Exercise 6

Camera model and calibration

IAML 2020

Purpose

These exercises aim to introduce you to the pinhole model and calibration of pinhole cameras. The goal of camera calibration is to find the parameters of the camera matrix P e.g. that projects points $(X, Y, Z)^\top$ (in world coordinates) onto the 2D points (x, y) in the image such that $x \sim PX$ ($\sim$ means "given up to a scalar multiplication"). In these exercises you will:

- Load and detect calibration patterns.
- Use camera calibration to measure intrinsic and extrinsic parameters.
- Measure and perform correction of lens correction.
- Apply camera calibration matrix for measuring distances and drawing tree-dimensional objects on the images calibration parameters.

This week also contains **extra** exercises for those who would like get more insight into camera calibration and its application to augmented reality. They are guided exercises but will take some hours to complete – but no need to stress if you cannot make them.

Notes

- All exercises are non-mandatory and we have purposefully added (hopefully) enough exercises to keep everyone busy. We mark less important exercises with ^{extra}. If you don't have time to do all exercises, wait until we release the solutions and use them as a guide.
- In the exercises based on script (i.e. `.py`) files, you are expected to implement code at certain points in the file. Although it is typically clear from context which piece of code you have to change, we have included specifier comments of the type `# <Exercise x.x (n)>`, where `x.x` is the exercise number and `(n)` is the letter associated with the specific task.
- Exercises marked by ^{extra}should only be attempted after completing the other exercises. The mark is added to indicate that the exercise is not essential but still important.

Exercise 6.1

Camera basics

These exercises are intended for illustrating aspects of the the pinhole camera model e.g. the intrinsic and extrinsic parameters of the camera.

1. Let

$$P = A[R|t], \qquad K = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad t = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{1}$$

   What is the effect of a applying the projection matrix P on a 3D point p?

2. Now set R to

$$R = \begin{bmatrix} 0.9397 & 0.3420 & 0 \\ -0.3420 & 0.9397 & 0 \\ 0 & 0 & 1.000 \end{bmatrix}$$

3. Project the points p_i using the new P. You may choose p_i to be the vertices of a cube.

   (a) What effect did it have?

   (b) Explain this by the appearance of R.

4. Now change the translation to

$$t = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$$

   Use the projection with the new P. What is the effect? Explain this by the appearance of $R|t$.

5. **Intrinsic parameters:** The intrinsic parameters is a model of the camera optics. Recall the simplified definition:

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

   where f is the focal length, and c_x, c_y is the image plane centre.

   Use the identity parameters from the first exercise (1) to experiment with the effect of f on the projection of points. There is a concept from transformations (e.g. rotation, translation, scaling, skew, etc.) that is closely related to the way f is used here. Can you see which?

Hint: *If you experiment with the effect of using different values of f for a single point, the solution should become clear.*

Exercise 6.2

Camera Calibration

In this exercise, you will capture images of planes for use in camera calibration. To achieve stable calibration results, you will need around ten images. Use the script `pattern_detector.py` to perform this task.

1. **Choose the calibration pattern:** The file `pattern_detector.py` attempts to find feature points in a calibration pattern directly from the camera or video. These points are used to make the relation between world coordinates and image coordinates. The feature points (e.g. centers of circles or corner points) are relatively easy to detect automatically. Figure 1 shows two calibration patterns overlayed with the detected features.

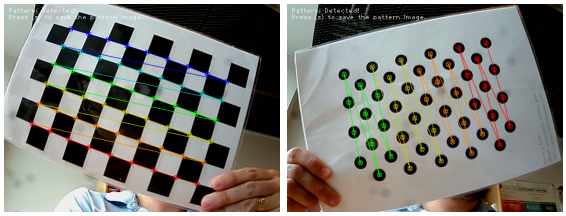

Figure 1: Detected calibration patterns.

