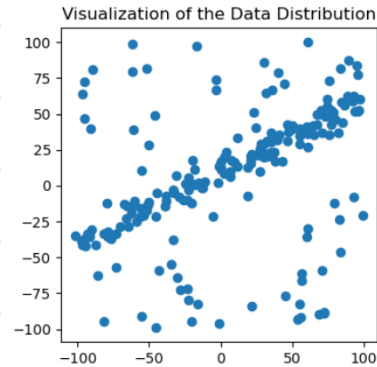


# ICV S24 Assignment #3. 장원근 (2020-10240)

## 1. Least Square Line Fitting.

- Visualization of the data distribution  $\Rightarrow$  ( $N=200$ )



- Second moment :  $U^T U$

where  $U = \begin{bmatrix} x_1 - \mu_x & y_1 - \mu_y \\ \vdots & \vdots \\ x_n - \mu_x & y_n - \mu_y \end{bmatrix} \Rightarrow$

and  $\mu_x = \frac{1}{n} \sum_{i=1}^n x_i, \mu_y = \frac{1}{n} \sum_{i=1}^n y_i$

second moment:

$$\begin{bmatrix} 680239.88956594 & 206911.73976855 \\ 206911.73976855 & 433367.46101812 \end{bmatrix}$$

eigenvalue of second moment:

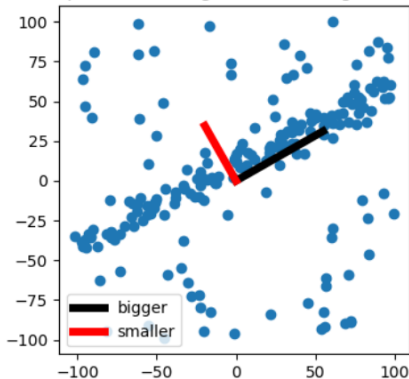
$$[797737.20772021 \quad 315870.14286384]$$

eigenvector of second moment:

$$\begin{bmatrix} 0.86957598 & -0.49379917 \\ 0.49379917 & 0.86957598 \end{bmatrix}$$

- The eigenvalues & eigenvectors w.r.t the data distribution :

Interpretation of eigenvalues & eigenvectors



Set  $(v_1, \lambda_1), (v_2, \lambda_2)$  as eigen pairs where

$$\|v_1\| = \|v_2\| = 1, \lambda_1 > \lambda_2.$$

Black & red lines are scaled eigenvectors from origin.

$$\begin{cases} \text{Black} : \sqrt{\lambda_1/n} v_1 \\ \text{Red} : \sqrt{\lambda_2/n} v_2 \end{cases}$$

We can see that  $v_1$  is parallel with the line that points make.

And  $v_2$  is vertical to  $v_1$ .

- Find the optimal line  $l: ax+by=d$  in the total least square sense.

$$E = \sum_{i=1}^n (ax_i + by_i - d)^2$$

$$\frac{\partial E}{\partial d} = -2 \sum_{i=1}^n (ax_i + by_i - d) = 0 \rightarrow d = a \cdot \mu_x + b \cdot \mu_y.$$

$$E = \sum_{i=1}^n \{a(x_i - \mu_x) + b(y_i - \mu_y)\}^2$$

$$= a^2 \sum_{i=1}^n (x_i - \mu_x)^2 + 2ab \sum_{i=1}^n (x_i - \mu_x)(y_i - \mu_y) + b^2 \sum_{i=1}^n (y_i - \mu_y)^2$$

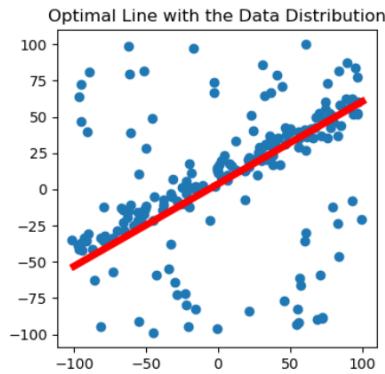
$$= \|U\mathbf{n}\|^2 \quad \text{where } \mathbf{n} = \begin{bmatrix} a \\ b \end{bmatrix}$$

$$\mathbf{n}^* = \underset{\mathbf{n}}{\operatorname{argmin}} \|U\mathbf{n}\|^2$$

= Singular vector corresponding to the smallest singular value of  $U$ .

