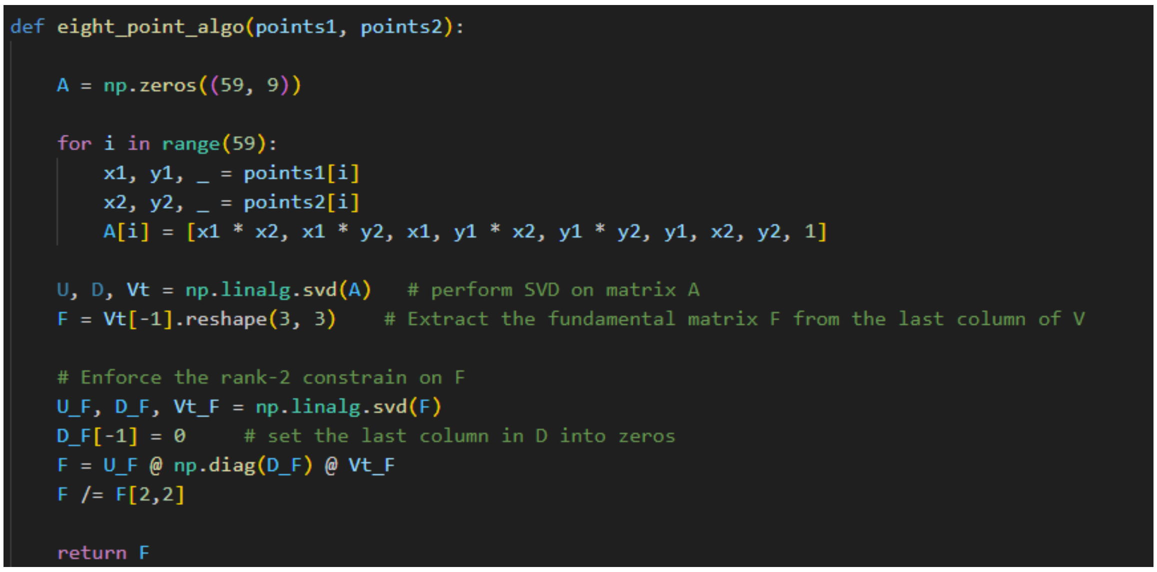
Q1 Fundamental Matrix Estimation from Point Correspondences

Implementation

Step 1 of (a.)

- Code



```python
def eight_point_algo(points1, points2):

    A = np.zeros((59, 9))

    for i in range(59):
        x1, y1, _ = points1[i]
        x2, y2, _ = points2[i]
        A[i] = [x1 * x2, x1 * y2, x1, y1 * x2, y1 * y2, y1, x2, y2, 1]

    U, D, Vt = np.linalg.svd(A)    # perform SVD on matrix A
    F = Vt[-1].reshape(3, 3)     # Extract the fundamental matrix F from the last column of V

    # Enforce the rank-2 constrain on F
    U_F, D_F, Vt_F = np.linalg.svd(F)
    D_F[-1] = 0      # set the last column in D into zeros
    F = U_F @ np.diag(D_F) @ Vt_F
    F /= F[2,2]

    return F
```

