CS231A: Computer Vision, From 3D Reconstruction to Recognition Homework #1 Solution

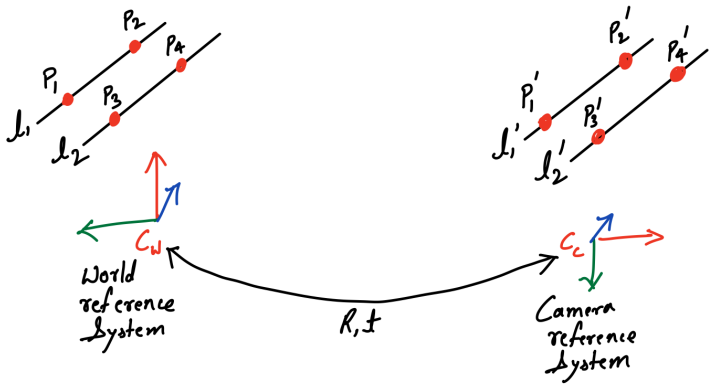
(Winter 2021)

Shubham Shrivastava email: shubhams@stanford.edu

1 Projective Geometry Problems [20 points]

In this question, we will examine properties of projective transformations. We define a camera coordinate system, which is only rotated and translated from a world coordinate system.

(a) Prove that parallel lines in the world reference system are still parallel in the camera reference system. [**4 points**]



→ Points in world coordinate system are defined as:

$P_n = (x_n, y_n, z_n)$

→ Points in camera coordinate system are defined as:

$P_n' = (x_n', y_n', z_n')$

→ Two lines in the world coordinate system are l_1 & l_2 and the corresponding lines in camera coordinate system are l_1' & l_2'.

→ World to Camera reference system transformation is given by rotation matrix, R & translation vector, t.

→ We need to prove that if $l_1 \parallel l_2$, then $l_1' \parallel l_2'$

→ Let's assume l_1 & l_2 are parallel lines in world reference system.

→ If line l_1 and l_2 are parallel, then

$$(\vec{P_1} - \vec{P_2}) \times (\vec{P_3} - \vec{P_4}) = 0$$

→ Let's take the cross product of lines l_3 & l_4.

i.e. $(\vec{P_1'} - \vec{P_2'}) \times (\vec{P_3'} - \vec{P_4'})$ —— (i)

→ After the coordinate transformation, new point coordinates are given as

$$P_1' = RP_1 + t$$
$$P_2' = RP_2 + t$$
$$P_3' = RP_3 + t$$
$$P_4' = RP_4 + t$$

$$\Rightarrow (\vec{P_1'} - \vec{P_2'}) = RP_1 + t - RP_2 - t = R(\vec{P_1} - \vec{P_2})$$

Similarly, $(\vec{P_3'} - \vec{P_4'}) = R(P_3 - P_4)$

So, $(\vec{P_1'} - \vec{P_2'}) \times (\vec{P_3'} - \vec{P_4'}) = R(\vec{P_1} - \vec{P_2}) \times R(\vec{P_3} - \vec{P_4})$

$$= R\left[(\vec{P_1} - \vec{P_2}) \times (\vec{P_3} - \vec{P_4})\right]$$

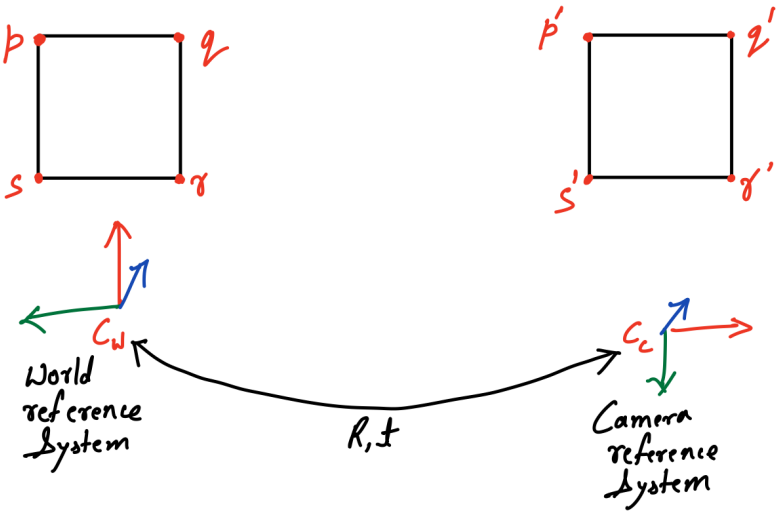
—— (ii)

Substituting (i) in (ii),

$$(\vec{P_1'} - \vec{P_2'}) \times (\vec{P_3'} - \vec{P_4'}) = 0$$

Hence, the lines are parallel.

(b) Consider a unit square $pqrs$ in the world reference system where p, q, r, and s are points. Will the same square in the camera reference system always have unit area? Prove or provide a counterexample. **[4 points]**



→ The square pqrs has unit area.