= eigenvector corresponding to the smallest eigenvalue of  $U^T U$ .

Second moment.



With the eigenvector we got &

$d = a \cdot \mu_x + b \cdot \mu_y$ , the optimal line is

Optimal Line:  
 $-0.4937991695004691 \cdot x + 0.8695759772444539 \cdot y = 3.310040389017186$

## 2. Another Interpretation of Least Square Line Fitting.

o Assume that  $\|v\| = 1$  without loss of generality.

Set  $(v_1, \lambda_1), (v_2, \lambda_2)$  as eigen pairs of  $\Sigma$  where

$$\|v_1\| = \|v_2\| = 1, \quad v_1 \cdot v_2 = 0, \quad \lambda_1 > \lambda_2 > 0 \quad \& \quad v = c_1 v_1 + c_2 v_2.$$

Then,  $\|v\|^2 = c_1^2 + c_2^2 = 1.$

$$\operatorname{Var}(v) = v^T \Sigma v = (c_1 v_1^T + c_2 v_2^T)(c_1 \Sigma v_1 + c_2 \Sigma v_2)$$

$$= (c_1 v_1^T + c_2 v_2^T)(c_1 \lambda_1 v_1 + c_2 \lambda_2 v_2) = c_1^2 \lambda_1 + c_2^2 \lambda_2 = c_1^2 (\lambda_1 - \lambda_2) + \lambda_2$$

So,  $\operatorname{Var}(v)$  is maximum when  $c_1^2 = 1 \leftarrow v = v_1.$

:  $v$  that maximizes  $\operatorname{Var}(v)$  is the principal eigenvector of  $\Sigma$ .

$$u_i (u_i)^T = \begin{bmatrix} x_i - \tilde{x} \\ y_i - \tilde{y} \end{bmatrix} \begin{bmatrix} x_i - \tilde{x} & y_i - \tilde{y} \end{bmatrix} = \begin{bmatrix} (x_i - \tilde{x})^2 & (x_i - \tilde{x})(y_i - \tilde{y}) \\ (x_i - \tilde{x})(y_i - \tilde{y}) & (y_i - \tilde{y})^2 \end{bmatrix}$$

$$\Rightarrow \Sigma = \frac{1}{n} \sum_{i=1}^n u_i (u_i)^T = \frac{1}{n} \begin{bmatrix} \sum_{i=1}^n (x_i - \tilde{x})^2 & \sum_{i=1}^n (x_i - \tilde{x})(y_i - \tilde{y}) \\ \sum_{i=1}^n (x_i - \tilde{x})(y_i - \tilde{y}) & \sum_{i=1}^n (y_i - \tilde{y})^2 \end{bmatrix} = \frac{1}{n} U^T U.$$

$\Rightarrow \mathbf{n}$  = eigenvector corresponding to the smallest eigenvalue of  $U^T U (= n\Sigma)$   
 =  $v_2$ . which is vertical to  $v$ .

$\Sigma = \frac{1}{n} U^T U, \quad \mathbf{n} \text{ and } v \text{ are perpendicular.}$

### 3. Homography and Image Stitching

#### Part 1: Feature Extraction & Matching

```
def evenly_distributed_SIFT(img, img_size=256, patch_size=128, feature_num=256):  
    """  
    Get evenly distributed feature points over the given image using SIFT.  
    Extract feature points from small patches, merge them, and return them.  
  
    Parameters  
    img: Square Image to extract feature points.  
    img_size: Size of a side of img.  
    patch_size: Size of a side of a single patch.  
    feature_num: The number of features to extract.  
  
    Returns  
    kp_list: List of keypoints.  
    des_list: Numpy of descriptors.  
    """  
    n = img_size // patch_size  
    nfeatures = feature_num // (n**2)  
    sift = cv2.SIFT_create(nfeatures=nfeatures, contrastThreshold=0)  
  
    kp_list = []  
    des_list = []  
    for i in range(n):  
        for j in range(n):  
            y, x = i * patch_size, j * patch_size  
            img_patch = img[y:y+patch_size, x:x+patch_size]  
            kp, des = sift.detectAndCompute(img_patch, None)  
            if des is None: continue  
            kp, des = kp[:nfeatures], des[:nfeatures]  
  
            for keypoint in kp:  
                x, y = keypoint.pt  
                x, y = x + j * patch_size, y + i * patch_size  
                keypoint.pt = (x, y)  
  
            kp_list += kp  
            des_list.append(des)  
  
    des_list = np.concatenate(des_list, axis=0)  
    return kp_list, des_list
```

