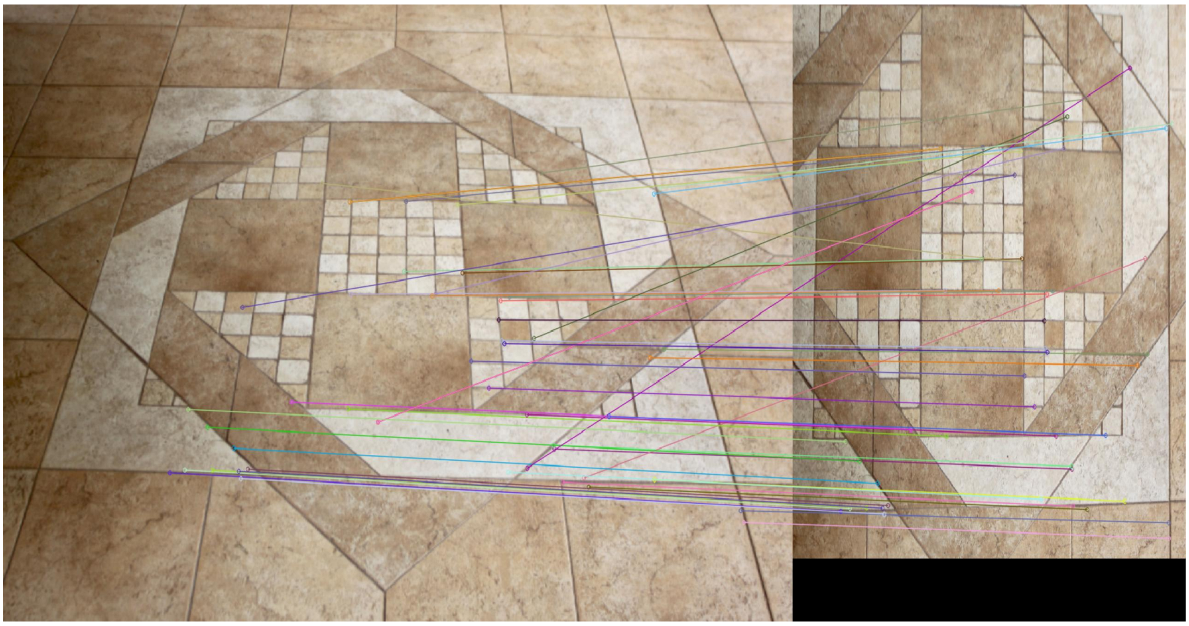


- img 1-0 and 1-2
  - k = 4

- k = 8

- k = 20

- k = 50



- Briefly explain code:
    - First of all, the function get_sift_correspondences acquires matched points of two images. In addition, with Lowe's ratio test, it rejects some outliers whose two best matches of a point is not sufficiently different. In addition, choose only the top k pairs of the correspondences as the matching points.

```python
def get_sift_correspondences(img1, img2, k_toppairs):
    '''
    Input:
        img1: numpy array of the first image
        img2: numpy array of the second image

    Return:
        points1: numpy array [N, 2], N is the number of correspondences
        points2: numpy array [N, 2], N is the number of correspondences
    '''

    # sift = cv.xfeatures2d.SIFT_create() # opencv-python and opencv-contrib-python version == 3.4.2.16 or enable nonfree
    sift = cv.SIFT_create()
    # find keypoints and descriptors
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    matcher = cv.BFMatcher() # BFMatcher: Brute-Force Matcher
    matches = matcher.knnMatch(des1, des2, k=2) # Match descriptors

    # Ratio test
    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)

    good_matches = sorted(good_matches, key=lambda x: x.distance)

    # Sample k pairs of correspondences
    good_matches = good_matches[:k_toppairs]

    points1 = np.array([kp1[m.queryIdx].pt for m in good_matches])
    points2 = np.array([kp2[m.trainIdx].pt for m in good_matches])

    img_draw_match = cv.drawMatches(img1, kp1, img2, kp2, good_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    cv.namedWindow('match', cv.WINDOW_NORMAL)
    cv.imshow('match', img_draw_match)
    cv.waitKey(0)

    return points1, points2
```

