TP CALIBRAGE

Camile Couturier
Paul Gueneau

Octobre 2020

1 Introduction

On cherche à calibrer la Webcam Logitech C270, on va utiliser pour cela une mire rerésentant un échiquier.

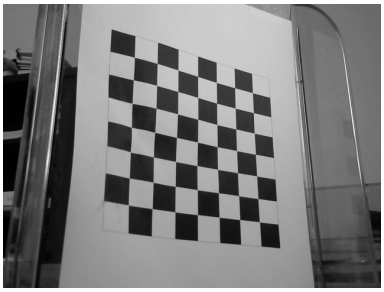

Figure 1: Mire échiquier

L'objectif sera d'avoir l'ensemble des points projetés sur les coins intérieurs de l'échiquier comme montré ci-dessous(figure 2). On saura alors si notre webcam est correctement calibrée ou non.

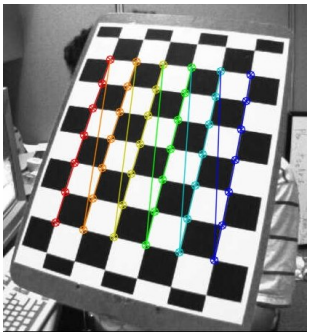

Figure 2: Résultat

2 Acquisition de la mire

Dans un premier temps, comme expliqué dans l'énoncé du TP on va détecter les coins intérieurs de la mire directement grâce à la fonction *cv2.findChessboardCorners* d'Open CV. L'échiquier étant de taille 8x8, on aura un tableau de coins de taille 7x7, en réalité on utilise deux mires, en effet la projection de l'espace 3D sur l'image 2D n'est pas bijective, deux points 3D peuvent se projeter sur un même point de l'image. En utilisant deux mires et en connaissant la distance entre les deux lors de l'acquisition, on peut déterminer l'intersection des droites projectives et ainsi déterminer l'unique antécédent du plan image dans l'espace 3D.

Voici le code en python :

Détection des coins

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
import math
import glob
images = glob.glob('*.png')


imnum=0
for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret,corners = cv2.findChessboardCorners(gray, (7,7), None)
```

3 Calibrage de la caméra avec la méthode de Tsai

Le problème de calibrage consiste en fait à détecter les coordonnées en pixels de l'image acquise dont les coordonnées dans l'espace 3D sont connues, on peut alors écrire u = Mx où M est la matrice de projection, u le vecteur des coordonnées pixels de l'image et x les coordonnées spatiales. La méthode de Tsai consiste à évaluer 12 paramètres que l'on va ensuite rentrer dans la matrice M en décomposant le problème en trois grandes étapes : - l'estimation d'un vecteur intermédiaire L par résolution du système U_1 = AL par calcul de la pseudo-inverse de A, en supposantque la matrice A est bien conditionnée. - le calcul analytique des coefficients de L afin d'obtenir l'ensemble des paramètres de translation et de rotation.

3.1 Construction des points dans le repère objet et dans le repère image

Tout d'abord nous commençons, à partir des valeurs de corners retournée par la fonction *cv2.findChessboardCorners*, par construire les points dans le repère

image, puis, à partir des données connues dans le repère objet (largeur d'un carreau de 20mm, distance relatives entre les mires pour les deux acquisitions de 120mm) nous construisons les points correspondant dans le repère objet. Le choix des axes du repère objet se fait à ce moment. Nous avons choisis l'origine du repère en haut à gauche de la mire, le vecteur ${e_o}^1$ comme allant de gauche à droite, le vecteur ${e_o}^2$ comme allant de haut en bas, et par construction, le vecteur ${e_o}^3$ comme "rentrant" dans la mire.

On fait ceci grâce au code suivant:

Construction des vecteurs image et objet

```
# Arrays to store object points and image points from all the images.
coord_mm = [] # 3d point in real world space
coord_px = [] # 2d points in image plane.
f = 4

i1 = 320
i2 = 240
images = glob.glob('*.png')
imnum=0
for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret,corners = cv2.findChessboardCorners(gray, (7,7), None)

    k=0
    for [[i,j]] in corners:
        i=int(i)
        j=int(j)
        x=20*(k//7)
        y=20*(k%7)
        coord_px.append([i-i1,j-i2])
        coord_mm.append([x,y,120*imnum])