→ Get a grey image and extract feature points using SIFT. The feature points are need to be evenly distributed over the image. Therefore, input image is divided into small patches and SIFT is performed on the patches respectively. By setting the patch size to 64 or 128, the feature points could be well distributed in the image.

```
def BF_matching(kp1, des1, kp2, des2, match_num):  
    """  
    Brute Force matching with ratio sorting.  
    Do Brute Force matching and sort them by the ratio of the best distance and second-best one.  
  
    Parameters  
    kp1, kp2: Keypoints from img1 & img2.  
    des1, des2: Descriptors from img1 & img2.  
    match_num: The number of matches to return.  
  
    Returns  
    sorted_match: Best matches. (# sorted_match == match_num)  
    """  
    bf = cv2.BFMatcher()  
    match = bf.knnMatch(des1, des2, k=2)  
    ratio = [m.distance / n.distance for m, n in match]  
    match = np.array([m for m, n in match])  
    idx = np.argsort(ratio)[:match_num]  
    sorted_match = match[idx]  
    return sorted_match
```

→ In order to get correspondences between two images, Brute-Force Matching can be applied. Using all of the matches made the running time of RANSAC too long and its performance was poor. Therefore, instead of using all, it would be a good alternative to choose some matches which have good uniqueness. The ratio of the distance between the best match and second-best one is used as an indicator of uniqueness.

## Part 2: Direct Linear Transform (DLT)

```
def DLT(x1, x2):  
    """  
    Direct Linear Transform using SVD method.  
  
    Parameters  
    x1, x2: Paired corresponding 2D points.  
    Returns  
    H: Homography from x1 to x2.  
    """  
    n = x1.shape[0]  
    x1 = np.concatenate([x1, np.ones((n, 1))], axis=1)  
  
    A = np.zeros((2*n, 9))  
    A[range(0, 2*n, 2), 3:6] = x1  
    A[range(1, 2*n, 2), 0:3] = x1  
    A[range(0, 2*n, 2), 6:9] = -x2[:, 1:2] * x1  
    A[range(1, 2*n, 2), 6:9] = -x2[:, 0:1] * x1  
  
    U, D, V = np.linalg.svd(A)  
    H = V[-1].reshape(3, 3)  
    return H
```

→ Direct Linear Transform to get homography matrix. Because homography has 9 parameters and 8 degrees of freedom, equal to or more than 4 correspondences are needed. Construct matrix A, solve  $h^* = \min_h ||Ah||$  using SVD method.

## Part 3: RANSAC

```
def warp(x, H):  
    """  
    Parameters  
    x: Points to warp.  
    H: Homography of the warp.  
    Returns  
    ret: Warped points.  
    """  
    x = np.concatenate([x, np.ones((x.shape[0], 1))], axis=1)  
    y = np.dot(x, H.T)  
    ret = np.divide(y[:, 0:2], y[:, 2:3])  
    return ret
```

→ Before entering RANSAC, warp function is implemented for simple implementation. It warps x according to homography H.

```
class RANSAC:  
    """  
    Random Sample Consensus for automatic computation of a homography matrix.  
  
    Initial parameters  
    pts1, pts2: Paired corresponding 2D points.  
    """  
    def __init__(self, pts1, pts2, **kwargs):  
        self.num = pts1.shape[0]  
        self.pts1 = pts1  
        self.pts2 = pts2  
  
        self.t = kwargs['t']  
        self.p = kwargs['p']  
        self.s = kwargs['s']  
        self.max_iter = kwargs['max_iter']  
        self.opt_estimation = kwargs['opt_estimation']  
  
    def go(self):  
        """  
        Main runner.  
        Do robust estimation. If opt_estimation is true, optimal estimation too.  
  
        Returns  
        H: Homography matrix using RANSAC.  
        error: Mean symmetric transfer error of H.  
        """  
        H = self.robust_estimation()  
        if self.opt_estimation:  
            H = self.optimal_estimation(H)  
        error = np.mean(self.dist_of_all_match(H))  
        print("RANSAC Successfully Terminated!")  
        return H, error
```

