Visión por computador. Práctica 3.

José Carlos Martínez Velázquez

12 de Diciembre de 2016

Contents

Estimación de la matriz de una cámara a partir del conjunto de puntos en correspondencias

a) Generar la matriz de una cámara finita P a partir de valores aleatorios en [0,1] . Verificar si representa una cámara finita y en ese caso quedársela.

En primer lugar debemos saber las propiedades de una cámara. Una matriz P es cámara, si sus dimensiones son 3×4 y además la submatriz M de tamaño 3×3 más a la izquierda tiene determinante distinto de cero, es decir, es invertible. Entonces, podemos denotar una cámara como sigue [1]:

$$P = [M_{3\times3}| - MC_{3\times1}]$$

Generalmente, C es un vector que representa la posición de la cámara. Existe una excepción por la cual una matriz donde $det(M) = 0$ puede ser cámara y es aquélla donde la última fila de la cámara es igual a $(0, 0, 0, 1)$. Este tipo de cámara se denomina cámara afín. Dado que se pide que generemos una cámara con valores aleatorios, debido a la poca probabilidad de este suceso consideraremos sólo el caso en el que el determinante de M sea distinto de 0. En Python, podríamos obtener una cámara, asegurándonos que es cámara, del siguiente modo:

```
"""
La funcion generateCamera devuelve una matriz 3x4
con determinante distinto de cero, esto es, una
cámara finita.
"""
def generateCamera():
    camera=np.random.rand(3,4)
    while(np.linalg.det(camera[0:3,0:3])==0):
        camera=np.random.rand(3,4)
    return camera
```

b) Suponer un patrón de puntos del mundo 3D compuesto por el conjunto de puntos con coordenadas $\{(0, x1, x2)\ y\ (x2, x1, 0),\ para\ x1 = 0.1 : 0.1 : 1\ y\ x2 = 0.1 : 0.1 : 1\}$. Esto supone una rejilla de puntos en dos planos distintos ortogonales

Generaremos una rejilla de puntos entre con coordenadas entre 0.1 y 1 en los planos $X = 0$ y $Z = 0$. Para entender lo que se pretende, procuraré ilustrarlo con la siguiente figura:

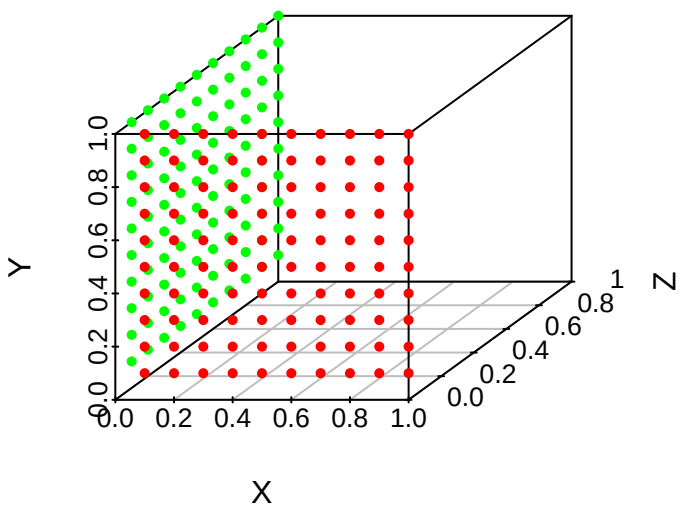


Figura 1. Grid de puntos en planos ortogonales.

Esta es la escena en el espacio con la que trabajaremos en los sucesivos apartados.

Las coordenadas del estos puntos pueden ser generados con la siguiente función:

```
"""
La funcion generatePointGrid genera una rejilla
de puntos en dos planos distintos ortogonales.
"""
def generatePointGrid():
    dec_vals=np.arange(0.1,1.1,0.1)
    grid=[]
    for x_1 in dec_vals:
        for x_2 in dec_vals:
            grid.append((0,x_1,x_2))
            grid.append((x_2,x_1,0))
    return grid
```

c) Proyectar el conjunto de puntos del mundo con la cámara simulada y obtener las coordenadas píxel de su proyección.

Dado que la cámara es una matriz $P_{3\times4}$, necesitamos que los puntos del mundo estén en coordenadas homogéneas. Entonces, dado un punto del mundo en coordenadas $Q^T_{mundo} = (X, Y, Z)$, pasa a convertirse en $Q^T_{mundo} = (X, Y, Z, 1)$. Entonces, para conseguir un punto proyectado con la cámara tenemos que hacer la siguiente operación:

$$\underbrace{\left(\begin{array}{ccc|c} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{array}\right)}_{Cámara} \cdot \underbrace{\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}}_{Q_{mundo}} = \underbrace{\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}}_{q_{proyectado}}$$

Se supone que un punto proyectado en un plano debe tener dos coordenadas, pero recordemos que estamos trabajando en coordenadas homogéneas, entonces, para obtener el punto proyectado en coordenadas píxeles, nos queda que dividir por la coordenada z', para obtener el 1 de las coordenadas homogéneas:

$$q_{mundo} = z' \cdot q_{pixel} \Rightarrow \frac{q_{mundo}}{z'} = q_{pixel} \Rightarrow q_{pixel} = \begin{pmatrix} \frac{x'}{z'} \\ \frac{y'}{z'} \\ 1 \end{pmatrix}$$

Así es cómo se consiguen las coordenadas píxeles. La función que realiza esta tarea es la siguiente:

```
"""
La funcion projectPointsPixelCoords, recibe la
matriz de camara y puntos en formato mundo.
Convierte los puntos a 1x4 y realiza la multiplicacion
Px. La tercera coordenada siempre es 1, luego las
coordenadas x e y quedan divididas por una constante
(la coordenada z proyectada).
"""
def projectPoints(camera,world_points):
    #Añado una coordenada más a los puntos para convertirlos en 1x4
    proj_points=np.array([[(x,y,z,1)] for (x,y,z) in world_points])
    #Realizo el producto (P*x^T)^T para cada punto
    proj_points=[np.transpose(np.dot(camera,np.transpose(point))) for point in proj_points]
    #Vuelvo a convertir a lista de tuplas en coordenadas píxel.
    #Para ello divido las coordenadas x e y por la tercera coordenada
    #y a continuacion me quedo con las dos primeras coordenadas.
    proj_points=[(point[0][0]/point[0][2],point[0][1]/point[0][2],1) for point in proj_points]
    return proj_points
```

Para poder visualizar los resultados, vamos a implementar una función que nos permita dibujar las "fotografías" realizadas. En primer lugar vamos a calcular un lienzo en función de los puntos que tenemos, los márgenes que queremos y lo alejada que queremos la escena. Esta función es la siguiente:

```
def giveCanvas(proj_points,marginw,marginh,scaling):
    max_value=np.max(proj_points)
    wcanvas=int((max_value*scaling)+(2*marginw))
    hcanvas=int((max_value*scaling)+(2*marginh))
    shapecanvas=np.zeros((hcanvas,wcanvas),dtype=np.uint8)
    image=cv2.merge([shapecanvas,shapecanvas,shapecanvas])
    return image
```

A continuación, obtendremos un lienzo con la función anterior y pintaremos todos los puntos. La función es la siguiente:

```
"""
La funcion drawProjectedGrid devuelve una imagen
con los puntos proyectados del el espacio.
"""
def drawProjectedGrid(proj_points,marginw,marginh,scaling):
    image=giveCanvas(proj_points,marginw,marginh,scaling)

    for i in xrange(1,len(proj_points),2):
        cv2.circle(
```

```
        image,
        (
          int((proj_points[i][0]*scaling)+(marginw)) ,
          int( hcanvas-(proj_points[i][1]*scaling+marginh))
        )
        , 1, (255,0,0), 2
      )

  for i in xrange(0,len(proj_points)-1,2):
  cv2.circle(
        image,
        (
          int((proj_points[i][0]*scaling)+(marginw)) ,
          int( hcanvas-(proj_points[i][1]*scaling+marginh))
        )
        , 1, (0,255,0), 2
      )

  return image
```

En primer lugar, dibujaremos los puntos vistos desde el sentido positivo de los ejes, vistos de frente (alzado). Para realizar esto, basta con ignorar la tercera coordenada (perdemos la profundidad). En la segunda imagen, dibujaremos esta misma escena proyectada con la cámara generada aleatoriamente. El resultado es el siguiente:

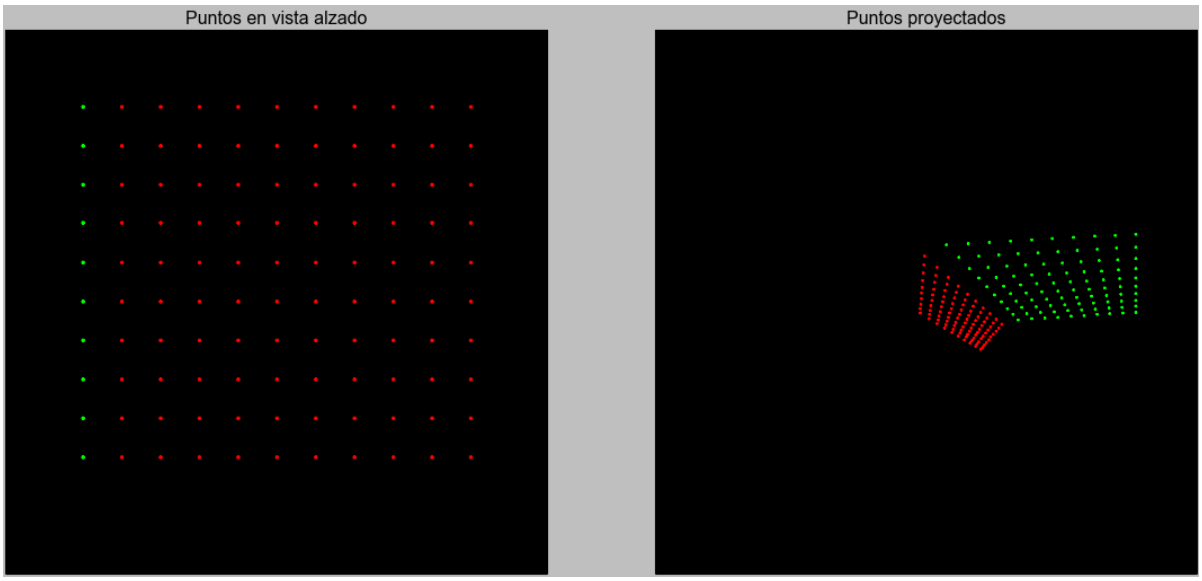


Figura 2. Puntos proyectados con una cámara aleatoria.

d) Implementar el algoritmo DLT para estimar la cámara P a partir de los puntos 3D y sus proyecciones en la imagen.

La teoría que implementa el algoritmo DLT es extensa y compleja, por ello iremos a lo básico. El algoritmo DLT trata de estimar una cámara a partir de un conjunto de puntos de mundo y puntos proyectados que se corresponden entre sí. Entonces, lo primero que necesitamos para estimar la cámara es dos conjuntos de puntos.

Por cada pareja de puntos en correspondencia $Q_{i_{mundo}} \leftrightarrow q_{i_{proyectado}}$ (en coordenadas homogéneas), se debe calcular la matriz A_i como sigue:

$$A_i = \begin{pmatrix} Q^T_{i_{mundo}} & 0^T & -x'i * Q^T_{i_{mundo}} \\ 0^T & Q^T_{i_{mundo}} & -y'i * Q^T_{i_{mundo}} \end{pmatrix} =$$

$$= \begin{pmatrix} X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x'_i X_i & -x'_i Y_i & -x'_i Z_i & -x'_i \\ 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 & -y'_i X_i & -y'_i Y_i & -y'_i Z_i & -y'_i \end{pmatrix}$$

Con cada una de estas matrices, construimos una matriz A, que engloba la información de todos los puntos en correspondencia:

$$A = \begin{pmatrix} A_{1_1} \\ A_{1_2} \\ \vdots \\ A_{i_1} \\ A_{i_2} \\ \vdots \\ A_{n_1} \\ A_{n_2} \end{pmatrix}$$

El objetivo es resolver el sistema siguiente:

$$A \cdot P = 0 \Rightarrow \begin{pmatrix} A_{1_1} \\ A_{1_2} \\ \vdots \\ A_{i_1} \\ A_{i_2} \\ \vdots \\ A_{n_1} \\ A_{n_2} \end{pmatrix} \cdot \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{pmatrix} = 0$$

Si nos fijamos, por cada pareja de puntos en correspondencia, tenemos dos filas de A. Para resolver el sistema, se necesita que A tenga, al menos, 12 filas linealmente independientes, luego necesitamos, al menos, 6 parejas de puntos en correspondencia diferentes.