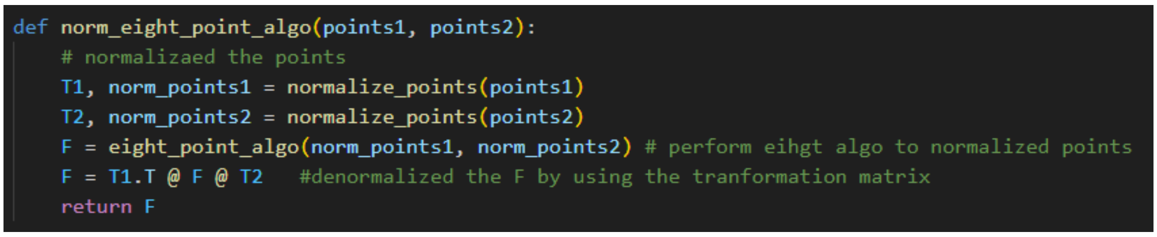
- Explanation

首先將對應 matching point 根據公式去計算矩陣 A，再對 A 做 SVD 分解成三個矩陣 U、D、Vt 的相乘。取 Vt 最後一行在 reshape 成 3x3 的大小作為 fundamental matrix F，再對 F 做 SVD 分解成三個矩陣 U_F、D_F、V_F。透過將 D_F 最後一行設成 0，讓 F 確保可以是一個 2-rank 的矩陣。最後在將改變後的 D_F 乘回去得到最終的 F。

Step 2 f (b)

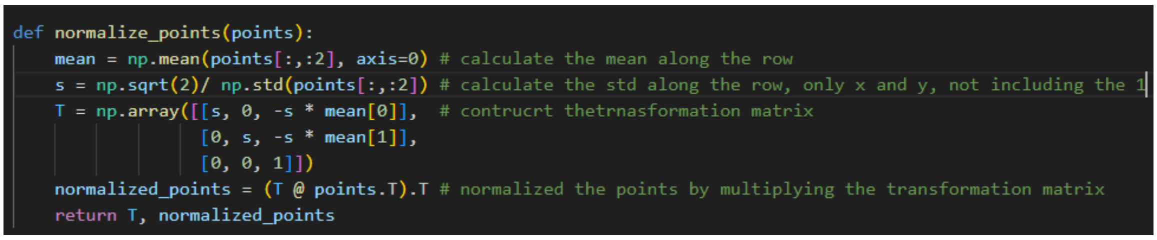
- Code

# normalized eight point algorithm



```python
def norm_eight_point_algo(points1, points2):
    # normalizaed the points
    T1, norm_points1 = normalize_points(points1)
    T2, norm_points2 = normalize_points(points2)
    F = eight_point_algo(norm_points1, norm_points2) # perform eihgt algo to normalized points
    F = T1.T @ F @ T2    #denormalized the F by using the tranformation matrix
    return F
```

# normalization



```python
def normalize_points(points):
    mean = np.mean(points[:,:2], axis=0) # calculate the mean along the row
    s = np.sqrt(2)/ np.std(points[:,:2]) # calculate the std along the row, only x and y, not including the 1
    T = np.array([[s, 0, -s * mean[0]],  # contrucrt thetrnasformation matrix
                  [0, s, -s * mean[1]],
                  [0, 0, 1]])
    normalized_points = (T @ points.T).T # normalized the points by multiplying the transformation matrix
    return T, normalized_points
```

Step 2 f (b)

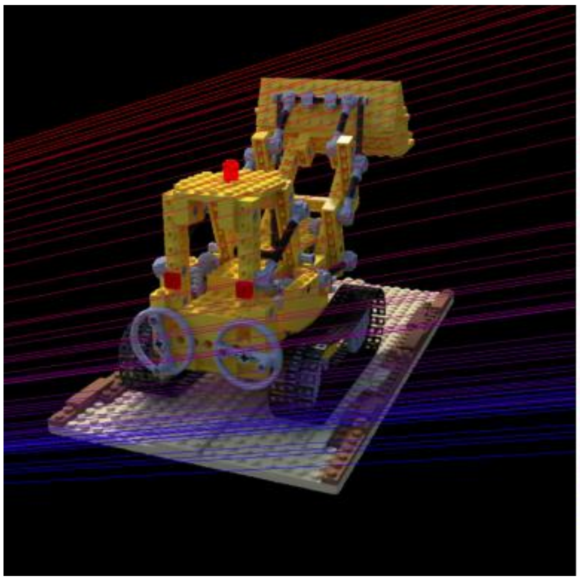
- Explanation

首先對點做 normalization，normalization 是將每個點減去 mean 乘上根號 2 再除與 std，根據這樣的想法創建出一個 transformation matrix T1、T2。將 normalized 完的點輸入進 eight point algorithm 中到 F。最後再將 F 乘上 T1 的轉置以及 T2 完成 denormalization 得到最終的 F。

