Computer Vision HW3 Report

Student ID: B10901151
Name: 林祐群

Part 1.

- **Paste your warped canvas**

Part 2.

● Paste the function code *solve_homography(u, v)* & *warping( )* (both forward & backward)

solve_homography(u, v):

```python
def solve_homography(u, v):
    """
    Compute the Homography matrix H, such that v ≈ H * u.
    This function uses the Direct Linear Transformation (DLT) method, and fixes h33 = 1.

    Parameters:
        u: numpy array of shape (N x 2), each row represents a point [u_x, u_y] in the source image
        v: numpy array of shape (N x 2), each row represents the corresponding point [v_x, v_y] in the target image
    Returns:
        H: 3x3 Homography matrix, such that v ≈ H * u (using homogeneous coordinates)
    """
    N = u.shape[0]
    H = None

    if v.shape[0] is not N:
        print('u and v should have the same size')
        return None
    if N < 4:
        print('At least 4 points should be given')

    u_x = u[:, 0]
    u_y = u[:, 1]
    v_x = v[:, 0]
    v_y = v[:, 1]
    ones_N = np.ones(N)

    # (1) h11*u_x + h12*u_y + h13 - v_x*(h31*u_x + h32*u_y + 1) = 0
    # (2) h21*u_x + h22*u_y + h23 - v_y*(h31*u_x + h32*u_y + 1) = 0
    # Arrange the unknowns as x = [h11,h12,h13, h21,h22,h23, h31,h32], and fix h33 = 1.
    # The equations can be written as A*x = b, where A is a (2N x 8) matrix and b is a (2N,) vector.
    A = np.zeros((2 * N, 8))
    b_vec = np.zeros(2 * N)
```

```python
    # Odd rows (corresponding to v_x equation)
    A[0::2, 0] = u_x
    A[0::2, 1] = u_y
    A[0::2, 2] = ones_N
    A[0::2, 6] = -v_x * u_x
    A[0::2, 7] = -v_x * u_y
    b_vec[0::2] = v_x

    # Even rows (corresponding to v_y equation)
    A[1::2, 3] = u_x
    A[1::2, 4] = u_y
    A[1::2, 5] = ones_N
    A[1::2, 6] = -v_y * u_x
    A[1::2, 7] = -v_y * u_y
    b_vec[1::2] = v_y

    # Solve the least squares problem Ax = b
    x, residuals, rank, s = np.linalg.lstsq(A, b_vec, rcond=None)
    # x contains: [h11, h12, h13, h21, h22, h23, h31, h32]

    # solve H with A
    H = np.array([[x[0], x[1], x[2]],
                  [x[3], x[4], x[5]],
                  [x[6], x[7], 1.0]])

    return H
```

warping():

```python
def warping(src, dst, H, ymin, ymax, xmin, xmax, direction='b'):
    h_src, w_src, ch = src.shape
    h_dst, w_dst, ch = dst.shape
    H_inv = np.linalg.inv(H)
    xs = np.arange(xmin, xmax)
    ys = np.arange(ymin, ymax)
    X, Y = np.meshgrid(xs, ys, sparse = False)  # X, Y shape = (region_h, region_w)
    ones_arr = np.ones(X.size)
    if direction == 'b':
        coords_dst = np.vstack((X.ravel(), Y.ravel(), ones_arr))  # (3, N)
        coords_src = H_inv @ coords_dst
        # Apply H_inv to the destination pixels and retrieve (u,v) pixels, then reshape to (ymax-ymin),(xmax-xmin)
        coords_src /= coords_src[2, :]  # Homogeneous coordinate normalization
        X_src = coords_src[0, :]
        Y_src = coords_src[1, :]
        valid_mask = (X_src >= 0) & (X_src < w_src - 1) & (Y_src >= 0) & (Y_src < h_src - 1)
        region_h, region_w = Y.shape
        warped_region = np.zeros((region_h * region_w, ch), dtype=src.dtype)
        valid_indices = np.where(valid_mask)[0]
        if valid_indices.size > 0:
            X_src_valid = X_src[valid_mask]
            Y_src_valid = Y_src[valid_mask]
            x0 = np.round(X_src_valid).astype(np.int32)
            y0 = np.round(Y_src_valid).astype(np.int32)

            I = src[y0, x0, :]  # shape (N_valid, ch)

            warped_region[valid_indices] = I
        region_h, region_w = Y.shape
        warped_img = warped_region.reshape((region_h, region_w, ch))
        dst_region = dst[ymin:ymax, xmin:xmax]
        valid_mask_reshaped = valid_mask.reshape((region_h, region_w))
        dst_region[valid_mask_reshaped] = warped_img[valid_mask_reshaped]
        dst[ymin:ymax, xmin:xmax] = dst_region

    elif direction == 'f':
```

```python
        # Forward warp: Map points in the source image to the destination position (using nearest neighbor interpolation)
        coords_src = np.vstack((X.ravel(), Y.ravel(), ones_arr))
        coords_dst = H @ coords_src
        coords_dst /= coords_dst[2, :]
        X_dst = coords_dst[0, :]
        Y_dst = coords_dst[1, :]
        X_dst_round = np.round(X_dst).astype(np.int32)
        Y_dst_round = np.round(Y_dst).astype(np.int32)
        valid_mask = (X_dst_round >= 0) & (X_dst_round < w_dst) & (Y_dst_round >= 0) & (Y_dst_round < h_dst)
        if np.any(valid_mask):
            X_src_valid = X.ravel()[valid_mask]
            Y_src_valid = Y.ravel()[valid_mask]
            X_dst_valid = X_dst_round[valid_mask]
            Y_dst_valid = Y_dst_round[valid_mask]
            dst[Y_dst_valid, X_dst_valid] = src[Y_src_valid, X_src_valid]

    return dst
```