→ Area of pqrs can be computed as

$$\| (\vec{q} - \vec{p}) \times (\vec{r} - \vec{q}) \| = 1 \quad \text{(iii)}$$

→ Applying the transformation, we get

$$p' = Rp + t$$
$$q' = Rq + t$$
$$r' = Rr + t$$
$$s' = Rs + t$$

$$\vec{p'q'} = \vec{q'} - \vec{p'} = R(\vec{q} - \vec{p})$$
$$\vec{q'r'} = \vec{r'} - \vec{q'} = R(\vec{r} - \vec{q})$$

$$\text{Area of } p'q'r's' = \| (\vec{p'q'}) \times (\vec{q'r'}) \|$$

$$= \| R(\vec{q} - \vec{p}) \times R(\vec{r} - \vec{q}) \|$$

$= \| R[(\vec{q}-\vec{p}) \times (\vec{r}-\vec{q})] \|$

Recall that $\|Qa\| = \|a\|$; if Q is an orthogonal matrix
i.e. $\|Q\| = 1$

Rotation matrix, R, is an orthogonal matrix.

So, area of $p'q'r's' = \|(\vec{q}-\vec{p}) \times (\vec{r}-\vec{q})\| = 1$

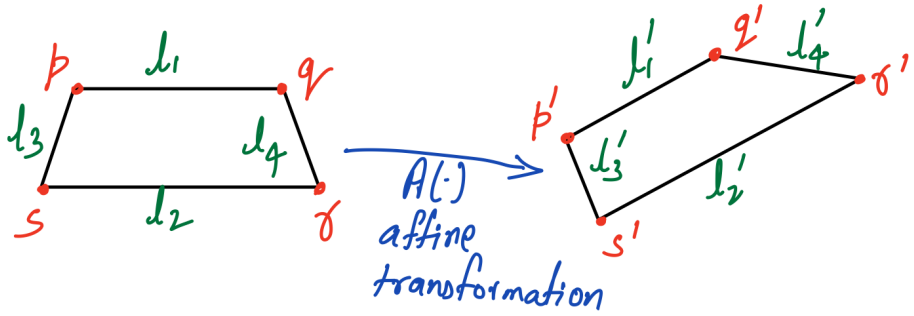
$= 1$ from (iii)

Hence, the area of transformed unit square is also 1.

(c) Now let's consider affine transformations, which are any transformations that preserve parallelism. Affine transformations include not only rotations and translations, but also scaling and shearing. Given some vector p, an affine transformation is defined as

$$A(p) = Mp + b$$

where M is an invertible matrix. Prove that under any affine transformation, the ratio of parallel line segments is invariant, but the ratio of non-parallel line segments is not invariant. **[6 points]**



→ l_1 and l_2 are parallel segments which affine transformed version are l_1' & l_2' respectively.

→ l_3 and l_4 are non-parallel segments which affine transformed version are l_3' & l_4' respectively.

→ We have to prove that

$$\frac{\|l_1\|}{\|l_2\|} = \frac{\|l_1'\|}{\|l_2'\|} \quad \text{and} \quad \frac{\|l_3\|}{\|l_4\|} \neq \frac{\|l_3'\|}{\|l_4'\|}$$

→ Let's take two points p & q on line l_1 and points r & s on line l_2. We can then define two parallel lines as: $(\vec{q} - \vec{p})$ and $(\vec{r} - \vec{s})$

→ Corresponding affine transformed lines are:
$(\vec{q'} - \vec{p'})$ and $(\vec{r'} - \vec{s'})$

→ Affine transformation is given as: $A(p) = Mp + b$

→ So,
$p' = Mp + b$
$q' = Mq + b$
$r' = Mr + b$
$s' = Ms + b$

→ $(\vec{q'} - \vec{p'}) = (Mq + b - Mp - b) = M(\vec{q} - \vec{p})$
$(\vec{r'} - \vec{s'}) = M(\vec{r} - \vec{s})$

→ For parallel lines $(\vec{q} - \vec{p})$ and $(\vec{r} - \vec{s})$, we can write $\|(\vec{q} - \vec{p})\| = a \cdot \|(\vec{r} - \vec{s})\|$ — (iv)

→ So, the ratio of two parallel lines before transformation

$$\frac{\|(\vec{q} - \vec{p})\|}{\|(\vec{r} - \vec{s})\|} = \frac{a \cdot \|(\vec{r} - \vec{s})\|}{\|(\vec{r} - \vec{s})\|} = a$$ — (v)

→ Ratio of the lines after transformation. "a" is scalar

$$\frac{\|(\vec{q}' - \vec{p}')\|}{\|(\vec{r}' - \vec{s}')\|} = \frac{\|M(\vec{q} - \vec{p})\|}{\|M(\vec{r} - \vec{s})\|} = \frac{\|M\, a(\vec{r} - \vec{s})\|}{\|M(\vec{r} - \vec{s})\|}$$

$$= \frac{a\,\|M(\vec{r} - \vec{s})\|}{\|M(\vec{r} - \vec{s})\|} = a$$

(vi)

