Analyse traitement d'image et vision 3D CM

Assignment #2

LE Thi Hoa - 12310380
TRAN Hai Linh - 12310487
Sophin CHHENG - 2312932
Saad Belaouad - 2313962

I. Introduction

This report consists of two sections equivalent to three tasks in assignment #2, including: computing vanishing points and fundamental matrix estimation (Epipolar lines).

We use the *OpenCV* library for reading and storing images, the *Matplotlib* library for displaying images, and the *NumPy* library for computations. We use *Python* for implementation.

II. Computing Vanishing Points

1. Introduction

In this part, we aim to compute vanishing points in images of a Rubik's cube and investigate various aspects related to the vanishing points. Vanishing points are crucial in computer vision for understanding the perspective and geometry of objects in images. The study of vanishing points in images of a Rubik's cube allows us to gain insights into the geometric properties of the cube and its representation in two-dimensional images.

2. Methodology and Procedure

a. Data Collection:

Image Acquisition: A diverse set of images featuring a Rubik's cube is collected. These images capture the cube from various angles and orientations to encompass its full geometric variability.

Variation in Camera Positions: The images are obtained by placing the camera at different positions around the Rubik's cube. This variation helps in capturing different perspectives and orientations of the cube.

b. Computing vanishing points

Vanishing points are crucial concepts in projective geometry and computer vision, particularly when analyzing perspective in images. They play a significant role in understanding the convergence of parallel lines in three-dimensional space when projected onto a two-dimensional image .

Vanishing Point Computation Methods:

- **Canny Edge Detection:**

  To create an image with clear edges, we use the Canny Edge Detection algorithm. This helps us accurately identify edges in the image.

```
def canny_edge_detection(image, low_threshold, high_threshold):
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply GaussianBlur to the image
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)

    # Apply Canny edge detection
    edges = cv2.Canny(blurred, low_threshold, high_threshold)
    return edges
```

- **Hough Transform**

  Next, we employ the Hough Transform to detect lines in the image. These lines correspond to edges and objects in 3D space.

```
def hough_transform(image, rho, theta, threshold):
    # Perform Hough Transform to detect lines
    lines = cv2.HoughLines(image, rho, theta, threshold)

    # Convert polar coordinates to Cartesian coordinates
    cart_lines = []
    for line in lines:
        rho, theta = line[0]
        if np.sin(theta) != 0:  # Avoid division by zero
            m = -1 / np.tan(theta)
            b = rho / np.sin(theta)
            cart_lines.append((m, b))

    return cart_lines
```

- **Finding Vanishing Points**

  We compute vanishing points by finding the intersection of the detected lines. This is achieved through solving a system of linear equations.

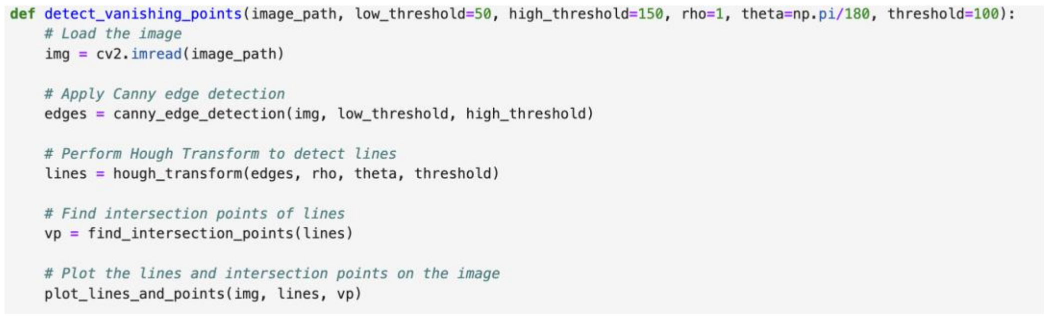
```python
def find_intersection_points(lines):
    # Find intersection points of lines
    vp = []
    for i in range(len(lines)):
        for j in range(i + 1, len(lines)):
            m1, b1 = lines[i]
            m2, b2 = lines[j]
            if abs(m1 - m2) > 1e-6:
                x0 = (b2 - b1) / (m1 - m2)
                y0 = m1 * x0 + b1
                if not np.isnan(x0) and not np.isnan(y0):
                    x0, y0 = int(np.round(x0)), int(np.round(y0))
                    vp.append((x0, y0))
    return vp
```

