Computer Vision, Assignment 1

Calibration and DLT

CID: karllun (cooperated partly with Erik Norlin)

2 - Calibrated vs uncalibrated reconstruction

Theoretical exercise 1

We want to show that if $\mathbf{x} \sim P\mathbf{X}$, then for some new camera matrix P' we can have $\mathbf{x} \sim P'T\mathbf{X}$ where T is a 3D-transformation. This is possible with $P' = PT^{-1}$ such that $\mathbf{x} \sim PT^{-1}T\mathbf{X}$.

In [ ]:
```
import numpy as np
from numpy.typing import ArrayLike
import matplotlib.pyplot as plt
import matplotlib.transforms as transforms
from scipy.io import loadmat
from icecream import ic
import cv2 as cv
import os
%matplotlib inline
# plt.switch_backend('TkAgg')
%config InlineBackend.figure_format = 'retina'
# from matplotlib import rc
# rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
# rc('text', usetex=True)
```

In [ ]:
```
def assemble_points(points: ArrayLike) -> np.ndarray:
    """
    Takes a list of 1d numpy arrays and assembles them in a matrix as column vectors
```

```
    Args:
    list_of_points: list of 1d numpy arrays

    Returns:
    A matrix with the input arrays as column vectors
    """
    if type(points) is list:
        return np.column_stack(points)
    else:
        return np.column_stack(points).T


def to_degrees(angles: ArrayLike):
    return np.array(angles) / np.pi * 180


def to_radians(angles: ArrayLike):
    return np.array(angles) / 180 * np.pi


class Points:
    NOT_HOMOGENEOUS = "Points need to be in homogeneous coordinates"

    def __init__(self, points: list, is_euclidian: bool, is_normalized: bool = False):
        self.data = assemble_points(points)
        self.is_euclidian = is_euclidian
        self.is_normalized = is_normalized
        self.homogenize()

    def __getattr__(self, name):
        if name in self.__dict__:
            return self.__dict__[name]
        else:
            return self.data.__getattribute__(name)

    def pflat(self):
        if not self.is_euclidian:
            if np.any(self.data[-1] == 0):
                raise ValueError("Homogeneous coordinate is zero")
```

```
            self.data = self.data / self.data[-1]
        return self.data

    def dehomogenize(self):
        self.pflat()
        # Remove the last row (homogeneous coordinate)
        if not self.is_euclidian:
            self.data = self.data[:-1]
            self.is_euclidian = True
        return self.data

    def homogenize(self):
        if self.is_euclidian:
            self.data = np.vstack((self.data, np.ones(self.data.shape[1])))
            self.is_euclidian = False
        else:
            self.pflat()
        self.data = self.data.squeeze()
        return self.data

    def normalize(self):
        """Normalizes a numpy array to have a mean of 0 and standard deviation of 1.

        Args:
            data (numpy.array): The data to be normalized.

        Returns:
            numpy.array: The normalized data.
        """
        if not self.is_normalized:
            is_euclidian = self.is_euclidian
            self.dehomogenize()
            self.mean = np.mean(self.data, axis=1).reshape(-1, 1)
            self.std = np.std(self.data, axis=1).reshape(-1, 1)
            self.data = (self.data - self.mean) / self.std
            self.is_normalized = True
            if self.data.shape[0] == 2:
                self.normalization_matrix = np.array(
                    [
                        [1 / self.std[0][0], 0, -self.mean[0][0] / self.std[0][0]],
                        [0, 1 / self.std[1][0], -self.mean[1][0] / self.std[1][0]],
```

```
                    [0, 0, 1],
                ]
            )

        if not is_euclidian:
            self.data = self.homogenize()
    else:
        print("Data is already normalized")
    return self.data

def denormalize(self):
    """Denormalizes a numpy array to restore it from having a mean of 0 and standard deviation of 1.

    Returns:
        numpy.array: The denormalized data.
    """
    if self.is_normalized:
        is_euclidian = self.is_euclidian
        self.dehomogenize()
        self.data = (self.data * self.std) + self.mean
        self.is_normalized = False

        if not is_euclidian:
            self.data = self.homogenize()
    else:
        print("Data is not normalized")

    return self.data

def plot(self, ax=None, marker_size=5):
    def plot_2d(self):
        nonlocal ax
        if ax is None:
            fig = plt.figure()
            ax = fig.add_subplot()
        ax.scatter(self.data[0], self.data[1], s=marker_size)
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.grid()
        ax.set_aspect("equal")
        return ax
```

```
        def plot_3d(self):
            nonlocal ax
            if ax is None:
                fig = plt.figure()
                ax = fig.add_subplot(projection="3d")
            ax.scatter(self.data[0], self.data[1], self.data[2], s=marker_size)
            ax.set_xlabel("X")
            ax.set_ylabel("Y")
            ax.set_zlabel("Z")
            ax.set_aspect("equal")
            return ax

        if self.data.shape[0] == 2:
            ax = plot_2d(self)
            ax.set_title("Euclidian space")
        elif self.data.shape[0] == 3 and self.is_euclidian:
            ax = plot_3d(self)
            ax.set_title("Euclidian space")
        elif self.data.shape[0] == 3 and not self.is_euclidian:
            ax = plot_2d(self)
            # ax.set_title("Projective space")
        elif self.data.shape[0] == 4:
            ax = plot_3d(self)
            # ax.set_title("Projective space")
        else:
            raise ValueError("Can only plot points in 2D or 3D")
        return ax

    def translate(self, t: ArrayLike, lambda_: float = 1.0):
        if not self.is_euclidian:
            I = np.eye(self.data.shape[0] - 1)
            zeros = np.zeros((self.data.shape[0] - 1, 1))
            t = t.reshape(-1, 1)
            H = np.block([[I, t], [zeros.T, 1]])
            self.latest_operation = lambda_ * H
            self.data = self.latest_operation @ self.data
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)
        return self.data
```

```
    def rotate(self, angles: ArrayLike, scaling: float = 1.0, lambda_: float = 1.0):
        """
        angles should be given in degrees
        """
        if not self.is_euclidian:
            angles = to_radians(angles)
            if self.data.shape[0] == 4:
                omega, phi, kappa = angles
                rotation_x = np.array(
                    [
                        [1, 0, 0],
                        [0, np.cos(omega), -np.sin(omega)],
                        [0, np.sin(omega), np.cos(omega)],
                    ]
                )
                rotation_y = np.array(
                    [[np.cos(phi), 0, np.sin(phi)], [0, 1, 0], [-np.sin(phi), 0, np.cos(phi)]]
                )
                rotation_z = np.array(
                    [
                        [np.cos(kappa), -np.sin(kappa), 0],
                        [np.sin(kappa), np.cos(kappa), 0],
                        [0, 0, 1],
                    ]
                )
                zeros = np.zeros((3, 1))

                rotation = rotation_z @ rotation_y @ rotation_x
            elif self.data.shape[0] == 3:
                (angle,) = angles
                rotation = np.array(
                    [[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.cos(angle)]]
                )
                zeros = np.zeros((2, 1))
            else:
                raise ValueError("Amount of angles must be either 2 or 3")

            H = np.block([[scaling * rotation, zeros], [zeros.T, 1]])
            self.latest_operation = lambda_ * H
            self.data = self.latest_operation @ self.data
            self.homogenize()
```

```
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)
        return self.data, self.latest_operation

    def rigid_body_transformation(self, angles: ArrayLike, t: ArrayLike, lambda_: float = 1.0):
        if not self.is_euclidian:
            self.rotate(angles)
            first_operation = self.latest_operation.copy()
            self.translate(t, lambda_)
            self.latest_operation = self.latest_operation @ first_operation
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)
        return self.data

    def similarity_transformation(
        self, angles: ArrayLike, t: ArrayLike, scaling: float, lambda_: float = 1.0
    ):
        if not self.is_euclidian:
            self.rotate(angles, scaling=scaling)
            first_operation = self.latest_operation.copy()
            self.translate(t, lambda_)
            self.latest_operation = self.latest_operation @ first_operation
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)
        return self.data

    def undo(self):
        if not self.is_euclidian:
            self.data = np.linalg.inv(self.latest_operation) @ self.data
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)
        return self.data

    def lies_on_line(self, line: ArrayLike, threshold: float = 1e-3):
        if not self.is_euclidian:
            return np.abs(self.data @ line) < threshold
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)

    def line_between(self, other: ArrayLike):
        if not self.is_euclidian:
```

```
            return np.cross(self.data.squeeze(), other.squeeze())
        else:
            raise ValueError(Points.NOT_HOMOGENEOUS)


def lines_intersect(line_1: ArrayLike, line_2: ArrayLike):
    non_euclidian_intersection = np.cross(line_1, line_2)
    intersection = Points(non_euclidian_intersection, is_euclidian=False)
    intersection.dehomogenize()
    return intersection.data.squeeze(), non_euclidian_intersection
```

In [ ]:

```
class Camera:
    def __init__(self, camera_matrix: ArrayLike):
        self.matrix = camera_matrix
        self.get_camera_center()
        self.get_viewing_direction()

    def get_camera_center(self):
        null_space = calculate_normalized_null_space(self.matrix)
        null_space = Points(null_space, is_euclidian=False)
        self.center = null_space.dehomogenize().squeeze()
        return self.center

    def get_viewing_direction(self):
        M = self.matrix[:, :-1]
        viewing_direction = np.linalg.det(M) * M[-1, :]
        self.viewing_direction = viewing_direction / np.linalg.norm(viewing_direction)
        return self.viewing_direction

    def project_cloud(self, X: Points, ax, size, homography=np.eye(3, 3), marker_size=2):
        height, width = size
        x = self.matrix @ X
        x = Points(x, is_euclidian=False).data
        x = homography @ x
        # x = np.where(x < 0, np.nan, x)
        # x = np.where(x[0] > (width - 1), np.nan, x)
        # x = np.where(x[1] > (height - 1), np.nan, x)
        ax.scatter(x[0], x[1], c="r", alpha=0.5, s=marker_size)
        return ax
```

```
    def plot_camera(
        self, ax=None, color="r", label="Viewing direction", length: ArrayLike = 1.0
    ):
        if ax is None:
            fig = plt.figure()
            ax = fig.add_subplot(projection="3d")
            ax.view_init(elev=30, azim=200, roll=15)

        arrow = self.center + length * self.viewing_direction * 0.5
        arrow = np.linspace(self.center, arrow, 400).T

        ax.plot(arrow[0], arrow[1], arrow[2], color=color, label=label)
        # ax.legend()
        return ax

    def RQ_decomposition(self):
        m, n = self.matrix.shape
        e = np.eye(m)
        p = e[:, ::-1]
        q_0, r_0 = np.linalg.qr(p @ self.matrix[:, :m].T @ p)

        r = p @ r_0.T @ p
        q = p @ q_0.T @ p

        fix = np.diag(np.sign(np.diag(r)))
        r = r @ fix
        q = fix @ q

        if n > m:
            q = np.concatenate((q, np.linalg.inv(r) @ self.matrix[:, m:n]), axis=1)
        r = r / r[-1, -1]
        return r, q
```

In [ ]:
```
def calculate_normalized_null_space(matrix: ArrayLike, is_euclidian: bool = False):
    null_space = np.linalg.svd(matrix)[2][-1]
    null = Points(null_space, is_euclidian=is_euclidian)
    return null.homogenize().squeeze()
```

In [ ]:
```
def plot_line(ax, size, line_parameters):
    """
```

```
    Plots a line given its parameters.

    Args:
        ax: Matplotlib axis to plot on.
        size: Tuple (height, width) specifying the plot size.
        line_parameters: Tuple (a, b, c) representing the line equation ax + by + c = 0.

    Returns:
        Updated Matplotlib axis with the plotted line.
    """
    a, b, c = line_parameters
    height, width = np.array(size) - 1

    x = np.linspace(0, width, 400)

    # Calculate corresponding y values
    y = (-a / b) * x - c / b

    # Remove entries where y is negative
    y = np.where(y < 0, np.nan, y)
    y = np.where(y > (height), np.nan, y)

    # Create the plot
    ax.plot(x, y, color="r", alpha=0.5)
    return ax


def plot_line_between_points(ax, size, points):
    p_1, p_2 = points.T
    p_1 = Points(p_1, is_euclidian=False)
    p_2 = Points(p_2, is_euclidian=False)
    line = p_1.line_between(p_2.data)
    plot_line(ax, size, line)
    return line
```

Computer Exercise 1

In the first 3D-plot it looks quite distorted the walls have different height, the angle between the two walls seem to wide, the depth is a little too shallow, etc.

Upon projecting the points they align really well with the photo. This was tried for many of the nine different photos, all working equally well.

Upon transforming the points using T_1 and 3D-plotting it made the distortions worse while transforming using T_2 corrected all distortions.

Projecting the transformed points using a transformed camera made no difference, the projected plots look the same. This is according to the theory of *Theoretical exercise 1*.