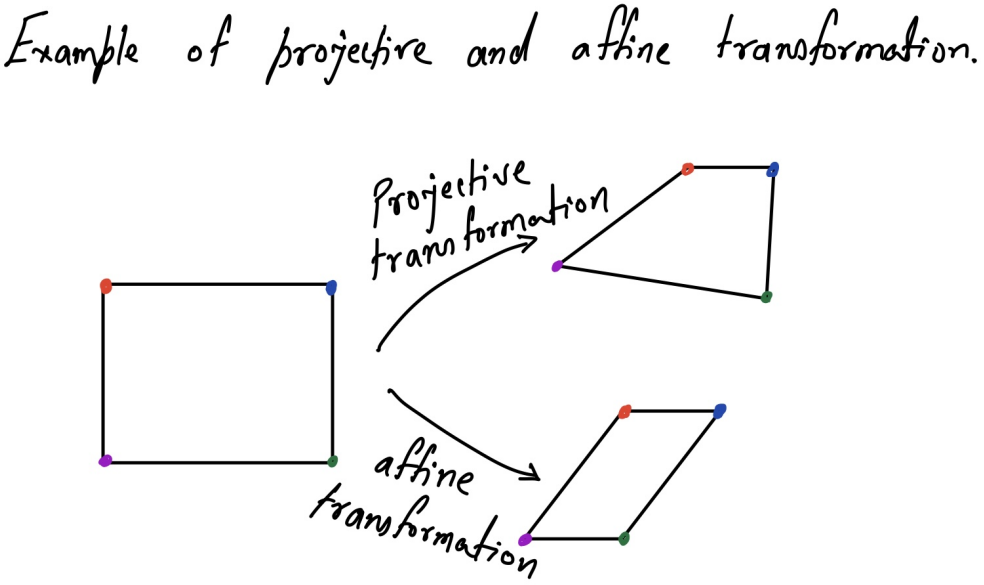
from eqn (v) & (vi), we find that the affine transform preserves ratio of parallel line segments.

→ For non-parallel lines, eqn (iv) does not hold true and hence the ratio between those line segments are not preserved after the affine transformation.

(d) You have explored whether these three properties hold for affine transformations. Do these properties hold under any projective transformation? Justify briefly in one or two sentences (no proof needed). **[6 points]**

No, affine transformation is a special case of projective transformation. Projective transformation does not preserve parallelism, length, or angle. In an affine transformation matrix, the last row is always [0, 0, 1] which preserves parallelism.

Projective transformation $\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$ $\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$ affine transformation



2 Affine Camera Calibration (35 points)

In this question, we will perform affine camera calibration using two different images of a calibration grid. First, you will find correspondences between the corners of the calibration grids and the 3D scene coordinates. Next, you will solve for the camera parameters.

It was shown in class that a perspective camera can be modeled using a 3×4 matrix:

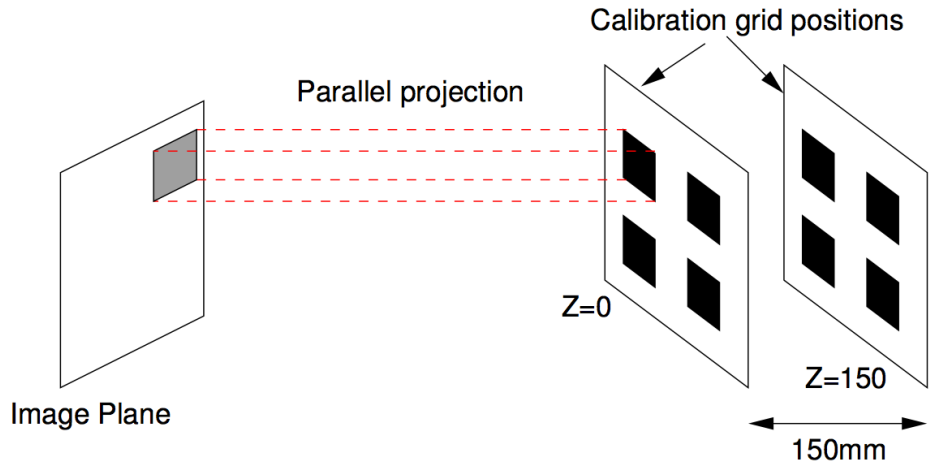
$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

which means that the image at point (X, Y, Z) in the scene has pixel coordinates $(x/w, y/w)$. The 3×4 matrix can be factorized into intrinsic and extrinsic parameters.

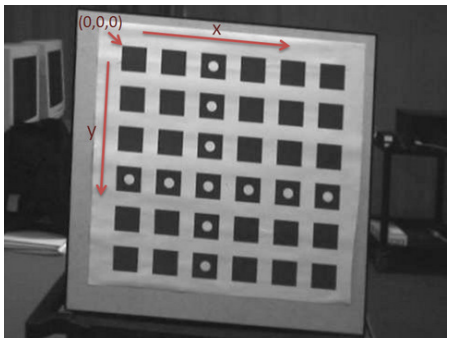
An *affine* camera is a special case of this model in which rays joining a point in the scene to its projection on the image plane are parallel. Examples of affine cameras include orthographic projection and weakly perspective projection. An affine camera can be modeled as:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

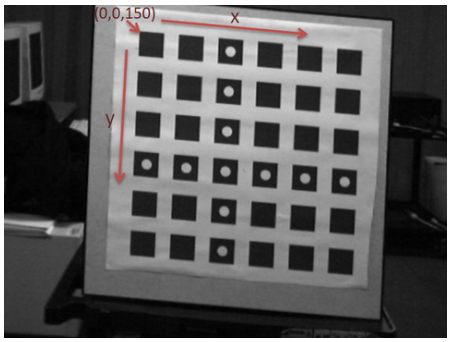
which gives the relation between a scene point (X, Y, Z) and its image (x, y). The difference is that the bottom row of the matrix is $[\ 0 \quad 0 \quad 0 \quad 1\ ]$, so there are fewer parameters we need to calibrate. More importantly, there is no division required (the homogeneous coordinate is 1) which means this is a *linear model*. This makes the affine model much simpler to work with mathematically - at the cost of losing some accuracy. The affine model is used as an approximation of the perspective model when the loss of accuracy can be tolerated, or to reduce the number of parameters being modeled. Calibration of an affine camera involves estimating the 8 unknown entries of the matrix in Eq. 2 (This matrix can also be factorized into intrinsics and extrinsics, but that is outside the scope of this homework). Factorization is accomplished by having the camera observe a calibration



(a) Image formation in an affine camera. Points are projected via parallel rays onto the image plane



(b) Image of calibration grid at Z=0



(c) Image of calibration grid at Z=150

Figure 1: Affine camera: image formation and calibration images.

pattern with easy-to-detect corners.

