

PracticaStereo

December 7, 2020

1 Práctica de reconstrucción. Parte II. Visión estéreo

Visión Computacional Practica 2.

Autor 1: Fernando Planes Ruiz

Autor 2: Alejandro Muñoz Navarro

Este enunciado está en el archivo “PracticaStereo.ipynb” o su versión “pdf” que puedes encontrar en el Aula Virtual.

1.1 Objetivos

Los objetivos de esta práctica son: * reconstruir puntos de una escena a partir de una serie de correspondencias manuales entre dos imágenes calibradas; * determinar la geometría epipolar de un par de cámaras a partir de sus matrices de proyección; * implementar la búsqueda automática de correspondencias que use las restricciones impuestas por la geometría epipolar, aplicando para ello métodos de cortes de grafos; * realizar una reconstrucción densa de la escena.

1.2 Requerimientos

Para esta práctica es necesario disponer del siguiente software: * Python 2.7 ó 3.X * Jupyter <http://jupyter.org/>. * Las librerías científicas de Python: NumPy, SciPy, y Matplotlib. * La librería OpenCV * La librería PyMaxFlow

El material necesario para la práctica se puede descargar del Aula Virtual en la carpeta **MaterialesPractica** del tema de visión estéreo. Esta carpeta contiene: * Una serie de pares estéreo en el directorio `images`; el sufijo del fichero indica si corresponde a la cámara izquierda (`_left`) o a la derecha (`_right`). Bajo el directorio `rectif` se encuentran varios pares estéreo rectificados. * Un conjunto de funciones auxiliares de Python en el módulo `misc.py`. La descripción de las funciones puede consultarse con el comando `help` o leyendo su código fuente. * El archivo `cameras.npz` con las matrices de proyección del par de cámaras con el que se tomaron todas las imágenes con prefijo `minoru`.

1.3 Condiciones

- La entrega consiste en dos archivos con el código, resultados y respuestas a los ejercicios:
 1. Un “notebook” de Jupyter con los resultados. Las respuestas a los ejercicios debes introducirlas en tantas celdas de código o texto como creas necesarias, insertadas inmediatamente después de un enunciado y antes del siguiente.

2. Un documento “pdf” generado a partir del fuente de Jupyter, por ejemplo usando el comando `jupyter nbconvert --execute --to pdf notebook.ipynb`, o simplemente imprimiendo el “notebook” desde el navegador en la opción del menú “File->Print preview”. Asegúrate de que el documento “pdf” contiene todos los resultados correctamente ejecutados.
- Esta práctica puede realizarse en parejas.

1.4 1. Introducción

En los problemas de visión estéreo se supondrá la existencia de un par de cámaras calibradas cuyas matrices de proyección \mathbf{P}_i vienen dadas por

$$\mathbf{P}_1 = \mathbf{K}_1 \cdot [\mathbf{I} \quad \mathbf{0}] \cdot \begin{bmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix},$$
$$\mathbf{P}_2 = \mathbf{K}_2 \cdot [\mathbf{I} \quad \mathbf{0}] \cdot \begin{bmatrix} \mathbf{R}_2 & \mathbf{t}_2 \\ \mathbf{0}^T & 1 \end{bmatrix}.$$

En esta práctica se usarán las matrices de proyección de dos cámaras para determinar la posición tridimensional de puntos de una escena. Esto es posible siempre que se conozcan las proyecciones de cada punto en ambas cámaras. Desafortunadamente, esta información no suele estar disponible y para obtenerla es preciso emplear el contenido de las imágenes (sus píxeles) en un proceso de búsqueda conocido como puesta en correspondencia. Conocer las matrices de proyección de las cámaras permite acotar el área de búsqueda gracias a las restricciones que proporciona la geometría epipolar.

```
[2]: # uncomment to show results in a window
%matplotlib tk
import numpy as np
import scipy.misc as scpm
import scipy.ndimage as scnd
import matplotlib.pyplot as ppl
import numpy.linalg as npla
import maxflow.fastmin
import misc
import cv2
```

1.5 1. Reconstrucción

Teniendo un conjunto de correspondencias entre dos imágenes, con matrices de calibración P_i conocidas, es posible llevar a cabo una reconstrucción tridimensional de dichos puntos. En el fichero `cameras.npz` se encuentran las matrices de proyección para las dos cámaras. Para cargar este fichero:

```
[3]: cameras = np.load("cameras.npz")
P1 = cameras["left"]
P2 = cameras["right"]
```

Todas las imágenes con el prefijo `minoru` comparten este par de matrices de proyección.

Leemos las imágenes y marcamos al menos seis puntos correspondientes en cada una de ella.

```
[4]: img1 = cv2.imread("images/minoru_cube3_left.jpg")
img2 = cv2.imread("images/minoru_cube3_right.jpg")
```

```
[5]: pt1, pt2 = misc.askpoints(img1, img2)
```

Ejercicio 1. Implementa la función `M = reconstruct(points1, points2, P1, P2)` que, dados una serie de N puntos 2D `points1` de la primera imagen y sus N homólogos `points2` de la segunda imagen (ambos en coordenadas homogéneas, $3 \times N$), y el par de matrices de proyección `P1` y `P2` de la primera y la segunda cámara respectivamente, calcule la reconstrucción tridimensional de cada punto. De ese modo, si `points1` y `points2` son $3 \times N$, la matriz resultante `M` debe ser $4 \times N$.

El tipo de reconstrucción debe ser algebraico, no geométrico.

```
[6]: def cartesiano(p):
    return p[0:-1,:]/p[-1]
```

```
[7]: def reconstruct2(points1, points2, P1, P2):
    """Reconstruct a set of points projected on two images."""

    # Transform homog to cartesian co-ordinates
    p1 = cartesiano(points1).T
    p2 = cartesiano(points2).T

    # build coefficient matrix and compute reconstruction by least-squares.
    # Useful functions are npla.lstsq() and npla.pinv()

    c = 2
    N = np.size(points1,1)
    M = np.ones((4,N))
    for i in range(N):
        A = np.zeros((2*c,3))
        A[0,:] = P1[0,0:3]-(P1[2,0:3]*p1[i,0])
        A[1,:] = P1[1,0:3]-(P1[2,0:3]*p1[i,1])
        A[2,:] = P2[0,0:3]-(P2[2,0:3]*p2[i,0])
        A[3,:] = P2[1,0:3]-(P2[2,0:3]*p2[i,1])

        b = np.zeros((2*c,1))
        b[0] = P1[0,3]-(P1[2,3]*p1[i,0])
        b[1] = P1[1,3]-(P1[2,3]*p1[i,1])
        b[2] = P2[0,3]-(P2[2,3]*p2[i,0])
        b[3] = P2[1,3]-(P2[2,3]*p2[i,1])

        m=npla.lstsq(A,-b)
        a = m[0]
        M[0,i] = a[0,0]
        M[1,i] = a[1,0]
```

```

        M[2,i] = a[2,0]

    return M

```

Reconstruye los puntos marcados y pinta su estructura 3D.

```

[8]: # reconstruct
M=reconstruct2(pt1, pt2, P1, P2)

# convert from homog to cartesian
mM = cartesiano(M)
#print(mM)
# plot 3D
ppl.close()
misc.plot3D(mM[0,:],mM[1,:],mM[2,:])

```

<ipython-input-7-40187e95d7e9>:27: FutureWarning: `rcond` parameter will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.

To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

```
m=npla.lstsq(A,-b)
```

```
[8]: <mpl_toolkits.mplot3d.axes3d.Axes3D at 0x7fd9ce75da30>
```

Ejercicio 2. Elige un par estéreo de las imágenes del conjunto “building” de la práctica de calibración y realiza una reconstrucción de un conjunto de puntos de dicho edificio estableciendo las correspondencias a mano.