- Calculate homography matrix by Direct Linear Transform.
    - Build the matrix A by the k pairs of correspondences.
    - Acquire V from SVD of A and homography matrix is the last row of V after reshaped.

```python
    # Direct Linear Transform
    def DLT(k_points1, k_points2):
        """
        Input:
            points1: numpy array [k, 2], k is the number of top k pairs of correspondences
            points2: numpy array [k, 2], k is the number of top k pairs of correspondences

        Return:
            Homography matrix: 3 x 3 matrix
        """

        # Construct A matrix from point correspondences
        k_toppairs = len(k_points1)
        A = np.zeros([k_toppairs * 2, 9], dtype=float)
        for i in range(k_toppairs):
            A[i * 2, 3] = -k_points1[i, 0]
            A[i * 2, 4] = -k_points1[i, 1]
            A[i * 2, 5] = -1
            A[i * 2, 6] = k_points2[i, 1] * k_points1[i, 0]
            A[i * 2, 7] = k_points2[i, 1] * k_points1[i, 1]
            A[i * 2, 8] = k_points2[i, 1]

            A[i * 2 + 1, 0] = k_points1[i, 0]
            A[i * 2 + 1, 1] = k_points1[i, 1]
            A[i * 2 + 1, 2] = 1
            A[i * 2 + 1, 6] = -k_points2[i, 0] * k_points1[i, 0]
            A[i * 2 + 1, 7] = -k_points2[i, 0] * k_points1[i, 1]
            A[i * 2 + 1, 8] = -k_points2[i, 0]

        # SVD
        _, _, V = np.linalg.svd(A)
        h = V[-1]
        H = h.reshape(3, 3)
        H /= H[2, 2]

        return H
```

- Calculate normalized homography matrix by normalized DLT.
    - First compute similarity transform for both matching points that translates the centroid to the origin and scales the average distance from the origin is the root of 2.
    - Apply DLT on the normalized points to obtain a homography matrix.
    - Last, denormalize and get the true homography.

```python
    # Normalized Direct Linear Transform
    def norm_DLT(k_points1, k_points2):
        """
        Input:
            points1: numpy array [k, 2], k is the number of top k pairs of correspondences
            points2: numpy array [k, 2], k is the number of top k pairs of correspondences
            k_toppairs: k_toppairs

        Return:
            Homography matrix: 3 x 3 matrix
        """

        k_toppairs = len(k_points1)

        # Calculate mean of u, v and u', v'
        mu_u1 = k_points1[:, 0].sum() / k_toppairs
        mu_v1 = k_points1[:, 1].sum() / k_toppairs
        mu_u2 = k_points2[:, 0].sum() / k_toppairs
        mu_v2 = k_points2[:, 1].sum() / k_toppairs

        # Scalar quantities
        s1 = 0
        for pnt in k_points1:
            s1 += np.sqrt(np.square(pnt[0] - mu_u1) + np.square(pnt[1] - mu_v1))
        s1 /= math.sqrt(2) * k_toppairs
        s1 =  np.reciprocal(s1)

        s2 = 0
        for pnt in k_points2:
            s2 += np.sqrt(np.square(pnt[0] - mu_u2) + np.square(pnt[1] - mu_v2))
        s2 /= math.sqrt(2) * k_toppairs
        s2 =  np.reciprocal(s2)
```

```python
    # Calculate similarity transform T, T'
    T1 = np.zeros([3, 3], dtype=float)
    T1[0, 0] = s1
    T1[0, 2] = -s1 * mu_u1
    T1[1, 1] = s1
    T1[1, 2] = -s1 * mu_v1
    T1[2, 2] = 1

    T2 = np.zeros([3, 3], dtype=float)
    T2[0, 0] = s2
    T2[0, 2] = -s2 * mu_u2
    T2[1, 1] = s2
    T2[1, 2] = -s2 * mu_v2
    T2[2, 2] = 1

    # pi = [ui, vi, 1], norm_p = T dot p
    col_ones = np.ones(k_toppairs)
    p1 = np.column_stack((k_points1, col_ones))
    p2 = np.column_stack((k_points2, col_ones))

    norm_p1 = np.matmul(p1, T1.T)
    norm_p2 = np.matmul(p2, T2.T)

    # DLT of normalized points
    H = DLT(norm_p1, norm_p2)

    # Denormalize to get the normalized Homography matrix
    norm_H = np.matmul(np.linalg.inv(T2), np.matmul(H, T1))
    norm_H /= norm_H[2, 2]

    return norm_H
```