Scene Coordinate System

The calibration pattern used is shown in Figure 1, which has a 6×6 grid of squares. Each square is $50mm \times 50mm$. The separation between adjacent squares is $30mm$, so the entire grid is $450mm \times 450mm$. For calibration, images of the pattern at two different positions were captured. These images are shown in Fig. 1 and can be downloaded from the course website. For the second image, the calibration pattern has been moved back (along its normal) from the rest position by $150mm$.

We will choose the origin of our 3D coordinate system to be the top left corner of the calibration pattern in the rest position. The X-axis runs left to right parallel to the rows of squares. The Y-axis runs top to bottom parallel to the columns of squares. We will work in units of millimeters. All the square corners from the first position corresponds to $Z = 0$. The second position of the calibration grid corresponds to $Z = 150$. The top left corner in the first image has 3D scene coordinates $(0, 0, 0)$ and the bottom right corner in the second image has 3D scene coordinates $(450, 450, 150)$. This scene coordinate system is labeled in Fig. 1.

(a) Given correspondences for the calibrating grid, solve for the camera parameters using Eq. 2. Note that each measurement $(x_i, y_i) \leftrightarrow (X_i, Y_i, Z_i)$ yields two linear equations for the

8 unknown camera parameters. Given N corner measurements, we have $2N$ equations and 8 unknowns. Using the given corner correspondences as inputs, complete the method `compute_camera_matrix()`. You will construct a linear system of equations and solve for the camera parameters to minimize the least-squares error. After doing so, you will return the 3×4 affine camera matrix composed of these computed camera parameters. In your written report, submit your code as well as the camera matrix that you compute. **[15 points]**

Affine camera matrix

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \vec{m_1} \\ \vec{m_2} \\ 0 \; 0 \; 0 \; 1 \end{bmatrix}$$

Camera Projection

$p = MP$ → Real-world points in homogeneous coordinates (4x1)

M → Camera matrix

p → Projected 2D points

Block-matrix multiplication yields

Homogenous coordinates $p' = \begin{bmatrix} m_1 P \\ m_2 P \\ 1 \end{bmatrix}$ → Since, the last row of $M = [0,0,0,1]$ and 4th element of $P = 1$

In Euclidean coordinate system

$$p = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} m_1 P / 1 \\ m_2 P / 1 \end{bmatrix} = \begin{bmatrix} m_1 P \\ m_2 P \end{bmatrix}$$

This gives us:

$u = m_1 P$ $\quad P \rightarrow 4 \times 1$

& $v = m_2 P$ $\quad m_i \rightarrow 1 \times 4$

If we have "n" correspondences available, then we can write:

$$
\begin{aligned}
m_1 P_1 + 0 &= u_1 \\
0 + m_2 P_1 &= v_1 \\
m_1 P_2 + 0 &= u_2 \\
0 + m_2 P_2 &= v_2 \\
&\vdots \\
m_1 P_n + 0 &= u_n \\
0 + m_2 P_n &= v_n
\end{aligned}
\quad \Rightarrow \quad
\underbrace{\begin{bmatrix} P_1 & 0 \\ 0 & P_1 \\ P_2 & 0 \\ 0 & P_2 \\ \vdots & \\ P_n & 0 \\ 0 & P_n \end{bmatrix}}_{P\,(2n \times 8)}
\underbrace{\begin{bmatrix} m_1 \\ m_2 \end{bmatrix}}_{m\ (8\times 1)}
= \underbrace{\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \vdots \\ u_n \\ v_n \end{bmatrix}}_{p\,(2n\times 1)}
$$

(1×4 1×4)

We can further use least-squares to solve the system of linear equations $[P \cdot m = p]$

This is done by minimizing the energy function

$$J(m) = \| P \cdot m - p \|_2^2$$

i.e. $\frac{\partial}{\partial m} J(m) = 0 \Rightarrow P^T P \cdot m = P^T \cdot p$