In [ ]:
```python
def mat_to_numpy(file_path: str) -> np.ndarray:
    # Load .mat file
    mat = loadmat(file_path)

    # Convert to numpy array
    arrays = {}
    for key in mat:
        if isinstance(mat[key], np.ndarray):
            arrays[key] = mat[key]
    return arrays
```

In [ ]:
```python
file_path = r"C:\Users\karllun\Desktop\Assignment 2\A2data\data\compEx1data.mat"
exercise_1 = mat_to_numpy(file_path)
x = exercise_1["x"].squeeze()
x = [points for points in x]
image_file_names = exercise_1["imfiles"].squeeze()
image_file_names = [file[0] for file in image_file_names]
P = exercise_1["P"].squeeze()
cameras = [Camera(camera) for camera in P]
reconstructed_3d_points = exercise_1["X"]
reconstructed_3d = Points(reconstructed_3d_points, is_euclidian=False)
wall_image_path = r"C:\Users\karllun\Desktop\Assignment 2\A2data\data\wall-photos"

images = [
    cv.imread(os.path.join(wall_image_path, filename))
    for filename in sorted(os.listdir(wall_image_path))
]
```

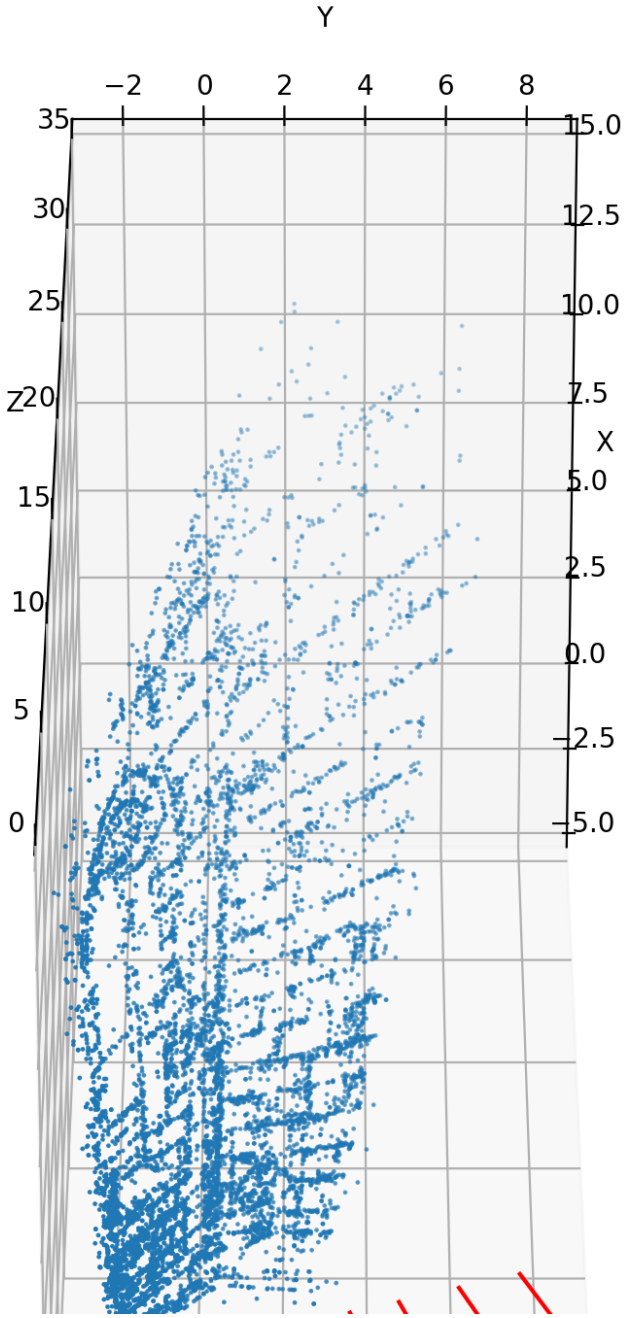
In [ ]:
```python
def plot_cameras(cameras, ax=None, length=5):
    if ax is None:
        fig = plt.figure(figsize=(12, 12))
        ax = fig.add_subplot(projection="3d")
```

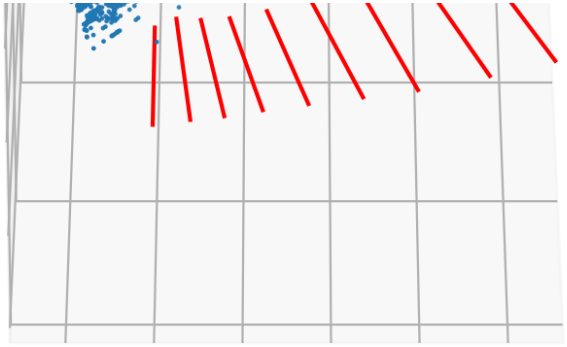
```
    for camera in cameras:
        ax = camera.plot_camera(ax=ax, length=length)
    return ax
```

In [ ]:
```
ax = plot_cameras(cameras, length=8)
reconstructed_3d.plot(ax=ax, marker_size=0.5)
ax.set_xlim(-5, 15)
ax.set_ylim(-3, 9)
ax.set_zlim(0, 35)
ax.set_aspect("equal")

ax.view_init(
    elev=-60, azim=0, roll=0)  # vertical_axis, azim, elev, roll. Easiest in that order.
ax.set_title("Untransformed 3D points and camera positions")
```

Out[ ]:
```
Text(0.5, 0.92, 'Untransformed 3D points and camera positions')
```

Untransformed 3D points and camera positions





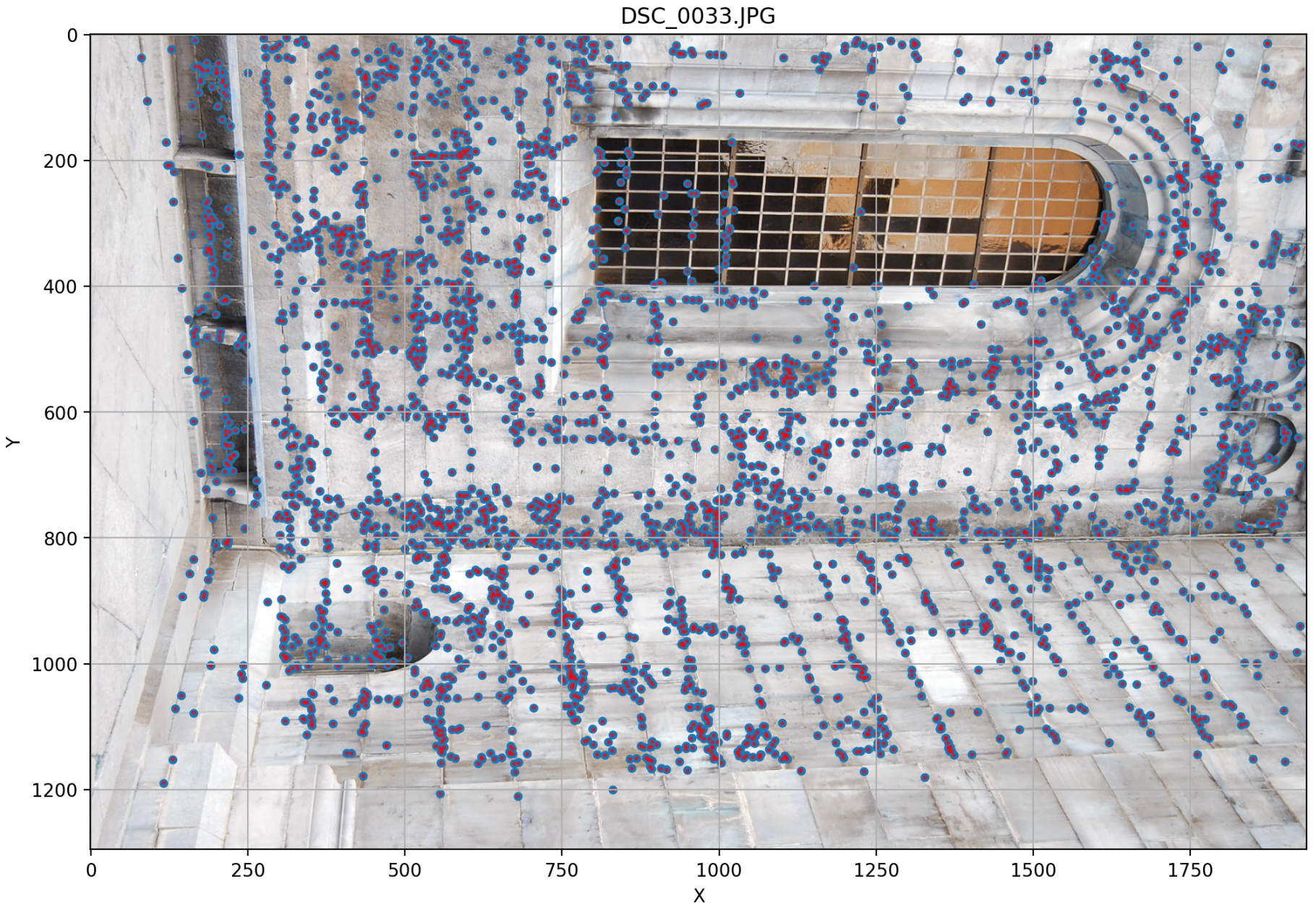
In [ ]:
```python
def plot_image(image_path: str, filename: str = None):
    if type(image_path) is str:
        image = cv.imread(image_path)
        image = cv.cvtColor(image, cv.COLOR_BGR2RGB)
        filename = os.path.basename(image_path)
    elif type(image_path) is np.ndarray and filename is not None:
        image = image_path
    else:
        raise ValueError("Image path needs to be a string or an already loaded image")
    size = image.shape[:-1]
    _, ax = plt.subplots(figsize=(12, 12))
    ax.imshow(image)
    ax.set_aspect("equal")
    ax.set_title(filename)
    return ax, size
```

In [ ]:
```python
def filter_nans(points_3d: ArrayLike, nan_filter: ArrayLike, image_number: int):
    nan_filter = nan_filter[image_number]
    mask = ~np.isnan(nan_filter)
    mask = mask[0]
    filtered_points_3d = points_3d[:, mask]
    return filtered_points_3d
```

In [ ]:
```python
def plot_projected_onto_image(points_3d, true_points, cameras, images, image_number, image_file_names):
    points_3d_filtered = filter_nans(points_3d, true_points, image_number)
```

```
    ax, size = plot_image(images[image_number], filename=image_file_names[image_number])
    true_points = x[image_number]
    true_points = Points(true_points, is_euclidian=False)
    true_points.plot(ax=ax, marker_size=15)

    camera = cameras[image_number]
    camera.project_cloud(points_3d_filtered, ax, size)
```

In [ ]:
```
photo_number = 8
plot_projected_onto_image(reconstructed_3d_points, x, cameras, images, photo_number, image_file_names)
```



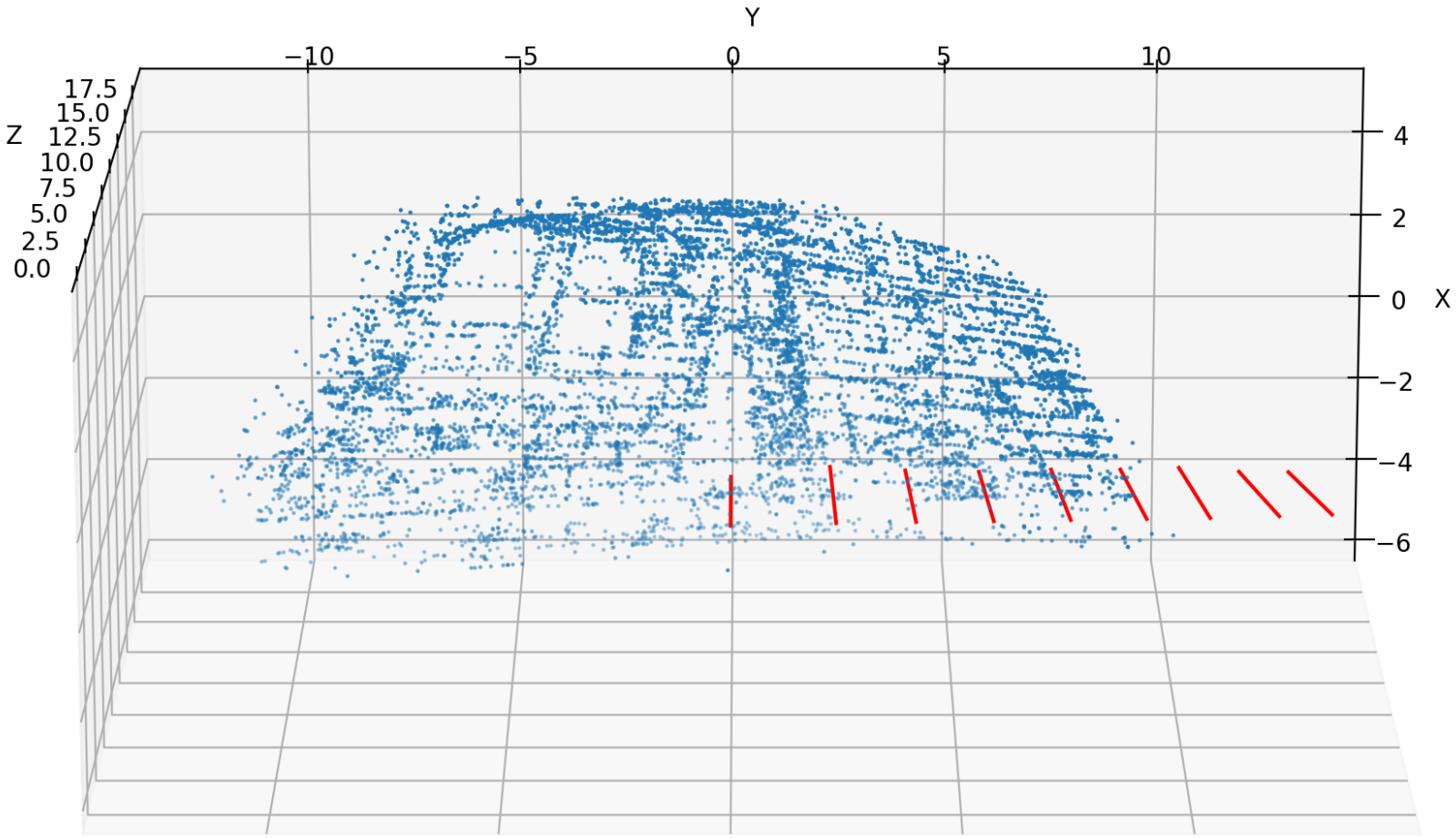
In [ ]:
```
row_1 = [1, 0, 0, 0]
row_2 = [0, 3, 0, 0]
```

```
row_3 = [0, 0, 1, 0]
row_4 = [1 / 8, 1 / 8, 0, 1]
transformation_1 = np.array([row_1, row_2, row_3, row_4])

row_1 = [1, 0, 0, 0]
row_2 = [0, 1, 0, 0]
row_3 = [0, 0, 1, 0]
row_4 = [1 / 16, 1 / 16, 0, 1]
transformation_2 = np.array([row_1, row_2, row_3, row_4])
```

In [ ]:
```
transformed_3d_1 = transformation_1 @ reconstructed_3d
transformed_3d_1 = Points(transformed_3d_1, is_euclidian=False)
transformed_cameras_1 = [Camera(camera @ np.linalg.inv(transformation_1)) for camera in P]

ax = plot_cameras(transformed_cameras_1, length=8)
transformed_3d_1.plot(ax=ax, marker_size=0.5)
# ax.set_xlim(-5, 15)
# ax.set_ylim(-3, 9)
# ax.set_zlim(0, 35)
ax.set_aspect("equal")

ax.view_init(
    elev=-75, azim=0, roll=0
)  # vertical_axis, azim, elev, roll. Easiest in that order.
ax.set_title("Transformation 1 3D points and camera positions")
```

Out[ ]:
```
Text(0.5, 0.92, 'Transformation 1 3D points and camera positions')
```

