Part 1

```python
def compute_h(p1, p2):
    # TODO ...

    # number of pairs of corresponding interest points
    N = p1.shape[0]

    # construct A
    A = [-p2[0][1], -p2[0][0], -1, 0, 0, 0, p1[0][1]*p2[0][1], p1[0][1]*p2[0][0], p1[0][1]]
    for i in range(N):
        arr1 = [-p2[i][1], -p2[i][0], -1, 0, 0, 0, p1[i][1]*p2[i][1], p1[i][1]*p2[i][0], p1[i][1]]
        arr2 = [0, 0, 0, -p2[i][1], -p2[i][0], -1, p1[i][0]*p2[i][1], p1[i][0]*p2[i][0], p1[i][0]]
        if i != 0:
            A = np.vstack((A, arr1))
        A = np.vstack((A, arr2))

    # apply SVD
    U, S, Vh = np.linalg.svd(A)
    V = Vh.T

    # find smallest eigenvalue and its corresponding eigenvector, which should be H
    minev = V[:, np.argmin(S)]
    H = np.reshape(minev, (3, 3))
    return H
```

In this section, I carefully followed instructions taught in the lectures and written in the lecture notes. First I switched $p_1 = H\, p_2$ into $A\, t = 0$, where size of matrices are 2n x 9, 9 x 1, 2n x 1 respectively, where vector t is just a flattened version of Homography matrix H. Then I computed SVD on matrix A to find out its smallest eigenvalue and its corresponding eigenvector, which should be vector t that minimized least squares under constraint that t is a unit vector. Then I reshaped vector t into H.

```python
def compute_h_norm(p1, p2):
    # TODO ...
    # number of pairs of corresponding interest points
    N = p1.shape[0]

    # construct Normalization matrix T1, T2
    p2_mean = np.mean(p2, axis=0)
    p1_mean = np.mean(p1, axis=0)
    sump2 = 0
    sump1 = 0
    for i in range(N):
        sump2 += ((p2[i][0]-p2_mean[0])**2 + (p2[i][1]-p2_mean[1])**2)**0.5
        sump1 += ((p1[i][0]-p1_mean[0])**2 + (p1[i][1]-p1_mean[1])**2)**0.5
    s2 = (math.sqrt(2)*N)/sump2
    s1 = (math.sqrt(2)*N)/sump1
    T2 = s2 * np.array([[1, 0, -p2_mean[1]], [0, 1, -p2_mean[0]], [0, 0, 1/s2]])
    T1 = s1 * np.array([[1, 0, -p1_mean[1]], [0, 1, -p1_mean[0]], [0, 0, 1/s1]])

    # normalize p1, p2 using T1, T2
    p2_norm = np.zeros(p2.shape)
    p1_norm = np.zeros(p1.shape)
    for i in range(N):
        coord2 = np.array([p2[i][1], p2[i][0], 1])
        result2 = np.matmul(T2, coord2.T)
        p2_norm[i][0] = result2[1]
        p2_norm[i][1] = result2[0]

        coord1 = np.array([p1[i][1], p1[i][0], 1])
        result1 = np.matmul(T1, coord1.T)
        p1_norm[i][0] = result1[1]
        p1_norm[i][1] = result1[0]

    # construct A
    A = [-p2_norm[0][1], -p2_norm[0][0], -1, 0, 0, 0, p1_norm[0][1]*p2_norm[0][1],
```

For compute_h_norm, I normalized each corresponding interest points so that average distance of them is equal to $\sqrt{2}$. For this purpose, I constructed normalization matrix T1 and T2. After normalizing I computed H as the same method in compute_h and undid normalization

$T_1 p_1 = H_{bar} T_2\, p_2 \;\; => \;\; H = T_1^{-1} H_{bar} T_2$

Part 2

```python
def warp_image(igs_in, igs_ref, H):
    # TODO ...
    H_inv = np.linalg.inv(H)
    new = np.zeros((1680, 2240, 3), dtype=np.uint8)
    new_pd = np.pad(igs_ref, ((350, 262), (1200, 0), (0, 0)), mode='constant', constant_values=0)
    for i in range(-350, 1330):
        for j in range(-1200, 1040):
            coord = np.array([j, i, 1])
            result = np.matmul(H_inv, coord.T)
            alpha = 1 / result[2]
            results = np.array([result[1]*alpha, result[0]*alpha])

            # bilinear interpolation
            if 0 <= results[0] < igs_in.shape[0] and 0 <= results[1] < igs_in.shape[1]:
                m = np.floor(results[0]).astype(int)
                n = np.floor(results[1]).astype(int)
                a = results[0] - np.floor(results[0])
                b = results[1] - np.floor(results[1])
                if m != igs_in.shape[0]-1 and n != igs_in.shape[1] -1:
                    pixel_ij = igs_in[m][n]
                    pixel_i_1_j = igs_in[m + 1][n]
                    pixel_i_j_1 = igs_in[m][n + 1]
                    pixel_i_1_j_1 = igs_in[m + 1][n + 1]

                    new_pixel = (1-a)*(1-b)*pixel_ij + a*(1-b)*pixel_i_1_j + a*b*pixel_i_1_j_1 + (1-a)*b*pixel_i_j_1
                    new[i + 350][j + 1200] = new_pixel
                    new_pd[i + 350][j + 1200] = new_pixel
        print(i)

    igs_warp = new
    igs_merge = new_pd

    return igs_warp, igs_merge
```

Criterion for selecting the correspondences : I manually selected 16 pairs of corresponding interest points which seemingly are at the corners and are local maxima.

I used inverse warping. First I constructed a reference image large enough. Then I applied inverse function and if the result falls within the area of input image, I applied bilinear interpolation and copied the pixel intensity. In this way I could avoid having holes in the reference image.

Part 3

```
def rectify(igs, p1, p2):
    # TODO ...
    H = compute_h_norm(p2, p1)
    H_inv = np.linalg.inv(H)
    igs_rec = np.zeros(igs.shape, dtype=np.uint8)
    for i in range(igs.shape[0]):
        for j in range(igs.shape[1]):
            coord = np.array([j, i, 1])
            result = np.matmul(H_inv, coord.T)
            alpha = 1 / result[2]
            results = np.array([result[1]*alpha, result[0]*alpha])

            # bilinear interpolation
            if 0 <= results[0] < igs.shape[0] and 0 <= results[1] < igs.shape[1]:
                m = np.floor(results[0]).astype(int)
                n = np.floor(results[1]).astype(int)
                a = results[0] - np.floor(results[0])
                b = results[1] - np.floor(results[1])
                if m != igs.shape[0]-1 and n != igs.shape[1] -1:
                    pixel_ij = igs[m][n]
                    pixel_i_1_j = igs[m + 1][n]
                    pixel_i_j_1 = igs[m][n + 1]
                    pixel_i_1_j_1 = igs[m + 1][n + 1]

                    new_pixel = (1-a)*(1-b)*pixel_ij + a*(1-b)*pixel_i_1_j + a*b*pixel_i_1_j_1 + (1-a)*b*pixel_i_j_1
                    igs_rec[i][j] = new_pixel
        print(i)

    return igs_rec
```

Criterion for selecting the correspondences : I manually selected 8 pairs of corresponding interest points (8 corner points) which seemingly are at the corners and are local maxima.

I used inverse warping. First I constructed a reference image whose size is same as the input image. Then I applied inverse function and if the result falls within the area of input image, I applied bilinear interpolation and copied the pixel intensity. In this way I could avoid having holes in the reference image.