2. **Save a set of patterns:** Run the script `pattern_detector.py` and place the calibration pattern in front of the camera. Detected features will be drawn as depicted in Figure 1. Press the `S` key to save the current frame with the detected calibration pattern. Figure 2 shows an example of a set of ten calibration patterns.

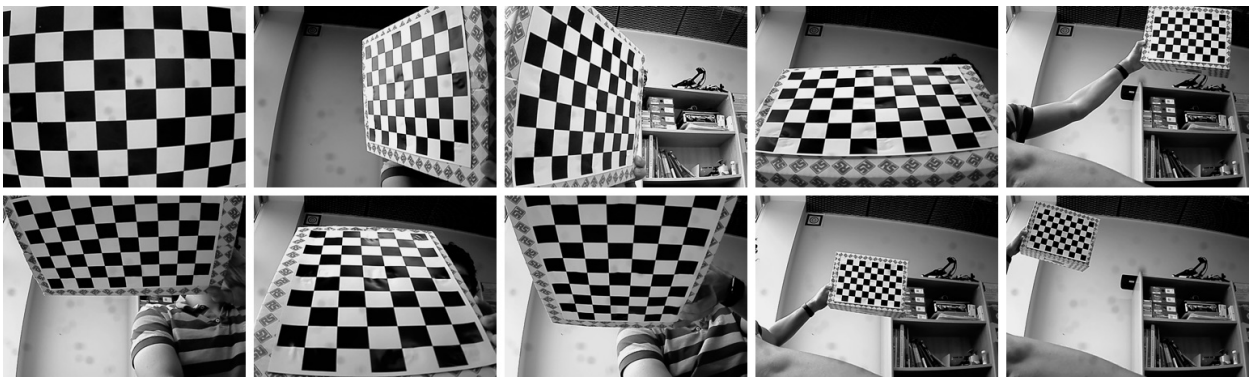

Figure 2: The calibration pattern at 10 different poses.

3. **Check the saved patterns:** The script saves the images of the calibration pattern as `Pattern_XY.png` in the folder `outputs`. Make sure that you **take at least 10 images** of the calibration pattern in different orientation and distances and and saved them in the folder `outputs` folder before starting the next exercise.

Exercise 6.3

Undistorting images

The purpose of this exercise is to gain some experience in calibrating a camera using OpenCV; that is calculating the intrinsic and extrinsic parameters of the camera, and export these parameters to be used in other applications that require calibrated cameras. Use the script `camera_calibration.py` to answer this exercise.

1. **Extract pattern corners** Read and understand the function `detect_pattern_image()` detects the calibration patterns in the current image and return two vectors with the (1) detected coordinates in the image (`corners`) and (2) the corresponding world coordinates (`pattern_points`). The function uses:
   - `cv2.findChessboardCorners()` [1] to find the chessboard pattern in the image and return the coordinates of pattern corners in the image;
   - `cv2.cornerSubPix()` [2] to improve the chessboard corners detection in the input image; and
   - Saves in the detected pattern as `Pattern_XY_Chessboard.png` or `Pattern_XY_Circles.png` in the `output` folder.
2. For each pattern image in `filenames` variable, append the returned values to the end of the vectors `imagePoints` and `objectPoints` .
3. **Calibrate the camera:** Use the function `cv2.calibrateCamera()` [3] to estimate the intrinsic and extrinsic parameters of your camera. Make sure to use several views (orientation and depth) of the calibration pattern. The function returns the following values:
   - `rms` : An output vector of the RMS re-prejection error estimated for each pattern;
   - `cameraMatrix` : A 3×3 camera matrix K;
   - `distCoeffs` : A 1×5 vector with the distortion coefficients of camera lens;
   - `rvecs` : A vector with all 3×1 rotation matrices for each pattern view; and
   - `tvecs` : A vector with all 3×1 translation matrices for each pattern view.
4. **Export the calibration data:** Use the function `np.save()` to export all calibration data in the following Numpy files: (i) `rms.npy` ; (ii) `cameraMatrix.npy` ; (iii) `distCoeffs.npy` ; (iv) `rvecs.npy` ; and (v) `tvecs.npy` . *Remember to save the Numpy files in the `outputs` folder.*
5. **Undistort images:** Your images are likely affected by lens distortion. In this exercise you have to remove the lens distortion of each image you just saved:

[1] An OpenCV function to find the positions of internal corners of the chessboard.

