Calib.py :

<u>Calibrate2D() :</u>

```
TODO : form the matrix equation Ap = b for the X-Z plane
```

We use the set of equations from the projective cameras et rewrite it :

$$
\begin{bmatrix}
X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 & -u_1 \\
0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 & -v_1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -u_iX_i & -u_iY_i & -u_iZ_i & -u_i \\
0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 & -v_iX_i & -v_iY_i & -v_iZ_i & -v_i \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_nX_n & -u_nY_n & -u_nZ_n & -u_n \\
0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -v_nX_n & -v_nY_n & -v_nZ_n & -v_n
\end{bmatrix}
\begin{bmatrix} p_{00} \\ p_{01} \\ \vdots \\ p_{10} \\ p_{11} \\ \vdots \\ p_{22} \\ p_{23} \end{bmatrix}
=
\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}
$$

We change two things :
1 columns containing Y disappears because we're in the X-Z plane.
2) the last column [-u1 –v1 … -un -vn] is put the other side of the equation. (instead of having the [0….0] vector.

The [X Y Z] are from ref3D.
The [u v] are from ref2D.

```
Y = ref3D[x+4][1]
Z = ref3D[x+4][2]
u = ref2D[x+4][0]
v = ref2D[x+4][1]
Ayz[2*x] = np.array([Y, Z, 1, 0, 0, 0, -u*Y, -u*Z])
Ayz[2*x+1] = np.array([0, 0, 0, Y, Z, 1, -v*Y, -v*Z])
b[2*x] = u
b[2*x+1] = v
```

```
TODO : solve for the planar projective transformation using linear least squares
```

We use the function np.linalglstsq() with our equation Ap=b found before :

```
n = np.linalg.lstsq(Ayz, b, rcond = None)[0]
```

```
And we set p22 to 1 for the general scale :
n = np.append(n,1)
```

(And we reshape our vector to have a 3x3 matrix :

```
Hyz = np.reshape(n, (3,3))
```

)

```
TODO : form the matrix equation Ap = b for the Y-Z plane
```

We use the set of equations from the projective cameras et rewrite it :

$$\begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 & -u_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -u_iX_i & -u_iY_i & -u_iZ_i & -u_i \\ 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 & -v_iX_i & -v_iY_i & -v_iZ_i & -v_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_nX_n & -u_nY_n & -u_nZ_n & -u_n \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -v_nX_n & -v_nY_n & -v_nZ_n & -v_n \end{bmatrix} \begin{bmatrix} p_{00} \\ p_{01} \\ \vdots \\ p_{10} \\ p_{11} \\ \vdots \\ p_{22} \\ p_{23} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

We change two things :
1) columns containing X disappears because we're in the Y-Z plane.
2) the last column [-u1 –v1 … -un -vn] is put the other side of the equation. (instead of having the [0….0] vector.

The [X Y Z] are from ref3D.
The [u v] are from ref2D.

```
TODO : solve for the planar projective transformation using linear least squares
```

We use the function np.linalglstsq() with our equation Ap=b found before.
And we set p22 to 1 for the general scale.
(And we reshape our vector to have a 3x3 matrix)

<u>Gen_correspondences() :</u>

```
TODO : define 3D coordinates of all the corners on the 2 calibration planes
```

We compute the values s*u and s*v from corners (where we have u,v,s) :

```
su = corners[k][0] * corners[k][2]
sv = corners[k][1] * corners[k][2]
s = corners[k][2]
```

We know :

- For a projective camera:

Pixel coordinates of the point observed in the image

$$\begin{bmatrix} s_i u_i \\ s_i v_i \\ s_i \end{bmatrix} = \begin{bmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}$$

Known world coordinates of a point in a controlled scene

So to get the coordinates : pixels_coordinates * inverse(proj matrix)

We compute the coordinates as if they were in the X-Z plane :

```
coord3Dxz = inv(Hxz).dot(coord)
```

