Mobile Camera Calibration Using Known Geometry Objects

Suyash Ajit Chavan (IMT2021048)
3D Vision

***Abstract*—This report presents a concise method for calibrating mobile cameras using known geometric objects like a Rubik's cube or chessboard. By capturing multiple views of the object and applying Singular Value Decomposition (SVD) and least squares, we estimate camera intrinsic and extrinsic parameters. The method, demonstrated through Python code, offers an accessible calibration solution suitable for mobile applications in computer vision and augmented reality.**

GitHub Repository: Camera_Calibration

I. Introduction

Camera calibration is a critical preprocessing step in various computer vision applications, enabling precise measurements and 3D reconstructions. Calibration techniques rely on estimating the intrinsic and extrinsic parameters of the camera, which define the geometric relationship between the 3D world and its 2D image projection. Traditionally, calibration methods employ checkerboards or grids; however, using alternative objects of known geometry, like a Rubik's cube, presents a versatile and accessible solution, particularly for mobile applications.

This report explores a method of mobile camera calibration using a Rubik's cube as the primary calibration object. The process involves capturing multiple perspectives of the object and identifying key correspondences between 3D coordinates in the scene and 2D image points. Using these correspondences, a projection matrix is computed by applying Singular Value Decomposition (SVD) and least squares optimization to minimize projection error. Through Python code snippets embedded within each stage, this report provides a detailed walkthrough of the calibration process, offering a practical, hands-on approach for implementation. The method is suitable for mobile device cameras, paving the way for applications in augmented reality, 3D modeling, and robotics where accurate image measurements are crucial.

II. Data Generation using Rubik's Cube

1. A *Rubik's cube* (an object of known geometry) will be utilized, ensuring the side length is known. If the side length is unknown, it will be calculated using a scale (in cm).

2. Multiple views of the Rubik's cube will be captured, as illustrated in the following images, using a *mobile camera* (*Vivo V40 Pro* mobile device is used for this project).

Fig. 1. Rubik's cube View 1

Fig. 2. Rubik's cube View 2

3. Identify correspondences between points in the 3D scene: (x, y, z) and their respective image points: (u, v) through manual identification, primarily focusing on corner points and edge points.
We require **atleast 6 points** to calibrate camera. The image points indicated can be plotted on the image as shown.

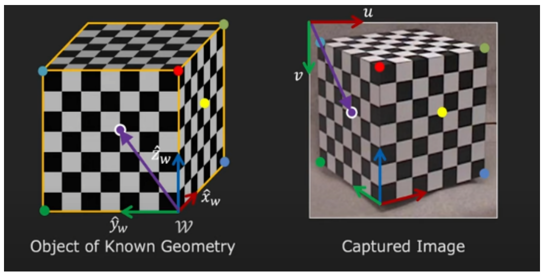


Fig. 3. Data generation

a. for View 1:

```
cube_side = 4.8 # cube side is in cm

world_frame_points1 = [
    (0, 0, 0), # origin of world frame
    (cube_side, 0, 0), # point on x-axis of
        world frame
    (0, cube_side, 0), # point on y-axis of
        world frame
    (0, 0, cube_side), # point on z-axis of
        world frame
    (cube_side, 0, cube_side), # right corner
    (0, cube_side, cube_side) # left corner
]

image_points1 = [
    (373, 807), # corresponding point to origin
         of world frame
    (480, 648), # corresponding point to point
        on x-axis of world frame
    (105, 710), # corresponding point to point
        on y-axis of world frame
    (384, 515), # corresponding point to point
        on z-axis of world frame
    (520, 378), # corresponding point to right
        corner
    (58, 432) # corresponding point to left
        corner
]

plot_points_on_image(img_path1, image_points1)
```