Añadimos el ejercicio 8 de la práctica anterior:

```

[ ]: import pygame
import numpy as np
import cv2

import glob
import copy
import scipy.misc as scpm
import matplotlib.pyplot as plt

def load_images(filenamees):
    """Load multiple images."""
    return [cv2.imread(filename) for filename in filenamees]

filenamees = sorted(glob.glob('./images/building/*'))

```

```

# Abrimos las imágenes
imgs = load_images(filenamees)

def refinar(img, xp, yp, scale_x, scale_y, RAD = 14):

    # Reescalar y redondear las coordenadas de entrada (corregir inversión de
    ↪pygame)
    x = round(yp * scale_y)
    y = round(xp * scale_x)

    # Extraer subimagen centrada en x, y
    s = img[x - RAD : x + RAD + 1, y - RAD : y + RAD + 1]

    gray = cv2.GaussianBlur(cv2.cvtColor(s, cv2.COLOR_BGR2GRAY), (5,5), 0)
    dst = cv2.cornerHarris(gray, 8, 5, 0.04)

    dst = cv2.dilate(dst, None)
    ret, dst = cv2.threshold(dst, 0.3 * dst.max(), 255, 0)
    dst = np.uint8(dst)

    # find centroids
    ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)

    # define the criteria to stop and refine the corners
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)
    corners = cv2.cornerSubPix(gray, np.float32(centroids), (5, 5), (-1,
    ↪-1), criteria)

    corner = np.int0(np.mean(cv2.cornerSubPix(gray, np.float32(centroids), (5,
    ↪5), (-1, -1), criteria), axis=0))

    return (x - RAD + corner[1]), (y - RAD + corner[0])

def getCroppingMask(img, radio = 1, screen_width = 1600, screen_height = 900):
    # Rescale images
    scale_x = img.shape[1] / screen_width
    scale_y = img.shape[0] / screen_height

    imgr = cv2.resize(img, dsize=(screen_width, screen_height),
    ↪interpolation=cv2.INTER_CUBIC)

    screen = pygame.display.set_mode(imgr.shape[-2::-1])

    imgpyg = pygame.image.frombuffer(imgr, imgr.shape[-2::-1], 'RGB')

```

```

screen.blit(imgpyg,(0,0))
pygame.display.flip() # update the display

polypoints = []

while True:
    e = pygame.event.wait()
    if e.type == pygame.QUIT:
        break;
    if e.type == pygame.MOUSEBUTTONDOWN:
        x, y = e.pos
        xr, yr = refinar(img, x, y, scale_x, scale_y)
        if pygame.mouse.get_pressed()[0] or len(polypoints) <= 3:
            polypoints.append([yr, xr])
            x = round(yr / scale_x)
            y = round(xr / scale_y)
            if len(polypoints) == 1:
                pygame.draw.circle(screen, (0,255,0), (x, y), radio)
            else:
                pygame.draw.line(screen, (0,255,0),
→(round(polypoints[-2][0] / scale_x),round(polypoints[-2][1] / scale_y)), (x,
→y), radio)
                elif pygame.mouse.get_pressed()[2]:
                    if len(polypoints) > 3:
                        pygame.draw.line(screen, (0,255,0),
→(round(polypoints[-1][0] / scale_x),round(polypoints[-1][1] / scale_y)),
→(round(polypoints[0][0] / scale_x),round(polypoints[0][1] / scale_y)), radio)

            pygame.display.flip()

pygame.quit()

mask = np.zeros(img.shape[0:2], dtype=np.uint8)
points = np.array([polypoints])

cv2.drawContours(mask, [points], -1, (255, 255, 255), -1, cv2.LINE_AA)

return mask

masks = [getCroppingMask(img) for img in imgs]

```

Creamos aquí unas máscaras como las obtenidas en el código anterior para no tener que repetirlo.

```

[ ]: polypoints = [np.array([[[1976, 135], [2973, 773], [3149, 2023], [3111, 2032],
→[3114, 2072], [3345, 2117], [3413, 2485], [1984, 2434], [809, 2537], [843,
→2188], [1053, 2119], [1056, 2092], [1018, 2084], [1139, 875]]]]),

```

```

np.array([[[1786, 113], [2776, 776], [2948, 2000], [2909, 2013], [2912, 2048],
→[3133, 2091], [3195, 2447], [1802, 2413], [600, 2533], [639, 2175], [854,
→2105], [861, 2081], [818, 2069], [935, 840]]]),
np.array([[[2082, 98], [2915, 857], [3088, 2085], [3062, 2084], [3063, 2114],
→[3272, 2181], [3329, 2543], [2207, 2472], [628, 2582], [684, 2198], [924,
→2140], [929, 2099], [891, 2098], [1029, 815]]]),
np.array([[[1974, 169], [2808, 1075], [3103, 2407], [3071, 2417], [3320, 2524],
→[3411, 2964], [2264, 2873], [1530, 2958], [158, 2967], [236, 2617], [550,
→2524], [560, 2489], [505, 2485], [760, 998]]]),
np.array([[[2763, 829], [3029, 2151], [2998, 2162], [3229, 2253], [3311, 2656],
→[2203, 2596], [109, 2866], [197, 2370], [504, 2276], [516, 2237], [458,
→2234], [671, 740], [1702, 16], [2048, 20]]]])

masks = [cv2.drawContours(np.zeros(imgs[0].shape[0:2], dtype=np.uint8), [pps],
→-1, (255, 255, 255), -1, cv2.LINE_AA) for pps in polypoints]

plt.imshow(masks[0])

```

```

[ ]: for i in range(len(imgs)):
    img = imgs[i].copy()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    canny = cv2.Canny(gray, 100, 180, 1)

    edges = cv2.bitwise_and(canny, canny, mask = masks[i])

    lines = cv2.HoughLines(edges, 1, np.pi/180, 250)
    for line in lines:
        rho, theta = line[0]
        # skip near-vertical lines
        if abs(theta-np.pi/90) < np.pi/9:
            continue
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a*rho
        y0 = b*rho
        x1 = int(x0 + 10000*(-b))
        y1 = int(y0 + 10000*(a))
        x2 = int(x0 - 10000*(-b))
        y2 = int(y0 - 10000*(a))
        cv2.line(img, (x1,y1), (x2,y2), (0,255,0), 3)

    fig, axs = plt.subplots(1, 2, figsize=(16, 8))
    axs[0].imshow(edges, cmap = 'gray')
    axs[1].imshow(img)

```

A continuación buscaremos los puntos de fuga. Para ello usaremos el kmeans para hacer 3 clusters