Si cumplimos los requisitos hasta ahora, plantear el sistema es verdaderamente sencillo, otra cosa es resolverlo. La teoría dice que podemos obtener una solución aproximada (que no exacta) siguiendo los siguientes pasos:

1. Descomponer A en valores singulares: $A = UDV^T$
2. Obtener la columna donde se encuentra el elemento más pequeño de la matriz D (diagonal).
3. La matriz $P_{12\times 1}$ es la columna de V (que no de V^T) correspondiente a la columna donde se encuentra el elemento más pequeño de D.
4. Reconstruímos la matriz P agrupando los elementos como filas de cuatro en cuatro. Es decir, la fila 1 de P serán los elementos $1, 2, 3, 4$ de la columna de V obtenida y así sucesivamente.

Para programar el algoritmo, vamos a dividirlo en dos funciones. La primera calculará la matriz A a partir de dos conjuntos de puntos en correspondencias. El código es el siguiente:

```
"""
La función calculate_A_matrix calcula
la matriz A del algoritmo DLT a partir
de dos conjuntos de puntos en correspondencias.
Por cada pareja de puntos se calcula una matriz
Ai 2x9 que se compone de:
Pmundoi         0^T                 -xi*Pmundoi
0^T             Pmundoi             -yi*Pmundoi
La matriz A definitiva estará compuesta por
todas las filas de las matrices Ai, es decir,
si tenemos n parejas de puntos en correspondencias,
entonces la matriz A definitiva tendrá tamaño 2nx9
"""
def calculate_A_matrix(world_points,projected_points):
    A=[]
    npoints=len(world_points)
    if(npoints==len(projected_points) and npoints>=6):
        for i in xrange(npoints):
            A.append([
                        world_points[i][0],                             #Xi
                        world_points[i][1],                             #Yi
                        world_points[i][2],                             #Zi
                        1.0,                                                #1
                        0.0,                                                #0
                        0.0,                                                #0
                        0.0,                                                #0
                        0.0,                                                #0
                        -projected_points[i][0]*world_points[i][0], #-xi*Xi
                        -projected_points[i][0]*world_points[i][1], #-xi*Yi
                        -projected_points[i][0]*world_points[i][2], #-xi*Zi
                        -projected_points[i][0]                         #-xi
                    ])
            A.append([
                        0.0,                                                #0
                        0.0,                                                #0
                        0.0,                                                #0
                        0.0,                                                #0
                        world_points[i][0],                             #Xi
                        world_points[i][1],                             #Yi
                        world_points[i][2],                             #Zi
                        1.0,                                                #1
                        -projected_points[i][1]*world_points[i][0], #-yi*Xi
                        -projected_points[i][1]*world_points[i][1], #-yi*Yi
                        -projected_points[i][1]*world_points[i][2], #-yi*Zi
                        -projected_points[i][1]                         #-yi
                    ])
    else:
        raise ValueError,
        "Must be at least 6 points in correspondence and both sets must have the same lenght."
    return np.array(A)
```

La segunda función es la que estima la cámara. Hace uso de la anterior función, para obtener A, la descompone, calcula el menor elemento de D y reagrupa los términos. El código es el siguiente:

```
"""
Dados dos conjuntos de puntos en correspondencia,
donde conj1[i] esta en correspondencia con conj2[i],
la funcion estimate_camera estima una cámara que
hace corresponder estos puntos.
"""
def estimate_camera(world_points, projected_points):
    A=calculate_A_matrix(world_points, projected_points)
    Dcv,Ucv,Vtcv=cv2.SVDecomp(A)
    column_min_D=np.where(Dcv == np.min(Dcv))[0][0]
    P=Vtcv[column_min_D].reshape(3,4)
    return P
```

e) Calcular el error de la estimación usando la norma de Frobenius (cuadrática)

La norma de Frobenius de una matriz A se calcula como sigue:

$$||A||_F = \sqrt{\sum_{i=1}^{nfilas} \sum_{j=1}^{ncols} a_{ij}^2}$$

Es lógico pensar la diferencia entre la cámara real y la estimada y obtenemos la norma de Frobenius, obtengamos el error, pero estaríamos cometiendo un error grave. Una propiedad de las cámaras es que se pueden agrupar en clases de equivalencia, entonces, la cámara real y la cámara estimada podrían estar compuestas de números diferentes pero ser, al fin y al cabo, la misma cámara. En otras palabras, podría ocurrir que la cámara generada fuese la cámara estimada multiplicada por una constante. Si esto ocurriese y calculásemos el error como comentaba anteriormente, los resultados estarían contaminados por dicha constante. Entonces, antes de calcular el error, deberíamos normalizar ambas matrices en un determinado rango, por ejemplo, entre 0 y 1. Opcionalmente, podríamos calcular el valor absoluto de ambas para trabajar con números positivos (una cámara con todos sus componentes en negativo está en la misma clase de equivalencia ella misma multiplicada por una constante, concretamente -1). Al restar ambas matrices **normalizadas**, obtendremos una matriz cuya norma de Frobenius nos dará el error correcto. Veamos una explicación más detallada de lo que quiero expresar.

Supogamos que la cámara real es:

$$P = \left(\begin{array}{ccc|c} \lambda a & \lambda b & \lambda c & \lambda d \\ \lambda e & \lambda f & \lambda g & \lambda h \\ \lambda i & \lambda j & \lambda k & \lambda l \end{array}\right)$$

Donde λ es una constante y que la cámara estimada es

$$P_{est} = \left(\begin{array}{ccc|c} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}\right)$$

Suponiendo que P y P_{est} sean exactamente la misma cámara (están en la misma clase de equivalencia), deberíamos obtener error 0, pero estamos obteniendo un error:

$$\begin{aligned} || \, |P - P_{est}| \, ||_F &= \left|\left((\lambda a - a)^2 + (\lambda b - b)^2 + ... + (\lambda l - l)^2\right)\right| \\ &= (\lambda - 1)^2 \cdot (a^2 + b^2 + ... + l^2) \end{aligned}$$

Sin embargo, si normalizamos en un determinado rango obtendríamos:

$$normalizada(P) = \left(\begin{array}{ccc|c} a_{norm} & b_{norm} & c_{norm} & d_{norm} \\ e_{norm} & f_{norm} & g_{norm} & h_{norm} \\ i_{norm} & j_{norm} & k_{norm} & l_{norm} \end{array}\right)$$

$$normalizada(P_{est}) = \left(\begin{array}{ccc|c} a_{norm} & b_{norm} & c_{norm} & d_{norm} \\ e_{norm} & f_{norm} & g_{norm} & h_{norm} \\ i_{norm} & j_{norm} & k_{norm} & l_{norm} \end{array}\right)$$

$$|| \ |P - P_{est}| \ ||_F = \left|\left((a_{norm} - a_{norm})^2 + (b_{norm} - b_{norm})^2 + ... + (l_{norm} - l_{norm})^2\right)\right| = 0$$

En estas condiciones, como vemos, podemos valorar el error obtenido correctamente. Entonces, si el error sale muy cercano a 0 (lo ideal sería exactamente 0 pero en informática el 0 exacto no existe), hemos estimado la cámara correctamente. En concreto, obtengo un error del orden de magnitud de 10^{-15}.

Las funciones que normalizan una matriz y que calculan su norma de Frobenius son las siguientes:

```
"""
Normaliza una matriz en un rango determinado.
Es necesario para igualar en rango al calcular
el error de dos matrices.
"""
def normalizeMatrix(M,min_d,max_d):
    if(min_d<max_d):
        max_v=np.max(M)
        min_v=np.min(M)
        return (((max_d - min_d)/(max_v-min_v))*(M-min_v))+min_d
    else:
        raise ValueError, "Desired min must be smaller than desired max."

"""
Calcula la norma de frobenius de una matriz.
"""
def frobenius_norm(matrix):
    return math.sqrt(np.sum(matrix*matrix))
```

Para calcular el error correctamente basta con ejecutar la siguiente línea

```
error= frobenius_norm(normalizeMatrix(np.abs(P),0,1) - normalizeMatrix(np.abs(P_estimated),0,1))
```

Fijémonos cómo trabajamos siempre con los valores absolutos. Cambiar una cámara por su valor absoluto no altera en nada el resultado, pues estaríamos multiplicando por -1 en caso de que alguna de las cámaras fuese negativa. Esto es simplementa para que la normalización se realize correctamente.

f) Mostrar en una única imagen los puntos 3D proyectados con la cámara estimada y la cámara simulada.

Vamos a comparar en una misma imagen los puntos obtenidos con la cámara generada y con la cámara estimada. Este es el resultado:

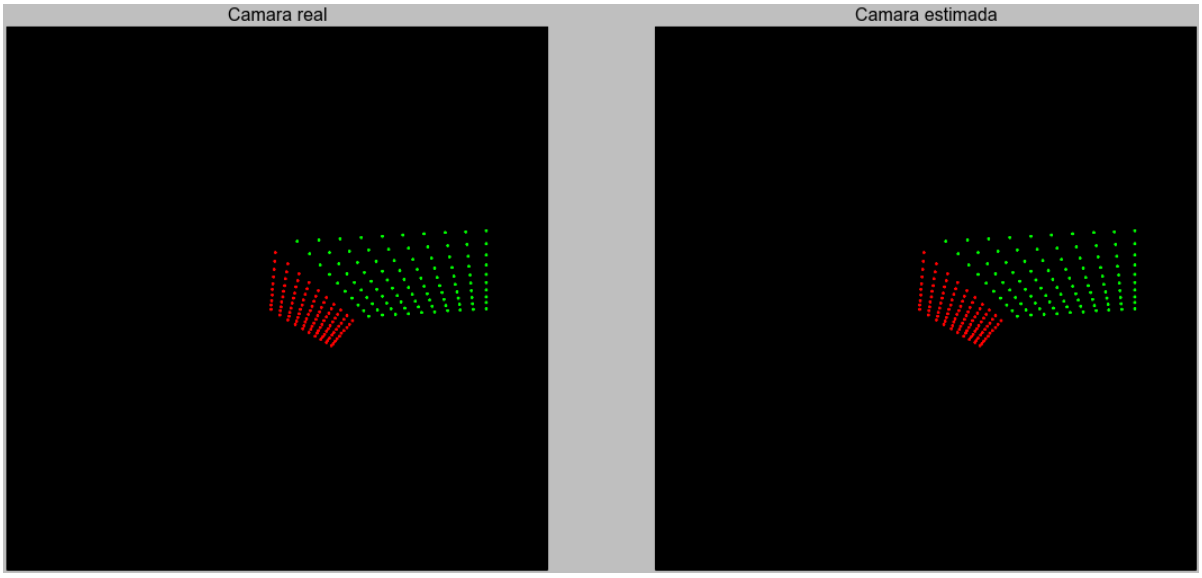


Figura 3. Cámara generada vs. cámara estimada.

Como se puede apreciar, apenas hay diferencia. Era de esperar, pues un error del orden de 10^{-15} no es apreciable por el ojo humano.