- **Displaying Results**

  Finally, we showcase the results by drawing the detected lines and vanishing points on the image.

```python
def plot_lines_and_points(image, lines, points):
    # Plot the lines on the image
    for line in lines:
        m, b = line
        x1 = 0
        y1 = int(b)
        x2 = image.shape[1]
        y2 = int(m * x2 + b)
        cv2.line(image, (x1, y1), (x2, y2), (0, 0, 255), 1)

    # Plot the intersection points on the image
    for p in points:
        cv2.circle(image, p, 5, (255, 0, 0), -1)

    # Display the image using matplotlib
    img_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    plt.imshow(img_rgb)
    plt.title('Vanishing Points Detection')
    plt.axis('off')
    plt.show()
```

- **Create a function to calculate vanishing points**



```
def detect_vanishing_points(image_path, low_threshold=50, high_threshold=150, rho=1, theta=np.pi/180, threshold=100):
    # Load the image
    img = cv2.imread(image_path)

    # Apply Canny edge detection
    edges = canny_edge_detection(img, low_threshold, high_threshold)

    # Perform Hough Transform to detect lines
    lines = hough_transform(edges, rho, theta, threshold)

    # Find intersection points of lines
    vp = find_intersection_points(lines)

    # Plot the lines and intersection points on the image
    plot_lines_and_points(img, lines, vp)
```

=> Results:

➔ Case 1:

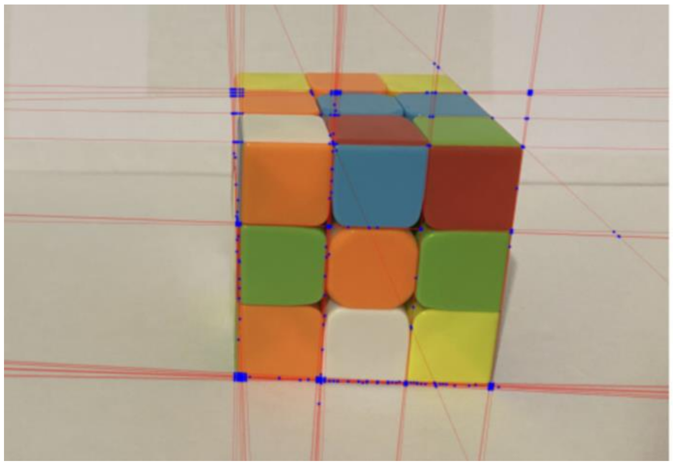


Figure 1: One vanishing point

➔ Case 2:

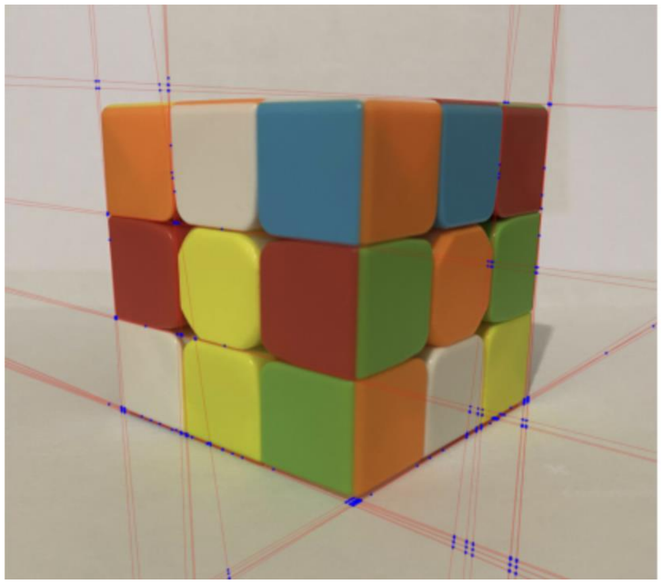


Figure 2: Two vanishing points

➔ Case 3:

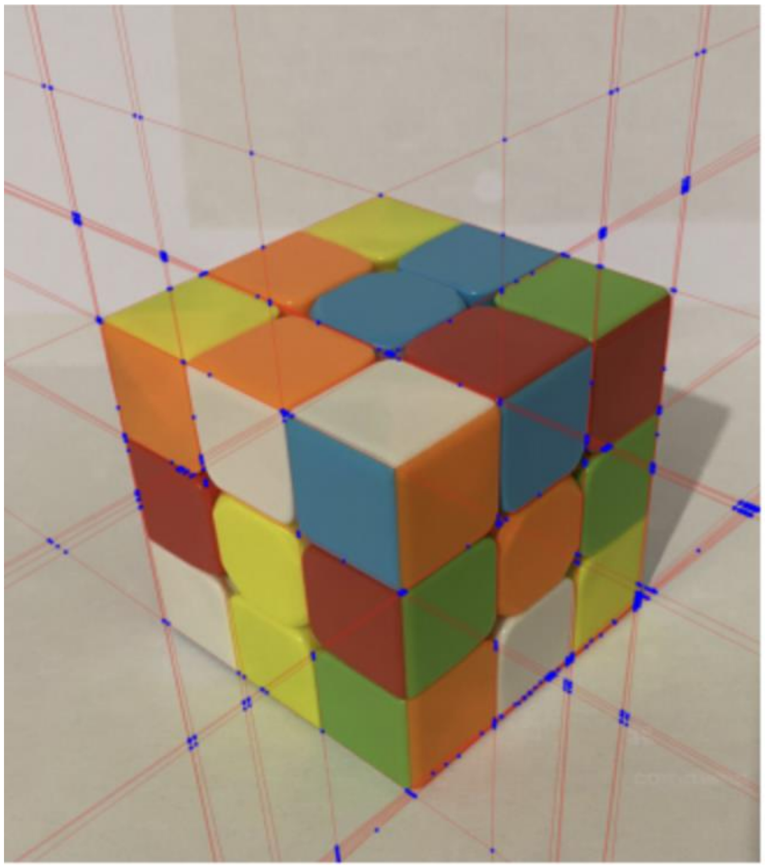


Figure 3: Three vanishing points

3. Conclusion And Answering Questions

What is the minimum and maximum number of finite vanishing points that can be found from all possible images of a Rubik's cube? Explain.

Minimum Number of Vanishing Points: Minimum Number of Vanishing Points: The minimum number of vanishing points that can be found in images of a Rubik's Cube is zero. The cube is photographed directly face-on, so all its sides are parallel to the image plane. No parallel lines converge; they remain parallel as they recede into the distance. This occurs in the case of an orthographic projection, where all the lines on the Rubik's Cube are represented as parallel and do not converge at any point. Orthographic projection is used to represent objects without distortion due to perspective, and all dimensions of the object are maintained in proportion.

Maximum Number of Vanishing Points: The maximum number of vanishing points in images of a Rubik's Cube is three. This occurs in the case of a three-point perspective, where each face of the Rubik's Cube faces towards a separate vanishing point. In a three-point perspective, the Rubik's Cube is often viewed from an angle where all three faces are visible, and each face points in a different direction, resulting in three vanishing points.

Thus, in all possible images of a Rubik's Cube, the minimum number of vanishing points is zero (in orthographic projection), and the maximum number is three (in three-point perspective).

Is there a relation between the number of visible faces and vanishing points? Explain with reasoning and experiments.

Yes, there is a relationship between the number of visible faces of a Rubik's Cube (or any cubic object) and the number of vanishing points in a perspective drawing. This relationship is governed by the principles of perspective in drawing and visualization.

Understanding the Rubik's Cube and Perspective:

- Rubik's Cube Structure: A Rubik's Cube is a cube-shaped puzzle consisting of smaller cubelets. As a cubic object, it has six faces, but in any given orientation, a maximum of three faces can be visible to the observer.
- Vanishing Points: Vanishing points are where parallel lines appear to converge in a drawing, creating the illusion of depth and perspective. The number of vanishing points corresponds to the perspective type: one-point, two-point, or three-point perspective.

Relationship Between Visible Faces and Vanishing Points

- One-Point Perspective:
  - Description: Here, there's a single vanishing point. This perspective is used for objects directly facing the viewer.
  - Rubik's Cube Visibility: In a one-point perspective, you typically see only one face of the Rubik's Cube fully, with slight parts of

    the adjacent faces depending on the cube's tilt and the observer's angle.

- Two-Point Perspective:
    - Description: This perspective uses two vanishing points, usually for objects at an angle to the viewer.
    - Rubik's Cube Visibility: You can usually see two or three faces of the Rubik's Cube in this perspective. The cube's orientation will determine how much of each face is visible.
- Three-Point Perspective:
    - Description: Involves three vanishing points and is often used for dramatic or high/low angle views.
    - Rubik's Cube Visibility: All three visible faces of the Rubik's Cube can be depicted with some level of distortion due to the perspective, especially if the viewpoint is from above or below.

Experimental Observation: To experiment and observe this relationship:

The three result images above (Figure 1, 2, 3) illustrate the relationship between the visible faces and the number of vanishing points.

Conclusion: In conclusion, the number of visible faces of a Rubik's Cube in a drawing is directly related to the number of vanishing points used in the perspective. One-point perspective typically shows one or two faces, two-point perspective can show two or three faces, and three-point perspective can show all three visible faces, albeit with perspective distortion. This relationship is essential for accurately representing cubic objects like a Rubik's Cube in drawings and art.

Can you find 4 vanishing points in Rubik’s cube images? If yes, demonstrate with an image. If no, explain your reasoning.

