

# Augmented reality wireframe cube

## Contents

<b>1 Preliminaries</b>	<b>1</b>
1.1 Outline of the exercise . . . . .	1
1.2 Description of the input data . . . . .	1
1.3 Notations and coordinate systems . . . . .	2
<b>2 Part 1: Drawing a cube on the undistorted images</b>	<b>2</b>
2.1 Reminder: Perspective projection . . . . .	3
2.1.1 Equation of perspective projection . . . . .	3
2.1.2 Axis-angle representation for rotations . . . . .	4
2.2 Writing and testing the projection function . . . . .	4
2.3 Drawing the cube . . . . .	4
2.4 (Optional) Generating a video from the images . . . . .	4
<b>3 Part 2: Accounting for lens distortion</b>	<b>5</b>
3.1 Lens distortion modelling . . . . .	5
3.2 Writing and testing the projection function with lens distortion . . . . .	5
3.3 Undistorting the images . . . . .	6
<b>4 Numerical Exercises</b>	<b>8</b>

## 1 Preliminaries

### 1.1 Outline of the exercise

The goal of this exercise is to superimpose a virtual cube on a video of a planar grid viewed from different orientations. **In this exercise, the 3D positions of the checkerboard, and the relative camera poses are provided**, as well as the intrinsics of the camera. The purpose of this exercise is to familiarize with the basics of perspective projection, change of coordinate systems and lens distortion, as well as basic image processing with Matlab/Python.

In the first part of the exercise, you will be given images that have already been compensated for distortion, and you will write a function that draws a virtual cube on the compensated image.

In the second part, you will implement a simple distortion and use it to undistort the camera image.

### 1.2 Description of the input data

The *data/* folder contains the inputs that you will need to complete these exercises. *You should be able to load all text files with the load command of Matlab or np.loadtxt in Python.*

- *images/* contains a sequence of images recorded by a camera moving around a checkerboard pattern.

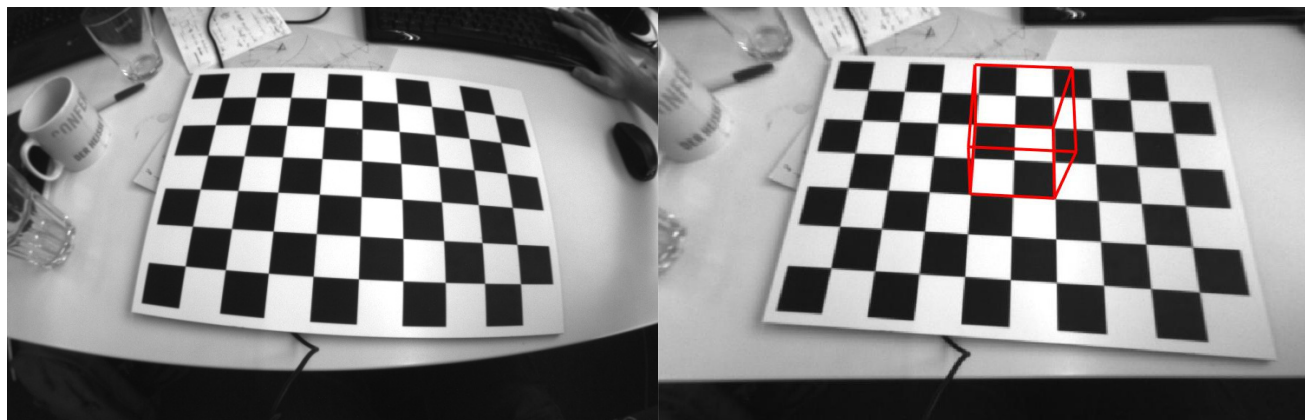


Figure 1: Input image (left) and expected output (right): undistorted image with a virtual cube superimposed

- *images\_undistorted/* contains images that have been processed to compensate for lens distortion. You will use them in the first part of the exercise. In the second part, you will write code to generate these compensated images from the original images yourself.
- *K.txt* and *D.txt* contain the intrinsics of the camera
- *poses.txt* contains the poses of the camera for each image, given as the transformation  ${}^{cam}T_w$  that maps points in the world coordinate system (defined below) to the camera coordinate system. Specifically, line  $i$  contains the pose of the camera  $i$ , given as a tuple:  $(\omega_x, \omega_y, \omega_z, t_x, t_y, t_z)$  where  $(\omega_x, \omega_y, \omega_z) = \omega$  is an axis-angle representation (section 2.1.2) of the rotational part of the transformation, and  $(t_x, t_y, t_z) = \mathbf{t}$  the translational part in meters.