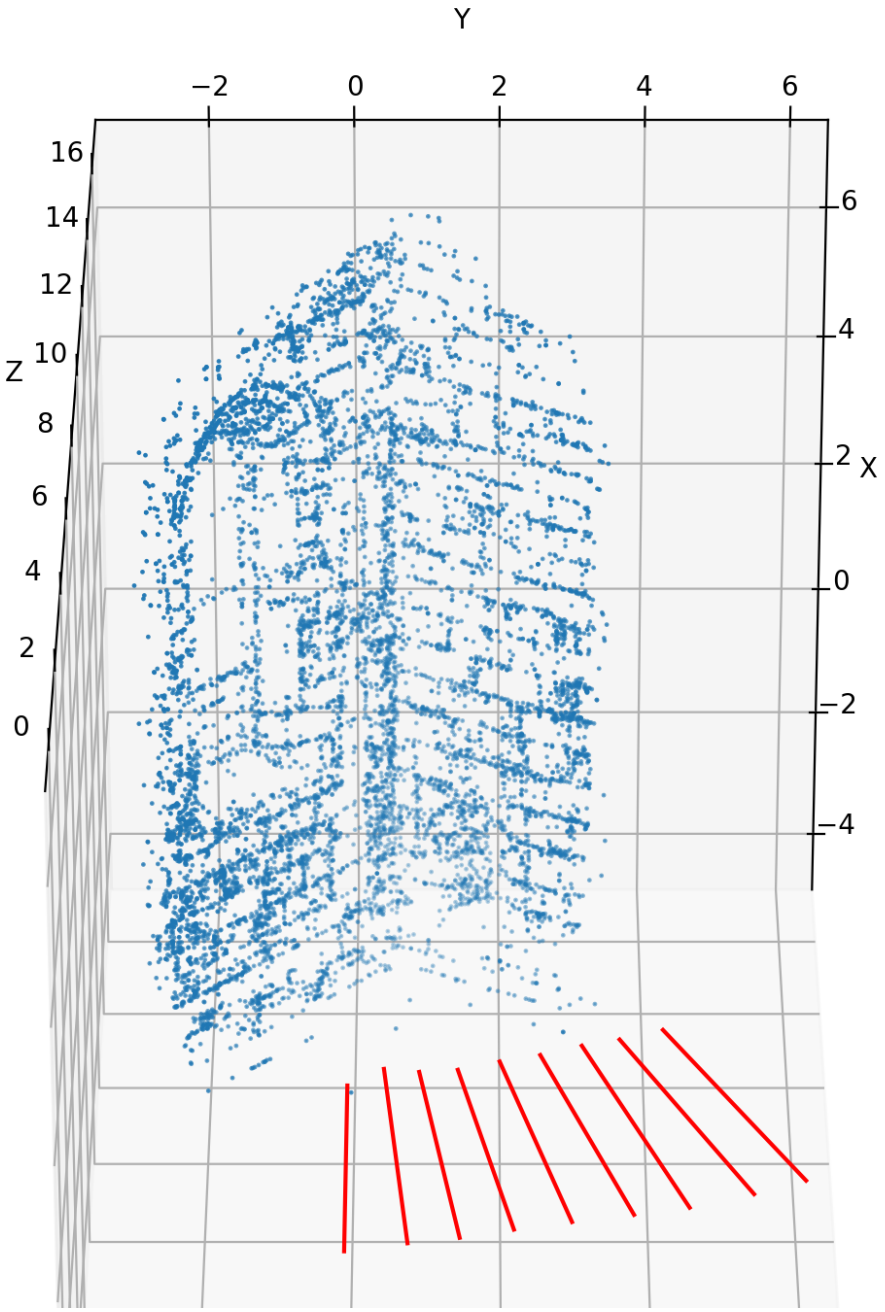
Transformation 1 3D points and camera positions

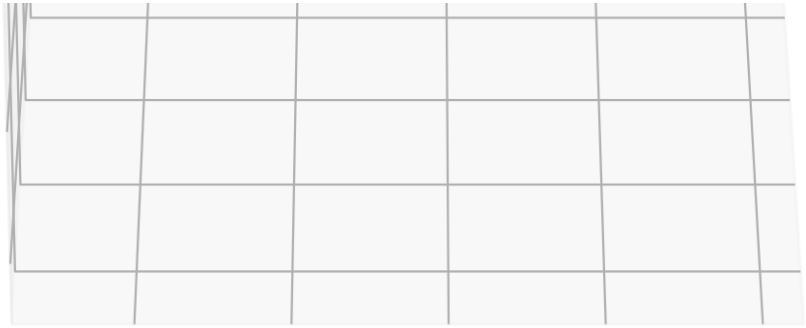


In [ ]:
```
transformed_cameras_2 = [Camera(camera @ np.linalg.inv(transformation_2)) for camera in P]
transformed_3d_2 = transformation_2 @ reconstructed_3d
transformed_3d_2 = Points(transformed_3d_2, is_euclidian=False)

ax = plot_cameras(transformed_cameras_2, length=8)
transformed_3d_2.plot(ax=ax, marker_size=0.5)
# ax.set_xlim(-5, 15)
# ax.set_ylim(-3, 9)
# ax.set_zlim(0, 35)
ax.set_aspect("equal")

ax.view_init(
    elev=-60, azim=0, roll=0
)  # vertical_axis, azim, elev, roll. Easiest in that order.
ax.set_title("Transformation 2 3D points and camera positions")
```

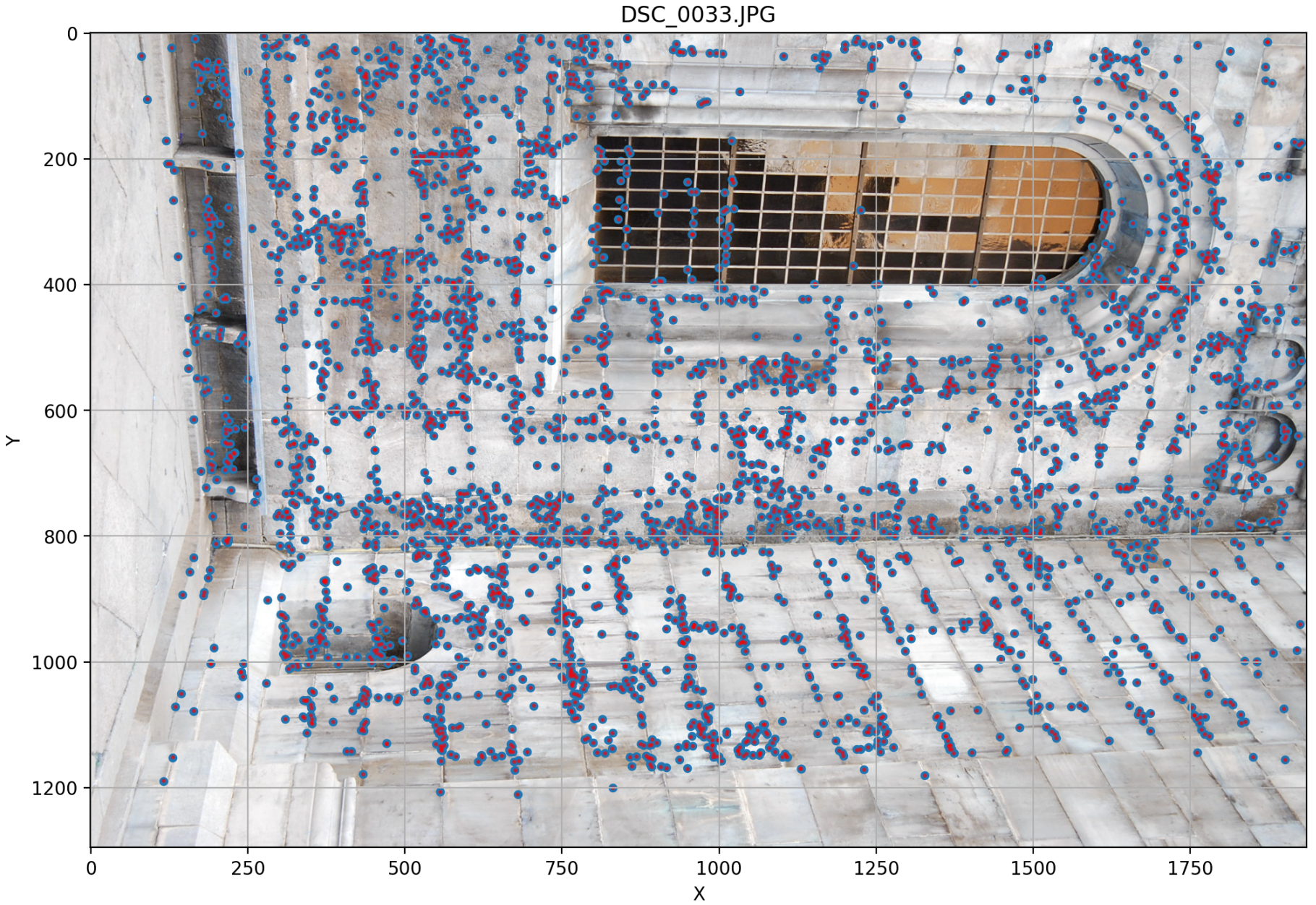
Out[ ]:
```
Text(0.5, 0.92, 'Transformation 2 3D points and camera positions')
```

Transformation 2 3D points and camera positions



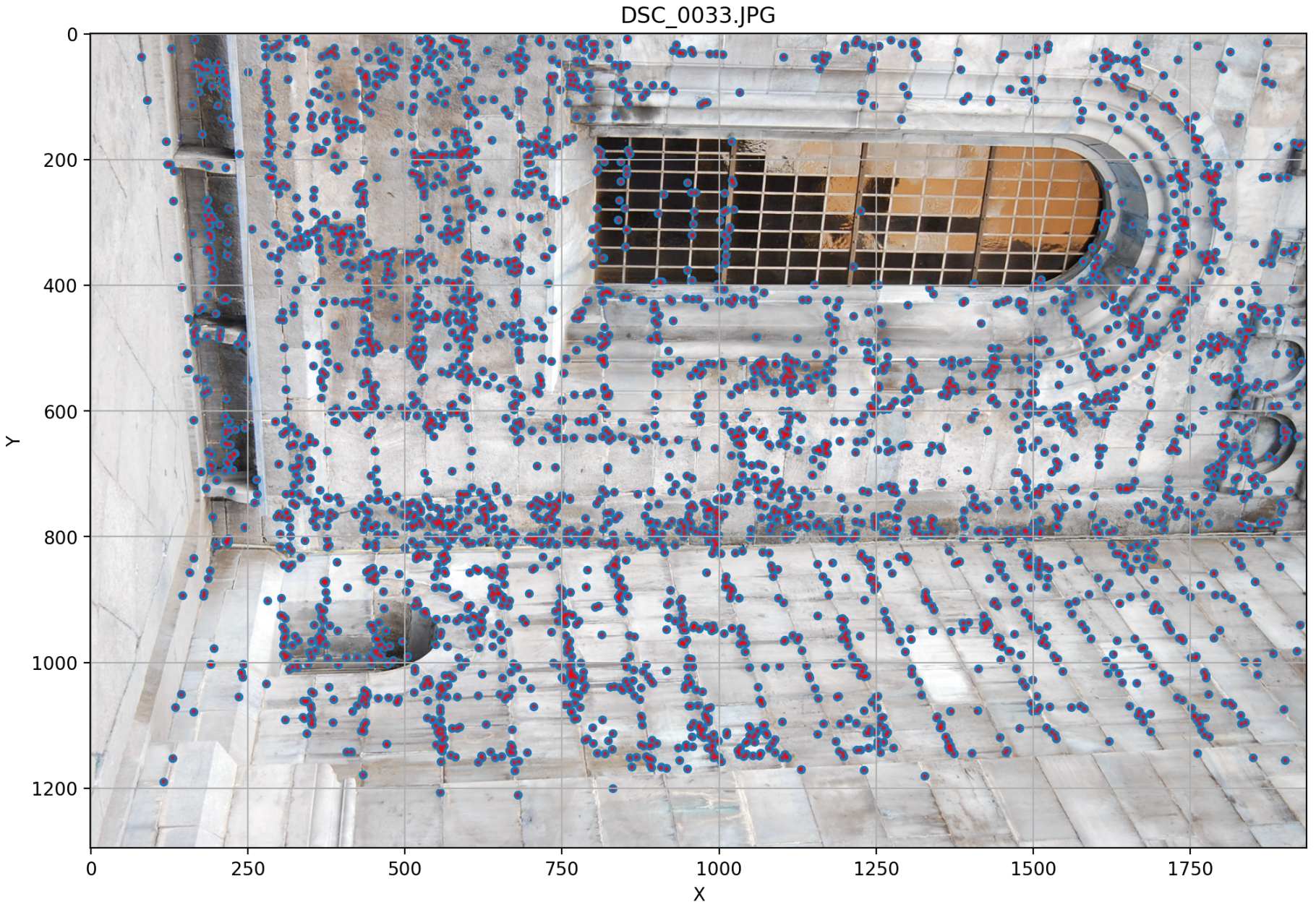


In [ ]:
```
photo_number = 8
plot_projected_onto_image(transformed_3d_1.data, x, transformed_cameras_1, images, photo_number, image_file_names)
```



In [ ]:
```
photo_number = 8
```

```
plot_projected_onto_image(transformed_3d_2.data, x, transformed_cameras_2, images, photo_number, image_file_names)
```



Theoretical exercise 2

When calibrated cameras are used we don't get projective distortions because the effect of K is reversed as $K^{-1}K$. However, the calibration process does not affect the projective ambiguity. It is still there. Looking at the projection equation from *Theoretical exercise 1* explains this.

$$P = K(R \mid t)X = K(R \mid t)T^{-1}TX$$
$$K^{-1}P = K^{-1}K(R \mid t)X = (R \mid t)T^{-1}TX$$

We can see that the projective ambiguity $T^{-1}T$ remains in the second equation.

3 - Camera Calibration

Theoretical exercise 3

For the calibration matrix $K = \begin{pmatrix} f & 0 & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{pmatrix}$ the inverse K^{-1} can be found by Gaussian elimination as

$$\left(\begin{array}{ccc|ccc} f & 0 & x_0 & 1 & 0 & 0 \\ 0 & f & y_0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{ccc|ccc} f & 0 & x_0 & 1 & 0 & 0 \\ 0 & f & 0 & 0 & 1 & -y_0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{ccc|ccc} f & 0 & 0 & 1 & 0 & -x_0 \\ 0 & f & 0 & 0 & 1 & -y_0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{ccc:ccc} f & 0 & 0 & 1 & 0 & -x_0 \\ 0 & 1 & 0 & 0 & \frac{1}{f} & \frac{-y_0}{f} \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{ccc:ccc} 1 & 0 & 0 & \frac{1}{f} & 0 & \frac{-x_0}{f} \\ 0 & 1 & 0 & 0 & \frac{1}{f} & \frac{-y_0}{f} \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

Here the right part of the above extended matrix is the inverse.

The factorization of K^{-1} is simply verified by multiplying the factorized matrices A and B together to again form K^{-1}. This is done and it holds. This is trivial, and I don't know how to explain it in more detail.

The geometric interpretation of A is a scaling matrix performing the operation of zooming by the focal length. and B is a translation matrix, aligning the image with the principal point.

When applying the transformation K^{-1} we are converting the image points from pixel coordinates to normalized image coordinates.

The principal point (x_0, y_0) ends up at the origin of the normalized image coordinate system.

A point with distance f to the principal point will end up at a distance of 1 from the principal point in the normalized image coordinate system.

We want to show that

$$\text{null}(K[R \mid t]) = \text{null}([R \mid t])$$

This means that

$$K[R \mid t]C = 0$$

$$K^{-1}K[R \mid t]C = K^{-1}0 = 0$$

$$I[R \mid t]C = [R \mid t]C = 0$$

Which is what should be shown.

Normalizing the points $(0, 300)$ and $(800, 300)$ is done by homogenizing the points and then multiplying by K^{-1} to get $(-1, 0)$ and $(1, 0)$.