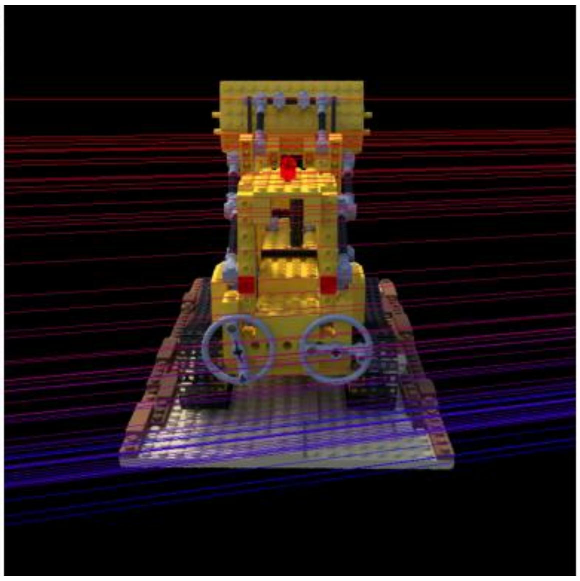
Step 3 of (c)

- Reuslt

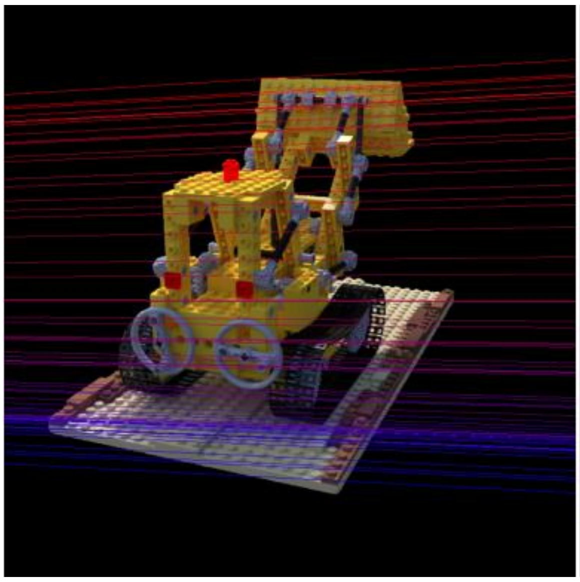
Distance: 25.64771846800745



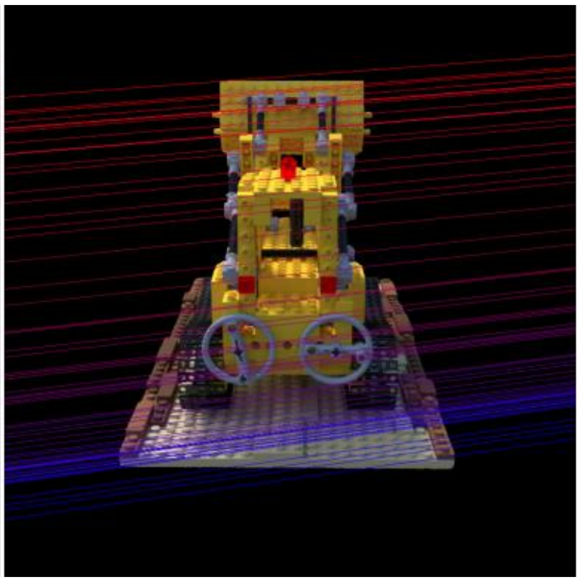
Distance: 25.260657259241153



Distance: 0.9005558198177118



Distance: 0.9154447659757211

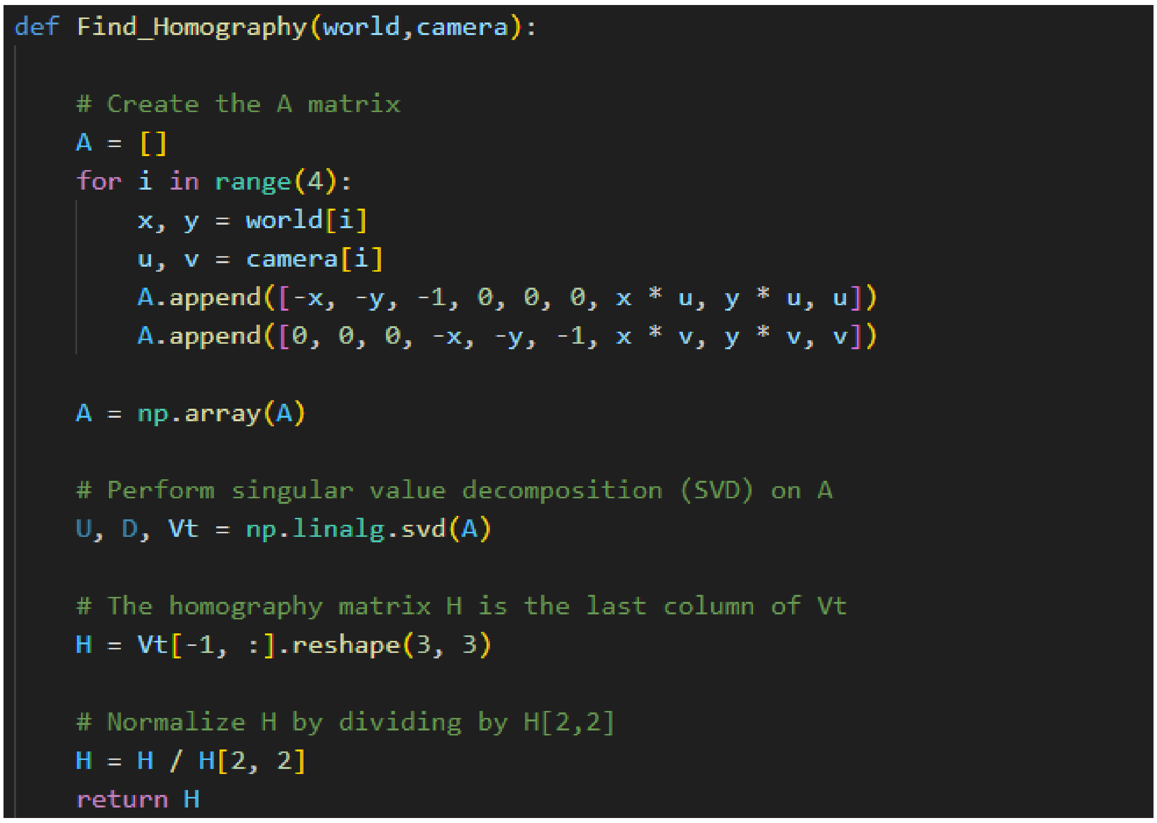


Q2 Homography Transform

Implementation

Step 1 of (a.)

- Code



```python
def Find_Homography(world,camera):

    # Create the A matrix
    A = []
    for i in range(4):
        x, y = world[i]
        u, v = camera[i]
        A.append([-x, -y, -1, 0, 0, 0, x * u, y * u, u])
        A.append([0, 0, 0, -x, -y, -1, x * v, y * v, v])

    A = np.array(A)

    # Perform singular value decomposition (SVD) on A
    U, D, Vt = np.linalg.svd(A)

    # The homography matrix H is the last column of Vt
    H = Vt[-1, :].reshape(3, 3)

    # Normalize H by dividing by H[2,2]
    H = H / H[2, 2]
    return H
```

