Camera Calibration using Zhang's method

Francesco Trevisan

May 27, 2024

1 Problem statement

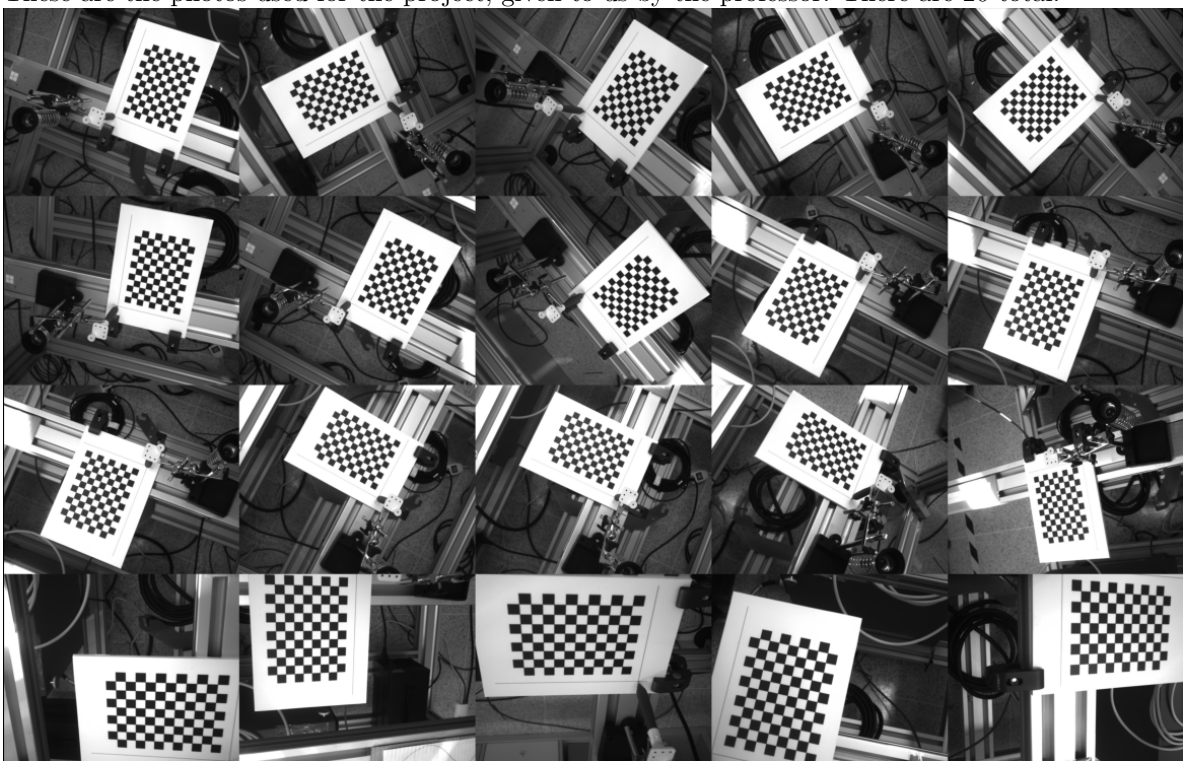
This project required to calibrate a camera using a series of photos of checkerboards utilizing Zhang's method. This included:

1. enstablishing correspondences between the real world and the image plane;
2. utilizing the correspondences found in the previous step, calculate the homographies of each photo;
3. calibrate the camera (find the intrinsic value K, then the extrinsic values R and t for each photo);
4. once K, R and t are found, calculate the total reprojection error on one of the images used;
5. then, superimpose an object on each photo used to calibrate the camera.

2 What was used

2.1 Calibration photos

These are the photos used for the project, given to us by the professor. There are 20 total.



2.2 Tools and Libraries

Everything was developed using Python with Visual Studio Code as an IDE and utilizing the following libraries:

- Numpy for easier matrix operations;
- OpenCV2 for image manipulation and most importantly for the findChessboardCorners() function;
- Pyplot for showing the images.