### 1.3 Notations and coordinate systems

In this exercise, we use the following conventions:

- $\mathbf{P}_A$  denotes that the point  $\mathbf{P}$  is expressed in the coordinate frame  $A$ .
- ${}^B T_A$  denotes the transformation that maps points in frame  $A$  to frame  $B$ , such that:

$$\mathbf{P}_B = {}^B T_A \mathbf{P}_A$$

The reference (or world) coordinate system, denoted  $W$ , is right-handed, and centered on the upper left corner of the checkerboard, as illustrated in Figure 2. The size of each square of the checkerboard is 4 cm.

## 2 Part 1: Drawing a cube on the undistorted images

In this section, you will work with an image that has been already compensated for lens distortion. Your goal will be to create a 3D cube lying on the checkerboard and project it into the image (Figure 2).

You will first write a function that projects world points to a given image (knowing the corresponding camera pose), and test it by reprojecting the checkerboard corners on the image. Once your projection function works properly, you will create a cube in the world frame and draw it on the image.

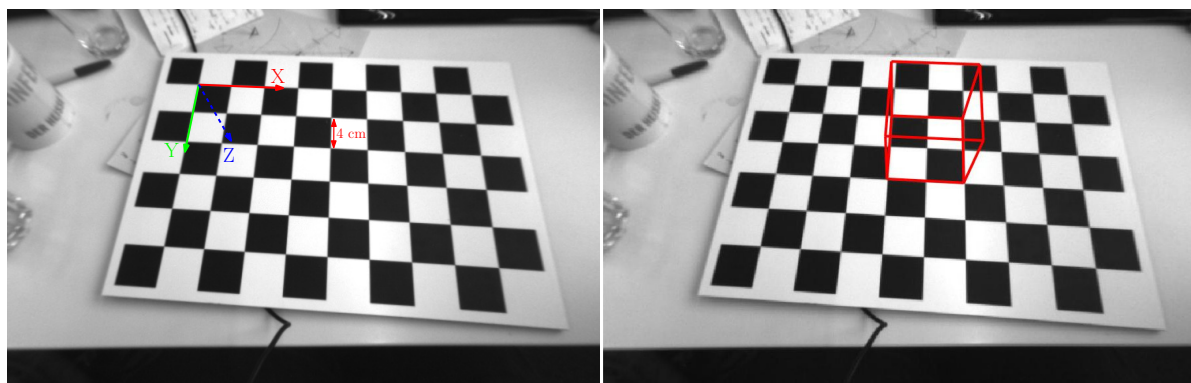


Figure 2: World coordinate system  $W$  (left), and superimposed cube lying on the checkerboard (right).

## 2.1 Reminder: Perspective projection

Figure 3 is a reminder of the different steps involved in projecting a 3D point  $\mathbf{P}_w$  (expressed in the world coordinate frame) to the image plane of camera  $C$ , when the intrinsics (camera matrix  $K$  and transformation  $[R|t]$ ) are known.

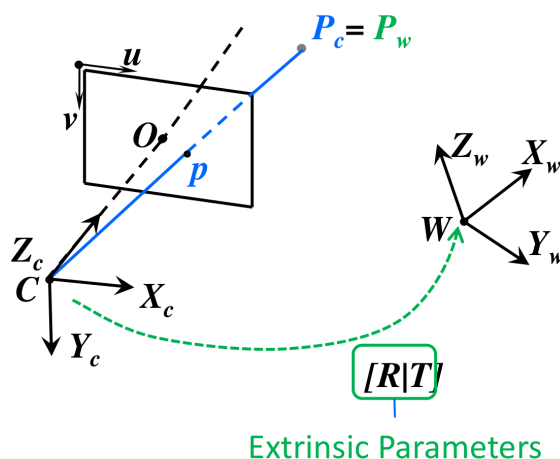


Figure 3: Perspective projection: the point  $\mathbf{P}_w$  is first expressed in the camera frame  $C$  through  $[R|t]$  (to  $\mathbf{P}_c = (X_c, Y_c, Z_c)^T$ ), then mapped to the image plane by perspective projection (to  $\mathbf{p} = (x, y)^T$ ), and finally converted to discretized pixel coordinates  $(u, v)$ .

