

10.2 Epipolar geometry

Epipolar geometry is the **set of geometric constraints between two views of a scene**. When two cameras capture a 3D scene from two different positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. These relations are derived based on the assumption that the cameras can be approximated by the pinhole camera model (e.g. their images are undistorted).

In this notebook, we will go deeper into epipolar geometry, learning:

- Basic concepts of epipolar geometry ([section 10.2.1](#))
- Stereo rectification ([section 10.2.2](#))
- Essential matrix ([section 10.2.3](#)), and
- Fundamental matrix ([section 10.2.4](#))

Problem context - Stereo rectification

In practice, as we depicted in the previous notebook, accurate triangulation is not fully feasible as **stereo systems does not fulfill the ideal configuration**, but the good news is that any two images taken with any configuration (as long as there is overlap between the images, of course) can be transformed so that they comply with an ideal stereo camera configuration. This is called **stereo rectification** or **image rectification**. Here, we apply homographies to the left and right images in order to force this ideal configuration.

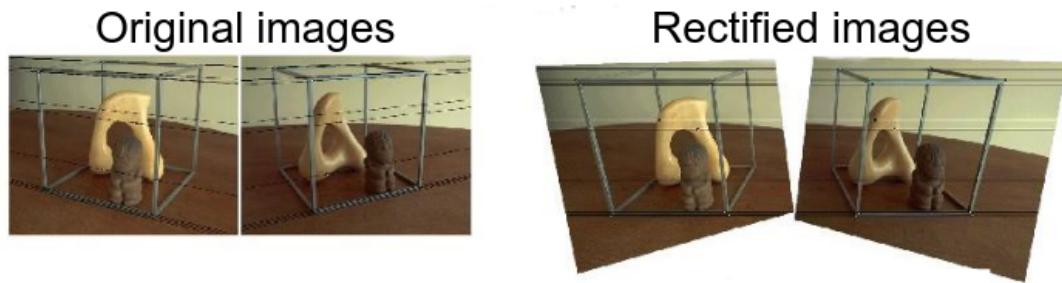
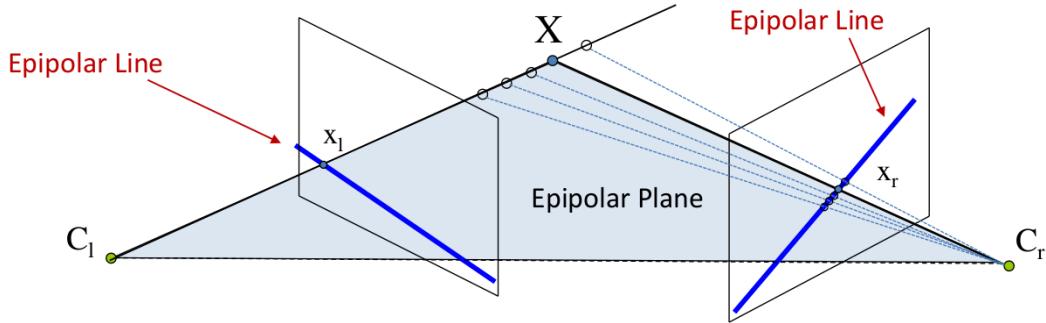


Image rectification is based in epipolar geometry so, let's start to learn how it works!

10.2.1 Concepts of epipolar geometry

Before starting to explain how epipolar geometry works, we have to understand some basic concepts: the **epipolar plane**, the **epipolar lines** and the **epipoles**.

First of all, we have the **epipolar plane**. This plane is formed by 3 points: the left and right optical centers (C_l , C_r), and a 3D point in the world (\mathbf{X}).

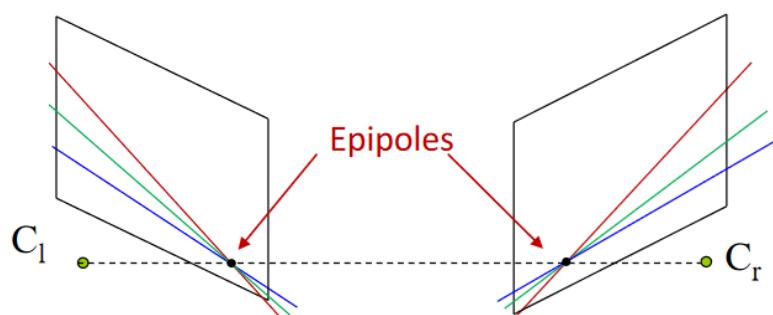


The interesting thing about the epipolar plane, is that it **intersects the images at the conjugate epipolar lines** of the stereo vision system. The 2D projections (x_l, x_r) of the 3D point X are contained also in the epipolar plane and **lie on their corresponding epipolar lines**. As can be seen in the image, these epipolar lines are the **intersection of the epipolar plane and the image planes**.

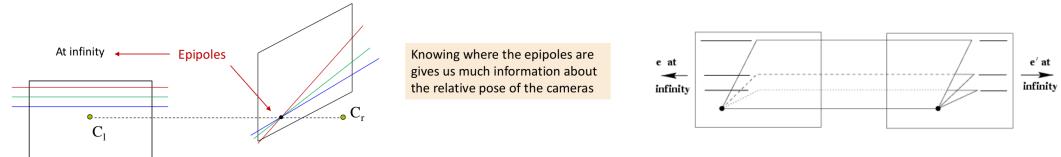
As you might realize, it would be interesting to compute these epipolar lines because they will help us to check **whether a pair of corresponding points is correct or not** (in order to find the matching point in the other image, you don't need to search the whole image, just to search along the epipolar line!). This is called the **epipolar constraint** useful to transform a search in 2D into a 1D problem. Interestingly, this is valid for all the pixels in the images, that is, all the pixels will have their corresponding epipolar line (or **epiline**, for short) in the other image. And even more interestingly, the direction of the epilines **do not depend on the scene** but only on the camera configuration! That is, if you have a rigid stereo camera pair, the epilines will not change regardless the scene you are viewing! So:

Given a two-camera configuration (represented by R, t), for each 3D point there is a **unique epipolar plane** and, consequently, a **unique pair of epipolar lines**.