3 Camera Calibration

3.1 Enstalishing Correspondences

We start by taking each photo we'll be using for the camera calibration and find the correspondences between the real world points and their image plane counterparts. The following code is the one used to load up the images into an array and then utilize the cv2.findChessboardCorners() function to extract the coordinates of the corners in the photo.

```
folderpath= 'images/'
images_path = [os.path.join(folderpath, imagename) for imagename
in os.listdir(folderpath) if imagename.endswith(".tiff")]
images_path.sort()
grid_size = (8,11)
homographies = []
for p in images_path:
    corners = []
    im = cv2.imread(p)
    return_value, corners =
    cv2.findChessboardCorners(im, patternSize=grid_size, corners=None)
    if not return_value:
        print(f"pattern not found for image {p}")
    corners=corners.reshape((88,2)).copy()
    real_coordinates = findRealCoordinates(corners)
    H = estimateHomography(corners, real_coordinates)
    homographies.append(H)
```

The function estimateHomography in the code is the one used for the next step.

3.2 Enstablishing Homographies

The function enstablishingHomographies calculates the homography of each calibration image based on the pixel coordinates of each corner and their real world coordinates(we find the real world coordinates by utilizing a findRealCoordinates() function). We find the homographies by solving the Ax=0 equation using the decomposition of the A matrix into USV^T and taking the last column of V.

```
    def estimateHomography(corners, real_coordinates):
    A = np.empty((0,9), dtype=float)

    for index, corner in enumerate(corners):
        Xpixel = corners[index, 0]
        Ypixel = corners[index, 1]
        Xmm = real_coordinates[index, 0]
        Ymm = real_coordinates[index, 1]

        m = np.array([Xmm, Ymm, 1]).reshape(1, 3)
        O = np.array([0, 0, 0]).reshape(1, 3)
```

```
        # Construct A
        A = np.vstack((A, np.hstack((m, 0, -Xpixel * m))))
        A = np.vstack((A, np.hstack((0, m, -Ypixel * m))))

    U, S, Vtransposed = np.linalg.svd(A) #decomposing A into  U,S and V transposed
    h =Vtransposed.transpose()[:,-1] #we need the last column of V
    H = h.reshape(3,3) #reshaping h into a 3x3 matrix nets us our homography
    return H
```

3.3 Application of Zhang's method

3.3.1 Finding K

Once we have our homographies, we utilize them to calibrate the camera to find the value of the camera's calibration matrix K, plus the R and t values of each calibration image. We start by constructing our V and b matrixes and solving the $Vb = 0$ equation. V is 2n x 6, where n is the number of found homographies. So for each image, we have 2 equations: $V = \begin{bmatrix} v_{12}^T \\ (v_{11}^T - v_{22}^T) \end{bmatrix}$ with $v_{ij} = \begin{bmatrix} H_{1i}H_{1j} \\ H_{1i}H_{2j} + H_{2i}H_{1j} \\ H_{2i}H_{2j} \\ H_{3i}H_{1j} + H_{1i}H_{3j} \\ H_{3i}H_{2j} + H_{2i}H_{3j} \\ H_{3i}H_{3j} \end{bmatrix}$.

The goal of this step is to find b, which is composed of 6 values that define the 3x3 symmetric matrix B, which we'll need to find our calibration matrix.
Once we compose V, we can find the values of b by decomposing V into $U\Sigma V^T$ and taking the value of the last column of V. The following code is the one that utilizes the found homographies to calculate the matrix B. findVijValue() calculates the value of v_{ij} given the current homography, i and j as parameters.

```
V = []
for h in homographies:
    v = []
    v.append(findVijValue(h,0,1))
    v11= findVijValue(h,0,0)
    v22= findVijValue(h,1,1)
    v.append(np.subtract(v11,v22))
    V.append(v)
V = np.array(V)
V = V.reshape(40,6)
U,E,Stransposed = np.linalg.svd(V)
S = Stransposed.transpose()
b = S[:,-1]
B = np.empty((3,3))
B[0]= [b[0],b[1],b[3]]
B[1]= [b[1],b[2],b[4]]
B[2]= [b[3],b[4],b[5]]
```