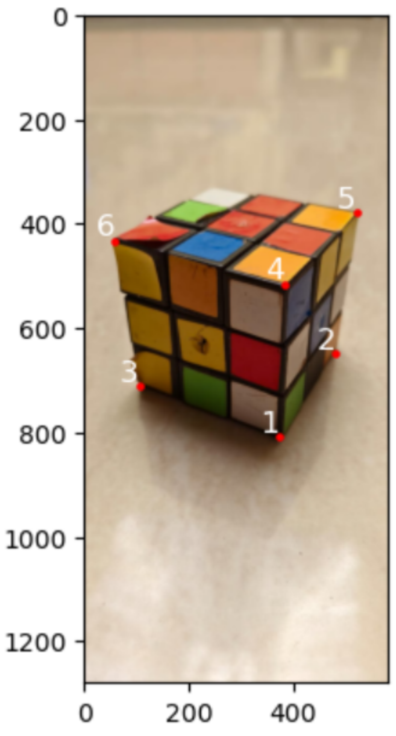


Fig. 4. Matching image points from View 1

b. for View 2:

```
cube_side = 4.8 # cube side is in cm

world_frame_points2 = [
    (0, 0, 0), # origin of world frame
    (cube_side, 0, 0), # point on x-axis of
        world frame
    (0, cube_side, 0), # point on y-axis of
        world frame
    (0, 0, cube_side), # point on z-axis of
        world frame
    (cube_side, 0, cube_side), # right corner
    (0, cube_side, cube_side) # left corner
]

image_points2 = [
    (650, 490), # corresponding point to origin
         of world frame
    (730, 360), # corresponding point to point
        on x-axis of world frame
    (410, 440), # corresponding point to point
        on y-axis of world frame
    (630, 230), # corresponding point to point
        on z-axis of world frame
    (735, 130), # corresponding point to right
        corner
    (355, 190) # corresponding point to left
        corner
]

plot_points_on_image(img_path2, image_points2)
```

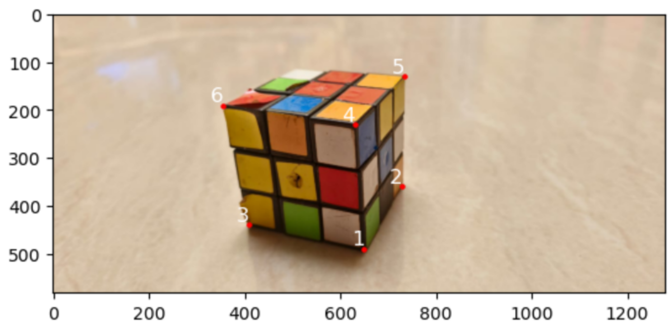


Fig. 5. Matching image points from View 2

4. Construct the final data as a list of pairs comprising 3D scene points and their corresponding image points:

$$data = [((x_1, y_1, z_1), (u_1, v_1)), \dots, ((x_n, y_n, z_n), (u_n, v_n))]$$

,where n denotes total number of correspondences.

```
data1 = get_data(world_frame_points1,
    image_points1)
print(data1)
```

III. Calculating Projection Matrix

The projection matrix (P), with dimensions 3×4, facilitates the conversion of three-dimensional (3D) world points into two-dimensional (2D) image coordinates. It encapsulates the intrinsic and extrinsic parameters of the camera, thereby defining the geometric relationship between the 3D scene and the captured images.

$$P = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{bmatrix}$$

1. For each matching pair i in scene and image (data), define equations as:

$$\underbrace{\begin{bmatrix} u^{(i)} \\ v^{(i)} \\ 1 \end{bmatrix}}_{\text{Known}} = \underbrace{\begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{bmatrix}}_{\text{Unknown}} \underbrace{\begin{bmatrix} X_w^{(i)} \\ Y_w^{(i)} \\ Z_w^{(i)} \\ 1 \end{bmatrix}}_{\text{Known}}$$