→ Initializer and main runner of RANSAC class. RANSAC class requires inputs of matched points and hyperparameters for setting. Function 'go' runs robust estimation as default, and optimal estimation optionally. It returns homography with mean symmetric transfer error.

```
def robust_estimation(self):
    """
    Robust Estimation with Adaptive determination of the number of samples for RANSAC.
    """
    opt_H = None
    opt_cnt = 0

    N = np.inf
    sample_count = 0
    while sample_count < self.max_iter and sample_count < N:
        H = self.DLT_step()
        dist = self.dist_of_all_match(H)
        in_cnt = np.sum(dist < self.t)
        if in_cnt > opt_cnt:
            opt_H = H
            opt_cnt = in_cnt

        N = self.adaptive_determination(in_cnt)
        sample_count += 1
    print(f"> Robust Estimation terminates at iter {sample_count} with {opt_cnt} inliers")
    return opt_H
```

→ Robust estimation function in RANSAC class. Select 4 random matches and compute homography using DLT\_step function described below. Compute the number of inliers using symmetric transfer error. Adaptive determination method for the number of samples is applied. Returns the homography with the largest number of inliers.

```
def optimal_estimation(self, opt_H):
    """
    Optimal Estimation by minimizing cost function with DLT.
    """
    dist = self.dist_of_all_match(opt_H)
    idx = np.where(dist < self.t)[0]
    opt_cnt = idx.shape[0]
    sample_count = 0
    while True:
        H = self.DLT_step(idx)
        dist = self.dist_of_all_match(H)
        idx = np.where(dist < self.t)[0]
        in_cnt = idx.shape[0]
        sample_count += 1

        if in_cnt <= opt_cnt: break
        opt_cnt = in_cnt
        opt_H = H
    print(f"> Optimal Estimation terminates at iter {sample_count} with {opt_cnt} inliers")
    return opt_H
```

→ Optimal estimation function in RANSAC class. Re-estimate homography from all inliers and determine more matches using the homography calculated. It iterates these two steps until the number of inliers does not increase anymore. Returns the homography with the largest number of inliers.

```

def DLT_step(self, idx=None):
    """
    Use DLT function implemented above to guess homography matrix.
    """
    if idx is None:
        idx = np.random.choice(self.num, 4, replace=False)
    x1 = self.pts1[idx]
    x2 = self.pts2[idx]
    H = DLT(x1, x2)
    return H

def dist_of_all_match(self, H):
    """
    Return symmetric transfer error of the given homography H.
    """
    try:
        H_inv = np.linalg.inv(H)
        x1 = self.pts1
        x2 = self.pts2
        y1 = warp(x1, H)
        y2 = warp(x2, H_inv)
        ds = np.sum(np.square(x1 - y2) + np.square(x2 - y1), axis=1)
    except:
        ds = np.ones(self.num) * np.inf
    return ds

def adaptive_determination(self, in_cnt):
    """
    Adaptive determination of the number of samples for RANSAC.
    """
    if in_cnt == 0: return np.inf
    eps = 1 - in_cnt / self.num
    return np.log(1 - self.p) / np.log(1 - (1-eps)**self.s)

```

→ Util functions in RANSAC class.

'DLT\_step': returns homography using DLT function. If index is not given, use random 4 matches, otherwise, use the matches of given indices.

'dist\_of\_all\_match': returns symmetric transfer error of given homography. If the homography is not invertible, it returns an array filled with infinity.