$$\Rightarrow m = \left(P^T P \cdot m\right)^{-1} P^T \cdot p$$

Note that m has 8 parameters. The complete camera matrix is obtained by appending $[0, 0, 0, 1]$ to the last row.

```
    '''
    COMPUTE_CAMERA_MATRIX
    Arguments:
        real_XY - Each row corresponds to an actual point on the 2D plane
        front_image - Each row is the pixel location in the front image where Z=0
        back_image - Each row is the pixel location in the back image where Z=150
    Returns:
        camera_matrix - The calibrated camera matrix (3x4 matrix)
    '''
    def compute_camera_matrix(real_XY, front_image, back_image):
        # compose real homogeneous coordinates
        real_XYZ_front = np.zeros((real_XY.shape[0], 4))
        real_XYZ_front[:,0:2] = real_XY
        real_XYZ_front[:,2] = 0.0 # z = 0
        real_XYZ_front[:,3] = 1.0 # addition of 4th dimension in homogeneous coordinate system

        real_XYZ_back = np.zeros((real_XY.shape[0], 4))
        real_XYZ_back[:,0:2] = real_XY
        real_XYZ_back[:,2] = 150.0 # z = 150
        real_XYZ_back[:,3] = 1.0 # addition of 4th dimension in homogeneous coordinate system

        # put together the real coordinates
        real_XYZ_homogeneous = np.vstack((real_XYZ_front, real_XYZ_back))

        # put together the corresponding image projection coordinates
        projected_XY = np.vstack((front_image, back_image))

        # This gives us [projected_XY_homogeneous = M.real_XYZ_homogeneous]
        # M is a 3x4 unknown matrix. With the Affine assumption, last row of M = [0, 0, 0, 1]
        # This leaves us 8 unknown elements to compute.
        # Because of the Affine assumption, last row of M = [0, 0, 0, 1], and hence,
        # projected_XY_homogeneous[2] = 1.0. This means [projected_XY_euclidean = projected_XY_homogeneous[0:2]]
        # We need to rearrange to a form [P.m = p] where 'm' is a 8x1 unknown vector to be determined,
        # and P is [2*n x 8] known elements, where n is number of corresponding points available.
        P = np.zeros((projected_XY.shape[0]*2, 8))
        p = np.zeros((projected_XY.shape[0]*2))
        for n in range(projected_XY.shape[0]):
            # (2*n)th row
            P[2*n, 0:4] = real_XYZ_homogeneous[n]
            p[2*n] = projected_XY[n, 0]
            # (2*n+1)th row
            P[2*n+1, 4:8] = real_XYZ_homogeneous[n]
            p[2*n+1] = projected_XY[n, 1]

        # For least-square solution of the problem [P.m = p], error is minimized by setting
        # [P_transpose.P.m = P_transpose.p] ref: https://eeweb.engineering.nyu.edu/iselesni/lecture_notes/least_squares/least_squares_SP.pdf
        # This gives us the solution: [m = {P_transpose.P}_inverse.P_transpose.p]
        # Compute the least-square solution for [P.m = p]
        P_transpose = np.transpose(P)
        m = np.dot(np.dot(np.linalg.inv(np.dot(P_transpose, P)), P_transpose),
```

```
    p)

        # Alternatively, numpy can be used to directly compute this solution:
    m = np.linalg.lstsq(P, p)[0]

        # Affine Camera Matrix
        camera_matrix = np.zeros((3, 4), dtype=np.float32)
        camera_matrix[:2, :] = np.reshape(m, (2, 4))
        camera_matrix[-1,:] = [0., 0., 0., 1.]

        # return the camera matrix
        return camera_matrix

```

Listing 1: Code Snippet

The camera matrix computed using code snippet in Listing 1.

```
    Camera Matrix:
         [[ 5.3127652e-01 -1.8088607e-02  1.2050967e-01  1.2972064e+02]
         [ 4.8497546e-02  5.3636640e-01 -1.0267522e-01  4.4387962e+01]
         [ 0.0000000e+00  0.0000000e+00  0.0000000e+00  1.0000000e+00]]

```

(b) After finding the calibrated camera matrix, you will compute the RMS error between the given N image corner coordinates and N corresponding calculated corner locations in `rms_error()`. Recall that

$$\text{RMS}_{\text{total}} = \sqrt{\sum((x - x')^2 + (y - y')^2)/N}$$

Please submit your code and the RMS error for the camera matrix that you found in part (a). **[15 points]**

```
    '''
    RMS_ERROR
    Arguments:
         camera_matrix - The camera matrix of the calibrated camera
         real_XY - Each row corresponds to an actual point on the 2D plane
         front_image - Each row is the pixel location in the front image where
     Z=0
         back_image - Each row is the pixel location in the back image where Z
    =150
    Returns:
        rms_error - The root mean square error of reprojecting the points back
                    into the images
    '''
    def rms_error(camera_matrix, real_XY, front_image, back_image):
        # compose real homogeneous coordinates
        real_XYZ_front = np.zeros((real_XY.shape[0], 4))
        real_XYZ_front[:,0:2] = real_XY
        real_XYZ_front[:,2] = 0.0 # z = 0
        real_XYZ_front[:,3] = 1.0 # addition of 4th dimension in homogeneous
    coordinate system

        real_XYZ_back = np.zeros((real_XY.shape[0], 4))
        real_XYZ_back[:,0:2] = real_XY
        real_XYZ_back[:,2] = 150.0 # z = 150
        real_XYZ_back[:,3] = 1.0 # addition of 4th dimension in homogeneous
    coordinate system

        # put together the real coordinates
```

```
25        real_XYZ_homogeneous = np.vstack((real_XYZ_front, real_XYZ_back))
26
27        # put together the ground-truth corresponding image projection
    coordinates
28        projected_XY_gt = np.vstack((front_image, back_image))
29
30        # project the points onto image plane using camera_matrix
31        projected_XY = np.transpose(np.dot(camera_matrix, np.transpose(
    real_XYZ_homogeneous)))
32
33        # compute RMS Error
34        rmse = 0.0
35        for n_pt in range(projected_XY_gt.shape[0]):
36            rmse += np.power((projected_XY_gt[n_pt, 0] - projected_XY[n_pt,
    0]), 2) + \
37                    np.power((projected_XY_gt[n_pt, 1] - projected_XY[n_pt,
    1]), 2)
38        # take mean
39        rmse /= projected_XY_gt.shape[0]
40        rmse = np.sqrt(rmse)
41
42        # return rms error
43        return rmse
44
```