No, It is not possible to find 4 vanishing points in the image of the Rubik's Cube. Let's consider the fundamental principles of perspective and the number of vanishing points that can appear in an image.

Fundamental Principles of Perspective:

- In a perspective image, for a three-dimensional object like a Rubik's Cube, parallel lines in the real world will converge at a vanishing point in the image.
- The maximum number of vanishing points we can have depends on the type of perspective: one point, two points, or three points.

Rubik's Cube and Number of Faces:

- A Rubik's Cube has six faces, and each face can correspond to a vanishing point.
- In the case of one-point perspective, we only see one face of the Rubik's Cube.
- In two-point perspective, we may see two or three faces, depending on the viewing angle and orientation of the Rubik's Cube.
- In three-point perspective, we can see all three faces, but with some distortion due to the perspective.

Cannot Have 4 Vanishing Points:

- Because a Rubik's Cube has only six faces, it is not possible to generate more than three vanishing points in an image without compromising the accuracy of the perspective. -If there were four vanishing points, it would result in an image that cannot accurately retain the distinctive features and angles of the Rubik's Cube.

Therefore, based on the fundamental principles of perspective and the structure of the Rubik's Cube, it is not possible to find 4 vanishing points without sacrificing the accuracy of representing the Rubik's Cube image.

Can you find a configuration with at least one vanishing point outside the image? If yes, demonstrate with an image. If no, explain your reasoning.

Yes, it's common for vanishing points to lie outside the actual image frame, especially in pictures with a wide-angle view or when the object is very close to the lens. The lines will appear to converge at a point beyond the borders of the image.

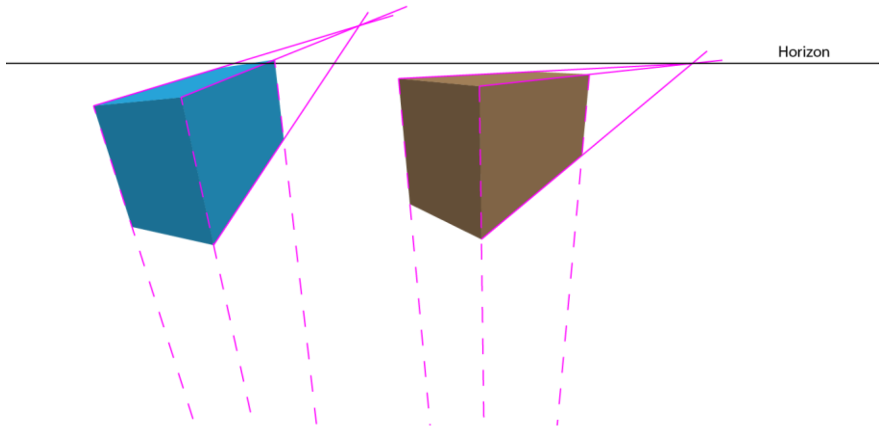


Figure 4: Example for this case

III. Fundamental Matrix Estimation

1. Introduction

In this part, we aim to estimate the Fundamental Matrix from two images of the same configuration of a Rubik's cube. The Fundamental Matrix is a crucial element in computer vision, particularly in stereo vision applications. It represents the epipolar geometry between two images, allowing us to establish correspondences between points in the two images.

2. Methodology and Procedure

Image Loading and Preprocessing:

- Description: This function reads two input images from specified paths.
- Purpose: It provides the initial data for further processing, allowing the algorithm to work with the actual images.

Feature Detection and Description:

- Description: Utilizes the ORB (Oriented FAST and Rotated BRIEF) algorithm to find keypoints and compute descriptors for each image.
- Purpose: Identifying distinctive points and extracting features that can be matched between the images.

Feature Matching:

- Description: Matches the descriptors of keypoints in both images using the Brute-Force Matcher.

- Purpose: Establishing correspondences between keypoints, forming the basis for subsequent geometric analysis.

Visualization of Feature Matches:

- Description: Draws lines connecting matched features between the two images.
- Purpose: Offers a visual representation of the matched keypoints, aiding in the assessment of feature matching accuracy.

