



IMAGE PROCESSING LABORATORY

Computer Vision Practical Session A.Y. 2016-2017

Camera Calibration - Stereovision

Antonino Furnari <http://www.dmi.unict.it/~furnari/> - furnari@dm.unict.it

Prof. Sebastiano Battiato <http://www.dmi.unict.it/~battiato/> - battiato@dm.unict.it

We recommend online consultation

In this practical session we will learn:

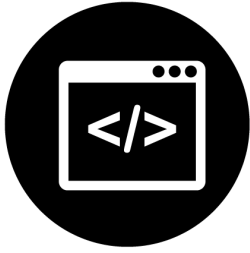
- How to calibrate a camera;
- How to undistort images using the intrinsic parameters;
- How to draw 3D objects on images using the extrinsic parameters;
- How to calibrate a stereo system;
- How to rectify a stereo image pair.

1. Conventions

The goal of this practical session is to guide the reader through the understanding of theoretical and practical concepts. To guide the understanding, the reader will be asked to answer some questions and do some exercises.



This icon indicates when the reader is asked to answer a question.



This icon indicates when the reader is asked to do an exercise.

1. Data

We will use the data included in the OpenCV samples, which are included in the source code of OpenCV. Specifically we will need the `left01.jpg` -- `left14.jpg` and `right01.jpg` -- `right14.jpg` image series located in the `samples/cpp/` directory. For convenience, you can also download the data [here](#). The images have been acquired using a stereo camera so that each `<leftxx.jpg-rightxx.jpg>` image pair represents the same scene. To start working with the images, extract the archive in the current directory.

Let's start loading and visualizing two corresponding images:

In [1]:

```
import cv2
im_left = cv2.imread('left01.jpg')
im_right = cv2.imread('right01.jpg')
```

Even if the images are actually grayscale, by default they are loaded as color images:

In [2]:

```
print im_left.shape
print im_right.shape
```

```
(480L, 640L, 3L)
```

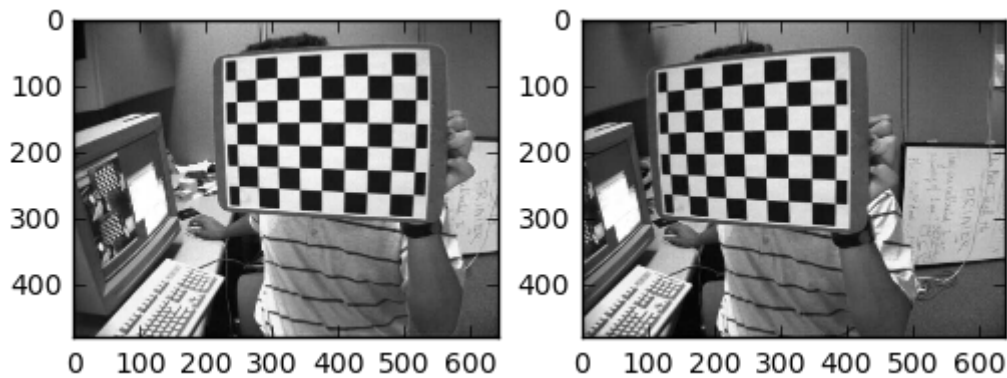
```
(480L, 640L, 3L)
```

We can now show the two images as follows:

In [3]:

```
from matplotlib import pyplot as plt

plt.subplot(121)
plt.imshow(im_left[...,::-1])
plt.subplot(122)
plt.imshow(im_right[...,::-1])
plt.show()
```



Question 1.1

The two images have been acquired using a stereo camera. What can we say about the baseline of the stereo camera? Is it small? Is it large?



Question 1.2

Each image of the dataset contains the same pattern (a checkerboard) seen from different points of view. Why is this data convenient for camera calibration? Why the pattern has to be acquired from different viewpoints?

2. Camera Calibration

Calibrating the camera basically means finding the two matrices:

$$M_{int} = \begin{pmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{pmatrix}, M_{ext} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix}$$



Question 2.1

How many parameters do we need to find? Why do we have two matrices? What is the meaning of each parameter? How can we "use" the two matrices once the camera has been calibrated?

To calibrate the camera, we need to establish correspondences a set of correspondences between 3D world points and 2D image points. Hence, we need to find the corners of the checkerboard in each image and relate them to the actual 3D coordinates in the arbitrary coordinated system associated to the checkerboard.

2.1 Finding Checkerboard Corners

Points can be easily found using the `cv2.findChessboardCorners` function:

In [4]:

```
ret, corners = cv2.findChessboardCorners(im_left, (7,6))
```

We had to specify (7,6) as `patternSize` parameter to specify the number of **inner** corners in the checkerboard. In practice, we are excluding incomplete rows and columns, as well as the first and last complete rows and columns in the checkerboard. This is done to obtain a more reliable detection.

The function returned a `ret` value which is set to `True` if the complete checkerboard was detected and `False` otherwise. The function also returns the list of the coordinates of detected corners **in the image plane**:

In [5]:

```
print corners.shape
```

```
(42L, 1L, 2L)
```

The `corners` variable contains 42 vectors of dimension 1×2 . Each vector represents the coordinates of a detected corner. For example we can see the values of the first corner as follows:

In [6]:

```
print corners[0]
```

```
[[ 475.44540405  264.75949097]]
```

As can be noted, `corners[0]` is an array containing an array of two values. This explicit shaping is probably derived from the C++ OpenCV api. To handle the `corners` array more easily, we can reshape it as follows:

In [7]:

```
corners=corners.reshape(-1,2)
print corners.shape
print corners[0]
```