con las líneas y aproximaremos la intersección de las líneas de cada grupo, que se corresponderá con un punto de fuga.

```
[ ]: from collections import defaultdict

def segment_by_angle_kmeans(lines, k = 2, **kwargs):
    """Groups lines based on angle with k-means.

    Uses k-means on the coordinates of the angle on the unit circle
    to segment `k` angles inside `lines`.
    """

    # Define criteria = (type, max_iter, epsilon)
    default_criteria_type = cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER
    criteria = kwargs.get('criteria', (default_criteria_type, 10, 1.0))
    flags = kwargs.get('flags', cv2.KMEANS_RANDOM_CENTERS)
    attempts = kwargs.get('attempts', 10)

    # returns angles in [0, pi] in radians
    angles = np.array([line[0][1] for line in lines])
    # multiply the angles by two and find coordinates of that angle
    pts = np.array([np.cos(2*angle), np.sin(2*angle)]
                    for angle in angles], dtype=np.float32)

    # run kmeans on the coords
    labels, centers = cv2.kmeans(pts, k, None, criteria, attempts, flags)[1:]
    labels = labels.reshape(-1) # transpose to row vec

    # segment lines based on their kmeans label
    segmented = defaultdict(list)
    for i, line in zip(range(len(lines)), lines):
        segmented[labels[i]].append(line)
    segmented = list(segmented.values())
    return segmented

def segmented_intersections(lines):
    intersections = []

    for i in range(len(lines)):
        A = np.array([[np.cos(line[0][1]), np.sin(line[0][1])] for line in
↪ lines[i]])
        b = np.array([[line[0][0]] for line in lines[i]])
        x, _, _, _ = np.linalg.lstsq(A, b)
        x0, y0 = int(np.round(x[0])), int(np.round(x[1]))
        intersections.append([x0, y0])
```



```

intersections = np.array(sorted(intersections))

return intersections

fig, axs = plt.subplots(1, len(imgs), figsize=(25, 25))

vanishing_points = []

for i in range(len(imgs)):
    gray = cv2.cvtColor(imgs[i], cv2.COLOR_BGR2GRAY)

    canny = cv2.Canny(gray, 100, 180, 1)

    edges = cv2.bitwise_and(canny, canny, mask = masks[i])

    lines = cv2.HoughLines(edges, 1, np.pi/180, 250)

    k = 3

    segmented = segment_by_angle_kmeans(lines, k)

    intersections = segmented_intersections(segmented)

    vanishing_points.append(intersections)

    maxx = max(intersections[:,0].max() + 1, imgs[i].shape[0])
    minx = min(intersections[:,0].min(), 0)
    maxy = max(intersections[:,1].max() + 1, imgs[i].shape[1])
    miny = min(intersections[:,1].min(), 0)
    imgwpts = 255 * np.ones(((maxy - miny), (maxx - minx), 3), dtype=np.uint8)
    imgwpts[-miny:img.shape[0]-miny, -minx:img.shape[1]-minx] = imgs[i][:,:]

    for j in range(k):
        for l in range(3):
            cv2.line(imgwpts, (intersections[l][0] - minx, intersections[l][1] - miny), (intersections[l-1][0] - minx, intersections[l-1][1] - miny), (120,120, 120), 20)

            for line in segmented[j]:
                rho,theta = line[0]
                # skip near-vertical lines
                if abs(theta-np.pi/90) < np.pi/9:
                    continue
                a = np.cos(theta)

```

```

        b = np.sin(theta)
        x0 = a*rho
        y0 = b*rho
        x1 = int(x0 + 10000*(-b))
        y1 = int(y0 + 10000*(a))
        x2 = int(x0 - 10000*(-b))
        y2 = int(y0 - 10000*(a))
        if j == 0:
            cv2.line(imgwpts,(x1 - minx,y1 - miny),(x2 - minx,y2 -
↪miny),(0,255,0),3)
        elif j == 1:
            cv2.line(imgwpts,(x1 - minx,y1 - miny),(x2 - minx,y2 -
↪miny),(255,0, 0),3)
        else:
            cv2.line(imgwpts,(x1 - minx,y1 - miny),(x2 - minx,y2 -
↪miny),(0,0, 255),3)

    axs[i].imshow(imgwpts)

```

Este proceso es muy lento, por lo que usaremos el siguiente código para no tener que repetirlo:

```

[199]: vanishing_points = [np.array([[ -898,  2691],
    [ 1915, -6619],
    [ 5784,  2594]]),
    np.array([[ -1273,  2729],
    [ 1708, -6576],
    [ 5342,  2515]]),
    np.array([[ -1933,  2783],
    [ 1860, -6430],
    [ 4872,  2643]]),
    np.array([[ -1858,  3020],
    [ 1650, -4003],
    [ 4583,  3027]]),
    np.array([[ -2861,  3223],
    [ 1526, -5275],
    [ 4418,  2725]])]
vanishing_points

```

```

[199]: [array([[ -898,  2691],
    [ 1915, -6619],
    [ 5784,  2594]]),
    array([[ -1273,  2729],
    [ 1708, -6576],
    [ 5342,  2515]]),
    array([[ -1933,  2783],
    [ 1860, -6430],
    [ 4872,  2643]]),

```

```

array([[ -1858,  3020],
       [ 1650, -4003],
       [ 4583,  3027]]),
array([[ -2861,  3223],
       [ 1526, -5275],
       [ 4418,  2725]])]

```

```

[206]: import scipy.linalg as la
# Calculamos K
K = []
mA = []
for points in vanishing_points:
    A = np.array([
        [points[0][0] * points[1][0] + points[0][1] * points[1][1],
        points[0][0] + points[1][0], points[0][1] * points[1][1], 1],
        [points[0][0] * points[2][0] + points[0][1] * points[2][1],
        points[0][0] + points[2][0], points[0][1] * points[2][1], 1],
        [points[1][0] * points[2][0] + points[1][1] * points[2][1],
        points[1][0] + points[2][0], points[1][1] * points[2][1], 1]
    ])
    mA.append(A)
    w = la.null_space(A)
    W = np.array([[w[0][0], 0, w[1][0]], [0, w[0][0], w[2][0]], [w[1][0],
    w[2][0], w[3][0]]])
    K.append(np.linalg.inv(np.linalg.cholesky(W)))

print(K)

```

```

[array([[ 3.41305751e+03, -6.34676413e-17, -9.14559699e-19],
       [ 0.00000000e+00,  3.41305751e+03,  0.00000000e+00],
       [ 2.39016382e+03,  6.76739124e-01,  1.22082759e+00]])], array([[
3.32371089e+03, -1.55149656e-16, -4.20687100e-17],
       [ 0.00000000e+00,  3.32371089e+03,  0.00000000e+00],
       [ 2.14771387e+03,  6.54010176e-01,  1.19060825e+00]])], array([[
3.41781815e+03, -3.50739663e-17, -1.98283885e-16],
       [ 0.00000000e+00,  3.41781815e+03,  0.00000000e+00],
       [ 2.32224082e+03,  6.53207826e-01,  1.20898886e+00]])], array([[
2.92884418e+03, -1.30786976e-16, -1.71296985e-17],
       [ 0.00000000e+00,  2.92884418e+03,  0.00000000e+00],
       [ 1.99333820e+03,  6.23759570e-01,  1.20962847e+00]])],
array([[3.28635761e+03, 7.08746940e-17, 9.91573359e-18],
       [0.00000000e+00, 3.28635761e+03, 0.00000000e+00],
       [2.00414982e+03, 5.70631033e-01, 1.17128303e+00]])]

```

```

[287]: # Matriz de rotación O
u0 = np.array([vanishing_points[0][0][0], vanishing_points[0][0][1], 1])
v0 = np.array([vanishing_points[0][2][0], vanishing_points[0][2][1], 1])

```

```

r1 = np.dot(np.linalg.inv(K[0]), u0.T)
r2 = np.dot(np.linalg.inv(K[0]), v0.T)
r3 = np.cross(r3, r1)

R0 = np.array([r1, r2, r3])
print("R0:",R0)

# Matriz de rotación 2
u2 = np.array([vanishing_points[2][0][0], vanishing_points[2][0][1], 1])
v2 = np.array([vanishing_points[2][2][0], vanishing_points[2][2][1], 1])

r1 = np.dot(np.linalg.inv(K[2]), u2.T)
r2 = np.dot(np.linalg.inv(K[2]), v2.T)
r3 = np.cross(r3, r1)

R2 = np.array([r1, r2, r3])
print("R2:",R2)

```

```

R0: [[-2.63107198e-01  7.88442618e-01  5.15499274e+02]
      [ 1.69466819e+00  7.60022353e-01 -3.31746183e+03]
      [-4.94276779e+66 -3.84280175e+68  5.85223693e+65]]
R2: [[-5.65565491e-01  8.14262163e-01  1.08673242e+03]
      [ 1.42547081e+00  7.73300359e-01 -2.73765271e+03]
      [-4.17610200e+71  5.37113501e+69 -2.21360314e+68]]