The angle between these can be obtained merely by inspection, 180 degrees.

We want to show that the principal axis of $K(R \mid t)$ and $(R \mid t)$ is the same. A general calibration matrix can be written as $K = \begin{pmatrix} \gamma f & sf & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{pmatrix}$. Noting that the last row of K is $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$, this means that when multiplying $K(R \mid t)$ this essentially "picks-out" the last element of the columns of R, which define the principal axis. More formally

$$K(R \mid t) = \left(\begin{array}{ccc} \gamma f & sf & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{array}\right) \left(\begin{array}{ccc:c} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}\right) = \left(\begin{array}{ccc:c} & & & \\ & & & \\ i & j & k & \end{array}\right)$$

where the rest of the elements don't matter, because i, j, k completely define the principal axis.

In [ ]:
```
def angle_between_rays(point_1, point_2):


    angle_radians = np.arccos(point_1 @ point_2 / (np.linalg.norm(point_1) * np.linalg.norm(point_2)))
    return angle_radians * 180 / np.pi
```

In [ ]:
```
row_1 = [400, 0, 400]
row_2 = [0, 400, 300]
row_3 = [0, 0, 1]
calibration_matrix = np.array([row_1, row_2, row_3])
point_1 = [0, 300]
point_2 = [800, 300]

points = Points([point_1, point_2], is_euclidian=True)
normalized_points = np.linalg.inv(calibration_matrix) @ points
normalized_points = Points(normalized_points, is_euclidian=False)
ic(normalized_points.dehomogenize())
```

```
a, b = normalized_points.T
angle_between_rays(a, b)
```

```
ic| normalized_points.dehomogenize(): array([[-1.,  1.],
                                             [ 0.,  0.]])
```

Out[ ]: 180.0

4 - RQ Factorization and Computation of K

Theoretical exercise 4

By performing the multiplication KR the equality can easily be verified

$$\begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} R_1^\top \\ R_2^\top \\ R_3^\top \end{pmatrix} = \begin{pmatrix} aR_1^\top + bR_2^\top + cR_3^\top \\ dR_2^\top + eR_3^\top \\ fR_3^\top \end{pmatrix}$$

Let us call this matrix $KR = A$.

Knowing that $P = K[R \mid t] = [A \mid Kt]$, matching elements yields $fR_3 = A_3$. Also, recalling that $\|R_3\|$ = 1 means that all stretching is done by f, such that $f = \|A_3\| = 1$. Now from we can calculate R_3 as

$$fR_3 = \|A_3\|R_3 = A_3 \iff R_3 = \frac{A_3}{\|A_3\|} = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{pmatrix}^\top$$

Progressing to the next row of A and using that $R_1 \perp R_2 \perp R_3$ we can use that $e = R_3^\top A_2 = 1400$. Now the same procedure as for calculating R_3 can be performed, giving

$$dR_2 + eR_3 = A_2$$
$$dR_2 = A_2 - eR_3$$
$$R_2 = \frac{A_2 - eR_3}{d}$$
$$d = \|A_2 - eR_3\| = 2800$$
$$R_2 = \frac{A_2 - eR_3}{\|A_2 - eR_3\|} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^\top$$

Progressing to the final row of A and using the same machinery we compute $c = R_3^\top A_1 = 1600$ and $b = R_2^\top A_1 = 0$

$$aR_1 + bR_2 + cR_3 = A_1$$
$$aR_1 = A_1 - bR_2 - cR_3$$
$$R_1 = \frac{A_1 - bR_2 - cR_3}{a}$$
$$a = \|A_1 - bR_2 - cR_3\| = 3200$$
$$R_2 = \frac{A_1 - bR_2 - cR_3}{\|A_1 - bR_2 - cR_3\|} = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{pmatrix}^\top$$

Finally, to summarize,

$$R = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

and

$$K = \begin{pmatrix} \gamma f & sf & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 3200 & 0 & 1600 \\ 0 & 2800 & 1400 \\ 0 & 0 & 1 \end{pmatrix}$$

.

This corresponds to

- Focal length: $f = 2800$
- Skew: $s = 0$
- Aspect ratio: $\gamma = \frac{8}{7}$ from $\gamma f = 3200$
- Principal point: $(x_0, y_0) = (1600, 1400)$

Theoretical exercise 5

By using that fact that a norm always is a positive quantity, and, of course, its square too, this cost function can never be negative and, due to no constant terms, must have a minimum at zero. It can easily be shown that

$$\min_v \|Mv\|^2$$

has the minimum zero. For $\|Mv\|^2$ to equal zero we need $Mv = 0$. This is the null space equation which has the solutions $v = \text{null}(M)$ and $v = 0$, the zero vector. Using the fact that we can use the singular value decomposition such as $M = U\Sigma V^\top$ we can show that

$$\|Mv\|^2 = \|\Sigma V^\top v\|^2$$

as the following:

$$\|Mv\|^2 = (Mv)^\top (Mv) = v^\top M^\top Mv = v^\top (U\Sigma V^\top)^\top U\Sigma V^\top v = v^\top V\Sigma^\top U^\top U\Sigma V^\top v = \\ = v^\top V\Sigma^\top \Sigma V^\top v = (\Sigma V^\top v)^\top \Sigma V^\top v = \|\Sigma V^\top v\|^2$$

which is what should be shown. To show that

$$\|V^\top v\| = 1 \text{ if } \|v\|^2 = 1$$

we use that $\|v\|^2 = 1 \iff \|v\| = 1$, because norms are always positive.

$$\|V^\top v\| = \sqrt{(V^\top v)^\top V^\top v} = \sqrt{v^\top VV^\top v} = \sqrt{v^\top v} = \|v\| = 1$$

By using the result from above, that

$$\|Mv\|^2 = \|\Sigma V^\top v\|^2$$

and

$$\|V^\top v\| = \|v\| = 1 \text{ if } \|v\|^2 = 1$$

one can rewrite the optimization problem as

$$\min_{\|v\|^2=1} \|Mv\|^2 \equiv \min_{\|v\|^2=1} \|\Sigma V^\top v\|^2 \equiv \min_{\|v\|^2=1} \|\Sigma \tilde{v}\|^2 \equiv \min_{\|V^\top v\|^2=1} \|\Sigma \tilde{v}\|^2 \equiv \min_{\|\tilde{v}\|^2=1} \|\Sigma \tilde{v}\|^2$$

with $\tilde{v} = V^\top v$, which can be used to transform back and forth from the solutions. This problem has at least two solutions for the same reason as explained in the very top of this exercise, except the trivial zero vector solution gets removed by the norm-constraint, as explained in the exercise-text.

The solutions are $v = \text{null}(M)$ which can point in either the positive or negative direction. For the case with a non-square matrix, there are possibly many vectors that could span the null space of M. There are not an infinite number of solutions as the null space vector(s) can not be scaled by a constant because any solution needs to conform to the constraint $\|v\|^2 = 1$. For the case of $\text{rank}(M) < n$, this means that at least one singular value is zero, in fact, we have $n - \text{rank}(M)$ zeros. Given that the singular values decreases on the diagonal of Σ the last column of V corresponds to the smallest singular value, which will be zero in this case. Singular values explains the stretching of a vector when multiplied by its matrix. So, when M is multiplied by this last right-singular vector whose (for the case of $\text{rank}(M) < n$) corresponding singular value is zero, this is analogous to performing the multiplication of the null space equation $Mv_{\text{last}} = 0$, i.e., $\text{null}(M) = v_{\text{last}}$. This was earlier concluded to be the solution to the minimization problem in Equation (10). All columns of V satisfy the constraint of being of unit length, these matrices (U as well) are even orthogonal and unitary. This means all columns of V are potential solutions, because they satisfy the constraint. However, only $n - \text{rank}(M)$ solutions will be exactly equal to zero.

For the case of $m \geq n$ we don't have the guarantee of having $\text{rank}(M) < n$, however, v_{last} still corresponds to the smallest singular value, yielding the solution to the minimization problem, although not a minimum equal to zero. (Somehow the instructions state this is still possible...? However, this is not necessary to prove here).

For the case of $m < n$ we have $\text{rank}(M) < n$, guaranteeing a non-trivial vector existing in the null space of M.

In [ ]:
```
def calculate_normalized_null_space(matrix: ArrayLike, is_euclidian: bool = False):
    null_space = np.linalg.svd(matrix)[2][-1]
    null = Points(null_space, is_euclidian=is_euclidian)
    return null.homogenize().squeeze()
```

Theoretical exercise 6

$$P = N^{-1}\tilde{P}$$

5 - Direct Linear Transformation DLT

Computer exercise 2

We cannot know for sure that the parameters found from the DLT algorithm followed by RQ-factorization, as they are approximations from data. This can be seen from the figures when projecting the 3D-points onto the image points. They match pretty well, but does not line up exactly.

Regarding the projective amgibuity, we now, in a sense, use calibrated cameras because of the normalization N. Also N is an upper triangular, just like K, taking the form

$$\begin{pmatrix} \frac{1}{\sigma_x} & 0 & -\frac{\mu_x}{\sigma_x} \\ 0 & \frac{1}{\sigma_y} & -\frac{\mu_y}{\sigma_y} \\ 0 & 0 & 1 \end{pmatrix}$$

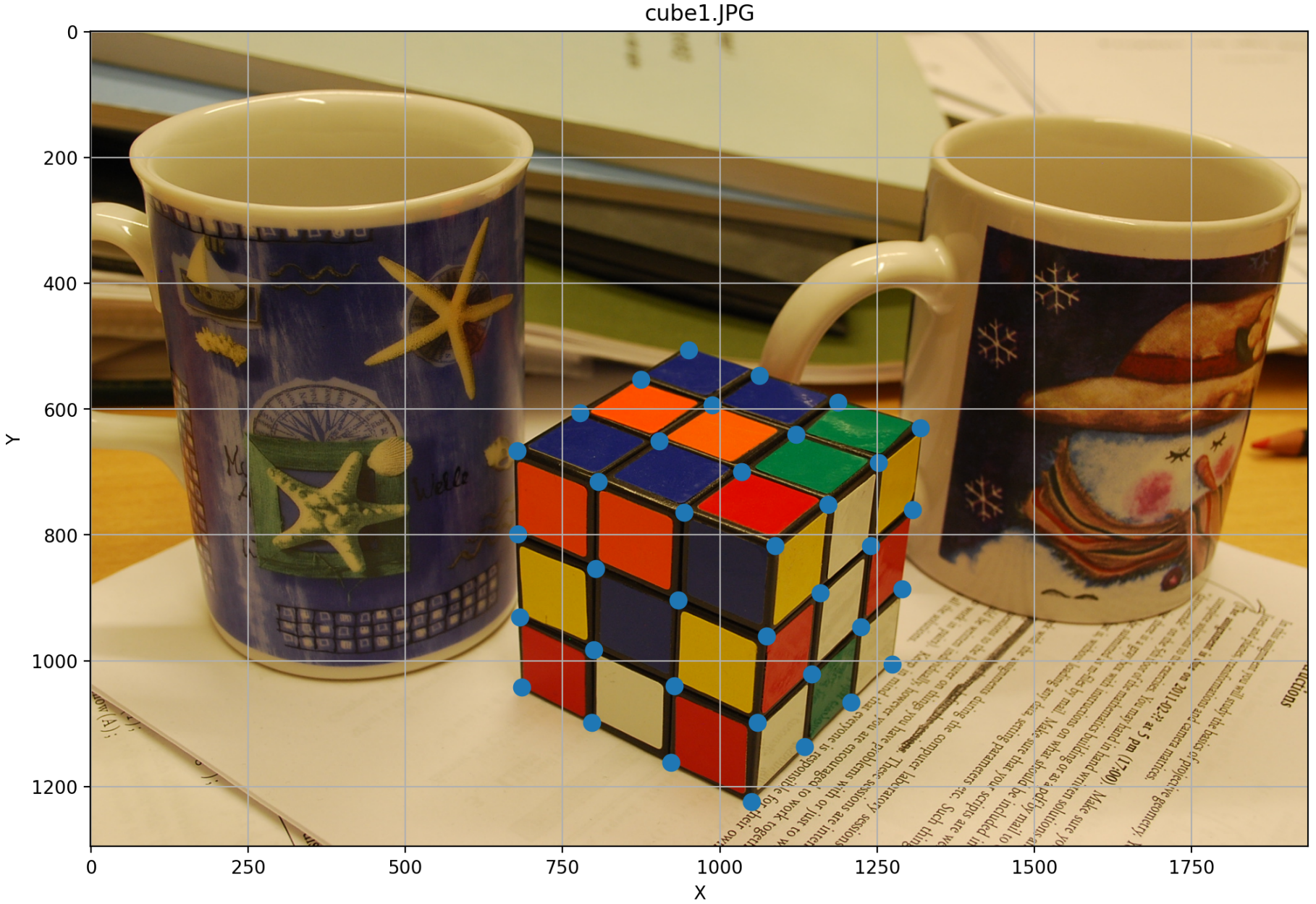
.

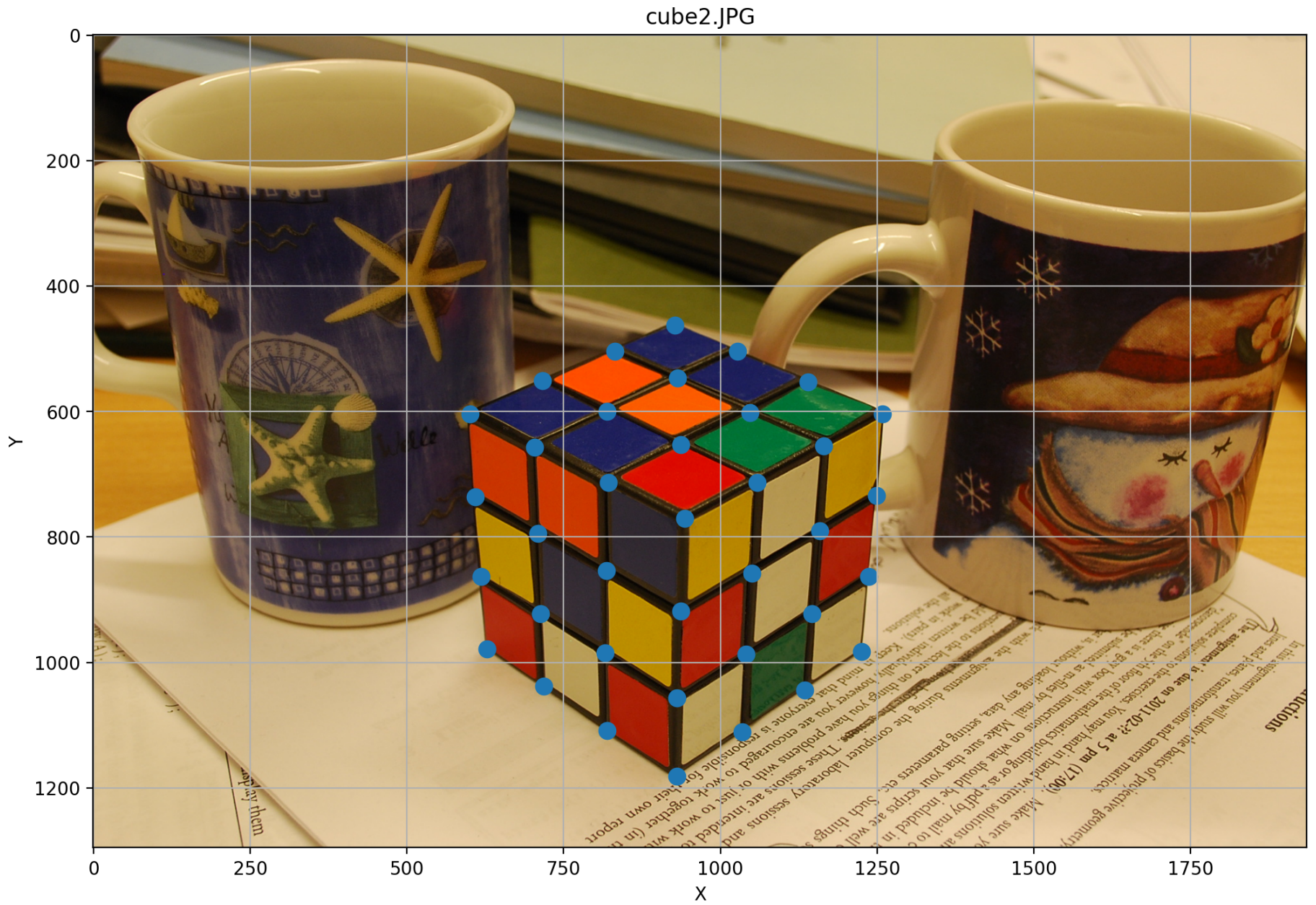
This too removes the effect the same way as knowing K would have done.

