Augmented reality wireframe cube

Contents

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.

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.*

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

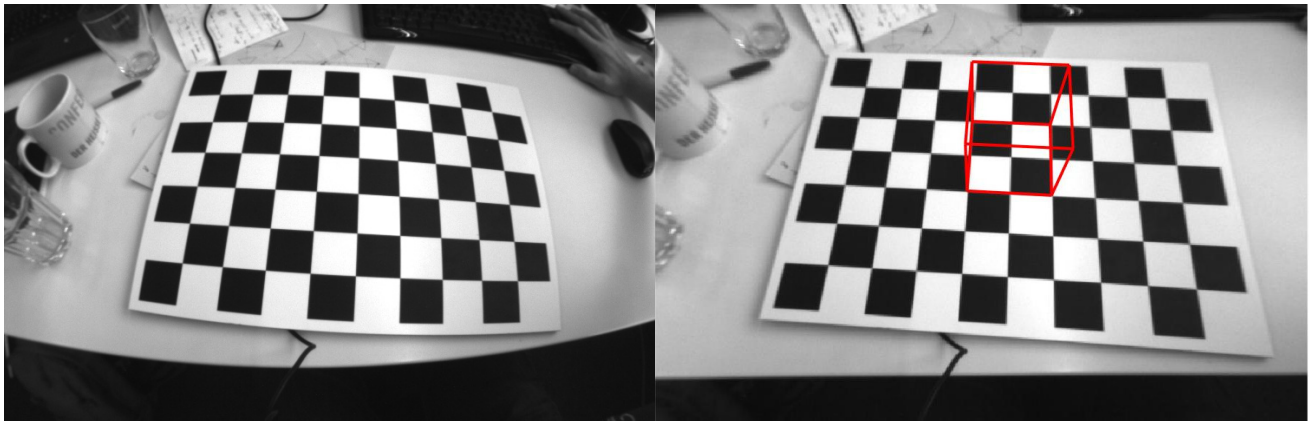

Figure 1: Input image (left) and expected output (right): undistorted image with a virtual cube superimposed

- ***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.
- ${}^BT_A$ denotes the transformation that maps points in frame A to frame B, such that:

$$\mathbf{P}_B = {}^BT_A\mathbf{P}_A$$

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 if the checkerboard is 4 cm.

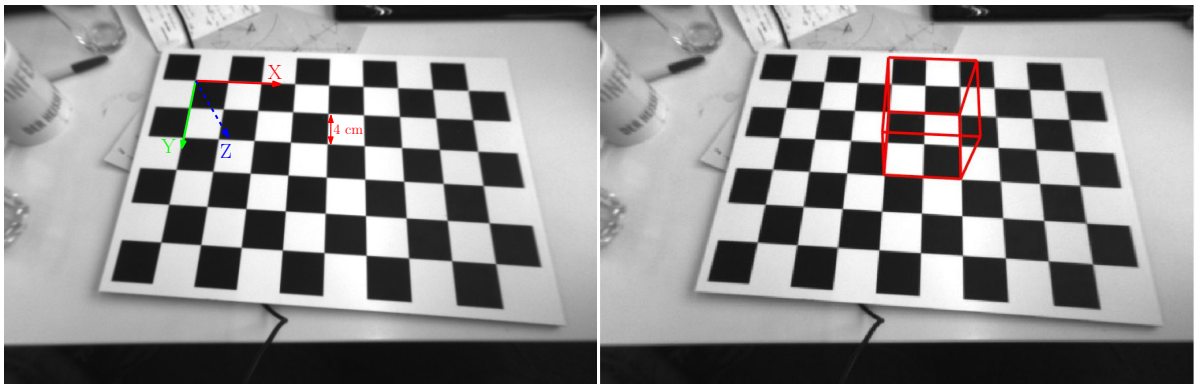

Figure 2: World coordinate system W (left), and superimposed cube lying on the checkerboard (right).

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.

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 $[R|\mathbf{t}]$) are known.

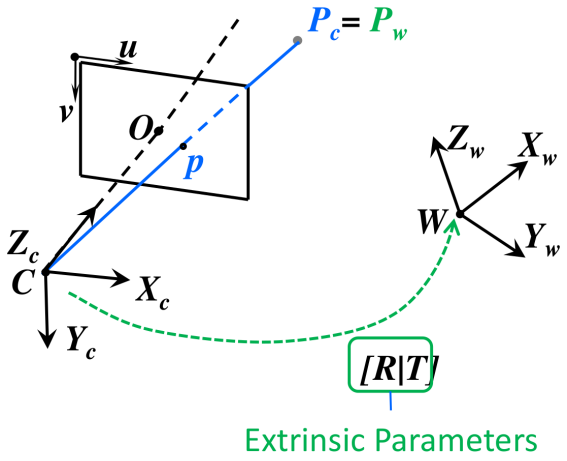

Figure 3: Perspective projection: the point $\mathbf{P}_w$ is first expressed in the camera frame C through $[R|\mathbf{t}]$ (to $\mathbf{P}_c = (X_c, Y_c, Z_c)^T$), then mapped to the image plane by perspective projection (to $\mathbf{p} = (x, y)^T$), and finally converted to discretized pixel coordinates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

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

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 3×3 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)* and convert it to grayscale (rgb2gray).
- Create a matrix containing the 3D positions of all the checkerboard corners $\mathbf{P}_w$. You can use the function *meshgrid* from Matlab 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* for example). The output should look like Figure 4 if your code works properly.