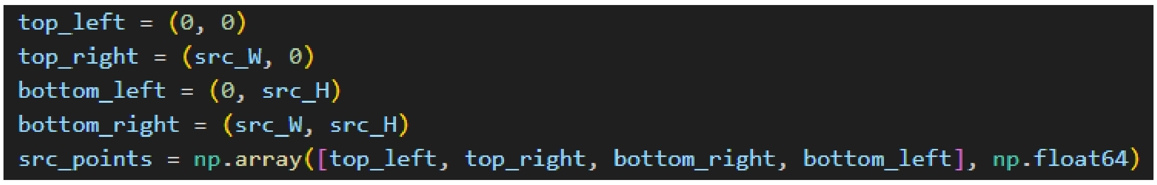
- Explanation

首先將對應點根據 homography 的公式去計算矩陣 A，再對 A 做 SVD 分解成三個矩陣 U、D、Vt 的相乘。取 Vt 最後一行在 reshape 成 3x3 大小作為 homography matrix H。最後就是利用右下角的數值對矩陣 H 做 normalization。

Step 1 of (b.)

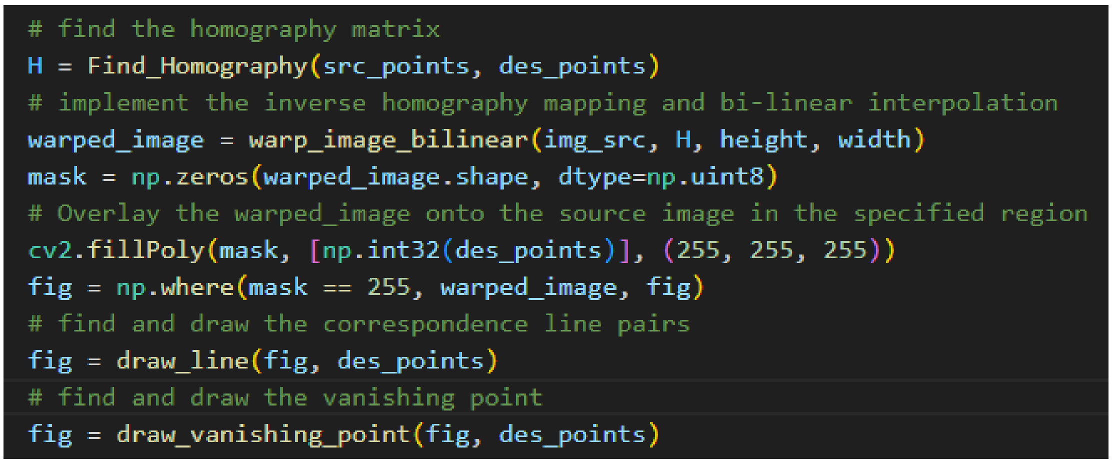
- Code

  # points for the target image



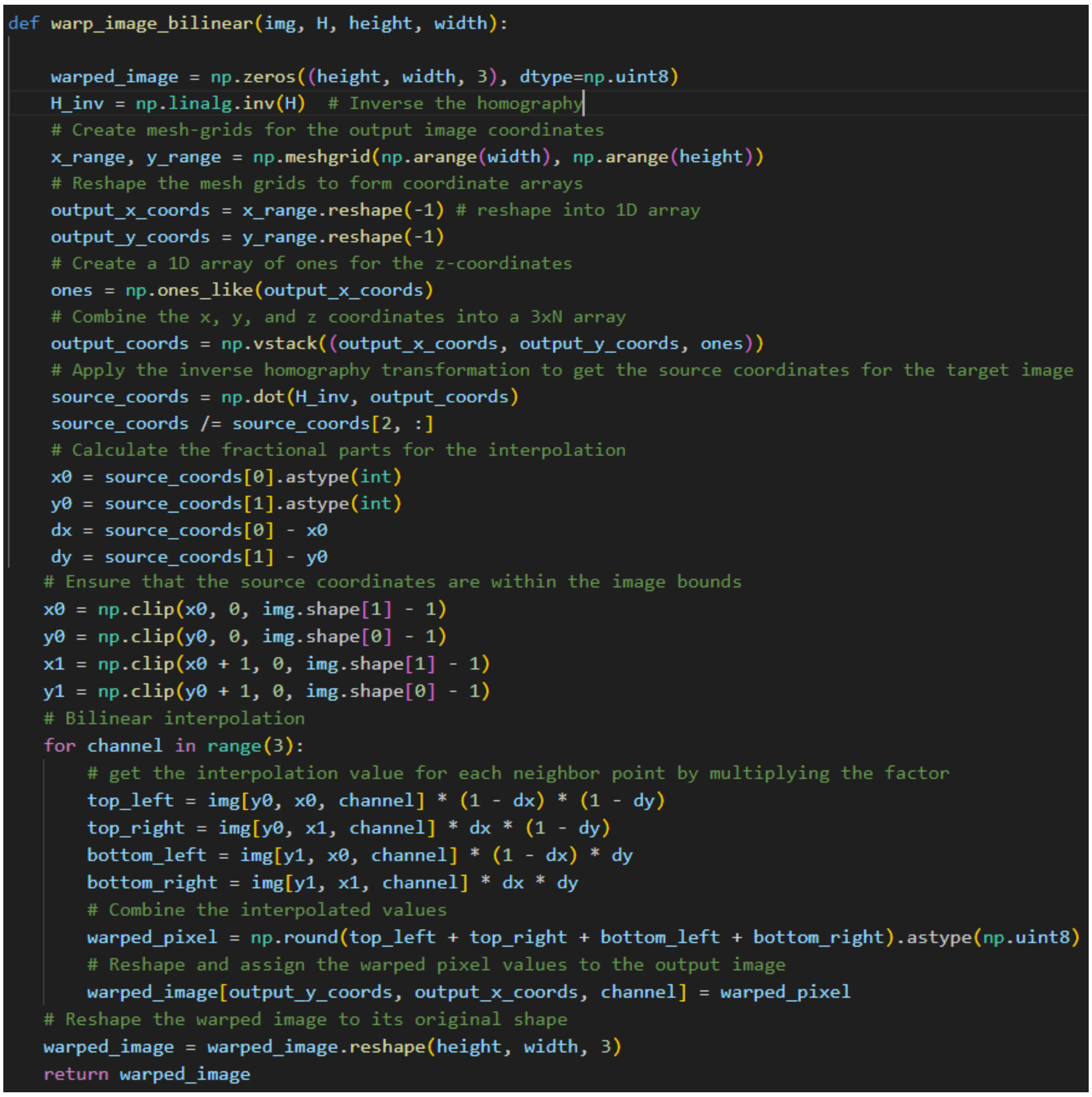
```python
top_left = (0, 0)
top_right = (src_W, 0)
bottom_left = (0, src_H)
bottom_right = (src_W, src_H)
src_points = np.array([top_left, top_right, bottom_right, bottom_left], np.float64)
```

# implement steps



```python
# find the homography matrix
H = Find_Homography(src_points, des_points)
# implement the inverse homography mapping and bi-linear interpolation
warped_image = warp_image_bilinear(img_src, H, height, width)
mask = np.zeros(warped_image.shape, dtype=np.uint8)
# Overlay the warped_image onto the source image in the specified region
cv2.fillPoly(mask, [np.int32(des_points)], (255, 255, 255))
fig = np.where(mask == 255, warped_image, fig)
# find and draw the correspondence line pairs
fig = draw_line(fig, des_points)
# find and draw the vanishing point
fig = draw_vanishing_point(fig, des_points)
```