2. Expanding the matrix as linear equations:

$$u^{(i)} = \frac{P_{11}X_w^{(i)} + P_{12}Y_w^{(i)} + P_{13}Z_w^{(i)} + P_{14}}{P_{31}X_w^{(i)} + P_{32}Y_w^{(i)} + P_{33}Z_w^{(i)} + P_{34}}$$

$$v^{(i)} = \frac{P_{21}X_w^{(i)} + P_{22}Y_w^{(i)} + P_{23}Z_w^{(i)} + P_{24}}{P_{31}X_w^{(i)} + P_{32}Y_w^{(i)} + P_{33}Z_w^{(i)} + P_{34}}$$

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

$$\underbrace{\mathbf{A}}_{\text{Known}}\underbrace{\mathbf{p}}_{\text{Unknown}} = \mathbf{0}$$

Here, A is *Data matrix* and p is *Projection vector*.

$$\begin{bmatrix} x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & 0 & 0 & 0 & 0 & -u_1x_w^{(1)} & -u_1y_w^{(1)} & -u_1z_w^{(1)} & -u_1 \\ 0 & 0 & 0 & 0 & x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & -v_1x_w^{(1)} & -v_1y_w^{(1)} & -v_1z_w^{(1)} & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & 0 & 0 & 0 & 0 & -u_ix_w^{(i)} & -u_iy_w^{(i)} & -u_iz_w^{(i)} & -u_i \\ 0 & 0 & 0 & 0 & x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & -v_ix_w^{(i)} & -v_iy_w^{(i)} & -v_iz_w^{(i)} & -v_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & 0 & 0 & 0 & 0 & -u_nx_w^{(n)} & -u_ny_w^{(n)} & -u_nz_w^{(n)} & -u_n \\ 0 & 0 & 0 & 0 & x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & -v_nx_w^{(n)} & -v_ny_w^{(n)} & -v_nz_w^{(n)} & -v_n \end{bmatrix} \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Fig. 6. Data matrix (A) and equation Ap = 0

4. Solve for p (projection vector):

$$\mathbf{Ap} = \mathbf{0}$$

A. Least Sqaures Solution for p (Projection vector):

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

$$\min_p \|Ap\|^2 \text{ such that } \|p\|^2 = 1$$

Equivalently:

$$\min_p (p^TA^TAp) \text{ such that } p^Tp = 1$$

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

$$L(p, \lambda) = p^TA^TAp - \lambda(p^Tp - 1)$$

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

$$2A^TAp - 2\lambda p = 0$$

$$\boxed{A^TAp = \lambda p} \qquad \text{(Eigenvalue Problem)}$$

$\therefore$ Eigenvector p with smallest eigenvalue λ of matrix A^TA minimizes the loss function $L(p)$.

$\Rightarrow$ projection vector (p) = Right singular vector of A corresponding to smallest singular value.
Rearrange the *Projection vector (p)* to get *Projection matrix (P)* of dimensions (3, 4).

```python
# Calculates Projection Matrix by using Least
    Sqaures Solution and SVD
class ProjectionMatrix:
    def __init__(self, A):
        _, cols = A.shape
        if cols != 12:
            raise ValueError("Expected number
                of columns is 12, but got {cols
                }")
        self.A = A

    # Returns Projection Vector, which is right
         singular vector of A corresponding to
        smallest Eigen value.
    def get_projection_vector(self):
        svd = SVD(self.A)
        _, cols = self.A.shape
        projection_vector = svd.
            get_right_singular_vector(cols)
        if len(projection_vector) != 12:
            raise ValueError(f"Expected length
                of 12, but got {len(
                projection_vector)}")
        return projection_vector

    # Returns Projection Matrix of size (3, 4)
        using Projection vector
    def get_projection_matrix(self):
        projection_vector = self.
            get_projection_vector()
        projection_matrix = projection_vector.
            reshape(3, 4)
        return projection_matrix

    # Prints Projection vector
    def print_projection_vector(self):
        projection_vector = self.
            get_projection_vector()
        print(f"Projection vector (Dimensions:
            {projection_vector.shape}):\n",
            projection_vector)

    # Prints Projection matrix
    def print_projection_matrix(self):
        projection_matrix = self.
            get_projection_matrix()
        print(f"Projection matrix (Dimensions:
            {projection_matrix.shape}):\n",
            projection_matrix)
```