```

Escogemos la imagen 1 y la imagen 3. Siguiendo el punto 2.1 del libro: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.920.2887&rep=rep1&type=pdf> Podemos calcular P como: $P = KR/[1\ 0\ 0; 0\ 1\ 0; 0\ 0\ 1; 0\ 0\ 0]$

```

[332]: aux1= [[u0[0],u0[1],u0[2]], [v0[0],v0[1],v0[2]], [1,1,1]]
aux2= K[0]*R0
P3 = aux2*np.linalg.inv(aux1)
uno = np.ones((1,3)).T

P32 = np.concatenate((P3,uno),axis=1)
aux3= [[u2[0],u2[1],u2[2]], [v2[0],v2[1],v2[2]], [1,1,1]]
aux4= K[2]*R2
P4 = aux4*np.linalg.inv(aux3)
P42 = np.concatenate((P4,uno),axis=1)
print(inv)
print("P0:",P32)
print("P2:",P42)

```

```

[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]
P0: [[ 1.30176381e-01 -7.52537367e-21  2.55661417e-21  1.00000000e+00]

```

```
[ 0.00000000e+00  1.30371602e-01  0.00000000e+00  1.00000000e+00]
[ 2.10689018e+66  5.21790373e+64  7.14727997e+65  1.00000000e+00]]
P2: [[ 2.73675281e-01 -4.25772067e-21  1.61662385e-18  1.00000000e+00]
[ 0.00000000e+00  2.73920502e-01  0.00000000e+00  1.00000000e+00]
[ 1.15840207e+71 -8.86670277e+65 -2.67721755e+68  1.00000000e+00]]
```

```
[333]: img3 = cv2.imread("images/building/build_001.jpg")
img4 = cv2.imread("images/building/build_003.jpg")

#pt3, pt4 = misc.askpoints(img3, img4)
# reconstruct
M=reconstruct2(pt3, pt4, P32, P42)

# convert from homog to cartesian
mM = cartesiano(M)
#print(mM)
# plot 3D
ppl.close()
misc.plot3D(mM[0,:], mM[1,:], mM[2,:])
```

<ipython-input-7-40187e95d7e9>:27: FutureWarning: `rcond` parameter will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.

To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

```
m=npla.lstsq(A,-b)
```

```
[333]: <mpl_toolkits.mplot3d.axes3d.Axes3D at 0x7fd96e997f70>
```

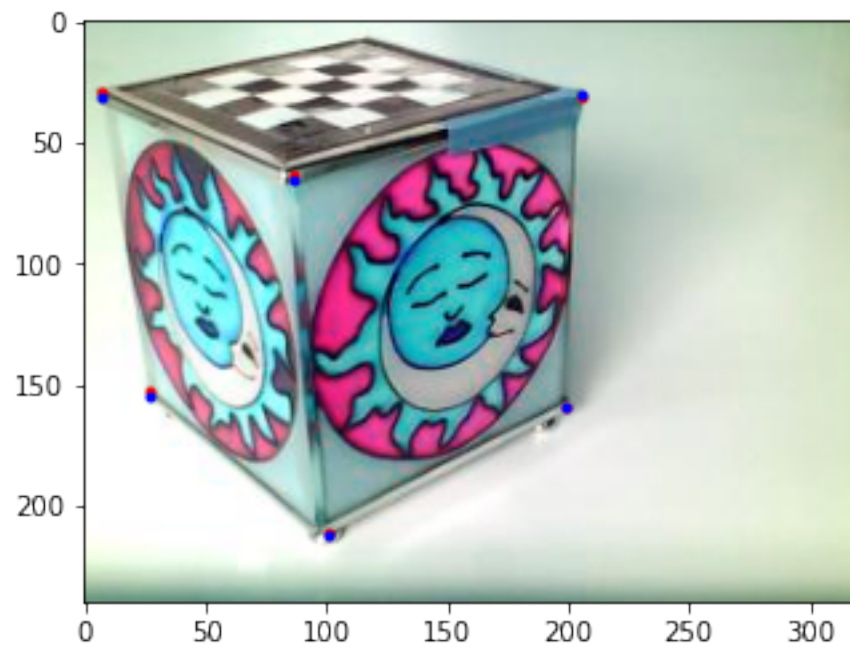
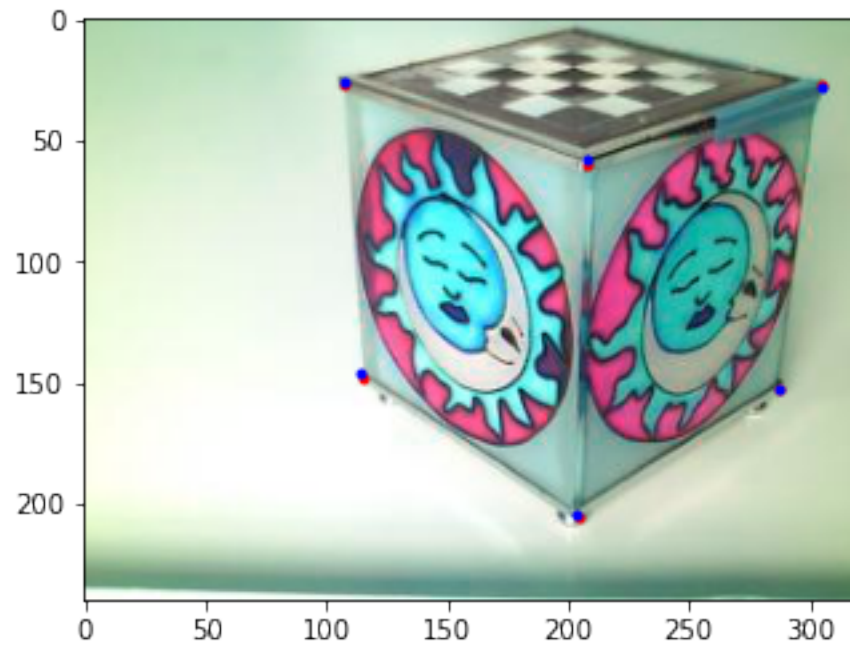
Ejercicio 3. Reproyecta los resultados de la reconstrucción en las dos cámaras y dibuja las proyecciones sobre las imágenes originales. Pinta también en otro color los puntos seleccionados manualmente. Comprueba si las proyecciones coinciden con los puntos marcados a mano. Comenta los resultados. Para dibujar los puntos puedes usar la función `plthom` de la práctica anterior o la versión que se distribuye con esta práctica (`misc.plthom`).

```
[9]: # Proyecto los puntos en ambas cámaras
proy1 = P1.dot(M)
proy2 = P2.dot(M)

%matplotlib inline
# Pinto con misc.plthom()
ppl.figure()
misc.plthom(proy1, 'r.')
misc.plthom(pt1, 'b.')
ppl.imshow(img1)
ppl.show()

ppl.figure()
```

```
misc.plothom(proy2,'r.')  
misc.plothom(pt2,'b.')  
ppl.imshow(img2)  
ppl.show()
```



1.6 2. Geometría epipolar

La geometría epipolar deriva de las relaciones que aparecen en las proyecciones de una escena sobre un par de cámaras. La matriz fundamental \mathbf{F} , que depende exclusivamente de la configuración de las cámaras y no de la escena que éstas observan, es la representación algebraica de dicha geometría: a partir de ella se pueden calcular los epipolos y las líneas epipolares. La relación entre un par de cámaras \mathbf{P}_1 , \mathbf{P}_2 y la matriz fundamental es de n-a-1 (salvo factor de escala). Es decir, dadas dos cámaras calibradas, sólo tienen una matriz fundamental (excepto un factor de escala); dada una matriz fundamental existen infinitas configuraciones de cámaras posibles asociadas a ella.

1.6.1 2.1 Estimación de la matriz fundamental

Ejercicio 4. Implementa la función $\mathbf{F} = \text{projmat2f}(\mathbf{P}_1, \mathbf{P}_2)$ que, dadas dos matrices de proyección, calcule la matriz fundamental asociada a las mismas. \mathbf{F} debe ser tal que, si m_1 de la imagen 1 y m_2 de la imagen 2 están en correspondencia, entonces $m_2^T \mathbf{F} m_1 = 0$.

