# ENSE 885AY

# Application of Deep Learning in Computer Vision

## Assignment A03

## Camera Calibration and Fundamental Matrix Estimation with RANSAC

**Instructed by**

**Dr. Kin-Choong Yow**

**Student:**

**Marzieh Zamani**

## List of Sections and Sub-sections

# 1. Introduction

## 1.1. Overview (Key points from the assignment description) [1]

**Assignment Subject:**

Camera Calibration and Fundamental Matrix Estimation with RANSAC

**Assignment objectives:**

- The goal of this project is to introduce you to camera and scene geometry
- Estimate the camera projection matrix, which maps 3D world coordinates to 2D image coordinates.
- Estimate the fundamental matrix, which relates points in one scene to epipolar lines in another perspective of the same scene.
- Estimating the fundamental matrix for the correspondences of two images using the RANSAC model-fitting algorithm.

**Steps to local feature matching between two images (image1 & image 2):**

1. Estimating the projection matrix:
   - ⇨ calculate_projection_matrix()
   - ⇨ calculate_camera_center()
2. Estimating the fundamental matrix:
   - ⇨ estimate_fundamental matrix()
3. Estimating the fundamental matrix with unreliable ORB matches using RANSAC:
   - ⇨ ransac_fundamental_matrix()

# 2. Student Code

## 2.1. Estimating the projection matrix (calculate_projection_matrix() & calculate_camera_center())

### Algorithm and implementation of calculate_projection_matrix() [1]:

The first step is to set up a system of equations using the corresponding 2D and 3D points:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \cong \begin{pmatrix} u * s \\ v * s \\ s \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

$$u = \frac{m_{11}X + m_{12}Y + m_{13}Z + m_{14}}{m_{31}X + m_{32}Y + m_{33}Z + m_{34}}$$

$$\rightarrow (m_{31}X + m_{32}Y + m_{33}Z + m_{34})u = m_{11}X + m_{12}Y + m_{13}Z + m_{14}$$

$$\rightarrow 0 = m_{11}X + m_{12}Y + m_{13}Z + m_{14} - m_{31}uX - m_{32}uY - m_{33}uZ - m_{34}u$$

$$v = \frac{m_{21}X + m_{22}Y + m_{23}Z + m_{24}}{m_{31}X + m_{32}Y + m_{33}Z + m_{34}}$$

$$\rightarrow (m_{31}X + m_{32}Y + m_{33}Z + m_{34})v = m_{21}X + m_{22}Y + m_{23}Z + m_{24}$$

$$\rightarrow 0 = m_{21}X + m_{22}Y + m_{23}Z + m_{24} - m_{31}vX - m_{32}vY - m_{33}vZ - m_{34}v$$

The above system can be transformed to following matrix form:

$A_{2N\times 11} * M_{11\times 1} = b_{2N\times 1}$

$$\begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_N & Y_N & Z_N & 1 & 0 & 0 & 0 & 0 & -u_NX_N & -u_NY_N & -u_NZ_N \\ 0 & 0 & 0 & 0 & X_N & Y_N & Z_N & 1 & -v_NX_N & -v_NY_N & -v_NZ_N \end{bmatrix}_{2N\times 11} \begin{bmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{14} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{24} \\ m_{31} \\ m_{32} \\ m_{34} \end{bmatrix}_{11\times 1}$$

$$= \begin{bmatrix} u_1 \\ v_1 \\ \vdots \\ u_N \\ v_N \end{bmatrix}_{2N\times 1}$$

There are two different rows in matrices A and b (row[2k] and row[2k+1]:

$$\begin{bmatrix} row_{2k}^{A} \\ row_{2k+1}^{A} \end{bmatrix}_{2N\times 11} M_{11\times 1} = \begin{bmatrix} row_{2k}^{b} \\ row_{2k+1}^{b} \end{bmatrix}_{2N\times 1}$$

where

$$\begin{bmatrix} row_{2k}^{A} \\ row_{2k+1}^{A} \end{bmatrix}_{2\times 11} = \begin{bmatrix} X_k & Y_k & Z_k & 1 & 0 & 0 & 0 & 0 & -u_kX_k & -u_kY_k & -u_kZ_k \\ 0 & 0 & 0 & 0 & X_k & Y_k & Z_k & 1 & -v_kX_k & -v_kY_k & -v_kZ_k \end{bmatrix}_{2\times 11}$$

$$\begin{bmatrix} row_{2k}^{b} \\ row_{2k+1}^{b} \end{bmatrix}_{2\times 1} = \begin{bmatrix} u_k \\ v_k \end{bmatrix}_{2\times 1}$$

Matrices A and b will be constructed using the corresponding 2D and 3D points as below:

Define constants and placeholders for matrices A and b

```
# Matrices A and b will be constructed using the corresponding 2D and 3D points as below

# Define constants and placeholders for matrices A and b
N = int(points_2d.shape[0])
b = np.zeros((2*N,1))
A = np.zeros((2*N,11))
```

For every row[k] in the corresponding 2D and 3D points

Construct row[2k] and row[2k+1] of matrices A and b

```
# For every row[k] in the corresponding 2D and 3D points
for k in range(0,N):
    row1 = 2*k
    row2 = 2*k+1

    # Construct row[2k] and row[2k+1] of matrices A and b
    b[row1,0] = points_2d[k,0]
    b[row2,0] = points_2d[k,1]

    A[row1,0:3] = points_3d[k,:]
    A[row1,3] = 1
    A[row1,8:11] = -points_2d[k,0]*points_3d[k,:]

    A[row2,4:7] = points_3d[k,:]
    A[row2,7] = 1
    A[row2,8:11] = -points_2d[k,1]*points_3d[k,:]
```

Solve A* $M_{11x1}$=b using *np.linalg.lstsq()* to obtain $M_{11x1}$

```
# Solve A*M_11 = b using np.linalg.lstsq() to obtain M_11
M11 = np.linalg.lstsq(A, b, rcond = None)[0]
```

Append $M_{34}$=1 to $M_{11x1}$ and reshape it as $M_{3x4}$

```
# Append M_34 = 1 to M_11 and reshape it as M_3x4
M12 = np.append(M11, [1])
M = M12.reshape((3, 4))
```

### Algorithm and implementation of calculate_camera_center() [1]:

Camera center is calculated using following equations:

Extract matrices Q and m4 from matrix M

$M = [M_{3\times 3}|M_{3\times 1}];$

$Q = M_{3\times 3}$

$m_4 = M_{3\times 1}$

```
# Extract matrices Q and m4 from matrix M
Q = M[0:3,0:3]
m4 = M[:,3]
```

Calculate camera center (matrix cc)

$cc = -Q^{-1}m_4$

```
# Calculate camera center (matrix cc)
cc = -np.linalg.inv(Q) @ m4
```

## 2.2. Estimating the fundamental matrix (estimate_fundamental matrix())

### Algorithm and implementation of estimate_fundamental matrix() [1]:

The first step is to set up a system of equations using the corresponding 2D points a and b:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0$$

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11}u + f_{12}v + f_{13} \\ f_{21}u + f_{22}v + f_{23} \\ f_{31}u + f_{32}v + f_{33} \end{pmatrix} = 0$$

$$\left( f_{11}uu' + f_{12}vu' + f_{13}u' + f_{21}uv' + f_{22}vv' + f_{23}v' + f_{31}u + f_{32}v + f_{33} \right) = 0$$

The above equation can be transformed to following matrix form:

$UV_{N\times 9} * F_{9\times 1} = 0_{N\times 1}$

$$[[u_k^A u_k^B]_{N\times 1} \quad [v_k^A u_k^B]_{N\times 1} \quad [u_k^B]_{N\times 1} \quad [u_k^A v_k^B]_{N\times 1} \quad [u_k^A v_k^B]_{N\times 1} \quad [v_k^B]_{N\times 1} \quad [u_k^A]_{N\times 1} \quad [u_k^A]_{N\times 1} \quad [1]$$

$$= \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}_{N\times 1}$$

Matrix UV will be constructed using the corresponding 2D points_a and points_b as below:

Set with_normalization to True or False

```
# Set with_normalization to True or False
with_normalization = True
```

If with_normalization = True: Normalize points_a and points_b and obtain their transformation matrices T_a and T_b

```python
# If with_normalization = True:
# Normalize points_a and points_b and obtain T_a and T_b
if with_normalization  ==  True:
    points_a, T_a = normalization_function(points_a)
    points_b, T_b = normalization_function(points_b)
```

Define constant N, and extract uA, vA, uB, vB, and I_N from points_a and points_

Construct matrix UV using uA, vA, uB, vB, and I_N and their multiplications

```python
# Extract uA, vA, uB, vB, and I_N from points_a and points_b
uA = points_a[:,0].reshape(N,1)
vA = points_a[:,1].reshape(N,1)
uB = points_b[:,0].reshape(N,1)
vB = points_b[:,1].reshape(N,1)
I_N = np.ones((N,1))

# Construct matrix UV using uA, vA, uB, vB, and I_N and their multiplications
UV = np.hstack((uA*uB, vA*uB, uB, uA*vB, vA*vB, vB, uA, vA, I_N ))
```

Solve UV* $F_{9x1} = 0$ using *np.linalg. svd ()* to obtain full-rank $F_{9x1}$

```python
# Solve UV* F9x1 = 0 to obtain full-rank F9x1
_, _, vh0 = np.linalg.svd(UV,full_matrices = True)
F3 = vh0[8,:].reshape(3,3)
```

Reduce F rank from 3 to 2

```python
# Reduce F rank from 3 to 2
u3, s3, vh3 = np.linalg.svd(F3, full_matrices = True)
s3[2] = 0
s2 = np.diag(s3)
F = np.dot(u3, np.dot( s2, vh3))
```

If with_normalization = True:  Transform F_norm to F_orig

$$F_{orig} = T_b^T * F_{norm} * T_a$$

```python
# If with_normalization = True:  Transform F_norm to F_orig
if with_normalization  ==  True:
    F = np.dot( np.transpose( T_b ), np.dot( F, T_a ))
```

## Algorithm and implementation of normalization_function() [1]:

Define constant N

```
# Define constant N
N = points_a.shape[0]
```

Obtain center of points

$$center_a = [\overline{u_a} \quad \overline{v_a}]$$

```
# Obtain center of points
center_a = np.average(points_a,axis = 0)
```

Center points around the center

$$centered\_a_{N\times 2} = [u_a \quad v_a]_{N\times 2} - [\overline{u_a} \quad \overline{v_a}] = [u_a - \overline{u_a} \quad v_a - \overline{v_a}]_{N\times 2}$$

```
# Center points around the center
centered_a = points_a-center_a.reshape(1,2)
```

Obtain standard deviation

$$std_a = \sqrt{\frac{([u_a - \overline{u_a} \quad v_a - \overline{v_a}]_{N\times 2})^2}{2N}}$$

```
# Obtain standard deviation
std_a = np.sqrt( (np.sum((centered_a)**2,axis = None)/(2*N)) )
```

Calculate scale

$$scale_a = 1/std_a$$

```
# Calculate scale
scale_a = 1/std_a
```

Obtain scaled and centered points_a

$$scaled - centered_a = scale_a * centered\_a_{N\times 2}$$

```
# Obtain scaled and centered points_a
scaled_centered_a = scale_a*centered_a
```

Obtain Ts and Tc matrices

$$Ts_a = \begin{bmatrix} scale_a & 0 & 0 \\ 0 & scale_a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$Tc_a = \begin{bmatrix} 1 & 0 & -\overline{u_a} \\ 0 & 1 & \overline{v_a} \\ 0 & 0 & 1 \end{bmatrix}$$

```
# Obtain Ts and Tc matrices
Ts_a = np.diag(np.array([scale_a, scale_a, 1]))

Tc_a = np.diag(np.array([1., 1., 1.]))
Tc_a[0,2] = -center_a[0]
Tc_a[1,2] = -center_a[1]
```

Calculate matrix T

$$T_a = Ts_a * Tc_a$$

```
# Calculate matrix T
T_a = Ts_a @ Tc_a
```

## 2.3. Estimating the fundamental matrix with unreliable ORB matches using RANSAC (ransac_fundamental_matrix())

### Algorithm and implementation of ransac_fundamental_matrix() [1]:

Define constants

```
# Define constants
N = matches_a.shape[0]
max_iter = 15000
threshold = 0.02
batch_size = 8
```

Define random indexes with

Range = 0 : N = number of points

Size = max_iter x batch_size = 15000 x 8

```
# Define random indexes with size = (max_iter, batch_size)
rand_idx = np.random.randint(N,size = (max_iter, batch_size))
```

Obtain UV matrix (algorithm described in previous section)

```
# Obtain UV matrix (algorithm described in previous section)
# Extract uA, vA, uB, vB, and I_N from points_a and points_b
uA = matches_a[:,0].reshape(N,1)
vA = matches_a[:,1].reshape(N,1)
uB = matches_b[:,0].reshape(N,1)
vB = matches_b[:,1].reshape(N,1)
I_N = np.ones((N,1))

# Construct matrix UV using uA, vA, uB, vB, and I_N and their multiplications
UV = np.hstack((uA*uB, vA*uB, uB, uA*vB, vA*vB, vB, uA, vA, I_N ))
```

Define placeholder for inlier_count

```
# Define placeholders
inlier_count = np.zeros(max_iter)
```

Iteration for k=0:max_iter

Estimate F matrix using random batch[k] of 8 pairs of points

Calculate cost of estimated F using following equation

$$cost = abs(UV_{N\times 9} * F_{9\times 1} - 0_{N\times 1})$$

$$cost = abs([[u_k^A u_k^B]_{N\times 1} \quad [v_k^A u_k^B]_{N\times 1} \quad [u_k^B]_{N\times 1} \quad [u_k^A v_k^B]_{N\times 1} \quad [u_k^A v_k^B]_{N\times 1} \quad [v_k^B]_{N\times 1} \quad [u_k^A]_{N\times 1} \quad [u_k^A]_{N\times}$$

$$* \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{34} \end{bmatrix}_{9\times 1} \quad )$$

Obtain and save number of inliers (with cost < threshold)

```
# Iteration for k=0:max_iter
```

```python
    for k in range(max_iter):

        # Estimate F matrix using random batch[k] of 8 pairs of points
        F = estimate_fundamental_matrix(matches_a[rand_idx[k,:],:],matches_b[rand_idx[k,:],:]
)

        # Calculate cost of estimated F using following equation
        cost_k = np.abs( UV @ F.reshape((9,1)) )

        # Obtain and save number of inliers (with cost < threshold)
        inlier_idx = cost_k < threshold
        inlier_count[k] = np.sum(inlier_idx)
```

Sort inlier_count from maximum to minimum

```python
    # Sort inlier_count from maximum to minimum
    sort_idx = np.argsort(-inlier_count)
```

Obtain best_batch with maximum inliers

```python
    # Obtain best_batch with maximum inliers
    best_idx = sort_idx[0]
    best_batch = rand_idx[best_idx,:]
```

Estimate best_F matrix using best_batch

```python
    # Estimate best_F matrix using best_batch
    best_F = estimate_fundamental_matrix(matches_a[best_batch,:], matches_b[best_batch,:])
```

Calculate best_cost of estimated best_F

```python
    # Calculate best_cost of estimated best_F
    best_cost = np.abs( UV @ best_F.reshape((9,1)) )
```

