Homography Estimation Using Manual and Feature-Based Point Matching

Suyash Ajit Chavan (IMT2021048)
3D Vision

Abstract—**Homography estimation is a fundamental technique in computer vision for aligning two or more images based on corresponding points, enabling tasks such as image stitching, camera calibration, and 3D scene reconstruction. This report presents a comprehensive approach to homography estimation using key algorithms: Scale-Invariant Feature Transform (SIFT) for feature-based matching points, Direct Linear Transformation (DLT) for initial homography computation, Symmetric Transfer Error for refining accuracy, and RANSAC for robust estimation under noisy conditions.**

GitHub Repository: Homography_Estimation

I. Introduction

Homography estimation is a critical technique in computer vision that enables the transformation of points between different views of the same scene. This transformation, represented by a homography matrix, allows applications such as image stitching, camera calibration, and augmented reality by mapping corresponding points across images taken from different perspectives.

This report outlines a robust approach to homography estimation using a combination of algorithms: Scale-Invariant Feature Transform (SIFT) for detecting and matching key points, Direct Linear Transformation (DLT) for initial matrix computation, Symmetric Transfer Error for accuracy refinement, and RANSAC to handle outliers. Together, these methods produce an accurate and reliable homography matrix, essential for achieving precise image alignment.

II. Data Generation using Manual matching of points

1. Multiple views of the *Chessboard* will be captured, as illustrated in the following images.

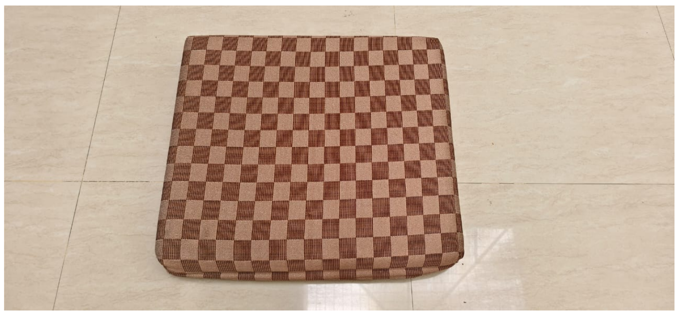


Fig. 1. Chessboard View 1

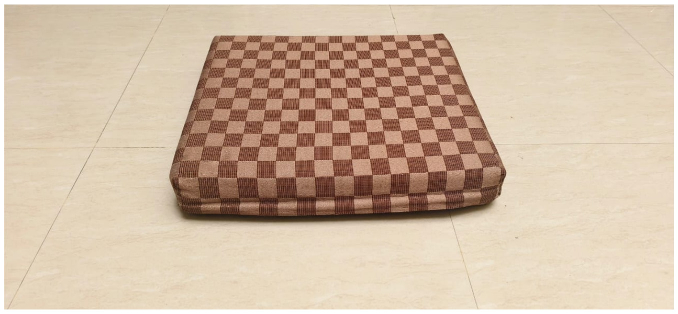


Fig. 2. Chessboard View 2

2. Identify correspondences between points in first view: (x, y) and their respective points in second view: (u, v) through manual identification, primarily focusing on corner points and edge points.
3. We require **atleast 4 points** to estimate homography. The image points indicated can be plotted on 2 views as shown.

a. for View 1:

```
first_image_points = [
    (337, 445),
    (832, 432),
    (382, 80),
    (805, 80),
]

plot_points_on_image(first_image_path,
    first_image_points)
```

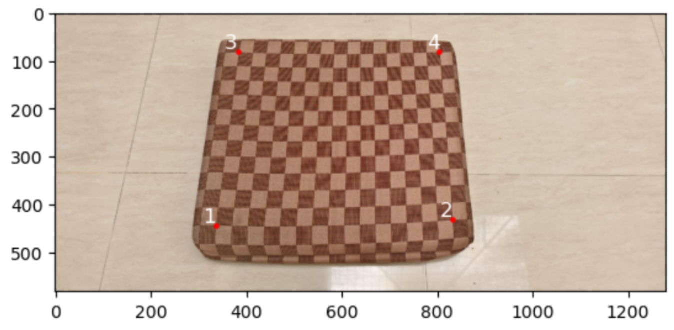


Fig. 3. Matching points of chessboard View 1

b. for View 2:

```
second_image_points = [
    (372, 295),
    (903, 283),
    (435, 70),
    (820, 68),
]

plot_points_on_image(second_image_path,
    second_image_points)
```