In [ ]:
```
compEx3data_path = r"C:\Users\karllun\Desktop\Assignment 2\A2data\data\compEx3data.mat"
compEx3data = mat_to_numpy(compEx3data_path)
compEx3data.keys()
x_1, x_2 = compEx3data["x"].squeeze()
X_model = compEx3data["Xmodel"]
startind = compEx3data["startind"].squeeze()
endind = compEx3data["endind"].squeeze()

cube_1_path = r"C:\Users\karllun\Desktop\Assignment 2\A2data\data\cube1.JPG"
cube_2_path = r"C:\Users\karllun\Desktop\Assignment 2\A2data\data\cube2.JPG"
ax_1, size_1 = plot_image(cube_1_path)
ax_2, size_2 = plot_image(cube_2_path)
x_1 = Points(x_1, is_euclidian=False)
```

```
x_2 = Points(x_2, is_euclidian=False)
x_1.plot(ax=ax_1, marker_size=80)
x_2.plot(ax=ax_2, marker_size=80)
```

Out[ ]: <Axes: title={'center': 'cube2.JPG'}, xlabel='X', ylabel='Y'>

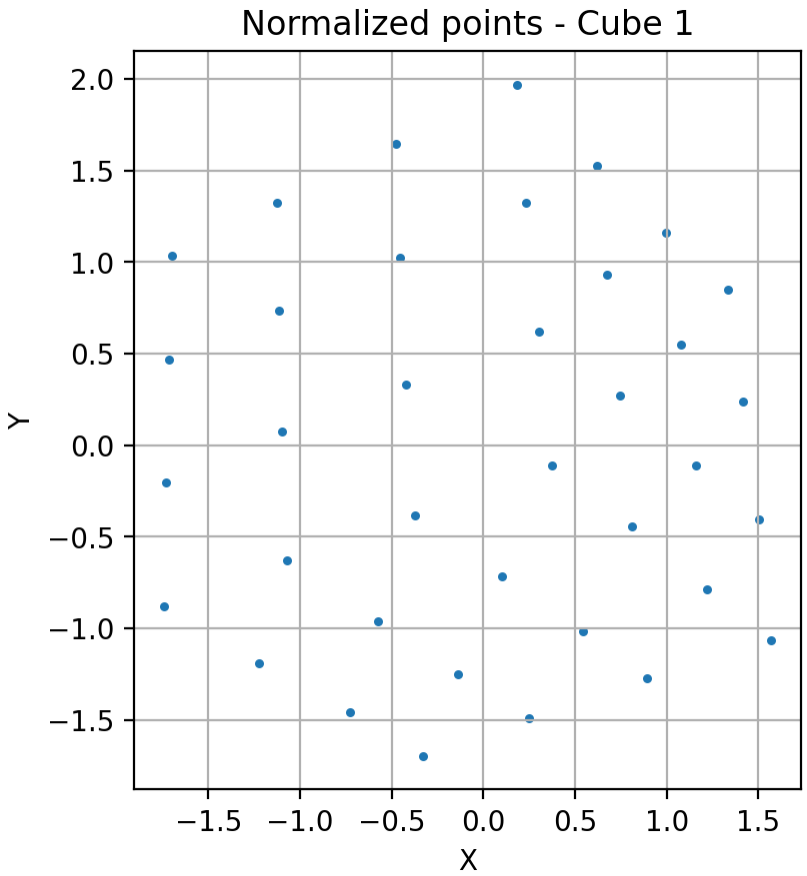

cube1.JPG
0
200
400
600
800
1000
1200
Y
0
250
500
750
1000
1250
1500
1750
X

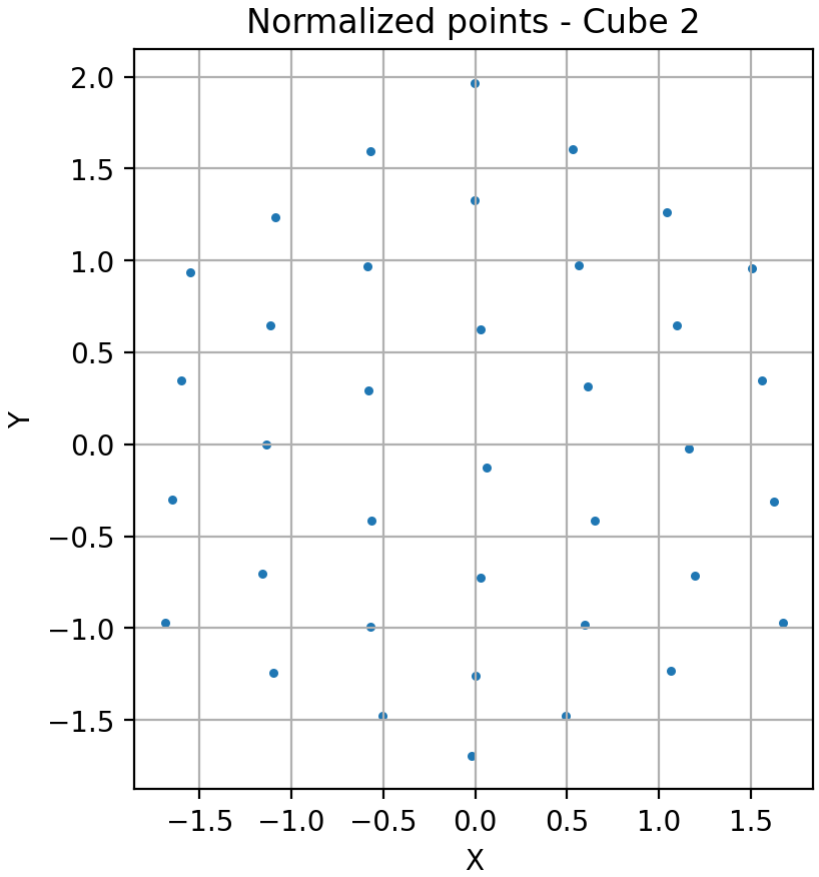


In [ ]:
```
X = Points(X_model, is_euclidian=True)
```

```
x_1.normalize()
ax_1 = x_1.plot()
ax_1.set_title("Normalized points - Cube 1")

x_2.normalize()
ax_2 = x_2.plot()
ax_2.set_title("Normalized points - Cube 2")
```

Out[ ]: Text(0.5, 1.0, 'Normalized points - Cube 2')





In [ ]:
```
ic(np.mean(x_1.data[:2]))
ic(np.mean(x_2.data[:2]))
ic(np.std(x_1.data[:2]))F
ic(np.std(x_2.data[:2]))
```

```
ic| np.mean(x_1.data[:2]): 3.1018731127196096e-16
ic| np.mean(x_2.data[:2]): 8.131633504686958e-16
ic| np.std(x_1.data[:2]): 1.0
ic| np.std(x_2.data[:2]): 0.9999999999999999
```

Out[ ]: 0.9999999999999999

In [ ]:
```python
def estimate_camera_DLT(xyz, uv, nd=3):
    xyz = xyz.T
    uv = uv.T
    n = xyz.shape[0]

    # Validating the parameters:
    if uv.shape[0] != n:
        raise ValueError(
            "Object (%d points) and image (%d points) have different number of points."
            % (n, uv.shape[0])
        )

    if xyz.shape[1] != 3:
        raise ValueError(
            "Incorrect number of coordinates (%d) for %dD DLT (it should be %d)."
            % (xyz.shape[1], nd, nd)
        )

    if n < 6:
        raise ValueError(
            "%dD DLT requires at least %d calibration points. Only %d points were entered."
            % (nd, 2 * nd, n)
        )

    A = []

    for i in range(n):
        x, y, z = xyz[i, 0], xyz[i, 1], xyz[i, 2]
        u, v = uv[i, 0], uv[i, 1]
        A.append([x, y, z, 1, 0, 0, 0, 0, -u * x, -u * y, -u * z, -u])
        A.append([0, 0, 0, 0, x, y, z, 1, -v * x, -v * y, -v * z, -v])

    # Convert A to array
    A = np.asarray(A)

    # Find the 11 parameters:
    _, S, V_T = np.linalg.svd(A)
    V = V_T.T
    V_last_column = V[:, -1]
    # The parameters are in the last column of V and normalize them
```

```
    P_vector = V_last_column / V[-1, -1]
    # Camera projection matrix
    P = P_vector.reshape(3, nd + 1)
    Mv = A @ V_last_column

    P_test = Camera(P)
    if P_test.get_viewing_direction()[-1] < 0:
        P = -P

    return P, S[-1], Mv
```

In [ ]:

```
def estimate_camera_DLT(Xmodel, img_pts):
    n = np.size(img_pts, 1)
    M = []

    for i in range(n):
        X = Xmodel[0, i]
        Y = Xmodel[1, i]
        Z = Xmodel[2, i]

        x = img_pts[0, i]
        y = img_pts[1, i]

        m = np.array(
            [
                [X, Y, Z, 1, 0, 0, 0, 0, -x * X, -x * Y, -x * Z, -x],
                [0, 0, 0, 0, X, Y, Z, 1, -y * X, -y * Y, -y * Z, -y],
            ]
        )

        M.append(m)

    M = np.concatenate(M, 0)
    ic(M.shape)
    U, S, VT = np.linalg.svd(M, full_matrices=False)

    P = np.stack([VT[-1, i : i + 4] for i in range(0, 12, 4)], 0)
    if P[2, 2] < 0:
        P = -P
```

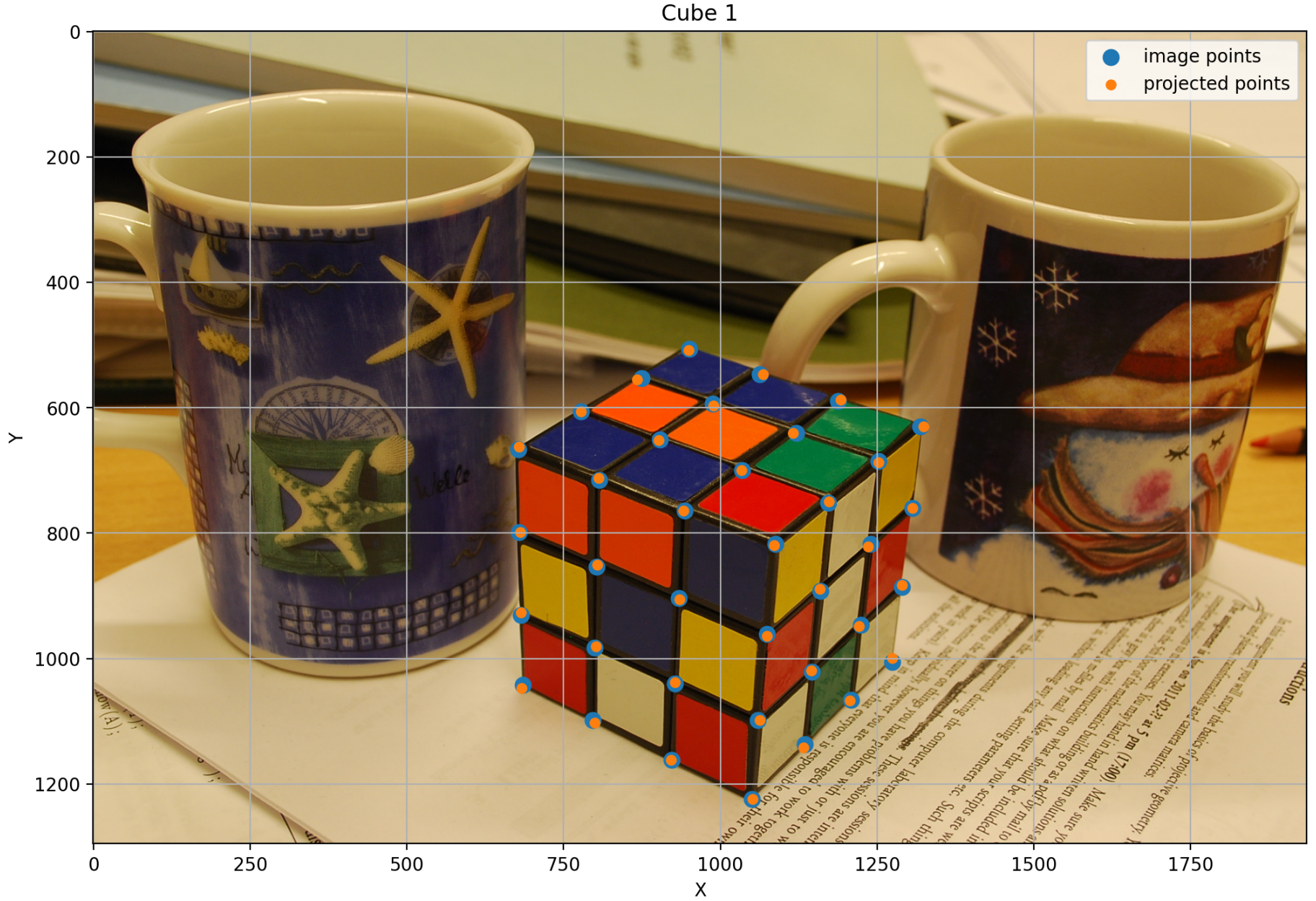
```
    M_approx = U @ np.diag(S) @ VT

    v = VT[-1, :]  # last row of VT because optimal v should be last column of V
    Mv = M @ v
    # print("||Mv||:", (Mv @ Mv) ** 0.5)
    ic(np.linalg.norm(Mv))
    # print(f"{(Mv @ Mv)**0.5=}")
    print("||v||^2:", v @ v)
    print("max{||M - M_approx||}:", np.max(np.abs(M - M_approx)))
    print("Smallest singular value:", S[-1])

    return P
```