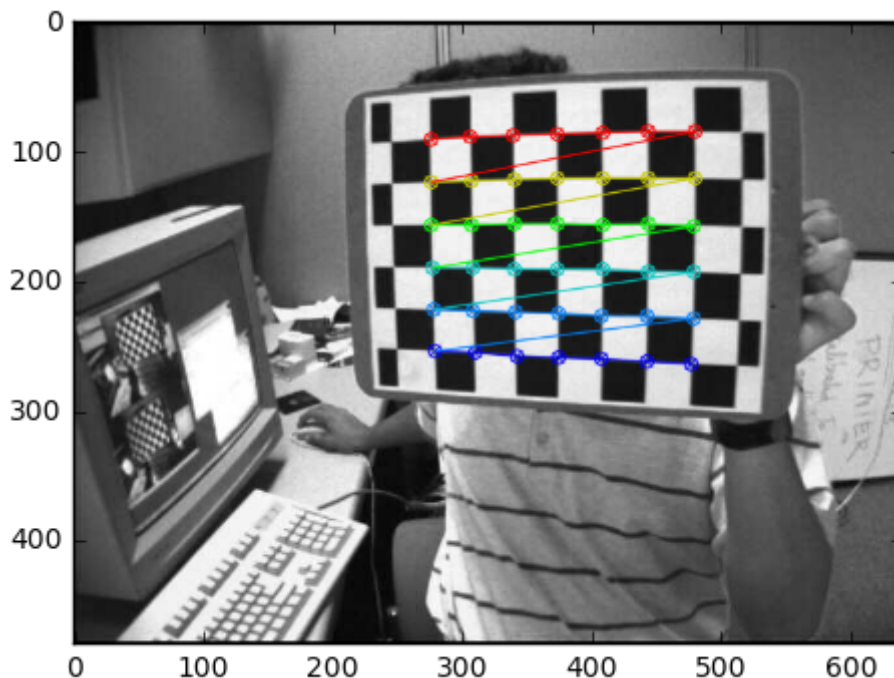
```
(42L, 2L)
```

```
[ 475.44540405  264.75949097]
```

We can now print the corners using the function `drawChessboardCorners`. Since this function would overwrite the content of `img1`, a good idea would be to create a copy for visualization only:

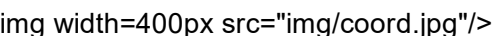
In [8]:

```
im_left_vis=im_left.copy()
cv2.drawChessboardCorners(im_left_vis, (7,6), corners, ret)
plt.imshow(im_left_vis)
plt.show()
```



Note that we had to pass the `ret` value to notify the `drawChessboardCorners` function if the checkerboard was completely or only partially detected. Moreover, as can be observed from the image, the 42 corners are detected from left to right, top to bottom.

2.2 Calibration

Now we need to specify a 3D coordinate system for each of the 42 detected points. We will choose the following coordinate system: 



Question 2.2

Could we choose another coordinate system? Why?

Therefore, the X and Y coordinates will vary, while the Z coordinate will be a constant zero.

```
[0,0,0]
[1,0,0]
[2,0,0]
...
[0,1,0]
[1,1,0]
...
[6,5,0]
```

To create such array, we first create a meshgrid which will give us all combinations of X-Y coordinates:

In [9]:

```
import numpy as np

x,y=np.meshgrid(range(7),range(6))
print "x:\n",x
print "y:\n",y
```

```
x:
[[0 1 2 3 4 5 6]
 [0 1 2 3 4 5 6]
 [0 1 2 3 4 5 6]
 [0 1 2 3 4 5 6]
 [0 1 2 3 4 5 6]
 [0 1 2 3 4 5 6]]
y:
[[0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1]
 [2 2 2 2 2 2 2]
 [3 3 3 3 3 3 3]
 [4 4 4 4 4 4 4]
 [5 5 5 5 5 5 5]]
```

Basically, every pair of values (x_{ij}, y_{ij}) represent a point in the X-Y space.

To obtain our vector, we first reshape the matrices x and y to obtain column vectors. Then, we stack the vectors vertically and add a vector with 42 zeros. We finally convert the array into an array of floats:

In [10]:

```
world_points=np.hstack((x.reshape(42,1),y.reshape(42,1),np.zeros((42,1)))).astype(np.float32)
print world_points
```

```
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 2.  0.  0.]
 [ 3.  0.  0.]
 [ 4.  0.  0.]
 [ 5.  0.  0.]
 [ 6.  0.  0.]
 [ 0.  1.  0.]
 [ 1.  1.  0.]
 [ 2.  1.  0.]
 [ 3.  1.  0.]
 [ 4.  1.  0.]
 [ 5.  1.  0.]
 [ 6.  1.  0.]
 [ 0.  2.  0.]
 [ 1.  2.  0.]
 [ 2.  2.  0.]
 [ 3.  2.  0.]
 [ 4.  2.  0.]
 [ 5.  2.  0.]
 [ 6.  2.  0.]
 [ 0.  3.  0.]
 [ 1.  3.  0.]
 [ 2.  3.  0.]
 [ 3.  3.  0.]
 [ 4.  3.  0.]
 [ 5.  3.  0.]
 [ 6.  3.  0.]
 [ 0.  4.  0.]
 [ 1.  4.  0.]
 [ 2.  4.  0.]
 [ 3.  4.  0.]
 [ 4.  4.  0.]
 [ 5.  4.  0.]
 [ 6.  4.  0.]
 [ 0.  5.  0.]
 [ 1.  5.  0.]
 [ 2.  5.  0.]
 [ 3.  5.  0.]
 [ 4.  5.  0.]
 [ 5.  5.  0.]
 [ 6.  5.  0.]
```

We now have our correspondences between 3D and 2D points, i.e., each row of `world_points` corresponds to a row of corners. We can show some of these correspondences:

In [11]:

```
print corners[0], '->', world_points[0]
print corners[35], '->', world_points[35]

[ 475.44540405  264.75949097] -> [ 0.  0.  0.]
[ 477.60940552   86.34799957] -> [ 0.  5.  0.]
```

We could already calibrate our camera using a single image, but this would likely result in an imprecise calibration. Therefore, we need to load all images, finding corners and creating two lists of corresponding 3D and 2D coordinates.