[2] An OpenCV function to refine the corner locations.

[3] An OpenCV function to calibrate a pinhole web camera.

(a) Use the function `cv2.getOptimalNewCameraMatrix()` [4] to return a new camera matrix based after lens distortions have been removed.

(b) Use the function `cv2.undistort()` [5] correct the image from lens distortion using `newCameraMatrix` as input.

(c) Save the undistorted image in the `output` folder with an unique name.

6. Check the outputs produced. Make sure that all patterns are detected and that the undistorted images seem reasonable.

Figure 3 shows the final results of undistorting calibration pattern images.

In your dataset, evaluate the lines of the calibration pattern and the environment in both original and undistorted images.

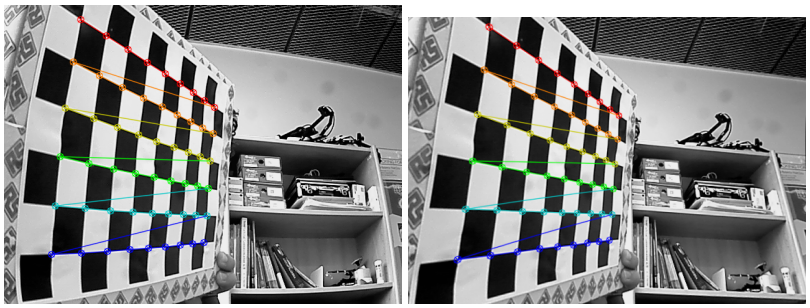

Figure 3: (*left*) the original pattern image used to calibrate the camera; (*right*) the undistorted image.

[4] An OpenCV function to return a new camera matrix, K, after removing lens distortion

[5] An OpenCV function to correct for lens distortion.

Exercise 6.4

Measuring distances^{extra}

In this task, you will use the intrinsic and extrinsic parameters of the camera to calculate the distance from the camera to the calibration pattern. Use the script `measure_distances.py` to answer this exercise.

Recall the projection relations from the lectures.

$$x = \frac{fX}{Z}, x = \frac{fY}{Z}$$

1. Assume the area of a rectangle in world coordinates is $\mathbf{A} = X * Y$, show that the apparent area, $\mathbf{a}$, of the rectangle in the image can be approximated with

   $$\mathbf{a} = \frac{f^2\mathbf{A}}{Z^2}$$

   As the area of each square in the calibration pattern is fixed (e.g. 3.0 centimeters) and the intrinsic parameters of the camera, f, have been calibrated then it is possible to find an approximation to the distance between the calibration pattern and the camera.

2. **Import the calibration data:** Use the function `np.load()` to load the intrinsic parameters of the camera that you have saved during the camera calibration. Load the camera matrix and the distortion coefficients into the variables `cameraMatrix` and `distCoeffs`.

3. **Detect the calibration pattern:** Use the function `cv2.findChessboardCorners()` to find the chessboard pattern.

4. Use the equation found in question 1 to estimate the depth of the calibration pattern in each image.

5. There are several ways to estimate the depth. A simple solution is to pick a single rectangle but more advanced methods would estimate the depth based on multiple rectangles. You can chose to investigate any of the following questions

   (a) Is there a difference between which rectangle is used e.g. in the center or on the boundary

   (b) How does the number of rectangles influence the accuracy of the estimates

   (c) When using several rectangles is there a difference between using the mean and median.