- **Respuesta:** Definimos la función para calcular una matriz antisimétrica usando la matriz de la web: https://es.qaz.wiki/wiki/Skew-symmetric_matrix Para ello hemos usado la función skew de la librería misc proporcionada por el profesor.

```
[10]: def projmat2f(P1,P2):  
    """ Calcula la matriz fundamental a partir de dos de proyeccion"""  
    # Calculamos la matriz fundamental como:  
    # P1 = [A|b] y P2 = [B|d]  
    # F = (B^-1).T [(B^-1)d - (A^-1)b] * (A^-1)  
  
    A = P1[:,0:3]  
    b = P1[:,3]  
    B = P2[:,0:3]  
    d = P2[:,3]  
  
    asim = misc.skew(np.dot(npla.pinv(B),d) - np.dot(npla.pinv(A),b))  
    return np.dot(np.dot(npla.pinv(B.T), asim), npla.pinv(A))
```

```
[11]: # compute Fundamental matrix  
F = projmat2f(P1, P2)  
print(F)
```

```
[[ 8.37918051e-09 -2.64352488e-06  8.61307712e-04]  
 [ 8.17120601e-06  4.36740640e-06  1.37641120e-01]  
 [-2.27541839e-03 -1.42176580e-01  7.61491838e-03]]
```

Ejercicio 5 ¿Cómo es la matriz fundamental de dos cámaras que comparten el mismo centro? (Por ejemplo, dos cámaras que se diferencian sólo por una rotación.)

- **Respuesta:** La matriz fundamental de dos cámaras que comparten el mismo centro sería la matriz identidad.

1.6.2 2.2 Comprobación de F (OPCIONAL)

En los siguientes dos ejercicios vamos a comprobar que la matriz F estimada a partir de P_1 y P_2 es correcta.

Ejercicio 6. Comprueba que F es la matriz fundamental asociada a las cámaras P_1 y P_2 . Para ello puedes utilizar el resultado 9.12, que aparece en la página 255 del libro Hartley, Zisserman. “Multiple View Geometry in Computer Vision.” (second edition). Cambridge University Press, 2003.

- “**Result 9.12.** A non-zero matrix F is the fundamental matrix corresponding to a pair of camera matrices P and P if and only if $P.T \times F \times P$ is skew-symmetric.”

```
[12]: result = np.dot(P2.T,np.dot(F,P1))
      result = result + result.T
      print(result)
```

```
[[ -2.28321845e-13 -4.71844785e-14 -2.84217094e-14  1.47338142e-10]
 [ -4.71844785e-14 -2.29130477e-15  8.03801470e-14  1.07149845e-11]
 [ -2.84217094e-14  8.03801470e-14 -3.10962325e-13  4.36557457e-11]
 [ 1.47338142e-10  1.07149845e-11  4.36557457e-11 -9.13934786e-08]]
```

- **Respuesta:** Es skew-simétrica, por lo que verificaríamos F .

También se puede comprobar geoméricamente la bondad de una matriz F , si las epipolares con ella estimadas pasan por el homólogo de un punto dado en una de las imágenes.

Dada la matriz fundamental F entre las cámaras 1 y 2, se puede determinar, para un determinado punto m_1 en la imagen de la cámara 1, cuál es la recta epipolar l_2 donde se encontrará su homólogo en la cámara 2:

$$l_2 = Fm_1.$$

Las siguientes dos funciones sirven para comprobar esta propiedad. En primer lugar, se necesita una función que dibuje rectas expresadas en coordenadas homogéneas, es decir, la versión de `plthom` para rectas en lugar de puntos.

Ejercicio 7. Implementa la función `plthline(line)` que, dada una línea expresada en coordenadas homogéneas, la dibuje.

```
[166]: def plthline(line, axes = None):
      """Plot a line given its homogeneous coordinates.

      Parameters
      -----
      line : array_like
          Homogeneous coordinates of the line.
      axes : AxesSubplot
          Axes where the line should be plotted. If not given,
          line will be plotted in the active axis.
      """
      if axes == None:
```



```

axes = ppl.gca()

[x0, x1, y0, y1] = axes.axis()

#      (x0, y0) .------. (x1, y0)
#      /                               /
#      /                               /
#      /                               /
#      /                               /
#      /                               /
#      /                               /
#      (x0, y1) .------. (x1, y1)

# TODO: Compute the intersection of the line with the image
# borders.
l0,l1,l2 = line[0], line[1], line[2]

y2 = (x0 * l0 + l2) / -l1
y3 = (x1 * l0 + l2) / -l1

# TODO: Plot the line with axes.plot.
#axes.plot(...)
plotline = axes.plot([x0,x1],[y2,y3], 'r-')

axes.axis([x0, x1, y0, y1])
return plotline

```

Ejercicio 8. Completa la función `plot_epipolar_lines(image1, image2, F)` que, dadas dos imágenes y la matriz fundamental que las relaciona, pide al usuario puntos en la imagen 1 y dibuje sus correspondientes epipolares en la imagen 2 usando `plotline`.

```

[167]: def plot_epipolar_lines(image1, image2, F):
        """Ask for points in one image and draw the epipolar lines for those points.

        Parameters
        -----
        image1 : array_like
            First image.
        image2 : array_like
            Second image.
        F : array_like
            3x3 fundamental matrix from image1 to image2.
        """

        # Prepare the two images.
        fig = ppl.gcf()
        fig.clf()
        ax1 = fig.add_subplot(1, 2, 1)

```

```

ax1.imshow(image1)
ax1.axis('image')
ax2 = fig.add_subplot(1, 2, 2)
ax2.imshow(image2)
ax2.axis('image')
ppl.draw()

ax1.set_xlabel("Choose points in left image (or right click to end)")
point = ppl.ginput(1, timeout=-1, show_clicks=False, mouse_pop=2,
↪mouse_stop=3)
while len(point) != 0:
    # point has the coordinates of the selected point in the first image.
    point = np.hstack([np.array(point[0]), 1])
    ax1.plot(point[0], point[1], '.r')

    # TODO: Determine the epipolar line.
    line = np.dot(F, point)

    # Plot the epipolar line with plotline (the parameter 'axes' should be
↪ax2).
    plotline(line, axes=ax2)

    ppl.draw()
    # Ask for a new point.
    point = ppl.ginput(1, timeout=-1, show_clicks=False, mouse_pop=2,
↪mouse_stop=3)

ax1.set_xlabel('')
ppl.draw()

```

Utiliza esta función con un par de imágenes llamándola de dos formas diferentes: seleccionando puntos en la imagen izquierda y dibujando las epipolares en la imagen derecha y viceversa. Comprueba en ambos casos que las epipolares siempre pasan por el punto de la segunda imagen correspondiente al seleccionado en la primera. Esto confirmará la corrección de la matriz F .

Añade dos figuras una que muestre la selección de puntos en la imagen izquierda y las rectas correspondientes en la imagen derecha, y otra que lo haga al revés. Indica para ambos casos qué matriz fundamental has usado al llamar a `plot_epipolar_lines`.

```

[197]: %matplotlib tk
ppl.close()
plot_epipolar_lines(img1, img2, F)

```

```

[198]: %matplotlib tk
ppl.close()
plot_epipolar_lines(img2, img1, F.T)

```

1.7 3. Rectificación de imágenes

Es recomendable trabajar a partir de ahora con imágenes en blanco y negro y con valores reales entre 0 y 1 para cada uno de sus píxeles. Eso se puede conseguir con

```
[13]: img1 = misc.rgb2gray(img1/255.0)
      img2 = misc.rgb2gray(img2/255.0)
```

La mayoría de algoritmos de puesta en correspondencia, incluyendo el que se va a implementar en esta práctica, requieren que las imágenes de entrada estén rectificadas.