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

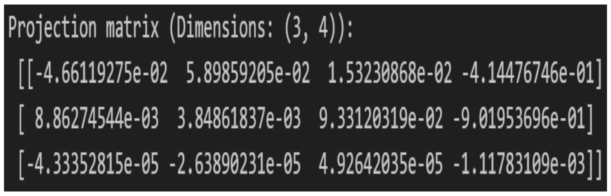
Singular Value Decomposition (SVD) is a matrix factorization technique that decomposes a matrix A into the product of three matrices: U, Σ, and V^T. Here, U and V^T are orthogonal matrices of row and column space, and Σ is a diagonal matrix containing the singular values of A.

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

```python
# 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, :]
```

Finally, Projection matrix for View 1 is calculated using *ProjectionMatrix* module as :

```python
proj_matrix = ProjectionMatrix(A)
projection_matrix = proj_matrix.
    get_projection_matrix()

proj_matrix.print_projection_vector()
proj_matrix.print_projection_matrix()
```



```
Projection matrix (Dimensions: (3, 4)):
 [[-4.66119275e-02  5.89859205e-02  1.53230868e-02 -4.14476746e-01]
 [ 8.86274544e-03  3.84861837e-03  9.33120319e-02 -9.01953696e-01]
 [-4.33352815e-05 -2.63890231e-05  4.92642035e-05 -1.11783109e-03]]
```

Fig. 7. Projection matrix for View 1

IV. Decomposition of Projection Matrix into Intrinsic and Extrinsic Matrices

We know that, projection matrix P can be written as :

$$P = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{bmatrix} = \begin{bmatrix} f_x & s & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

That is:

$$M = \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix} = \begin{bmatrix} f_x & s & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = KR$$

Here, K and R represents Intrinsic and Rotation matrix respectively.

$\Rightarrow$ Given that K is an Upper Right Triangular matrix and R is an Orthonormal matrix, it is possible to uniquely "decouple" K and R from their product using "RQ factorization".

To calculate translation vector (t):

$$\begin{bmatrix} P_{14} \\ P_{24} \\ P_{34} \end{bmatrix} = \begin{bmatrix} f_x & s & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = Kt$$

Therefore:

$$t = K^{-1} \begin{bmatrix} P_{14} \\ P_{24} \\ P_{34} \end{bmatrix}$$

```python
# This module Decomposes Projection Matrix into
     Intrinsic (I) and Extrinsic Matrices (E).
# Extrinsic Matrix is further decomposed into
    Rotation Matrix and Translation Vector.
class IEMatrices:
    def __init__(self, P):
        if P.shape != (3, 4):
            raise ValueError(f"Expected shape
                of P is (3, 4), but got {P.
                shape}")
        self.P = P
    # Decompose M ((3, 3): first 3 columns of P
        ) into K (I: Intrinsic Matrix) and R (
        Rotation matrix: part of E (Extrinsic
        Matrix)).
    def get_KR(self):
        M = self.P[:, :3]
        K, R = sp.linalg.rq(M)
        return K, R

    # Returns t (translation vector) using K
        and last column of P
    def get_t(self):
        K, _ = self.get_KR()
        if np.linalg.det(K) == 0:
            raise ValueError(f"K is singular
                Matrix, Inverse does not exist.
                ")

        K_inv = np.linalg.inv(K)
        p4 = self.P[:, 3]
        t = K_inv @ p4
        return t

    # Return K (Intrinsic matrix), R (Rotation
        matrix) and t (Translation vector)
    def get_KRt(self):
        K, R = self.get_KR()
        t = self.get_t()
        return K, R, t

    def print_KRt(self):
        K, R = self.get_KR()
        t = self.get_t()
        print(f"Intrinsic matrix - K (
            Dimensions: {K.shape}):\n", K)
        print(f"Rotation matrix - R (Dimensions
            : {R.shape}):\n", R)
        print(f"Translation vector - t (
            Dimensions: {t.shape}):\n", t)
```

V. Camera Calibration

The *Camera Calibration* process involves estimating a camera's internal parameters and external parameters to improve image accuracy and remove distortions. By using multiple images of a known calibration object (like a rubik's cube), calibration algorithms compute the intrinsic parameters (focal length, optical center) and distortion coefficients.