In [12]:

```
from glob import glob

_3d_points=[]
_2d_points=[]

img_paths=glob('*.jpg') #get paths of all all images
for path in img_paths:
    im=cv2.imread(path)
    ret, corners = cv2.findChessboardCorners(im, (7,6))

    if ret: #add points only if checkerboard was correctly detected:
        _2d_points.append(corners) #append current 2D points
        _3d_points.append(world_points) #3D points are always the same
```

To calibrate the camera, we can use the [cv2.calibrateCamera](#) function:

In [13]:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(_3d_points, _2d_points,
(im.shape[1],im.shape[0]))
print "Ret:",ret
print "Mtx:",mtx," -----> [",mtx.shape,"]"
print "Dist:",dist," -----> [",dist.shape,"]"
print "rvecs:",rvecs," -----> [",rvecs[0].shape,"]"
print "tvecs:",tvecs," -----> [",tvecs[0].shape,"]"
```

```

Ret: 0.524537950741
Mtx: [[ 533.67641112    0.          334.16207887]
 [ 0.          533.8040641  239.5316338 ]
 [ 0.          0.          1.          ]] -----
-----> [ (3L, 3L) ]
Dist: [[-0.31304359  0.22085038  0.00055066 -0.00114443 -0.18435855]] ---
-----> [ (1L, 5L) ]
rvecs: [array([[ -0.45450138],
 [ 0.26921647],
 [-3.08301824]]), array([[ 0.41360992],
 [ 0.67374122],
 [-1.33861046]]), array([[ -0.28482566],
 [-0.3723979 ],
 [-2.74608345]]), array([[ -0.40256361],
 [-0.16508729],
 [-3.11002863]]), array([[ -0.46590225],
 [-0.29351273],
 [-1.7587966 ]]), array([[ -0.29921519],
 [ 0.4073698 ],
 [-1.43266706]]), array([[ -0.31701638],
 [ 0.17847118],
 [-1.23616082]]), array([[ -0.46104095],
 [-0.07235785],
 [-1.33074758]]), array([[ -0.35428362],
 [-0.22265758],
 [-1.56731429]]), array([[ 0.48767265],
 [-0.16331655],
 [-1.73843409]]), array([[ -0.41437858],
 [ 0.24767265],
 [-3.09670779]]), array([[ -0.25770854],
 [-0.40584307],
 [-2.75391757]]), array([[ -0.12136293],
 [ 0.23179408],
 [-0.00935828]]), array([[ -0.30588363],
 [ 0.37737375],
 [-1.44049114]]), array([[ 0.64523052],
 [ 0.2878556 ],
 [-2.92785073]]), array([[ 0.52215575],
 [-0.45462528],
 [-1.70132633]]), array([[ -0.34908448],
 [-0.25936802],
 [-1.57661247]]), array([[ 0.31398162],
 [ 0.48562098],
 [-1.83241909]]), array([[ 0.4964921 ],
 [-0.19105458],
 [-1.73458594]])] -----
-----> [ (3L, 1L) ]
tvecs: [array([[ 4.00274793],
 [ 0.714909 ],
 [ 14.79811129]]), array([[ -1.94457219],
 [ 1.69343061],
 [ 12.85403145]]), array([[ 3.17372222],
 [ 2.62131208],
 [ 9.88139568]]), array([[ 2.970267 ],
 [ 2.09064088],
 [ 10.91857796]]), array([[ -1.005819 ],
 [ 2.55679502],
 [ 9.61864961]]), array([[ 1.82765836],
 [ 3.61735673],
 [ 16.09255423]]), array([[ -5.71345239],
 [ 2.18184018],
 [ 1.18184018]])]

```

```

[ 16.87737511]]), array([[ -3.11909148],
[ 2.00869959],
[ 11.76818562]]), array([[ -2.75218425],
[ 2.4921514 ],
[ 10.65070929]]), array([[ -1.46248285],
[ 3.45937613],
[ 12.16365378]]), array([[ 0.33246975],
[ 1.05230995],
[ 14.72918549]]), array([[ -0.38549616],
[ 2.86009238],
[ 9.79363835]]), array([[ -5.44769823],
[ -2.49808457],
[ 12.60695379]]), array([[ -1.8780948 ],
[ 3.98492211],
[ 15.94852228]]), array([[ -0.6271043 ],
[ 2.43645765],
[ 14.73788509]]), array([[ -4.48959289],
[ 3.58151311],
[ 11.25716545]]), array([[ -6.32265639],
[ 2.7734046 ],
[ 10.42408883]]), array([[ -4.70079135],
[ 3.7076963 ],
[ 15.50762798]]), array([[ -5.07629578],
[ 3.7624035 ],
[ 11.97560939]]]) -----
-----> [ (3L, 1L) ]

```

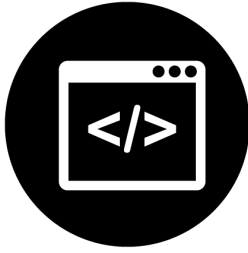
Note that we need to specify the dimensions of each image in the *width* \times *height* format. Since the shape attribute contains the number of **rows** and **columns**, the two numbers need to be inverted (rows=height, columns=width), i.e., using (im1.shape[1],im1.shape[0]) rather than (im1.shape[0],im1.shape[1]).

According to the [documentation](#), The function returns the following values:

- ret: the mean reprojection error (it should be as close to zero as possible);
- mtx: the matrix of intrinsic parameters;
- dist: the distortion parameters;
- rvecs: the rotation vectors (one per image);
- tvecs: the translation vectors (one per image).

Notes:

- Distortion parameters allow to define a model to explicitly remove distortion from images (see http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html#calibration);
- The rotation vector is a convenient way to represent a 3×3 rotation matrix (which has only 3 Degrees Of Freedom) and can be easily converted back to matrix using the [cv2.Rodrigues](#) function;



Exercise 2.1

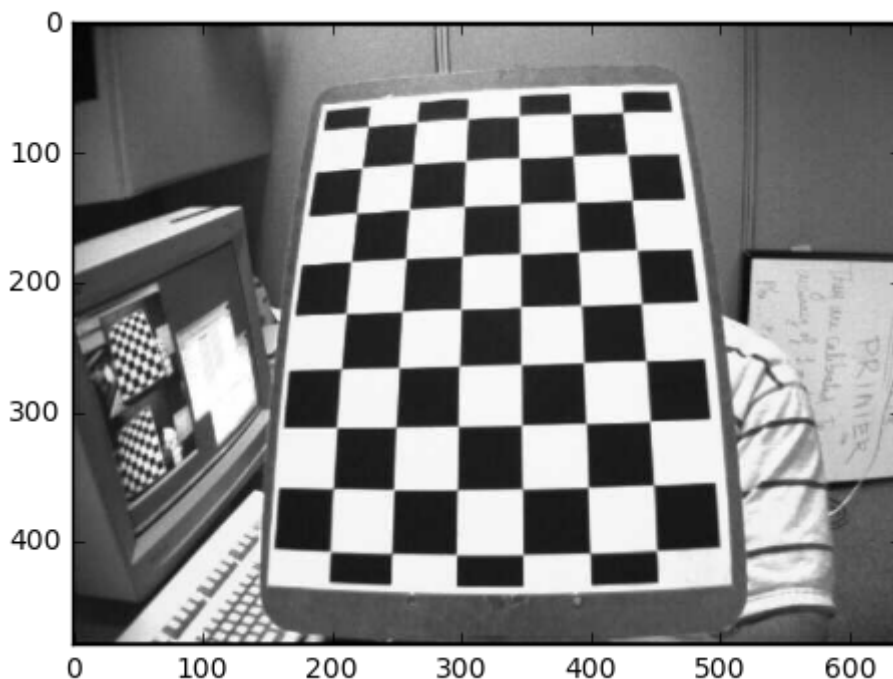
The output of the `cv2.calibrateCamera` allows to obtain both the matrix of intrinsic and extrinsic parameters. Define two variables `'Mint'` and `'Mext'` containing the two matrices and use them to project a point in the 3D space (i.e., a row of "world_points") to the image plane. Is the result close to the "ground truth" value contained in "corners"?

2.3 Image Rectification

While cameras are designed to adhere to the pinhole camera model, in practice, real cameras tend to deviate from it. The effect of such deviation is that some parts of the image tend to look "distorted", e.g., lines which we know should be straight, do not look so straight. For instance, let's visualize image `left12.jpg`:

In [14]:

```
plt.imshow(cv2.imread('left12.jpg')[...,:-1])  
plt.show()
```



Images tend to be affected by two different "kinds" of distortion:

- Radial distortion: lines far from the principal point look distorted;
- Tangential distortion: occurring because the lens is not perfectly parallel to the camera plane.

Radial distortion is modeled using the following relationship between the undistorted coordinates x_u and y_u and the distorted ones x and y :

$$\begin{aligned}x_u &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_u &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)\end{aligned}$$

where r is the "radius" of the point, (i.e., its distance from the principal point):
$$r^2 = (x - o_x)^2 + (y - o_y)^2$$

Tangential distortion is modeled using the following relationship:

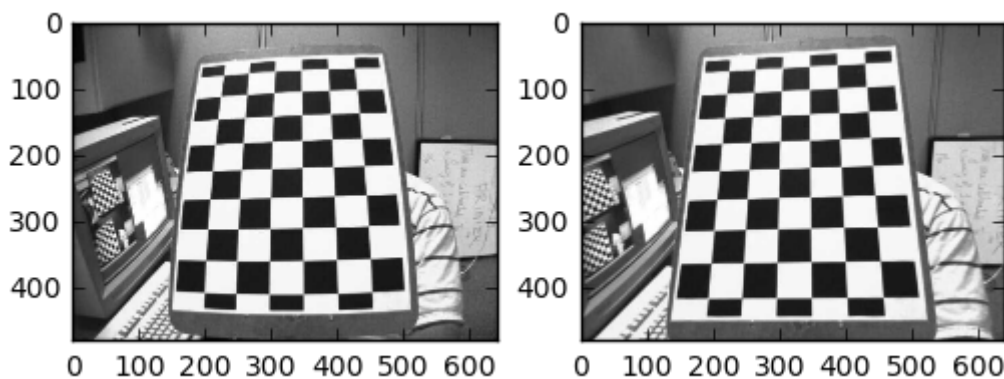
$$\begin{aligned}x_u &= x + [2p_1xy + p_2(r^2 + 2x^2)] \\y_u &= y + [p_1(r^2 + 2y^2) + 2p_2xy]\end{aligned}$$

In sum, we need five parameters to model camera distortion: $[k_1, k_2, k_3, p_1, p_2]$, which are the five values returned by the function `cv2.calibrateCamera` in the `dist` variable.

The image can be rectified using the `cv2.undistort` function:

In [15]:

```
im=cv2.imread('left12.jpg')[...::-1]
im_undistorted=cv2.undistort(im, mtx, dist)
plt.subplot(121)
plt.imshow(im)
plt.subplot(122)
plt.imshow(im_undistorted)
plt.show()
```



Question 2.3

Why the rectification process is important? What is the advantage of working on undistorted images?



Question 2.4

Suppose that a new image is acquired using the same camera we just calibrated. What steps are needed to rectify the image? Do we need the image to contain a checkerboard?

2.4 Drawing 3D points on the scene

Now that the camera is calibrated for both the extrinsic and intrinsic parameters, we can project points from the 3D world to the 2D image plane. This can be used, for instance, to implement "augmented reality" algorithms which draw 3D objects on the image. Let's see how to draw a cube on the checkerboard. First, define the 8 corners of a cube of side 3:

In [16]:

```
_3d_corners = np.float32([[0,0,0], [0,3,0], [3,3,0], [3,0,0],  
                          [0,0,-3],[0,3,-3],[3,3,-3],[3,0,-3]])
```

We can project points to the 2D image plane using the function [cv2.projectPoints](#). Since we need to project them on a specific image, we first need to choose one of the considered images:

In [17]:

```
image_index=7  
cube_corners_2d,_ =  
cv2.projectPoints(_3d_corners,rvecs[image_index],tvecs[image_index],mtx,dist)  
#the underscore allows to discard the second output parameter (see doc)  
  
print cube_corners_2d.shape #the output consists in 8 2-dimensional points  
  
(8L, 1L, 2L)
```

We can now plot lines on the 3D image using the [cv2.line](#) function:

In [18]:

```
img=cv2.imread(img_paths[image_index]) #Load the correct image

red=(0,0,255) #red (in BGR)
blue=(255,0,0) #blue (in BGR)
green=(0,255,0) #green (in BGR)
line_width=5