In [ ]:

```
# X.normalize()
X.dehomogenize()
x_1.normalize()
x_1.dehomogenize()

# P_1, min_singular_value, Mv = estimate_camera_DLT(X.data, x_1.data)
P_1 = estimate_camera_DLT(X.data, x_1.data)
# ic(min_singular_value)
# ic(np.linalg.norm(Mv))

unnormalized_P_1 = np.linalg.inv(x_1.normalization_matrix) @ P_1
unnormalized_P_1 = Camera(unnormalized_P_1)

P_1 = Camera(P_1)

X.homogenize()
# P_1.project_cloud(X, ax_1, size_1)
projected_points = unnormalized_P_1.matrix @ X
projected_points = Points(projected_points, is_euclidian=False)

x_1.denormalize()
ax, size_1 = plot_image(cube_1_path)
ax = x_1.plot(ax=ax, marker_size=70)
ax = projected_points.plot(ax=ax, marker_size=25)
ax.legend(["image points", "projected points"])
ax.grid()
```

```
ax.set_title("Cube 1")
plt.show()
```

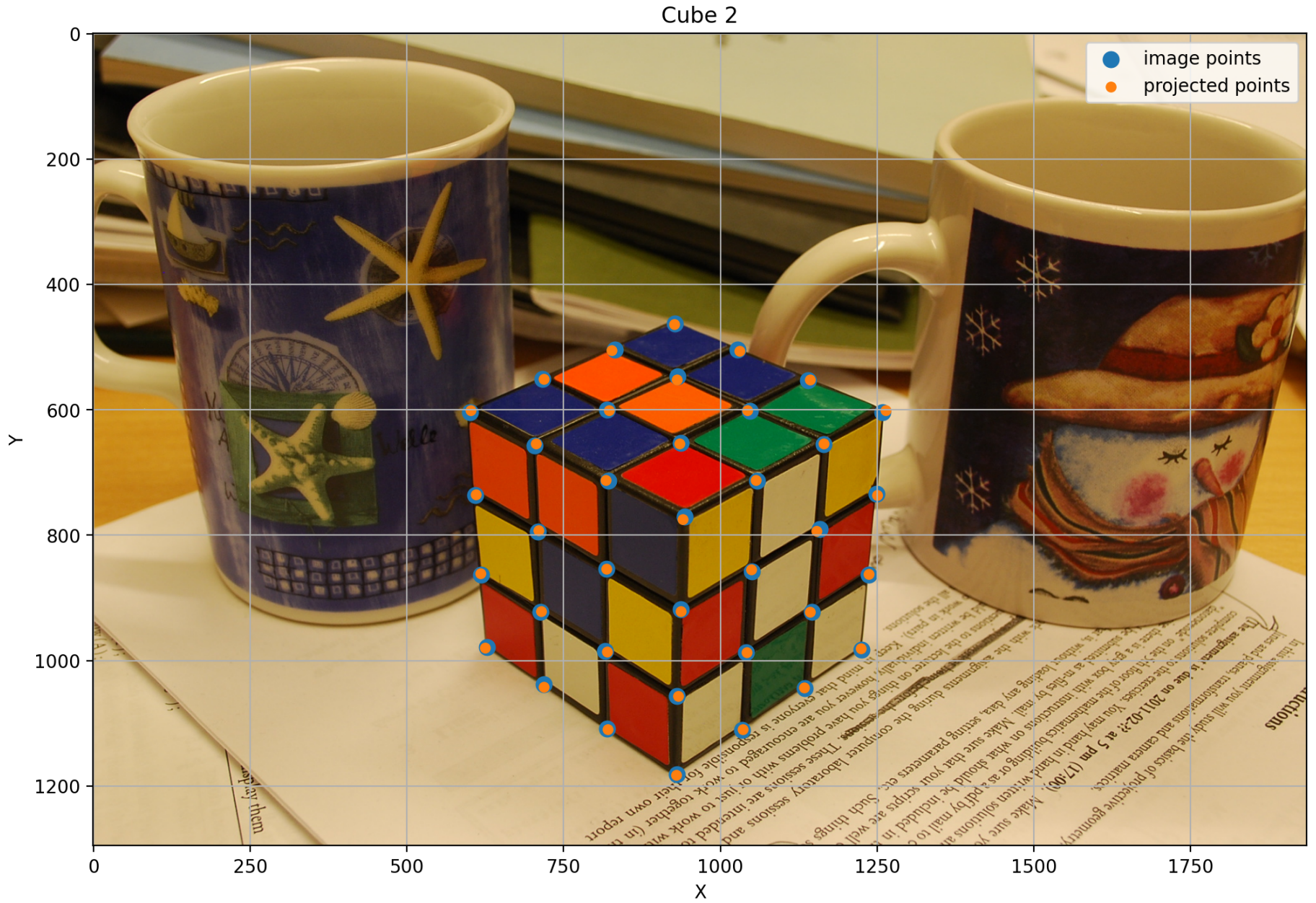
```
ic| M.shape: (74, 12)
ic| np.linalg.norm(Mv): 0.05103414113484469
Data is already normalized
||v||^2: 0.9999999999999993
max{||M - M_approx||}: 1.0658141036401503e-14
Smallest singular value: 0.051034141134845
```



In [ ]:
```
X.dehomogenize()
x_2.normalize()
```

```
x_2.dehomogenize()

# P_2, min_singular_value, Mv = estimate_camera_DLT(X.data, x_2.data)
P_2 = estimate_camera_DLT(X.data, x_2.data)
# ic(min_singular_value)
# ic(np.linalg.norm(Mv))

unnormalized_P_2 = np.linalg.inv(x_2.normalization_matrix) @ P_2
unnormalized_P_2 = Camera(unnormalized_P_2)

P_2 = Camera(P_2)
ic(P_2.get_camera_center())
ic(P_2.get_viewing_direction())
ic(P_2.matrix)

X.homogenize()
projected_points = unnormalized_P_2.matrix @ X
projected_points = Points(projected_points, is_euclidian=False)

x_2.denormalize()
ax, size_2 = plot_image(cube_2_path)
ax = x_2.plot(ax=ax, marker_size=70)
ax = projected_points.plot(ax=ax, marker_size=25)
ax.legend(["image points", "projected points"])
ax.grid()
ax.set_title("Cube 2")
plt.show()
```

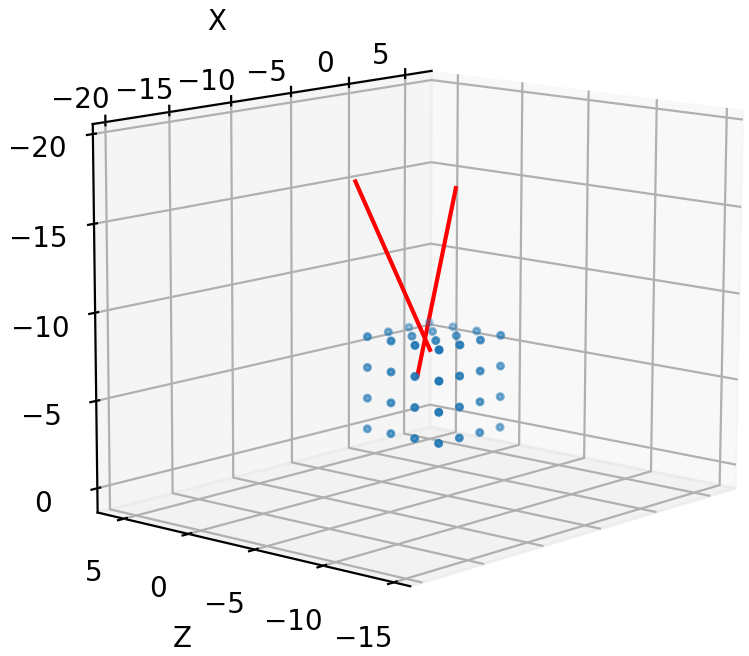
```
ic| M.shape: (74, 12)
ic| np.linalg.norm(Mv): 0.04378218304600183
ic| P_2.get_camera_center(): array([-15.75100944, -18.8353502 , -14.66359393])
ic| P_2.get_viewing_direction(): array([0.5978841 , 0.52250217, 0.60788658])
ic| P_2.matrix: array([[ 0.13122619, -0.00362511, -0.1364546 , -0.00224999],
                       [-0.0709404 ,  0.1604854 , -0.0713749 ,  0.85880317],
                       [ 0.00926213,  0.00809435,  0.00941708,  0.43643588]])
Data is already normalized
||v||^2: 0.9999999999999998
max{||M - M_approx||}: 1.7763568394002505e-14
Smallest singular value: 0.04378218304600207
```



In [ ]:
```
X.denormalize()
ax = X.plot()
```

```
ax = plot_cameras([unnormalized_P_1, unnormalized_P_2], ax=ax, length=60)
ax.view_init(
    elev=-40, azim=-165, roll=80
)  # vertical_axis, azim, elev, roll. Easiest in that order.
ax.set_aspect("equal")
```

```
Data is not normalized
```



In [ ]:
```
K_1, R_1 = unnormalized_P_1.RQ_decomposition()
K_2, R_2 = unnormalized_P_2.RQ_decomposition()
```

In [ ]:
```
def point_line_distance(point: ArrayLike, line_parameters: ArrayLike):
    point = Points(point, is_euclidian=True)
    point.homogenize()
    numerator = np.abs(point.data.squeeze() @ line_parameters)
```

```
    denominator = np.linalg.norm(line_parameters[:-1])
    return numerator / denominator
```

6 - Feature Extraction and Matching using SIFT

Computer exercise 3

In [ ]:
```
# cube_1_path = "/Users/karllundgrens/Nextcloud/Skola/Chalmers/Year 2/Period 2/Computer vision/Assignment 2/A2data/data/cube1.
# cube_2_path = "/Users/karllundgrens/Nextcloud/Skola/Chalmers/Year 2/Period 2/Computer vision/Assignment 2/A2data/data/cube2.
cube_1 = cv.imread(cube_1_path)
cube_2 = cv.imread(cube_2_path)
```

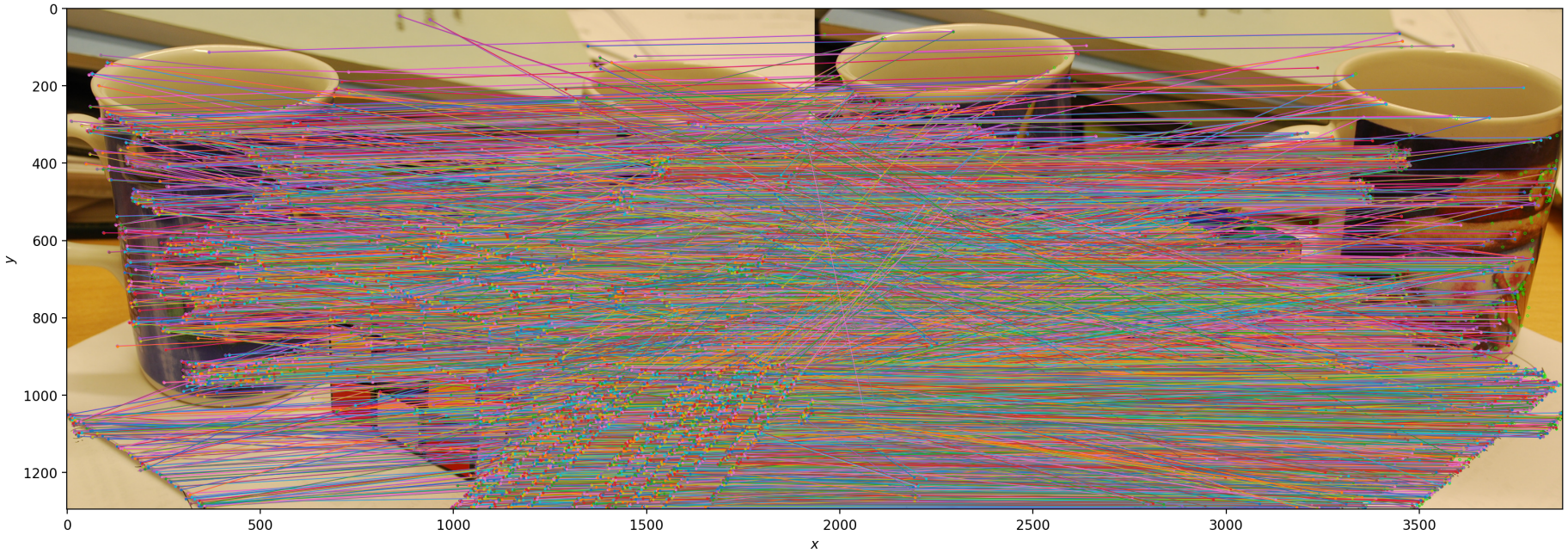
In [ ]:
```
def get_sift_plot_points(img1_pts, img2_pts, img1):
    x = [img1_pts[0, :], np.size(img1, 1) + img2_pts[0, :]]
    y = [img1_pts[1, :], img2_pts[1, :]]
    return x, y
```

In [ ]:
```
def compute_sift_points(img1, img2, marg):
    sift = cv.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)  # or pass empty dictionary

    flann = cv.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)

    good_matches = []
    for m, n in matches:
        if m.distance < marg * n.distance:
            good_matches.append([m])

    draw_params = dict(
        matchColor=(255, 0, 255),
        singlePointColor=(0, 255, 0),
```

```
        matchesMask=None,
        flags=cv.DrawMatchesFlags_DEFAULT,
    )
    img_match = cv.drawMatchesKnn(img1, kp1, img2, kp2, good_matches, None, **draw_params)

    img1_pts = np.stack([kp1[match[0].queryIdx].pt for match in good_matches], 1)
    img2_pts = np.stack([kp2[match[0].trainIdx].pt for match in good_matches], 1)

    print("Number of good matches:", np.size(img1_pts, 1))

    return img1_pts, img2_pts, img_match
```

In [ ]:
```
cube_1_points, cube_2_points, cube_match = compute_sift_points(cube_1, cube_2, marg=2)

x, y = get_sift_plot_points(cube_1_points, cube_2_points, cube_1)

cube_1_points = Points(cube_1_points, is_euclidian=True)
cube_2_points = Points(cube_2_points, is_euclidian=True)

fig, ax = plt.subplots(figsize=(20, 9))
ax.plot(x, y, "o-", ms=1, lw=0.5)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_aspect("equal")
ax.imshow(cv.cvtColor(cube_match, cv.COLOR_BGR2RGB))
plt.show()
```

```
Number of good matches: 6413
```