### 2.1.1 Equation of perspective projection

Assuming the lens distortion has already been compensated (which is the case in this section), the perspective projection can be written *linearly* in homogeneous coordinates as shown in the lecture:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K[R|t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad (1)$$

where  $(u, v)^T$  is the desired projection given in pixel coordinates, and  $\mathbf{P}_w = (X_w, Y_w, Z_w)^T$ .  $K$  is a  $3 \times 3$  matrix also called the *camera matrix*. This matrix is provided to you in ***K.txt***.

### 2.1.2 Axis-angle representation for rotations

In this exercise, the rotation  $R$  from the world frame to the camera frame is given using the axis-angle representation for rotations. Specifically, a 3D rotation is parameterized by a 3D vector  $\omega = (\omega_x, \omega_y, \omega_z)^T$ , where  $\mathbf{k} = \frac{\omega}{\|\omega\|}$  is a unit vector indicating the axis of rotation, and  $\|\omega\| = \theta$  is the magnitude of the rotation about the axis. Rodrigues' rotation formula allows to convert this representation to a rotation matrix:

$$R = I + (\sin \theta) [\mathbf{k}]_{\times} + (1 - \cos \theta) [\mathbf{k}]_{\times}^2$$

where  $[\mathbf{k}]_{\times} = \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix}$  is the *cross-product matrix* for the vector  $\mathbf{k}$ .

## 2.2 Writing and testing the projection function

We will work with the first image, located in `/data/images_undistorted/img_0001.jpg`.

- Read the image
  - into Matlab (`imread`)
  - into Python (`cv2.imread`)
 and convert it to grayscale (`rgb2gray` in MATLAB or `cv2.IMREAD_GRAYSCALE` in Python).
- Create a matrix containing the 3D positions of all the checkerboard corners  $\mathbf{P}_w$ . You can use
  - The function `meshgrid` from MATLAB
  - The numpy function `np.meshgrid` in Python
 to achieve this.
- Write a function to project the corners  $\mathbf{P}_w$  on the image plane. You will need the transformation  $[R|\mathbf{t}]$  from world coordinates to camera coordinates, which you can read from the first line of the file `poses.txt`, as a tuple  $(\omega_x, \omega_y, \omega_z, t_x, t_y, t_z) = (\omega, \mathbf{t})$ . You will find it convenient to write two functions `poseVectorToTransformationMatrix` and `projectPoints`.
- Superimpose the projected corners to the undistorted image using
  - `scatter` in MATLAB
  - matplotlib function `pyplot.plot` in Python

for example. The output should look like Figure 4 if your code works properly.

## 2.3 Drawing the cube

- Write some code to create a matrix containing the 8 vertices of a cube lying on the checkerboard's plane. The position of the cube on the checkerboard and its size should be customizable.
- Project the cube's vertices on the image and draw a line (*line*) for each edge of the cube.

Figure 1 illustrates the expected output.

## 2.4 (Optional) Generating a video from the images

Repeat the process above for all the images in the sequence and generate a small movie (at 30 frames per second). <https://ch.mathworks.com/help/matlab/examples/convert-between-image-sequences-and-video.html>

