# HW3 Writeup

## M1522.001000 Computer Vision

Junyeong Park (2017-19460)

---

# Part 1

## Part 1.1 H = compute_h(p1, p2)

```
1  def compute_h(p1, p2):
       # TODO ...
3      """
       svd(A) : Gets U, S, V^T that satisfies A = U S V^T
5      S : Eigenvalue of A^T*A
       The eigenvector when the eigenvalue is 0 can be calculated by V_T[-1, :]
7      """
       n = len(p1) - 1
9      A = []  # 2N x 9 matrix
       # Build A
11     while n > 0:
           A.append([p2[n][0], p2[n][1], 1, 0, 0, 0, -p1[n][0] * p2[n][0], -p1[n][0] * p2[n][1], -p1[n
       ↪ ][0]])
13         A.append([0, 0, 0, p2[n][0], p2[n][1], 1, -p1[n][1] * p2[n][0], -p1[n][1] * p2[n][1], -p1[n
       ↪ ][1]])
           n -= 1
15     A = np.asarray(A)

17     # SVD
       U, S, V_T = np.linalg.svd(A)
19     h = V_T[-1, :]
       H = np.reshape(h, (3, 3))
21     # Normalize
       H /= H[2][2]
23
       return H
25
```

Line 8∼15 are for building **A**, a $2N \times 9$ matrix. Line 17∼24 are for building **H**, a homography matrix. The expressions of these matrices are mentioned on **Theory Question 3**.

To get **H**, we have to find **h** that minimizes $|\mathbf{At}|^2 = (\mathbf{Ah})^T(\mathbf{Ah}) = \lambda$. After we take a derivative w.r.t. **h** on this equation, we get a result that $\lambda$ is an eigenvalue of $\mathbf{A}^T\mathbf{A}$, and the solution **h** is its eigenvector.

Getting **h** that minimizes $\lambda$ can be done easily by using SVD, which stands for singular value decomposition. `np.linalg.svd(A)` returns three matrices $\mathbf{U}, \mathbf{S}, \mathbf{V}^T$. **S** is a diagonal matrix that contains eigenvalues of $\mathbf{A}^T\mathbf{A}$, and **V** is a matrix that contains the corresponding eigenvectors. We can get **h** that makes $\lambda = 0$ from the last row of $\mathbf{V}^T$. After reshaping **h**, we get an unnormalized homography matrix. By dividing **H** by $h_{22}$, we finally get a normalized homography matrix **H**.

## Part 1.2 H = compute_h_norm(p1, p2)

```
def compute_h_norm(p1, p2):
    # TODO ...
    # Transformation matrix
    # (x, y) -> (x/col, y/row)
    T = [[1/400, 0, 0], [0, 1/302, 0], [0, 0, 1]]
    T = np.asarray(T)
```

```
    inv_T = np.linalg.inv(T)

    p1_norm, p2_norm = p1, p2
    # Compute normalized coordinates
    for i in range(len(p1)):
        p1_coord = np.matmul(T, [p1_norm[i][0], p1_norm[i][1], 1])
        p2_coord = np.matmul(T, [p2_norm[i][0], p2_norm[i][1], 1])
        p1_norm[i][0], p1_norm[i][1] = p1_coord[0], p1_coord[1]
        p2_norm[i][0], p2_norm[i][1] = p2_coord[0], p2_coord[1]

    H_norm = compute_h(p1_norm, p2_norm)

    # H = T^-1 * H_norm * T
    H = np.matmul(np.matmul(inv_T, H_norm), T)
    # Floating point problem
    H /= H[2][2]

    return H
```

The basic idea of this function is actually the same with `compute_h`. The difference is that there is a transformation matrix $\mathbf{T}$ in `compute_h_norm`. $\mathbf{T}$ is a matrix that normalizes the coordinates by scaling $x$ by $\frac{1}{400}$ and $y$ by $\frac{1}{302}$. 400 and 302 are the width and height of the images, i.e. `wdc1.png` and `wdc2.png`, respectively. After computing `H_norm` from two normalized coordinates, we can finally get $\mathbf{H} = \mathbf{T}^{-1}\mathbf{H}_{norm}\mathbf{T}$.

Since it is not able to infer the shape of `igs_in` and `igs_ref` from the arguments, I had to hard-code parameters of the width and the height of the images.