# detail of function: wrap_image_bilinear



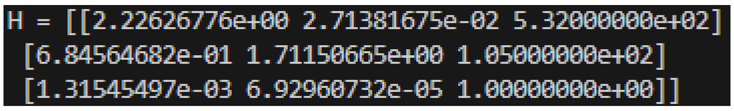
```python
def warp_image_bilinear(img, H, height, width):

    warped_image = np.zeros((height, width, 3), dtype=np.uint8)
    H_inv = np.linalg.inv(H)  # Inverse the homography
    # Create mesh-grids for the output image coordinates
    x_range, y_range = np.meshgrid(np.arange(width), np.arange(height))
    # Reshape the mesh grids to form coordinate arrays
    output_x_coords = x_range.reshape(-1) # reshape into 1D array
    output_y_coords = y_range.reshape(-1)
    # Create a 1D array of ones for the z-coordinates
    ones = np.ones_like(output_x_coords)
    # Combine the x, y, and z coordinates into a 3xN array
    output_coords = np.vstack((output_x_coords, output_y_coords, ones))
    # Apply the inverse homography transformation to get the source coordinates for the target image
    source_coords = np.dot(H_inv, output_coords)
    source_coords /= source_coords[2, :]
    # Calculate the fractional parts for the interpolation
    x0 = source_coords[0].astype(int)
    y0 = source_coords[1].astype(int)
    dx = source_coords[0] - x0
    dy = source_coords[1] - y0
    # Ensure that the source coordinates are within the image bounds
    x0 = np.clip(x0, 0, img.shape[1] - 1)
    y0 = np.clip(y0, 0, img.shape[0] - 1)
    x1 = np.clip(x0 + 1, 0, img.shape[1] - 1)
    y1 = np.clip(y0 + 1, 0, img.shape[0] - 1)
    # Bilinear interpolation
    for channel in range(3):
        # get the interpolation value for each neighbor point by multiplying the factor
        top_left = img[y0, x0, channel] * (1 - dx) * (1 - dy)
        top_right = img[y0, x1, channel] * dx * (1 - dy)
        bottom_left = img[y1, x0, channel] * (1 - dx) * dy
        bottom_right = img[y1, x1, channel] * dx * dy
        # Combine the interpolated values
        warped_pixel = np.round(top_left + top_right + bottom_left + bottom_right).astype(np.uint8)
        # Reshape and assign the warped pixel values to the output image
        warped_image[output_y_coords, output_x_coords, channel] = warped_pixel
    # Reshape the warped image to its original shape
    warped_image = warped_image.reshape(height, width, 3)
    return warped_image
```

Step 2 of (b.)