1. Construct Data Matrix (A) from the data generated using the Rubik's cube, as outlined in Section II.

2. For each view, calculate Internal and External parameters such as Projection matrix (P), Intrinsic matrix (K), Rotation Matrix (R) and Translation vector (t).

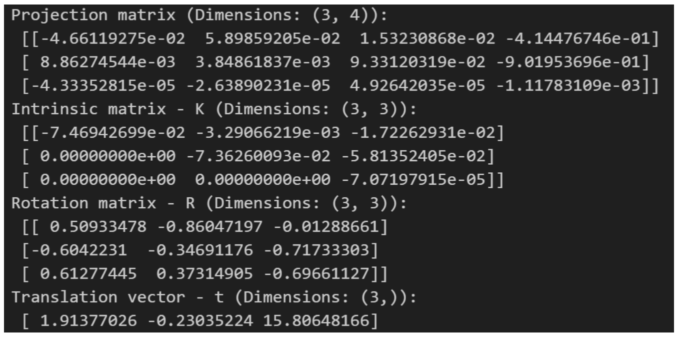
3. Calculate the Intrinsic Matrix (K) across multiple views to refine the estimation of the camera's internal parameters.

```
class CameraCalibration:
    def __init__(self, data):
        """data contains pairs of matching
            points:
        1. (x, y, z): 3D point in world co-
            ordinate frame with origin at one
            of the corners of Rubics cube/
            Chessboard
        2. (u, v): 2D point in captured image
            with origin at left top corner
        Note: data should have atleast 6 pairs
            of matching points to calculate
            Projection matrix.
        """
        total_pairs = len(data)
        if total_pairs < 6:
            raise ValueError(f"Expected at
                least 6 pairs of matching
                points, but got {total_pairs}")
        self.data = data

    def get_data_matrix(self):
        data_matrix = []
        for (x_w, y_w, z_w), (u, v) in self.
            data:
            # First row for each pair
            row1 = [
                x_w, y_w, z_w, 1,
                0, 0, 0, 0,
                -u * x_w, -u * y_w, -u * z_w, -
                    u
            ]
            # Second row for each pair
            row2 = [
                0, 0, 0, 0,
                x_w, y_w, z_w, 1,
                -v * x_w, -v * y_w, -v * z_w, -
                    v
            ]
            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_camera_projection_matrix(self):
        data_matrix = self.get_data_matrix()
        proj_mat_obj = ProjectionMatrix(
            data_matrix)
        projection_matrix = proj_mat_obj.
            get_projection_matrix()
        return projection_matrix

    def print_camera_projection_matrix(self):
        data_matrix = self.get_data_matrix()
        proj_mat_obj = ProjectionMatrix(
            data_matrix)
        proj_mat_obj.print_projection_matrix()

    def get_camera_KRt(self):
        projection_matrix = self.
            get_camera_projection_matrix()
        ieMats_obj = IEMatrices(
            projection_matrix)
        K, R, t = ieMats_obj.get_KRt()
        return K, R, t

    def print_camera_KRt(self):
        projection_matrix = self.
            get_camera_projection_matrix()
        ieMats_obj = IEMatrices(
            projection_matrix)
        ieMats_obj.print_KRt()
```

Now, calculate Intrinsic and extrinsic matrices for 2 views of rubik's cube using *CameraCalibration* module.