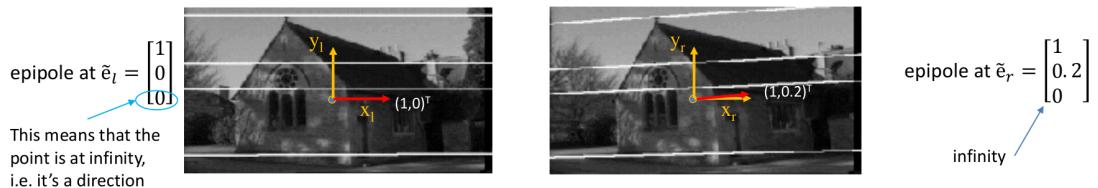
On the other hand, maybe you have noticed that all the epipolar lines **intersect in one single point at each image**, this point is called the **epipole**, and represents the projection of the optical center of each camera on the other image plane (note that it may be outside the boundaries of the image!). All the epilines pass through its epipole. So, one way to determine the position of the epipole is to compute a set of epilines and to find their intersection point.



There are some special cases where one or both epipoles **are placed at the infinity** (if the image plane is parallel to the line $C_l - C_r$), which implies that we do not know their location but their *direction*:

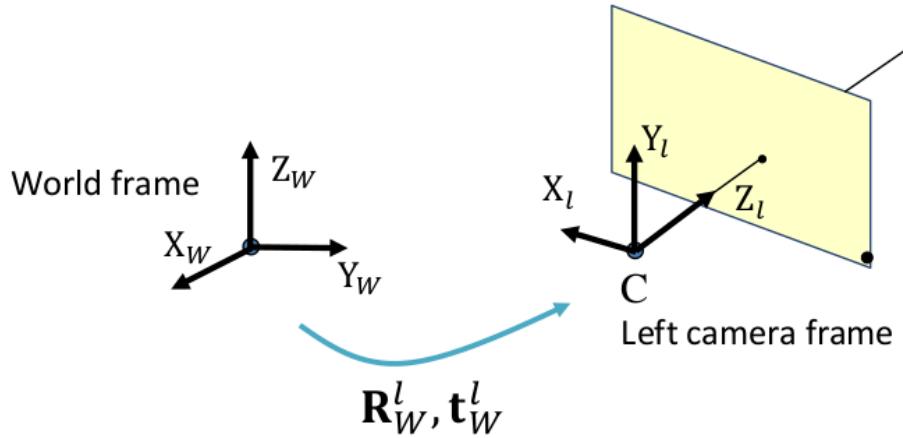


This means that, in this situation, the epilines are **horizontal and parallel**, which is quite handy as we depicted in the previous notebook. Notice that, in such case, the third coordinate of the homogeneous representation of the **epipole position is zero**. In these cases, we can compute the direction in which the epipole can be found:



Where are the epipoles?

First of all, let's **find the coordinates of an optical center \mathbf{C}** in the **WORLD** reference frame, that is \mathbf{C}^W . (*Remember that the superscript in a point/vector indicates the reference frame*). Obviously, in the local reference system of the same camera, its coordinates are equal to zero (the optical center is the origin of coordinates!): $\mathbf{C}^C = \mathbf{0}$:



$$\begin{cases} \mathbf{C}^C = \mathbf{R}_W^C \mathbf{C}^W + \mathbf{t}_W^C \\ \mathbf{C}^C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0} \end{cases} \rightarrow \mathbf{0} = \mathbf{R}_W^C \mathbf{C}^W + \mathbf{t}_W^C \rightarrow \mathbf{C}^W = -(\mathbf{R}_W^C)^T \mathbf{t}_W^C$$

For the left camera, and since the **WORLD** frame is assumed to be placed at the optical center of the left camera, we have that: $\mathbf{R}_W^{C_l} = I$, $\mathbf{t}_W^{C_l} = \mathbf{0}$, that is: $\mathbf{C}_l^W = \mathbf{0}$.

Analogously, the right optical center has these coordinates:

$$C_r^W = -(\mathbf{R}_W^{C_r})^T \mathbf{t}_W^{C_r} = -\mathbf{R}^T \mathbf{t}$$

That is, the left optical center is placed at the origin of coordinates while the right one has an arbitrary rotation and a translation with respect to it, which, for simplicity we will call \mathbf{R} and \mathbf{t} .

Now, let's find the left and right epipoles (e_l, e_r) by determining the projection of the optical centers on the other image plane. Remember the general equation for image projection in previous notebooks. Note: in these equations, the apostrophe in \tilde{e}' means the coordinates are expressed in pixels, while the tilde \sim means it is in homogeneous coordinates.:

$$\begin{aligned} e_l : \lambda \tilde{e}'_l &= \mathbf{P}_l \tilde{C}_r^W = \mathbf{K}_l [\mathbf{I} | \mathbf{0}] \begin{bmatrix} -\mathbf{R}^T \mathbf{t} \\ 1 \end{bmatrix} = -\mathbf{K}_l \mathbf{R}^T \mathbf{t} \\ e_r : \lambda \tilde{e}'_r &= \mathbf{P}_r \tilde{C}_l^W = \mathbf{K}_r [\mathbf{R} | \mathbf{t}] \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} = \mathbf{K}_r \mathbf{t} \end{aligned}$$

where \mathbf{K}_l and \mathbf{K}_r are the calibration matrices of the left and right camera, respectively.

Finally, the epipoles in the **ideal configuration** for triangulation (cameras with parallel optical axis, separated by a baseline, and with the same intrinsic parameters):

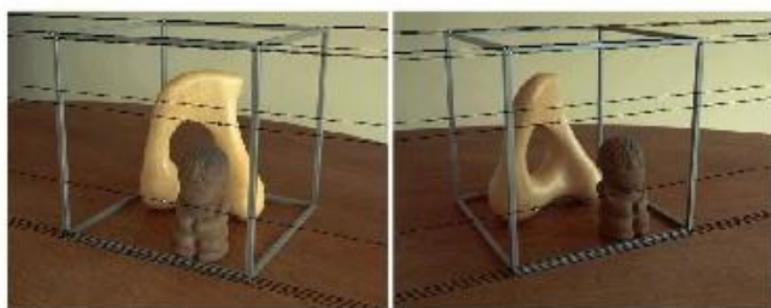
$$\begin{aligned} \mathbf{R} &= \mathbf{I}, \quad \mathbf{t} = [-b \ 0 \ 0]^T, \quad \mathbf{K} = \mathbf{K}_l = \mathbf{K}_r \\ \lambda \tilde{e}'_l &= -\mathbf{K} \mathbf{t} = [bk_x f \ 0 \ 0]^T \\ \lambda \tilde{e}'_r &= \mathbf{K} \mathbf{t} = [-bk_x f \ 0 \ 0]^T \end{aligned}$$

As we said before, **they are points at infinity in the direction of the X axis!**

10.2.2 Stereo rectification

In practice, our camera configuration will not be ideal (i.e.