Dos imágenes están rectificadas si sus correspondientes epipolares están alineadas horizontalmente. La rectificación de imágenes facilita enormemente los algoritmos de puesta en correspondencia, que pasan de ser problemas de búsqueda bidimensional a problemas de búsqueda unidimensional sobre filas de píxeles de las imágenes. En el material de la práctica se han incluido dos funciones que rectifican (mediante un método lineal) dos imágenes. La función `H1, H2 = misc.projmat2rectify(P1, P2, imsize)` devuelve, dadas las dos matrices de proyección y el tamaño de las imágenes en formato (filas,columnas), las homografías que rectifican, respectivamente, la imagen 1 y la imagen 2. La función `projmat2rectify` hace uso de `projmat2f`, por lo que es necesario que esta función esté disponible.

Ejercicio 9. Se tienen dos imágenes no rectificadas `im1` e `im2`, y su matriz fundamental asociada \mathbf{F} . Con el procedimiento explicado, se encuentran un par de homografías \mathbf{H}_1 y \mathbf{H}_2 que dan lugar a las imágenes rectificadas `O1` y `O2`. ¿Cuál es la matriz fundamental \mathbf{F} asociada a estas dos imágenes? ¿Por qué?

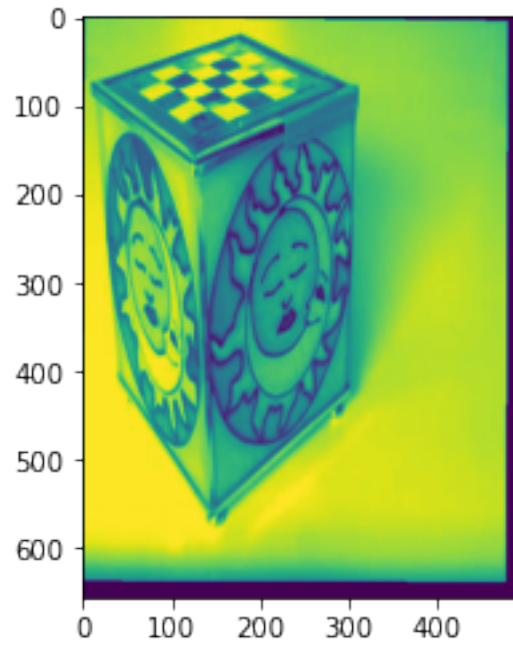
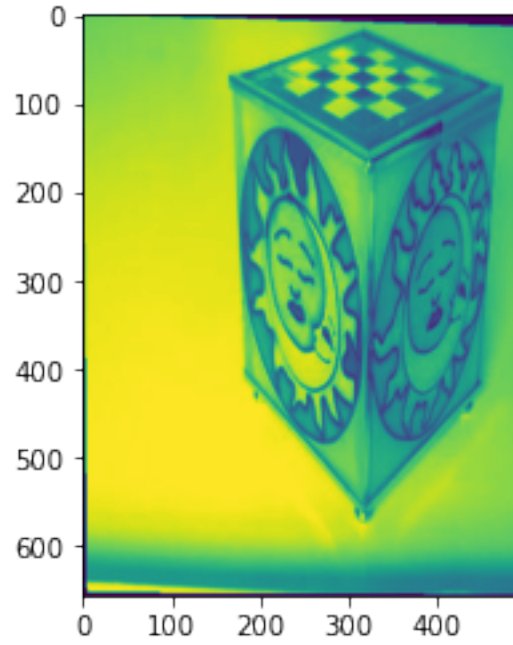
Nota: \mathbf{F} depende exclusivamente de \mathbf{F} , \mathbf{H}_1 y \mathbf{H}_2 .

Ejercicio 10. Rectifica el par de imágenes estéreo `img1` e `img2` y calcula la matriz fundamental asociada a estas imágenes.

```
[14]: H1, H2 = misc.projmat2rectify(P1,P2,projmat2f, img1.shape)
      O1, O2 = misc.rectify_images(img1,img2,H1,H2)
```

```
[15]: %matplotlib inline
      ppl.imshow(O1)
      ppl.figure()
      ppl.imshow(O2)
```

```
[15]: <matplotlib.image.AxesImage at 0x7fd9cfc77e20>
```



Ejercicio 11. Calcula y muestra la matriz fundamental de las imágenes rectificadas. Justifica el resultado obtenido (mira la sección 9.3.1 del libro de Hartley y Zisserman, pág. 248 y 249).

```
[20]: Fr=np.dot(np.dot(npla.inv(H2.T),F),npla.inv(H1))
      Fr=Fr/Fr[2,1]
      Fr
```

```
[20]: array([[ 2.98355387e-22, -5.16663144e-17,  2.36802336e-15],
             [ 1.10553811e-16, -4.87134944e-18, -1.00000000e+00],
             [ 5.21968769e-16,  1.00000000e+00,  3.53641254e-12]])
```

Ejercicio 12. Usa `plot_epipolar_lines` para dibujar varias líneas epiplares de las imágenes rectificadas. Muestra los resultados.

```
[ ]: %matplotlib tk
      ppl.close()
      plot_epipolar_lines(O1,O2,Fr)
```

1.8 4. Búsqueda de correspondencias

La búsqueda de correspondencias consigue establecer automáticamente las correspondencias de puntos entre dos imágenes (lo que se ha hecho manualmente en el ejercicio 2) haciendo uso de las restricciones que proporciona la geometría epipolar.

1.8.1 4.1 Cálculo de las medidas de similaridad

Una vez rectificadas las dos imágenes de un par estéreo, se pueden buscar las correspondencias. Una matriz de disparidades \mathbf{S} indica, para cada píxel de la imagen 1 rectificada, a cuántos píxeles de diferencia está su correspondencia en la imagen 2 rectificada. En nuestra práctica, para simplificar el problema, vamos a considerar que los elementos de \mathbf{S} son enteros. Para el píxel en la posición (x, y) en la imagen 1, su correspondiente está en $(x + S[y, x], y)$ en la imagen 2. Si $S[y, x] < 0$, la correspondencia está hacia la izquierda; si $S[y, x] > 0$, la correspondencia está hacia la derecha; si $S[y, x] = 0$, las coordenadas de los dos puntos coinciden en ambas imágenes.

La búsqueda de correspondencias requiere ser capaz de determinar el parecido visual entre píxeles de dos imágenes. Si los píxeles m_1 y m_2 son visualmente parecidos, tienen más probabilidad de estar en correspondencia que otros que sean visualmente diferentes. Como la apariencia (el nivel de gris) de un único píxel es propensa al ruido y poco discriminativa, el elemento de puesta en correspondencia será una ventana centrada en el píxel. Dado un píxel m de una imagen, llamaremos vecindad del píxel de radio K al conjunto de píxeles de la imagen que se encuentren dentro de una ventana de tamaño $(2K + 1) \times (2K + 1)$ píxeles centrada en m . El número de píxeles de una vecindad de radio K es $N = (2K + 1)^2$. Dadas dos vecindades w_1 y w_2 de dos píxeles, el parecido visual entre ellas puede calcularse con la suma de *diferencias al cuadrado (SSD)* de cada una de sus componentes

$$d_{SSD}(\mathbf{v}, \mathbf{w}) = \sum_{i=1}^N (\mathbf{v}_i - \mathbf{w}_i)^2.$$

La distancia d_{SSD} es siempre positiva, es pequeña cuando dos ventanas son visualmente parecidas y grande en caso contrario.

Ejercicio 13. Implementa la función `C = localssd(im1, im2, K)` que calcula la suma de diferencias al cuadrado entre las ventanas de radio K de la imagen 1 y la imagen 2. El resultado debe

ser una matriz del mismo tamaño que las imágenes de entrada que contenga en cada punto el valor de d_{SSD} para la ventana de la imagen 1 y la ventana de la imagen 2 centradas en él. Es decir, $C[i, j]$ debe ser el resultado de d_{SSD} para las ventanas centradas en $im1[i, j]$ e $im2[i, j]$.