Obtain and save number of inliers (with best_cost < threshold)

```python
    # Obtain and save number of inliers (with best_cost < threshold)
    inlier_idx = best_cost < threshold
    best_inlier_count = np.sum(inlier_idx)
```

Sort cost from minimum (best match) to maximum (worst match)

```python
# Sort cost from minimum (best match) to maximum (worst match)
index=np.argsort(best_cost[:,0])[:50]
```

Obtain best pairs of matching points with minimum cost

```python
# Obtain best pairs of matching points with minimum cost
inliers_a=matches_a[index,:]
inliers_b=matches_b[index,:]
```

Print results

```python
# Print results
print('Found', best_inlier_count,'inliers / ', N, 'points')
print('inliers / total points :', int(100*best_inlier_count/N), '%')
```

# 3. Results and Discussion

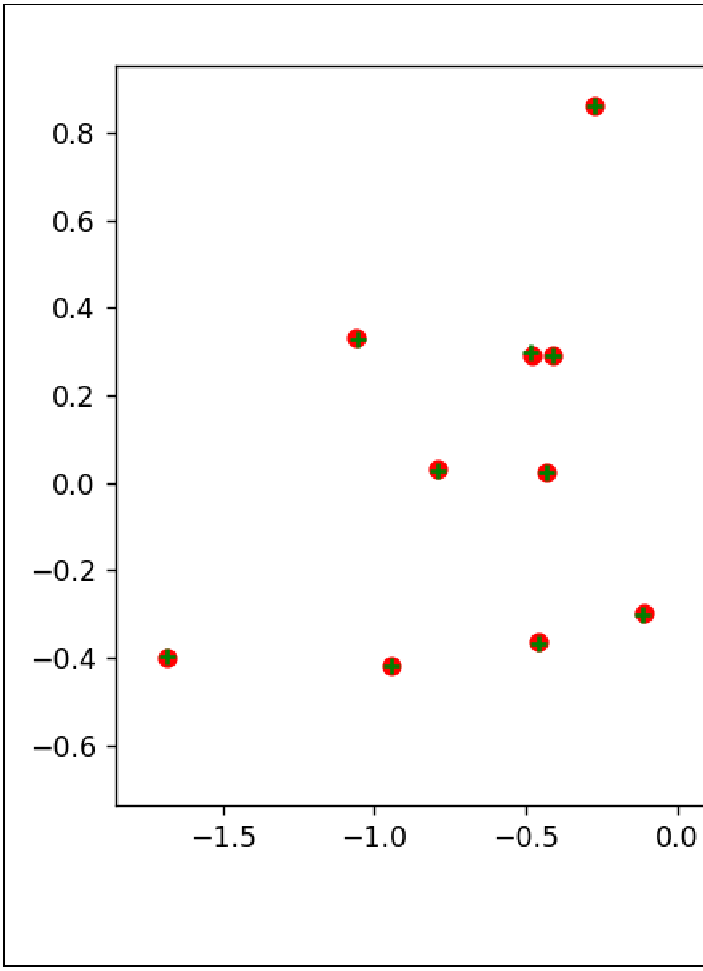
## 3.1. Estimation of the projection matrix and camera center

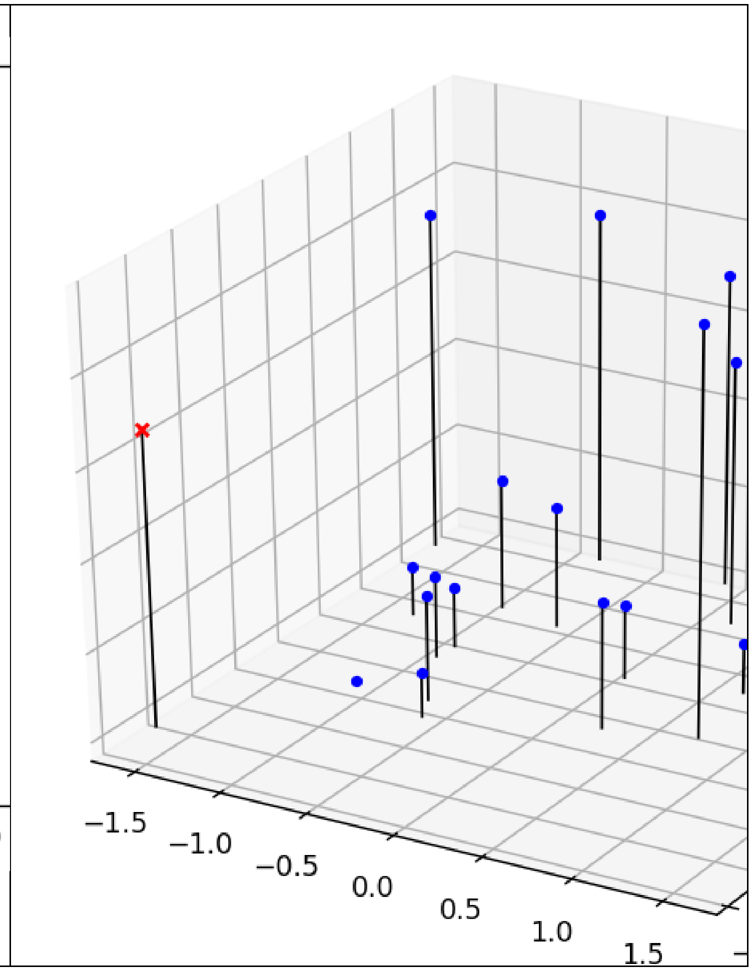
**Table 1: Estimation of the projection matrix and camera center**

| Projection Matrix | Total Residual | Camera Center |
|---|---|---|
| **[ 0.76785834 -0.49384797 -0.02339781 0.00674445]**<br>**[-0.0852134  -0.09146818 -0.90652332 -0.08775678]**<br>**[ 0.18265016  0.29882917 -0.07419242  1.        ]** | **0.044535** | **<-1.5126, -2.3517, 0.2827>** |

