HW2 Report

> 111062566 劉緒紳

Part 1 - Implementation

a. eight-point algorithm

程式碼如下：

```
def eight_point(pt1, pt2):
    uv_matrix = (pt1[:, :, np.newaxis] @ pt2[:, np.newaxis]).reshape((-1, 9))

    f_hat = svd_least_square(uv_matrix)
    fundamental_mat = enforce_rank2(f_hat)
    return fundamental_mat
```

首先利用兩張圖的對應點（`pt1`、`pt2`）計算出 uv_matrix，計算方式如下圖所示。

$$\mathcal{U} = \begin{pmatrix} x_1x_1' & x_1y_1' & x_1 & y_1x_1' & y_1y_1' & y_1 & x_1' & y_1' & 1 \\ x_2x_2' & x_2y_2' & x_2 & y_2x_2' & y_2y_2' & y_2 & x_2' & y_2' & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_nx_n' & x_ny_n' & x_n & y_nx_n' & y_ny_n' & y_n & x_n' & y_n' & 1 \end{pmatrix}$$

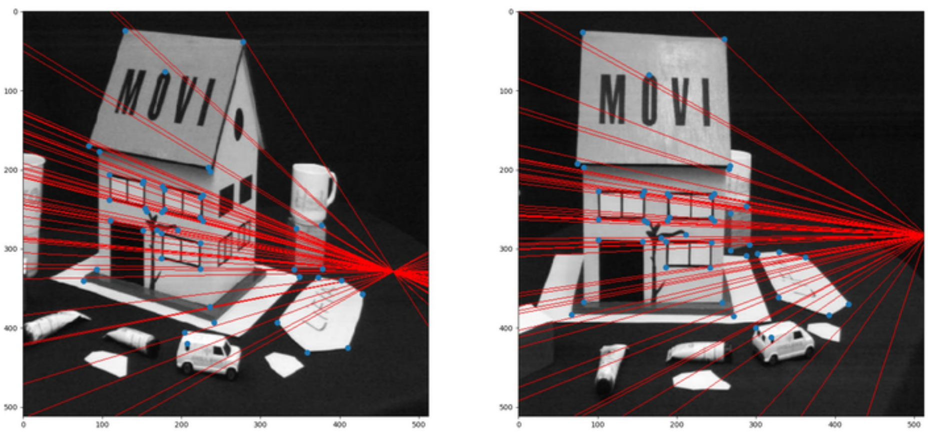
接著計算 $\mathcal{U}\mathbf{f} = 0$ 的解，這相當於求解下列 least square 問題：

$$\min_{\mathbf{f}} \|\mathcal{U}\mathbf{f}\|, \text{ subject to } \|\mathbf{f}\| = 1$$

解法為先將 $\mathcal{U}$ 經過 SVD 分解成 UDV^T，此時 V^T 的最後一個 row 即為 $\mathbf{f}$ 的解，再將 $\mathbf{f}$ reshape 成 3×3 的矩陣 $\hat{F}$。

最後，因為最終的 Fundamental matrix 必須是 rank-2 的，因此我們必須對前一步取得的 $\hat{F}$ 再做一次 SVD 分解成 $\mathbf{udv}$ 後，將 $\mathbf{d}$ 的最後一個 entry 設為 0，得到 $\mathbf{d}'$，再計算最後的 Fundamental matrix $F = \mathbf{ud}'\mathbf{v}$。

下圖為找出的 epipolar lines，這些線明顯地與藍點有一些距離，而計算得到的點與線平均距離為 9.7 和 14.5。



b. normalized eight-point algorithm

Normalized 版本會先將對應點進行置中以及縮放，以下是計算 normalization 矩陣的程式碼：

```
def normalization_matrix(pts):
    center = pts[..., :2].mean(axis=0)
    diff = (pts[..., :2] - center).T
    mean_dis = (diff[0] ** 2 + diff[1] ** 2).mean() / 2
    s = 1 / np.sqrt(mean_dis)

    trans_mat = np.array([[s, 0, -s * center[0]],
                          [0, s, -s * center[1]],
                          [0, 0, 1]])
    return trans_mat
```

下方以數學式來表達 Normalization 的計算過程，而在程式中則是以矩陣形式來計算的。

$$
\begin{aligned}
q_i &= s(p_i - \bar{p}), \\
\text{where } \bar{p} &= \frac{1}{N}\sum_{i=1}^{n} p_i, \\
s &= \sqrt{2/\text{mean square distance}}
\end{aligned} \tag{1}
$$

$$
q_i = \mathcal{T} p_i \tag{2}
$$

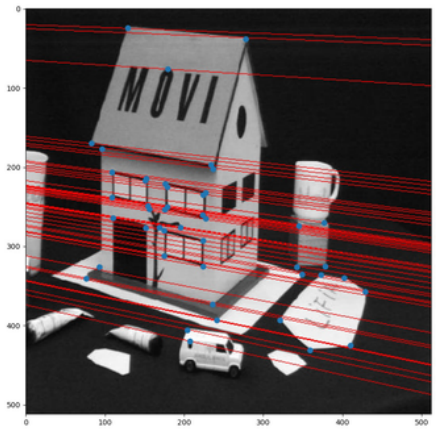
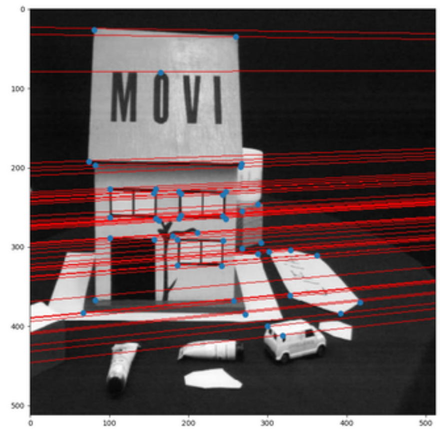
將兩張圖的對應點都 normalized 後，就可以套用原本的 eight-point 演算法，計算 normalized 後的 Fundamental matrix。最後，再將其還原成未 normalized 的 Fundamental matrix。

$$
\begin{aligned}
& p^T F p' = 0 \\
& \Rightarrow q^T F_q q' = 0 \\
& \Rightarrow p^T \mathcal{T}^T F_q \mathcal{T}' p' = 0 \\
& \Rightarrow F = \mathcal{T}^T F_q \mathcal{T}'
\end{aligned}
$$