7 - Triangulation using DLT

Computer exercise 4

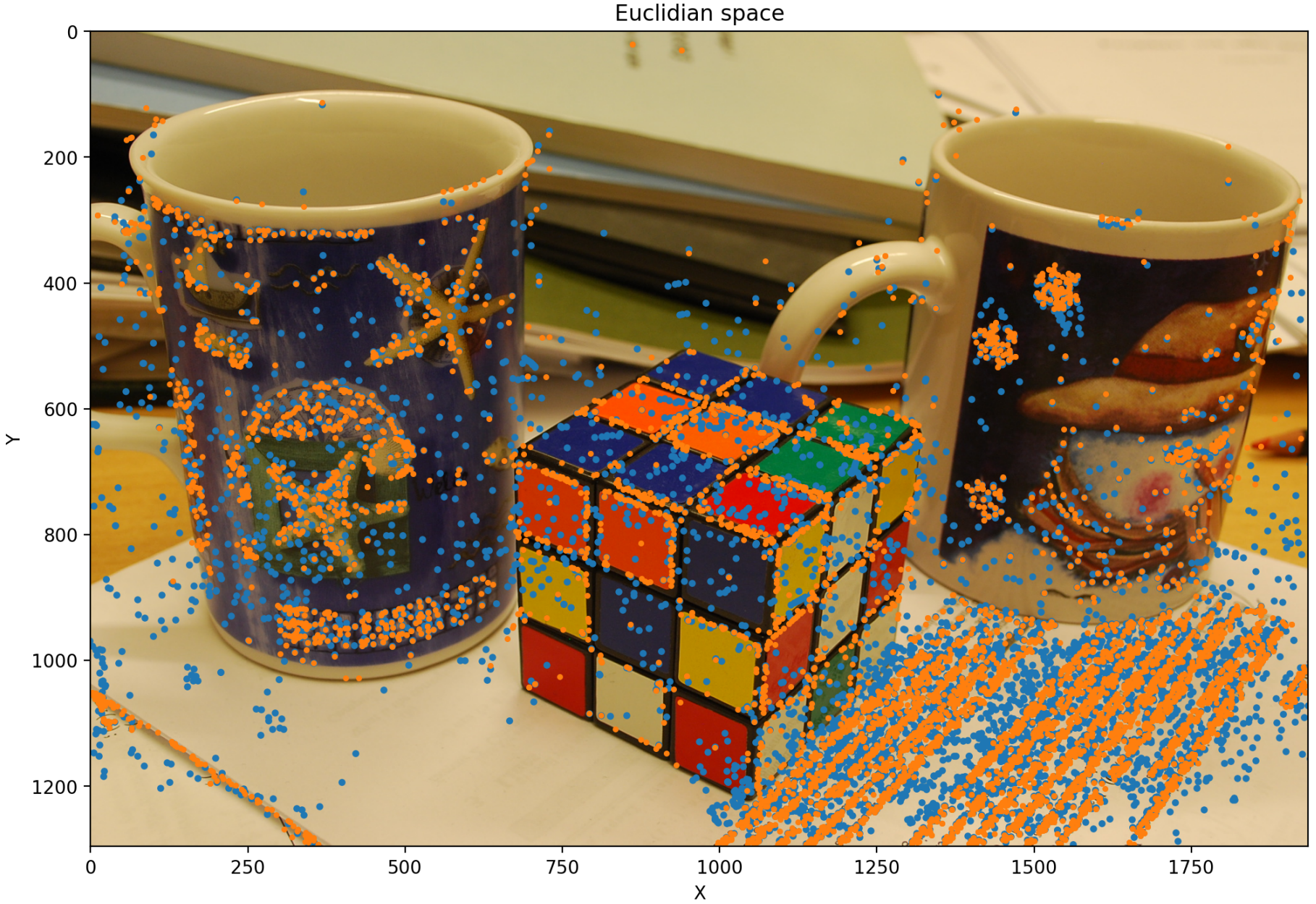
From the 3D plot found at the bottom, it is possible to distinguish the cups and paper and also somewhat of the cube.

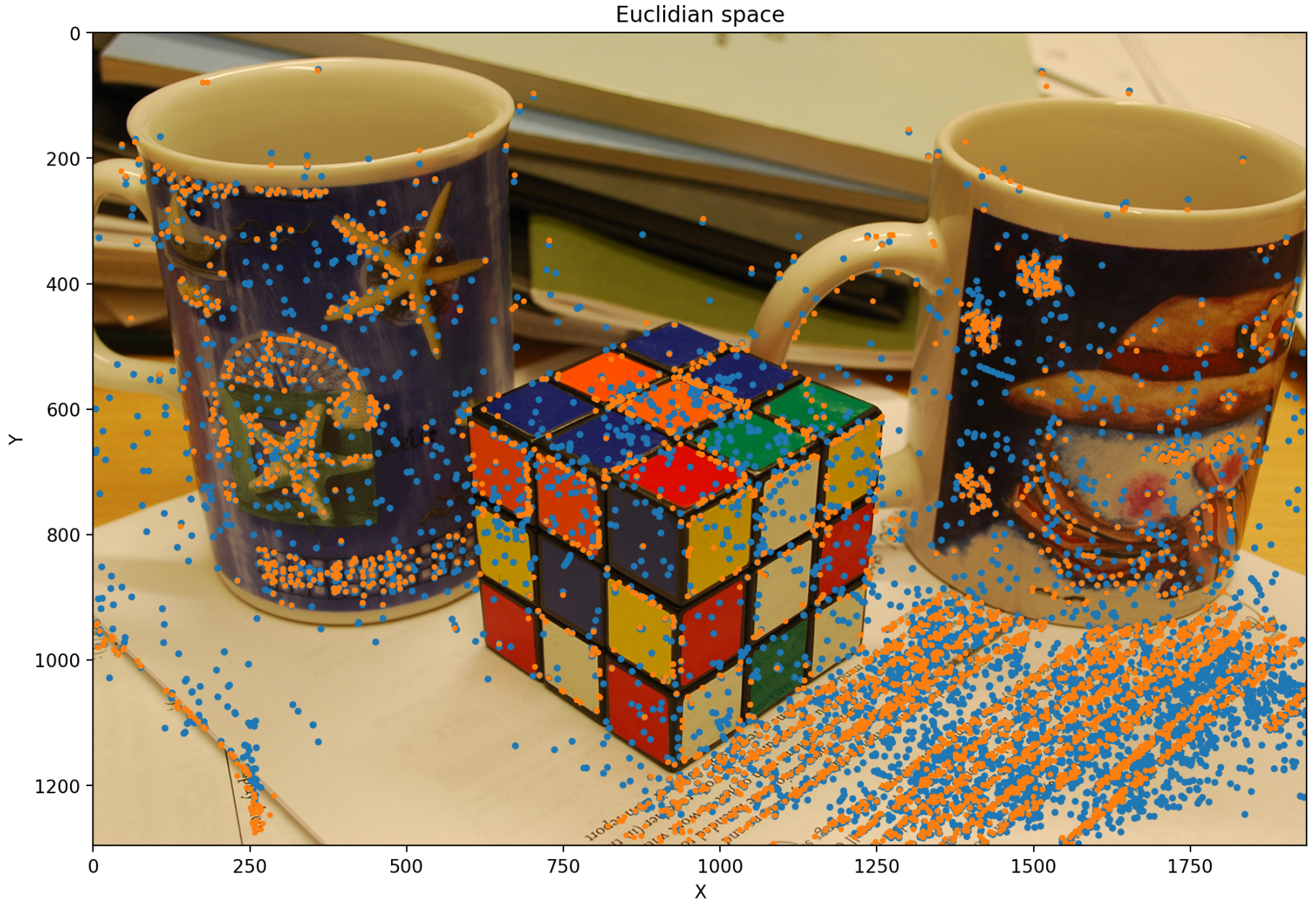
In [ ]:
```
def triangulate_3D_point_dlt(camera_matrix_1, camera_matrix_2, xy_1: Points, xy_2: Points):
    xs_1, ys_1 = xy_1.dehomogenize()
    xs_2, ys_2 = xy_2.dehomogenize()
    n = len(xs_1)

    Vs = []
    for i in range(n):
        A = []
        x_1, y_1 = xs_1[i], ys_1[i]
        x_2, y_2 = xs_2[i], ys_2[i]
        A.append(
            [
```

```
                camera_matrix_1[0, 0] - x_1 * camera_matrix_1[2, 0],
                camera_matrix_1[0, 1] - x_1 * camera_matrix_1[2, 1],
                camera_matrix_1[0, 2] - x_1 * camera_matrix_1[2, 2],
                camera_matrix_1[0, 3] - x_1 * camera_matrix_1[2, 3],
            ]
        )
        A.append(
            [
                camera_matrix_1[1, 0] - y_1 * camera_matrix_1[2, 0],
                camera_matrix_1[1, 1] - y_1 * camera_matrix_1[2, 1],
                camera_matrix_1[1, 2] - y_1 * camera_matrix_1[2, 2],
                camera_matrix_1[1, 3] - y_1 * camera_matrix_1[2, 3],
            ]
        )
        A.append(
            [
                camera_matrix_2[0, 0] - x_2 * camera_matrix_2[2, 0],
                camera_matrix_2[0, 1] - x_2 * camera_matrix_2[2, 1],
                camera_matrix_2[0, 2] - x_2 * camera_matrix_2[2, 2],
                camera_matrix_2[0, 3] - x_2 * camera_matrix_2[2, 3],
            ]
        )
        A.append(
            [
                camera_matrix_2[1, 0] - y_2 * camera_matrix_2[2, 0],
                camera_matrix_2[1, 1] - y_2 * camera_matrix_2[2, 1],
                camera_matrix_2[1, 2] - y_2 * camera_matrix_2[2, 2],
                camera_matrix_2[1, 3] - y_2 * camera_matrix_2[2, 3],
            ]
        )
        A = np.asarray(A)

        _, _, V_T = np.linalg.svd(A)
        V = V_T.T
        V_last_column = V[:, -1]
        Vs.append(V_last_column)
    Vs = np.asarray(Vs).T
    return Vs
```

In [ ]:
```
Vs = triangulate_3D_point_dlt(unnormalized_P_1.matrix, unnormalized_P_2.matrix, cube_1_points, cube_2_points)

Vs = Points(Vs, is_euclidian=False)
x_1_projected = unnormalized_P_1.matrix @ Vs.data
x_2_projected = unnormalized_P_2.matrix @ Vs.data

ax_1, size_1 = plot_image(cube_1_path)
ax_2, size_2 = plot_image(cube_2_path)
x_1_projected = Points(x_1_projected, is_euclidian=False)
x_2_projected = Points(x_2_projected, is_euclidian=False)
x_1_projected.plot(ax=ax_1, marker_size=8)
x_2_projected.plot(ax=ax_2, marker_size=8)
ax_1.set_xlim(0, size_1[1])
ax_1.set_ylim(size_1[0], 0)
ax_2.set_xlim(0, size_2[1])
ax_2.set_ylim(size_2[0], 0)

cube_1_points.plot(marker_size=5, ax=ax_1)
cube_2_points.plot(marker_size=5, ax=ax_2)
```

Out[ ]:
```
<Axes: title={'center': 'Euclidian space'}, xlabel='X', ylabel='Y'>
```


Euclidian space
0
200
400
600
800
1000
1200
Y
0
250
500
750
1000
1250
1500
1750
X



In [ ]:
```
def remove_error_points(x1_proj, x1_img, x2_proj, x2_img, X, tol):
    x1_keep = (
```

```
        (x1_proj[0, :] - x1_img[0, :]) ** 2 + (x1_proj[1, :] - x1_img[1, :]) ** 2
    ) ** 0.5 < tol
    x2_keep = (
        (x2_proj[0, :] - x2_img[0, :]) ** 2 + (x2_proj[1, :] - x2_img[1, :]) ** 2
    ) ** 0.5 < tol
    x_keep = x1_keep * x2_keep

    x1_proj_keep = x1_proj[:, x_keep]
    x1_img_keep = x1_img[:, x_keep]

    x2_proj_keep = x2_proj[:, x_keep]
    x2_img_keep = x2_img[:, x_keep]

    X_keep = X[:, x_keep]

    return x1_proj_keep, x1_img_keep, x2_proj_keep, x2_img_keep, X_keep
```

In [ ]:

```
def remove_error_points(x1_proj, x1_img, x2_proj, x2_img, X, tol):
    x1_keep = (
        (x1_proj[0, :] - x1_img[0, :]) ** 2 + (x1_proj[1, :] - x1_img[1, :]) ** 2
    ) ** 0.5 < tol
    x2_keep = (
        (x2_proj[0, :] - x2_img[0, :]) ** 2 + (x2_proj[1, :] - x2_img[1, :]) ** 2
    ) ** 0.5 < tol
    x_keep = x1_keep * x2_keep

    x1_proj_keep = x1_proj[:, x_keep]
    x1_img_keep = x1_img[:, x_keep]

    x2_proj_keep = x2_proj[:, x_keep]
    x2_img_keep = x2_img[:, x_keep]

    X_keep = X[:, x_keep]

    return x1_proj_keep, x1_img_keep, x2_proj_keep, x2_img_keep, X_keep
```

In [ ]:

```
tol = 3
x_1_projected_filt, cube_1_points_filt, x_2_projected_filt, cube_2_points_filt, X_filt = remove_error_points(
    x_1_projected.data, cube_1_points.data, x_2_projected.data, cube_2_points.data, Vs.data, tol
)
```