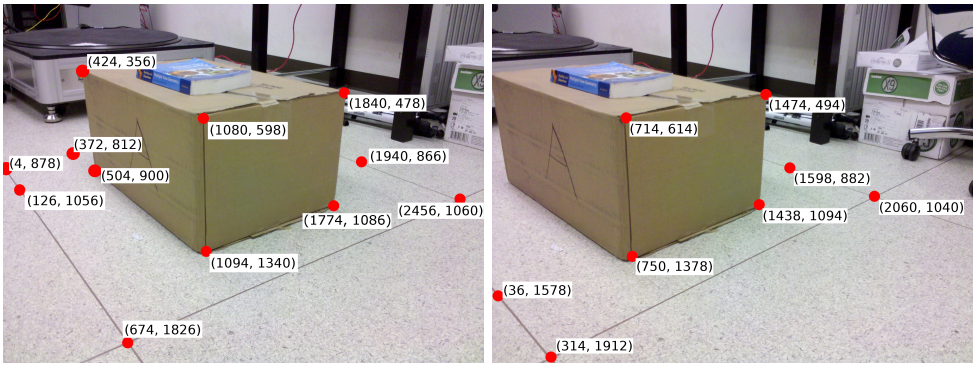
Listing 2: Code Snippet

The RMSE computed using code snippet in Listing 2: 0.9938304833006997.

(c) Could you calibrate the matrix with only one checkerboard image? Explain briefly in one or two sentences. **[5 points]**

Trying to calibrate the camera with only one checkerboard image will not provide enough constraints and will result in a degenerate solution since all points lie on the same plane in world coordinate system.

3 Single View Geometry (45 points)



(a) Image 1 (1.jpg) with marked pixels (b) Image 2 (2.jpg) with marked pixels

Figure 2: Marked pixels in images taken from different viewpoints.

In this question, we will estimate camera parameters from a single view and leverage the projective nature of cameras to find both the camera center and focal length from vanishing points present in the scene above.

(a) In Figure 2, we have identified a set of pixels to compute vanishing points in each image. Please complete `compute_vanishing_point()`, which takes in these two pairs of points on parallel lines to find the vanishing point. You can assume that the camera has zero skew and square pixels, with no distortion. **[5 points]**

```
    '''
    COMPUTE_LINE_EQUATIONS
        points - a list of two points on a line as [[x1, y1], [x2, y2]]
    Returns:
        coefficients a, b, c of line ax + by + c = 0
    '''
    def compute_line_equation(points):
        pt1 = points[0]
        pt2 = points[1]
        slope = (pt2[1] - pt1[1]) / (pt2[0] - pt1[0])
        # compute intercept by substituting one of the points in
        # equation y = slope*x + intercept
        intercept = pt2[1] - slope*pt2[0]

        # get line equation coefficients
        a = -slope
        b = 1.0
        c = -intercept
        return a, b, c

    '''
    COMPUTE_POINT_OF_INTERSECTION
        line1 - defined by its coefficients [a1, b1, c1]
        line2 - defined by its coefficients [a2, b2, c2]
    Returns:
        point of intersection (x, y)
    '''
    def compute_point_of_intersection(line1, line2):
        # ref: https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection
        (a1, b1, c1) = line1
        (a2, b2, c2) = line2
        x = (-c2 + c1) / (-a1 + a2)
        y = (-a1 * x) - c1
        return (x, y)

    '''
    COMPUTE_VANISHING_POINTS
    Arguments:
        points - a list of all the points where each row is (x, y). Generally,
                 it will contain four points: two for each parallel line.
                 You can use any convention you'd like, but our solution uses
    the
                 first two rows as points on the same line and the last
                 two rows as points on the same line.
    Returns:
        vanishing_point - the pixel location of the vanishing point
    '''
    def compute_vanishing_point(points):
        # compute line equations for two lines
        a1, b1, c1 = compute_line_equation([points[0], points[1]])
        a2, b2, c2 = compute_line_equation([points[2], points[3]])

        # compute point of intersection
        vanishing_point = compute_point_of_intersection((a1, b1, c1), (a2, b2,
     c2))
        return vanishing_point

```

(b) Using three vanishing points, we can compute the intrinsic camera matrix used to take the image. Do so in `compute_K_from_vanishing_points()`. **[10 points]**

* Angles between two vanishing points v_1 & v_2:

$$\cos\theta = \frac{v_1^T \omega v_2}{\sqrt{v_1^T \omega v_1} \cdot \sqrt{v_2^T \omega v_2}} \quad ; \text{ where, } \omega = (KK^T)^{-1}$$

for orthogonal planes, $\theta = 90^\circ \Rightarrow v_1^T \omega v_2 = 0$

$$\omega = \begin{bmatrix} w_1 & w_2 & w_4 \\ w_2 & w_3 & w_5 \\ w_4 & w_5 & w_6 \end{bmatrix}$$

If, zero-skew $\Rightarrow w_2 = 0$

square-pixel $\Rightarrow w_3 = w_1$

$$\Rightarrow \omega = \begin{bmatrix} w_1 & 0 & w_4 \\ 0 & w_1 & w_5 \\ w_4 & w_5 & w_6 \end{bmatrix}$$