Since $B = (KK^T)^{-1}$, we can utilize the Cholesky factorization to decompose B into LL^T, with L lower triangular. Since $K = (L^T)^{-1}$, by imposing $K_{33} = 1$ for proper scale, we found our calibration matrix. The following code is the one used for this step.

```
if not is_pos_def(B):
    B = -1 * B
#this step is to prevent the matrix not being positive definite, thus allowing us to
```

```
#use the cholesky factoriazion
L= np.linalg.cholesky(B)
Lt = L.transpose()
K = np.linalg.inv(Lt)
K[2][2] = 1
```

These are the values of K that i found:

$$K = \begin{bmatrix} 1.93290349 \times 10^3 & -1.91486347 & 6.83169960x10 \times 2 \\ 0 & 1.92982446 \times 10^3 & 5.48764193 \times 10^2 \\ 0 & 0 & 1 \end{bmatrix}$$

3.3.2 Finding R and t

Once we have K, we can utilize it and the homographies we found to calculate R and t of each of the photos:

$$r1 = \lambda K^{-1} h_1, \quad r2 = \lambda K^{-1} h_2, \quad t = \lambda K^{-1} h_3, \quad r3 = r1 \times r2$$

with

$$\lambda = \frac{1}{||K^{-1} h_1||}$$

The following code calculates R and t and puts them into an array.

```
rtMatrix = []

for h in homographies:

    h1 = np.array(h[:,0])
    h2 = np.array(h[:,1])
    h3 = np.array(h[:,2])
    denonim = np.linalg.norm(np.matmul(np.linalg.inv(K),h1.transpose()))
    lambd = 1/denonim
    r1 = lambd *(np.matmul(np.linalg.inv(K),h1.transpose()))
    r2 = lambd *(np.matmul(np.linalg.inv(K),h2.transpose()))
    r3 = np.cross(r1,r2)
    t = lambd *(np.linalg.inv(K)@h3)
    rtMatrix.append([r1,r2,r3,t])
rtMatrix= np.array(rtMatrix)
```

3.4 Total reprojection error

I calculated the total reprojection error by taking the coordinates found by findChessboardCorners() ($[X_{corners}, Y_{corners}]$) and comparing them with the ones found by projecting the real world coordinates onto the image utilizing the projection matrix P i calculated ($[X_{proj}, Y_{proj}]$). Thus the error is calculated by

$$\varepsilon = (X_{proj} - X_{corners})^2 + (Y_{proj} - Y_{corners})^2$$

I used "image05.tiff" to calculate the reprojection error and the result was $\varepsilon = 61.031773081282154$; below is the code used, plus an image of the projected points with the ones found by the findChessboardCorners() function.

```
totRepErr= 0
immagine = cv2.imread(folderpath + "image05.tiff")
paramIntrinsechi = np.array(rtMatrix[5]).transpose()
P = K@paramIntrinsechi
return_value, puntipixel = cv2.findChessboardCorners(immagine, patternSize=grid_size, corners=Non
puntipixel= puntipixel.reshape((88,2)).copy()
coordReali =  findRealCoordinates(puntipixel)
for i, coord in enumerate(coordReali):
```