A. *Intrinsic and Extrinsic matrices for View 1:*

```
camera_cal1 = CameraCalibration(data1)
camera_cal1.print_data_matrix()
camera_cal1.print_camera_projection_matrix()
projection_matrix1 = camera_cal1.
    get_camera_projection_matrix()
camera_cal1.print_camera_KRt()
```



```
Projection matrix (Dimensions: (3, 4)):
 [[-4.66119275e-02  5.89859205e-02  1.53230868e-02 -4.14476746e-01]
 [ 8.86274544e-03  3.84861837e-03  9.33120319e-02 -9.01953696e-01]
 [-4.33352815e-05 -2.63890231e-05  4.92642035e-05 -1.11783109e-03]]
Intrinsic matrix - K (Dimensions: (3, 3)):
 [[-7.46942699e-02 -3.29066219e-03 -1.72262931e-02]
 [ 0.00000000e+00 -7.36260093e-02 -5.81352405e-02]
 [ 0.00000000e+00  0.00000000e+00 -7.07197915e-05]]
Rotation matrix - R (Dimensions: (3, 3)):
 [[ 0.50933478 -0.86047197 -0.01288661]
 [-0.6042231  -0.34691176 -0.71733303]
 [ 0.61277445  0.37314905 -0.69661127]]
Translation vector - t (Dimensions: (3,)):
 [ 1.91377026 -0.23035224 15.80648166]
```

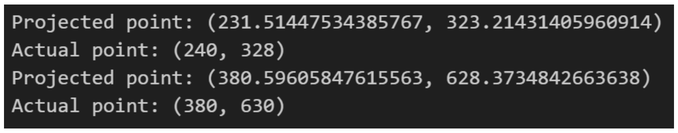
Fig. 8. Intrinsic and Extrinsic matrices for View 1

Validating projection matrix:

$$x = PX$$

here, x denotes 2D point on image and X denotes 3D point from scene.

```
validation_world_frame_points1 = [
    (cube_side, cube_side, cube_side), # back
        corner
    (0, 0, 2*cube_side/3)
]
validation_image_points1 = [
    (240, 328), # corresponding point to back
        corner
    (380, 630)
]
```

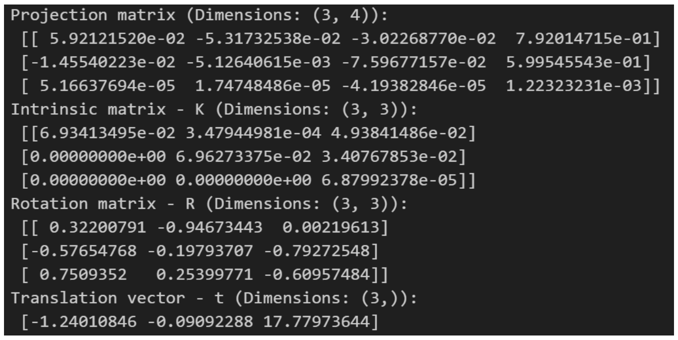


```
Projected point: (231.51447534385767, 323.21431405960914)
Actual point: (240, 328)
Projected point: (380.59605847615563, 628.3734842663638)
Actual point: (380, 630)
```

Fig. 9. Validating projected points of View 1

B. *Intrinsic and Extrinsic matrices for View 2:*

```
camera_cal2 = CameraCalibration(data2)
camera_cal2.print_data_matrix()
camera_cal2.print_camera_projection_matrix()
projection_matrix2 = camera_cal2.
    get_camera_projection_matrix()
camera_cal2.print_camera_KRt()
```



```
Projection matrix (Dimensions: (3, 4)):
 [[ 5.92121520e-02 -5.31732538e-02 -3.02268770e-02  7.92014715e-01]
 [-1.45540223e-02 -5.12640615e-03 -7.59677157e-02  5.99545543e-01]
 [ 5.16637694e-05  1.74748486e-05 -4.19382846e-05  1.22323231e-03]]
Intrinsic matrix - K (Dimensions: (3, 3)):
 [[6.93413495e-02 3.47944981e-04 4.93841486e-02]
 [0.00000000e+00 6.96273375e-02 3.40767853e-02]
 [0.00000000e+00 0.00000000e+00 6.87992378e-05]]
Rotation matrix - R (Dimensions: (3, 3)):
 [[ 0.32200791 -0.94673443  0.00219613]
 [-0.57654768 -0.19793707 -0.79272548]
 [ 0.7509352   0.25399771 -0.60957484]]
Translation vector - t (Dimensions: (3,)):
 [-1.24010846 -0.09092288 17.77973644]
```