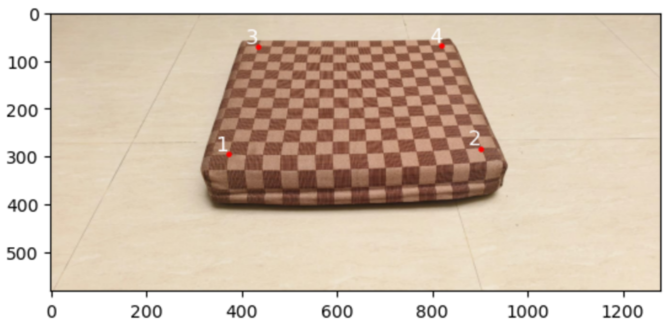


Fig. 4. Matching points of chessboard View 2

4. Construct the final data as a list of pairs comprising first view points and their corresponding second view points:

$$data = [((x_1, y_1), (u_1, v_1)), \ldots, ((x_n, y_n), (u_n, v_n))]$$

,where n denotes total number of correspondences.

```
matching_points1 = get_data(first_image_points,
     second_image_points)
print(matching_points1)
```

III. Data Generation using Feature-based matching of points : SIFT

The SIFT (Scale-Invariant Feature Transform) algorithm is employed to generate feature-based data by detecting, describing, and matching keypoints between two images. This process enables the identification of corresponding points across different views of a scene. The key steps in this approach are outlined as follows:

- **Step 1: Keypoint Detection and Descriptor Computation**
  - First, keypoints and descriptors are detected and computed for each image individually. Keypoints represent distinctive, scale-invariant points in the image, while descriptors capture local visual information around these points.
- **Step 2: Feature Matching Between Descriptors**
  - Descriptors from the two images are then matched using a feature matching technique, such as the brute-force matcher with L2 norm. The matches are sorted by distance to prioritize the best correspondences, with smaller distances indicating closer similarity between features.
- **Step 3: Extracting Matching Points**
  - From the set of matches, corresponding points in the two images are extracted. Each match provides a pair of points: one from the first image and the corresponding point from the second image. These matched points form the basis for further transformation estimation.
- **Step 4: Visualization of Matching Points**
  - The top matches can be visualized by marking the matching points on each image, which provides an effective way to validate the accuracy of feature matching. Visualization aids in assessing the quality of matches by displaying keypoints in both images that correspond to the same locations in the scene.

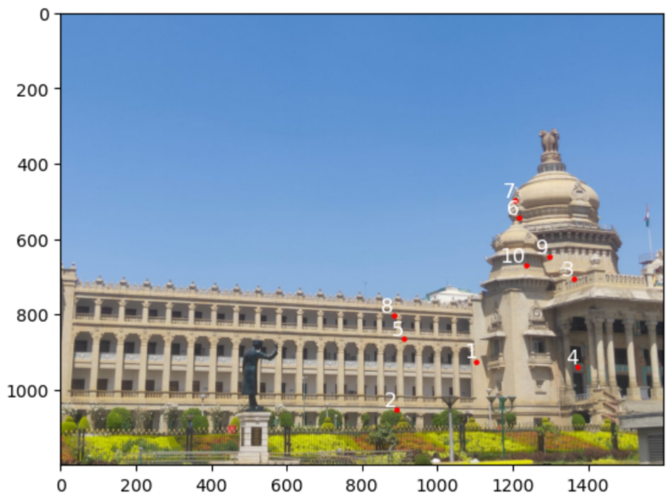


Fig. 5. Top Matching points in View 1 of Building using SIFT

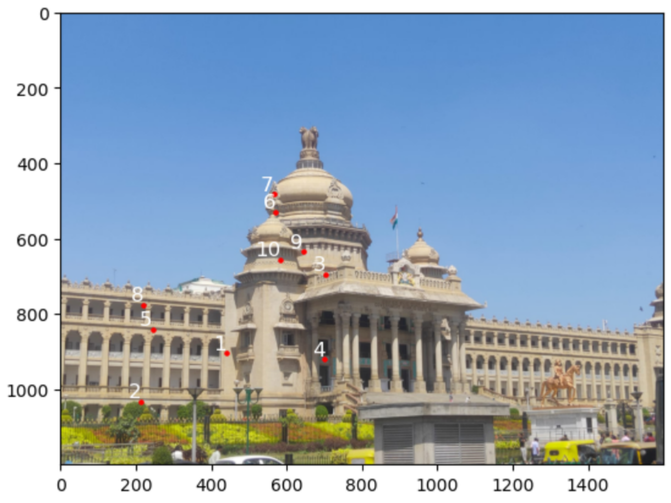


Fig. 6. Top Matching points in View 2 of Building using SIFT

This feature-based matching approach, using SIFT keypoints and descriptors, provides a robust and scale-invariant method for establishing correspondences between images. The generated data can then be used further for homography estimation, to align and transform one image to another.