**Table 2: Point Visualization and Point 3D View**

| Point Visualization | Point 3D View |
|---|---|



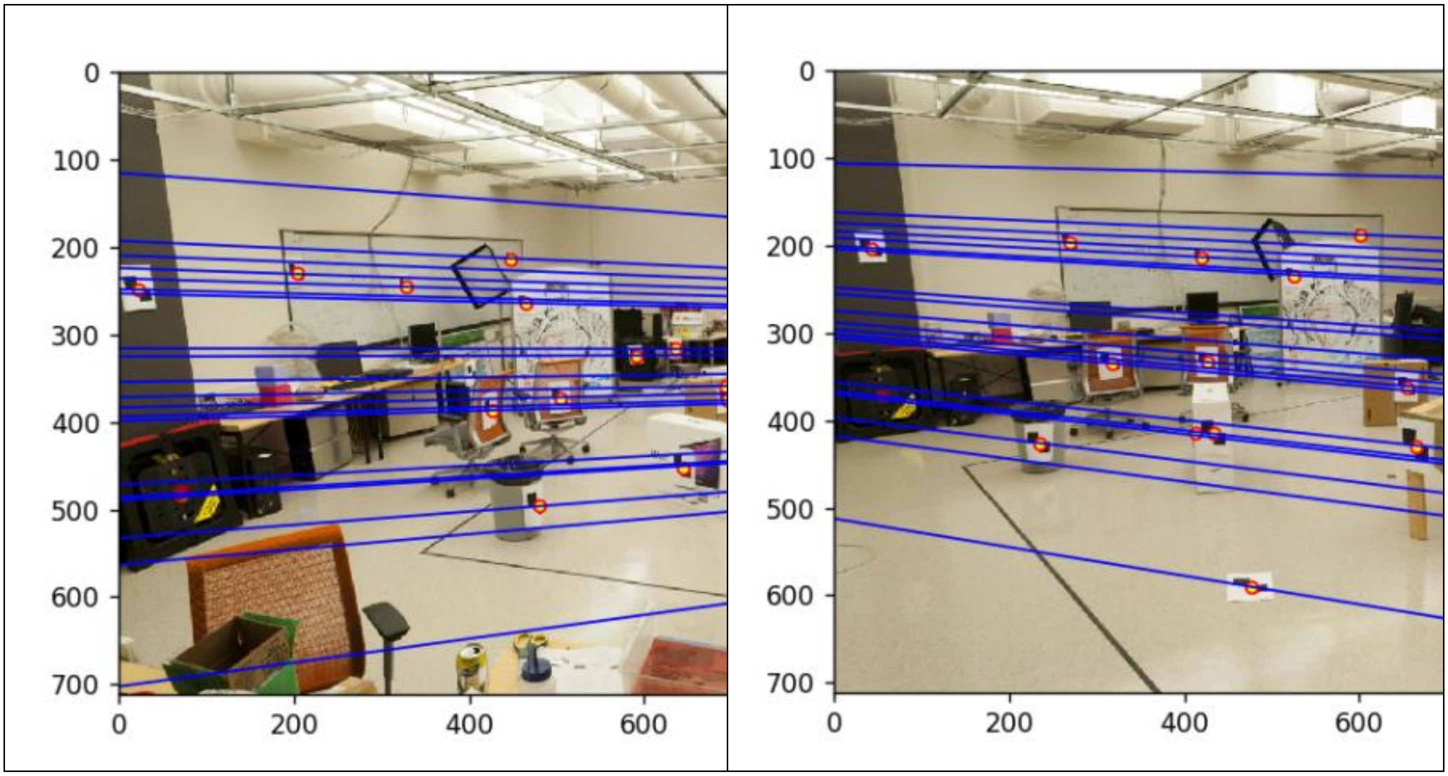




## 3.2. Estimation of the fundamental matrix

Table 3: Estimation of the fundamental matrix

| Fundamental Matrix without Normalization | Fundamental Matrix with Normalization |
|---|---|
| ([[-5.36264198e-07, 7.90364771e-06, -1.88600204e-03],<br>[ 8.83539184e-06, 1.21321685e-06, 1.72332901e-02],<br>[-9.07382264e-04, -2.64234650e-02, 9.99500092e-01]]) | ([[-1.17248591e-07, 1.60824663e-06, -4.01980786e-04],<br>[ 1.11212887e-06, -2.73443755e-07, 3.23319884e-03],<br>[-2.36400817e-05, -4.44404958e-03, 1.03455561e-01]]) |

Table 4: Estimation of the fundamental matrix




## 3.3. Estimation of the fundamental matrix with unreliable ORB matches using RANSAC

For four pairs of images, fundamental matrix was estimated with / without normalization. The results are summarized in following table:

For all images expect Woofruff, normalization has a positive effect on number of inliers.

Table 5: Inlier count for fundamental estimated with / without normalization

| | | **inliers / total points** | **inliers / total points %** |
|---|---|---|---|
| **Notre Dame** | Without normalization | 626 / 1282 | 48 % |
| | With normalization | 1043 / 1282 | 81 % |

| | | | |
|---|---|---|---|
| **Mount Rushmore** | Without normalization | 519 / 1177 | 44 % |
| | With normalization | 547 / 1177 | 46 % |
| **Gaudi** | Without normalization | 350 / 1037 | 33 % |
| | With normalization | 478 / 1037 | 46 % |
| **Woodruff** | Without normalization | 506 / 1137 | 44 % |
| | With normalization | 424 / 1137 | 37 % |

### Table 6: Result for Notre Dame without Normalization

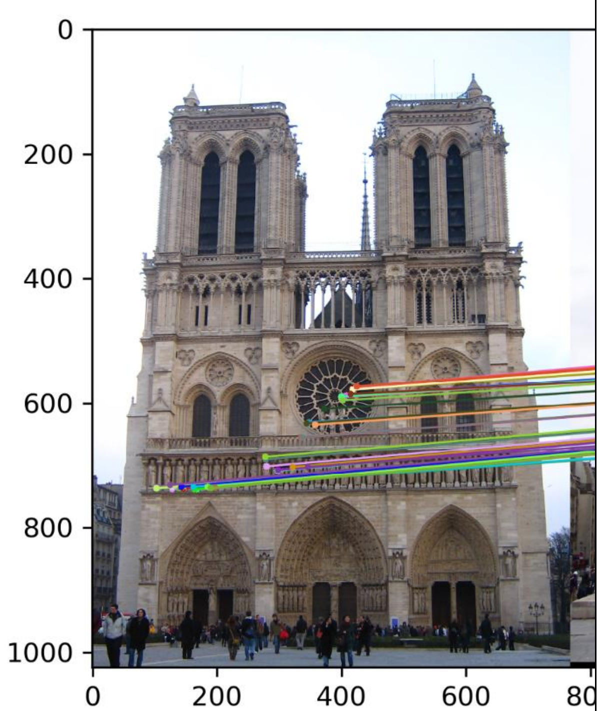
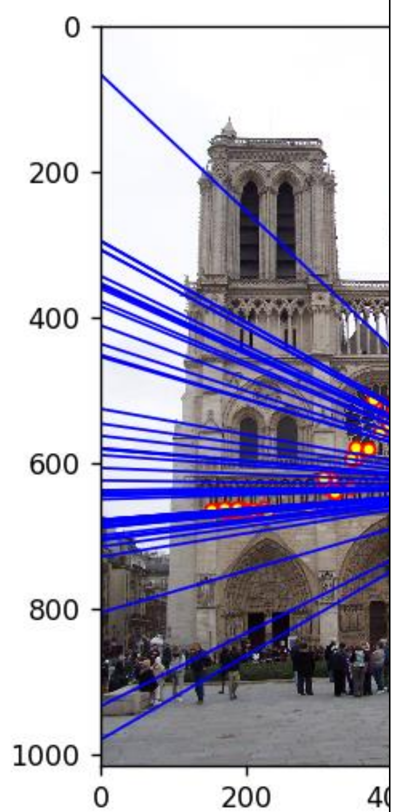
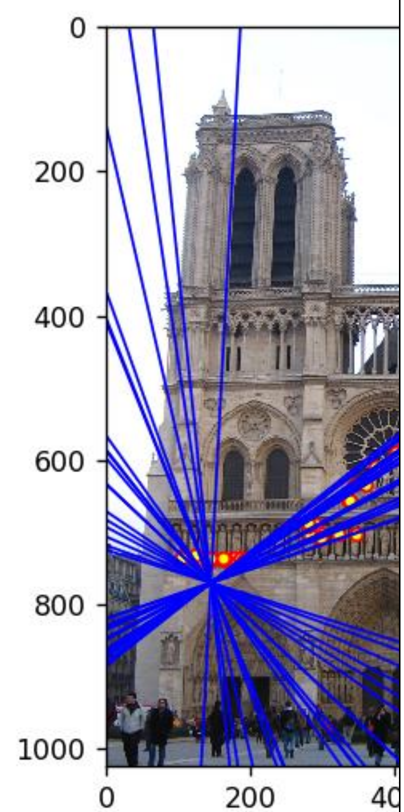
| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|
|   |   |   |

