Computer Vision (LE48) – Exercise 1

Jan Ondras (jo356)
Trinity College

November 1, 2017

1 Camera calibration

Intrinsic camera parameters are specific to the camera used. They include information such as focal length f, scaling factors k_u, k_v and optical centers c_u, c_v which are all stored in the camera calibration matrix K. Here, (u, v) refer to pixel coordinates in the image plane and the ratio k_v/k_u is called the aspect ratio.

$$K = \begin{bmatrix} fk_u & 0 & c_u \\ 0 & fk_v & c_v \\ 0 & 0 & 1 \end{bmatrix}$$

Namely, I obtained the following parameters:

$$fk_u = 698.53 \qquad fk_v = 698.67 \qquad c_u = 417.76 \qquad c_v = 314.59$$

Distortion coefficients are used to correct tangential distortion (caused by camera lenses not being perfectly parallel to the imaging plane) and radial distortion ("fish-eye" effect) of the image. The following distortion coefficients were determined during the calibration:

$$dist_c = \begin{bmatrix} 3.31 \times 10^{-1} & -1.92 & 7.74 \times 10^{-4} & 1.15 \times 10^{-3} & 3.41 \end{bmatrix}$$

Extrinsic camera parameters specify the transformation (rotation and translation) between camera-centered and object-centered world coordinates. They are also stored in the matrix form:

$$P_{RT} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix}$$

For instance, for one of the images used for calibration the following values were calculated

$$\boldsymbol{r} = \begin{bmatrix} 0.36 & -0.76 & -0.11 \end{bmatrix}^T, \qquad \begin{bmatrix} t_x & t_y & t_z \end{bmatrix} = \begin{bmatrix} -3.23 & -1.56 & 10.14 \end{bmatrix}$$

where $\boldsymbol{r}$ is the Rodrigues' rotation vector. The conversion between the vector $\boldsymbol{r}$ and the rotation matrix with entires $r_{\{1,2,3\}\{1,2,3\}}$ can be done using the method `cv2.Rodrigues`.

The complete transformation between object-centered world coordinates (X, Y, Z) and pixel coordinates (u, v) can then be expressed as matrix operation (leveraging homogeneous coordinates with a scale parameter s):

$$\begin{bmatrix} su \\ sv \\ s \end{bmatrix} = KP_{RT} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

To calibrate my camera using OpenCV I first captured 60 images of 10×7 checkerboard having 9×6 internal corners. I also prepared a mesh of points representing the checkerboard pattern in object-centered coordinates. Then, for each image I obtained image-space coordinates of inner corners in the

checkerboard pattern using the method `cv2.findChessboardCorners` and to improve the accuracy of the detected corners I used the method `cv2.cornerSubPix`. An example of the 9×6 pattern of inner chessboard corners (detected in one of the original distorted images) is shown in Fig. 1.

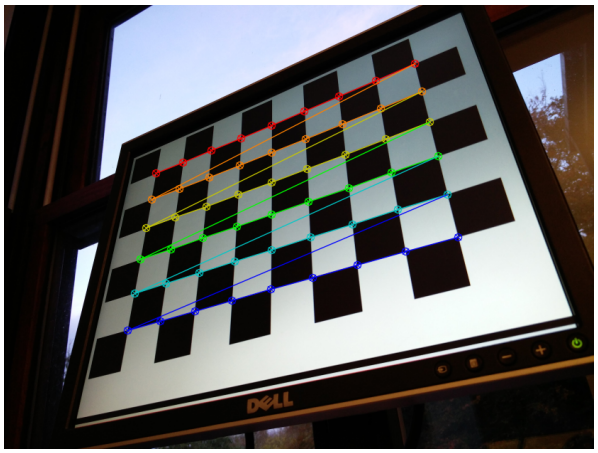

Figure 1: Camera calibration: 9×6 pattern of detected inner chessboard corners.

After all available images were processed, I called the method `cv2.calibrateCamera` which inputs the detected corners from all the viewpoints along with the object-centered pattern points and returns the camera matrix, distortion coefficients, rotation and translation vectors minimising the re-projection error. The re-projection error (the total sum of squared distances between observed feature points and projected object points) is minimised by Levenberg-Marquardt algorithm that is usually used to solve non-linear least squares problems. The obtained parameters (camera matrix and distortion coefficients) were then saved to `.npz` file for later use.