IV. Calculating Homography matrix using Direct Linear Transformation (DLT)

The homography matrix (H), with dimensions 3×3, establishes a relationship between two views of the same planar surface, mapping corresponding points from one image to another. It is essential in applications where the transformation between images can be represented by rotation, translation, and scaling on a plane, such as in stitching panoramas, rectifying images, or analyzing perspectives. Homography estimation involves finding the optimal matrix H to align corresponding points across images, allowing for a precise spatial mapping between views.

$$H = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix}$$

1. For each matching pair i in first view and second view, define equations as:

$$\underbrace{\begin{bmatrix} u^{(i)} \\ v^{(i)} \\ 1 \end{bmatrix}}_{\text{Known}} = \underbrace{\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix}}_{\text{Unknown}} \underbrace{\begin{bmatrix} x^{(i)} \\ y^{(i)} \\ 1 \end{bmatrix}}_{\text{Known}}$$

$$\boxed{x_i^{'} = Hx_i}$$

2. Expanding the matrix as linear equations:

$$u^{(i)} = \frac{H_{11}x^{(i)} + H_{12}y^{(i)} + H_{13}}{H_{31}x^{(i)} + H_{32}y^{(i)} + H_{33}}$$

$$v^{(i)} = \frac{H_{21}x^{(i)} + H_{22}y^{(i)} + H_{23}}{H_{31}x^{(i)} + H_{32}y^{(i)} + H_{33}}$$

3. Rearranging the terms, we get single equation as:

$$\mathbf{Ah} = \mathbf{0}$$

$$\underline{\text{Known}} \quad \underline{\text{Unknown}}$$

Here, A is *Data matrix* and h is *Homography vector*.

$$\begin{bmatrix} x_s^{(i)} & y_s^{(i)} & 1 & 0 & 0 & 0 & -x_d^{(i)}x_s^{(i)} & -x_d^{(i)}y_s^{(i)} & -x_d^{(i)} \\ 0 & 0 & 0 & x_s^{(i)} & y_s^{(i)} & 1 & -y_d^{(i)}x_s^{(i)} & -y_d^{(i)}y_s^{(i)} & -y_d^{(i)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Fig. 7. Data matrix (A) and equation Ah = 0

4. Solve for h (homography vector):

$$\mathbf{Ah} = \mathbf{0}$$

A. *Least Sqaures Solution for h (Homography vector):*

Set scale so that: $\|h\|^2 = 1$.
We want Ah as close to 0 as possible and $\|h\|^2 = 1$:

$$\min_h \|Ah\|^2 \text{ such that } \|h\|^2 = 1$$

Equivalently:

$$\min_h (h^T A^T Ah) \text{ such that } h^T h = 1$$

Define Loss function $L(h, \lambda)$:

$$L(h, \lambda) = h^T A^T Ah - \lambda(h^T h - 1)$$

Taking derivatives of $L(h, \lambda)$ w.r.t h:

$$2A^T Ah - 2\lambda h = 0$$

$$\boxed{A^T Ah = \lambda h} \qquad \text{(Eigenvalue Problem)}$$

$\therefore$ Eigenvector h with smallest eigenvalue λ of matrix $A^T A$ minimizes the loss function $L(h)$.
$\Rightarrow$ homography vector (h) = Right singular vector of A corresponding to smallest singular value.