**Table 7: Result for Notre Dame with Normalization**

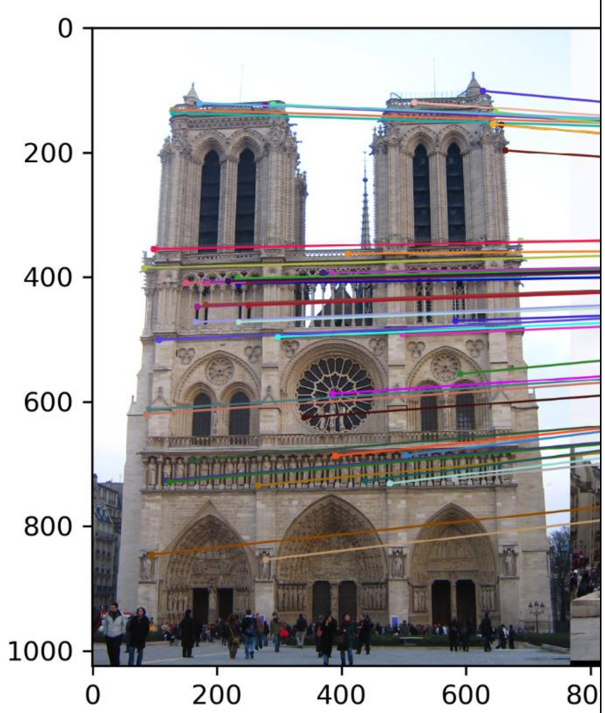
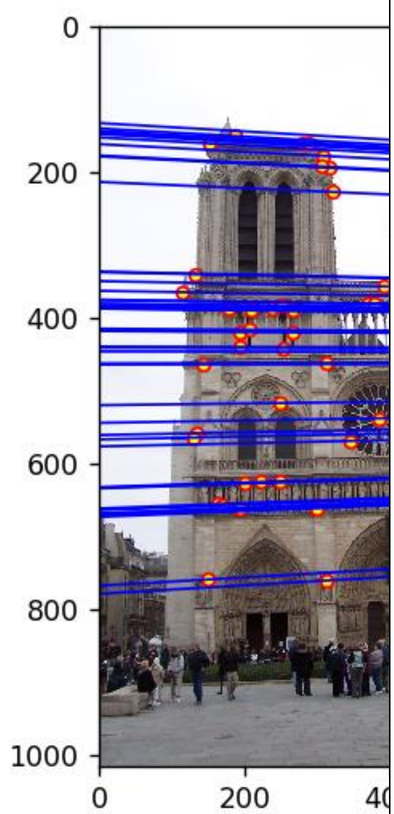
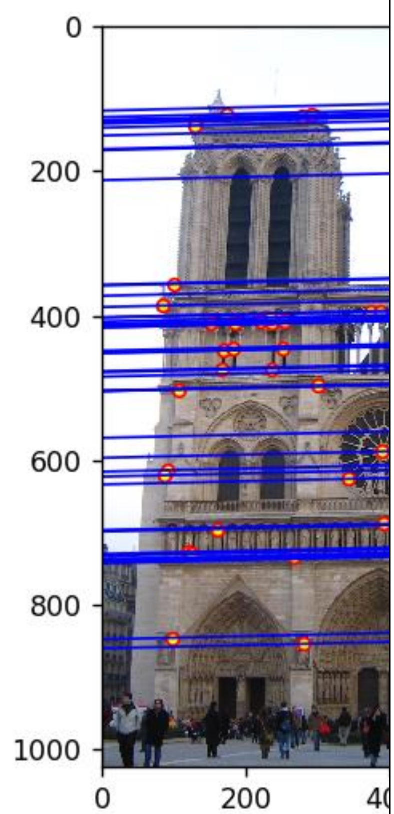
| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|
|  |  |  |

**Table 8: Results for Mount Rushmore without Normalization**

| Matching points between Image_a & Image_b | Epipo |
|---|---|

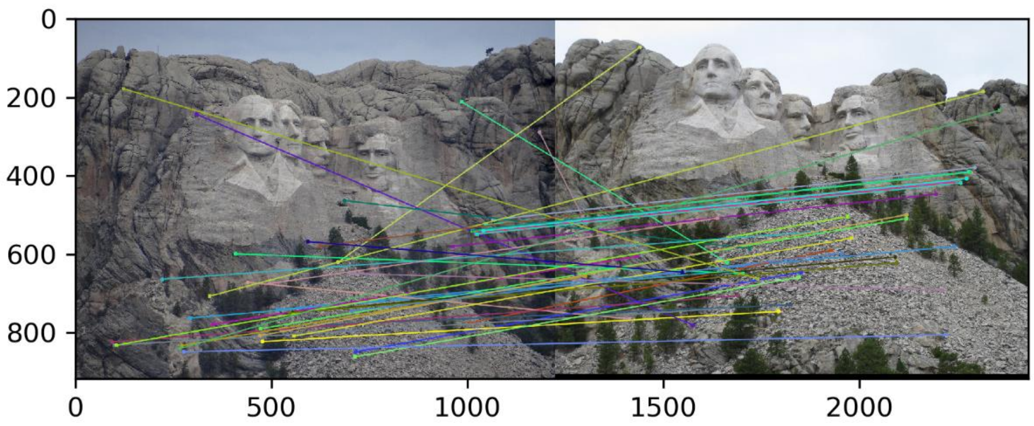
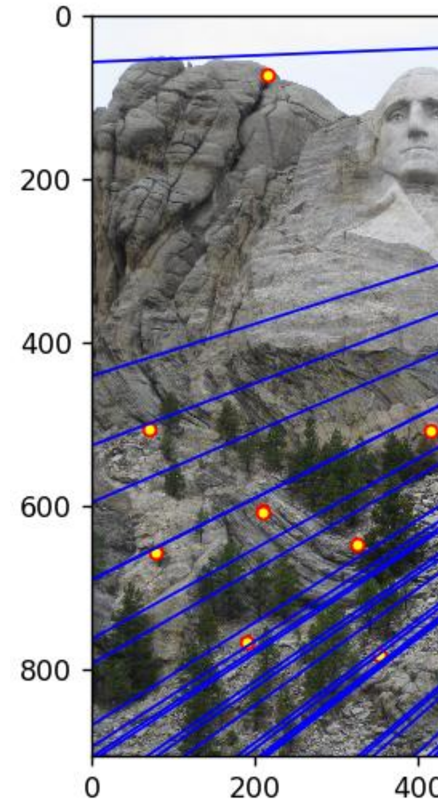
| | |
|---|---|
|  |  |