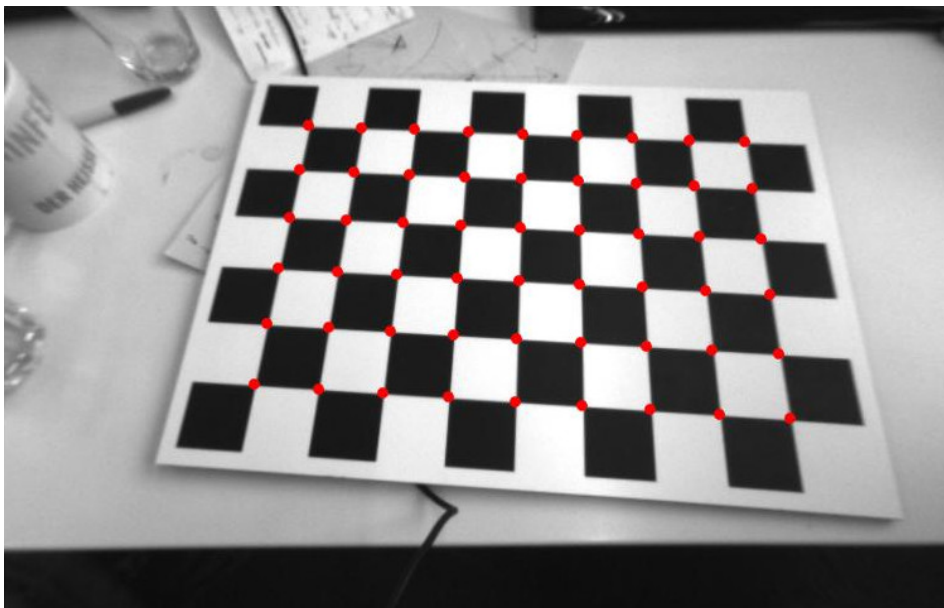


Figure 4: Expected output: the checkerboard corners are reprojected (in red) at their correct position on the undistorted image.

## 3 Part 2: Accounting for lens distortion

### 3.1 Lens distortion modelling

Real camera lenses are not ideal and introduce some distortion in the image. To account for these non-idealities, it is necessary to add a distortion model to the equations of perspective projection. A simple *radial distortion model* was introduced during the lecture. In this exercise, we use this model, and simply add a higher-order term, parameterized by an additional variable  $k_2$ . The distortion model is therefore fully parameterized by two variables ( $k_1, k_2$ ) that are provided in the file ***D.txt***.

Because the distortion model is not linear, the projection function needs to split into two steps.

- We start again by mapping the world point  $\mathbf{P}_w$  to pixel coordinates  $\mathbf{p} = (u, v)^T$ , as in Equation 1):

$$\lambda \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K[R|\mathbf{t}] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

- Next, we apply lens distortion to  $\mathbf{p}$  to get the *distorted pixel coordinates*  $\mathbf{p}_d = (u_d, v_d)^T$ :

$$\begin{pmatrix} u_d \\ v_d \end{pmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{pmatrix} u - u_0 \\ v - v_0 \end{pmatrix} + \begin{pmatrix} u_0 \\ v_0 \end{pmatrix} \quad (2)$$

where  $r^2 = (u - u_0)^2 + (v - v_0)^2$  is the radial component of  $\mathbf{p}$  and the values  $u_0$  and  $v_0$  denote the optical center (derived from the calibration matrix  $K$ ).

### 3.2 Writing and testing the projection function with lens distortion

- Read the image `/data/images/img_0001.jpg` which is, this time, not compensated for distortion.
- Modify your function `project_points` to take into account the lens distortion, as described above.

- Project the checkerboard corners in the distorted image. The expected output is shown in Figure 5.