# Part 2

## Part 2.1 p_in, p_ref = set_cor_mosaic()

```
def set_cor_mosaic():
    # TODO ...
    """
    p_in : N x 2 matrices of corresponded (x, y)^T coordinates in igs_in
    p_ref : N x 2 matrices of corresponded (x, y)^T coordinates in igs_ref
    N=13 point corresondences are chosen.
    Points that I chose are mentioned in the writeup.
    """
    p_in = [
        [370, 244],
        [155, 162],
        [214, 190],
        [164, 125],
        [339, 92],
        [173, 187],
        [217, 161],
        [88, 174],
        [223, 103],
        [384, 137],
        [359, 194],
        [357, 110],
        [325, 288]
    ]

    p_ref = [
        [111, 98],
        [347, 68],
        [272, 74],
        [387, 97],
        [259, 253],
        [305, 58],
        [298, 90],
```

```
        [376, 34],
        [368, 149],
        [161, 209],
        [146, 135],
        [218, 232],
        [130, 58]
    ]

    return p_in, p_ref

```

I chose 13 point correspondences from the images. Figure 1 denotes the points that I chose. Since the function doesn't get anything as an argument, I decided to hard-code these points. Theoretically, 4 point correspondences are enough. However, the more correspondences lead to a better result, because we use the method of least squares to compute **H**. Interestingly, I found out that choosing points sparsely also leads to a better result.

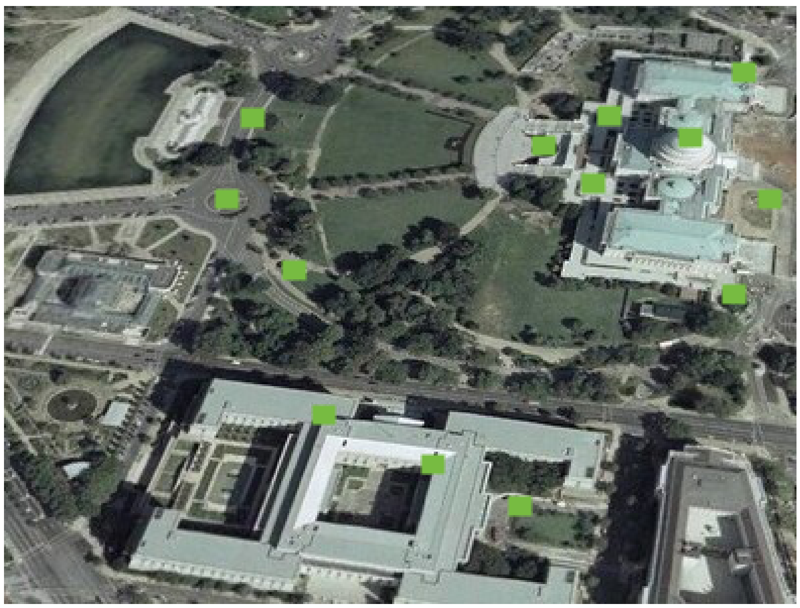

**Figure 1: 13 point correspondences on wdc2.png**

## Part 2.2 igs_warp, igs_merge = warp_image(igs_n, igs_ref, H)

```
def warp_image(igs_in, igs_ref, H):
    # TODO ...
    inv_H = np.linalg.inv(H)    # Inverse warping
    in_y, in_x, _ = igs_in.shape
    ref_y, ref_x, _ = igs_ref.shape

    # Corners of igs_in & igs_ref
    in_corners = [(0, 0), (in_y, in_x), (0, in_x), (in_y, 0)]
    ref_corners = [(0, 0), (ref_y, ref_x), (0, ref_x), (ref_y, 0)]

    # Maximum and minimum values of x & y in igs_warp
    min_x, min_y = float("inf"), float("inf")
    max_x, max_y = float("-inf"), float("-inf")

    # Compute min, max
    for i, j in in_corners:
        # Homogeneous coordinate
        x, y, w = np.matmul(H, [j, i, 1])
        x = x/w
        y = y/w
```

```python

        if x > max_x:
            max_x = int(x)
        if x < min_x:
            min_x = int(x)
        if y > max_y:
            max_y = int(y)
        if y < min_y:
            min_y = int(y)

    igs_warp = np.zeros((max_y - min_y, max_x - min_x, 3))

    # Compute igs_warp
    for i in range(0, max_x - min_x):
        for j in range(0, max_y - min_y):
            # Homogeneous coordinate
            x, y, w = np.matmul(inv_H, [i + min_x, j + min_y, 1])
            # interpolate
            x = int(x/w)
            y = int(y/w)
            # Colors
            r, g, b = 0, 0, 0
            if not (y < 0 or y >= in_y or x < 0 or x >= in_x):
                r, g, b = igs_in[y, x, :]
            igs_warp[j, i, :] = [r, g, b]

    # old : min & max of igs_warp
    old_min_x = min_x
    old_min_y = min_y
    old_max_x = max_x
    old_max_y = max_y

    # Compute min, max of igs_merge
    for i, j in ref_corners:
        if j > max_x:
            max_x = int(j)
        if j < min_x:
            min_x = int(j)
        if i > max_y:
            max_y = int(i)
        if i < min_y:
            min_y = int(i)

    igs_merge = np.zeros(((max_y - min_y), (max_x - min_x), 3))

    # Compute igs_merge
    for i in range(min_x, max_x):
        for j in range(min_y, max_y):
            r, g, b = 0, 0, 0
            # igs_warp
            if not (j < old_min_y or j >= old_max_y or i < old_min_x or i >= old_max_x):
                r, g, b = igs_warp[j - old_min_y, i - old_min_x, :]
                if r * g * b == 0.0:
                    if not (j < 0 or j >= ref_y or i < 0 or i >= ref_x):
                        r, g, b = igs_ref[j, i, :]
            # igs_ref
            else:
                if not (j < 0 or j >= ref_y or i < 0 or i >= ref_x):
                    r, g, b = igs_ref[j, i, :]
            igs_merge[j - min_y, i - min_x, :] = [r, g, b]

    return igs_warp, igs_merge