**Table 9: Results for Mount Rushmore with Normalization**

| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|

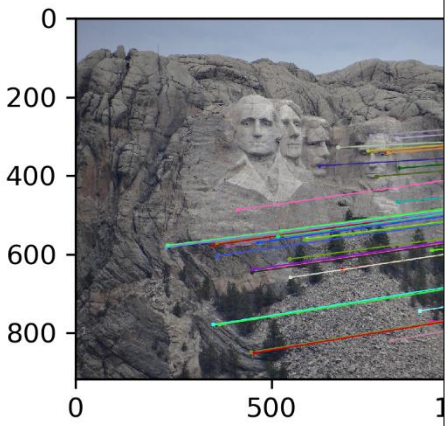
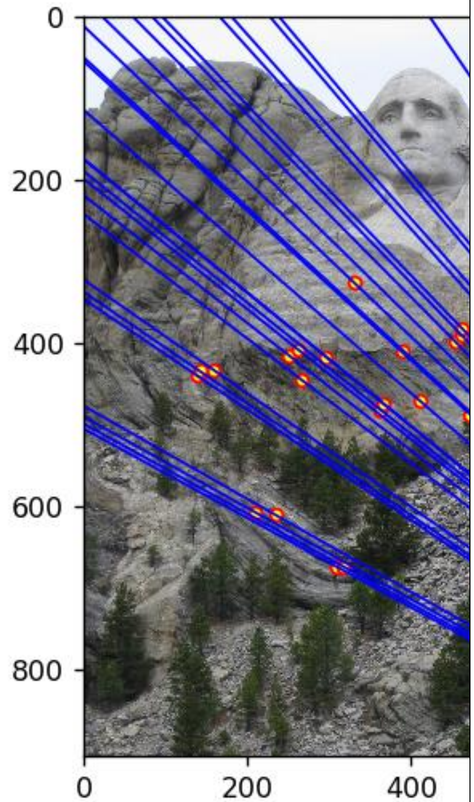
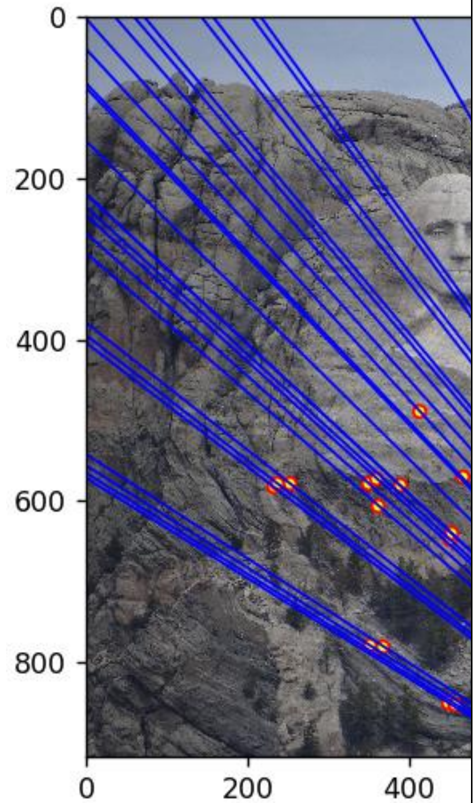
| | | |
|---|---|---|
|  |  |  |

**Table 10: Results for Gaudi without Normalization**

| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|

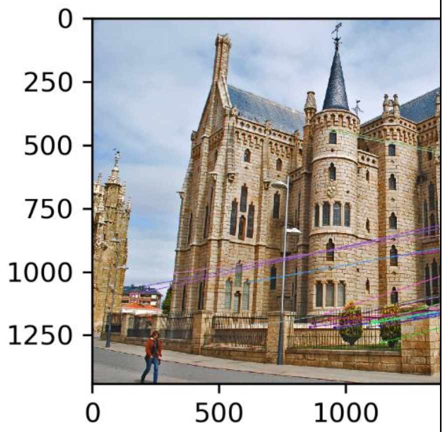
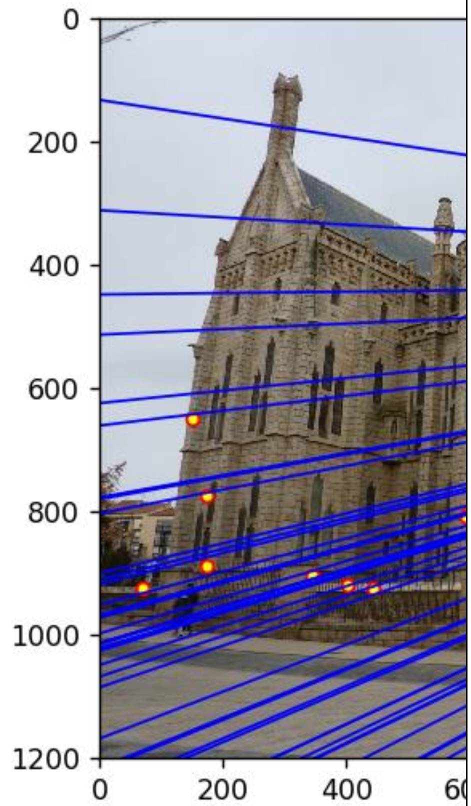
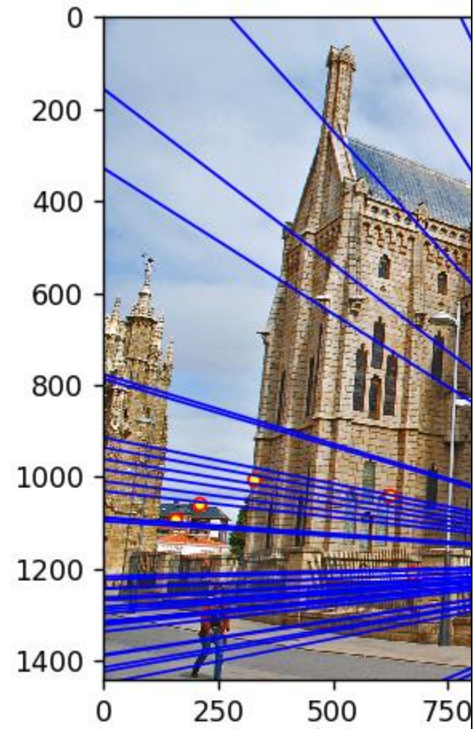
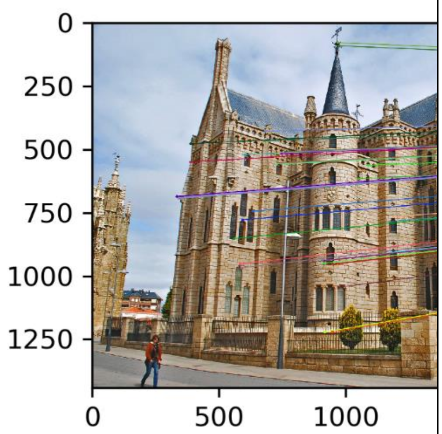
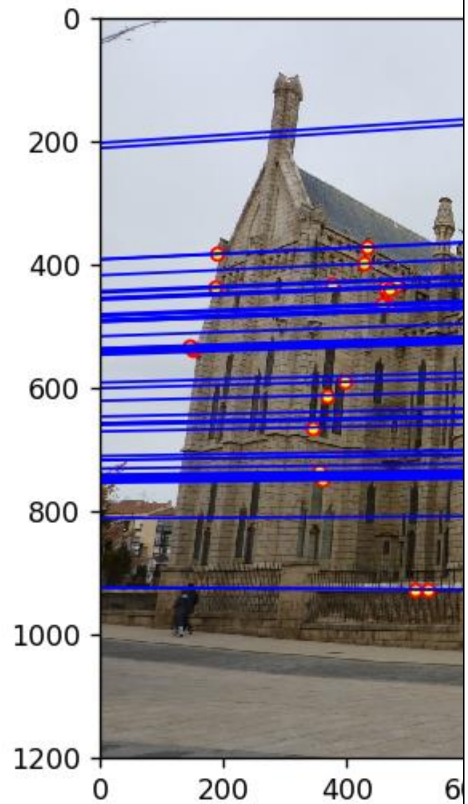
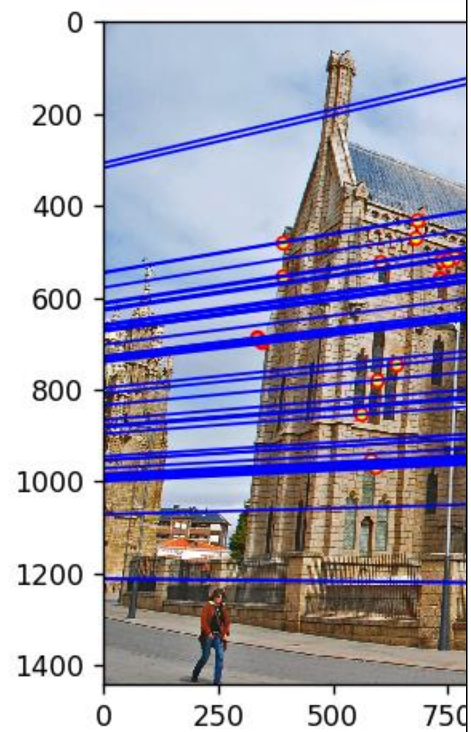
| | | |
|---|---|---|
|  |  |  |