Rearrange the *Homography vector (h)* to get *Homography matrix (H)* of dimensions (3, 3).

```
# Calculates Homography matrix using various
    methods
class HomographyMatrix:
    def __init__(self, A):
        _, cols = A.shape
        if cols != 9:
            raise ValueError("Expected number
                of columns is 9, but got {cols}
                ")
        self.A = A

    # Returns Homography Matrix of size (3, 3)
        using Homography vector
    def get_homography_matrix(self):
        pass

    # Prints Homography matrix
    def print_homography_matrix(self):
        pass

class DirectLinearTransformation(
    HomographyMatrix):
    def __init__(self, A):
        super().__init__(A)

    # Returns Homography Vector, which is right
         singular vector of A corresponding to
        smallest Eigen value.
    def get_homography_vector(self):
        svd = SVD(self.A)
        _, cols = self.A.shape
        homography_vector = svd.
            get_right_singular_vector(cols)
        if len(homography_vector) != 9:
            raise ValueError(f"Expected length
                of 9, but got {len(
                homography_vector)}")
        return homography_vector

    # Returns Homography Matrix of size (3, 3)
        using Homography vector
    def get_homography_matrix(self):
        homography_vector = self.
            get_homography_vector()
        homography_matrix = homography_vector.
            reshape(3, 3)
        return homography_matrix

    # Prints Homography vector
    def print_homography_vector(self):
        homography_vector = self.
            get_homography_vector()
        print(f"Homography vector (Dimensions:
            {homography_vector.shape}):\n",
            homography_vector)

    # Prints Homography matrix
    def print_homography_matrix(self):
        homography_matrix = self.
            get_homography_matrix()
        print(f"Homography matrix (Dimensions:
            {homography_matrix.shape}):\n",
            homography_matrix)
```

B. Singular Value Decomposition (SVD) for Data matrix (A):

Singular Value Decomposition (SVD) is a matrix factorization technique that decomposes a matrix A into the product of three matrices: U, Σ, and V^T. Here, U and V^T are orthogonal matrices of row and column space, and Σ is a diagonal matrix containing the singular values of A.

$$A = U\Sigma V^T$$

```
# Singular Value Decomposition (SVD) Module
class SVD:
    def __init__(self, A):
        self.A = A

    # Returns U, S and VT of SVD in Decreasing
        Order of Singular values (S).
    def apply_svd(self):
        """
            U: Left singular vectors
            S: Singular values (as a 1D array)
            VT: Right singular vectors (
                transposed)
        """
        U, S, VT = np.linalg.svd(self.A)
        return U, S, VT

    # Prints SVD Decomposition matrices U, S
        and VT.
    def print_svd(self):
        U, S, VT = self.apply_svd()
        print(f"U (Dimensions: {U.shape}):\n",
            U)
        print(f"S (Dimensions: {S.shape}):\n",
            S)
        print(f"VT (Dimensions: {VT.shape}):\n"
            , VT)

    # Returns the ith left singular vector from
         the U matrix (1-based indexing).
    def get_left_singular_vector(self, ind=1):
        U, _, _ = self.apply_svd()
        if ind < 1 or ind > U.shape[1]:
            raise IndexError(f"Index {ind} is
                out of bounds for the left
                singular vectors. Valid range:
                1 to {U.shape[1]}.")
        return U[:, ind - 1]

    # Returns the ith right singular vector
        from the VT matrix (1-based indexing).
    def get_right_singular_vector(self, ind=1):
        _, _, VT = self.apply_svd()
        if ind < 1 or ind > VT.shape[0]:
            raise IndexError(f"Index {ind} is
                out of bounds for the right
                singular vectors. Valid range:
                1 to {VT.shape[0]}.")
        return VT[ind - 1, :]
```

C. Normalized DLT:

1. Points are translated so that their centroid is at the origin.
2. Points are then scaled so that the average distance from the origin is equal to $\sqrt{2}$.
3. Transformation is applied to each of the two images independently.

V. Calculating Homography matrix using Symmetric transfer error

The symmetric transfer error is used to evaluate the accuracy of a homography transformation between two images by measuring the distance between corresponding points. It helps to optimize the homography matrix $\mathbf{H}$ to achieve the best alignment between two views of a scene.

Given a set of matching points between two images, denoted as $\{(\mathbf{x}_i, \mathbf{x}'_i)\}$, where $\mathbf{x}_i$ and $\mathbf{x}'_i$ represent corresponding points in the source and destination images, respectively, the symmetric transfer error can be computed as follows:

$$\sum_i \left(d(\mathbf{x}_i, \mathbf{H}^{-1}\mathbf{x}'_i)^2 + d(\mathbf{x}'_i, \mathbf{H}\mathbf{x}_i)^2 \right)$$

where:

- $d(\mathbf{a}, \mathbf{b})$ denotes the Euclidean distance between points $\mathbf{a}$ and $\mathbf{b}$.
- $\mathbf{H}$ is the homography matrix that maps points from the first image to the second image.
- $\mathbf{H}^{-1}$ is the inverse of the homography matrix, which maps points from the second image back to the first.