- Compute error by the L2 norm of the estimated target points and ground truth matching pairs.

```python
# Compute the reprojection error with the ground truth matching pairs
def compute_error(gt_correspondences, homography_mat):
    # p_s: 100 source points, p_t: 100 target points
    num_pts = len(gt_correspondences[0]) # 100
    p_s = gt_correspondences[0]
    p_t = gt_correspondences[1]

    col_ones = np.ones(num_pts)
    p_s = np.column_stack((p_s, col_ones))
    p_t = np.column_stack((p_t, col_ones))
    p_t_est = np.matmul(p_s, homography_mat.T)

    error = np.sum(np.linalg.norm((p_t[:, :2] - p_t_est[:, :2]), ord=2, axis=0)) / num_pts

    return error
```

- Compare the errors:
  - Error of img 1-1 with Lowe's ratio = 0.75

img 1-1	k = 4	k = 8	k = 20	k = 1467
DLT	74.5550	7.3490	7.7314	791.5816
Norm DLT	74.5550	7.3710	7.7348	11.6377

  - Error of img 1-2 with Lowe's ratio = 0.75

img 1-2	k = 4	k = 8	k = 20	k = 50	k = 88
DLT	75.2951	102.7948	102.8110	112.1455	122.6268
Norm DLT	75.2951	101.5104	94.4720	65.2852	36.0418

- Discussion
  - The error between DLT and normalized DLT is not obvious in img 1-1, but it is significant in img 1-2. It is because that the viewpoint of 1-2 is much different than 1-1. It might be harder to get the correct SIFT correspondences in img 1-2.

- When setting k top points to all correspondences, the error is significantly different between DLT and normalized DLT. In addition, the errors of normalized DLT in img 1-2 are all smaller with various k. It might be caused by the advantage of normalization, which reduces the problem of ill-conditioning number and finds a better result of the homography matrix by solving SVD.

Problem 2: Document rectification

- Settings:
  - Execute codes:
    `python .\2.py .\images\2-1 [path to save the warped result]`
    - if prefer to save the warped image, add saving path at the end`
    - ex: `python .\2.py .\images\2-1 .\images\warped_img.png`
  - Package:
    - sys, numpy, cv2
  - Environment:
    - python: 3.7.9
    - opencv-python == 4.5.1.48
- The input document image (must be captured by yourself)

- Rectified results

- Briefly explain your method (how you choose the corners, warping efficiency)
    - I make use of the function in mouse_click_example.py to manually get four corners by the user (order: top-left, top-right, bottom-left, bottom-right). Also, it will automatically end the window after selecting four points.

```
def on_mouse(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        param[0].append([x, y])

# Use mouse to get 4 corner points of the image
def get_corners(img):
    points_add= []
    cv.namedWindow('get_corners', cv.WINDOW_NORMAL)
    cv.setMouseCallback('get_corners', on_mouse, [points_add])

    # Break when it gets 4 points
    while len(points_add) < 4:
        img_ = img.copy()
        for i, p in enumerate(points_add):
            # draw points on img_
            cv.circle(img_, tuple(p), 2, (0, 255, 0), -1)
        cv.imshow('get_corners', img_)

        key = cv.waitKey(20) % 0xFF
        if key == 27: break # exist when pressing ESC

    cv.destroyAllWindows()

    print('{} corner points added'.format(len(points_add)))
    print(points_add)

    return np.array(points_add)
```