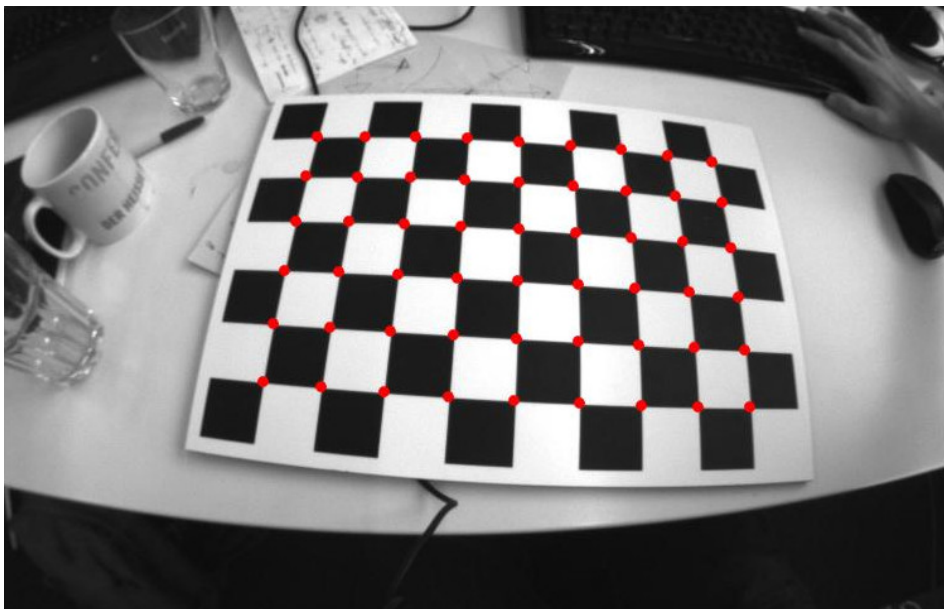


Figure 5: Expected output: the checkerboard corners are reprojected (in red) at their correct position on the non-corrected image.

### 3.3 Undistorting the images

We will now use the new projection function (that takes distortion into account) to generate an undistorted image from the original image. Let  $I_d$  and  $I_u$  be respectively the distorted and undistorted images.

A naive way to undistort  $I_d$  would be through forward warping, i.e warp every pixel  $(u', v')^T$  in  $I_d$  to  $I_u$  as follows:

$$I_u(\Gamma^{-1}(u', v')) = I_d(u', v')$$

where  $\Gamma(u, v) = (u', v')$  is the distortion function that maps undistorted pixel coordinates  $(u, v)^T$  to distorted pixel coordinates  $(u', v')^T$ .

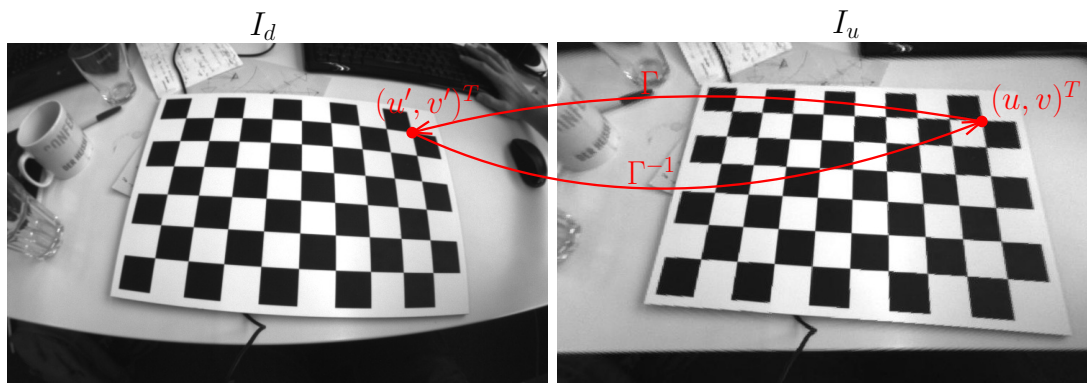


Figure 6: Distorted pixel coordinates  $(u', v')^T$  are undistorted to  $(u, v)^T$  through the distortion function  $\Gamma$ .

However, due to the undistorted pixel locations being non-integer, the resulting image would have some artifacts. Moreover, inverting the distortion function  $\Gamma$  amounts to solving a system of polynomial system of equations, which is costly.

In image processing, this is commonly solved by doing backward warping, i.e. warping pixel locations from the *destination image* (undistorted image in our case) to the *source image* (distorted image in our case):

$$I_u(u, v) = I_d(\Gamma(u, v)) \quad (3)$$

Since  $\Gamma(u, v) = (u', v')^T$  are non-integer pixel locations, the image intensity  $I_d(u', v')$  must be estimated at the *non-integer* pixel location  $(u', v')$ . The most simple way to do it is through *nearest-neighbor interpolation*, i.e. approximating  $I_d(u', v') \simeq I_d(\lfloor u' \rfloor, \lfloor v' \rfloor)$ , where  $\lfloor x \rfloor$  denotes the closest integer to  $x$ .