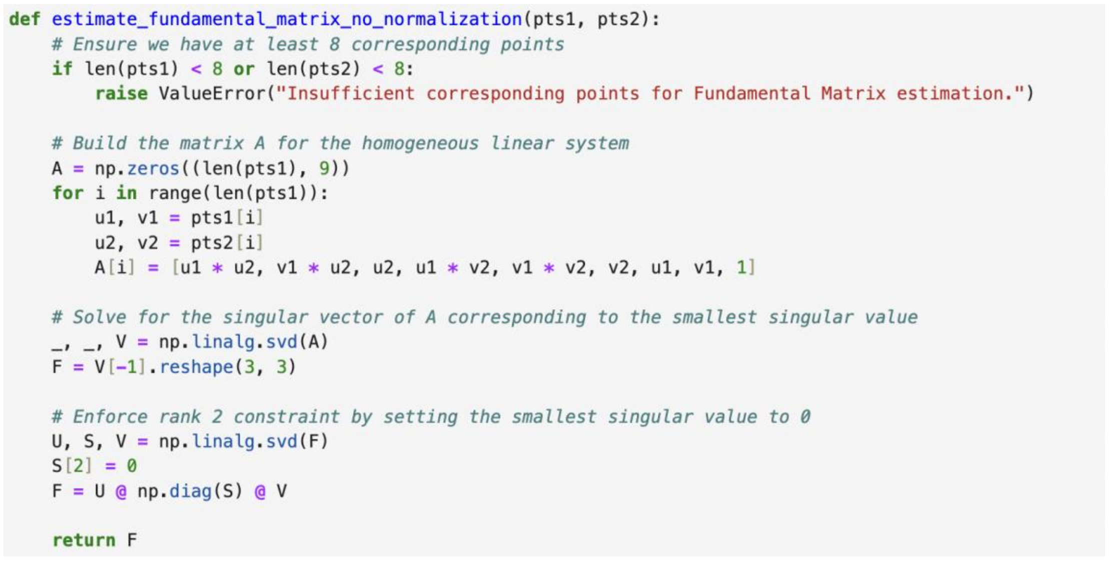
Visualization of Epipolar Lines:

- Description: Estimates the fundamental matrix using a set of matched keypoints.
- Purpose: Provides a mathematical model representing the geometric relationship between the two camera views.

Complete Workflow and Visualization:

- Description: Computes and visualizes epipolar lines corresponding to matched keypoints in both images.
- Purpose: Facilitates the qualitative evaluation of the accuracy of the estimated fundamental matrix.

❖ **Display the epipolar lines without normalization data**



```python
def estimate_fundamental_matrix_no_normalization(pts1, pts2):
    # Ensure we have at least 8 corresponding points
    if len(pts1) < 8 or len(pts2) < 8:
        raise ValueError("Insufficient corresponding points for Fundamental Matrix estimation.")

    # Build the matrix A for the homogeneous linear system
    A = np.zeros((len(pts1), 9))
    for i in range(len(pts1)):
        u1, v1 = pts1[i]
        u2, v2 = pts2[i]
        A[i] = [u1 * u2, v1 * u2, u2, u1 * v2, v1 * v2, v2, u1, v1, 1]

    # Solve for the singular vector of A corresponding to the smallest singular value
    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)

    # Enforce rank 2 constraint by setting the smallest singular value to 0
    U, S, V = np.linalg.svd(F)
    S[2] = 0
    F = U @ np.diag(S) @ V

    return F
```

To compute the fundamental matrix F from corresponding points between two images of the same object, with Rubik's Cube, the following method is employed:

Minimum Condition Check

- First, the function checks if there are enough corresponding points (at least 8 points) between two images to perform the estimation of the fundamental matrix. This ensures the accuracy of the estimation.

Build A Matrix for the Homogeneous Linear System:

- Using the corresponding points, the function constructs matrix A, where each row of the matrix corresponds to a point. Each row contains components of the feature vector used to estimate the fundamental matrix.

Solve the Homogeneous Linear System:

- Apply the Singular Value Decomposition (SVD) method to solve the linear equation Ax = 0. The last eigenvector of matrix A is used to construct the 3x3 fundamental matrix.

Build Fundamental Matrix and Apply Rank-2 Constraint:

- Construct the fundamental matrix F from the eigenvector and enforce the rank-2 constraint by setting the smallest eigenvalue to 0. This ensures the rank-2 property of the fundamental matrix.

Result

- The result of this method is the estimated fundamental matrix. For images of a Rubik's Cube, this matrix describes the relationship between corresponding points and is used to draw epipolar lines on the images.

