Trabajo Práctico Final Fundamentos Matemáticos de la Visión Robótica (67.61) 2020

Román Vázquez Lareu, 100815 PONER FECHA

Introducción

En el siguiente trabajo, se buscará desarrolar un software capaz de determinar la pose de ciertos objetos dispuestos en una zona de trabajo. Este proceso constará de 5 etapas.

- Calibración de los parámetros intrínsecos de la cámara: esto se hará a partir de un set de imágenes, donde se deberá justificar su elección.
- Calibración de los parámetros extrínsecos: esto se hará a partir de una unica imágen, logrando definir la terna de la zona de trabajo
- Desarrollo de algoritmo de detección de bloques: se buscará identificar su centro y vertices
- Desarrollo de método de validación del agloritmo generado
- Desarrollo de algoritmo de medición de bloques: se buscará identificar sus dimensiones

1 Calibración de parámetros Intrínsecos

Esta primera etapa se realizará haciendo uso de la función calibrateCamera() de openCV, basada en el algoritmo de Zhang. Tanto esta funcion como cualquier otra a usar en esta seccion y en la siguiente usan el modelo de pinhole. este modelo se caracteriza por formar la vista de una imagen proyectando los puntos 3D en el plano de la imagen usando la siguiente transformacion de perspectiva:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

Sin embargo, los lentes en general tienen cierta distorsión. En este caso particular, el lente de una camara web, tendrá un coeficiente k_1 de distorsión radial ($k_1 > 0$ será de barril, de lo contrario será de almohadon)

Previo al proceso de calibración conviene estudiar cada parámetro intrínseco con el objetivo de elegir correctamente el set de imagenes a usar basado en el resultado obtenido.

En primer lugar las distancias focales, f_x y f_y, se corresponden con la distancia entre el pinhole y el plano imagen. En el caso ideal de una camara pinhole, tendrán el mismo valor. En la práctica pueden diferir por varias razones: distorsión, errores en la calibración, etc. Sin embargo, su similitud será una propiedad a tener en cuenta al momento de elegir un set.

En segundo lugar el centro óptico, c_x y c_y, hacen referencia al punto en el plano imagen por donde pasa perpendicular al mismo y que atrviesa el pinhole. En un caso ideal, estará lo más cerca del centro posible.

En tercer lugar el skew, s, el cual tiene que ver con que el sensor no se encuentre perfectamente alineado al plano focal, pero en este caso será nulo para ambos sets de imágenes.

A continuación se presenta el código de calibración intrinseca. A lo largo de esta primera etapa, se indican los puntos del mundo real 3D que se querrán reconocer, con las distintas vistas del tablero de ajedrez, se utilizará la función de openCv $findChessCorners$ para encontrar las coordenadas en pixeles de esos puntos, que se corresponderán con los puntos 3D provistos. Finalmente, con ambos set de puntos, se llama a la funcion $calibrateCamera$ con el objetivo de obtener la matriz de parametros intrinsecos que relaciona ambos conjuntos de puntos y los coeficientes de distorsión.

Listing 1: Puntos a reconocer en tablero de ajedrez de dimensiones conocidas

```
# esquinas internas del tablero:
ch_size = (8, 6)

# lista de todos los puntos que vamos a recolectar
obj_points = list()
img_points = list()

# Lista de los puntos que vamos a reconocer en el mundo
# objp={(0,0,0), (1,0,0), (2,0,0) .... }
# corresponden a las coordenadas en el tablero de ajedrez.
objp = np.zeros((np.prod(ch_size), 3),  dtype=np.float32)
objp[:, :2] = np.mgrid[0:ch_size[0], 0:ch_size[1]].T.reshape(-1, 2)
for i in range(len(objp)):
    objp[i][0],objp[i][1],objp[i][2] =  objp[i][0]*28,objp[i][1]*28,objp[i↩
       ][2]*28
```

Listing 2: coordenadas en pixeles de los puntos a reconocer

```
# Criterio de corte para el proceso iterativo de refinamientoi de esquinas↩
   .
# Parar si iteramos maxCount veces o si las esquinas se mueven menos de ↩
   epsilon
maxCount = 30
```

```
epsilon = 0.001
criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_MAX_ITER, maxCount, epsilon)
cb_flags = cv2.CALIB_CB_ADAPTIVE_THRESH
#cb_flags = cv2.CALIB_CB_FAST_CHECK

%matplotlib qt

for image_fname in calib_fnames:
    print("Procesando: " + image_fname , end='... ')
    img = cv2.imread(image_fname)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # para subpixel solamente gray
    ret, corners = cv2.findChessboardCorners(img_gray, ch_size, flags=cb_flags)
    if ret:
        print('Encontramos esquinas!')
        obj_points.append(objp)
        print('Buscando esquinas en resoluci n subpixel', end='... ')
        corners_subp = cv2.cornerSubPix(img_gray, corners, (5, 5), (-1, -1), criteria)
        print('OK!')
        img_points.append(corners_subp)
        cv2.drawChessboardCorners(img, ch_size, corners_subp, ret)
        if mostrar_figuras:
            plt.figure()
            plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
            plt.show()
```