- Write a function `undistort_image` that performs the image undistortion using Equation 3 and nearest-neighbor interpolation. Keep in mind that the distortion function (defined by Equation 2) works with *normalized pixel coordinates* and not pixel coordinates. The expected output is shown in Figure 7 (left image).

**Note** `for` loops are very inefficient in Matlab, although we do it here for simplicity. As an additional question, you can try to implement the `undistort_image` function using vectorization (hint: Matlab's and Python's `reshape` function might be handy). The resulting code will be faster.

**Note** Matlab has a `imwarp` function dedicated to this kind of operations, similar to the OpenCV function `cv2.warpAffine` in Python.

**Bonus exercise** Implement *bilinear interpolation* to get rid of the artifacts introduced by nearest-neighbor interpolation (see Figure 7 for comparison).

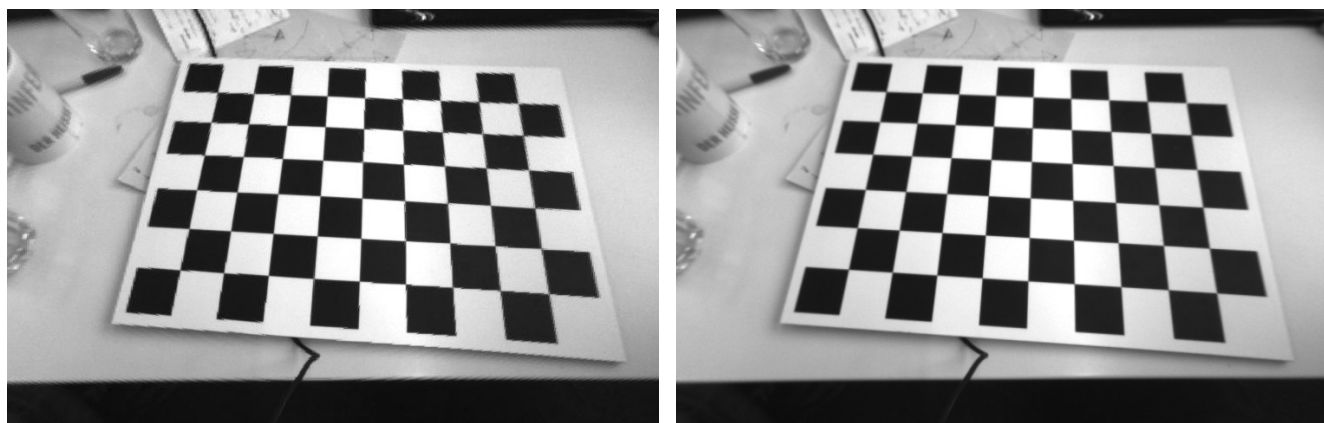


Figure 7: Undistorted images. Left: nearest-neighbor interpolation (observe the artifacts on the edges). Right: bilinear interpolation.

## 4 Numerical Exercises

1. Which of the following points written in homogeneous coordinates represent the same point in  $\mathbb{P}^2$  as  $[6, 3, 2]$ ?

(a)  $[18, 9, 6]$

(b)  $[12, -6, 4]$

(c)  $[1, \frac{1}{2}, \frac{1}{3}]$

(d)  $[1, 2, 3]$

2. Consider the following projective transformation  $\lambda q = HP$ , which maps a world point  $P$  to the image point  $q$  with

$$H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Compute the vanishing points for lines in the XZ plane (1) parallel to the Z-axis and (2) at  $45^\circ$  to the Z axis.

3. Consider a camera with the following camera matrix

$$K = \begin{bmatrix} f & 0 & u_x \\ 0 & f & u_y \\ 0 & 0 & 1 \end{bmatrix}.$$

State the modified camera matrix  $\hat{K}$  if the image resolution is increased by a factor of two.

4. Compute the image point  $q$  in pixels corresponding to the point  $P = [2, 5, 5]$  expressed in the world frame in meters. Consider the following camera extrinsics: rotation matrix  $R$  and translation vector  $T$  (in meters) and camera matrix  $K$

$$R = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$T = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

$$K = \begin{bmatrix} 420 & 0 & 355 \\ 0 & 420 & 250 \\ 0 & 0 & 1 \end{bmatrix}.$$