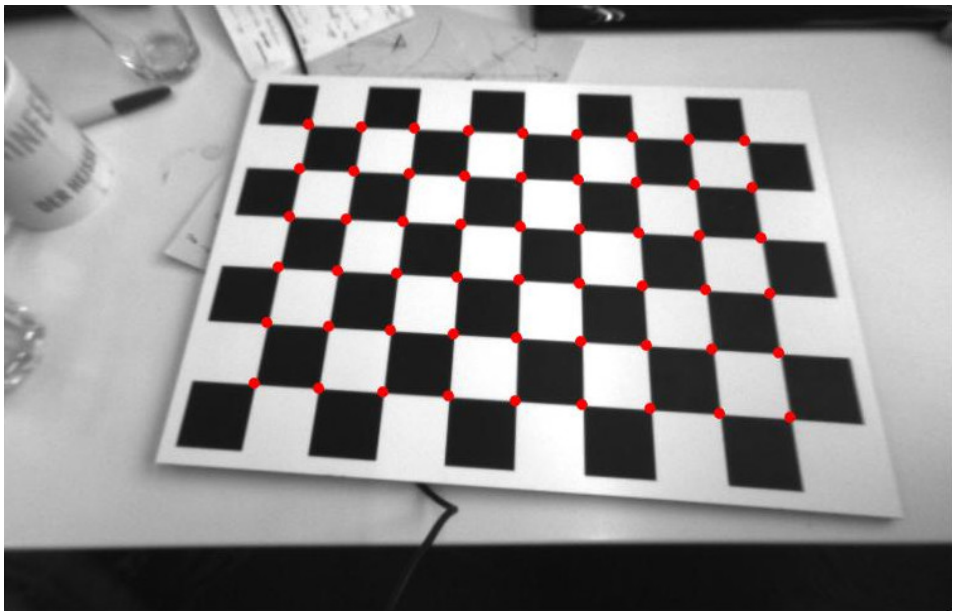

Figure 4: Expected output: the checkerboard corners are reprojected (in red) at their correct position on the undistorted image.

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

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 is not linear in homogeneous coordinates anymore (as opposed to Equation 1), thus the projection function needs to be split into several steps as follows:

- Map the world point $\mathbf{P}_w$ to the camera frame: $\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$
- Project the point to the image plane to get the *normalized coordinates* $\mathbf{p} = (x, y)^T$:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{X_c}{Z_c} \\ \frac{Y_c}{Z_c} \end{pmatrix}$$

- Apply lens distortion to $\mathbf{p}$ to get the *distorted normalized coordinates* $\mathbf{p}_d = (x', y')^T$:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{pmatrix} x \\ y \end{pmatrix} \tag{2}$$

  where $r^2 = x^2 + y^2$ is the radial component of $\mathbf{p}$.

- Convert the distorted normalized coordinates $\mathbf{p}_d$ to get the discretized pixel coordinates $(u, v)^T$:

$$\lambda \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

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.

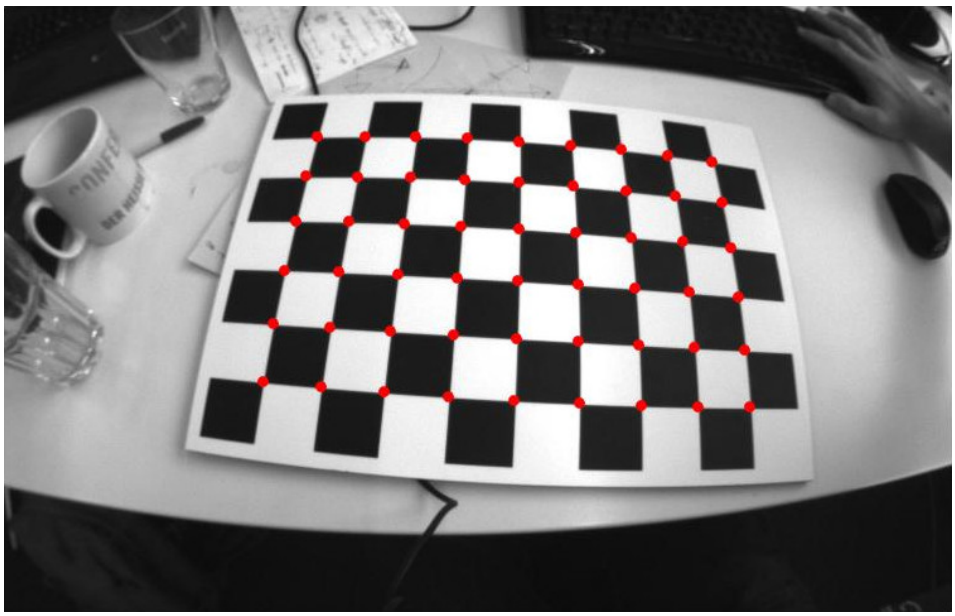

Figure 5: Expected output: the checkerboard corners are reprojected (in red) at their correct position on the non-corrected image.