Para este ejercicio puede resultar útil la función `scipy.ndimage.convolve`.

```
[21]: def localssd(im1, im2, K):
        """
        The local sum of squared differences between windows of two images.

        The size of each window is (2K+1)x(2K+1).
        """
        ventana = np.ones((2*K+1, 2*K+1))
        d_SSD = (im1-im2)**2
        return scnd.convolve(d_SSD, ventana, mode='constant', cval=0.0)

print(localssd(O1, O2, 5).shape)
print(O1.shape)
print(O2.shape)
```

(656, 495)

(656, 495)

(656, 495)

Ejercicio 14. Implementa la función `D = ssd_volume(im1, im2, disps, K)` que calcula la suma de diferencias al cuadrado entre las ventanas de la imagen `im1` y la imagen `im2` desplazada horizontalmente. El parámetro `disps` debe ser una lista de valores indicando las disparidades que se usarán para desplazar la imagen `im2`. Por ejemplo, si `disps` es `np.arange(-3,2)`, se llamará 5 veces a `localssd` para la imagen 1 y la imagen 2 desplazada -3 , -2 , -1 , 0 y 1 píxeles en sentido horizontal. `K` es el parámetro que indica el radio de las ventanas usado por `localssd`.

El valor devuelto `D` será un array de tamaño $M \times N \times L$, donde `L` es el número de disparidades indicadas por `disps`, `L = len(disps)` (es decir, el número de veces que se ha llamado a `localssd`); `M` y `N` son, respectivamente, el número de filas y de columnas de las imágenes de entrada. El elemento `D[y,x,l]` debe ser la SSD entre la ventana centrada en `im1[y,x]` y la ventana centrada en `im2[y,x + disps[l]]`.

`D[y,x,l]` debe ser muy grande para aquellos valores en los que `im2[y,x + disps[l]]` no esté definido, es decir, el índice `(y,x+disps[l])` se sale de la imagen 2.

```
[23]: def ssd_volume(im1, im2, disps, K):
        """
        Calcula el volumen de disparidades SSD
        """
        L = len(disps)
        D = np.zeros((im1.shape[0], im1.shape[1], L))

        for i in range(L):
            d = disps[i]
```

```

    # desplazamos la imagen horizontalmente
    im3 = np.ones((im2.shape))*np.inf

    if d<0: #por la izq
        im3[:, -d:im2.shape[1]]=im2[:, 0:im2.shape[1]+d]

    else: #por la derecha
        im3[:, 0:im2.shape[1]-d]=im2[:, 0+d:im2.shape[1]]

    # calculamos D con la nueva imagen desplazada
    D[:, :, i] = localssd(im1, im3, K)
return D

```

Ejercicio 15. El conjunto de disparidades `disps` debe ser lo más pequeño posible, para mejorar el rendimiento de la optimización. Determina un procedimiento para estimar manualmente el conjunto de disparidades posibles y aplícalo a las imágenes O1 y O2.

```

[173]: # cerramos las posibles figuras anteriores
%matplotlib tk
ppl.close()
# buscamos puntos en las nuevas imagenes rectificadas
pt1, pt2 = misc.askpoints(O1,O2)

# restamos ambas
p = pt1-pt2

# redondeamos y los pasamos a enteros
puntos = np.around(p[0],decimals=0)

# escogemos los números entre el mínimo y el máximo y los pasamos a enteros
disps = np.arange(int(min(puntos)), int(max(puntos)))
print(disps)

```

```

[139. 165. 152. 155. 187. 155.]
[139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156
 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174
 175 176 177 178 179 180 181 182 183 184 185 186]

```

Aplica la función `ssd_volume` al par de imágenes O1 y O2 con las disparidades estimadas en el ejercicio anterior.

```

[184]: D = ssd_volume(O2, O1, disps, 5)

# to speed-up the optimization ahead, discard the par of the image showing only
↳ background

D.shape

```

[184]: (656, 495, 48)

1.8.2 4.1 Estimación de la disparidad sin regularizar

La matriz D calculada en el ejercicio anterior proporciona los costes unitarios D_i de una función de energía sin regularización de la forma

$$E(x) = \sum_i D_i(x_i),$$

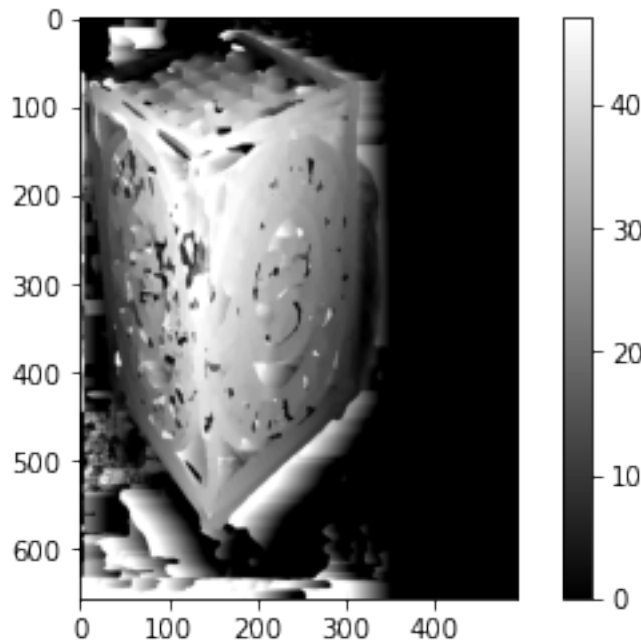
donde $D_i(l)$ viene dado por $D[y, x, l]$, suponiendo que el píxel i tenga coordenadas (x, y) . Las variables $x = (x_1, \dots, x_{NM})$ indican las etiquetas de cada uno de los píxeles. En este caso, las etiquetas son los índices del array `disps`, que a su vez son las disparidades horizontales. Por eso, a partir de aquí se hablará indistintamente de etiquetas y disparidades. Sólo es necesario recordar que la etiqueta l está asociada a la disparidad `disps[l]`.

Minimizando la energía $x = \arg \min_x E(x)$, se obtiene un vector de etiquetas óptimo x^* que indica, para cada píxel, cuál es su disparidad horizontal entre las dos imágenes.

Ejercicio 16. Minimiza $E(x)$ y muestra las disparidades resultantes.

```
[190]: res = np.argmin(D,axis=2)
      %matplotlib inline
      ppl.imshow(res,cmap='gray')
      ppl.colorbar()
```

[190]: <matplotlib.colorbar.Colorbar at 0x7fd9a9733af0>



1.8.3 4.2 Estimación de la disparidad regularizada

El etiquetado usando exclusivamente términos unitarios es muy sensible al ruido y propenso a que aparezcan zonas de píxeles cercanos con mucha variación en las etiquetas. Esto es especialmente notable en zonas planas (es decir, sin textura) de las imágenes originales, donde no hay suficiente información para establecer una correspondencia basándose exclusivamente en la apariencia visual de ventanas pequeñas. Por eso es necesario incluir un término de suavizado o regularización en la función de energía. Los tipos de saltos de etiquetas que aparecerán en el resultado final dependerán de cómo sea ese término de suavizado.

La función de energía que utilizaremos para calcular las disparidades en la práctica será el resultado de añadir a la expresión (6) un término que penalice los cambios de disparidad en los píxeles vecinos:

$$E_r(x) = \sum_i D_i(x_i) + \lambda \sum_{ij} \min(k, |x_i - x_j|).$$

Siendo j los índices de los píxeles vecinos del i en la imagen. La solución al problema de la correspondencia vendrá dado por el conjunto de etiquetas (disparidades) de los píxeles de la imagen que minimicen $E_r(x)$.

En [Yuri Boykov, Olga Veksler, and Ramin Zabih. “Fast approximate energy minimization via graph cuts”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:1222–1239, 2001.] se presentan métodos para resolver algunos problemas de optimización con varias etiquetas empleando algoritmos de cortes de grafos. Es recomendable repasar las secciones 5 y 8.