   (d) **Write the distance in the image:** Use the function `cv2.putText()`[6] to write the distance measurement in the current video frame. Remember to specify if the distance is in centimeter or millimeter.

[6] An OpenCV function to draw a text string.

Figure 4 shows an example of three frames from the distance measurement algorithm. It shows the calibration pattern at 27.57 cm, 101.48 cm and 147.89 cm from the calibrated camera. You can find a video with the result of the distance measurement using Python and OpenCV on https://youtu.be/LxCMzDMkYCM.

Figure 4: (*left*) the calibration pattern at 27.57 cm; (*center*) the calibration pattern at 101.48 cm; and (*right*) the calibration pattern at 147.89 cm.

6. An alternative solution (and a bit overkill) is to use the function `cv2.solvePnP()` [7] to find the calibration pattern pose from 3D-2D point correspondences. This function returns the three-dimensional rotation and translation matrices from the current pattern pose in the image. The translation vector contains sufficient information to estimate the distance.

[7] An OpenCV function to find an object pose from 3D-2D point correspondences.

Exercise 6.5

Augmented reality^{extra}

In this task, you will use the intrinsic and extrinsic parameters of the camera to draw three-dimensional objects over the calibration pattern. Given the calibration pattern, you can calculate its pose through the rotation and translation matrices. With a known object pose and the camera calibration parameters it is possible to augment the objects with 3D figures. Use the script `augmented_cube.py` to answer this exercise. Most of the structure of grabbing images, defining the cube in world coordinates etc are already given in `augmented_cube.py`.

(a) **Import the calibration data:** Use the function `np.load()` to load the intrinsic parameters of the camera that you have saved during the camera calibration. Load the camera matrix and the distortion coefficients in the variables `cameraMatrix` and `distCoeffs`.

(b) **Create the vector with pattern coordinates in the world coordinates:** Use the function `createPatternVectors()` to create a vector of calibration pattern points in image coordinates. This function returns a vector with the homogeneous coordinates of the pattern corners and the corners distribution.

(c) The next step is to project the 3D points of the cube onto the image plane e.g.

$$P = K\,[R|t] \tag{2}$$

Use the function `cv2.projectPoints()` [8] to project the 3D vertices of the cube to the image.

[8] An OpenCV function to project 3D points to an image plane.

Notice `cv2.projectPoints()` performs not only a rotation and translation but also an inverse z-distance scaling. The assumption is that the camera is located at z=0 and pointed along the z-axis.

(d) *Extra* Implement your own projection by using the estimated rotation and translation matrices and the calibration matrix to project the cube onto the image using equation 2

such that the projection of $x \sim PX$ is in camera coordinates. It is important here to ensure that the rotation and translation matrices are in the right

(e) **Draw the augmented objects:** Use the functions `draw3DAxes()` and `draw3DCube()` to draw the augmented objects on the calibration pattern. Figure 5 shows an example of two augmented cubes drawn in different poses.

(f) **Undistorted the images:** Use the intrinsic parameters of the camera to undistorted the captured frame, before you draw the augmented object on it.

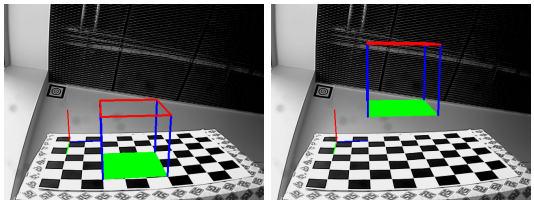

Figure 5: (*left*) the augmented cube on the calibration pattern; and (*right*) the augmented cube 9 centimeters over the calibration pattern.

(g) **Rotate the cube:** Make the cube rotate around its own axis, independently of the actual object in the scene.

(h) **Draw different objects:**$^{(Extra)}$ Try to draw other augmented object, e.g. a pyramid.

You can find a video with the result of drawing augmented objects using Python and OpenCV on https://youtu.be/bb4Jby2Kunc.