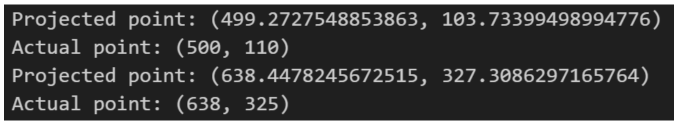
Fig. 10. Intrinsic and Extrinsic matrices for View 2

Validating projection matrix:

$$x = PX$$

here, x denotes 2D point on image and X denotes 3D point from scene.

```
validation_world_frame_points2 = [
    (cube_side, cube_side, cube_side), # back
        corner
    (0, 0, 2*cube_side/3)
]
validation_image_points2 = [
    (500, 110), # corresponding point to back
        corner
    (638, 325)
]
```



```
Projected point: (499.2727548853863, 103.73399498994776)
Actual point: (500, 110)
Projected point: (638.4478245672515, 327.3086297165764)
Actual point: (638, 325)
```

Fig. 11. Validating projected points of View 2

C. *Calculating Internal Parameter of Camera:*

We can identify internal parameters of camera using K (Intrinsic matrix) as follows:

Parameter	**View 1**	**View 2**
Focal length in x-direction (f_x)	7.46e-02	6.93e-02
Focal length in y-direction (f_y)	7.36e-02	6.96e-02
Principal point in x-direction (p_x)	1.72e-02	4.93e-02
Principal point in y-direction (p_y)	5.81e-02	3.40e-02
Skew parameter (s)	3.29e-03	3.47e-04

TABLE I
CAMERA CALIBRATION PARAMETERS FOR 2 VIEWS

VI. OBSERVATIONS

1) **Effectiveness of Manual Point Matching:** The manual matching of 3D-2D point correspondences significantly improves the accuracy of the calibration results, allowing for more precise computation of camera parameters.
2) **Consistency of Internal Parameters Across Views:** Calculations of the internal camera parameters using

data from two separate views yield highly similar values, demonstrating the robustness and reliability of the calibration algorithm employed.

3) **Insight into Camera Geometry through Intrinsic Matrix and Extrinsic Parameters:** The intrinsic matrix K, rotation matrix R, and translation vector t obtained from both views provide valuable insights into the camera's internal structure and relative positioning in space, enhancing the understanding of the camera's spatial orientation and internal configuration.
4) **Minimal Projection Error through Optimization Techniques:** The use of Singular Value Decomposition (SVD) and least-squares optimization effectively minimizes projection errors, validating the accuracy of the computed projection matrix and the overall calibration results.
5) **Applicability for Mobile Devices:** The proposed method demonstrates potential applicability for mobile device cameras, showing that reliable calibration is feasible even without traditional checkerboard patterns, making it accessible for on-the-go computer vision tasks.
6) **Feasibility under Different Viewing Angles:** The calibration process maintains accuracy across varying viewing angles, suggesting that the method is adaptable and can handle perspective changes effectively.

VII. Conclusion

This report presented a mobile camera calibration method using a Rubik's cube as a calibration object. By leveraging known 3D-2D correspondences from multiple viewpoints, we computed the camera's intrinsic and extrinsic parameters through Singular Value Decomposition (SVD) and least-squares optimization. The proposed approach offers a practical and accessible solution for camera calibration in mobile applications, especially in scenarios where standard calibration tools may not be readily available.

With accurate calibration, this method has potential applications in augmented reality, 3D reconstruction, and robotics, where precise spatial measurements are essential. Future work could explore enhancing robustness under varying lighting conditions and refining accuracy through additional optimization techniques.

References

1. First Principles of Computer Vision : Camera Calibration (Youtube channel)