Ejercicio 17. Escribe la función `find_corresp_aexpansion(D, initLabels, lmb, maxV)`, que recibe un volumen `ssd`, `D`, un conjunto inicial de etiquetas, `initLabels`, que puede ser el obtenido en el ejercicio 5, el valor de la constante λ , y el valor máximo de la función de coste $|x_i - x_j|$, que tendrás que establecer empíricamente. El resultado de esta función serán las etiquetas que minimizan $E_r(x)$. Para ello debes utilizar la función `maxflow.fastmin.aexpansion_grid(D, V, max_cycles=None, labels=None)` del paquete *PyMaxFlow*, que resuelve el problema anterior mediante un algoritmo de cortes de grafos empleando una α -expansión.

```
[191]: def find_corresp_aexpansion(D, initialLabels, lmb, maxV):
        L = D.shape[2]
        V = np.zeros([L,L])
        for i in range(L):
            for j in range(L):
                V[i, j] = min(lmb * np.abs(i - j), maxV)

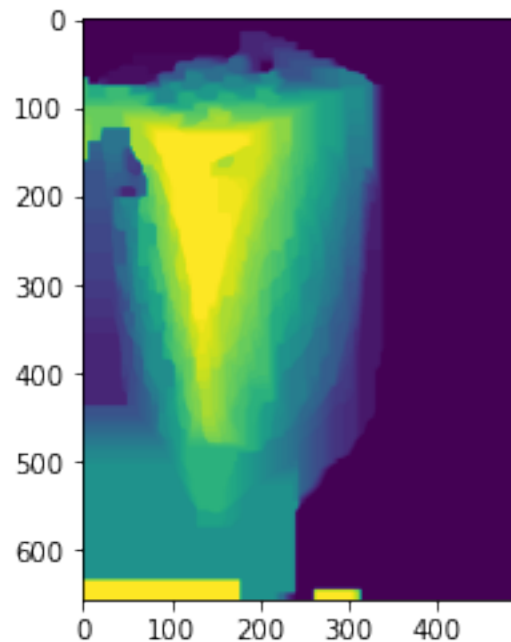
        return maxflow.fastmin.aexpansion_grid(D, V, max_cycles=None,
        ↪ labels=initialLabels)
```

Llama a esta función y muestra una figura con las etiquetas que resulten de la minimización de la energía para el volumen `ssd` `D` (este proceso puede durar varios minutos).

```
[192]: labels = find_corresp_aexpansion(D, res, 1.8 , 27)
```

```
[193]: %matplotlib inline
        ppl.imshow(labels)
```

[193]: <matplotlib.image.AxesImage at 0x7fd9a992a220>



La matriz de etiquetas óptimas X obtenida de la minimización de la función de energía puede transformarse en la matriz de disparidades S indexando en cada una de sus celdas el array de disparidades `disps` $S = \text{disps}[X]$. Ahora, el píxel de coordenadas (x, y) de la primera imagen rectificada tendrá su correspondencia en el píxel de coordenadas $(x + S[y, x], y)$ de la segunda imagen rectificada.

El siguiente ejercicio usa la matriz de disparidades para establecer automáticamente las correspondencias entre un par de imágenes sin rectificar.

Ejercicio 18. Implementa la función `plot_correspondences(image1, image2, S, H1, H2)` que, dado un par de imágenes sin rectificar, la matriz de disparidades entre las imágenes rectificadas y las homografías que llevan de las imágenes sin rectificar a las imágenes rectificadas, pida al usuario puntos en la primera imagen y dibuje sus correspondencias en la segunda.

```
[194]: def plot_correspondences(image1, image2, S, H1, H2):  
    """  
    Ask for points in the first image and plot their correspondences in  
    the second image.  
  
    Parameters  
    -----  
    image1, image2 : array_like  
        The images (before rectification)  
    S : array_like  
        The matrix of disparities.
```

```

H1, H2 : array_like
    The homographies which rectify both images.
"""
# Prepare the two images.
fig = plt.gcf()
fig.clf()
ax1 = fig.add_subplot(1, 2, 1)
ax1.imshow(image1)
ax1.axis('image')
ax2 = fig.add_subplot(1, 2, 2)
ax2.imshow(image2)
ax2.axis('image')
plt.draw()

ax1.set_xlabel("Choose points in left image (or right click to end)")
point = plt.ginput(1, timeout=-1, show_clicks=False, mouse_pop=2,
→mouse_stop=3)
while len(point) != 0:
    # point has the coordinates of the selected point in the first image.
    point = np.c_[np.array(point), 1].T
    ax1.plot(point[0,:], point[1:], '.r')

    # TODO: Determine the correspondence of 'point' in the second image.
    # perhaps you have to swap the image co-ordinates.

    # Aplicamos H1 y pasamos a coordenadas cartesianas
    punto = np.dot(H1, point)
    punto = punto/punto[2]
    # redondeamos y pasamos a entero para usar x, y como índices
    indice = np.around(punto)
    disparidad = S[(int(indice[1]),int(indice[0]))]
    # calculamos x en la segunda imagen como  $x = x + S(y, x)$ 
    punto[0] = punto[0] - disparidad

    # Aplicamos H2 y pasamos a coordenadas cartesianas
    punto = np.dot(npla.inv(H2), punto)
    punto = (punto / punto[2])

    # TODO: Plot the correspondence with ax2.plot.

    ax2.plot(punto[0], punto[1], '.r')

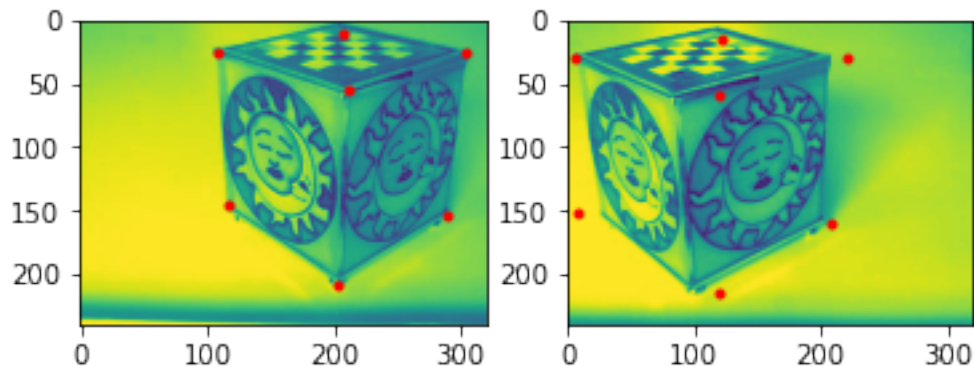
    plt.draw()
    # Ask for a new point.
    point = plt.ginput(1, timeout=-1, show_clicks=False, mouse_pop=2,
→mouse_stop=3)

```

```
ax1.set_xlabel('')
ppl.draw()
```

```
[195]: %matplotlib tk
ppl.close()
S = disps[labels]
plot_correspondences(img1,img2,S,H1,H2)
```

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "/Users/alejandro/opt/anaconda3/lib/python3.8/tkinter/__init__.py", line 1883, in __call__
    return self.func(*args)
  File "/Users/alejandro/opt/anaconda3/lib/python3.8/tkinter/__init__.py", line 804, in callit
    func(*args)
  File "/Users/alejandro/opt/anaconda3/lib/python3.8/site-packages/matplotlib/backends/_backend_tk.py", line 253, in idle_draw
    self.draw()
  File "/Users/alejandro/opt/anaconda3/lib/python3.8/site-packages/matplotlib/backends/backend_tkagg.py", line 9, in draw
    super(FigureCanvasTkAgg, self).draw()
  File "/Users/alejandro/opt/anaconda3/lib/python3.8/site-packages/matplotlib/backends/backend_agg.py", line 407, in draw
    self.figure.draw(self.renderer)
  File "/Users/alejandro/opt/anaconda3/lib/python3.8/site-packages/matplotlib/backend_bases.py", line 1544, in _draw
    def _draw(renderer): raise Done(renderer)
matplotlib.backend_bases._get_renderer.<locals>.Done:
<matplotlib.backends.backend_agg.RendererAgg object at 0x7fd9a94b1070>
```



```
[ ]:
```