**Table 11: Results for Gaudi with Normalization**

| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|

| | | |
|---|---|---|
|  |  |  |

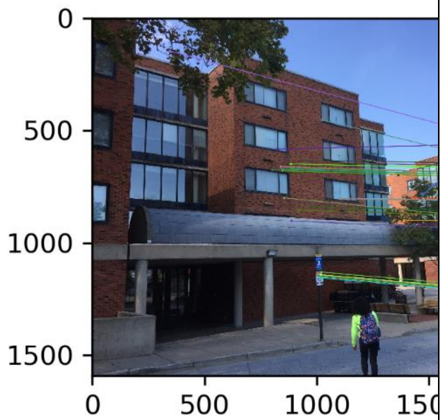


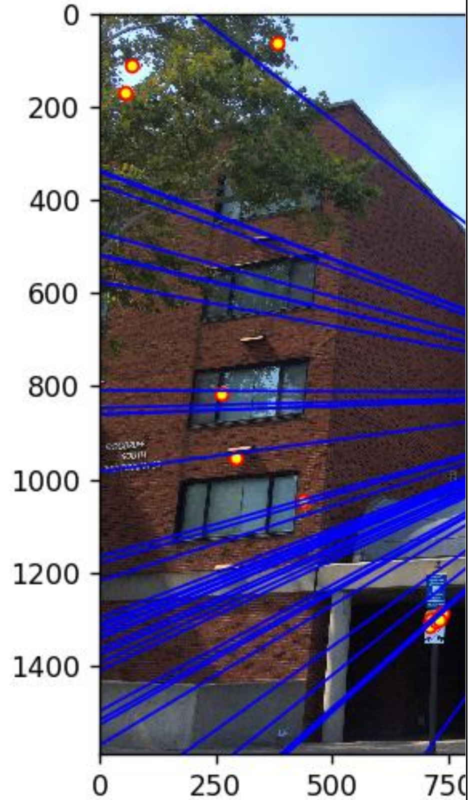




### Table 12: Results for Woodruff without Normalization

| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|



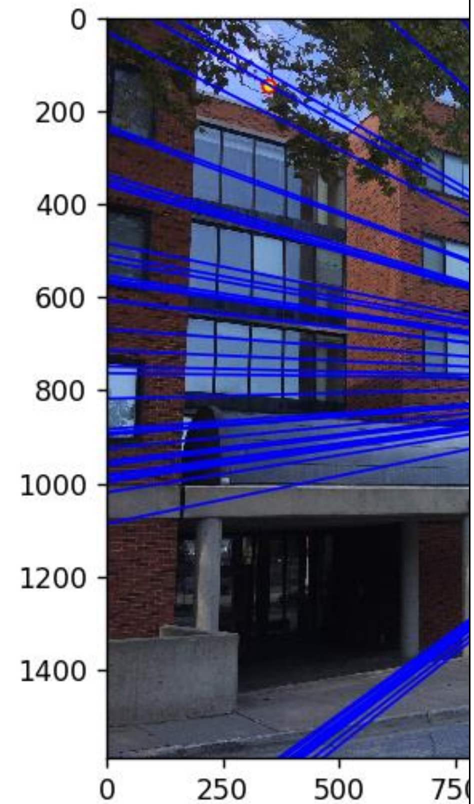
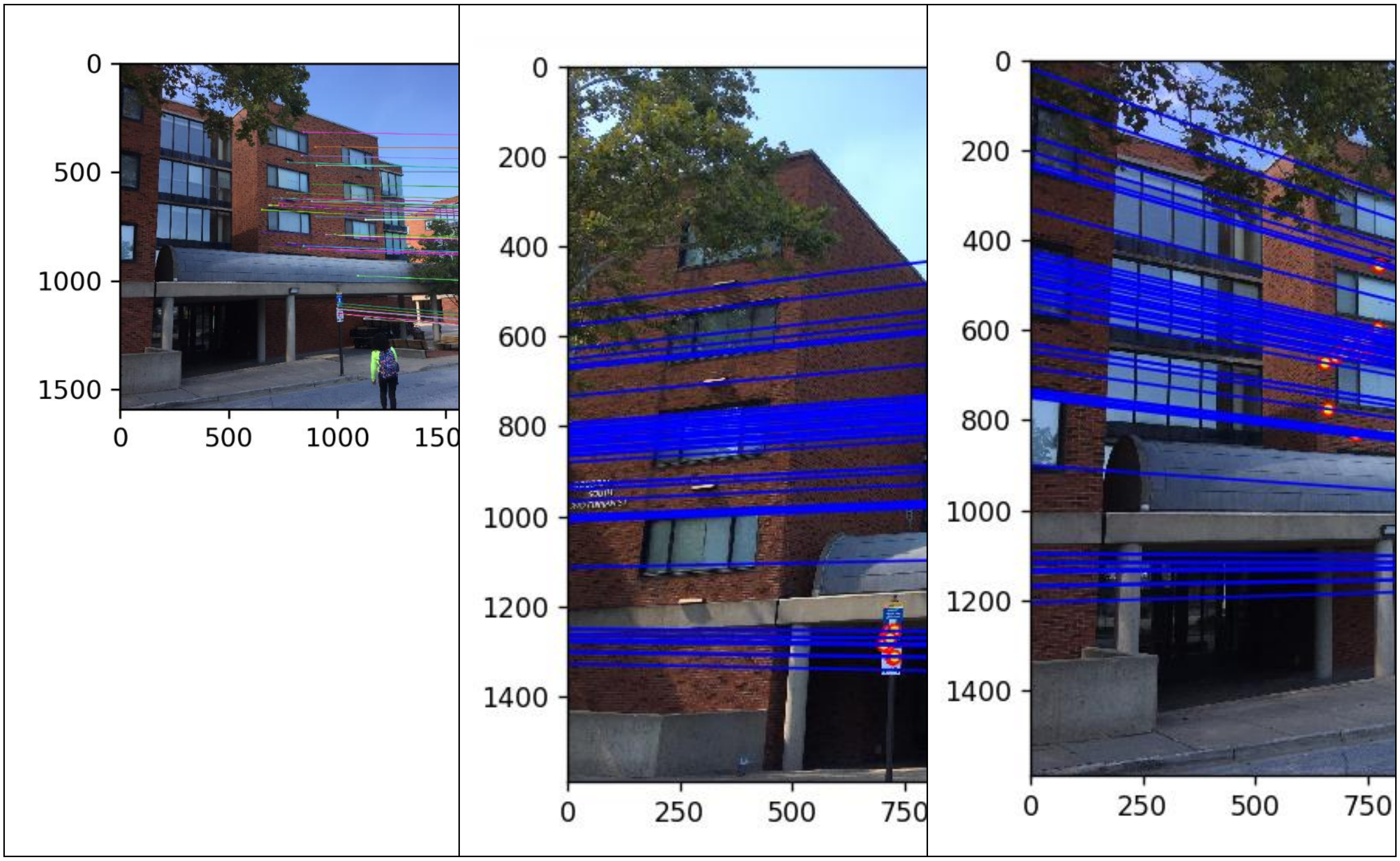




**Table 13: Results for Woodruff** with Normalization

| Matching points between Image_a & Image_b | Epipolar lines in Image_a | Epipolar lines in Image_b |
|---|---|---|




## 3.4. Tuning threshold for ransac_fundamental_matrix()

I have written and used this function to obtain best threshold for ransac_fundamental_matrix().