Calibración de la cámara usando homografías

a) Escribir una función que sea capaz de ir leyendo las sucesivas imágenes en chessboard.rar y determine cuáles son válidas para calibrar una cámara. Usar las 25 imágenes tiff que se incluyen en el fichero datos. Usar la función cv::findChessboardCorners(). Determinar valores precisos de las coordenadas de las esquinas presentes en las imágenes seleccionadas usando cv::cornerSubpix().

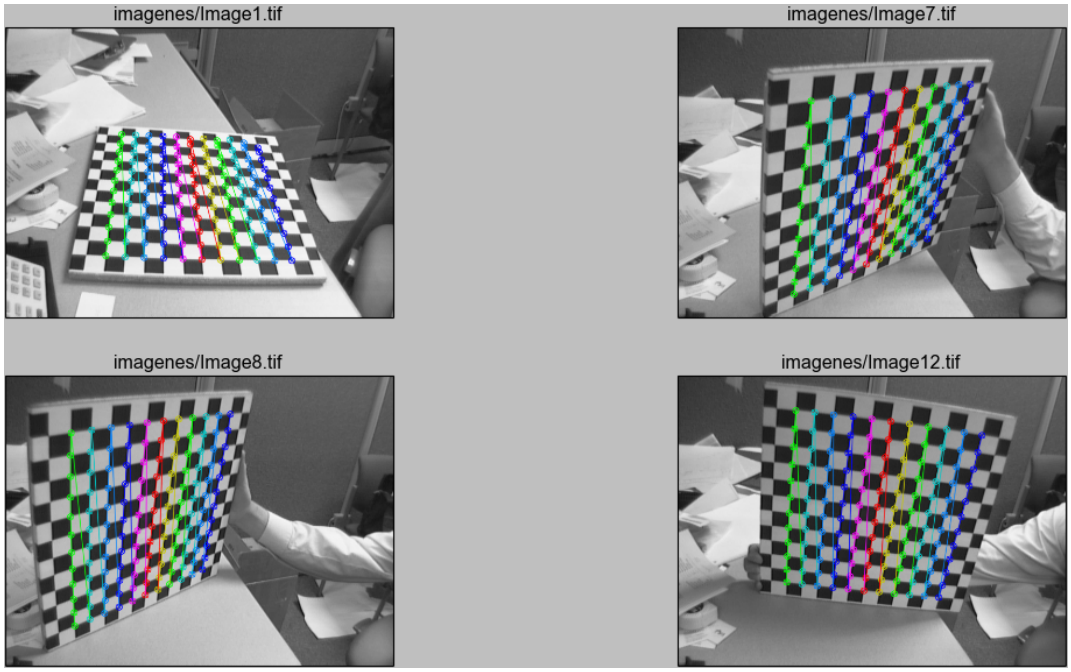
La función `findChessboardCorners()` tratará de encontrar un patrón del estilo del tablero de ajedrez en las imágenes que procesemos. Para que una imagen sea válida, se requiere, evidentemente, que la función sea capaz de encontrar el patrón en dicha imagen. El tablero que aparece en las distintas imágenes es de 13×12 cuadrados iguales (los que no son iguales, no cuentan), que coincide con el número de esquinas interiores. Las esquinas interiores son aquéllas que `findChessboardCorners()` va a detectar. Para ser un poco más flexibles, vamos a permitir que el patrón que encuentre sea de un cuadro menos de alto y un cuadro menos de ancho, es decir, el patrón que vamos a buscar es 12×11.

Cuando pasamos una imagen a `findChessboardCorners()` con el tamaño del patrón, vamos a obtener dos cosas: la primera es si la imagen es válida para calibrar, esto es, si fue posible encontrar el patrón en la imagen. La segunda es, en caso de que la imagen sea válida, la lista de puntos que conforman el patrón indicado en la imagen. A partir de aquí no queda más que llamar a la función `cornerSubPix()` con dicha lista de puntos y un determinado tamaño de ventana que nos permita refinar los puntos en un espacio acotado, usualmente 11×11. Hecho todo esto, no tenemos más que pintar en la imagen que resultó válida el patrón encontrado con `drawChessboardCorners()`. Para poder visualizar los patrones encontrados, guardo en una lista todas las imágenes que resultaron válidas. El código que realiza todo el proceso descrito es el siguiente:

```
"""
La funcion selectImages recibe una
lista de rutas de imagenes, un tamaño de tablero
a buscar y un tamaño de ventana de refinamiento.
Con estos datos carga y selecciona las imagenes
válidas para calibrar.
"""
def selectImages(arrayNames,size_corners,size_refine):
    #Me creo los flags con los valores de la documentacion de OpenCV
    #Esto es necesario porque en Python no se pueden pasar flags
    #a la funcion findChessboardCorners()
    CV_CALIB_CB_ADAPTIVE_THRESH =   1
    CV_CALIB_CB_FILTER_QUADS    =   4
    CV_CALIB_CB_NORMALIZE_IMAGE =   2

    #Criterio de parada para todas las busquedas
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 50, 0.0001)

    # Lista que contendra las imagenes originales
    originalImages=[]
    # Lista que contendra los puntos de las esquinas encontradas
    imgpoints=[]
    # Lista que contendra los nombres de las imagenes validas
    validimgnames=[]
    # Lista que contendra las imagenes con los puntos pintados
    validimages=[]

    for name in arrayNames:
        #Cargar la imagen y convertir a escala de grises
```

```
        image=loadImage(name,"COLOR")
        image_gray=cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
        # Encontrar las esquinas
        isvalid, corners = cv2.findChessboardCorners(
          image_gray, size_corners,
          flags=CV_CALIB_CB_ADAPTIVE_THRESH or
                CV_CALIB_CB_FILTER_QUADS or
                CV_CALIB_CB_NORMALIZE_IMAGE)

        #Si la imagen es válida
        if isvalid == True:
            originalImages.append(np.copy(image))
            #Añadimos el nombre de la imagen
            validimgnames.append(name)
            #Refinamiento de los puntos encontrados
            refined_corners=cv2.cornerSubPix(image_gray,corners,size_refine,(-1,-1),criteria)
            #Añado los puntos refinados a la lista que se devolvera
            imgpoints.append(refined_corners)
            #Dibujo los puntos refinados en la imagen correspondiente
            img=cv2.drawChessboardCorners(image,size_corners,refined_corners,isvalid)
            #Añado la imagen con los puntos pintados
            validimages.append(img)
    return imgpoints,validimgnames,validimages
```

La combinación de banderas (flags) para `findChessboardCorners()` utilizada es aquélla que devolvía el máximo número de imágenes válidas, que en este caso particular resultó ser 7. Las imágenes que resultaron válidas se muestran a continuación:



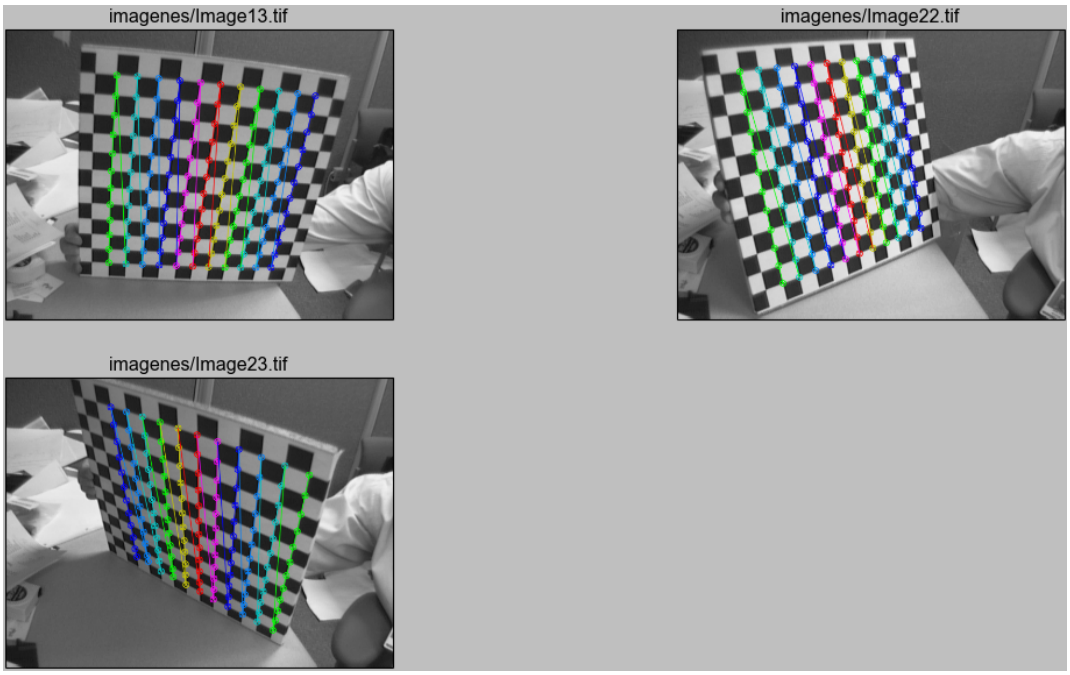


Figura 4. Imágenes válidas para calibrar.

Si quisiéramos plasmar en una sola imagen todos los puntos detectados en los distintos planos vistos por la misma cámara, esto es lo que veríamos:

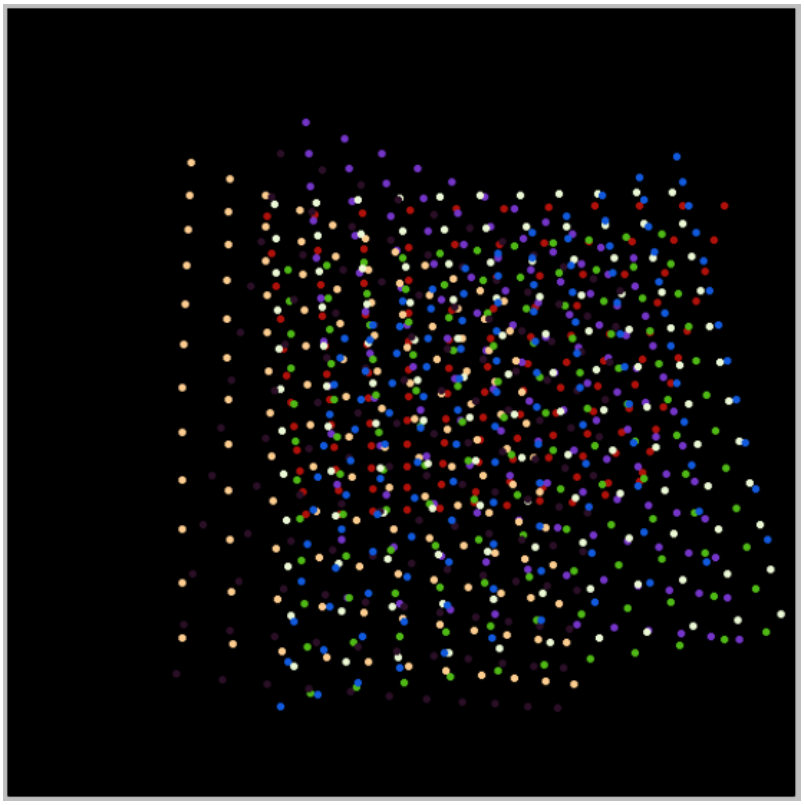


Figura 5. Todos los planos vistos desde un sólo punto de vista.

b) Usando las coordenadas de los puntos extraídos en las imágenes seleccionadas del punto anterior, calcular los valores de los parámetros intrínsecos y extrínsecos de la cámara para cada una de dichas imágenes. Usar la función cv::calibrateCamera(). Suponer dos situaciones: a) sin distorsión óptica y b) con distorsión óptica. Valorar la influencia de la distorsión óptica en la calibración y la influencia de la distorsión radial frente a la distorsión tangencial.

La calibración de la cámara se hace a partir de los datos que devuelve la función del apartado anterior, entonces, usaremos la misma función para seleccionar imágenes y calibrar, es decir, en este apartado extenderemos la función del apartado anterior para que, además de detectar las imágenes válidas, sea capaz de calibrar la cámara.