'adaptive\_determination': manages the number of samples for RANSAC. It uses the equation below.

$$N = \frac{\log(1 - p)}{\log(1 - (1 - \epsilon)^s)}$$

where  $\epsilon = 1 - \frac{\text{number of inliers}}{\text{total number of points}}$  and  $p, s$  are given.

```

def HbyRANSAC(kp1, kp2, match, **kwargs):
    """
    Get homography using RANSAC class.

    Parameters
    kp1, kp2: Keypoints of the images.
    match: Match between kp1, kp2.

    Returns
    Homography matrix by RANSAC using (kp1, kp2, match).
    """
    pts1 = np.array([kp.pt for kp in kp1])
    pts2 = np.array([kp.pt for kp in kp2])
    match = np.array([[mm.queryIdx, mm.trainIdx] for mm in match])
    matched_pts1 = pts1[match[:, 0]]
    matched_pts2 = pts2[match[:, 1]]
    ransac = RANSAC(matched_pts1, matched_pts2, **kwargs)
    return ransac.go()

```

→ Receives keypoints of two images and its matching. Returns homography using RANSAC class with matched keypoints.

## Part 4: Warp & Merge

```
def warp_by_H(from_img, to_img, H, sz):  
    """  
    Warp (from_img) to (to_img) with homography (H).  
    Warp is performed on an image with each side tripled.  
  
    Parameters  
        from_img, to_img: image to warp from/to.  
        H: Homography of the warp.  
        sz: Size of a side of img.  
    Returns  
        to_img: Warped and merged image.  
    """  
    warp_img = np.zeros((3*sz, 3*sz, 3))  
    H_inv = np.linalg.inv(H)  
  
    idx = np.arange(3*sz)  
    x, y = np.meshgrid(idx, idx)  
    x, y = x.reshape(-1), y.reshape(-1)  
    res_pts = np.stack([x, y], axis=1)  
    pts = warp(res_pts - sz, H_inv)  
    xx, yy = pts[:, 0], pts[:, 1]  
  
    i, j = np.floor(yy).astype(int), np.floor(xx).astype(int)  
    a = np.expand_dims(yy - i, -1)  
    b = np.expand_dims(xx - j, -1)  
  
    def get_inside(fi, fj):  
        ok = (fi >= 0) & (fi < sz) & (fj >= 0) & (fj < sz)  
        fi, fj = np.where(ok, fi, 0), np.where(ok, fj, 0)  
        return np.where(np.expand_dims(ok, -1), from_img[fi, fj], 0)  
  
    warp_img[y, x] += get_inside(i, j) * (1-a)*(1-b)  
    warp_img[y, x] += get_inside(i+1, j) * a*(1-b)  
    warp_img[y, x] += get_inside(i, j+1) * (1-a)*b  
    warp_img[y, x] += get_inside(i+1, j+1) * a*b  
  
    warp_img = warp_img.astype(np.uint8)  
    to_img = np.where(to_img > warp_img, to_img, warp_img)  
    return to_img
```

→ Warping function that transforms from\_img to to\_img using homography given. Backward warping is applied as a vectorized version using inverse homography matrix. Size of a side of warped image is triple of the original image.

```
def warp_and_merge(img1, img2, img3, H12=None, H32=None, sz=256):  
    """  
    Merge images using warp_by_H function. If homography is None, warp is not performed.  
  
    Parameters  
        img1, img2, img3: Images to merge.  
        H12, H32: Homography of the warp.  
        sz: Size of one sides of the three images which are same.  
    Returns  
        result_img: Merged image.  
    """  
    result_img = np.zeros((3*sz, 3*sz, 3), dtype=np.uint8)  
    if H12 is not None: result_img = warp_by_H(img1, result_img, H12, sz)  
    if H32 is not None: result_img = warp_by_H(img3, result_img, H32, sz)  
    result_img[sz:2*sz, sz:2*sz] = img2  
    return result_img
```

→ Using 'warp\_by\_H' function, it warps and merges two side images into the center image.