        k+=1
    imnum+=1
```

On notera que puisque l'origine de notre repère objet se trouve projetée à gauche du centre de l'image dans notre repère image, cela implique que par la suite, notre paramètre ${o_c}^2$ soit négatif.

3.2 Construction de A et U_1

Grâce aux points $\tilde{u}$ et x construits précédemment, nous pouvons alors construire la matrice A et le vecteur U1 qui nous permettront par la résolution d'une équation linéaire, d'obtenir le vecteur L, qui nous permettra d'obtenir un certains nombre des paramètres du système projectif.

$$A = \begin{pmatrix} {\tilde{u_2}}^1 x_1^1 & {\tilde{u_2}}^1 x_2^1 & {\tilde{u_2}}^1 x_3^1 & {\tilde{u_2}}^1 & -{\tilde{u_1}}^1 x_1^1 & -{\tilde{u_1}}^1 x_2^1 & -{\tilde{u_1}}^1 x_3^1 \\ {\tilde{u_2}}^2 x_1^2 & {\tilde{u_2}}^2 x_1^2 & {\tilde{u_2}}^2 x_1^2 & {\tilde{u_2}}^2 & -{\tilde{u_1}}^2 x_1^2 & -{\tilde{u_1}}^2 x_1^2 & -{\tilde{u_1}}^2 x_1^2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ {\tilde{u_2}}^n x_1^n & {\tilde{u_2}}^n x_1^n & {\tilde{u_2}}^n x_1^n & {\tilde{u_2}}^n & -{\tilde{u_1}}^n x_1^n & -{\tilde{u_1}}^n x_1^n & -{\tilde{u_1}}^n x_1^n \end{pmatrix}$$

$$U = \begin{pmatrix} \tilde{u_1}^1 \\ \tilde{u_1}^2 \\ \vdots \\ \tilde{u_1}^n \end{pmatrix}$$

Nous allons simplément implémenter les formules du cours sur python, pour construire A et U1, et laisser la fonction linalg calculer L (voir *Annexe 1*).

```
L= [[-0.00197282]
 [ 0.00903351]
 [ 0.00431061]
 [ 1.70520102]
 [ 0.00572668]
 [ 0.00331257]
 [ 0.00685274]]
```

Figure 3: Matrice L

On obtient bien une matrice 7x1 comme attendu. Cette matrice va nous permettre d'évaluer certains paramètres nécessaires au calcul de M : f_2 et o_2^c

Grâce au vecteur L calculé à l'étape précédente on peut trouver de manière analytique neuf paramètres du système : les paramètres de translation o_1^c et o_2^c le paramètre β qui est égal au ratio Largeur/Hauteur des pixels du capteur ainsi que les six premiers paramètres de rotation du système projectif soit les deux premières lignes de la matrice R. Cete dernière étant une matrice de rotation, elle représente une base orthonormée et on a on a donc : $(r_{11}, r_{12}, r_{13}) \wedge (r_{21}, r_{22}, r_{23}) = (r_{31}, r_{32}, r_{33})$ grâce à cette propriété on peut donc obtenir la matrice R et les angles d'Euler ϕ, γ, ω. Le code associé est en annexe 2. Il reste à discuter du signe du paramètre o_2^c, sachant que nous avons choisi un repère de la mire en haut à gauche, il faudra se déplacer du centre vers la gauche par rapport à la caméra, notre o_2^c est donc négatif.

3.3 Estimation de M

La dernière étape de la méthode de Tsaï consiste à calculer le vecteur M en résolvant de la même façon qu'à l'étape 1 un système matriciel à l'aide de la fonction *linalg*.
Le système que l'on cherche à résoudre est le suivant : $BM = R$ où R est la matrice de rotation calculée à l'étape précédente et

$$\mathrm{B} = \begin{pmatrix} \tilde{u_2}^1 & -(r_{21}x_1^1 + r_{22}x_2^1 + r_{23}x_3^1 + o_2^c) \\ \tilde{u_2}^2 & -(r_{21}x_1^2 + r_{22}x_2^2 + r_{23}x_3^2 + o_2^c) \\ \tilde{u_2}^n & -(r_{21}x_1^n + r_{22}x_2^n + r_{23}x_3^n + o_2^c) \end{pmatrix}$$

Cela nous permet de trouver M une matrice 2x1 contenant o_3^c et f_2 les deux derniers paramètres manquants pour calculer la matrice Mpass de passage des

coordonnées spatiales vers les coordonnées pixels.
Le code correspondant est en annexe 3.

On trouve une distance d'environ 44cm entre la mire et le capteur et une taille de pixel d'environ 6µm*6µm (en divisant f_2 par la focale de la caméra (et en supposant $f_2 = f_1$ car beta vaut environ 1)) ce qui nous semble cohérent.

3.4 Construction des matrices intrinsèques et extrinsèque, puis de la matrice de passage, et test du calibrage

Maintenant que tout les paramètres de calibrage sont déterminés, nous pouvons construire les matrices intrinsèques et extrinsèques puis la matrice de passage du monde objet à l'image.

Calcul de la matrice de passage

```
Mint = np.array([[f1,0,i1,0],[0,f2,i2,0],[0,0,1,0]])
Mext = np.array([[r11,r12,r13,o1c],[r21,r22,r23,o2c],[r31,r32,r33,o3c],[0,0,0,1]])
Mpass = Mint.dot(Mext)
```

Nous pouvons ensuite projeter les points de l'espace construits au début du tp objet sur l'image en appliquant la matrice de passage sur ces points (avec l'ajout d'un 1 en fin de vecteur pour passer en coordonnées homogènes)

Voici le code qui projette les points sur chaque image et les affiche :

Projection et affichage