$v_{13} = 1$ & $v_{23} = 1 \rightarrow$ homogeneous coordinates

$$\Rightarrow v_1^T \omega v_2 = \begin{bmatrix} v_{11} & v_{12} & v_{13} \end{bmatrix} \begin{bmatrix} w_1 & 0 & w_4 \\ 0 & w_1 & w_5 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} v_{21} \\ v_{22} \\ v_{23} \end{bmatrix}$$

$$= \begin{bmatrix} v_{11} & v_{12} & v_{13} \end{bmatrix} \begin{bmatrix} v_{21} w_1 + v_{23} w_4 \\ v_{22} w_1 + v_{23} w_5 \\ v_{21} w_4 + v_{22} w_5 + v_{23} w_6 \end{bmatrix}$$

$$= v_{11} v_{21} w_1 + v_{11} v_{23} w_4 + v_{12} v_{22} w_1 + v_{12} v_{23} w_5 + v_{13} v_{21} w_4 + v_{13} v_{22} w_5 + v_{13} v_{23} w_6$$

$$\Rightarrow v_1^T \omega v_2 = \underbrace{\begin{bmatrix} (v_{11}v_{21} + v_{12}v_{22}) & (v_{11}v_{23} + v_{13}v_{21}) & (v_{12}v_{23} + v_{13}v_{22}) & (v_{13}v_{23}) \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} w_1 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}}_{w}$$

ω can be obtained by first taking SVD of $A = UDV^T$ and then taking the last column of V^T. Elements of this column will correspond to $[w_1\ w_4\ w_5\ w_6]^T$.

K can be obtained from $\omega = (KK^T)^{-1}$ by Cholesky factorization of ω followed by inversion.

```
    '''
    COMPUTE_K_FROM_VANISHING_POINTS
    Arguments:
        vanishing_points - a list of vanishing points

    Returns:
        K - the intrinsic camera matrix (3x3 matrix)
    '''
    def compute_K_from_vanishing_points(vanishing_points):
        # form equations A.w = 0 with 4 constraints of omega (w)
        # A = np.zeros((vanishing_points.shape[0], 4), dtype=np.float32)
        A = []
        for i, point_i in enumerate(vanishing_points):
            for j, point_j in enumerate(vanishing_points):
                if i != j and j > i:
                    point_i_homogeneous = [point_i[0],point_i[1],1.0]
                    point_j_homogeneous = [point_j[0],point_j[1],1.0]
                    A.append([point_i_homogeneous[0]*point_j_homogeneous[0]+
point_i_homogeneous[1]*point_j_homogeneous[1], \
                              point_i_homogeneous[0]*point_j_homogeneous[2]+
point_i_homogeneous[2]*point_j_homogeneous[0], \
                              point_i_homogeneous[1]*point_j_homogeneous[2]+
point_i_homogeneous[2]*point_j_homogeneous[1], \
                              point_i_homogeneous[2]*point_j_homogeneous[2]])
        A = np.array(A, dtype=np.float32)
        u, s, v_t = np.linalg.svd(A, full_matrices=True)
        # 4 constraints of omega (w) can be obtained as the last column of v
or last row of v_transpose
        w1, w4, w5, w6 = v_t.T[:,-1]
        # form omega matrix
        w = np.array([[w1, 0., w4],
                      [0., w1, w5],
                      [w4, w5, w6]])
        # w = (K.K_transpose)^(-1)
        # K can be obtained by Cholesky factorization followed by its inverse
        K_transpose_inv = np.linalg.cholesky(w)
        K = np.linalg.inv(K_transpose_inv.T)
        # divide by the scaling factor
        K = K / K[-1, -1]

        # return intrinsic matrix
        return K

```

(c) Is it possible to compute the camera intrinsic matrix for any set of vanishing points? Similarly,

is three vanishing points the minimum required to compute the intrinsic camera matrix? Justify your answer. **[5 points]**

No, it is not possible to compute the camera intrinsic matrix for any set of vanishing points. If all the vanishing points used lie in the same plane in world coordinate system, then this will result in a degenerate solution. We need at least 3 sets of vanishing points which lie on a different plane (not necessarily orthogonal) - with the assumption that the camera has *zero skew* and *square pixels*. If the assumptions do not hold, then we will need at least 5 sets of vanishing point to get enough constraints for computing the camera intrinsic matrix.

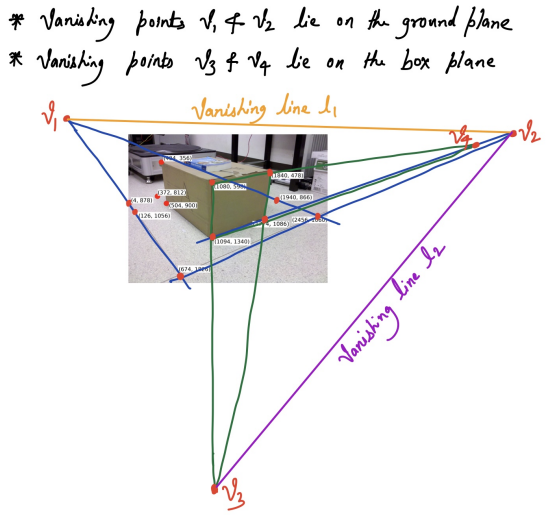
(d) The method used to obtain vanishing points is approximate and prone to noise. Discuss approaches to refine this process. **[5 points]**

The method used here is approximate and prone to noise. We use only two lines per plane to compute a vanishing point by computing the intersection point - this is not optimal. A better method of computing vanishing point would be to first use multiple points per line and then fit a line through them - rather than using just two points per line. Secondly, we would need to determine a set of line segments that intersect in a single point, and which minimizes the sum of squared orthogonal distances from the endpoints of the measured line segments - this minimization could be performed using the Levenberg-Marquardt algorithm.