➔ Display the epipolar lines without normalization data:

➔ Example 1:

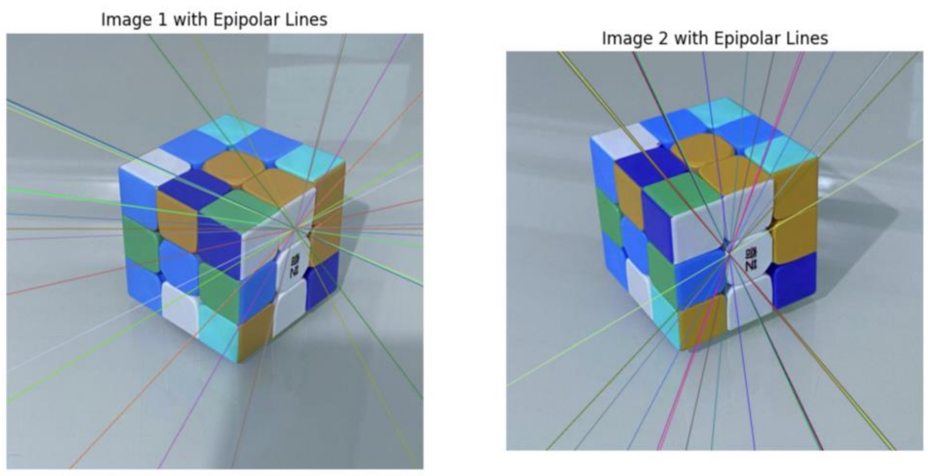


Figure 5: example 1: the epipolar lines without normalization data

➔ Example 2:

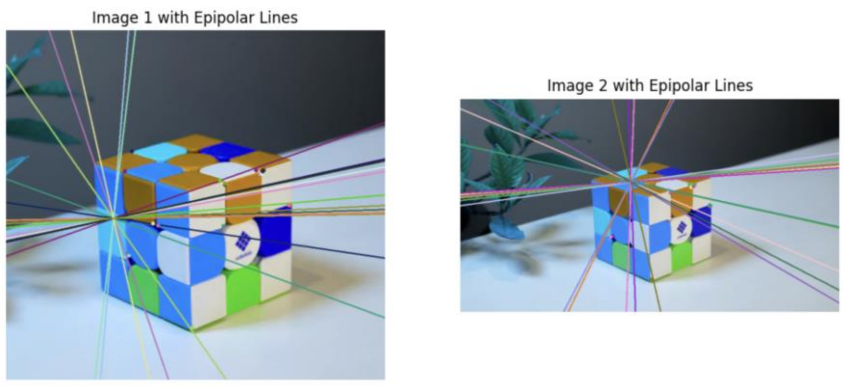


Figure 6: example 2: the epipolar lines without normalization data

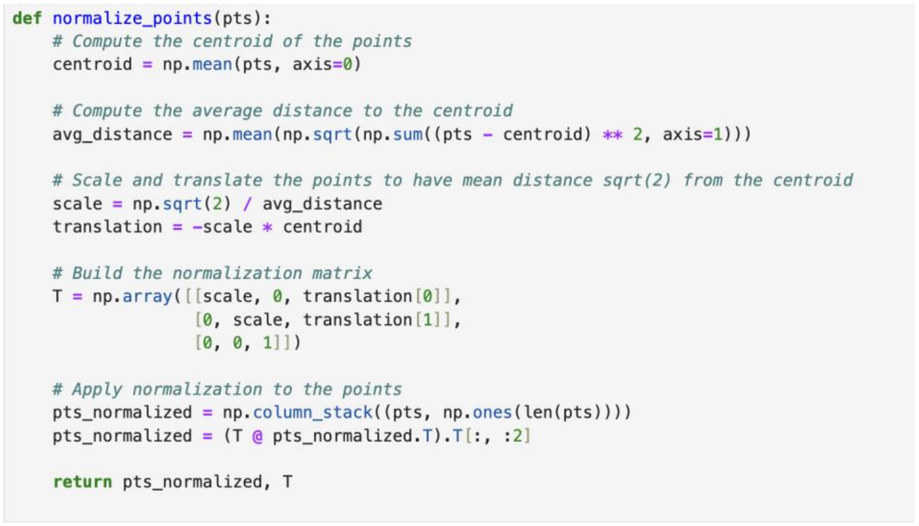
Conclusion: The epipolar lines appear to converge at different points, indicating that the stereo images are uncalibrated. The lines do not intersect at the same point on the cube, which can lead to inaccuracies when trying to match points between the two views.