Finally, the camera matrix and distortion coefficients were used to remove image distortion by calling the method `cv2.undistort` on one of the images. An alternative approach is to call the methods `cv2.initUndistortRectifyMap` (finds a mapping function from distorted image to undistorted image) and then `cv2.remap` (applies this mapping function to get undistorted image). This is more efficient if we need to remove distortion in multiple images (captured by the same camera) because `cv2.initUndistortRectifyMap` is called only once for a given set of images as opposed to the case of `cv2.undistort` that would result in a call to `cv2.initUndistortRectifyMap` for every image.

An example of an original (distorted) and undistorted image is shown in Fig. 2.

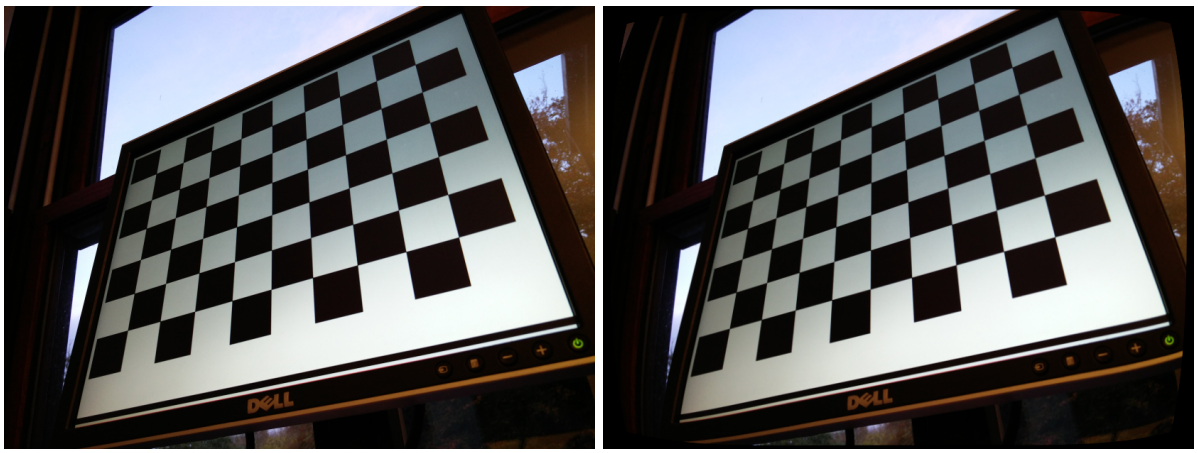

Figure 2: Camera calibration: a sample distorted (left) and undistorted (right) image.

2 Perspective correction

2.1 Overview

In general, to carry out the perspective correction we need to detect at least six unique points. However, in our case we assume that the stamp is a planar object and so only four points are needed, namely, four outer corners of the stamp. Given a set of rubber stamp photos I first detected edges. Then I fit closed contours onto them and used an appropriate heuristic to choose the contour that captures the outer outline of the stamp. Approximating the contour by a quadrilateral I obtained coordinates of the four corners. These were then used to calculate the perspective transform matrix that was in turn applied to the region of the image bounded by the corners.

More detailed description of OpenCV methods I employed follows.

2.2 Detailed description

Firstly, I tried to remove the camera distortion in every image using the camera parameters obtained in the previous section.

```
14 gray = cv2.GaussianBlur(gray, (13, 13), 0)
```

I converted the input image to grayscale image (gray) and blurred it using a Gaussian filter with kernel of size 13×13 and standard deviations determined automatically by the kernel size, namely, $0.3 \times ((13-1) \times 0.5 - 1) + 0.8 = 2.3$, same for both directions. Choice of appropriate kernel size allowed me to disregard thin edges around the stamp which was essential for later processing steps.

```
15 edges = cv2.Canny(gray, 75, 200)
```

To detect edges I used the Canny edge detection algorithm. It consists of the following steps: noise reduction, finding intensity gradient of the image, non-maximum suppression, and hysteresis thresholding. I had to tune the second and third arguments that specify threshold values $minV$ and $maxV$ respectively. Edges above the $maxV$ are considered as sure-edges and those below $minV$ as non-edges. Edges between the two thresholds are classified based on their connectivity: they are not discarded only if they are connected to sure-edges.

```
21 contours, _ = cv2.findContours(edges.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
22 contours = sorted(contours, key=cv2.contourArea, reverse=True)[:5]
```

The heuristic I used is the following: assume that the largest contour in the image with exactly four points is outer border of the stamp. Therefore, I first detected all the contours by the above method where the contour retrieval mode is specified by the second argument (list in this case) and the third argument determines the contour approximation method (CHAIN_APPROX_SIMPLE removes all redundant contour points which allows contour compression and memory savings). Consequently, I sorted the contours by area they enclose in descending order and kept only five largest ones.

```
27 epsilon = 0.05 * cv2.arcLength(c, True)
28 c_approx = cv2.approxPolyDP(c, epsilon, True)
```

Next, I iterated over the contours and for each I determined its length that was used to choose the tuning parameter epsilon. This parameter determines the maximum distance between contour and its approximation, i.e. it specifies the precision of contour approximation. The approximation algorithm (by Ramer–Douglas–Peucker) takes a contour consisting of line segments and finds a similar one (based on the epsilon) but with fewer points. Last argument in both of the above methods specifies whether the contour is closed or not. The first approximating contour with four points was then considered to be the approximation of the outer outline of the stamp and the coordinates of the four corners were saved.