## Main function

```
def main(path1, path2, path3, **kwargs):
    img_size = kwargs['img_size']
    patch_size = kwargs['patch_size']
    feature_num = kwargs['feature_num']
    match_num = kwargs['match_num']

    # Three Panoramic Pictures
    color_img1 = cv2.imread(path1)
    color_img2 = cv2.imread(path2)
    color_img3 = cv2.imread(path3)
    title = "Three Panoramic Pictures"
    imshow(title, [color_img1, color_img2, color_img3], '3_color')

    # Feature Extraction
    img1 = cv2.cvtColor(color_img1, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(color_img2, cv2.COLOR_BGR2GRAY)
    img3 = cv2.cvtColor(color_img3, cv2.COLOR_BGR2GRAY)
    kp1, des1 = evenly_distributed_SIFT(img1, img_size, patch_size, feature_num)
    kp2, des2 = evenly_distributed_SIFT(img2, img_size, patch_size, feature_num)
    kp3, des3 = evenly_distributed_SIFT(img3, img_size, patch_size, feature_num)
    title = "Feature Extraction"
    sift_img1 = cv2.drawKeypoints(img1, kp1, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    sift_img2 = cv2.drawKeypoints(img2, kp2, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    sift_img3 = cv2.drawKeypoints(img3, kp3, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    imshow(title, [sift_img1, sift_img2, sift_img3], '3_gray')

    # Feature Matching
    match12 = BF_matching(kp1, des1, kp2, des2, match_num)
    match32 = BF_matching(kp3, des3, kp2, des2, match_num)
    title = "Best 80 Feature Matching btw 1-2 & btw 2-3"
    match_img12 = cv2.drawMatches(img1, kp1, img2, kp2, match12[:80], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    match_img32 = cv2.drawMatches(img3, kp3, img2, kp2, match32[:80], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    imshow(title, [match_img12, match_img32], '2_rec')

    # Homography Estimation using RANSAC
    print("H12 by RANSAC Running...")
    H12, error12 = HbyRANSAC(kp1, kp2, match12, **kwargs)
    print("\n\nHomography Matrix from 1 to 2:")
    print(H12)
    print(f"Mean Symmetric Transfer Error: {error12:.2f} \n\n")

    print("H32 by RANSAC Running...")
    H32, error32 = HbyRANSAC(kp3, kp2, match32, **kwargs)
    print("\n\nHomography Matrix from 3 to 2:")
    print(H32)
    print(f"Mean Symmetric Transfer Error: {error32:.2f} \n\n")

    # Warping Images & Merge
    merge_img12 = warp_and_merge(color_img1, color_img2, color_img3, H12, None, img_size)
    merge_img32 = warp_and_merge(color_img1, color_img2, color_img3, None, H32, img_size)
    title = "Pairwise matching Results 1-2 & 2-3"
    imshow(title, [merge_img12, merge_img32], '2_squ')

    result_img = warp_and_merge(color_img1, color_img2, color_img3, H12, H32, img_size)
    nz_y, nz_x, _ = np.where(result_img > 0)
    result_img = result_img[np.min(nz_y):np.max(nz_y), np.min(nz_x):np.max(nz_x)]
    title = "Final Mosaiced Image"
    imshow(title, [result_img], '1')
```

With all implemented functions, it runs whole procedure and show results.

1. Get three panoramic pictures ready.
2. Convert them into gray pictures and extract feature points.
3. With Brute-Force matching, get some correspondences which have good uniqueness.
4. Calculate homography between (1→2) and (3→2) using RANSAC class.
5. Warp and merge side images into the center image.
6. Show the non-zero rectangle cut of result image.



```

kwargs = {
    'img_size': 256,      # Size of a side of a image.
    'patch_size': 128,    # Size of a side of a single patch.
    'feature_num': 256,   # The number of extracted features.
    'match_num': 64,      # The number of matches to use.

    'max_iter': 100000,   # Maximum iteration of robust estimation in RANSAC
    'opt_estimation': True, # Use optimal estimation in RANSAC

    ## Parameters for Adaptive Determination
    't': 1.25,
    'p': 0.99,
    's': 4,
}

path1 = 'data/img1.jpg'
path2 = 'data/img2.jpg'
path3 = 'data/img3.jpg'
main(path1, path2, path3, **kwargs)

```

→ Run main function with hyperparameters above. **Result** is as follows:

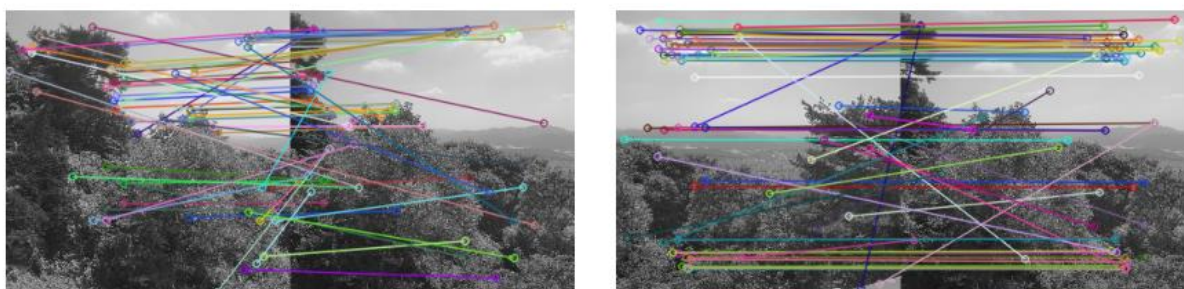
Three Panoramic Pictures



Feature Extraction



Feature Matching btw 1-2 & btw 2-3



```
H12 by RANSAC Running...
> Robust Estimation terminates at iter 1265 with 17 inliers
> Optimal Estimation terminates at iter 2 with 19 inliers
RANSAC Successfully Terminated!
```

```
Homography Matrix from 1 to 2:
[[-1.15095849e-02 -3.31397700e-04  9.64194688e-01]
 [-1.03713908e-03 -1.06820526e-02  2.64592093e-01]
 [-1.06690984e-05 -1.31689337e-06 -8.47734770e-03]]
Mean Symmetric Transfer Error: 20027.80
```

```
H32 by RANSAC Running...
> Robust Estimation terminates at iter 709 with 24 inliers
> Optimal Estimation terminates at iter 3 with 28 inliers
RANSAC Successfully Terminated!
```

```
Homography Matrix from 3 to 2:
[[-3.78193517e-03  1.31315230e-04 -9.96581055e-01]
 [ 1.32608816e-03 -6.01849514e-03 -8.20316254e-02]
 [ 1.10189864e-05 -4.94547116e-08 -6.68666186e-03]]
Mean Symmetric Transfer Error: 34959.65
```