Para calibrar la cámara debemos estimar sus parámetros intrínsecos y los extrínsecos, esto es, obtener información sobre la cámara que realizó la fotografía y su movimiento. Una proyección de un punto X del mundo por una cámara P puede expresarse como sigue:

$$\lambda x = PX \Rightarrow \lambda x = K_{3\times3}[R_{3\times3}|t_{3\times1}]X \Rightarrow$$
$$P = K[r_1\ r_2\ r_3|t]$$

Donde K representa la matriz de parámetros intrínsecos. Codifica información que concierne exclusivamente a la cámara, concretamente la distancia focal y el punto de origen de coordenadas con respecto al centro de la imagen (offset). Entonces, si hacemos este análisis con varias imágenes distintas tomadas por la misma cámara, la matriz K es común a todas. La matriz K tiene la siguiente forma:

$$K = \begin{pmatrix} f_x & s & t_x \\ 0 & f_y & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Donde (f_x, f_y) codifican la distancia focal y (t_x, t_y) indican la distancia en el eje x e y desde el centro de la imagen hasta el punto de origen de coordenadas. El parámetro s trata de corregir una transformación de sesgado producida por un mal alineamiento de los receptores de fotones de la cámara. En las cámaras modernas este parámetro suele tomar el valor 0, pues las CCD de las cámaras alinean con alta precisión los píxeles.

Por otro lado, tenemos los parámetros extrínsecos: $[R|t]$. R representa los vectores de rotación y t los de translación. Estos vectores son diferentes para cada imagen, pues no dependen de la cámara, sino del entorno que percibe.

El proceso de calibración de una cámara trata de estimar K,R y t. El primer paso del proceso es estimar una homografía H entre la imagen y el plano del mundo $Z = 0$ (para lo que se necesitan puntos en correspondencia). Si estamos suponiendo que los puntos del mundo tienen coordenada $Z = 0$, entonces podemos representar la descomposición como sigue:

$$\underbrace{\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}_{3\times1} = \underbrace{K}_{3\times3} \underbrace{[r_1\ r_2\ \cancel{r_3}|t]}_{3\times4} \underbrace{\begin{pmatrix} X \\ Y \\ \cancel{0} \\ 1 \end{pmatrix}}_{4\times1} \Rightarrow$$
$$\Rightarrow \underbrace{\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}_{3\times1} = \underbrace{K[r_1\ r_2|t]}_{H_{3\times3}} \underbrace{\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}}_{3\times1}$$

Esto no significa otra cosa que perdemos la profundidad, pero no hay que alarmarse, pues puede ser recuperada haciendo el producto vectorial de $r_1 \times r_2$. Esto es porque la matriz de rotación es un triedro de vectores

ortogonales. Al perder la profundidad obtenemos una matriz 3×3 con determinante distinto de 0. Entonces ya no tenemos una cámara, sino una homografía:

$$H = \underbrace{[h_1 \ h_2 \ h_3]}_{3\times3} = \underbrace{K[r_1 \ r_2|t]}_{3\times3}$$

De donde se puede deducir que:

$$\begin{cases} h_1 = Kr_1 \\ h_2 = Kr_2 \\ h_3 = Kt \end{cases}$$

Como comentábamos anteriormente, r_1, r_2 y r_3 son ortogonales entre sí, entonces, debe cumplirse que $r_1^T r_2 = 0$, es decir, que r_1 y r_2 forman un ángulo recto. Aunque es una obviedad, también debe cumplirse que $r_1^T r_1 = 1$, es decir, que el vector de rotación r_1 es coincidente (y por ende paralelo) consigo mismo. Despejando r_1 y r_2 de las ecuaciones anteriores, tenemos que:

$$K^{-1}h_1 = r_1$$
$$K^{-1}h_2 = r_2$$

Si volvemos a las ecuaciones que proponíamos tenemos:

$$r_1^T r_2 = 0 \Rightarrow (K^{-1}h_1)^T K^{-1}h_2 = 0 \Rightarrow h_1^T K^{-T}K^{-1}h_2 = 0 \Rightarrow h_1^T B h_2 = 0$$
$$r_1^T r_1 = 1 \Rightarrow (K^{-1}h_1)^T K^{-1}h_1 = 0 \Rightarrow h_1^T K^{-T}K^{-1}h_1 = 0 \Rightarrow h_1^T B h_1 = 1$$

Donde hemos llamado $B = K^{-T}K^{-1}$.

Dado que la matriz $B_{3\times3}$ resultante es el producto de una matriz por su traspuesta, la matriz $B_{3\times3}$ es simétrica y, por ende, sólo necesitamos conocer la diagonal y tres de sus parámetros libres, esto es, 6 elementos de B. Sin entrar en detalles de cómo hacerlo, la matriz K puede ser obtenida a partir de B mediante su descomposición de Cholesky. Todo este proceso matemático es llevado a cabo por la función `calibrateCamera()` de OpenCV, que recibe los puntos en correspondencias para comenzar a *tirar del hilo*. La función del apartado anterior completamente extendida, incluyendo la calibración de cámara es la siguiente. La extensión comienza a partir de donde se comienza a comentar:

```
"""
La funcion selectImagesAndCalibrate recibe una
lista de rutas de imagenes, un tamaño de tablero
a buscar y un tamaño de ventana de refinamiento.
Con estos datos carga y selecciona las imagenes
válidas para calibrar. A continuación calibra la
cámara y devuelve las imágenes corregidas.
"""
def selectImagesAndCalibrate(arrayNames,size_corners,size_refine):
    CV_CALIB_CB_ADAPTIVE_THRESH     =   1
    CV_CALIB_CB_FILTER_QUADS        =   4
    CV_CALIB_CB_NORMALIZE_IMAGE     =   2
    CV_CALIB_ZERO_TANGENT_DIST      =   8
    CV_CALIB_FIX_K1                 =   32
    CV_CALIB_FIX_K2                 =   64
```

```
CV_CALIB_FIX_K3                 =   128
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 50, 0.0001)
originalImages=[]
imgpoints=[]
validimgnames=[]
validimages=[]
for name in arrayNames:
    image=loadImage(name,"COLOR")
    image_gray=cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
    isvalid, corners = cv2.findChessboardCorners(
      image_gray,
      size_corners,
      flags=CV_CALIB_CB_ADAPTIVE_THRESH
        or CV_CALIB_CB_FILTER_QUADS
        or CV_CALIB_CB_NORMALIZE_IMAGE )
    if isvalid == True:
        originalImages.append(np.copy(image))
        validimgnames.append(name)
        refined_corners=cv2.cornerSubPix(
            image_gray,corners,size_refine,(-1,-1),criteria)
        imgpoints.append(refined_corners)
        img=cv2.drawChessboardCorners(image,size_corners,refined_corners,isvalid)
        validimages.append(img)

#Si hay al menos tres imagenes válidas, procedemos a la calibración
if(len(validimages)>3):
    #Creo los puntos del mundo en mis unidades de medida (de uno en uno)
    objpoints=np.float32([
      np.array([
        [j,i,0.0] for i in xrange(size_corners[1])
        for j in xrange(size_corners[0])
      ],dtype=np.float32) for image in validimages])

    #Calibro la cámara en base a los puntos medidos
    #en mis unidades y los puntos de la imagen:
    #1. Suponiendo que no hay distorsion
    retNoDist,
    KNoDist,
    distortionCoeffNoDist,
    rotationVectorsNoDist,
    translationVectorsNoDist = cv2.calibrateCamera(
                  objpoints, imgpoints, image_gray.shape,None,None,
                  flags= CV_CALIB_FIX_K1
                               + CV_CALIB_FIX_K2
                               + CV_CALIB_FIX_K3
                               + CV_CALIB_ZERO_TANGENT_DIST)

    #2. Suponiendo que hay solo distorsion radial
    retOnlyRadial,
    KOnlyRadial,
    distortionCoeffOnlyRadial,
    rotationVectorsOnlyRadial,
    translationVectorsOnlyRadial = cv2.calibrateCamera(
```

```
                objpoints, imgpoints, image_gray.shape,None,None,
                                flags= CV_CALIB_ZERO_TANGENT_DIST)

    #3. Suponiendo que hay solo distorsion tangencial
    retOnlyTan,
    KOnlyTan,
    distortionCoeffOnlyTan,
    rotationVectorsOnlyTan,
    translationVectorsOnlyTan = cv2.calibrateCamera(
                objpoints, imgpoints, image_gray.shape,None,None,
              flags= CV_CALIB_FIX_K1
                                + CV_CALIB_FIX_K2
                                + CV_CALIB_FIX_K3)

    #2. Suponiendo que hay ambas distorsiones
    retDist,
    KDist,
    distortionCoeffDist,
    rotationVectorsDist,
    translationVectorsDist = cv2.calibrateCamera(
                objpoints, imgpoints, image_gray.shape,None,None)


    # A partir de aqui solo queda corregir o
    # no la distorsion y añadir las imagenes a
    # las listas correspondientes.
    correctedImagesNoDist=[]
    correctedImagesDist=[]
    croppedImagesDist=[]

    for c_img in originalImages:
        h,  w = c_img.shape[:2]

        newcameramtx, roi=cv2.getOptimalNewCameraMatrix(
             KNoDist,distortionCoeffNoDist,(w,h),1,(w,h))
        corr_image = cv2.undistort(
              c_img, KNoDist , distortionCoeffNoDist, None, newcameramtx)
        correctedImagesNoDist.append(corr_image)


        newcameramtx, roi=cv2.getOptimalNewCameraMatrix(
            KDist,distortionCoeffDist,(w,h),1,(w,h))
        corr_image_dist = cv2.undistort(
            c_img, KDist, distortionCoeffDist, None, newcameramtx)
        correctedImagesDist.append(corr_image_dist)


        croppedImagesDist.append(cropUndistordedImage(corr_image_dist))

    return  imgpoints,
            validimgnames,
            validimages,
            correctedImagesNoDist,
```

```
                correctedImagesDist,
                croppedImagesDist,
                retNoDist,retOnlyRadial,retOnlyTan,retDist



    else:
        raise IndexError, "Must be at least 3 images where pattern is detected."
```

Cuando hacemos la corrección de la distorsión aparecen zonas "vacías" así que me he creado una función que me recorta (*crop*) esas zonas vacías, buscando las zonas de cambio de color en ambos ejes. Esta función es `cropUndistortedImage()` cuyo código es el siguiente:

```
"""
La funcion cropUndistordedImage elimina las zonas
vacias de una imagen con la distorsion corregida
"""
def cropUndistordedImage(originalImage):
    # Separamos las capas y trabajamos con una sola
    r,g,b=cv2.split(originalImage)
    h, w = r.shape[0:2]
    xleft, xright, ysup, yinf = (0,0,0,0)
    # Color la parte vacia
    empty_color = 0
    # Lista de colores en las lineas verticales
    # y horizontales en la mitad
    vertical = [r[i][int(w/2)] for i in range(h)]
    horizontal = [r[int(h/2)][i] for i in range(w)]
    # Las mismas listas de antes del revés (voltea la imagen)
    vertical_rev = vertical[::-1]
    horizontal_rev = horizontal[::-1]
    # Buscamos un cambio de color en las lineas trazadas
    for i in range(2,h):
        if vertical[i] > empty_color and ysup == 0:
            ysup = i
        if vertical_rev[i] > empty_color and yinf == 0:
            yinf = i
        if ysup != 0 and yinf != 0:
            break
    for i in range(2,w):
        if horizontal[i] > empty_color and xleft == 0:
            xleft = i
        if horizontal_rev[i] > empty_color and xright == 0:
            xright = i
        if xleft != 0 and xright != 0:
            break
    #Recortamos conforme a las filas y columnas obtenidas cada capa.
    r = r[ysup:h-yinf, xleft:w-xright]
    g = g[ysup:h-yinf, xleft:w-xright]
    b = b[ysup:h-yinf, xleft:w-xright]
    #Mezclamos las capas de color ya recortadas
    cropped_img=cv2.merge([r,g,b])
    return cropped_img
```