```

In this function, we get `igs_warp`, a warped version of `igs_in`, and `igs_merge`, a single mosaic image with a larger field of view containing both of the input images. Figure 2 is a mosaic image generated from `igs_merge`. I used an inverse warping technique, and interpolated the non-integer valued coordinates by type-casting them as `int()`. I tried to use other interpolation techniques such as bilinear interpolation, but I decided not to use them because they didn't really helped the quality of the output image and also took a lot of computation time to get the result. The rest of the code is explained in the comments.

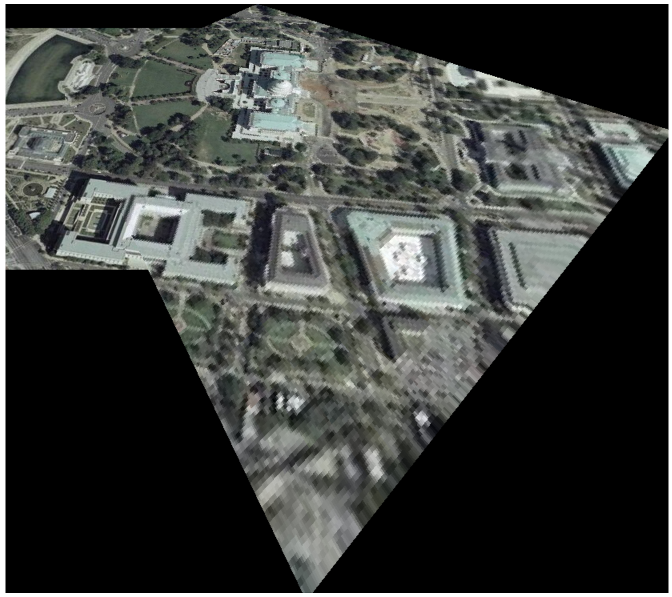

**Figure 2: Mosaic image**

# Part 3

## Part 3.1 c_in, c_ref = set_cor_rec()

```
def set_cor_rec():
    # TODO ...
    """
    p_in : N x 2 matrices of corresponded (x, y)^T coordinates in igs_in
    p_ref : N x 2 matrices of corresponded (x, y)^T coordinates in igs_ref
    N=4 point corresondences are chosen.
    Points that I chose are mentioned in the writeup.
    """
    c_in = [
        [163, 14],
        [260, 25],
        [163, 258],
        [260, 245]
    ]

    c_ref = [
        [200, 70],
        [260, 1],
        [150, 252],
        [258, 250]
    ]

    return c_in, c_ref
```

Figure 3 denotes the points that I chose. The points where red lines meet consist `c_in`. For `c_out`, I adjusted the points manually to get a better result.

## Part 3.2 igs_rec = rectify(igs, p1, p2)


**Figure 3: iphone.png**


**Figure 4: Rectified version of Figure 3**

```python
def rectify(igs, p1, p2):
    # TODO ...
    # Get homography matrix of p1->p2
    H = compute_h(p2, p1)
    # Use warp_image to get igs_rec
    igs_rec, _ = warp_image(igs, np.zeros(igs.shape), H)

    return igs_rec

```

In this function, I reused the functions from **Part 1** and **Part 2**. Figure 4 is an image generated from `igs_rec`. The problem is that I was not able to use `compute_h_norm` because the transformation matrix fits exclusively for the images in **Part 2**. This led to the manual adjustment on `c_ref`, since the scale is different on the input and the output.