It is important to note that I had to keep the consistent ordering of the corners (top-left, top-right, bottom-right, bottom-left) so that it matched the ordering of transformed ones.

```
Pt = cv2.getPerspectiveTransform(corners_detected, corners_transformed)
```

Having obtained the coordinates of corners in the input image and coordinates of the corresponding points in the output image I could calculate the 3×3 transformation matrix. This works only if any triple among the four points is not collinear which is assumed to be true in the case of quadrilateral shape of the stamp.

```
img_corr = cv2.warpPerspective(img_org, Pt, (new_width, new_height))
```

Finally, the transformation matrix was applied to the image and perspective corrected image of size $new\_width \times new\_height$ was obtained.

Examples of an edged image, image with fitted contours, and final perspective corrected image are shown in Fig. 3.

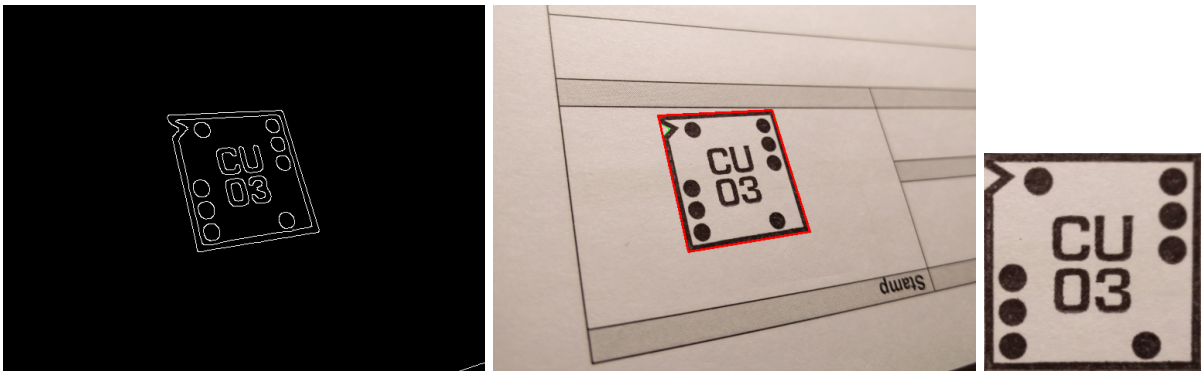

Figure 3: Perspective correction: a sample edged image (left), image with fitted contours (middle), and perspective corrected image (right). The red contour is a four-point approximation of the green contour.

2.3 Perspective correction with three points

Under the assumption that the depth of the object in the scene is small compared to the distance of the camera from the scene, i.e. weak perspective projection is assumed, only three points are needed to carry out the perspective correction. In this case the projection matrix for viewing a plane would be

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ 0 & 0 & p_{33} \end{bmatrix} \text{ instead of } \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}$$

where the number of non-zero entries (p_{ij}) is equal to the number of degrees of freedom plus one, since the overall scale of the projection matrix does not matter and we can set e.g. $p_{33} = 1$.

2.4 Utilizing camera parameters

Initially, I tried to use the intrinsic camera parameters to undistort every image before carrying out the perspective correction. However, it seemed that it did not improve the results much. This is probably caused by the fact that most modern cameras try to remove the distortion automatically.

If we knew the extrinsic camera parameters, then the perspective correction would be much more simple. We could directly use these parameters to perform the desired perspective correction without the need to detect the corners.