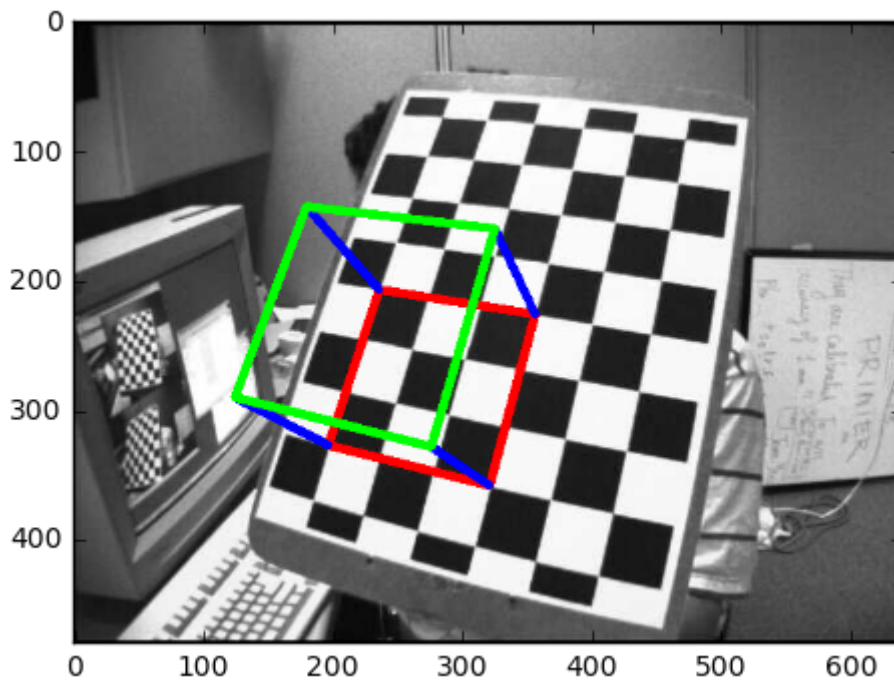
#first draw the base in red
cv2.line(img, tuple(cube_corners_2d[0][0]), tuple(cube_corners_2d[1]
[0]),red,line_width)
cv2.line(img, tuple(cube_corners_2d[1][0]), tuple(cube_corners_2d[2]
[0]),red,line_width)
cv2.line(img, tuple(cube_corners_2d[2][0]), tuple(cube_corners_2d[3]
[0]),red,line_width)
cv2.line(img, tuple(cube_corners_2d[3][0]), tuple(cube_corners_2d[0]
[0]),red,line_width)

#now draw the pillars
cv2.line(img, tuple(cube_corners_2d[0][0]), tuple(cube_corners_2d[4][0]),blue,line_widt
h)
cv2.line(img, tuple(cube_corners_2d[1][0]), tuple(cube_corners_2d[5][0]),blue,line_widt
h)
cv2.line(img, tuple(cube_corners_2d[2][0]), tuple(cube_corners_2d[6][0]),blue,line_widt
h)
cv2.line(img, tuple(cube_corners_2d[3][0]), tuple(cube_corners_2d[7][0]),blue,line_widt
h)

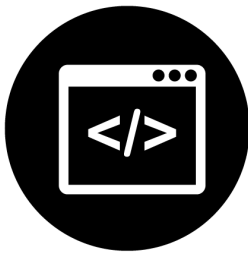
#finally draw the top
cv2.line(img, tuple(cube_corners_2d[4][0]), tuple(cube_corners_2d[5][0]),green,line_wid
th)
cv2.line(img, tuple(cube_corners_2d[5][0]), tuple(cube_corners_2d[6][0]),green,line_wid
th)
cv2.line(img, tuple(cube_corners_2d[6][0]), tuple(cube_corners_2d[7][0]),green,line_wid
th)
cv2.line(img, tuple(cube_corners_2d[7][0]), tuple(cube_corners_2d[4][0]),green,line_wid
th)

#cv2.line(img, tuple(start_point), tuple(end_point),(0,0,255),3) #we set the color to r
ed (in BGR) and line width to 3

plt.imshow(img[...,:-1])
plt.show()
```



Note that the side of the cube is equal to 3 cells of the checkerboard.



Excercise 2.2

Write a function which takes an image index as input and draws a cube on that image. Draw the cube on 4 different images and show them side by side using the `plt.subplot` function.

3. Stereovision

We will now calibrate the stereo system. This means finding the essential matrix and the fundamental matrix of the system. To calibrate the camera, we need to find corresponding points in left-right image pairs. To do this, we should make sure that the `cv2.findChessboardCorners` assigns the same coordinate system to both images in the pair, which is not always guaranteed. To make things simple, in this practical session, we will work only the image pairs with indices 1, 3, 6, 12, 14, where we tested in advance that corners are detected coherently.

3.1 Calibration

First, we need to collect all corners from right and left images:

In [19]:

```
all_right_corners=[]
all_left_corners=[]
all_3d_points=[]
idx=[1, 3, 6, 12, 14] #we use only some image pairs
valid_idxes=[] #we will also keep an list of valid indices, i.e., indices for which the
procedure succeeded
for i in idx:
    im_left=cv2.imread("left%02d.jpg"%i) #load left and right images
    im_right=cv2.imread("right%02d.jpg"%i)

    ret_left,left_corners=cv2.findChessboardCorners(im_left,(7,6))
    ret_right,right_corners=cv2.findChessboardCorners(im_right,(7,6))

    if ret_left and ret_right: #if both extraction succeeded
        valid_idxes.append(i)
        all_right_corners.append(right_corners)
        all_left_corners.append(left_corners)
        all_3d_points.append(world_points)
```

Let's see some statistics on the data:

In [20]:

```
print len(all_right_corners)
print len(all_left_corners)
print len(all_3d_points)

print all_right_corners[0].shape
print all_left_corners[0].shape
print all_3d_points[0].shape

print all_right_corners[0].reshape(-1,2)[0]
```

```
5
5
5
(42L, 1L, 2L)
(42L, 1L, 2L)
(42L, 3L)
[ 346.26754761  278.19436646]
```

To calibrate the stereo system we now use the [`cv2.stereoCalibrate`](#) function:

In [21]:

```
retval, _, _, _, _, R, T, E, F=cv2.stereoCalibrate(all_3d_points, all_left_corners, al
l_right_corners, (im.shape[1],im.shape[0]),mtx,dist,mtx,dist,flags=cv2.cv.CV_CALIB_FIX_
INTRINSIC)
```

The function requires the following inputs:

- `all_3d_points`: list of 3D points;
- `all_left_corners`: list of corresponding 2D points in the first image (left image);
- `all_right_corners`: list of corresponding 2D points in the second image (right image);
- `(im.shape[1], im.shape[0])`: image size (width,height);
- `mtx`: matrix of intrinsic parameters for the first image (left image);
- `dist`: matrix of distortion parameters for the first image (left image);
- `mtx`: matrix of intrinsic parameters for the second image (right image). We assume these parameters to be the same as the first image;
- `dist`: matrix of distortion parameters for the second image (right image). We assume these parameters to be the same as the first image;
- flag `cv2.CV_CALIB_FIX_INTRINSIC` to tell the system to use the precomputed intrinsic parameters.

The function returns the following values:

- `retval`: mean reprojection error;
- `cameraMatrix1`, `distCoeffs1`, `cameraMatrix2`, `distCoeffs2`: we are discarding these values because they would be equal to `mtx` and `dist`;
- `R`: output rotation matrix between the 1st and the 2nd camera coordinate systems;
- `T`: output translation vector between the coordinate systems of the cameras;
- `E`: essential matrix
- `F`: fundamental matrix

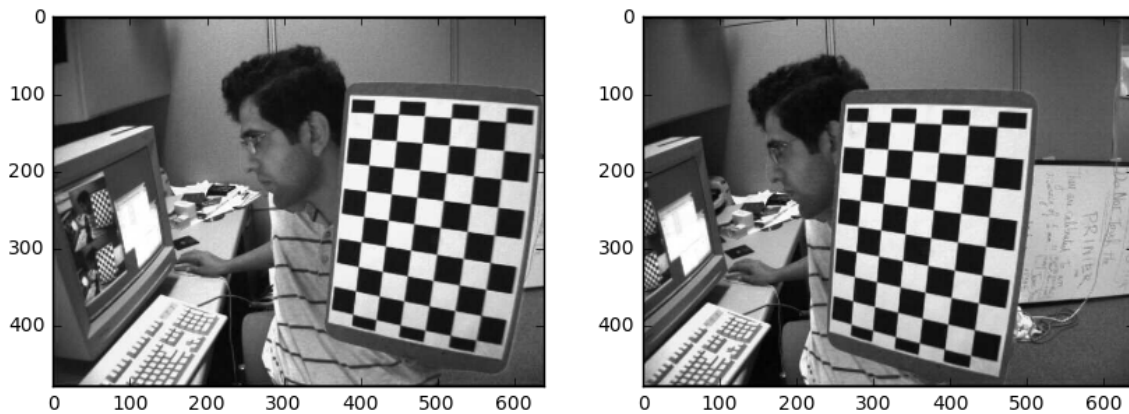
3.2 Epipolar Lines

Let's consider a given pair of stereo images, e.g.:

In [22]:

```
selected_image=2
left_im=cv2.imread("left%02d.jpg"%valid_idx[selected_image])
right_im=cv2.imread("right%02d.jpg"%valid_idx[selected_image])
left_corners=all_left_corners[selected_image].reshape(-1,2)
right_corners=all_right_corners[selected_image].reshape(-1,2)

plt.figure(figsize=(10,4))
plt.subplot(121)
plt.imshow(left_im)
plt.subplot(122)
plt.imshow(right_im)
plt.show()
```

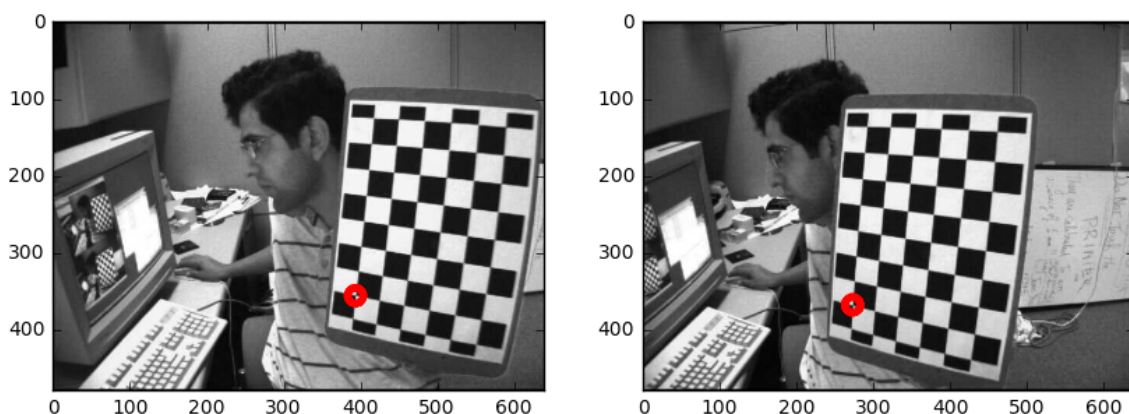


Given a point on the left image, the matching point in the right one will lie on the corresponding epipolar line. In general, for each point in **one of the two images** it is possible to find the corresponding epipolar line in **the other image**. Let's consider for instance the first corner detected in the left image and its corresponding point in the right one:

In [23]:

```
cv2.circle(left_im,(left_corners[0,0],left_corners[0,1]),10,(0,0,255),10)
cv2.circle(right_im,(right_corners[0,0],right_corners[0,1]),10,(0,0,255),10)

plt.figure(figsize=(10,4))
plt.subplot(121)
plt.imshow(left_im[...,:-1])
plt.subplot(122)
plt.imshow(right_im[...,:-1])
plt.show()
```



We can compute the epipolar lines corresponding to points of the left image using the function `cv2.computeCorrespondEpilines`. The lines will lie on the right image:

In [24]:

```
lines_right = cv2.computeCorrespondEpilines(all_left_corners[selected_image], 1,F)
print lines_right.shape
lines_right=lines_right.reshape(-1,3) #reshape for convenience
print lines_right.shape
```

```
(42L, 1L, 3L)
(42L, 3L)
```

Note that, apart from the corners and fundamental matrix, we had to specify an index "1" to signal that points belong to the first (left) image. Each of the 42 lines of `lines_right` contains three values, specifying the coefficients of the equation:

$$ax + by + c = 0$$

To simplify the visualization of such lines, we can write the following function:

In [25]:

```
def drawLine(line,image):
    a=line[0]
    b=line[1]
    c=line[2]

    #ax+by+c -> y=(-ax-c)/b
    #define an inline function to compute the explicit relationship
    def y(x): return (-a*x-c)/b

    x0=0 #starting x point equal to zero
    x1=image.shape[1] #ending x point equal to the last column of the image

    y0=y(x0) #corresponding y points
    y1=y(x1)

