Robot Perception

General Information:

Please do not add or delete any cells. Answers belong into the corresponding cells (below the question). If a function is given (either as a signature or a full function), you should not change the name, arguments or return value of the function.

If you encounter empty cells underneath the answer that can not be edited, please ignore them, they are for testing purposes.

When editing an assignment there can be the case that there are variables in the kernel. To make sure your assignment works, please restart the kernel and run all cells before submitting (e.g. via *Kernel -> Restart & Run All*).

Code cells where you are supposed to give your answer often include the line ```raise NotImplementedError```. This makes it easier to automatically grade answers. If you edit the cell please outcomment or delete this line.

The server resource is limited to 2 core cpu and 1GB RAM at max per user. If you use more than that, the kernel may die. Nevertheless, you can bring it up again by restaring the kernel (Kernel -> Restart and clear output).

Submission:

Upload all attachments required to run the notebook and provide a correct path to them.

~~Please submit your notebook via the web interface (in the main view -> Assignments -> Submit)~~.

Starting from RP-HW07 onwards, submit the notebook and pdf version of it via LEA.

The assignments are due on **Monday at 0:00.** (i.e. Sunday 23:59 + 1 min)

Group Work:

You are allowed to work in groups of up to two people. Please enter the UID (your username here) of each member of the group into the next cell. We apply plagiarism checking, so do not submit solutions from other people except your team members. If an assignment has a copied solution, the task will be graded with 0 points for all people with the same solution.

YOU SHOULD ONLY SUBMIT EXACTLY ONE PER GROUP

Questions about the Assignment:

If you have questions about the assignment please post them in the LEA forum before the deadline. Don't wait until the last day to post questions.

Put your answer in the PROVIDED CELLS only!
Any new cell is not visible during the grading.

We provide additional code and markdown cells for each question, so that you do not have to add the new ones.

Do not copy the metadata from one cell to another as it is unique to that cell only.

In [1]:

```
'''
Group Work:
Enter the UID (i.e. student2s) of each team member into the variables.
If you work alone please leave the second variable empty, or extend this list i
'''
member1 = 'dpadma2s'
member2 = 'jbandl2s'
member3 = 'smuthi2s'
```

Q&As: Pose three questions (and ans.) to last lecture, 3 Q&As per member [1 point]

The format of the question and answer should be [Q1,A1,Q2,A2,Q3,A3,...,Qn,An], where Q1 is the question and A1 is the answer.
If you work in a group, the total of the Q&As is $3xn$, where n is the total number of members.

Put your answer in the provided cell below!
If you work in a group, you can extend the provided Q&A template, but please use the same format:

1. Q and A are separated by ONE
2. Q&A and other Q&As are separated by TWO

Or copy the provided format in the answer cell, and change the Q&A number.
Do not remove any markdown tag like
 in the answer cell.

Q1 = What are normalized coordinates and normalized camera matrices?
A1 = The camera matrix P for image 1 is given by, $P = K[R|t]$ and image point $x = PX$. K is the intrinsic parameter matrix, R is the rotation matrix and t is the translation matrix. If K is known, the inverse of K is applied to an image point x to obtain $\hat{x} = K^{-1}x$. Then, $\hat{x} = [R|t]X$, where $\hat{x}$ is an image point in the normalized coordinates. The camera matrix, $K^{-1}P = [R|t]$ is called normalized camera matrix, removing the effect of the known calibration matrix.

Q2 = What is Fundamental matrix, F and Essential matrix, E?
A2 =

Fundamental matrix:

The fundamental matrix, F is a 3x3 rank 2 matrix mapping a point in image 1 to the corresponding line l' in image 2. ie., $l' = Fx$. Since the point x' corresponding to x necessarily lies on l', we know that $x'^T l' = 0$ and therefore, $x'^T Fx = 0$. This allows us to compute F from a set of point correspondences $\{x_i \leftrightarrow x_i'\}$

Essential matrix:

The essential matrix, E is a specialization to the case of normalized image coordinates. The camera matrix P for image 1 is given by, $P = K[R|t]$ and image point $x = PX$. K is the intrinsic parameter matrix, R is the rotation matrix and t is the translation matrix. If K is known, the inverse of K is applied to an image point x to

obtain $\hat{x} = K^{-1}x$. Then, $\hat{x} = [R|t]X$, where $\hat{x}$ is an image point in the normalized coordinates. The camera matrix, $K^{-1}P = [R|t]$ is called normalized camera matrix, removing the effect of the known calibration matrix. On considering, a pair of normalized camera matrices, $P = [I|0]$ and $P' = [R|t]$. The fundamental matrix corresponding to the pair of normalized cameras is the essential matrix.

$$E = [t]_x R = R[R^t]_x$$

Therefore, E is given by $\hat{x}'^T E\hat{x} = 0$ where, $\hat{x}'$ and $\hat{x}$ are normalized image cooridinates for the corresponding points $x \leftrightarrow x'$ on image 1 and image 2.

Q3 = Give the degrees of freedom (DOF) of camera matrix, fundamental matrix and homography.
A3 =

Camera matrix, P: 11 DOF

- 3 x 4 matrix, total 12 elements in the matrix.
- One of the element can be used for scaling and normalizing, therefore a total of 11 DOF.

Fundamental matrix, F: 7 DOF

- 3 x 3, rank 2 matrix, total 9 elements in the matrix.
- One of the element can be used for scaling and normalizing, therefore, now 8 elements.
- Since F is a rank 2 matrix mapping a point in image 1 to line in image 2, it looses another parameter resulting in a total of 7 DOF.

Homography, H: 15 DOF

- 4 x 4 matrix, total 16 elements in the matrix.
- One of the element can be used for scaling and normalizing, therefore a total of 15 DOF.