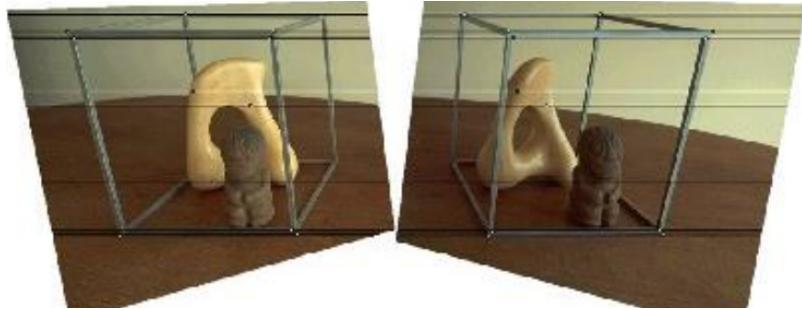
$\mathbf{R} \neq \mathbf{I} \wedge \mathbf{t} \neq [-b \ 0 \ 0]^T \wedge \mathbf{K}_l \neq \mathbf{K}_r$), and this involves a difficulty to **find the correspondences and to perform stereo triangulation**:



As you can see in the images, **the epipolar lines are not horizontal** as in the ideal configuration, but this can be solved. The process of **stereo rectification** projects the

images on a common plane so that the epipolar lines become horizontal in both images and at the same height, that is, transform them as if they had been captured in an ideal configuration, taking the **epipoles to infinity**.

For that, we can compute a **homography for each camera** that transforms the images to such ideal configuration:



OpenCV hint

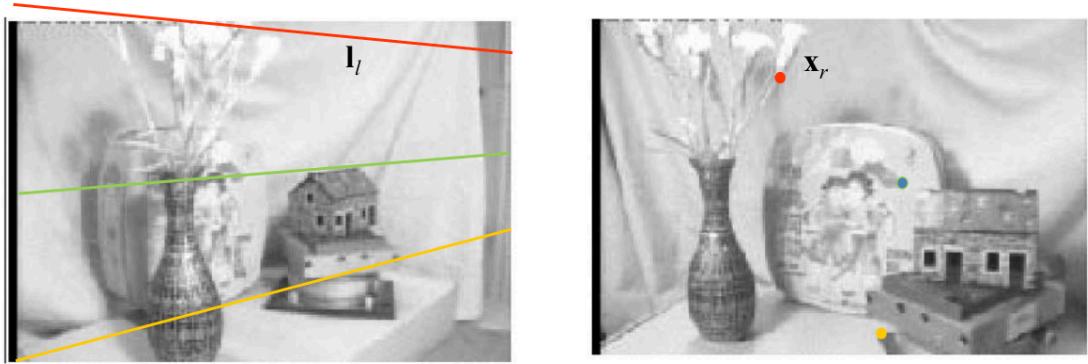
OpenCV defines a method for image rectification called (`cv2.stereoRectify`), which computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. This method asks for the matrices \mathbf{R}_r^l and \mathbf{t}_r^l , as well as the calibration matrices and then applies the camera model to compute such rotation matrices. As we don't have a stereo system that have a fixed relative pose, we cannot obtain those matrices (they could be obtained through [stereo calibration](#)).

Fortunately, there exists the method `cv2.stereoRectifyUncalibrated`, which returns the **rectification homographies** for both images given a set of matching points and the **fundamental matrix** (explained later). This method is used when **you don't know the relative pose of the cameras** that captured the pictures (e.g. two images taken with the same camera but in different positions).

10.2.3 The Essential Matrix

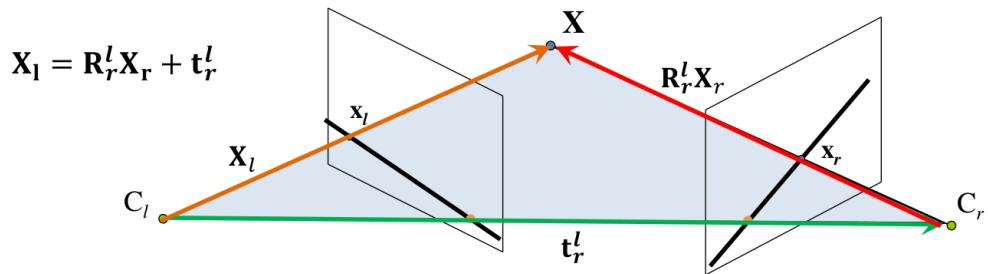
At the beginning of the notebook, we stated that every point in an image will have a corresponding epipolar line in the other image, which was given by the intrinsic epipolar geometry between the views. **The Essential Matrix (E)** is a 3×3 matrix that encodes such epipolar geometry **when the cameras are calibrated**. So, given a point in one image, multiplying it by the essential matrix will tell us the epipolar line in the second view:

$$l_l = \mathbf{E} \tilde{x}_r$$



where l_l is the epiline in the left image corresponding to the point x_r in the right one.

But, **how can we compute the essential matrix?**



As can be seen in the image, vectors \mathbf{X}_l (the coordinates of a 3D point in the left reference system), \mathbf{t}_r^l (the coordinates of the right optical center in the left reference system) and $\mathbf{R}\mathbf{X}_r$ (the coordinates of a 3D point in the right reference system) **are coplanar**, so, by definition, their triple product is zero:

$$\mathbf{X}_l^T \cdot [\mathbf{t}_r^l \times (\mathbf{R}\mathbf{X}_r)] = 0$$

This can be simplified by using the coordinate vectors of the projected points on the image plane: $\mathbf{X} = \lambda \tilde{\mathbf{x}}$, which are also coplanar.

$$\begin{aligned} \tilde{\mathbf{x}}_l^T [\mathbf{t}_r^l \times (\mathbf{R}\tilde{\mathbf{x}}_r)] &= \tilde{\mathbf{x}}_l^T \underbrace{[\mathbf{t}_r^l \times \mathbf{R}]}_{\mathbf{E}} \tilde{\mathbf{x}}_r = 0 \implies \\ \boxed{\tilde{\mathbf{x}}_l^T \mathbf{E} \tilde{\mathbf{x}}_r = 0} \end{aligned}$$

The problem of this matrix is that we are working in sensor coordinates (meters), so **the cameras need to be calibrated** in order to transform such coordinates to pixels, that is, \mathbf{K} must be known, so that (assuming identical intrinsic parameters):

$$\begin{aligned} \tilde{\mathbf{x}}'_r &= \mathbf{K} \tilde{\mathbf{x}}_r \\ \tilde{\mathbf{x}}'_l &= \mathbf{K} \tilde{\mathbf{x}}_l \end{aligned}$$

Given this limitation, the use of the **Fundamental Matrix** seems to be more appealing, since it can work with **uncalibrated cameras**. Let's discover it!