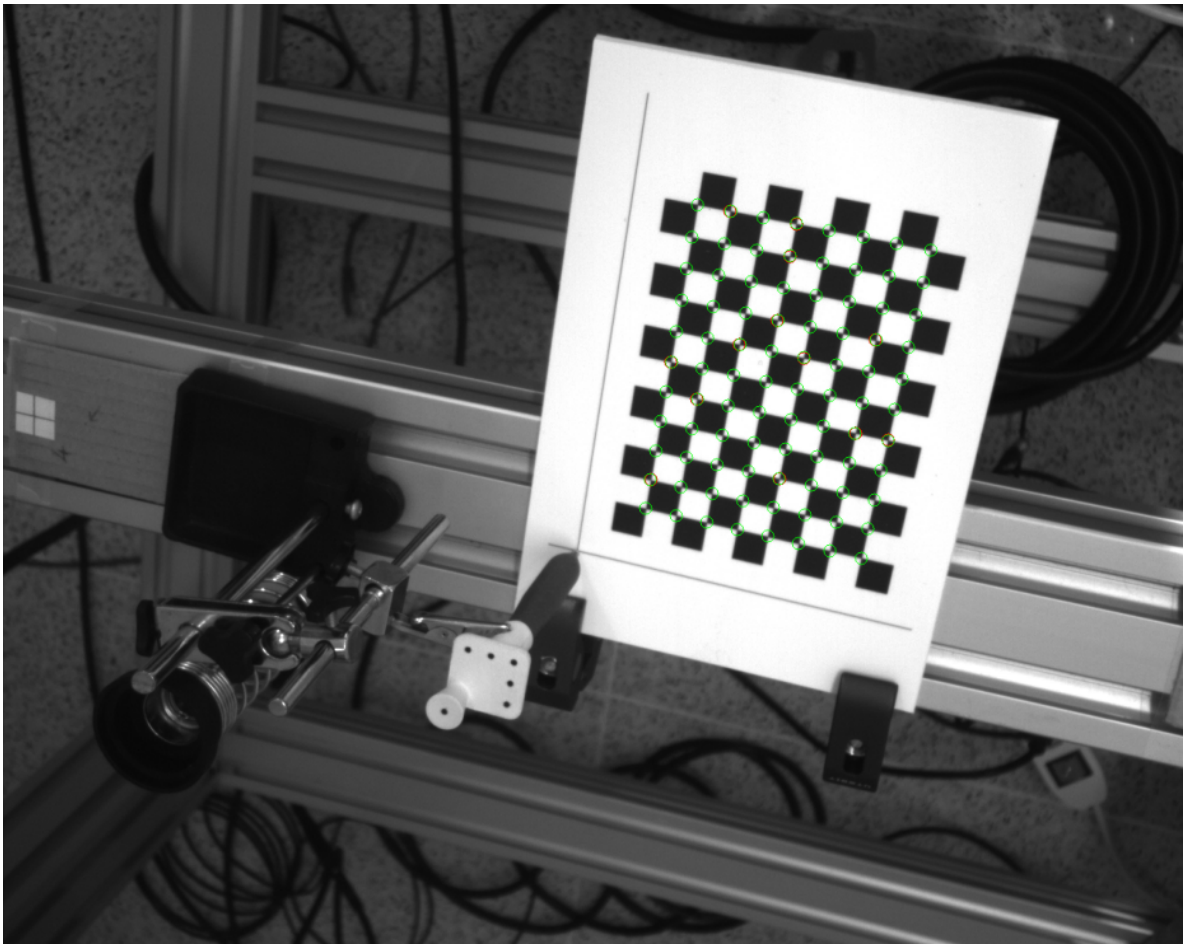


Figure 1: The red circles are the ones found by findChessboardCorners() overlapped by the green ones, found by the projections

```
homog = np.array([coord[0],coord[1],0,1])
project_hom = P@homog
projections = np.empty(2)
projections[0]= int(project_hom[0]/project_hom[2])
projections[1] = int(project_hom[1]/project_hom[2])
immagine = cv2.circle(immagine, (int(projections[0]),int(projections[1])), radius=5, color=(2
immagine = cv2.circle(immagine, (int(puntipixel[i][0]),int(puntipixel[i][1])), radius=5, colo
totRepErr += (projections[0]-puntipixel[i][0])**2 + (projections[1]-puntipixel[i][1])**2
```

3.5 Superimposing an object

Utilizing the found homographies, i superimposed a parallelepiped by projecting a rectangle on the checkboard and then projecting another one with an added height variable. Below the code used to do so, plus a collection of the images with the superimposed objects.

```
rect_height = 44

bottom_left_corner_x = -11
bottom_left_corner_y = 33
superimposed = []
for i,im in enumerate(images_path):
    #bottom game
```