(e) This process gives the camera internal matrix under the specified constraints. For the remainder of the computations, use the following internal camera matrix:

$$K = \begin{bmatrix} 2448 & 0 & 1253 \\ 0 & 2438 & 986 \\ 0 & 0 & 1 \end{bmatrix}$$

Identify a sufficient set of vanishing lines on the ground plane and the plane on which the letter A exists, written on the side of the cardboard box, (plane-A). Use these vanishing lines to verify numerically that the ground plane is orthogonal to the plane-A. Fill out the method `compute_angle_between_planes()` and submit your code and the computed angle. **[10 points]**



Angles between the two planes can be calculated using the two vanishing lines as:

$$\theta = \cos^{-1}\left\{\frac{\ell_1^T \omega^{-1} \ell_2}{\sqrt{\ell_1^T \omega^{-1} \ell_1}\sqrt{\ell_2^T \omega^{-1} \ell_2}}\right\}$$

```
    '''
    COMPUTE_ANGLE_BETWEEN_PLANES
    Arguments:
        vanishing_pair1 - a list of a pair of vanishing points computed from lines within the same plane
        vanishing_pair2 - a list of another pair of vanishing points from a different plane than vanishing_pair1
        K - the camera matrix used to take both images

    Returns:
        angle - the angle in degrees between the planes which the vanishing point pair comes from2
    '''
    def compute_angle_between_planes(vanishing_pair1, vanishing_pair2, K):
        # compute vanishing line from first pair of vanishing points
        vanishing_line1 = np.array(compute_line_equation(vanishing_pair1)).transpose()
        # compute vanishing line from second pair of vanishing points
        vanishing_line2 = np.array(compute_line_equation(vanishing_pair2)).transpose()
        # compute omega inverse
        w_inv = np.dot(K, K.transpose())
        # compute angle between these two planes
        l1T_winv_l2 = np.dot(vanishing_line1.transpose(), np.dot(w_inv, vanishing_line2))
        sqrt_l1T_winv_l1 = np.sqrt(np.dot(vanishing_line1.transpose(), np.dot(w_inv, vanishing_line1)))
        sqrt_l2T_winv_l2 = np.sqrt(np.dot(vanishing_line2.transpose(), np.dot(w_inv, vanishing_line2)))
        theta = np.arccos(l1T_winv_l2 / np.dot(sqrt_l1T_winv_l1, sqrt_l2T_winv_l2))
        # convert the angle between planes to degrees and return
        return np.degrees(theta)

```

Listing 3: Code Snippet

The computed angle between planes using code in Listing 3 is: 89.97263875896901 degrees. Hence, we numerically verify that the two planes are orthogonal.

(f) Assume the camera rotates but no translation takes place. Assume the internal camera parameters remain unchanged. An Image 2 of the same scene is taken. Use vanishing points to estimate the rotation matrix between when the camera took Image 1 and Image 2. Fill out the method `compute_rotation_matrix_between_cameras()` and submit your code and your results. [**10 points**]

```
    '''
    COMPUTE_ROTATION_MATRIX_BETWEEN_CAMERAS
    Arguments:
        vanishing_points1 - a list of vanishing points in image 1
        vanishing_points2 - a list of vanishing points in image 2
        K - the camera matrix used to take both images

    Returns:
        R - the rotation matrix between camera 1 and camera 2
    '''
    def compute_rotation_matrix_between_cameras(vanishing_points1, vanishing_points2, K):
        ## estimate real-world direction vectors given vanishing points
        # first image
        d1i = []
```

```
        for v1i in vanishing_points1:
            # vanishing point (v) and 3-dimensional direction vector (d) are related as [d = K.v]
            v1i_homogeneous = np.array([v1i[0], v1i[1], 1.0])
            KinvV = np.dot(np.linalg.inv(K), v1i_homogeneous.T)
            d1i.append(KinvV / np.sqrt(KinvV[0]**2 + KinvV[1]**2 + KinvV[2]**2)) # normalize to make sure you obtain a unit vector
        d1i = np.array(d1i)
        # second image
        d2i = []
        for v2i in vanishing_points2:
            # vanishing point (v) and 3-dimensional direction vector (d) are related as [d = K.v]
            v2i_homogeneous = np.array([v2i[0], v2i[1], 1.0])
            KinvV = np.dot(np.linalg.inv(K), v2i_homogeneous.T)
            d2i.append(KinvV / np.sqrt(KinvV[0]**2 + KinvV[1]**2 + KinvV[2]**2)) # normalize to make sure you obtain a unit vector
        d2i = np.array(d2i)

        # the directional vectors in image 1 and image 2 are related by a rotation, R i.e. [d2i = R.d1i] => [R = d2i.d1i_inverse]
        R = np.dot(d2i.T, np.linalg.inv(d1i.T))
        return R

```

Listing 4: Code Snippet

Camera rotation between the two image captures computed from vanishing points using code snippet in Listing 4 is given as:

```
Rotation between two cameras:
     [[ 0.96154157  0.04924778 -0.15783349]
     [-0.01044314  1.00703585  0.04571333]
     [ 0.18940319 -0.06891607  1.00470583]]

Angle around z-axis (pointing out of camera): -2.931986 degrees
Angle around y-axis (pointing vertically): -8.918793 degrees
Angle around x-axis (pointing horizontally): -2.605117 degrees

```