10.2.4 The Fundamental Matrix

The **Fundamental Matrix (F)** is similar to the Essential Matrix, but it works in the **image plane instead of with sensor coordinates**, but they have similar properties and expressions:

$$\tilde{\mathbf{x}}_l'^T \mathbf{F} \tilde{\mathbf{x}}_r' = 0$$

We can obtain the fundamental matrix from the essential one through:

$$\tilde{\mathbf{x}}_l^T \mathbf{E} \tilde{\mathbf{x}}_r = \left(\mathbf{K}^{-1} \tilde{\mathbf{x}}_l' \right)^T [\mathbf{t}]_\times \mathbf{R} \left(\mathbf{K}^{-1} \tilde{\mathbf{x}}_r' \right) \quad (1)$$

$$= (\tilde{\mathbf{x}}_l'^T \mathbf{K}^{-T}) [\mathbf{t}]_\times \mathbf{R} \left(\mathbf{K}^{-1} \tilde{\mathbf{x}}_r' \right) \quad (2)$$

$$= \tilde{\mathbf{x}}_l'^T \left(\mathbf{K}^{-T} [\mathbf{t}]_\times \mathbf{R} \mathbf{K}^{-1} \right) \tilde{\mathbf{x}}_r' \quad (3)$$

$$= \tilde{\mathbf{x}}_l'^T \mathbf{F} \tilde{\mathbf{x}}_r' = 0 \quad (4)$$

So, we have that

$$\mathbf{F} = \mathbf{K}^{-T} [\mathbf{t}]_\times \mathbf{R} \mathbf{K}^{-1} = \mathbf{K}^{-T} \mathbf{E} \mathbf{K}^{-1}$$

The good thing about working in the image plane, is that we can obtain the matrix \mathbf{F} from 8 pairs of points in both images, without needing any knowledge about the calibration matrix \mathbf{K} .

The **Fundamental matrix is the cornerstone of the epipolar geometry**, as it relates points and their corresponding epipolar lines in these ways:

F relating two corresponding points:

We can check if a correspondence of points is valid (points are contained in corresponding epipolar lines):

$$\tilde{\mathbf{x}}_l'^T \mathbf{F} \tilde{\mathbf{x}}_r' = 0$$

F relating points with their epipolar lines:

It's also possible to find the corresponding epipolar line of a point:

$$\mathbf{l}'_l = \mathbf{F} \tilde{\mathbf{x}}_r' \quad \mathbf{l}'_r = \mathbf{F}^T \tilde{\mathbf{x}}_l'$$

F used to compute the epipoles:

As the epipole is the intersection point of all epipolar lines, the epipole is the null space of the Fundamental matrix:

$$\tilde{\mathbf{x}}_l'^T \mathbf{F} \tilde{\mathbf{e}}'_r = 0, \quad \forall \tilde{\mathbf{x}}_l' \quad (5)$$

$$\mathbf{F} \tilde{\mathbf{e}}'_r = \mathbf{F}^T \tilde{\mathbf{e}}'_l = 0 \quad (6)$$

BONUS: F used to feel the rhythm:

The Fundamental matrix is so popular in computer vision that it even has a [funny song](#)!

```
In [1]:  
import numpy as np  
import cv2  
import matplotlib.pyplot as plt  
import matplotlib  
matplotlib.rcParams['figure.figsize'] = (20.0, 20.0)  
  
images_path = './images/'  
  
import sys  
sys.path.append("..")  
from utils.drawlines import drawlines
```

ASSIGNMENT 1: Retrieving the Fundamental matrix

In this task, we are going to rectify two images using the above-mentioned method `cv2.stereoRectifyUncalibrated()`. For that, you have to take two images of the same scene from different positions, imitating a stereo system. Note: you can use images `iml3.png` (left) and `imr3.png` (right) provided to you with this notebook if you can't use your own images.

The first step for image rectification is **finding the Fundamental matrix**, and for this we need a set of matching points! Use your favourite keypoint detector + descriptor in order to get proper a set of matches (we recommend **ORB for big images**).

What to do? Compute the Fundamental matrix that relates the epipolar geometry between your images and show it. For that you have to:

1. Load the images and convert them to grayscale.

```
In [2]:  
# ASSIGNMENT 1  
# Write your code here!  
# Read left image  
im_l = cv2.imread(images_path + 'iml3.png') # queryimage  
im_l = cv2.cvtColor(im_l, cv2.COLOR_BGR2RGB)  
  
# Read right image  
im_r = cv2.imread(images_path + 'imr3.png') # trainimage  
im_r = cv2.cvtColor(im_r, cv2.COLOR_BGR2RGB)  
  
# Get gray images  
gray_l = cv2.cvtColor(im_l, cv2.COLOR_RGB2GRAY)  
gray_r = cv2.cvtColor(im_r, cv2.COLOR_RGB2GRAY)  
  
print("Image sizes:", gray_l.shape, "and", gray_r.shape)
```

Image sizes: (363, 439) and (363, 438)

2. Detect keypoints in both images and find correspondences (do matching).

2.1 Option 1, **using ORB**.

```
In [3]: # ANOTHER OPTION: USE ORB KEYPOINTS
# Detect the ORB keypoints using the OpenCV method
# -- create the ORB detector
orb = cv2.ORB_create()

# -- detect ORB keypoints
kp_l = orb.detect(gray_l,None) # Leave the second None as it is
kp_r = orb.detect(gray_r,None) # Leave the second None as it is

# -- compute the descriptors with ORB
kp_l, des_l = orb.compute(gray_l, kp_l)
kp_r, des_r = orb.compute(gray_r, kp_r)

# Match descriptors.
matches = cv2.BFMatcher(cv2.NORM_HAMMING,crossCheck=True).match(des_l,des_r)

print("Number of ORB matches:",len(matches))
```