The key steps to calculate the symmetric transfer error are as follows:

- **Step 1: Initialize Corresponding Points**
  Obtain pairs of corresponding points $(\mathbf{x}_i, \mathbf{x}'_i)$ from two images, where $\mathbf{x}_i$ is the point in the first image and $\mathbf{x}'_i$ is the corresponding point in the second image.
- **Step 2: Project Points Using Homography**
  Project each point $\mathbf{x}_i$ in the source image to the destination image using the homography matrix $\mathbf{H}$, and normalize the resulting coordinates. Similarly, project each point $\mathbf{x}'_i$ in the destination image back to the source image using $\mathbf{H}^{-1}$.

- **Step 3: Calculate Euclidean Distances**
  For each pair of points $(\mathbf{x}_i, \mathbf{x}'_i)$, compute the Euclidean distance between $\mathbf{x}_i$ and its re-projected counterpart $\mathbf{H}^{-1}\mathbf{x}'_i$. Similarly, compute the distance between $\mathbf{x}'_i$ and $\mathbf{H}\mathbf{x}_i$.
- **Step 4: Compute Total Symmetric Transfer Error**
  Sum the squared Euclidean distances obtained in Step 3 for all point pairs. This sum represents the symmetric transfer error, which can be minimized to improve the accuracy of the homography matrix.
- **Step 5: Optimize Homography Matrix**
  Using an optimization algorithm, adjust $\mathbf{H}$ to minimize the total symmetric transfer error. The optimized matrix $\mathbf{H}_{\text{opt}}$ provides the best alignment between the two images.

```
class SymmetricTransferError():
    def __init__(self, data):
        total_pairs = len(data)
        if total_pairs < 4:
            raise ValueError(f"Expected at
                least 4 pairs of matching
                points, but got {total_pairs}")
        self.data = data

    def euclidean_distance(self, pt1, pt2):
        return np.linalg.norm(pt1 - pt2)

    def calculate_symmetric_transfer_error(self
        , H):
        """Compute the symmetric transfer error
             for a given homography matrix H.
            """
        error = []
        H = H.reshape(3, 3)

        for (x1, y1), (x2, y2) in self.data:
            src_point = np.array([x1, y1, 1.0])
            dst_point = np.array([x2, y2, 1.0])

            # Project src_point to the
                destination image using H
            projected_dst_point = H @ src_point
            projected_dst_point /=
                projected_dst_point[2]  #
                Normalize

            # Project dst_point back to the
                source image using H^-1
            projected_src_point = np.linalg.inv
                (H) @ dst_point
            projected_src_point /=
                projected_src_point[2]  #
                Normalize

            # Symmetric transfer error
                components
            error.append(self.
                euclidean_distance(src_point
                [:2], projected_src_point[:2]))
            error.append(self.
                euclidean_distance(dst_point
                [:2], projected_dst_point[:2]))

        return error

    def get_homography_matrix(self):
        """Optimize the homography matrix H to
            minimize the symmetric transfer
            error."""
        # Initial estimate of H using cv2.
            findHomography
        pts_src = np.array([pt[0] for pt in
            self.data])
        pts_dst = np.array([pt[1] for pt in
            self.data])
        # H_initial, _ = cv2.findHomography(
            pts_src, pts_dst, method=0)
        H_initial = np.random.rand(3, 3)

        # print(H_initial)

        H_initial_flat = H_initial.flatten()

        # Use least_squares to minimize the
            symmetric transfer error
        result = least_squares(self.
            calculate_symmetric_transfer_error,
             H_initial_flat, method='lm')

        H_optimized = result.x.reshape(3, 3)
        return H_optimized

    # Prints Homography matrix
    def print_homography_matrix(self):
        homography_matrix = self.
            get_homography_matrix()
        print(f"Homography matrix (Dimensions:
            {homography_matrix.shape}):\n",
            homography_matrix)
```

By minimizing the symmetric transfer error, we ensure that the estimated homography matrix provides the closest possible alignment between corresponding points in the two images, resulting in an accurate transformation.