## Pairwise matching Results 1-2 & 2-3



## Final Mosaiced Image

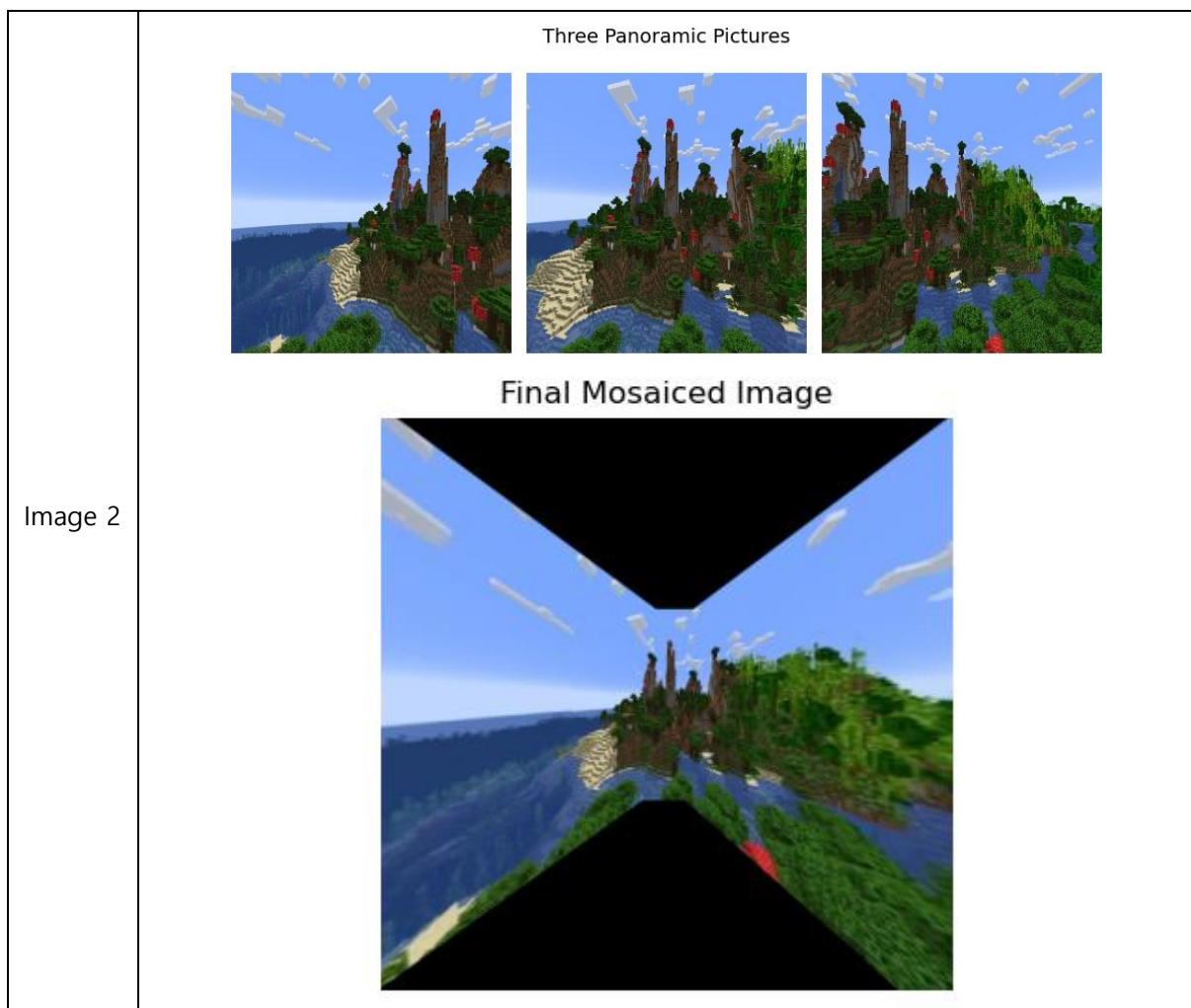


## Discussion





The three panoramic pictures are taken by the writer at Namhansanseong, and they are successfully mosaiced. The feature points are evenly distributed over the image with patch size of 128. Also, many parallel lines can be found in feature matching showing consistency. Using relatively small number of matches (about 16) allows running time of RANSAC to decrease without loss of performance. However, 64 matches are chosen between running time and stability of RANSAC.

From image 1 to image 2, RANSAC found 19 inliers with mean symmetric transfer error of 20027. From image 3 to image 2, RANSAC found 28 inliers with error of 34959. The reasons of the huge error could be (a) large outlier ratio and (b) the transferred point may be outside of the possible coordinates. Despite the large error, the algorithm above usually performs well for a variety of images, and some examples are shown in Appendix.

## Appendix





<p>Image 3</p>	<p>Three Panoramic Pictures</p> <div data-bbox="410 291 1318 582">Three individual panoramic photographs of a rocky area in a forest. Each photo shows a large, light-colored rock formation in the center, surrounded by green foliage and trees. The ground is covered with brown leaves and some small green plants.</div> <p>Final Mosaiced Image</p> <div data-bbox="566 627 1161 1003">A final mosaiced panoramic image of the rocky area. It shows a wide view of the rock formation, with the three individual photos seamlessly blended together. The image has a slight fisheye effect, curving at the top and bottom edges.</div>
<p>Image 4</p>	<p>Three Panoramic Pictures</p> <div data-bbox="410 1086 1318 1377">Three individual panoramic photographs of a forest path. Each photo shows a dirt path winding through a dense forest with tall trees and green foliage. The path is covered with fallen leaves and shadows from the trees.</div> <p>Final Mosaiced Image</p> <div data-bbox="576 1422 1152 1798">A final mosaiced panoramic image of the forest path. It shows a wide view of the path, with the three individual photos seamlessly blended together. The image has a slight fisheye effect, curving at the top and bottom edges.</div>