Listing 3: llamado a calibrateCamera de openCv

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points,img_gray.shape[::-1], None, None,flags=cv2.CALIB_ZERO_TANGENT_DIST)
```

En este punto, basandonos en el estudio de los parametros de la matriz de cámara, se elige el set de imagenes a utilizar.

Listing 4: llamado a calibrateCamera de openCv

```
Matriz parametros Intrinsecos para conjunto de im genes 1:
K= [[812.10141061   0.         316.58328056]
 [  0.         812.44382868 247.47338425]
 [  0.           0.           1.        ]]
coeficientes de distorsion:
```

```
[[0.02314617 0.         0.         0.         0.        ]]

Matriz parametros Intrinsecos para conjunto de im genes 2:
K= [[660.98217052   0.         334.47300716]
 [  0.         663.73471966 229.39333938]
 [  0.           0.           1.        ]]
coeficientes de distorsion:
 [[0.08695324 0.         0.         0.         0.        ]]
```

A simple vista se observa en primer lugar la ampliamente mayor similitud de las distancias focales correspondientes al set 1, respecto del set 2. Además, sabido el tamaño de las imágenes (640x480 pixeles), el centro óptico del set 2 presenta un mayor offset, de esta manera es conveniente elegir el set 1 de imágenes

Calibración de parámetros Extrínsecos

Esta etapa es muy similar a la anterior, pero lo que se busca es, por el mismo método, las coordenadas del mundo real de los puntos 3D usando una sola vista del patrón de ajedrez de tamaño conocido

Listing 5: Obtención de parámetros extrinsecos de la cámara

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points,↩
    img_gray.shape[::-1], None, None,flags=cv2.CALIB_ZERO_TANGENT_DIST)

rvecs_mtx = cv2.Rodrigues(rvecs[0])[0]
tvecs = tvecs[0]


rotacion
r= [[ 0.99608415 -0.01044154  0.08779143]
 [ 0.00811795  0.99960832  0.02678268]
 [-0.08803669 -0.02596512  0.99577877]]
traslacion
t= [[-108.21183383]
 [-258.32313044]
 [2587.78879303]]
```

Algoritmo de detección de bloques

En primer lugar se le realizará un preprocesamiento a la imagen con el objetivo de destacar los bloques lo mayor posible. Este consta de 5 partes:

- filtro de mediana: eliminaar cualquier ruido presente
- Pasaje de imagen a escala de grises: de BGR a grises
- Binarización global por el método de otsu: en este caso los bloques destacan bastante del fondo, tanto su color como sus bordes, por lo que es favorable aplicar esta binarización
- Apertura: regularizar las formas
- Cierre: eliminar los contornos pequeños

Listing 6: preprocesamiento de imágenes

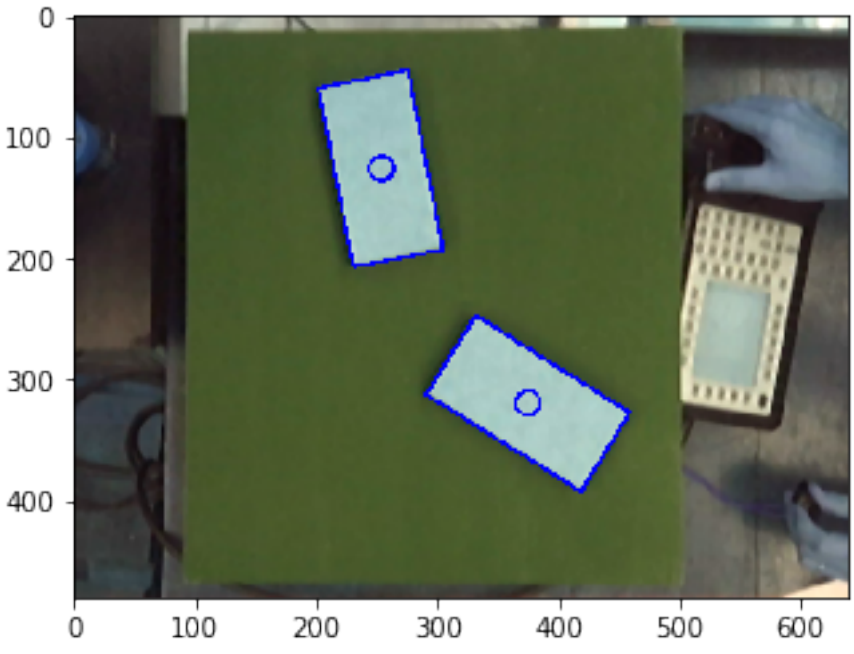
```
1
2     #aplico filtro suaviza
3     img = cv.medianBlur(img,5)
4
5     #paso a gris
6     imggray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
7
8     #binarizacion global, otsu(el contraste entre elementos es alto)
9     ret, img_bin = cv.threshold(imggray,30,255,cv.THRESH_BINARY+cv.↩
          THRESH_OTSU)
10
11    #regularizar las formas
12    #apertura
13
14    kernel = np.ones ((1,1), np.uint8)
15    apertura = cv.morphologyEx(img_bin, cv.MORPH_OPEN, kernel)
16
17    #cierre
18    kernel = np.ones ((2,2), np.uint8)
19    cierre = cv.morphologyEx(apertura, cv.MORPH_CLOSE, kernel)
```