Appendix

Camera calibration code

```
import numpy as np
import cv2
import glob

images = glob.glob('./imgCalib/*.jpg')
image_to_undistort = './imgCalib/IMG_20171028_174106.jpg'

square_size = 1.0
pattern_size = (9, 6)

# Prepare pattern points in model world coord. (z=0): (0,0,0), (1,0,0), ... (1,2,0) ...
pattern_points = np.zeros( (np.prod(pattern_size), 3), np.float32)
pattern_points[:,:2] = np.indices(pattern_size).T.reshape(-1, 2)
pattern_points *= square_size

# Arrays to store object points and image points from all images
obj_points = [] # 3D points in real world space
img_points = [] # 2D points in image plane

for image_name in images:
    img = cv2.imread(image_name)
    img = cv2.resize(img, None, fx=0.2, fy=0.2, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # Find chessboard corners
    found, corners = cv2.findChessboardCorners(gray, pattern_size, None)

    # If found, add object points, image points (after refining them)
    if found == True:
        obj_points.append(pattern_points)

        # Refine corners (search window = 23x23)
        cv2.cornerSubPix(gray, corners, (11,11), (-1,-1),
                         (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001))
        img_points.append(corners)

        # Draw the corners
        cv2.drawChessboardCorners(img, pattern_size, corners, found)
        cv2.imshow('Chessboard corners: ' + image_name, img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
    else:
        print 'Chessboard not found! ', image_name
    continue

# Calibrate camera to get camera matrix and distortion coefficients
rms, camera_matrix, dist_coefs, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points,
                                                                   gray.shape[::-1], None, None)
print "RMS:", rms
print "Camera matrix:\n", camera_matrix
print "Rotation vectors:\n", rvecs
print "Translation vectors:\n", tvecs
print "Distortion coefficients:\n", dist_coefs.ravel()

# Save the camera matrix and distortion coefficients for future use
np.savez('./camera_config.npz', camera_matrix=camera_matrix, dist_coefs=dist_coefs)

# Distortion removal
img2 = cv2.imread(image_to_undistort)
img2 = cv2.resize(img2, None, fx=0.2, fy=0.2, interpolation=cv2.INTER_AREA)
res = cv2.undistort(img2, camera_matrix, dist_coefs)
cv2.imshow('Distorted', img2)
cv2.imshow('Undistorted', res)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Perspective correction code

```python
import numpy as np
import cv2
import glob

images = glob.glob('./img/*.jpg')
new_width = new_height = 250

for img_name in images:
    img_org = cv2.imread(img_name)
    img_org = cv2.resize(img_org, None, fx=0.2, fy=0.2, interpolation=cv2.INTER_AREA)
    img_org_show = img_org.copy()

    gray = cv2.cvtColor(img_org, cv2.COLOR_RGB2GRAY)
    gray = cv2.GaussianBlur(gray, (13, 13), 0)
    edges = cv2.Canny(gray, 75, 200)
    cv2.imshow("Edges", edges)

    # Heuristic: assume that the largest contour in the image with exactly four points is the outer border
    # of the stamp. Find contours in the edged image, sort them by area enclosed by the contour, keep only
    # few largest ones. "CHAIN_APPROX_SIMPLE" removes all redundant contour points => contour is compressed
    contours, _ = cv2.findContours(edges.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    contours = sorted(contours, key=cv2.contourArea, reverse=True)[:5]

    # Iterate over contours
    for c in contours:
        # Approximate the contour (tuning epsilon)
        epsilon = 0.05 * cv2.arcLength(c, True)
        c_approx = cv2.approxPolyDP(c, epsilon, True)

        # If approximating contour has 4 points, then assume that outer border of the stamp was detected
        if len(c_approx) == 4:
            # Show non-approximating contour (green)
            cv2.drawContours(img_org_show, [c], -1, (0, 255, 0), 1)
            # Show approximating contour (red)
            cv2.drawContours(img_org_show, [c_approx], -1, (0, 0, 255), 2)
            stamp_contour = c_approx.reshape(4, 2)
            break

    # To keep consistent ordering of corners: top-left, top-right, bottom-right, bottom-left
    corners_detected = np.zeros((4, 2), dtype = "float32")
    sum_xy = stamp_contour.sum(axis=1)
    corners_detected[0] = stamp_contour[np.argmin(sum_xy)]  # top-left
    corners_detected[2] = stamp_contour[np.argmax(sum_xy)]  # bottom-right
    sub_xy = np.diff(stamp_contour, axis = 1)
    corners_detected[1] = stamp_contour[np.argmin(sub_xy)]  # top-right
    corners_detected[3] = stamp_contour[np.argmax(sub_xy)]  # bottom-left

    # Desired image-plane points
    corners_transformed = np.array([
        [0, 0],
        [new_width-1, 0],
        [new_width-1, new_height-1],
        [0,  new_height-1]], dtype='float32')

    # Obtain and apply perspective transform matrix Pt
    Pt = cv2.getPerspectiveTransform(corners_detected, corners_transformed)
    img_corr = cv2.warpPerspective(img_org, Pt, (new_width, new_height))

    # Show original and perspective corrected images
    cv2.imshow("Original image", img_org_show)
    cv2.imshow("Perspective corrected image", img_corr)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```