En primer lugar vamos a analizar los errores arrojados suponiendo que hay o no hay deformaciones. La función `calibrateCamera()` calcula dichos errores mediante mínimos cuadrados. Obtenemos los siguientes resultados:

- Error en la calibración suponiendo que no hay distorsión: 1.03872258255
- Error en la calibración suponiendo sólo distorsión tangencial: 1.03452455944
- Error en la calibración suponiendo sólo distorsión radial: 0.54806944554
- Error en la calibración suponiendo que hay ambas distorsiones: 0.546933466291

Como se puede apreciar, suponiendo que no hay distorsión, obtenemos un error de aproximadamente 1.04, esto es, que cada punto se desvía por culpa de la distorsión 1.04 píxeles de media de donde debería estar. Normalmente, con un error por encima de 1 se considera que la corrección de la distorsión no tuvo éxito. Dado que estamos suponiendo que no hay distorsión y obtenemos un error superior a 1, es de suponer que hay distorsión en las imágenes, ahora hay que ver de qué tipo. Si suponemos que existe sólo distorsión tangencial, el decremento del error no es sustancial. Esto quiere decir que hay quizás un poco de distorsión tangencial pero que no afecta de forma significativa a la forma de las imágenes. Este tipo de distorsión desvía de media a cada punto de su lugar correcto aproximadamente 0.004 píxeles, algo inapreciable. Si suponemos que sólo existe distorsión radial ya sí, conseguimos mejorar significativamente el error cometido por culpa de esta distorsión y el error, al estar por debajo de 1, se considera que ha habido una calibración exitosa. Si suponemos ambos tipos de distorsión, como es de suponer, casi obtendremos el mismo error que si sólo hubiera distorsión radial, pero disminuyendo también la parte residual de distorsión tangencial que teníamos. Entonces, optaremos por corregir ambos tipos de distorsión. De entre las imágenes que resultaron válidas vamos a analizar visualmente los resultados numéricos vistos anteriormente en la siguiente figura:

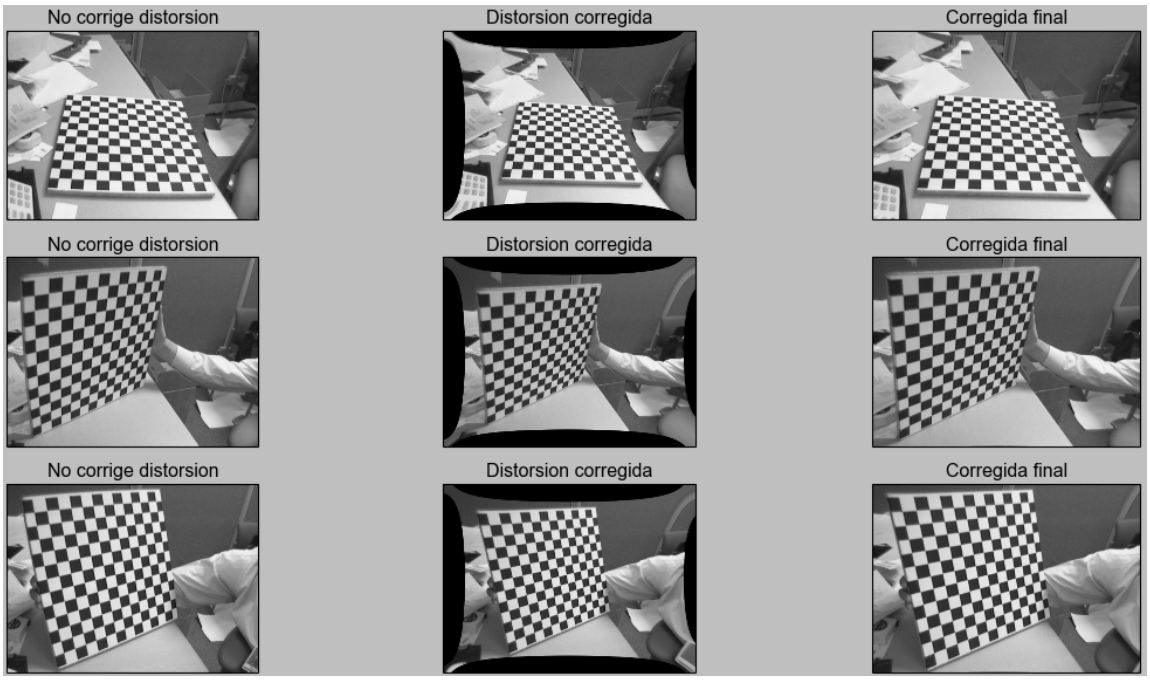


Figura 6. Corrección de la distorsión.

Como se puede apreciar, todo el peso de la corrección va centrado a corregir la distorsión radial (abarrilamiento o distorsión hacia afuera), pues la corrección, se realiza *metiendo hacia adentro* zonas de la imagen (aparecen zonas "vacías"). La corrección de la distorsión tangencial debería corregir haciendo lo contrario, con lo que no aparecerían zonas "vacías". Entonces, podemos dilucidar que las proporciones de la distorsión radial en comparación a la distorsión tangencial son muy grandes por lo que la mayor parte del esfuerzo del algoritmo se centra en corregir la distorsión radial. Esto ya quedaba claro cuando analizábamos los errores.

Estimación de la matriz fundamental F

a) Obtener puntos en correspondencias sobre las imágenes Vmort[*].pgm de forma automática usando las funciones de BRISK/ORB/AKAZE

Probaremos los tres detectores de puntos en correspondencias. La función que busca puntos en correspondencias es la siguiente:

```
"""
La funcion findCorrespondencesBRISK encuentra puntos
en correspondencia usando BRISK
"""
#def findCorrespondencesORB
#def findCorrespondencesBRISK
def findCorrespondencesAKAZE(img1,img2,distance_thr):
    #orb = cv2.ORB_create()
    #brisk = cv2.BRISK_create()
    akaze = cv2.AKAZE_create()

    kp1, des1 = akaze.detectAndCompute(img1,None)
    kp2, des2 = akaze.detectAndCompute(img2,None)

    bfdetector = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
    matches = bfdetector.knnMatch(des1,des2,k=2)

    good_matches = []
    pts1 = []
    pts2 = []
    for i,(m,n) in enumerate(matches):
        if m.distance < n.distance*distance_thr:
            good_matches.append(m)
            pts2.append(kp2[m.trainIdx].pt)
            pts1.append(kp1[m.queryIdx].pt)


    good_matches=np.array(good_matches)
    pts1=np.array(pts1)
    pts2=np.array(pts2)

    imgresult=np.zeros((1,1))
    imgresult = cv2.drawMatches(img1,kp1,img2,kp2,good_matches,imgresult,flags=2)
    return imgresult, pts1, pts2
```

Esta función indica cómo se detectan puntos con AKAZE. Las lineas comentadas indican cómo serían las funciones para los otros dos detectores. Cambian solo unas pocas linas de código. Lo importante es que necesitamos el mayor número de puntos posibles, así que utilizaremos el detector que nos devuelva más puntos.

Cabe destacar, sea cual sea el método utilizado, la función del parámetro `distance_thr`. Tal como indica Lowe en su artículo [4], rechazaremos todos los puntos cuyo ratio de distancia sea mayor a un determinado umbral, en este caso, el *número mágico* es 0.8 (`distance_thr=0.8`) que permite descartar el 90% de falsas correspondencias comprometiendo menos del 5% de correspondencias buenas.

El número de puntos detectados por cada método son los siguientes:

- Puntos encontrados por ORB: 203
- Puntos encontrados por BRISK: 2210
- Puntos encontrados por AKAZE: 1551

Como es lógico, en este caso optaremos por BRISK.

b) Calcular F por el algoritmo de los 8 puntos + RANSAC (usar un valor pequeño para el error de RANSAC)

La matriz fundamental F codifica información de la geometría epipolar. Para estimar F sin saber nada de la cámara, necesitamos correspondencias de puntos. Si las correspondencias son verdaderas, se debe cumplir que:

$$x'^T F x = 0$$

Esta relación nos está diciendo que si dos puntos están en correspondencia, para cada punto x en la primera imagen, existe una recta en la otra imagen en la que está incluido el punto correspondiente x'. Partiendo de esta ecuación, sean dos puntos en correspondencia:

$x = (a, b, 1)^T \; x' = (a', b', 1)^T$

Tenemos la ecuación siguiente:

$$\begin{pmatrix} a' & b' & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = 0 \Rightarrow$$

$$\Rightarrow (aa'f_{11} + ab'f_{21} + af_{31}) + (ba'f_{12} + bb'f_{22} + bf_{32}) + (a'f_{13} + b'f_{23} + f_{33}) = 0$$

Como se aprecia, por cada pareja de puntos obtenemos una ecuación, pero necesitamos 9 ecuaciones para resolver este sistema. Dado que F no es una homografía, debemos usar una ecuación para restringir que el determinante de F sea 0, por lo que el resto de ecuaciones necesarias se sacan de los puntos, por este motivo el algoritmo es llamado el de los 8 puntos. Los detalles de cómo escapar de que el sistema resultante es no lineal, quedan en el campo de la teoría. Por lo que a nosotros respecta, con suficientes puntos en correspodencia la función `findFundamentalMat()` de OpenCV se encarga de realizar todos estos cálculos. Entonces, para obtener la matriz fundamental basta con hacer la llamada

```
F, mask = cv2.findFundamentalMat(
  points1BRISK,points2BRISK,method=cv2.FM_RANSAC+cv2.FM_8POINT,param1=0.1)
```

Que se encargará de buscar las soluciones usando el algoritmo de los 8 puntos en combinación con RANSAC, donde el margen de distancia aceptado por RANSAC es de tan solo 0.1.

c) Dibujar las líneas epipolares sobre ambas imágenes (< 200).

En primer lugar, vamos a descartar puntos *malos* usando la máscara que nos devuelve la función `findFundamentalMat()`. Es una matriz de 0 y 1 donde un 1 indica que ese punto se punto pudo ser usado para estimar la matriz fundamental y un 0 indica que no pudo ser usado. Entonces la siguiente función se encarga de ese cometido:

```
def getFineCorrespondences(mask,points1,points2):
    return points1[mask.ravel()==1], points2[mask.ravel()==1]
```

Una vez obtenidos los puntos *buenos*, tenemos que llamar a la función `cv2.computeCorrespondEpilines()`. Hay que tener mucho cuidado con esta función pues hay que tener claro que un punto en una imagen se encuentra en algún lugar de una recta en otra imagen, por lo que tendremos que hacer una especie de llamadas cruzadas. Si tenemos dos conjuntos de puntos en correspondencias `points1` correspondientes a la imagen 1 y `points2` correspondientes a la imagen2, las siguientes llamadas son correctas:

```
lines1 = cv2.computeCorrespondEpilines(points2, 2,F).reshape(-1,3)
lines2 = cv2.computeCorrespondEpilines(points1, 1,F).reshape(-1,3)
```

En este caso, `lines1` contendrá las rectas epipolares en las que se encontrarán respectivamente los puntos en correspondencias de `points2`. Análogamente, `lines2` contendrá lo opuesto. Tal y como se han realizado las llamadas, cada variable `linesx` será una lista de elementos, tal que cada uno tiene la forma $[A, B, C]$, es decir `linesx` es una matriz de $n \times 3$. Cada elemento $[A, B, C]$ define una recta epipolar en su forma general:

$$r \equiv Ax + By + C = 0$$

Para dibujar una recta en una imagen con OpenCV, se necesitan dos puntos. Entonces, necesitamos extraer dos puntos de cada recta. Como lo que queremos es dibujar la recta cruzando la imagen, necesitamos que los valores de x sean 0 y el número de columnas respectivamente. Entonces:

$$x = 0 \Rightarrow A \cdot 0 + By + C = 0 \Rightarrow y = -\frac{C}{B} \rightarrow p_1 = (0, -\frac{C}{B})$$
$$x = nc \Rightarrow A \cdot nc + By + C = 0 \Rightarrow y = \frac{-C - A \cdot nc}{B} \rightarrow p_2 = (nc, \frac{-C - A \cdot nc}{B})$$

Siendo `nc` el número de píxeles de ancho de la imagen.