程式碼如下所示：

```
def normalized_eight_point(pt1, pt2):
    t1 = normalization_matrix(pt1)
    t2 = normalization_matrix(pt2)
    q1 = (t1 @ pt1[..., np.newaxis]).reshape((-1, 3))
    q2 = (t2 @ pt2[..., np.newaxis]).reshape((-1, 3))

    return t1.T @ eight_point(q1, q2) @ t2
```

下圖為找出的 epipolar lines，可以看到找到的線都有穿過藍點。這個版本的點與線平均距離為 0.88、0.89，明顯較原始版本來得小，表示求出的 epipolar line 較為精確。

Part 2 - Implementation

a. 首先，每組對應點可以產生 2×9 的矩陣：

$$\begin{bmatrix} x_i & y_i & 1 & 0 & 0 & 0 & -x'_i x_i & -x'_i y_i & -x'_i \\ 0 & 0 & 0 & x_i & y_i & 1 & -y'_i x_i & -y'_i y_i & -y'_i \end{bmatrix}$$

而 homography matrix 共有 8 個自由度（DOF），因此解 homography matrix 需要 4 組對應點。

有了對應點之後，就可以透過 SVD 來求解 least square solution，這部分和前一題很類似，主要差別在於 homography matrix 並沒有 rank-2 的限制，因此不需要做第二次的 SVD 分解。

以下是這部分的程式碼：

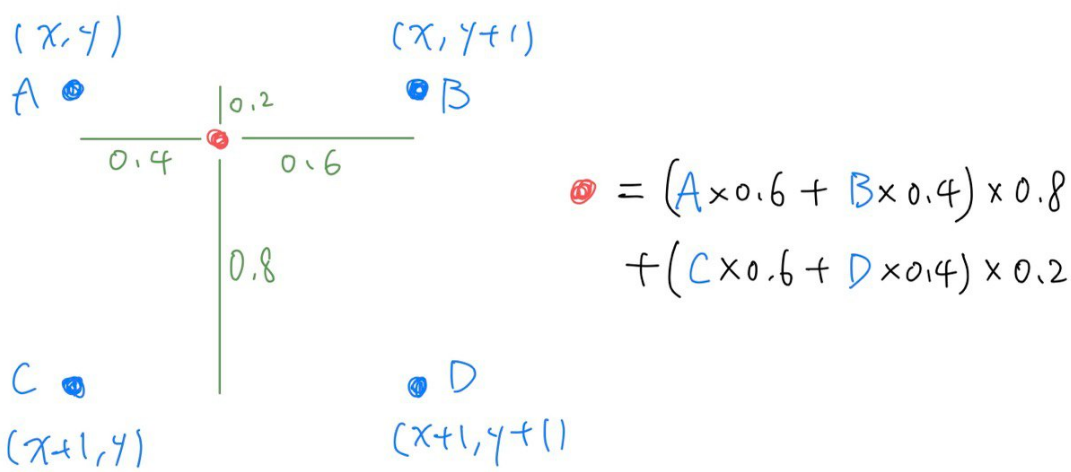
```
def relation_matrix(pt1, pt2):
    x1, y1 = pt1
    x2, y2 = pt2
    return np.array(
        [[x1, y1, 1, 0, 0, 0, -x2 * x1, -x2 * y1, -x2],
         [0, 0, 0, x1, y1, 1, -y2 * x1, -y2 * y1, -y2]]
    )

def svd_least_square(a_matrix):
    _, _, vt = np.linalg.svd(a_matrix)
    x = vt[-1].reshape((3, 3))
    return x
```

```
A = np.vstack([relation_matrix(p, q) for p, q in zip(pts1, pts2)])
H = svd_least_square(A)
```

b. 有了 Homography matrix 後，就可以把 target image 的每個 pixel warp 回 source 的座標：

```
warp_matrix = (homography @ pixel_indices[..., np.newaxis]).reshape((-1, 3))
# normalize homogeneous coordinate
warp_matrix = (warp_matrix / warp_matrix[:, 2:])
```

因為 warp 所取得的座標不會位在整數點，所以要用周圍的 pixel 計算 bilinear interpolation：



bilinear interpolation 計算範例

而這個部分的程式碼如下所示：

```
def bilinear_pixel_color(src, pixels, dw):
    a = src[pixels[:, 0], pixels[:, 1]]
    b = src[pixels[:, 0], pixels[:, 1] + 1]
    c = src[pixels[:, 0] + 1, pixels[:, 1]]
    d = src[pixels[:, 0] + 1, pixels[:, 1] + 1]

    return ((a * (1 - dw[:, 1:2]) + b * dw[:, 1:2]) * (1 - dw[:, 0:1]) +
            (c * (1 - dw[:, 1:2]) + d * dw[:, 1:2]) * (dw[:, 0:1]))
```

另外，這一題的所有運算（包含 warping、bilinear interpolation）皆使用了 vectorized 的方式來寫，因此執行速度比起使用 for 迴圈快了許多。

下圖為最後 rectified 完成的圖片（角落黑色部分是因為 warp 後落在 source image 的外面，因此沒有顏色可以填入）：