VI. Homography Estimation

The *Homography Estimation* process aims to determine a transformation matrix (homography) that maps points between two images of the same scene from different viewpoints. This transformation is crucial in computer vision tasks such as image alignment, panorama stitching, and perspective correction. Given pairs of corresponding points in two images, homography estimation calculates the matrix that relates these points through a projective transformation.

1) Construct the Data Matrix A from pairs of matching points between the two images. Each pair of points contributes specific constraints to the system of equations used to calculate the homography matrix.
2) Use the *Direct Linear Transformation (DLT)* method to solve for the elements of the homography matrix H. This involves solving a system of linear equations formed by the data matrix A to estimate the transformation.
3) Normalize the computed homography matrix H to ensure consistency in scale, which standardizes the transformation between the two images.
4) Optionally, refine the homography matrix H by minimizing the symmetric transfer error across all matching points, enhancing the accuracy and robustness of the transformation.

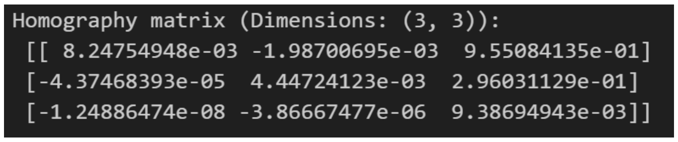
The resulting homography matrix H enables accurate point mapping between the two images, preserving the geometric relationships between corresponding features.

```
class HomographyEstimation:
    def __init__(self, data):
        """data contains pairs of matching
            points:
        1. (x1, y1): matched 2D point in first
            image
        2. (x2, y2): matched 2D point in second
             image
        Note: 1. HX1 = X2
              2. data should have atleast 4
                  pairs of matching points to
                  calculate Projection matrix.
        """
        total_pairs = len(data)
        if total_pairs < 4:
            raise ValueError(f"Expected at
                least 4 pairs of matching
                points, but got {total_pairs}")
        self.data = data

    def get_data_matrix(self):
        data_matrix = []
        for (x1, y1), (x2, y2) in self.data:
            # First row for each pair
            row1 = [
                x1, y1, 1,
                0, 0, 0,
                -x2 * x1, -x2 * y1, -x2
            ]
            # Second row for each pair
            row2 = [
                0, 0, 0,
                x1, y1, 1,
                -y2 * x1, -y2 * y1, -y2
            ]
            data_matrix.append(row1)
            data_matrix.append(row2)
        return np.array(data_matrix)

    def print_data_matrix(self):
        data_matrix = self.get_data_matrix()
        print(f"Data matrix - A (Dimensions: {
            data_matrix.shape}):\n",
            data_matrix)

    def get_homography_matrix(self):
        data_matrix = self.get_data_matrix()
        homo_mat_obj =
            DirectLinearTransformation(
            data_matrix)
        homography_matrix = homo_mat_obj.
            get_homography_matrix()
        return homography_matrix

    def print_homography_matrix(self):
        data_matrix = self.get_data_matrix()
        homo_mat_obj =
            DirectLinearTransformation(
            data_matrix)
        homo_mat_obj.print_homography_matrix()
```

Now, calculate Homography matrix for 2 views of chessboard using *HomographyEstimation* module.

A. *Homography matrix for 2 Views of Chessboard:*

```
matching_points1 = get_data(first_image_points,
     second_image_points)
homo_est1 = HomographyEstimation(
    matching_points1)
homo_est1.print_homography_matrix()
homography_matrix1 = homo_est1.
    get_homography_matrix()
```



```
Homography matrix (Dimensions: (3, 3)):
 [[ 8.24754948e-03 -1.98700695e-03  9.55084135e-01]
 [-4.37468393e-05  4.44724123e-03  2.96031129e-01]
 [-1.24886474e-08 -3.86667477e-06  9.38694943e-03]]
```

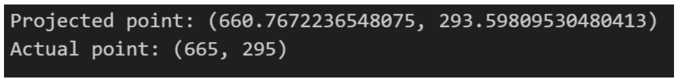
Fig. 8. Homography matrix for Chessboard views

Validating projection matrix:

$$x^{'} = Px$$

here, $x^{'}$ denotes 2D point on second image and x denotes 2D point on first image.

```
first_image_validation_points = [
    (605, 445)
]
second_image_validation_points = [
    (665, 295)
]
```