A continuación, nos valemos de la función $findContours$ para hallar todos los contorno presentes en la imagen. Luego se le aplica a cada contorno la función $minAreaRect$, que devuelve el mínimo rectangulo orientado que contiene al contorno. De esta manera, se obtiene un parámetro al cual es mucho más facil aplicarle condiciones para corroborar si se trata del bloque buscado, además esta función devuelve entre otras propiedades, el centro del rectangulo y el ancho y largo.

Listing 7: detección de bloques

```
1     for cnt in contours:
2         rect = cv.minAreaRect(cnt)
```

```
3           centro = rect[0]
4           dimension=rect[1]
5           rotacion = rect[2]
6           area = cv.contourArea(cnt)
7
8
9           if((dimension[0]*dimension[1] != 0)  and (area/(dimension[0]*↩
                dimension[1])>0.95) and area>10000):
10              print("Encontre bloque!")
11              print('Area: {} - Centro: {} - Dimensiones: {} - Rotacien: {}'↩
                    .format(area,centro,dimension,cv.fitEllipse(cnt)[2]))
12              cv.circle(img, (int(centro[0]),int(centro[1])), 10,(0,0,255)↩
                    ,2)
13              box = cv.boxPoints(rect)
14              box = np.int0(box)
15              cv.drawContours(img,[box],0,(0,0,255),2)
16              centros.append(centro)
17              dimensiones.append(dimension)
18
19              plt.figure(i)
20              plt.imshow(img)
21              plt.show()
22
23  #Ejemplo de imagen con bloque detectado
24      Encontre bloque!
25  Area: 11148.0 - Centro: (254.31228637695312, 126.7426986694336) - ↩
        Dimensiones: (75.33026123046875, 150.46812438964844) - Rotacion: ↩
        169.06480407714844
26
```




```
27  \                                                                   *
```

A esta altura, ya se tiene la matriz de parámetros intrínsecos y extrínsecos, los coeficientes de distorsión y las coordenadas de los centros de los bloques. De esta manera, ya se estaría en condiciones de expresar las coordenadas de los centros de los bloques en las coordenadas del mundo 3D:

$$H^{-1} * F^{-1}_{distorsion}(K^{-1} * \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}) = \begin{bmatrix} x_w \\ y_w \\ 0 \\ 1 \end{bmatrix} \tag{2}$$

Con el fin de evitar calcular la inversa de la función, se realiza una aproximación de la misma con un método iterativo.

Listing 8: Resolución de ecuación

```
1
2  #defino funcion de distorsion
3  def f_dist(x):
4      #coeficiente de distorsion radial k1
5      k1 = dist[0][0]
6      #me interesa (u,v), la tercera posicion no
7      xn = x[0:2]/x[2]
8      #radio al cuadrado
9      r_cuadrado = xn[0]**2+xn[1]**2
10     xd = xn*(1+k1 * r_cuadrado)
11     #xd hmogenea
12     xd = np.vstack((xd,1))
13     return xd;
14
15 #hallo inversa de matriz de camara
16 K_inversa = np.linalg.inv(mtx)
17
18 #defino Rt
19 Rt = np.hstack((rvecs_mtx,tvecs))
20 #hago z=0
21 Rt_z0 = np.delete(Rt,2,1)
22 #calculo su inversa
23 Rt_inv = np.linalg.inv(Rt_z0,)
24
25 #lista para almacenar los centros de los bloques en 3D(2D)
26 centros_en_mm = list()
27
28
29 #centros = (x_centro,Y_centro)
30 #calculo posicion para cada centro
31 for i in range(len(centros)):
```

```
32      centro = centros[i]
33      #creo (u,v,1)
34      uv = np.matrix([[centro[0]],[centro[1]],[1]])
35      #mutliplico a K**-1 por (u,v,1)
36      x_dist = K_inversa.dot(uv)
37
38      #metodo iterativo
39      iter = 0
40      delta_x = x_dist - f_dist(x_dist)
41      while(iter<3):
42          X = x_dist - delta_x
43          delta_x = x_dist - f_dist(X)
44          iter +=1
45      #X vector buscado
46      X_final = Rt_inv.dot(X)
47      centros_en_mm.append(X_final)
```