We compute the coordinates as if they were in the X-Z plane :

```
coord3Dyz = inv(Hyz).dot(coord)
```

We « normalize » the coordinates vectors :
we divide the coordinate by the « scale » and set the scale at 1.

```
coord3Dxz[0] = coord3Dxz[0]/coord3Dxz[2]
coord3Dxz[1] = coord3Dxz[1]/coord3Dxz[2]
coord3Dxz[2] = 1
newCoordXZ[k] = coord3Dxz.T

coord3Dyz[0]= coord3Dyz[0]/coord3Dyz[2]
coord3Dyz[1]= coord3Dyz[1]/coord3Dyz[2]
coord3Dyz[2]= 1
```

```
TODO : project corners on the calibration plane 1 onto the image
```

We test if the coordinate are in the X-Z plane :

```
newCoordXZ[k][0] > 0 and newCoordXZ[k][1] > 0 and newCoordXZ[k][0] <= 10 and newCoordXZ[k][1] <= 8):
```

if they are :
we compute the [u s v] with the same equation as before from the projective camera:

```
transfo = Hxz.dot(newCoordXZ[k])
```

(we want u and v and not su and sv :

```
transfo = transfo/transfo[2]
```

)
And we ad dit to the list of vector from this plane :

```
cornersXZ2d.append(transfo[0:2])
cornersXZ3d.append(newCoordXZ[k][0:2])
```

At the end,
We add 0 to the Y values of the world coordinate vectors (because we're in X-Z plane):

```
vect1 = np.zeros((cornersXZ3d.shape[0],1))
cornersXZ3d = np.hstack((cornersXZ3d[:,:1], vect1, cornersXZ3d[:,1:]))
```

```
TODO : project corners on the calibration plane 2 onto the image (YZ-plane)
```

We test if the coordinate are in the Y-Z plane :

```
if ((newCoordYZ[k][0] >  0)  and (newCoordYZ[k][1] > 0) and (newCoordYZ[k][0] <= 10 and (newCoordYZ[k][1] <= 8))):
```

if they are :
we compute the [u s v] with the same equation as before from the projective camera:

```
transfo = Hyz.dot(newCoordYZ[k])
```

(we want u and v and not su and sv :

```
transfo = transfo/transfo[2]
```

)
And we ad dit to the list of vector from this plane :

```
cornersYZ2d = np.array(cornersYZ2d)
cornersYZ3d = np.array(cornersYZ3d)
```

At the end,
We add 0 to the X values of the world coordinate vectors :

```
vect1 = np.zeros((cornersYZ3d.shape[0],1))
cornersYZ3d = np.hstack((vect1,cornersYZ3d))
```

```
TODO : locate the nearest detected corners
```

We put together all the coordinates :

```
ref3D = np.concatenate((cornersXZ3d, cornersYZ3d), axis=0)
ref2D = np.concatenate((cornersXZ2d, cornersYZ2d), axis=0)
```

<u>Calibrate3D() :</u>

```
TODO : form the matrix equation Ap = b for the camera
```

$$\begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 & -u_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -u_iX_i & -u_iY_i & -u_iZ_i & -u_i \\ 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 & -v_iX_i & -v_iY_i & -v_iZ_i & -v_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_nX_n & -u_nY_n & -u_nZ_n & -u_n \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -v_nX_n & -v_nY_n & -v_nZ_n & -v_n \end{bmatrix} \begin{bmatrix} p_{00} \\ p_{01} \\ \vdots \\ p_{10} \\ p_{11} \\ \vdots \\ p_{22} \\ p_{23} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

This time we use the entire matrix.

```
for x in range(xdiv):
    X = ref3D[x][0]
    Y = ref3D[x][1]
    Z = ref3D[x][2]
    u = ref2D[x][0]
    v = ref2D[x][1]
```

```
    A[2*x] = np.array([X, Y, Z, 1, 0, 0, 0, 0, -u*X, -u*Y, -u*Z])
    A[2*x+1] = np.array([0, 0, 0, 0, X, Y, Z, 1, -v*X, -v*Y, -v*Z])
    b[2*x] = u
    b[2*x+1] = v
```

```
TODO : solve for the projection matrix using linear least squares
```