- **Briefly introduce the interpolation method you use**

I use nearest neighbor interpolation in the warping function. When transforming pixels from one image to another, the function rounds the calculated coordinates to the nearest integer rather than using more sophisticated interpolation methods like bilinear or bicubic. This is because I found it difficult to get decent QRCode in part 3 by utilizing bilinear

interpolation, while the implementation using this naive approach performs significantly well. The implementation can be discovered in the following lines:

```
x0 = np.round(X_src_valid).astype(np.int32)
y0 = np.round(Y_src_valid).astype(np.int32)
```

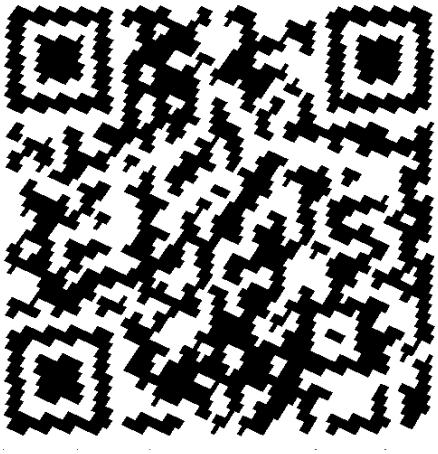
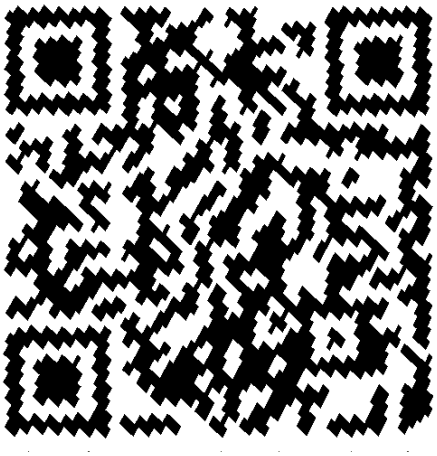
and also

```
X_dst_round = np.round(X_dst).astype(np.int32)
Y_dst_round = np.round(Y_dst).astype(np.int32)
```

Generally speaking, Nearest neighbor is computationally efficient but can result in blockier or less smooth results compared to higher-order interpolation methods.

Part 3.

- **Paste the 2 warped images and the link you find**

	Warped image	Link
output_3_1		http://media.ee.ntu.edu.tw/courses/cv/25S/
output_3_2		http://media.ee.ntu.edu.tw/courses/cv/25S/ (which is the same to the above one!)

● Discuss the difference between 2 source images, are the warped results the same or different?

In this task, I use the given four corner points to calculate the Homography, and then perform a backward warp on the distorted QR code region. Theoretically and practically as well, since the QR code is located on the same plane and the calibration of the four corresponding points is sufficiently accurate, the orthographic result calculated through Homography is consistent regardless of the source image's shooting angle. That is to say, the warped QR code is consistent. In this task and with my method, minor differences such as clarity, and slight distortions still lead to small differences between the two results, but the overall quality and clarity are almost the same.

● If the results are the same, explain why. If the results are different, explain why?

The result shows that the two images are quite similar to each others (same to each other). This results illustrate that the QR code is located on the same plane and the calibration of the four corresponding points is sufficiently accurate, the orthographic result calculated through Homography is consistent regardless of the source image's shooting angle. The final results after warping show that it’s possible to retrieve the QR code in desired canvas while the shooting angles are different as long as certain requirements are made.

Strictly speaking, the second image has slight higher spatial frequency (information) than the first image, I think this is because of the shooting angle, the size, and the shape of QR Code in source images. With these consideration, the two results still bear some difference; however, the transforming results still improve a lot and make them look more similar to each others.

Part 4.

- **Paste your stitched panorama**

- **Can all consecutive images be stitched into a panorama?**

In this work, it's fine to stitch all (three) consecutive images into a panorama. Though the connection place for different images still have discontinuity (especially in the place of the cables), the stitched panorama has great quality and global continuity which is consistent in human eye system.

However, not all consecutive images can be stitched into a panorama.

- **If yes, explain your reason. If not, explain under what conditions will result in a failure?**

The reason that not all consecutive images can be stitched into a panorama is because that as long as some conditions are met can the consecutive images be stitched into a panorama. Below are some conditions that need to be made:

1. Sufficient overlap between images (typically 30-50%)
2. Similar exposure and lighting conditions
3. Limited parallax effects (camera should rotate around its optical center)
4. Scene content must be mostly static
5. Similar focal lengths between images

And some common issues that prevent successful stitching, as a result:

- Moving objects causing ghosting artifacts
- Large differences in lighting/exposure
- Extreme perspective changes

- Insufficient distinct features for matching
- Heavy lens distortion

Thus, we know that even with advanced stitching algorithms, some image sets simply cannot be properly aligned due to these fundamental limitations.

Reference1: https://www.youtube.com/watch?v=5zzIAgw6z_U

Reference2: https://ptgui.com/man/projections.html