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 $(u', v')^T$.

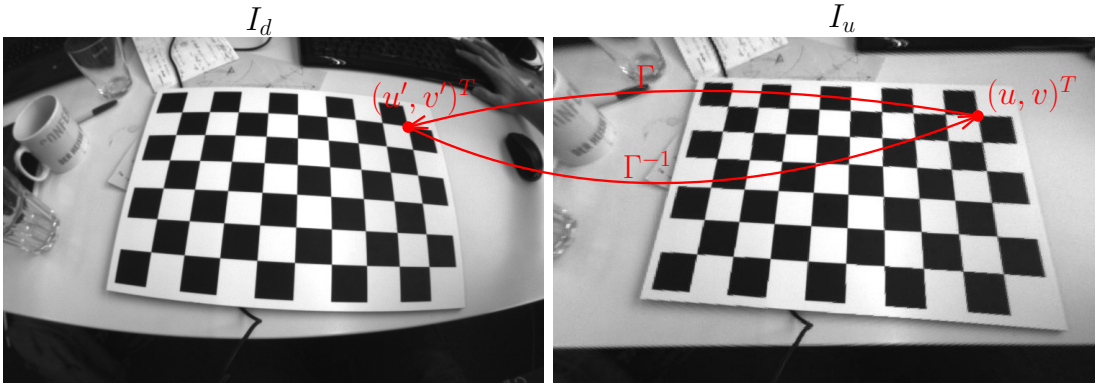

Figure 6: Distorted pixel coordinates $(u', v')^T$ are undistorted to $(u, v)^T$ through the distortion function Γ.

However, due to the undistorted pixel locations being non-integer, the resulting image would have some artifacts. Moreover, inverting the distortion function Γ 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):

$$I_u(u, v) = I_d(\Gamma(u, v)) \tag{3}$$

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 $I_d(u', v') \simeq I_d(\lfloor u' \rceil, \lfloor v' \rceil)$, where $\lfloor x \rceil$ 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 *reshape* function might be handy). The resulting code will be faster.

Note Matlab has a *imwarp* function dedicated to this kind of operations.

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

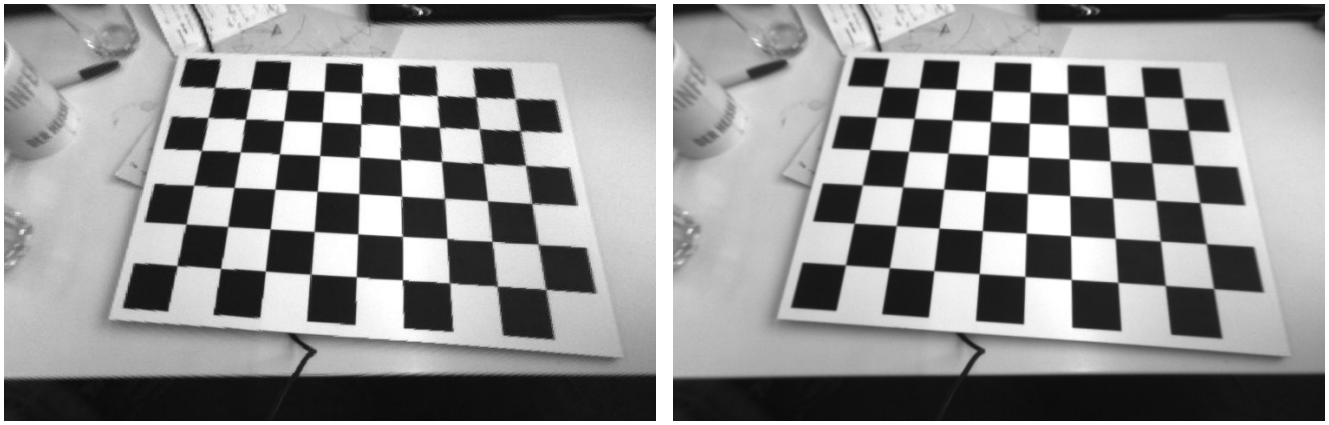

Figure 7: Undistorted images. Left: nearest-neighbor interpolation (observe the artifacts on the edges). Right: bilinear interpolation.