❖ **Normalize the data and recompute the fundamental matrix**

Normalize corresponding points to enhance the accuracy of fundamental matrix estimation. This process ensures points are uniformly scaled and positioned, optimizing accuracy in subsequent fundamental matrix computations. Implementation steps:

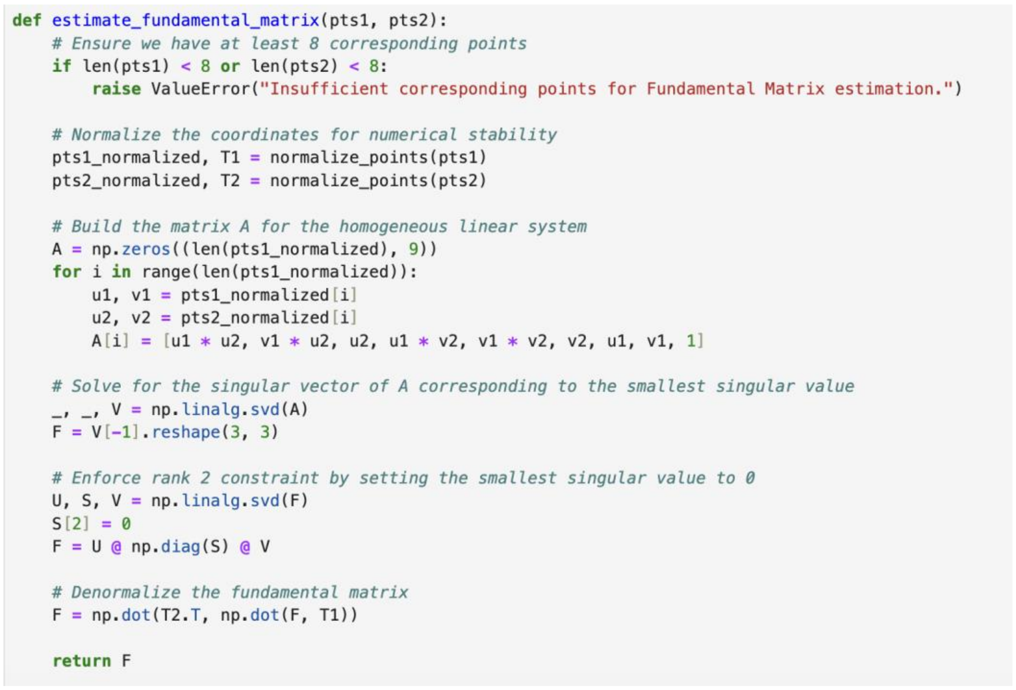
- Compute Centroid: Calculate the average position of points.

- Compute Average Distance: Find the average distance from each point to the centroid**.**
- Scale and Translate: Adjust points to have a mean distance of sqrt(2) from the centroid.
- Build Normalization Matrix (T): Create a matrix encapsulating scaling and translation.
- Apply Normalization: Transform points using the normalization matrix.



```python
def normalize_points(pts):
    # Compute the centroid of the points
    centroid = np.mean(pts, axis=0)

    # Compute the average distance to the centroid
    avg_distance = np.mean(np.sqrt(np.sum((pts - centroid) ** 2, axis=1)))

    # Scale and translate the points to have mean distance sqrt(2) from the centroid
    scale = np.sqrt(2) / avg_distance
    translation = -scale * centroid

    # Build the normalization matrix
    T = np.array([[scale, 0, translation[0]],
                  [0, scale, translation[1]],
                  [0, 0, 1]])

    # Apply normalization to the points
    pts_normalized = np.column_stack((pts, np.ones(len(pts))))
    pts_normalized = (T @ pts_normalized.T).T[:, :2]

    return pts_normalized, T
```

Then, recompute the fundamental matrix. Similar to the method described above, we normalize the data and then compute the fundamental matrix.



```python
def estimate_fundamental_matrix(pts1, pts2):
    # Ensure we have at least 8 corresponding points
    if len(pts1) < 8 or len(pts2) < 8:
        raise ValueError("Insufficient corresponding points for Fundamental Matrix estimation.")

    # Normalize the coordinates for numerical stability
    pts1_normalized, T1 = normalize_points(pts1)
    pts2_normalized, T2 = normalize_points(pts2)

    # Build the matrix A for the homogeneous linear system
    A = np.zeros((len(pts1_normalized), 9))
    for i in range(len(pts1_normalized)):
        u1, v1 = pts1_normalized[i]
        u2, v2 = pts2_normalized[i]
        A[i] = [u1 * u2, v1 * u2, u2, u1 * v2, v1 * v2, v2, u1, v1, 1]

    # Solve for the singular vector of A corresponding to the smallest singular value
    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)

    # Enforce rank 2 constraint by setting the smallest singular value to 0
    U, S, V = np.linalg.svd(F)
    S[2] = 0
    F = U @ np.diag(S) @ V

    # Denormalize the fundamental matrix
    F = np.dot(T2.T, np.dot(F, T1))

    return F
```

➔ Result with normalization data:

➔ Example 1:

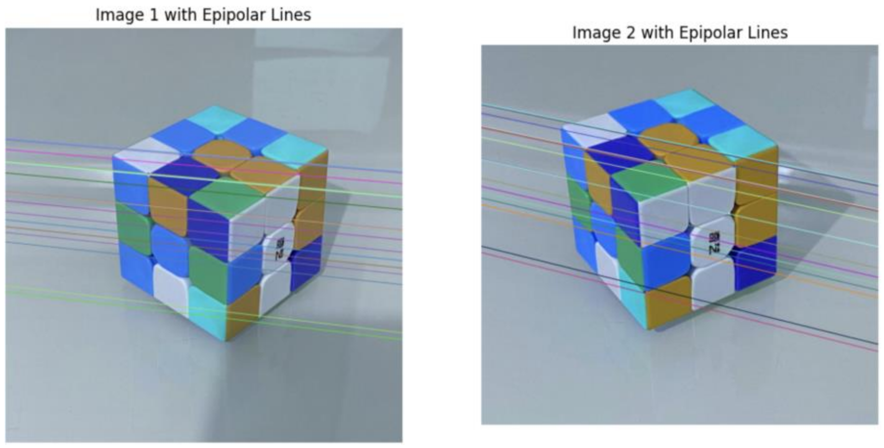


Figure 7: the epipolar lines with normalization data

➔ Example 2*:*

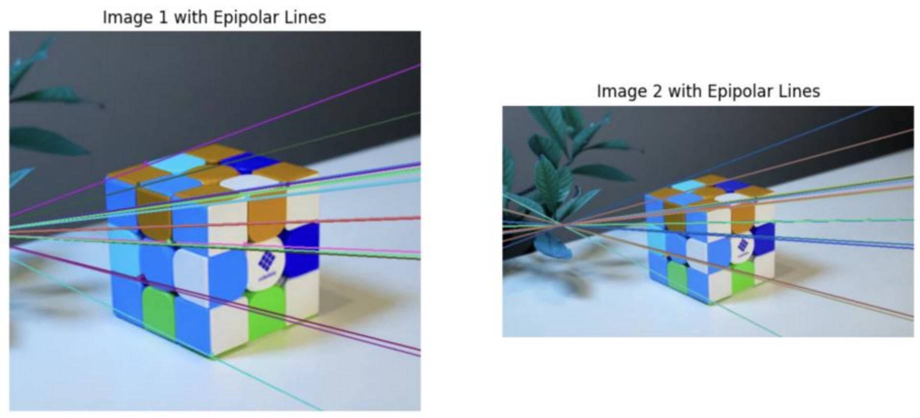


Figure 8: the epipolar lines with normalization data

Conclusion:

- The epipolar lines are more parallel and organized. This suggests that the data has been normalized through a process such as rectification, which aligns the images and makes the corresponding epipolar lines horizontal and parallel. This is a crucial step in stereo vision processing as it simplifies the problem of finding corresponding points to a one-dimensional search along the epipolar lines.
- *Epipolar Line Convergence*: In a well-normalized image, the epipolar lines should ideally converge at the same epipole in each image.

Normalizing data improves accuracy for several reasons:

- Uniform Scale and Units: Normalization ensures a consistent scale and units for data points, making them independent of factors like image size, aspect ratio, or resolution. This reduces distortion and ensures that corresponding points in both images belong to the same spatial frame.
- Reduction of Scale Discrepancies: Normalizing images makes epipolar lines less skewed and often straight, facilitating the estimation of the fundamental matrix. This results in easier and more accurate estimation.
- Consistency in Measurement Units: Data normalization unifies the measurement units of image points, making operations and computations consistent and easy to interpret.

- Increased Stability of Algorithms: Algorithms related to epipolar lines, such as fundamental matrix estimation, are often designed to work well with normalized data. This enhances the stability and accuracy of the algorithms.
- Reduced Impact of Noise and Variability: Normalization can reduce the impact of noise and variability in the data, making prediction and estimation methods more reliable.

3. Conclusion

When displaying the epipolar lines, certain visual indicators can provide information about the accuracy of the estimated fundamental matrix:

- *Consistency of Epipolar Lines*: Epipolar lines should be consistent and intersect at corresponding points in both images. Convergence of lines to a common point indicates an accurate fundamental matrix.
- *Intersection at the Epipole*: Ideally, all epipolar lines should intersect at a common point known as the epipole. Inaccuracy in the fundamental matrix estimation may lead to non-intersecting lines or intersection at non-common points.
- *Parallelism:* Epipolar lines should run uniformly and closely to the image edges without significant deviation. Significant deviations may indicate inaccuracy.
- *Symmetry:* For a well-estimated fundamental matrix, epipolar lines should exhibit symmetry concerning corresponding points. Asymmetrical lines may suggest errors in the estimation process.