Entonces, podemos convertir cada elemento de `linesx` de la forma $[A, B, C]$ en dos puntos $[p_1, p_2]$. Esto lo haremos, desde código, así:

```
line_points1=[
  [
    (0, int(-eq[2]/eq[1]) ),
    (img1.shape[1], int(-(eq[2]+eq[0]*img1.shape[1])/eq[1] ))
  ]  for eq in lines1]
line_points2=[
  [
    (0, int(-eq[2]/eq[1]) ),
    (img2.shape[1], int(-(eq[2]+eq[0]*img1.shape[1])/eq[1] ))
  ]  for eq in lines2]
```

Ahora que ya hemos convertido cada ecuación de la recta en dos puntos, ya solo resta pintar los puntos en correspondencia y sus rectas en cada imagen. Pintaremos una de cada diez para poder comprobar que el proceso se está llevando a cabo correctamente. Para pintar en una imagen las lineas y los puntos, basta con llamar a este bucle:

```
#Con [::10] pinta solo una de cada 10
for points,point in zip(line_points1[::10],points1[::10]):
  color=tuple(np.random.randint(0,255,3).tolist())
  cv2.line(img1, points[0], points[1], color,1)
  cv2.circle(img1,(int(point[0]),int(point[1])),5,color,-1)
```

Este es el resultado:

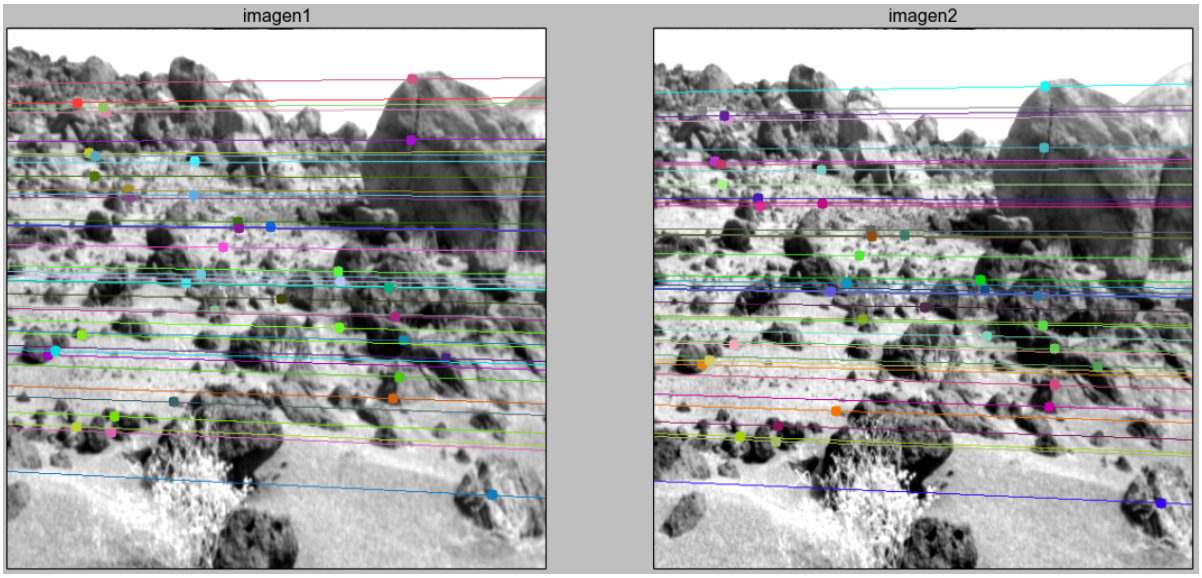


Figura 7. Rectas epipolares.

Como se puede apreciar en la figura, las lineas y puntos en correspondencia son correctos. Gracias a la similitud entre las imágenes las correspondencias encontradas se hacen muy evidentes una vez señaladas.

d) Verificar la bondad de la F estimada calculando la media de la distancia ortogonal entre los puntos soporte y sus líneas epipolares en ambas imágenes. Mostrar el valor medio del error.

Cuando tenemos puntos y rectas, el error se define en cuánto se desvia un punto de la recta que lo debería contener. En definitiva, el error de un punto con respecto a su recta no es más que la distancia de dicho punto a dicha recta. Si aplicamos la expresión de la distancia a un punto $p = (x, y)$ a una recta r, podemos obtener el error en cada imagen:

$$error = d(p, r) = \frac{|Ax + By + C|}{\sqrt{A^2 + B^2}}$$

Para obtener el error medio, basta con hacer la media de la expresión anterior por cada punto y su recta correspondiente de entre los n puntos y las n rectas:

$$error_m = \frac{1}{n}\sum_{i=1}^{n} d(p[i], r[i]) = \frac{1}{n}\sum_{i=1}^{n} \frac{|A_i x_i + B_i y_i + C_i|}{\sqrt{A_i^2 + B_i^2}}$$

Implementar el cálculo del error es tan sencillo como esto:

```
"""
Calcula el error de los puntos en correspondencia
con respecto a las lineas epipolares
"""
def calculateMeanError(points,lines):
    if(len(points)==len(lines)):
        error_sum=sum([
          abs(lines[i][0]*points[i][0]+lines[i][1]*points[i][1]+lines[i][2])
```

```
            /
            math.sqrt(lines[i][0]*lines[i][0]+lines[i][1]*lines[i][1])
        for i in xrange(len(lines))])
        return (error_sum/len(lines))
    else:
        raise IndexError, "Both lists must have the same lenght."
```

Una vez obtenidas las medias de errores en cada imagen, se suman y se dividen por dos, esto corresponde a la media de error total en el proceso completo. Los errores obtenidos son:

- Media de error en la imagen 1: 0.0497574550405
- Media de error en la imagen 2: 0.0497817082956
- Media de error en ambas imagenes: 0.0497695816681

El error obtenido es aceptable, teniendo en cuenta que la exactitud no puede ser absolutamente estricta.

Calcular el movimiento de la cámara (R,t) asociado a cada pareja de imágenes calibradas.

a) Usar las imágenes y datos de calibración dados en el fichero reconstruccion.rar

En dicho archivo encontramos tres imágenes y tres archivos que nos indican la matriz de parámetros intrínsecos K de cada imagen, los coeficientes de distorsión, la matriz de rotación R y el vector de translación t. En definitiva, nos dan la cámara que realizó cada foto. Recordemos que una cámara se puede expresar como:

$$P = K[R|t]$$

Como curiosidad, se puede ver que la matriz K es la misma en todas las imágenes, entonces, con lo que sabemos hasta ahora sólo podemos decir que todas las imágenes fueron tomadas con la misma cámara, aunque desde distintos lugares y ángulos.

Usar los datos no es más que cargar las imágenes y guardar las matrices como constantes en el código:

```
# K es comun a todas las imagenes
rdimageK=np.float32([
  [1839.6300000000001091,0,1024.2000000000000455],
  [0,1848.0699999999999363,686.5180000000000291],
  [0,0,1]
])
rdimage000=loadImage("imagenes/rdimage.000.ppm","COLOR")
rdimage000distortion=np.float32([0,0,0])
rdimage000rotation=np.float32([
  [0.99989455260937387671, 0.004939590228957159 4346, 0.013655918514328903648],
  [-0.004862100000000002758, 0.99997192395009415478, -0.005701867688477964 0954],
  [-0.0136837000000000003, 0.005634870000000000 1564, 0.99989049630166648708]
])
rdimage000translation   =   np.float32(
  [-0.24386599999999999944, 0.2348540000000000072, 0.44275100000000000566]
)
rdimage001=loadImage("imagenes/rdimage.001.ppm","COLOR")
rdimage001distortion=np.float32([0,0,0])
rdimage001rotation= np.float32([
  [0.99975480066385657985, 0.019390648831902848603, 0.010693048557372427862],
  [-0.019846499999999999558, 0.99882047750746361103, 0.044314446284616657024],
  [-0.009821150000000000 7127, -0.044515800000000001202, 0.99896040390149476451]
])
rdimage001translation=np.float32(
  [-0.35059299999999998798, -0.70982999999999996099, 0.75288600000000005519]
)
rdimage004=loadImage("imagenes/rdimage.004.ppm","COLOR")
rdimage004distortion=np.float32([0,0,0])
rdimage004rotation=np.float32([
  [0.99996050522707646824, 0.008791445647177747319, -0.001303253406947181 5585 ],
  [-0.008751540000000000 3605, 0.99957003802442112583, 0.027984810728066047275 ],
  [0.001548719999999999 9578 ,-0.027972299999999998554, 0.99960749892098743619],
])
rdimage004translation   =   np.float32(
  [-0.86724299999999998612, -0.17648199999999999998, 3.98545000000000001592]
)
```

b) Calcular parejas de puntos en correspondencias entre las imágenes

Ahora tenemos 3 imágenes: 0, 1, y 4 (por sus nombres). Vamos a buscar correspondencias entre 0 y 1, entre 0 y 4 y entre 1 y 4, es decir, el número menor indica que suponemos que fue tomada por la cámara que se situaba a la izquierda. Procederemos como en el apartado número 3a), probando con los tres detectores disponibles.

Con ORB

- Se han encontrado 139 puntos entre 0 y 1.
- Se han encontrado 70 puntos entre 0 y 4.
- Se han encontrado 72 puntos entre 1 y 4.

Con BRISK

- Se han encontrado 1918 puntos entre 0 y 1.
- Se han encontrado 918 puntos entre 0 y 4.
- Se han encontrado 947 puntos entre 1 y 4.

Con AKAZE

- Se han encontrado 2939 puntos entre 0 y 1.
- Se han encontrado 1159 puntos entre 0 y 4.
- Se han encontrado 1264 puntos entre 1 y 4.

Parece lógico quedarnos con AKAZE en este caso. La llamada que he realizado ha sido la siguiente, aplicando de nuevo el umbral de 0.8 que explicábamos en el mencionado apartado:

```
imgresult_01, points1_01, points2_01=findCorrespondencesAKAZE(rdimage000,rdimage001,0.8)
imgresult_04, points1_04, points2_04=findCorrespondencesAKAZE(rdimage000,rdimage004,0.8)
imgresult_14, points1_14, points2_14=findCorrespondencesAKAZE(rdimage001,rdimage004,0.8)
```

c) Estimar la matriz esencial y calcular el movimiento.