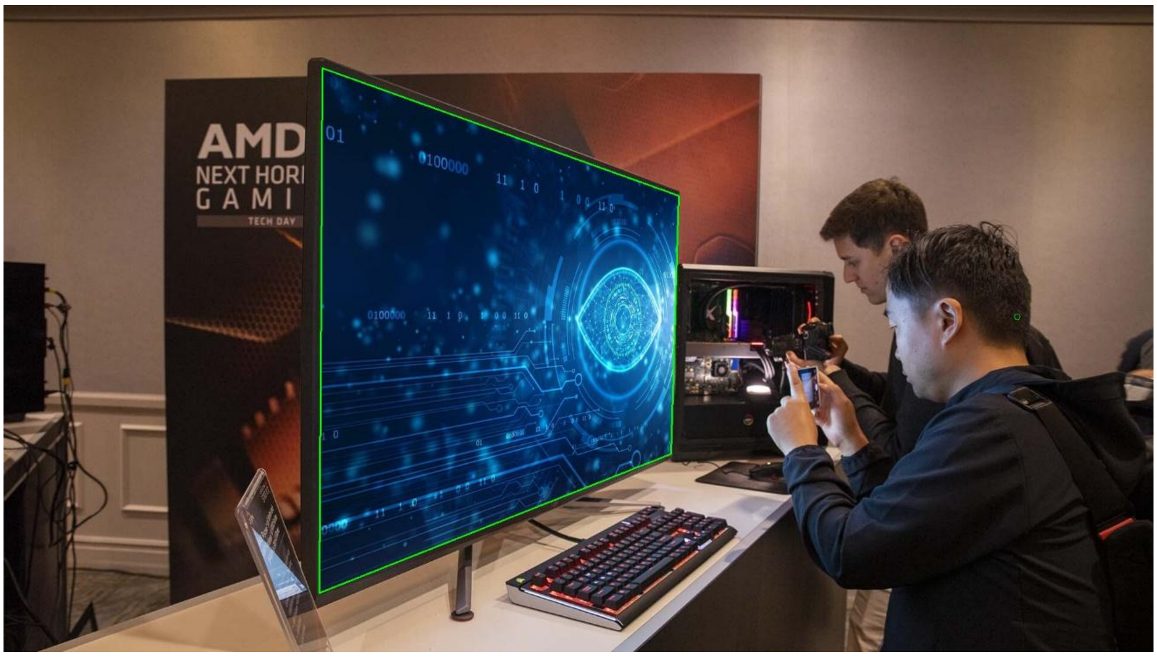
- Explanation

首先對 H 做 inverse 再乘上 target 的點座標，得到 target 在 source image 上 mapping 的座標點。在計算出 interpolate 的小數部分通過乘以小數部分來取得每個相鄰點的插值值。最後相加並將相加結果分配給輸出圖像，最後將所得到的結果覆蓋在 figure 上

- Result

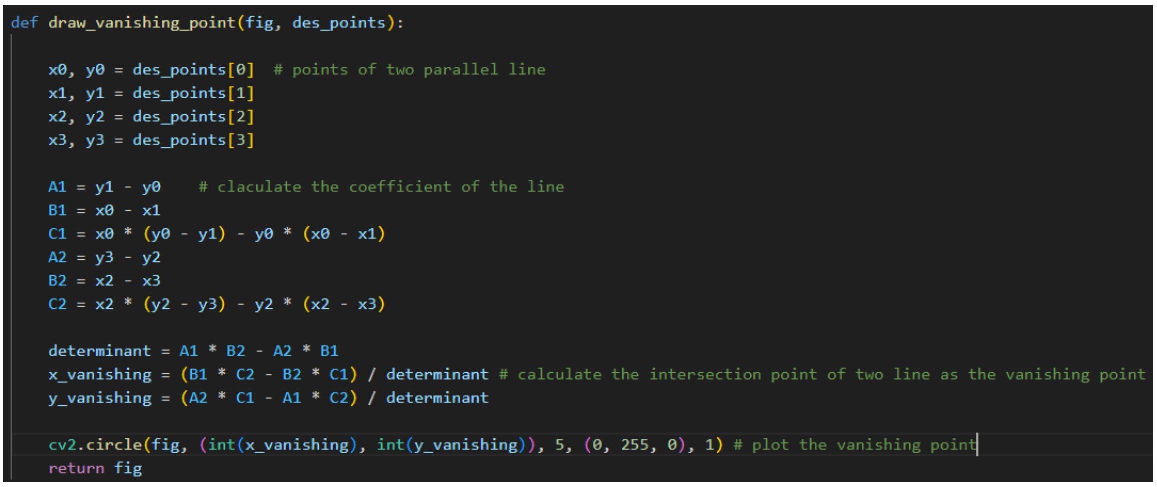


```
H = [[2.22626776e+00 2.71381675e-02 5.32000000e+02]
 [6.84564682e-01 1.71150665e+00 1.05000000e+02]
 [1.31545497e-03 6.92960732e-05 1.00000000e+00]]
```



Step 2 of (c.)

Code



```python
def draw_vanishing_point(fig, des_points):

    x0, y0 = des_points[0]  # points of two parallel line
    x1, y1 = des_points[1]
    x2, y2 = des_points[2]
    x3, y3 = des_points[3]

    A1 = y1 - y0    # claculate the coefficient of the line
    B1 = x0 - x1
    C1 = x0 * (y0 - y1) - y0 * (x0 - x1)
    A2 = y3 - y2
    B2 = x2 - x3
    C2 = x2 * (y2 - y3) - y2 * (x2 - x3)

    determinant = A1 * B2 - A2 * B1
    x_vanishing = (B1 * C2 - B2 * C1) / determinant # calculate the intersection point of two line as the vanishing point
    y_vanishing = (A2 * C1 - A1 * C2) / determinant

    cv2.circle(fig, (int(x_vanishing), int(y_vanishing)), 5, (0, 255, 0), 1) # plot the vanishing point
    return fig
```

- Explanation

利用兩條線延伸求出交集的點作為 vanishing paoint。