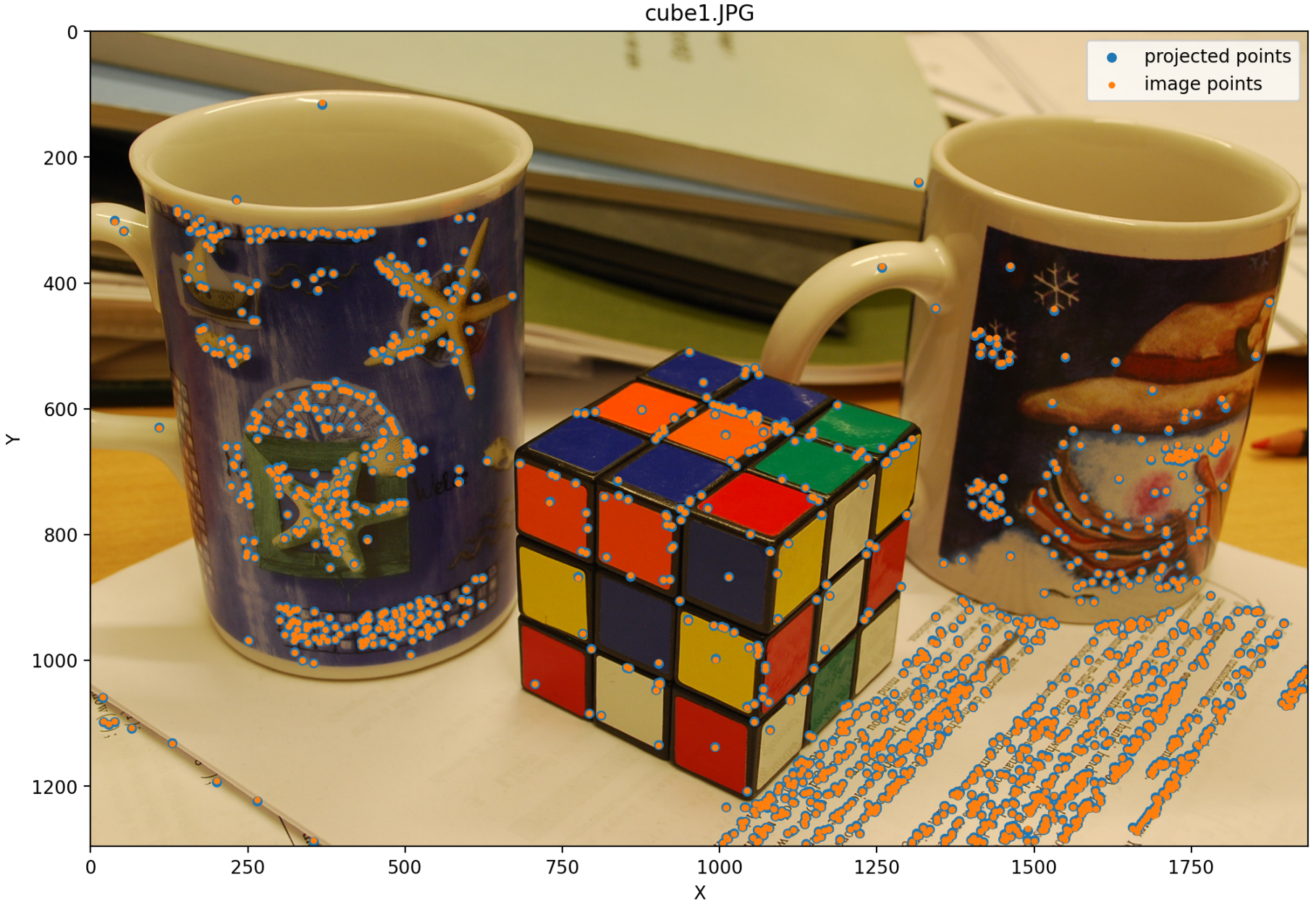
```
ic(x_1_projected_filt.shape)
ic(cube_1_points_filt.shape)
ic(X_filt.shape)
```

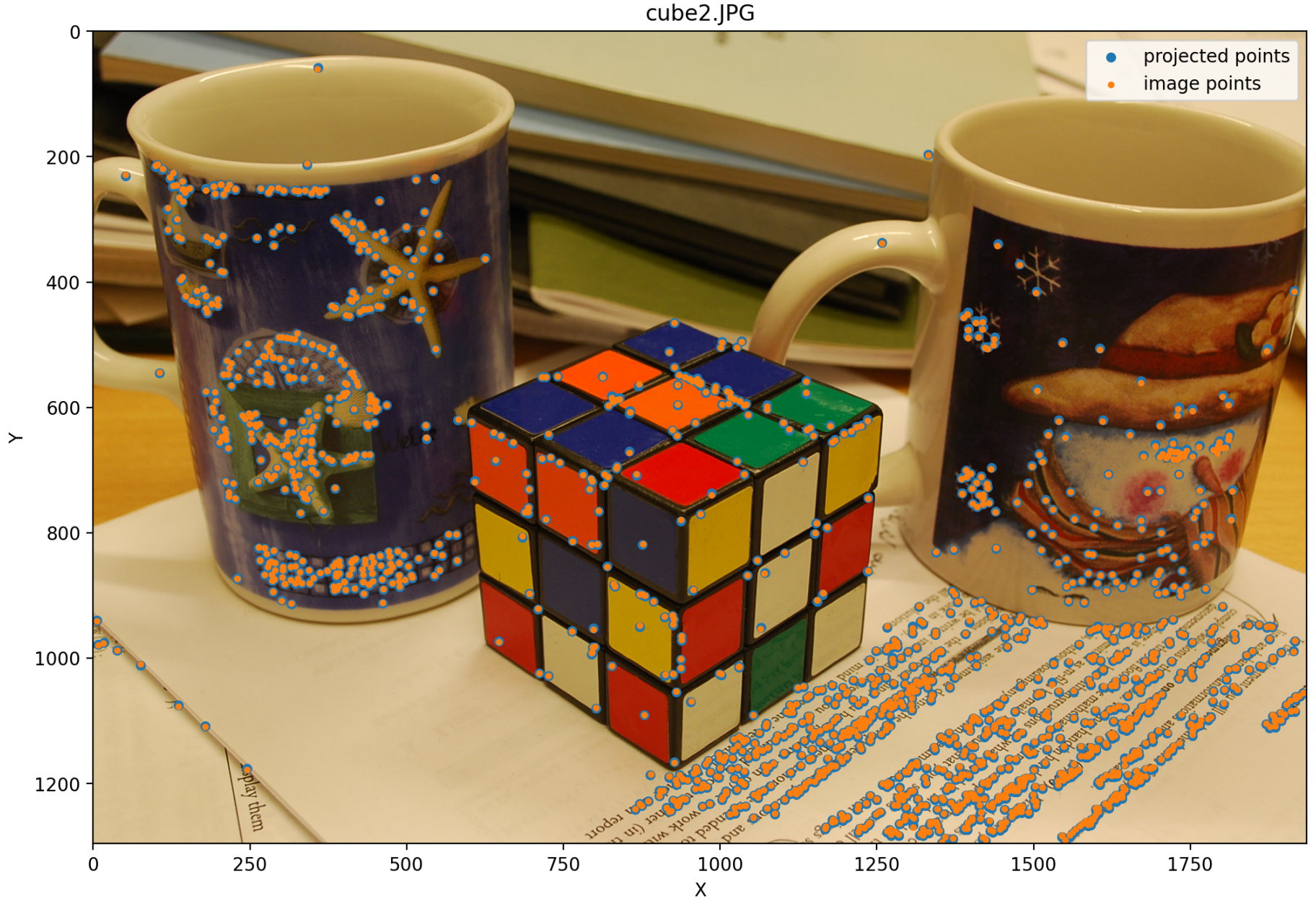
```
ic| x_1_projected_filt.shape: (3, 1920)
ic| cube_1_points_filt.shape: (2, 1920)
ic| X_filt.shape: (4, 1920)
```

Out[ ]: (4, 1920)

In [ ]:
```
ax_1, size_1 = plot_image(cube_1_path)
ax_2, size_2 = plot_image(cube_2_path)
x_1_projected_filt = Points(x_1_projected_filt, is_euclidian=False)
x_2_projected_filt = Points(x_2_projected_filt, is_euclidian=False)
x_1_projected_filt.plot(ax=ax_1, marker_size=20)
x_2_projected_filt.plot(ax=ax_2, marker_size=20)
ax_1.set_xlim(0, size_1[1])
ax_1.set_ylim(size_1[0], 0)
ax_2.set_xlim(0, size_2[1])
ax_2.set_ylim(size_2[0], 0)


cube_1_points_filt = Points(cube_1_points_filt, is_euclidian=True)
cube_2_points_filt = Points(cube_2_points_filt, is_euclidian=True)
cube_1_points_filt.plot(marker_size=7, ax=ax_1)
cube_2_points_filt.plot(marker_size=7, ax=ax_2)
ax_1.legend(["projected points", "image points"])
ax_2.legend(["projected points", "image points"])
```

Out[ ]: <matplotlib.legend.Legend at 0x18b19e7fd00>


cube1.JPG
projected points
image points
0
200
400
600
800
1000
1200
Y
0
250
500
750
1000
1250
1500
1750
X



In [ ]:
```
normalized_P_1 = np.linalg.inv(K_1) @ unnormalized_P_1.matrix
normalized_P_2 = np.linalg.inv(K_2) @ unnormalized_P_2.matrix
```

```
cube_1_points.homogenize()
cube_2_points.homogenize()
normalized_cube_1_points = np.linalg.inv(K_1) @ cube_1_points.data
normalized_cube_1_points = Points(normalized_cube_1_points, is_euclidian=False)
normalized_cube_2_points = np.linalg.inv(K_2) @ cube_2_points.data
normalized_cube_2_points = Points(normalized_cube_2_points, is_euclidian=False)



Vs = triangulate_3D_point_dlt(normalized_P_1, normalized_P_2, normalized_cube_1_points, normalized_cube_2_points)
Vs = Points(Vs, is_euclidian=False)
x_1_projected = normalized_P_1 @ Vs.data
x_2_projected = normalized_P_2 @ Vs.data

x_1_projected = K_1 @ x_1_projected
x_1_projected = Points(x_1_projected, is_euclidian=False)
x_2_projected = K_2 @ x_2_projected
x_2_projected = Points(x_2_projected, is_euclidian=False)
```

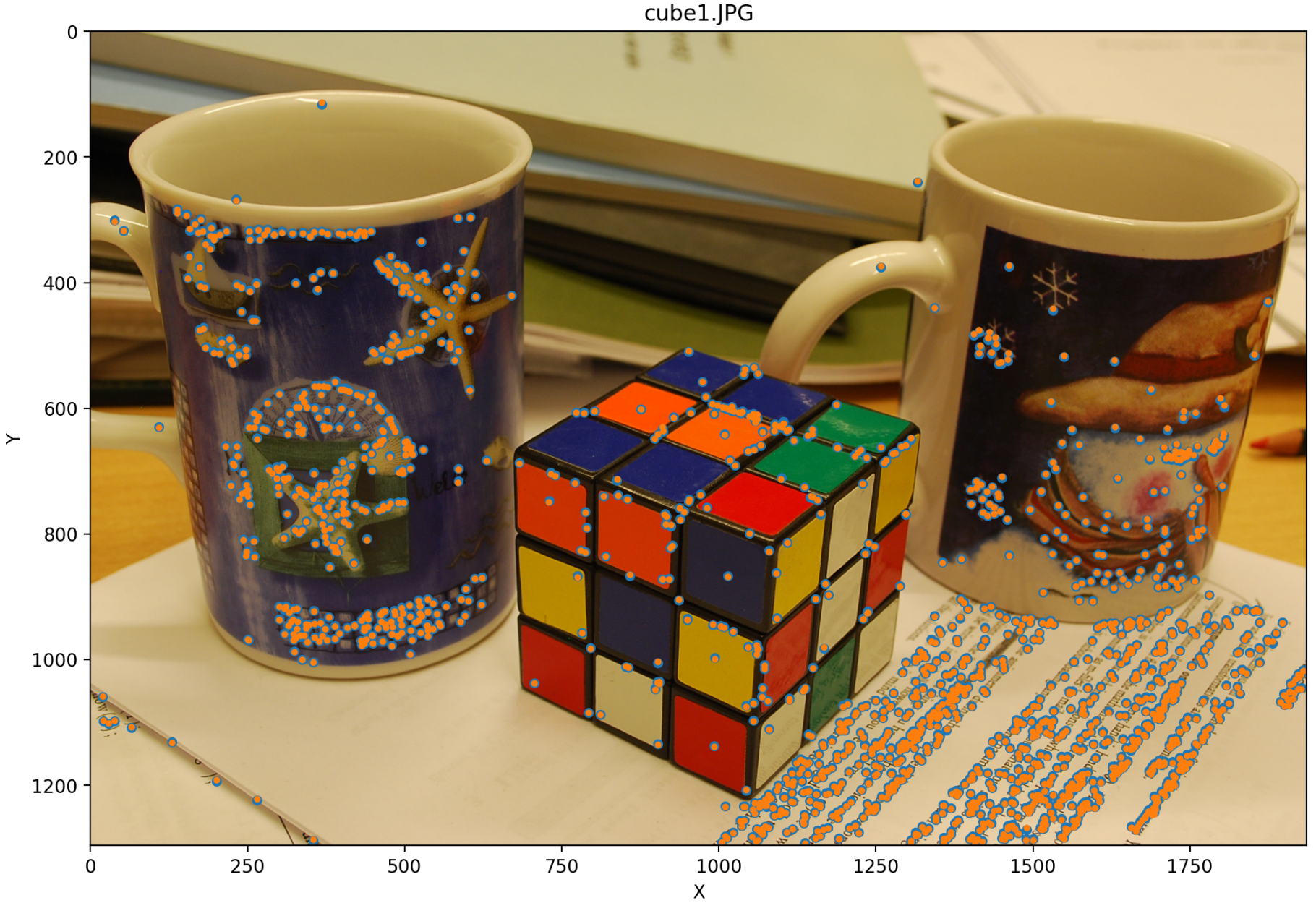
In [ ]:

```
tol = 3
x_1_projected_filt, cube_1_points_filt, x_2_projected_filt, cube_2_points_filt, X_filt = remove_error_points(
    x_1_projected.data, cube_1_points.data, x_2_projected.data, cube_2_points.data, Vs.data, tol
)
ic(x_1_projected_filt.shape)
ic(cube_1_points_filt.shape)
ic(X_filt.shape)

ax_1, size_1 = plot_image(cube_1_path)
ax_2, size_2 = plot_image(cube_2_path)
x_1_projected_filt = Points(x_1_projected_filt, is_euclidian=False)
x_2_projected_filt = Points(x_2_projected_filt, is_euclidian=False)
x_1_projected_filt.plot(ax=ax_1, marker_size=20)
x_2_projected_filt.plot(ax=ax_2, marker_size=20)
ax_1.set_xlim(0, size_1[1])
ax_1.set_ylim(size_1[0], 0)
ax_2.set_xlim(0, size_2[1])
ax_2.set_ylim(size_2[0], 0)


cube_1_points_filt = Points(cube_1_points_filt, is_euclidian=False)
```

```
cube_2_points_filt = Points(cube_2_points_filt, is_euclidian=False)
cube_1_points_filt.plot(marker_size=7, ax=ax_1)
cube_2_points_filt.plot(marker_size=7, ax=ax_2)
```

```
ic| x_1_projected_filt.shape: (3, 1921)
ic| cube_1_points_filt.shape: (3, 1921)
ic| X_filt.shape: (4, 1921)
```

Out[ ]: <Axes: title={'center': 'cube2.JPG'}, xlabel='X', ylabel='Y'>



In [ ]:
```
X_filt = Points(X_filt, is_euclidian=False)
```

In [ ]:
```
ax = X_filt.plot()
ax = X.plot(ax=ax, marker_size=20)

ax = plot_cameras([unnormalized_P_1, unnormalized_P_2], ax=ax, length=65)
ax.view_init(
    elev=-50, azim=-165, roll=80
)  # vertical_axis, azim, elev, roll. Easiest in that order.

ax.set_xlim(-20, 10)
ax.set_ylim(-17.5, 1)
ax.set_zlim(-10, 14)
ax.set_aspect("equal")
plt.show()
```