    - Set the projected image size half of the original image. Then, calculate the homography matrix by Direct Linear Transform (the result is as good as the one of normalized DLT).

```
    # left-top, right-top, left-bottom, right-bottom
    corners = get_corners(img)
    #corners = np.array([[696, 1270], [2951, 294], [1395, 2747], [3556, 1841]], dtype=float)

    proj_corners = np.array([[0, 0],
                            [img_w - 1, 0],
                            [0, img_h - 1],
                            [img_w - 1, img_h - 1]])

    H = DLT(corners, proj_corners)
```

    - Warp the image by the function, back_warping with the parameters of original image and homography matrix.

- At first, I take the inverse of the homography matrix for inverse mapping from destination pixels to original pixel positions.

```python
# Inverse/Backward warping
def backward_warping(ori_img, homography_mat):
    """
    Input:
        ori_img: original image
        homography_mat: 3 x 3 matrix

    Return:
        warp_img: the warped image after doing backward warping
    """

    inv_homography_mat = np.linalg.inv(homography_mat)
    img_h, img_w, _ = ori_img.shape
    img_h, img_w = int(img_h/2), int(img_w/2)
    warp_img = np.empty((img_h, img_w), dtype=np.uint8)
```

- Then, construct matrix dst_coordinates (wi, hi, 1), which size is # of pixels x 3. And warp all of those coordinates back to original coordinates.

```python
    # Construct matrix dst_coordinates [wi hi 1], size = (img_w * img_h, 3)
    # ex: image size = 2 x 3
    #     | 0 0 1 |
    #     | 1 0 1 |
    #     | 2 0 1 |
    #     | 0 1 1 |
    #     | 1 1 1 |
    #     | 2 1 1 |

    w_arr = np.arange(img_w).reshape((img_w, 1))
    h_arr = np.zeros((img_w, 1))
    dst_coordinates = np.hstack((w_arr, h_arr))

    for i in range(1, img_h):
        h_arr = np.ones((img_w, 1)) * i
        dst_coordinates = np.vstack((dst_coordinates, np.hstack((w_arr, h_arr))))

    one_arr = np.ones((len(dst_coordinates), 1))
    dst_coordinates = np.hstack((dst_coordinates, one_arr))
```

- Calculate RGB values by the four nearest points of the original image.

```python
    # Inverse warping to source coordinates
    src_coordinates = np.matmul(dst_coordinates, inv_homography_mat.T)
    src_coordinates = src_coordinates.T / src_coordinates[:, 2]
    src_w_arr, src_h_arr = src_coordinates[0], src_coordinates[1]

    # Calculate values of each pixel
    h_l_arr, h_r_arr = np.floor(src_h_arr).astype(int), np.ceil(src_h_arr).astype(int)
    w_l_arr, w_r_arr = np.floor(src_w_arr).astype(int), np.ceil(src_w_arr).astype(int)

    warp_img = (ori_img[h_l_arr, w_l_arr, :] * (src_h_arr - h_l_arr).reshape(-1, 1) * (src_w_arr - w_l_arr).reshape(-1, 1)
                + ori_img[h_l_arr, w_r_arr, :] * (src_h_arr - h_l_arr).reshape(-1, 1) * (w_r_arr - src_w_arr).reshape(-1, 1)
                + ori_img[h_r_arr, w_l_arr, :] * (h_r_arr - src_h_arr).reshape(-1, 1) * (src_w_arr - w_l_arr).reshape(-1, 1)
                + ori_img[h_r_arr, w_r_arr, :] * (h_r_arr - src_h_arr).reshape(-1, 1) * (w_r_arr - src_w_arr).reshape(-1, 1)).reshape((img_h, img_w, 3)).astype(np.uint8)

    return warp_img
```

- Discussion
  - My warped result was nearly blanked at first, and I figured out I should set the type to uint32.
  - Besides, I used two for loops to calculate values of each pixel which was super slow. Therefore, I improved the computational time by constructing a matrix with destination pixel values and multiplying it with inverse homogrphy matrix.