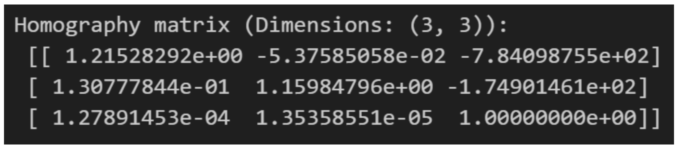


```
Projected point: (660.7672236548075, 293.59809530480413)
Actual point: (665, 295)
```

Fig. 9. Validating projected points on Second image

B. *Homography matrix for 2 Views of Building using RANSAC:*

```
matcher = SIFTMatcher()
matching_points_data = matcher.
    get_matching_points(image1, image2)
top_k = min(100, len(matching_points_data))
matching_points_data = matching_points_data[:
    top_k]
ransac = RansacHomography(matching_points_data)
homography_matrix_ransac, _ = ransac.
    get_ransac_homography_matrix()
ransac.print_ransac_homography_matrix()
```



```
Homography matrix (Dimensions: (3, 3)):
 [[ 1.21528292e+00 -5.37585058e-02 -7.84098755e+02]
 [ 1.30777844e-01  1.15984796e+00 -1.74901461e+02]
 [ 1.27891453e-04  1.35358551e-05  1.00000000e+00]]
```

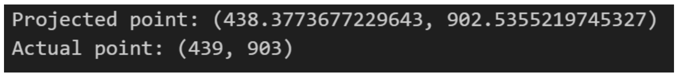
Fig. 10. Homography matrix for Building using RANSAC

Validating projection matrix:

$$x^{'} = Px$$

here, $x^{'}$ denotes 2D point on second image and x denotes 2D point on first image.

```
ransac_validation_points = [
    (1103, 924)
]
ransac_actual_points = [
    (439, 903)
]
```



```
Projected point: (438.3773677229643, 902.5355219745327)
Actual point: (439, 903)
```

Fig. 11. Validating projected points on Second image

VII. RANSAC for Homography Estimation

The *RANSAC (RANdom SAmple Consensus)* algorithm is a robust estimation technique used to estimate the homography matrix, which maps points between two views, even in the presence of outliers. By iteratively sampling subsets of point correspondences and computing the homography matrix for each subset, RANSAC aims to maximize the number of inliers, thus providing a reliable transformation.

1) Randomly select a sample of 4 point correspondences and compute the homography matrix H based on these points.
2) Calculate the reprojection error or distance d for each remaining correspondence, measuring how well H transforms each point pair.
3) Count number of inliers, which are correspondences with reprojection error less than a defined threshold t.
4) Adaptively repeat the sampling and homography calculation process to maximize the number of inliers, adjusting the number of iterations N based on the inlier ratio observed so far.
5) Choose the homography matrix H with the highest inlier count as the final estimated homography.

```
class RansacHomography:
    def __init__(self, data):
        total_pairs = len(data)
        if total_pairs < 4:
            raise ValueError(f"Expected at
                least 4 pairs of matching
                points, but got {total_pairs}")
        self.data = data

    def compute_threshold_percentile_distance(
        self, percentile=90):
        distances = []
        for (src_pt, dst_pt) in self.data:
            distance = np.linalg.norm(np.array(
                src_pt) - np.array(dst_pt))
            distances.append(distance)

        # Set threshold as the chosen
            percentile of the distances
        threshold = np.percentile(distances,
            percentile)
        return threshold

    # Returns Homography Matrix of size (3, 3)
    def get_ransac_homography_matrix(self,
        initial_p=0.99, s=4):
        best_H = None
        best_inliers = []
        sample_count = 0
        N = float(’inf’)  # Start with N as
            infinity for adaptive stopping
        threshold = self.
            compute_threshold_percentile_distance
            ()

        while sample_count < N:
            # Randomly sample 4 correspondences
            sample = random.sample(self.data, s
                )
            pts_src = np.float32([pt[0] for pt
                in sample])
            pts_dst = np.float32([pt[1] for pt
                in sample])

            # Compute homography
            H, _ = cv2.findHomography(pts_src,
                pts_dst, method=0)

            # Calculate distance for each
                correspondence and count
                inliers
            inliers = []
            for (src_pt, dst_pt) in self.data:
                src_pt_h = np.array([src_pt[0],
                     src_pt[1], 1]).reshape(3,
                    1)
                dst_pt_estimated = H @ src_pt_h
                if dst_pt_estimated[2, 0] == 0:
                    dst_pt_estimated[2, 0] = 1e
                        -10
                dst_pt_estimated /=
                    dst_pt_estimated[2, 0]   #
                    Normalize by z-coordinate

                distance = np.linalg.norm([
                    dst_pt[0] -
                    dst_pt_estimated[0, 0],
                    dst_pt[1] -
                    dst_pt_estimated[1, 0]])
                if distance < threshold:
                    inliers.append((src_pt,
                        dst_pt))

            # Update best model if this one has
                 more inliers
            if len(inliers) > len(best_inliers)
                :
                best_H = H
                best_inliers = inliers

                # Update w (inlier ratio) and
                    adaptively calculate N
                w = len(best_inliers) / len(
                    self.data)
                if w > 0:
                    if 1 - w**s > 0:
                        denominator = np.log(1
                            - w**s)
                    else:
                        denominator = 1e6
                    N = max(1000, 1e18 * np.log
                        (1 - initial_p) /
                        denominator)

            # Increment sample count
            sample_count += 1

        # print(sample_count)
        return best_H, best_inliers

    # Prints Homography matrix
    def print_ransac_homography_matrix(self,
        initial_p=0.99, s=4):
        homography_matrix, _ = self.
            get_ransac_homography_matrix(
            initial_p, s)
        print(f"Homography matrix (Dimensions:
            {homography_matrix.shape}):\n",
            homography_matrix)
```