### Algorithm and implementation of tune_ransac_fundamental_matrix() [1]:

The general algorithm is the same as ransac_fundamental_matrix() with following differences:

- Instead of a single threshold value, 5 different values are used:
    - threshold = [0.005, 0.01, 0.02, 0.04, 0.1]
- I have also tried different threshold for training and testing:
    - Train threshold is the threshold used for first loop (that obtains best random batch of points);
    - Test threshold is the threshold used for best_F;
- best_inlier_count[i,j] is obtained for every train_thresh_j and test_thresh_i

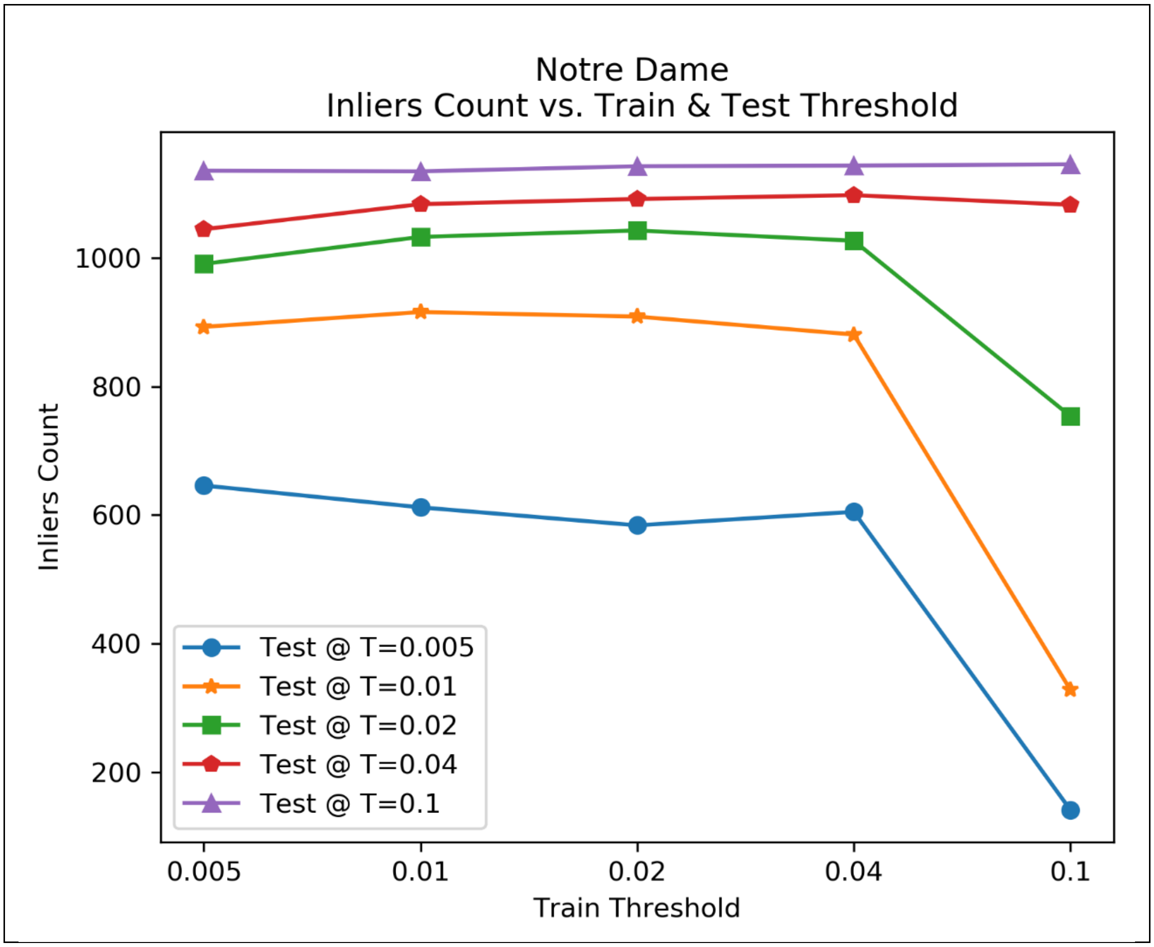
- train_thresh_j = threshold[j]
- test_thresh_i = threshold[i]

```python
def tune_ransac_fundamental_matrix(matches_a, matches_b):
    """
    threshold = [0.005, 0.01, 0.02, 0.04, 0.1]

    For every train_thresh_j and test_thresh_i
        Obtain and save best_inlier_count[i,j]

    """

    ###########################################################################
    # TODO: YOUR RANSAC CODE HERE
    ###########################################################################

    # Define constants
    N = matches_a.shape[0]
    max_iter = 15000
    threshold = [0.005, 0.01, 0.02, 0.04, 0.1]
    batch_size = 8

    # Define random indexes with size = (max_iter, batch_size)
    rand_idx = np.random.randint(N,size = (max_iter, batch_size))
    # np.save('rand_idx_.npy',rand_idx)
    # rand_idx = np.load('rand_idx_526.npy')


    # Obtain UV matrix (algorithm described in previous section)
    # Extract uA, vA, uB, vB, and I_N from points_a and points_b
    uA = matches_a[:,0].reshape(N,1)
    vA = matches_a[:,1].reshape(N,1)
    uB = matches_b[:,0].reshape(N,1)
    vB = matches_b[:,1].reshape(N,1)
    I_N = np.ones((N,1))

    # Construct matrix UV using uA, vA, uB, vB, and I_N and their multiplications
    UV = np.hstack((uA*uB, vA*uB, uB, uA*vB, vA*vB, vB, uA, vA, I_N ))

    # Define placeholders
    inlier_count = np.zeros((len(threshold), max_iter))
    best_idx = np.zeros(len(threshold)).astype(int)
    best_inlier_count = np.zeros((len(threshold),len(threshold))).astype(int)


    # Iteration for k=0:max_iter
    for k in range(max_iter):

        # Estimate F matrix using random batch[k] of 8 pairs of points
```

```
        F = estimate_fundamental_matrix(matches_a[rand_idx[k,:],:],matches_b[rand_idx[k,:],:]
)

        # Calculate cost of estimated F using following equation
        cost_k = np.abs( UV @ F.reshape((9,1)) )


        # Obtain and save number of inliers (with cost < threshold_j)
        for j, train_thresh_j in enumerate(threshold):
            inlier_idx = cost_k < train_thresh_j
            inlier_count[j,k] = np.sum(inlier_idx)


    # For every train_thresh_j and test_thresh_i
    # Obtain and save best_inlier_count[i,j]

    for j, train_thresh_j in enumerate(threshold):
        sort_idx = np.argsort(-inlier_count[j,:])
        best_idx[j] = sort_idx[0]
        best_batch = rand_idx[best_idx[j],:]
        best_F = estimate_fundamental_matrix(matches_a[best_batch,:], matches_b[best_batch,:]
)

        cost = np.abs( UV @ best_F.reshape((9,1)) )

        for i, test_thresh_i in enumerate(threshold):

            inlier_idx = cost < test_thresh_i
            best_inlier_count[i,j] = np.sum(inlier_idx)

    ###########################################################################
    # END OF YOUR CODE
    ###########################################################################

    return best_inlier_count
```

### Table 14: Inliers Count vs. Train & Test Threshold for Notre Dame

| Inliers Count vs. Train & Test Threshold for Notre Dame |
| --- |




**Remarks on the Inliers Count vs. Train & Test Threshold:**

- Obviously, the higher test threshold value, the higher number of inliers. Therefore, it is not reasonable to tune threshold value using same value for training and testing;
- Also, higher inliers is not necessarily desirable because it increases the chance of false inliers;
- Therefore, it is best to compare different train thresholds based on constant test threshold;
- For test threshold = 0.02, highest inlier is obtained for train threshold = 0.02;
- Therefore, threshold value was chosen to be 0.02;

# Extra Works

Following tasks were done outside assignment requirements:

- Tuning threshold for ransac_fundamental_matrix()

# References


[1] Assignment 01 description by Dr. Kin-Choong Yow

[2] Szeliski, R. (2010). Computer vision: algorithms and applications. Springer Science & Business Media.