Number of ORB matches: 277

2.2 Option 2. Using SURF.

```
In [4]: # # detect keypoints using the SURF method
# surf = cv2.xfeatures2d.SURF_create(2000)

# kp_l, des_l = surf.detectAndCompute(gray_l,None)
# kp_r, des_r = surf.detectAndCompute(gray_r,None)

# # Call knnMatch
# pre_matches = cv2.BFMatcher().knnMatch(des_l,des_r, k=2)

# # Filter non-robust matches
# matches = [m for m,n in pre_matches if m.distance < 0.60*n.distance]

# print("Number of SURF matches:",Len(matches), "out of",len(pre_matches))
```

3. Compute the Fundamental matrix. OpenCV already implements a method for computing the Fundamental matrix (`cv2.findFundamentalMat()`) from two lists of matching points (unfortunately using their coordinates, not `cv2.keyPoint()` objects). Apart from the Fundamental matrix, this method also returns a vector `mask`, which states the points used for the matrix calculation. Play with the `ransacReprojThreshold` parameter in such function (take a look at it in the OpenCV documentation) and check how many matches are considered inliers depending on it. The expected output is obtained with `ransacReprojThreshold=0.9`.

```
In [5]: # Compute fundamental matrix

# Create lists of corresponding keypoints
pts_l = []
pts_r = []
for match in matches:
    pts_l.append(kp_l[match.queryIdx].pt)
    pts_r.append(kp_r[match.trainIdx].pt)
```

```

# Cast to integers
pts_l = np.int32(pts_l)
pts_r = np.int32(pts_r)

# F, mask = cv2.findFundamentalMat(pts_l,pts_r,cv2.FM_LMEDS)
F, mask = cv2.findFundamentalMat(pts_l, pts_r, cv2.FM_RANSAC, 0.9, 0.99)

# keep only the inliers (matches)
matches_in = [matches[i] for i in range(len(mask)) if mask[i]==1]
pts_l = [kp_l[m.queryIdx].pt for m in matches_in]
pts_r = [kp_r[m.trainIdx].pt for m in matches_in]

# Cast to integers
pts_l = np.int32(pts_l)
pts_r = np.int32(pts_r)

print("Number of inliers =", len(matches_in), "out of", len(matches))
print("Fundamental matrix =")
print(F)

```

Number of inliers = 173 out of 277
Fundamental matrix =
[[-3.49654807e-07 -7.20777510e-05 1.99345790e-02]
[6.49820354e-05 2.66731011e-05 4.27861666e-01]
[-1.99589277e-02 -4.36546231e-01 1.00000000e+00]]

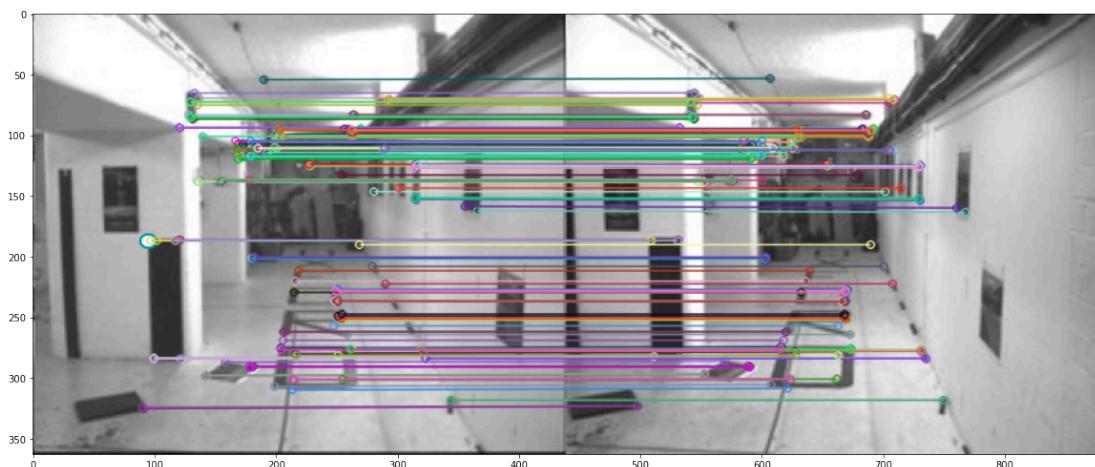
Check if your results are correct:

Expected output:

Number of inliers = 173 out of 277
Fundamental matrix =
[[-3.49654807e-07 -7.20777510e-05 1.99345790e-02]
[6.49820354e-05 2.66731011e-05 4.27861666e-01]
[-1.99589277e-02 -4.36546231e-01 1.00000000e+00]]

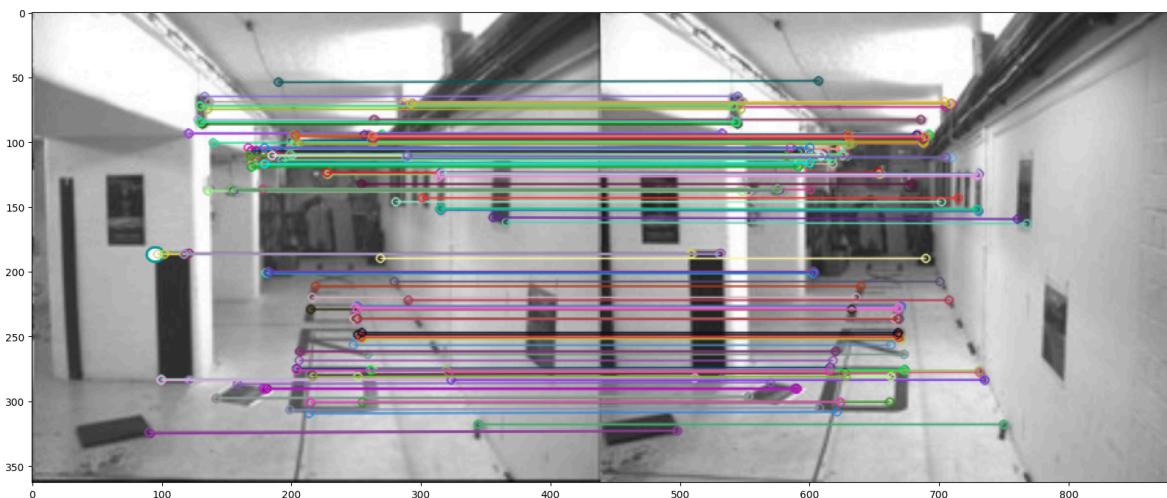
4. And finally, show the used matches!