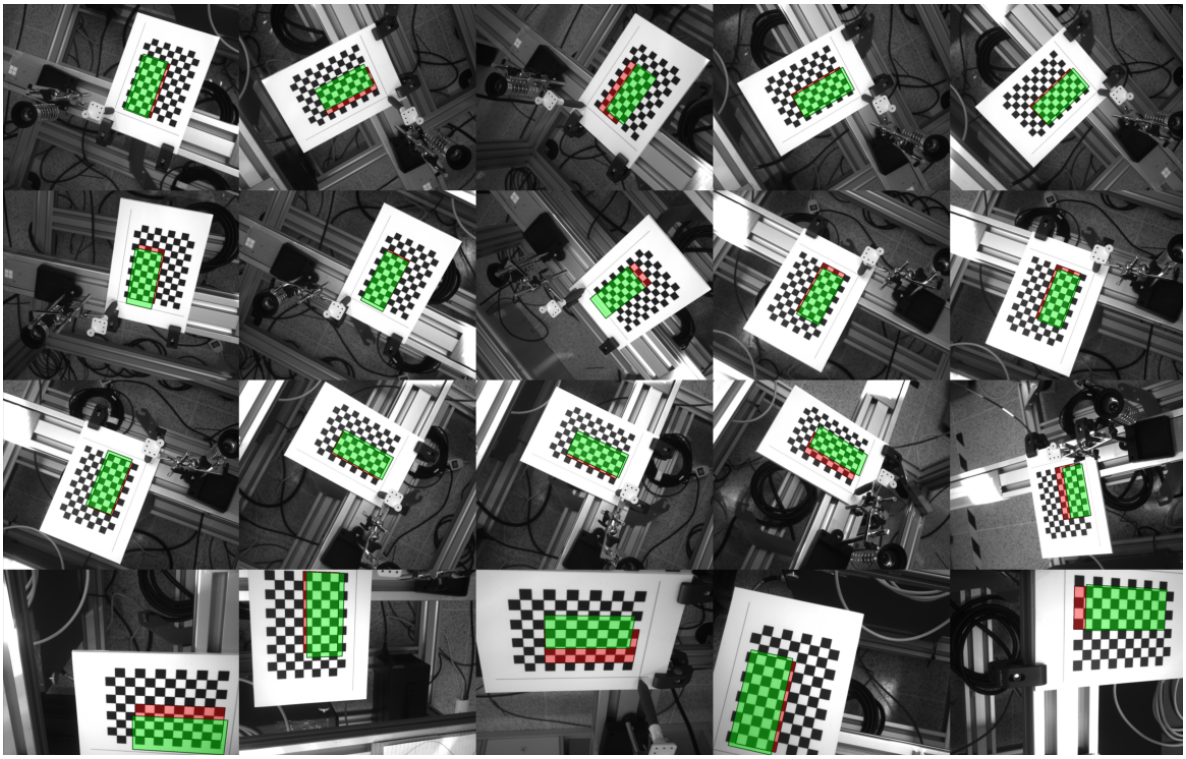


Figure 2: The parallelepiped superimposed on all the calibration images (red rectangle is the base and the green one is the top of the object

```
image = cv2.imread(im)
overlay = image.copy()
vx = bottom_left_corner_x + np.array([0, 0, rect_width, rect_width])
vy = bottom_left_corner_y + np.array([0, rect_height, rect_height, 0])
homogeneousIN = np.vstack((vx, vy, np.ones_like(vx)))
homogeneousOUT = homographies[i] @ homogeneousIN
xyOUT = (homogeneousOUT/homogeneousOUT[2])[:2]
cv2.polylines(image,np.int32([xyOUT.transpose()]), isClosed=True, color=(0,0,0),thickness=4)
cv2.fillPoly(overlay,np.int32([xyOUT.transpose()]),color=(255,0,0))
#top half
height = 25
homogeneousIN2 = np.vstack((vx, vy, np.full_like(vx,height),np.ones_like(vx)))
paramIntrinsechi = np.array(rtMatrix[i])
P = K@paramIntrinsechi.transpose()
homogeneousOUT2 = P @homogeneousIN2
xyzOUT = (homogeneousOUT2/homogeneousOUT2[2])[:2]
cv2.polylines(image,np.int32([xyzOUT.transpose()]), isClosed=True, color=(0,0,0),thickness=4)
cv2.fillPoly(overlay,np.int32([xyzOUT.transpose()]), color=(0,255,0))
image_new = cv2.addWeighted(overlay, 0.4, image, 1 - 0.4, 0)
superimposed.append(image_new)
```