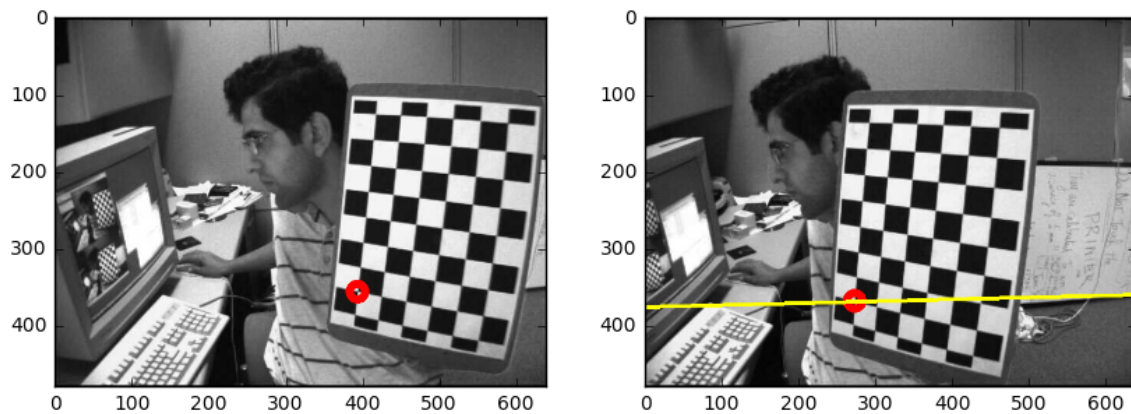
    #draw the line
    cv2.line(image,(x0,int(y0)),(x1,int(y1))),(0,255,255),3)#draw the image in yellow with line_width=3
```

We can now use the function to draw the corresponding epipolar line:

In [26]:

```
drawLine(lines_right[0],right_im)

plt.figure(figsize=(10,4))
plt.subplot(121)
plt.imshow(left_im[...,:-1])
plt.subplot(122)
plt.imshow(right_im[...,:-1])
plt.show()
```



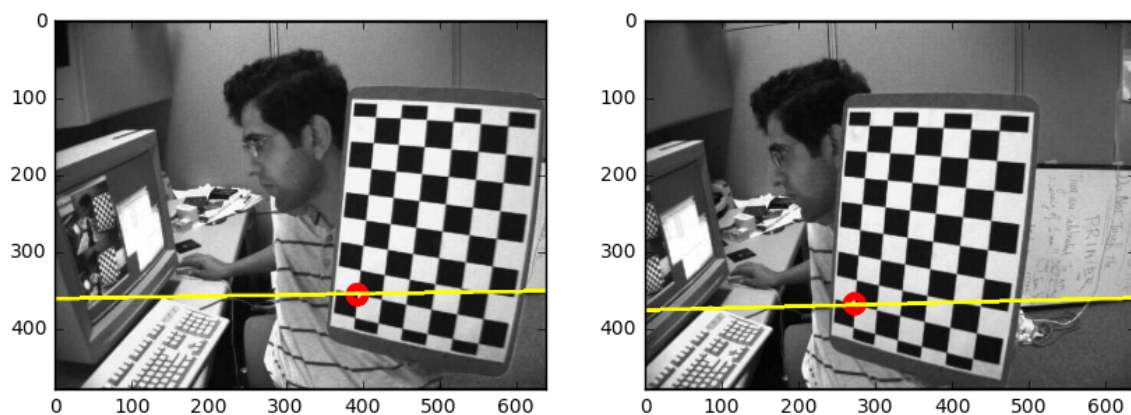
Let's now compute the epipolar lines for the left image and print the first of them:

In [27]:

```
lines_left = cv2.computeCorrespondEpilines(all_right_corners[selected_image], 2,F)
lines_left=lines_left.reshape(-1,3)

drawLine(lines_left[0],left_im)

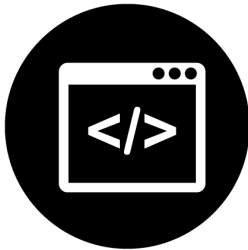
plt.figure(figsize=(10,4))
plt.subplot(121)
plt.imshow(left_im[...,:-1])
plt.subplot(122)
plt.imshow(right_im[...,:-1])
plt.show()
```





Question 3.1

What do the epipolar lines represent with respect to the others? Are the lines convincing? Why?



Exercise 3.1

Write the code to display all points and corresponding epipolar lines on both images.

3.3 Rectification

We can now rectify the stereo pair using the function `cv2.cv.StereoRectify`. Unluckily, the function uses the old cv API (rather than the modern cv2 one), so we will need some extra code (e.g., format conversions) to make it work properly:

In [28]:

```
R1=cv2.cv.fromarray(np.zeros((3,3))) #output 3x3 matrix
R2=cv2.cv.fromarray(np.zeros((3,3))) #output 3x3 matrix
P1=cv2.cv.fromarray(np.zeros((3,4))) #output 3x4 matrix
P2=cv2.cv.fromarray(np.zeros((3,4))) #output 3x4 matrix

roi1,roi2=cv2.cv.StereoRectify(cv2.cv.fromarray(mtx), #intrinsic parameters of the first camera
    cv2.cv.fromarray(mtx), #intrinsic parameters of the second camera
    cv2.cv.fromarray(dist), #distortion parameters of the first camera
    cv2.cv.fromarray(dist), #distortion parameters of the second camera
    (left_im.shape[1],left_im.shape[0]), #image dimensions
    cv2.cv.fromarray(R), #Rotation matrix between first and second cameras (returned by cv2.stereoCalibrate)
    cv2.cv.fromarray(T), #Translation vector between coordinate systems of the cameras (returned by cv2.stereoCalibrate)
    R1,R2,P1,P2) #last 4 parameters point to initialized output variables

R1=np.array(R1) #convert output back to numpy format
R2=np.array(R2)
P1=np.array(P1)
P2=np.array(P2)
```

The function returns the following values:

- R1: Output 3x3 rectification transform (rotation matrix) for the first camera.
- R2: Output 3x3 rectification transform (rotation matrix) for the second camera.
- P1: Output 3x4 projection matrix in the new (rectified) coordinate systems for the first camera.
- P2: Output 3x4 projection matrix in the new (rectified) coordinate systems for the second camera.

In practice, the matrices allow to bring both images on the same image plane through the transformations defined by the R and P matrices. In the common image plane, all epipolar lines are parallel.

To perform rectification, we first need to define two maps, i.e.,

- a map from the image plane of the left camera to the common image plane;
- a map from the image plane of the right camera to the common image plane.

This is done using the `cv2.initUndistortRectifyMap` function:

In [29]:

```
map1_x,map1_y=cv2.initUndistortRectifyMap(mtx, dist, R1, P1,
(left_im.shape[1],left_im.shape[0]), cv2.cv.CV_32FC1)
map2_x,map2_y=cv2.initUndistortRectifyMap(mtx, dist, R2, P2,
(left_im.shape[1],left_im.shape[0]), cv2.cv.CV_32FC1)
```

The function takes the following inputs:

- `mtx`: the intrinsic parameters of the input camera;
- `dist`: the distortion parameters of the input camera;
- `R`, `P`: the rectification and projection matrices of the camera we want to rectify;
- the image dimensions;
- the type (float) of the maps and returns:
- `map_x`: the map of x coordinates;
- `map_y`: the map of y coordinates.

Now we can use the `cv2.remap` function to warp the images to the reference image plane:

In [30]:

```
im_left=cv2.imread('left07.jpg')
im_right=cv2.imread('right07.jpg')

im_left_remapped=cv2.remap(im_left,map1_x,map1_y,cv2.INTER_CUBIC)
im_right_remapped=cv2.remap(im_right,map2_x,map2_y,cv2.INTER_CUBIC)
```

The `cv2.remap` function remaps the input image using the specified maps:

$$out(x, y) = in(map_x(x, y), map_y(x, y))$$

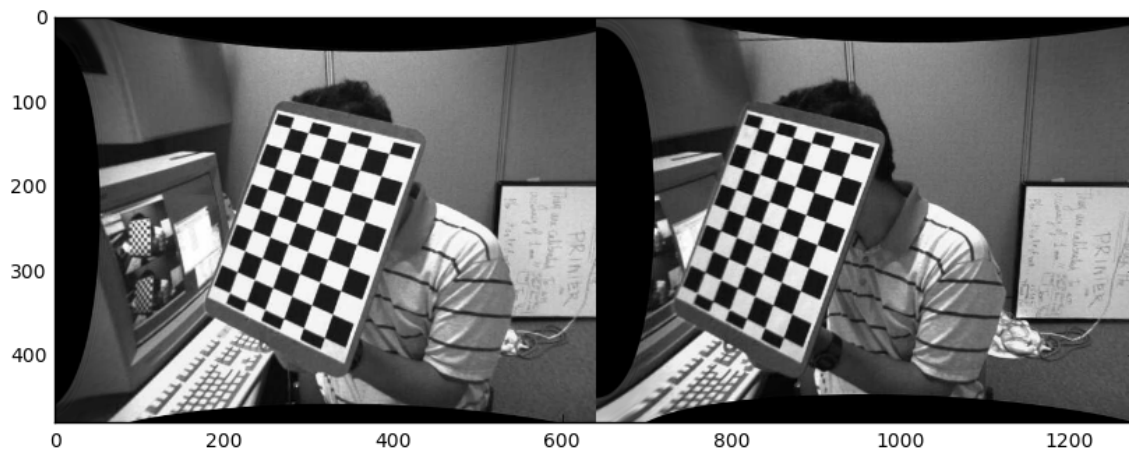
The `cv2.INTER_CUBIC` specifies to use the cubic interpolation.

We can now display rectified images side by side:

In [31]:

```
out=np.hstack((im_left_remapped,im_right_remapped))

plt.figure(figsize=(10,4))
plt.imshow(out[...,::-1])
plt.show()
```

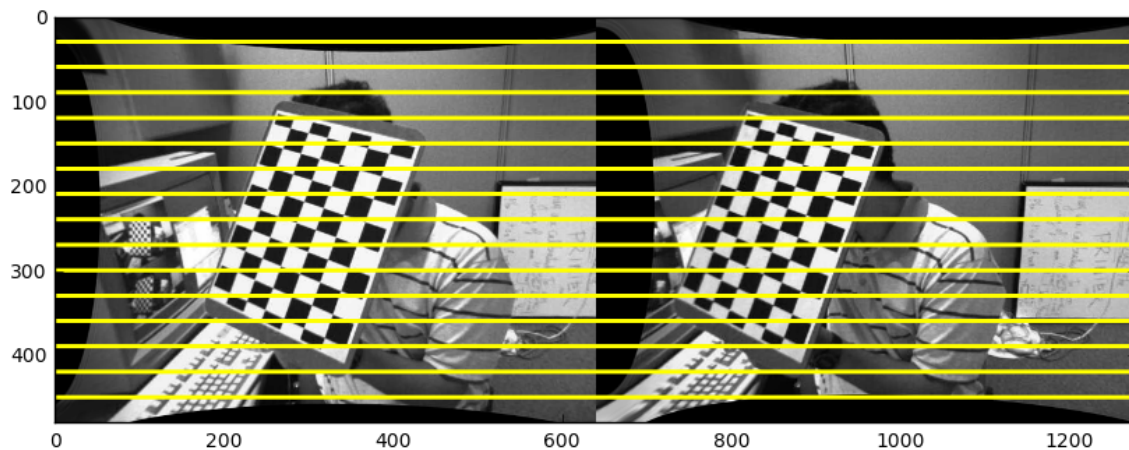


We can also display some epipolar lines:

In [32]:

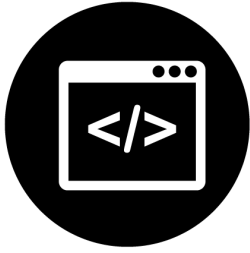
```
for i in range(0,out.shape[0],30):
    cv2.line(out,(0,i),(out.shape[1],i),(0,255,255),3)

plt.figure(figsize=(10,4))
plt.imshow(out[...,::-1])
plt.show()
```



Question 3.2

What is the advantage of having a rectified stereo pair?



Excercise 3.2

Draw two images from a **non-rectified** stereo pair side by side along with horizontal lines (as done above). What is the difference with respect to the rectified stereo pair?



Excercise 3.3

Rectify and visualize another stereo pair as done above. The epipolar lines are convincing also in this case?

4. References

[1] Camera calibration tutorial for Python+OpenCV: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html#calibration

[2] Pose estimation tutorial for Python+OpenCV: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_pose/py_pose.html#pose-estimation

[3] Epipolar geometry tutorial for Python+OpenCV: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_epipolar_geometry/py_epipolar_geometry.html#geometry