La matriz fundamental F pasa a convertirse en la matriz esencial E cuando trabajamos en un escenario de cámara calibrada. En este caso, tenemos que normalizar las coordenadas. Conocida K, un punto x es normalizado ($\hat{x}$) mediante la siguiente expresión:

$$\hat{x} = K^{-1}x$$

Salvo por este paso intermedio, podemos definir E igual que F, en lo que a correspondencias de puntos (en este caso normalizados) se refiere:

$$\hat{x}'^{T} E \hat{x} = 0$$

Vamos sustituir y desarrollar en esta ecuacion para despejar E:

$$(K'^{-1}x')^{T} E K^{-1}x = 0 \Rightarrow x'^{T} K'^{-T} E K^{-1}x = 0 \Rightarrow E = K'^{T} x'^{-T} x^{-1} K$$

Vamos a establecer un paralelismo entre la ecuación de la matriz fundamental y la matriz esencial:

$$x'^T F x = 0 \Rightarrow F = x'^T x^{-1}$$
$$E = K'^T \underbrace{x'^{-T} x^{-1}}_{F} K$$

Acabamos de dilucidar que la matriz esencial E puede ser calculada a partir de las matrices K', K y F. Como en este caso particular todas las matrices K son la misma, tenemos que, la matriz esencial E puede ser calculada como:

$$E = K^T F K$$

Hemos conseguido el primer objetivo, calcular la matriz esencial. Ahora tenemos que conseguir el segundo, es decir, extraer de E el movimiento de la cámara, esto es, una pareja $[R|t]$ donde R es la matriz de rotación y t el vector de translación. Seguiremos la filosofía que aplica el libro de Hartley & Zisserman, en su sección 9.6 [5]. Dadas dos cámaras, se trata de suponer que una cámara está en forma canónica:

$$P_c = [I|0]$$

Y la otra cámara se mueve con respecto a esta. En el mismo libro se puede encontrar la demostración de por qué suponer esto siempre es posible y correcto. La incógnita que buscamos entonces es la otra cámara P'. En primer lugar debemos definir un sistema de referencia construido con vectores ortogonales (matriz ortogonal). Si realizamos la descomposición en valores singulares de la matriz esencial, $SVD(E) = U, D, V^T$, obtenemos que D es $diag(v_s, v_s, 0)$. Nuestra matriz ortogonal debería ser:

$$W = \begin{pmatrix} 0 & -v_s & 0 \\ v_s & 0 & 0 \\ 0 & 0 & v_s \end{pmatrix}$$

Dado que W está compuesta por vectores ortogonales, nos quedamos con el mismo sistema representado por vectores unitarios, esto es, dividiendo por v_s. Entonces la matriz W definitiva sería:

$$W = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A partir de la descomposición en valores singulares de E y la matriz W, podemos obtener fácilmente los vectores de translación además de las matrices de rotación. Los vectores de translación son ambos signos de la tercera columna de U. Podemos encontrar la demostración en [5]:

$$t_1 = u_3$$
$$t_2 = -u_3$$

Los vectores de rotación se obtienen como sigue:

$$R_1 = UW^T V^T$$
$$R_2 = UW^T V$$

Tanto la demostración como la interpretación geométrica de estas soluciones también pueden ser consultadas en [5]. En estas circunstancias, podemos obtener cuatro cámaras como sigue:

$$P_1 = K[R_1|t_1]$$
$$P_2 = K[R_1|t_2]$$
$$P_3 = K[R_2|t_1]$$
$$P_4 = K[R_2|t_2]$$

Nos encontramos en la tesitura de elegir entre las siguientes situaciones para ser P':

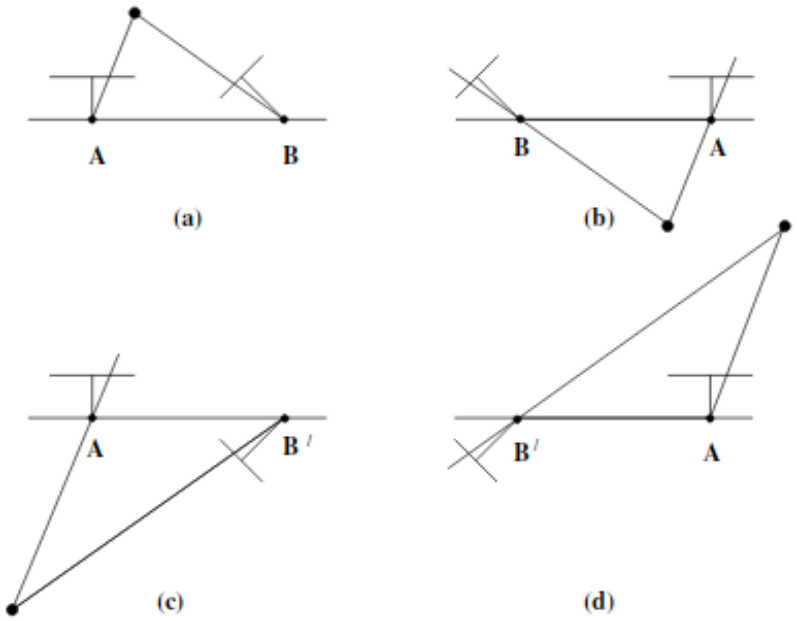


Figura 8. Posibles situaciones de las cámaras.
(Multiple View Geometry in Computer Vision. Hartley & Zisserman)

Matemáticamente todas son correctas, sin embargo, geométricamente sólo una, es el caso a), donde todos los puntos se encuentran delante de la cámara, esto es, tienen profundidad positiva. El problema es que nuestros puntos tienen coordenadas 2D. La solución es triangular los puntos. La triangulación se basa en ver dónde se cortan los rayos proyectados por las dos imágenes en cuestión y obtener la profundidad.

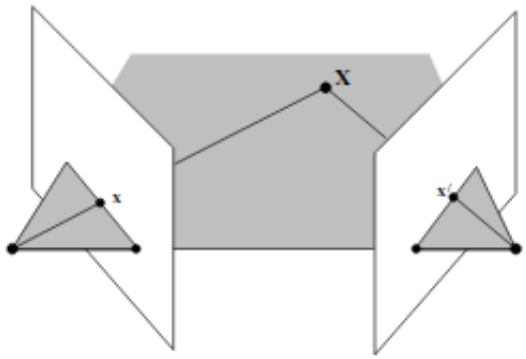


Figura 9. Triangulación de puntos.
(Multiple View Geometry in Computer Vision. Hartley & Zisserman)

Para llevar a cabo este cometido, sean ahora los dos conjuntos de puntos en correspondencias X y X', se trata de encontrar el signo de la profundidad, que es lo que realmente nos interesa. Dado un par de puntos en correspondencia, $x_i \in X$ y $x'_i \in X'$, deberíamos resolver la ecuación siguiente:

$$\begin{bmatrix} x'_i \times Rx_i & x'_i \times T \end{bmatrix} \begin{bmatrix} \lambda_i \\ \gamma \end{bmatrix} = 0$$

En este momento, hay varias cosas que explicar. La constante λ nos habla acerca de la profundidad, mientras que la constante γ de la translación. Tal como sugieren los subíndices, podemos dilucidar que hay un λ_i para cada pareja de puntos en correspondencia $x_i \leftrightarrow x'_i$, mientras que la constante γ es común a todos los puntos en correspondencia. Esto es lógico si tenemos en cuenta que cada uno de los puntos del mundo tiene una profundidad con respecto de la cámara, pero todos los puntos fueron trasladados proporcionalmente. La ecuación anterior no significa ni más ni menos que el punto x_i rotado y trasladado mediante las matrices R y T hace corresponder sus coordenadas con las de x'_i salvo una constante que no podemos conocer.

Fijémonos cómo $[a, b, c]^T = x'_i \times Rx_i$ y $[d, e, f]^T = x'_i \times T$ son, a su vez, matrices de tamaño 3×1. Entonces, podríamos reescribir la anterior ecuación como sigue:

$$\begin{bmatrix} \begin{bmatrix} a & b & c \end{bmatrix}^T & \begin{bmatrix} d & e & f \end{bmatrix}^T \end{bmatrix} \begin{bmatrix} \lambda_i \\ \gamma \end{bmatrix} = 0$$

Cuyo desarrollo sería el siguiente:

$$a\lambda_i + b\lambda_i + c\lambda_i + d\gamma + e\gamma + f\gamma = 0 \Rightarrow \lambda_i(a + b + c) + \gamma(d + e + f) \Rightarrow$$

$$\Rightarrow \lambda_i \sum (x'_i \times Rx_i) + \gamma \sum (x'_i \times T) = 0$$

Computacionalmente, cada uno de los elementos de la matriz izquierda de la primera ecuación serían números. Dado que deberíamos conocer cada λ_i, para obtener la ecuación inicial para todas las parejas de puntos, podríamos montar el sistema siguiente:

$$\underbrace{\begin{bmatrix} x'_1 \times Rx_1 & 0 & 0 & 0 & \dots & x'_1 \times T \\ 0 & x'_2 \times Rx_2 & 0 & 0 & \dots & x'_2 \times T \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & x'_n \times Rx_n & x'_n \times T \end{bmatrix}}_{M\Lambda} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \gamma \end{bmatrix} = 0$$

Como se puede apreciar, la matriz $M\Lambda$ tiene tamaño $n \times n + 1$, donde n es el número total de parejas de puntos en correspondencia. Si realizamos el mismo desarrollo de antes, esto es análogo al sistema:

$$\begin{cases} \lambda_1 \sum (x'_1 \times Rx_1) + \gamma \sum (x'_1 \times T) = 0 \\ \lambda_2 \sum (x'_2 \times Rx_2) + \gamma \sum (x'_2 \times T) = 0 \\ \vdots \\ \lambda_n \sum (x'_n \times Rx_n) + \gamma \sum (x'_n \times T) = 0 \end{cases}$$

Para resolver este sistema usaremos la descomposición en valores singulares de la matriz $M\Lambda$, es decir, $U, D, V^T = SVD(M\Lambda)$. Obtendremos la columna de V que se encuentra en el lugar en el que se encuentra el mínimo elemento de D. La solución (λ_1 λ_2 ... γ) se corresponde con esta columna.

Una vez repedido este proceso, tendremos cuatro conjuntos:

- Solución para $[R1|t1]$: $(\lambda_1\ \lambda_2\ ...\ \gamma)_{11}$
- Solución para $[R1|t2]$: $(\lambda_1\ \lambda_2\ ...\ \gamma)_{12}$
- Solución para $[R2|t1]$: $(\lambda_1\ \lambda_2\ ...\ \gamma)_{21}$
- Solución para $[R2|t2]$: $(\lambda_1\ \lambda_2\ ...\ \gamma)_{22}$

Teniendo esto, bastaría contar en cada uno de los anteriores conjuntos cuántas λ_i son negativas. Idealmente, debería haber sólo uno que devolviera 0, pero debido a redondeos y otras imprecisiones relacionadas con la falta de exactitud del computador, se entiende que la combinación correcta de $[R|t]$ es aquélla para la que haya menos λ_i negativas.

Todo este arduo proceso queda implementado en la siguiente función:

```
"""
Calcula el movimiento (R,t) de una camara
a partir de sus parametros intrinsecos,
la matriz fundamental y puntos en correspondencia
"""
def calculateCameraMotion(K,F,points1,points2):
    npoints=len(points1)

    if(npoints==len(points2)):
        E = np.transpose(K).dot(F).dot(K)
        D,U,Vt=cv2.SVDecomp(E)

        #Cálculo de los vectores de translacion
        t=np.transpose(U)[-1]
        t1=t
        t2=-t

        #Calculo de las matrices de rotacion
        w=[[0,-1,0],[1,0,0],[0,0,1]]
        R1=U.dot(np.transpose(w)).dot(Vt)                #[UW^TV^T]
        R2=U.dot(np.transpose(w)).dot(np.transpose(Vt)) #[UW^TV]

        #Agrupamos las combinaciones y convertimos los puntos a coords homogeneas
        combinations=[[R1,t1],[R1,t2],[R2,t1],[R2,t2]]
        pts1=[[x,y,1.0] for [x,y] in points1]
        pts2=[[x,y,1.0] for [x,y] in points2]

        #Sistema para la combinacion [R1|t1]
        system=np.zeros( (npoints,npoints+1) )
        for i in xrange(npoints):
            system[i][i]=sum(np.float32([np.cross(pts2[i],R1.dot(pts1[i]))]).ravel())
            system[i][-1]=sum(np.cross(pts2[i],t1))
        D,U,Vt=cv2.SVDecomp(system)
        min_index=np.argmin(D)
        solC1=Vt[min_index]
        cont_neg_1=len(solC1[solC1<0])


        #Sistema para la combinacion [R1|t2]
        system=np.zeros( (npoints,npoints+1) )
        for i in xrange(npoints):
            system[i][i]=sum(np.float32([np.cross(pts2[i],R1.dot(pts1[i]))]).ravel())
```

```
            system[i][-1]=sum(np.cross(pts2[i],t2))
        D,U,Vt=cv2.SVDecomp(system)
        min_index=np.argmin(D)
        solC2=Vt[min_index]
        cont_neg_2=len(solC2[solC2<0])


        #Sistema para la combinacion [R2|t1]
        system=np.zeros( (npoints,npoints+1) )
        for i in xrange(npoints):
            system[i][i]=sum(np.float32([np.cross(pts2[i],R2.dot(pts1[i]))]).ravel())
            system[i][-1]=sum(np.cross(pts2[i],t1))
        D,U,Vt=cv2.SVDecomp(system)
        min_index=np.argmin(D)
        solC3=Vt[min_index]
        cont_neg_3=len(solC3[solC3<0])

        #Sistema para la combinacion [R2|t2]
        system=np.zeros( (npoints,npoints+1) )
        for i in xrange(npoints):
            system[i][i]=sum(np.float32([np.cross(pts2[i],R2.dot(pts1[i]))]).ravel())
            system[i][-1]=sum(np.cross(pts2[i],t2))
        D,U,Vt=cv2.SVDecomp(system)
        min_index=np.argmin(D)
        solC4=Vt[min_index]
        cont_neg_4=len(solC4[solC4<0])

        negative_depths=np.array([cont_neg_1,cont_neg_2,cont_neg_3,cont_neg_4])
        #devuelve E,R,t
        index_valid_sol=np.argmin(negative_depths)
        return E,combinations[index_valid_sol][0],combinations[index_valid_sol][1]
    else:
        raise IndexError, "Both lists of points must have the same lenght."
```