This approach ensures that the homography matrix is computed with minimal influence from outliers, resulting in a robust estimation that aligns the images accurately.

VIII. Observations and Results

1) **Performance of SIFT-Based Matching:** SIFT (Scale-Invariant Feature Transform) performs inconsistently depending on the nature of the scene. In tests with a chessboard pattern, SIFT-based matching yielded poor results due to the repetitive nature of the pattern, which leads to multiple similar features and ambiguity in matching. In contrast, SIFT demonstrated reliable performance with images of natural scenes, such as buildings, where features are more distinct and less repetitive.
2) **Dependency of Symmetric Transfer Error on Homography Initialization:** The accuracy of the Symmetric Transfer Error metric is highly sensitive to the initial values of the homography matrix. Random initialization of the homography matrix can lead to poor convergence, which suggests that careful or informed initialization is essential to achieve accurate error minimization and reliable results.
3) **Manual Point Matching with Direct Linear Transformation (DLT) and Singular Value Decomposition (SVD):** When using DLT combined with SVD for homography estimation, manual matching of points provides accurate results. The manual selection of points allows for precise control, reducing the likelihood of erroneous matches, which is especially useful when the number of points is limited. This method is preferred in situations where high accuracy is required with a limited set of correspondences.
4) **SIFT-Based Matching Combined with RANSAC for Robust Homography Estimation:** In scenarios requiring a large number of correspondences to ensure robustness, SIFT-based matching paired with RANSAC is highly effective. RANSAC, being an iterative approach, benefits from the high volume of feature points generated by SIFT, allowing it to filter out outliers and compute a homography matrix that is more resilient to noise and mismatches.
5) **Handling Different Image Sizes in Homography Computation:** When dealing with images of different sizes or resolutions, homography estimation requires additional preprocessing steps. Simply applying a homography matrix between images of different dimensions produces inaccurate results. Therefore, either resizing one image to match the other or scaling the coordinates of matched points is essential for accurate homography computation. This preprocessing ensures the coordinate transformations remain consistent across different image resolutions.

IX. Conclusion

This report demonstrates a systematic approach to homography estimation through the integration of feature detection, linear transformation, and robust estimation techniques. The use of SIFT for identifying matching points provides a strong foundation for accurate feature correspondence, while the Direct Linear Transformation (DLT) method offers an initial homography estimate. The application of Symmetric Transfer Error refines this estimate, improving alignment precision, and RANSAC effectively mitigates the impact of outliers, ensuring a robust final homography matrix.

The combination of these methods enables accurate and efficient homography estimation, making it suitable for various computer vision applications. Future work could explore improvements in computational efficiency for real-time applications or investigate alternative feature detection techniques to further enhance robustness in complex scenes with significant variation in lighting or viewpoint. Overall, the homography estimation pipeline presented in this report provides a reliable framework for image alignment in both academic and practical computer vision tasks.

References

1. First Principles of Computer Vision : Computing Homography (Youtube channel)