```
i=0
for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    for k in range(49):
        dot_mm=coord_mm[i*49+k]
        dot_mm.append(1)
        dot_px=Mpass.dot(dot_mm)
        dot_px=dot_px/dot_px[2] #division par alpha
        img = cv2.circle(gray,(int(dot_px[0]),int(dot_px[1])),5,(255, 0, 0),2)
    cv2.imshow("image",gray)
    key = cv2.waitKey(10000)
    i+=1

cv2.destroyAllWindows()
```

On obtient alors les image suivantes, avec une calibration assez cohérente, malgré quelques imprécisions dues probablement à de legères erreurs dans les valeurs issues de mesure physiques, ou à des distorsions non linéaires (feuille de la mire legèrement bombée par exemple) dont nous n'avons pas tenu compte pour la calibration.

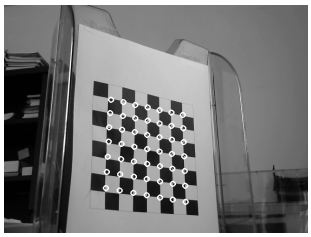

Figure 4: Image 1 + calibrage

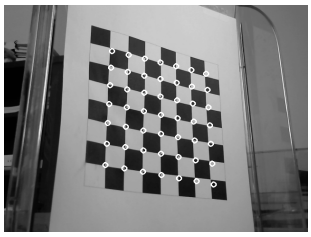

Figure 5: Image 2 + calibrage

4 Annexes et scripts

Annexe 1

Création de A et U_1

```
import numpy as np

A_p=[]
U1_p=[]
for i in range(len(coord_mm)):
    U1_p.append([coord_px[i][0]])
    line=[]
    line.append((coord_px[i][1])*coord_mm[i][0])
    line.append((coord_px[i][1])*coord_mm[i][1])
    line.append((coord_px[i][1])*coord_mm[i][2])
    line.append(coord_px[i][1])
    line.append((-coord_px[i][0])*coord_mm[i][0])
    line.append((-coord_px[i][0])*coord_mm[i][1])
    line.append((-coord_px[i][0])*coord_mm[i][2])
    A_p.append(line)

A=np.array(A_p)
U1=np.array(U1_p)


L=np.linalg.pinv(A).dot(U1)
```

Annexe 2

Calcul de la matrice de rotation

```
o2c_abs=1/math.sqrt(L[4]**2+L[5]**2+L[6]**2)
beta=o2c_abs*math.sqrt(L[0]**2+L[1]**2+L[2]**2)
o2c=-o2c_abs
o1c=o2c*L[3]/beta
r11=L[0][0]*o2c/beta
r12=L[1]*o2c/beta
r13=L[2]*o2c/beta
r21=L[4]*o2c
r22=L[5]*o2c
r23=L[6]*o2c
R1=np.array([r11,r12,r13])
R2=np.array([r21,r22,r23])
R3=np.cross(R1.transpose(),R2.transpose())
r31=R3[0][0]
r32=R3[0][1]
r33=R3[0][2]


theta=-math.atan(r23/r33)
gamma=-math.atan(r12/r11)
omega=math.atan(r13/(-r23*math.sin(theta)+r33*math.cos(theta)))
```

Annexe 3

Calcul de M

```
B_p=[]
R_p=[]
for i in range(len(coord_mm)):
    B_p.append([(coord_px[i][1]),-(r21*coord_mm[i][0]+r22*coord_mm[i][1]+r23*coord_mm[i][2]+o2c)])
    R_p.append((-coord_px[i][1])*(r31*coord_mm[i][0]+r32*coord_mm[i][1]+r33*coord_mm[i][2]))

B=np.array(B_p,dtype='float')
R=np.array(R_p)
M=np.linalg.pinv(B).dot(R)
f2 = M[1]
f1=f2*beta
o3c = M[0]
```

5 Conclusion

Finalement, nous avons pu calibrer la caméra. Nous avons commencé par reconstruire les points du repère objet à partir de mesures physiques (taille des cases, espaces entre les mires sur les deux acquisitions différentes), puis a partir d'un fonction de detection de mire, nous avons pu obtenir la projection de ces différents points en coordonnées pixels sur le capteur de la caméra. A partir de

ces deux représentations de la mire, nous avons pu déterminer les paramètres de la matrice de passage du monde monde objet 3d vers le plan image 2d. Nous avons enfin utilisé la matrice que nous avons créée pour projeter les points objets sur le plan image et vérifier que notre calibration était correcte. Nous avons constaté quelques légères imprécisions, dues soit à des mesures physiques imprécises, soit à de légères distorsions dans l'acquisition, qui montre les limites de l'algorithme de Tsaï à traiter les cas non linéaires.

References

[1] Optique ingénieur. Calibrage géométrique d'une caméra. `http://www.optique-ingenieur.org/fr/cours/OPI_fr_M04_C01/co/Contenu91.html`.

[2] Eric Van Reeth. Cours. *Cours CPE Modélisation et calibrage de caméra*, 2020-2021.

[3] Open Cv Python Tutorials. Open cv camera calibration. `http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html`.
[2] [1] [3]