Para nuestro problema en concreto, los resultados obtenidos son los siguientes:

- Imágenes 0 a 1:

$$E_{01} = \begin{pmatrix} -0.41653086 & 446.46363204 & 962.18072938 \\ -406.81639601 & 3.84343291 & -75.20194524 \\ -973.21357388 & 88.59639998 & -8.76772386 \end{pmatrix}$$

$$R_{01} = \begin{pmatrix} -0.99984381 & 0.01621657 & -0.00702718 \\ -0.01647447 & -0.99912879 & 0.03834382 \\ 0.00639925 & -0.0384536 & -0.9992399 \end{pmatrix}$$

$$t_{01} = \begin{pmatrix} 0.06843489 \\ 0.92046169 \\ -0.38479469 \end{pmatrix}$$

- Imágenes 0 a 4:

$$E_{04} = \begin{pmatrix} 0.15447011 & 55.54182745 & 7.01005629 \\ -55.26044592 & 0.01749502 & -10.91654988 \\ -9.07529653 & 10.25319688 & -0.50471683 \end{pmatrix}$$

$$R_{04} = \begin{pmatrix} -0.99990085 & 0.00576922 & -0.01284576 \\ -0.00624666 & -0.99927927 & 0.03744219 \\ 0.01262049 & -0.03751872 & -0.99921623 \end{pmatrix}$$

$$t_{04} = \begin{pmatrix} 0.17915069 \\ 0.15992265 \\ -0.97073672 \end{pmatrix}$$

- Imágenes de 1 a 4:

$$E_{14} = \begin{pmatrix} 2.42450306 & -255.25058969 & 30.76326681 \\ 254.62092929 & 2.48313994 & 42.19805078 \\ -34.78419731 & -39.30995539 & -1.1297002 \end{pmatrix}$$

$$R_{14} = \begin{pmatrix} 0.76617002 & -0.54811861 & -0.33548396 \\ 0.62480925 & 0.75746552 & 0.18936574 \\ 0.15032265 & -0.35469983 & 0.92281695 \end{pmatrix}$$

$$t_{14} = \begin{pmatrix} -0.1495275 \\ 0.13523036 \\ 0.97946632 \end{pmatrix}$$

Reconstruir coordenadas 3D

Usando las imágenes del punto anterior reconstruir las coordenadas 3D, desde el sistema de referencia de la cámara izquierda, de parejas de puntos en correspondencias

Por cada pareja de imágenes, buscaremos puntos en correspondencias, construiremos las cámaras a partir de los datos proporcionados, calcularemos puntos de mundo (en coordenadas 3D) con la función `triangulatePoints()` de OpenCV y obtendremos los puntos píxel que podrían ser reconstruídos. Finalmente los pintaremos en lienzo con un color en escala de grises donde más claro representa más cercano y más oscuro representa más lejano. Así, los puntos para reconstrucción entre las imágenes `rdimage000.ppm` y `rdimage0001.ppm` se obtendrían así:

```
imgresult_01, points1_01, points2_01
    =findCorrespondencesAKAZE(rdimage000,rdimage001,0.8)
P=rdimageK.dot(
    np.float32([
        np.append(np.copy(rdimage000rotation[i]),rdimage000translation[i])
        for i in xrange(len(rdimage000rotation))
    ])
  )
P1=rdimageK.dot(
     np.float32([
         np.append(np.copy( rdimage001rotation[i]),rdimage001translation[i])
         for i in xrange(len(rdimage001rotation))
     ])
   )

world_points=np.float32([
        cv2.triangulatePoints(P, P1, points1_01[i], points2_01[i])
        for i in xrange(len(points1_01))
    ])
world_points=np.float32([ [x/k,y/k,z/k] for [x,y,z,k] in world_points])
pix_points=np.float32([ [int(1000*(x/z)),int(1000*(y/z))]
    for [x,y,z] in world_points])

canvas=giveCanvas(pix_points,10,10,5)
rdimage000copy=np.copy(rdimage000)
rdimage001copy=np.copy(rdimage001)

for point,pz,p_real_1,p_real_2 in zip(pix_points,world_points,points1_01,points2_01):
        cv2.circle(
            canvas,(point[0],point[1]), 1,
            (int(1000*pz[2])%255,int(1000*pz[2])%255,int(1000*pz[2])%255), 2)
        cv2.circle(
            rdimage000copy,(int(p_real_1[0]),int(p_real_1[1])), 2,
            (int(1000*pz[2])%255,int(1000*pz[2])%255,int(1000*pz[2])%255), 10)
        cv2.circle(
            rdimage001copy,(int(p_real_2[0]),int(p_real_2[1])), 2,
            (int(1000*pz[2])%255,int(1000*pz[2])%255,int(1000*pz[2])%255), 10)
```

Los puntos obtenidos son los siguientes:

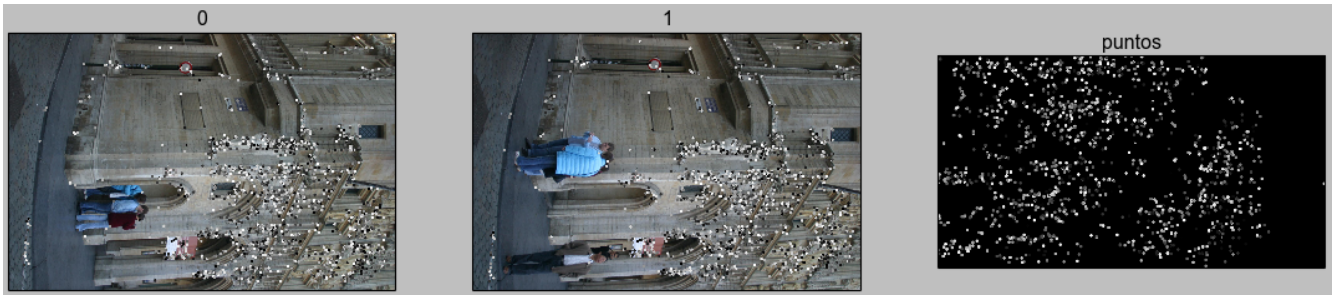


Figura 10. Puntos que se pueden reconstruir de entre 0 y 1.

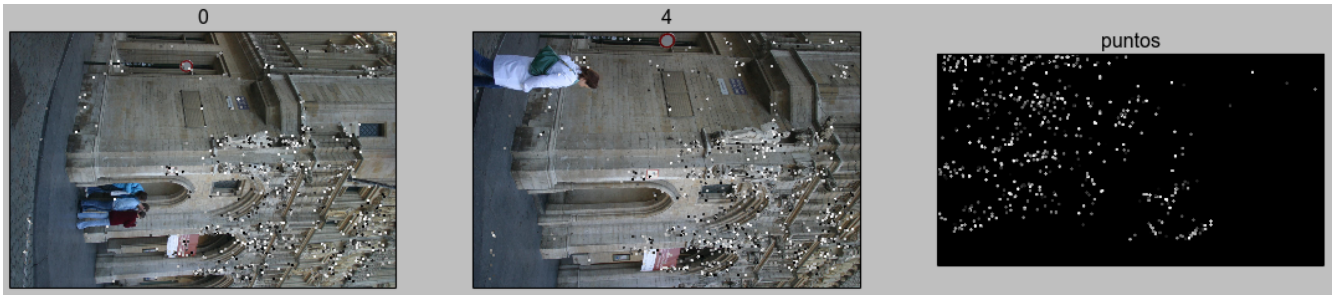


Figura 11. Puntos que se pueden reconstruir de entre 0 y 4.

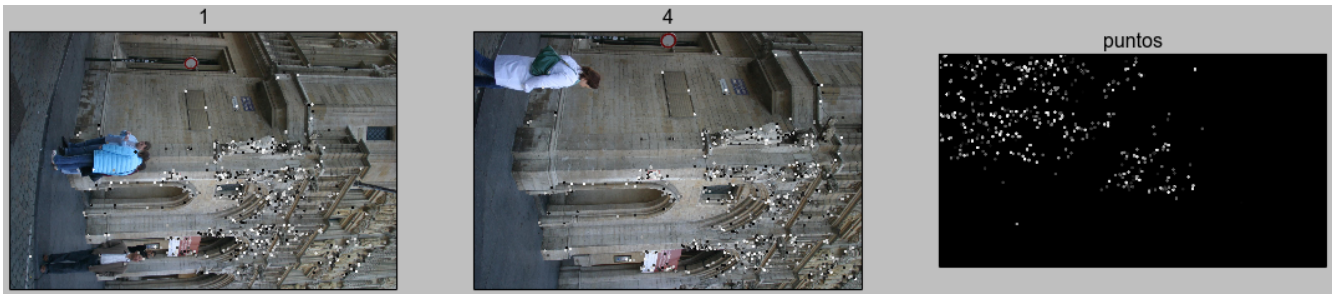


Figura 12. Puntos que se pueden reconstruir de entre 1 y 4.

Cuestiones técnicas

El proyecto ha sido implementado y probado en Ubuntu 14.04 LTS, con la versión Python 2.7.6. No se ha utilizado ninguna biblioteca no estándar de Python, excepto OpenCV, que hay que tener correctamente instalado y configurado en las variables de entorno. Para ejecutar correctamente el proyecto, hay que ejecutar el comando `python P3.py` (o `python2 P3.py` en caso de no ser la 2.7 la versión de Python por defecto) en una terminal desde el directorio donde se encuentra dicho archivo. Las imágenes que se usan deben estar dentro de la carpeta llamada `imagenes`, que se adjunta y no debería ser modificada.

Referencias

1. *Dissecting the Camera Matrix, Part 1: Extrinsic/Intrinsic Decomposition.* Sightations, a computer vision blog. Agosto 2012.
   http://ksimek.github.io/2012/08/14/decompose/

2. *Multiple View Geometry.* R. Hartley, A. Zisserman. Junio 1999.
   http://users.cecs.anu.edu.au/~hartley/Papers/CVPR99-tutorial/tut_4up.pdf

3. *A Flexible New Technique for Camera Calibration.* Z. Zhang. Diciembre 1998.
   https://www.researchgate.net/profile/Zhengyou_Zhang/publication/3193178_A_Flexible_New_Technique_for_Camera_Calibration/links/568ac79108ae1975839daf25.pdf

4. *Distinctive Image Features from Scale-Invariant Keypoints.* D.G. Lowe. Enero 2004. Pág 20
   http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

5. *Multiple View Geometry in Computer Vision.* R. Hartley A. Zisserman. 2004. Sección 9.6 Págs. 275-279
   http://cvrs.whu.edu.cn/downloads/ebooks/Multiple%20View%20Geometry%20in%20Computer%20Vision%20(Second%20Edition).pdf