You should get something like:



In [6]: # Plot matches
matches_image = np.copy(im_l)
matches_image = cv2.drawMatches(im_l, kp_l, im_r, kp_r, matches_in,

```
matches_image, flags=cv2.DrawMatchesFlags_NOT_DRAW_OVER_OUTIMG  
plt.imshow(matches_image);
```



ASSIGNMENT 2: Rectifying images!

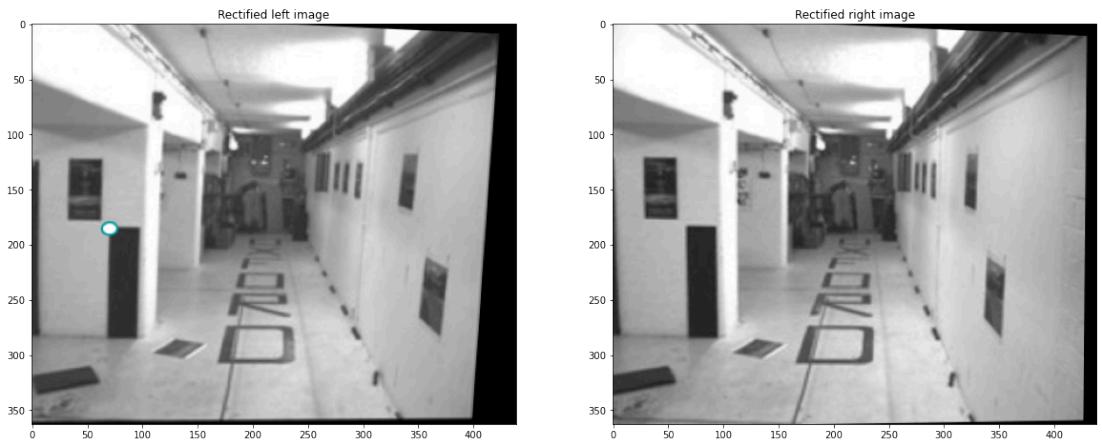
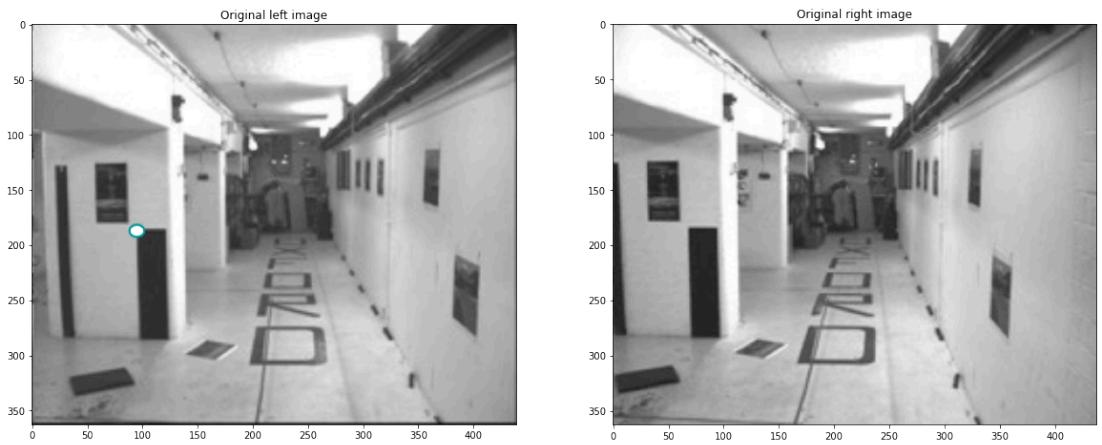
Now, we can **compute the homographies** needed to rectify the images with the method `cv2.stereoRectifyUncalibrated()`, which takes as input

- **the lists of matching points** used in the previous method,
- **the Fundamental matrix**,
- and **the shape of the images**.

and returns the computed homographies. Once you have them, you already know how to apply an homography to an image, right? Well, then do it!

What to do? Compute the homographies, use them to rectify your images and show them.

You should get something like:



In [7]:

```
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (20.0, 20.0)

# ASSIGNMENT 2
# Write your code here!
# Obtain homographies
ret, H_l, H_r = cv2.stereoRectifyUncalibrated(pts_l, pts_r, F, gray_r.shape)

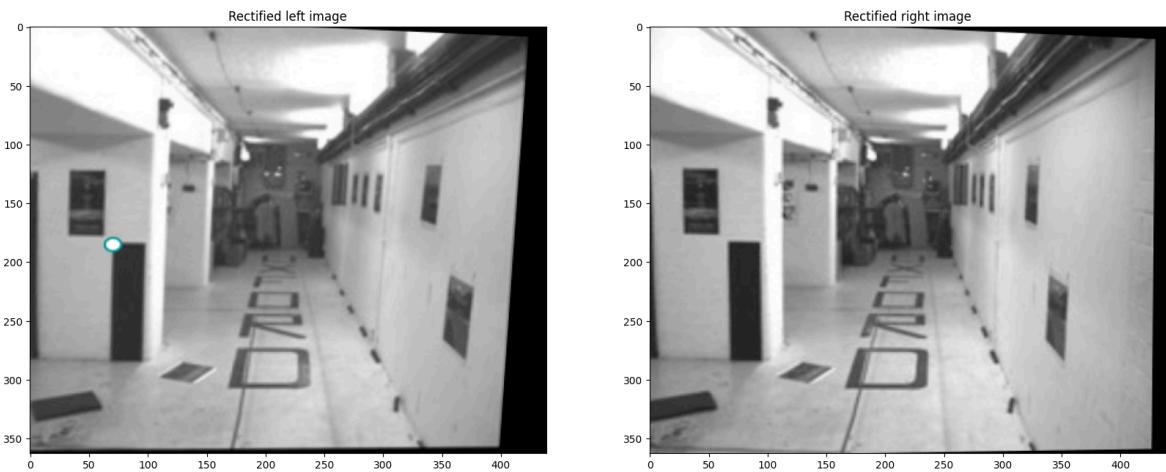
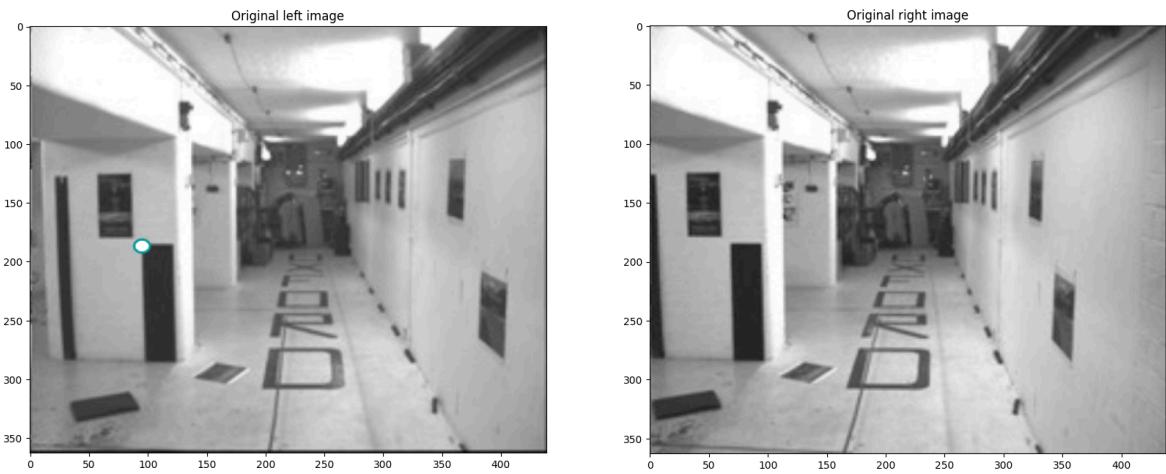
# Apply homographies
rectified_l = cv2.warpPerspective(im_l, H_l, (im_l.shape[1],im_l.shape[0]))
rectified_r = cv2.warpPerspective(im_r, H_r, (im_l.shape[1],im_l.shape[0]))

# Show rectified images
plt.subplot(221)
plt.title("Original left image")
plt.imshow(im_l)

plt.subplot(222)
plt.title("Original right image")
plt.imshow(im_r)

plt.subplot(223)
plt.title("Rectified left image")
plt.imshow(rectified_l)

plt.subplot(224)
plt.title("Rectified right image")
plt.imshow(rectified_r);
```



ASSIGNMENT 3: Checking the rectification

Now, plot the images side-by-side (i.e., concatenate them with `np.concatenate()`) and check that corresponding points fall in the same row! For that, just pick some points in the left image and draw a line on the same row in the right image.

You should get something like:



```
In [8]: # ASSIGNMENT 3
# Write your code here!
```

```

total = np.concatenate((rectified_l, rectified_r), axis=1)

_, offset = rectified_r.shape[:2]
size = 1
cont = 0
for p in pts_l:
    # for each point

        # apply left homography
    hom_p = np.vstack([p[0], p[1], 1]) # homogeneous point
    tran_hom_p = H_l @ hom_p

    x = np.int32(tran_hom_p[0]/tran_hom_p[2])
    y = np.int32(tran_hom_p[1]/tran_hom_p[2])

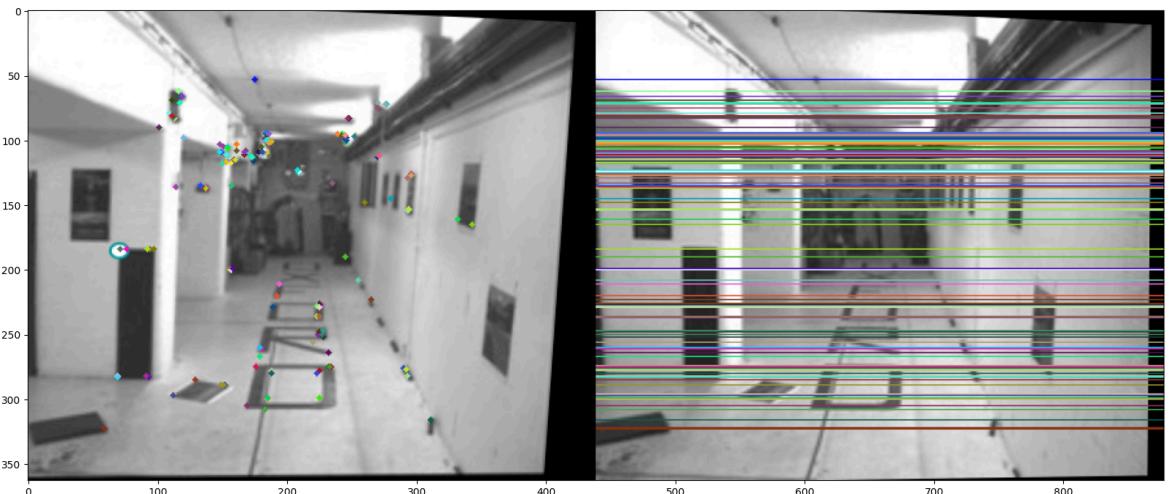
    if y > 0:
        cont+=1
        color = tuple(np.random.randint(0,255,3).tolist())
        cv2.circle(total,(int(x),int(y)), 2*size, color, -1)
        cv2.line(total, (int(offset),int(y)), (int(2*offset),int(y)), color, size)

plt.figure()
plt.imshow(total)
print("Number of points:", cont)

```

C:\Users\vital\AppData\Local\Temp\ipykernel_17020\1141877267.py:21: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
cv2.circle(total,(int(x),int(y)), 2*size, color, -1)
C:\Users\vital\AppData\Local\Temp\ipykernel_17020\1141877267.py:22: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
cv2.line(total, (int(offset),int(y)), (int(2*offset),int(y)), color, size)

Number of points: 173



In [9]: # Print homographies values
print("H_l = ")
print(H_l)

print("H_r = ")
print(H_r)

```

H_1 =
[[ -4.48201778e-01  2.02246376e-02  8.79537497e+00]
 [-1.94501778e-02 -4.36633942e-01  4.44143313e+00]
 [-6.31237657e-05 -2.52599836e-05 -4.15827289e-01]]
H_r =
[[ 1.02945316e+00 -9.58077348e-03 -3.24241377e+00]
 [ 4.48323907e-02  9.99626066e-01 -8.03314514e+00]
 [ 1.62963903e-04 -1.51665010e-06  9.70834163e-01]]

```

Check if your results are correct:

Expected output:

```

H_1 =
[[ -4.48201778e-01  2.02246376e-02  8.79537497e+00]
 [-1.94501778e-02 -4.36633942e-01  4.44143313e+00]
 [-6.31237657e-05 -2.52599836e-05 -4.15827289e-01]]
H_r =
[[ 1.02945316e+00 -9.58077348e-03 -3.24241377e+00]
 [ 4.48323907e-02  9.99626066e-01 -8.03314514e+00]
 [ 1.62963903e-04 -1.51665010e-06  9.70834163e-01]]

```

ASSIGNMENT 4: Computing the epipolar lines

Having the Fundamental matrix, **we can also calculate the epipolar lines** that correspond to any point, **even if the images are not rectified**. Remember that the Fundamental matrix encompasses the intrinsic geometric information between two views, regardless the scene! The OpenCV's method `cv2.computeCorrespondEpilines` facilitates this, having as inputs:

- `points` : set of N input points of which corresponding epipolar lines will be computed
- `whichImage` : index of the image (1 for left or 2 for right) that contains the points
- `F` : the Fundamental matrix

It returns the **coefficients of the epipolar lines** in a $(N \times 1 \times 3)$ matrix.

To show the epipolar lines, we have provided to you a method called `drawlines` (in `utils`) that plots the epilines in one image, and their corresponding keypoints (with the same color) in the other image, its inputs are:

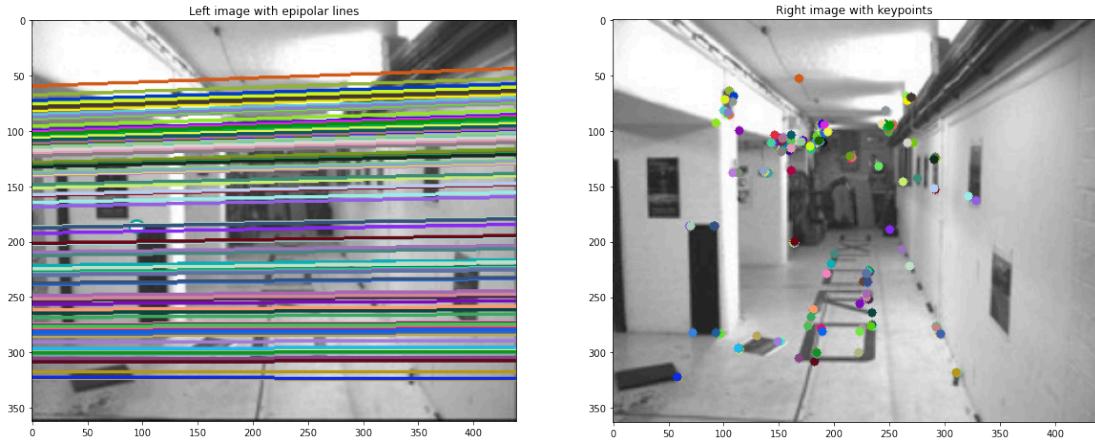
- `im_points` : input image in which the points are going to be drawn
- `pts` : array containing the points' coordinates $(N \times 2)$ of `im_points`
- `im_lines` : input image in which the epipolar lines are going to be drawn
- `lines` : coefficients of the epipolar lines drawn in `im_lines`
- `size` : size of the elements drawn (default 5). Change this if you cannot see the lines and points or they are too big.

It returns two images, `im_lines` (with the lines superimposed) and `im_points` (where the points have been drawn).

What to do? Calculate the **epipolar lines** of the keypoints previously detected in the right image. Then, **draw the epipolar lines in the left image** and the points in the right one.

Finally, display both images.

You should get something like:



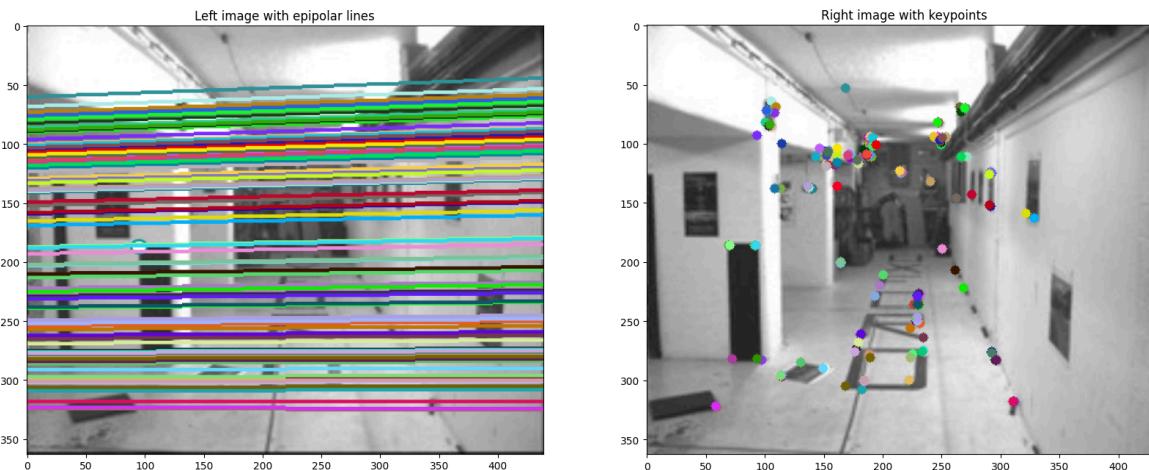
```
In [10]: # ASSIGNMENT 4
# Write your code here!
```

```
# Find epilines corresponding to points in right image
lines_l = cv2.computeCorrespondEpilines(pts_r.reshape(-1,1,2), 2, F)
lines_l = lines_l.reshape(-1,3)

# Draw epipolar lines
epipolar_l, epipolar_r = drawlines(im_r,pts_r,im_l,lines_l,2)

# Print images
plt.subplot(121)
plt.title('Left image with epipolar lines')
plt.imshow(epipolar_l)

plt.subplot(122)
plt.title('Right image with keypoints')
plt.imshow(epipolar_r);
```



Conclusion

Brilliant! Note that if we had a fixed stereo system, **the Fundamental matrix will be constant for any pair of images**. Anyway, once rectification have been applied, triangulation is more precise, and so would be the depth computation!

In this notebook, we have learnt:

- the fundamentals of epipolar geometry
- how to perform stereo rectification for uncalibrated and calibrated cameras
- how to find the Fundamental matrix from a pair of images