As before, we use the np.linalg.lstsq function.
We set the scale at 1.
We reshape the vector into a matrix.

```
n = np.linalg.lstsq(A, b, rcond = None)[0]
n = np.append(n,1)
P = np.reshape(n, (3,4))
```

<u>Decompose P() :</u>

```
TODO: extract the 3 x 3 submatrix from the first 3 columns of P
```

We delete the 4th column.

```
P = np.delete(P,3,axis = 1)
```

```
TODO : perform QR decomposition on the inverse of [P0 P1 P2]
```

$$\begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 \end{bmatrix}^{-1} = \mathbf{QR}'$$

Taking the inverse of both sides gives

$$\begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 \end{bmatrix} = (\mathbf{QR}')^{-1}$$
$$= \mathbf{R}'^{-1}\mathbf{Q}^{-1}$$

So to get QR we use the inverse of P :

```
p = np.linalg.inv(P)
```

And we appy the function :

```
q, r = np.linalg.qr(p)
```

```
TODO : obtain K as the inverse of R
```

```
K = np.linalg.inv(r)
```

```
TODO : obtain R as the tranpose of Q
```

```
R = q.T
```

```
TODO : normalize K
```

We use this « algorithm » :

- Normalization of the camera calibration matrix $\mathbf{K}$
  - If the element k_{22} is not 1, extract it as a scale factor α and divide the whole matrix by α, i.e.,

$$\begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 \end{bmatrix} = \alpha\mathbf{K}\mathbf{R}$$

  - If the element k_{00} is negative, multiply the 1^{st} column of $\mathbf{K}$ and the 1^{st} row of $\mathbf{R}$ by -1 respectively to make it positive
  - If the element k_{11} is negative, multiply the 2^{nd} column of $\mathbf{K}$ and the 2^{nd} row of $\mathbf{R}$ by -1 respectively to make it positive

```
alpha = K[2][2]
K = K/alpha

k00 = K[0][0]
if k00 < 0 :
    K[:,0] = -K[:,0]
    R[0,:] = -R[0,:]

k11 = K[1][1]
if k11 < 0:
    K[:,1] = -K[:,1]
    R[1,:] = -R[1,:]
```

```
TODO : obtain T from P3
```

We've learnt this formula in class :

$$\mathbf{T} = \frac{1}{\alpha}\mathbf{K}^{-1}\mathbf{p}_3$$

```
T = (1/alpha) * np.linalg.inv(K).dot(P3)
T = np.reshape(T, (3,1))
```

SO to get RT : we just add the vector T to R :

```
RT = np.hstack((R,T))
```

Epipolar .py :

Compose E() :

```
TODO : compute the relative rotation R
```

We got RT but we just want R to we take the 3 first columns :

```
R1 = RT1[:, 0:3]
R2 = RT2[:, 0:3]
```

To get the relative rotation we use the formula :
Rrelativ = inv(R1) * R2

```
R = np.linalg.inv(R1).dot(R2)
```

```
TODO: compute the relative translation T
```

We want Ti so we take the last column of RTi :

```
T1 = RT1[:, 3]
T2 = RT2[:, 3]
```

The relative translation :
T = inv(R1) * (T2-T1)

```
T = np.linalg.inv(R1).dot(T2 - T1)
```

```
TODO : compose E from R and T
```

We have seen :

$$[\mathrm{T}]_\times = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

$$\mathbf{E} = [\mathbf{T}]_\times \mathbf{R}$$

So :

```
Tx = np.array([[0, -T[2], T[1]], [T[2], 0, -T[0]], [-T[1], T[0], 0]])
E = Tx.dot(R)
```

Worked on this with Vasilis Ntogramatzis.