Q4 = Why is F also known as correlation ? Is it a proper correlation ?
A4 = A correlation is defined as an inverse mapping from points in IP^2 to lines in IP^2. In case of Fundamental matrix F, a point x in the first image defines a line in the second image i,e l = Fx, which is the epipolar line of x. If there exists another l' = $F^T x'$ such that l and l' are corresponding epipolar lines which makes F a correlation. But this doesn't hold true for any point on the image and is restrocited to points on the epipolar lines. Hence, it's not a proper correlation.

Q5 = How can a 3D points be projected in case of a rigid motion ? How does knowledge of parameters effect the reconstruction ?
A5 = A point X subjected to a rigid body motion is represented as $X' = M_p X$, Where M_p is a projective transformation matrix $M_p = [R|t]$.

The final projection in homogeneous form looks as follows

$$\boldsymbol{x} = \hom^{-1}\left[\mathbf{M_P} \cdot \underline{\boldsymbol{X}'}\right] = \hom^{-1}\left[\mathbf{M_P} \cdot \mathbf{M}_{\text{rb}} \cdot \hom(\boldsymbol{X})\right]$$

Considering a ideal pin-hole camera model, with focal lenght 'f'

$$\begin{pmatrix} x \\ y \end{pmatrix} = \hom^{-1}\left[\begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}\right]$$

=

$$= \hom^{-1}\left[\mathbf{M}_f \cdot (\mathbf{R}|t) \cdot \hom(X)\right]$$

Incase, f = 1, The projection becomes a normalized projection.

Incase f is unknown, occurence of projective ambiguity is resulting in a scene reconstruction upto only projective transformation. But not on to the camera image plane.

Q6 = What is metric reconstruction ?
A6 = Metric reconstruction is same as similarirty reconstruction, owing to a fact that in metric reconstruction we try to perform the reconstruction based on the angles between lines and ratios of lengths which can be considered as similarity invariants. The similarity reconstruction is also called Euclidean reconstruction when the invariants used for matching are length and angle measurements.

Q7 = What is projective reconstruction theorem?
A7 = If the point correspondence set F is uniquely determined in the two views, the scene can be reconstructed, and both such reconstructions are projective.

Q8 = What is reconstruction ambiguity?
A8 = We do not get latitude and longitude from the images From images alone we cannot fix the size or scale of the scene.

Q9 = What are the properties of F matrix?
A9 = The fundamental matrix is of rank 2. Its kernel defines the epipole.

In [2]:

```
'''
Provide the time required to solve the assignment per task as well as the sum (
Extend this list if needed.
'''

Task1 = 300
Task2 = 120
Task3 = 180
Task4 = 120
Sum = 0

```

Task1: Read this MATLAB explanation (https://de.mathworks.com/help/vision/examples/measuring-planar-objects-with-a-calibrated-camera.html). Rehearse this experiment using your own smart phone camera (or some BETTER camera if available) , in openCV (if possible) and use a different object (like a book, match box etc). Then:

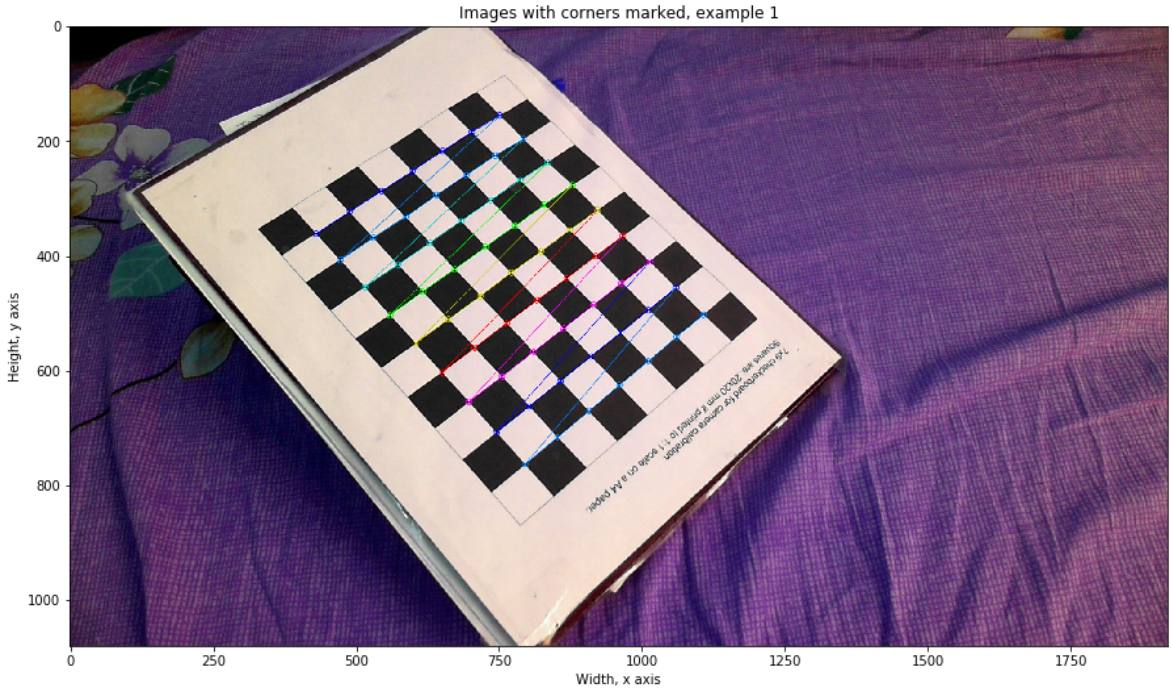
- Compare your result to the measurements done on the real object. How large is the difference?

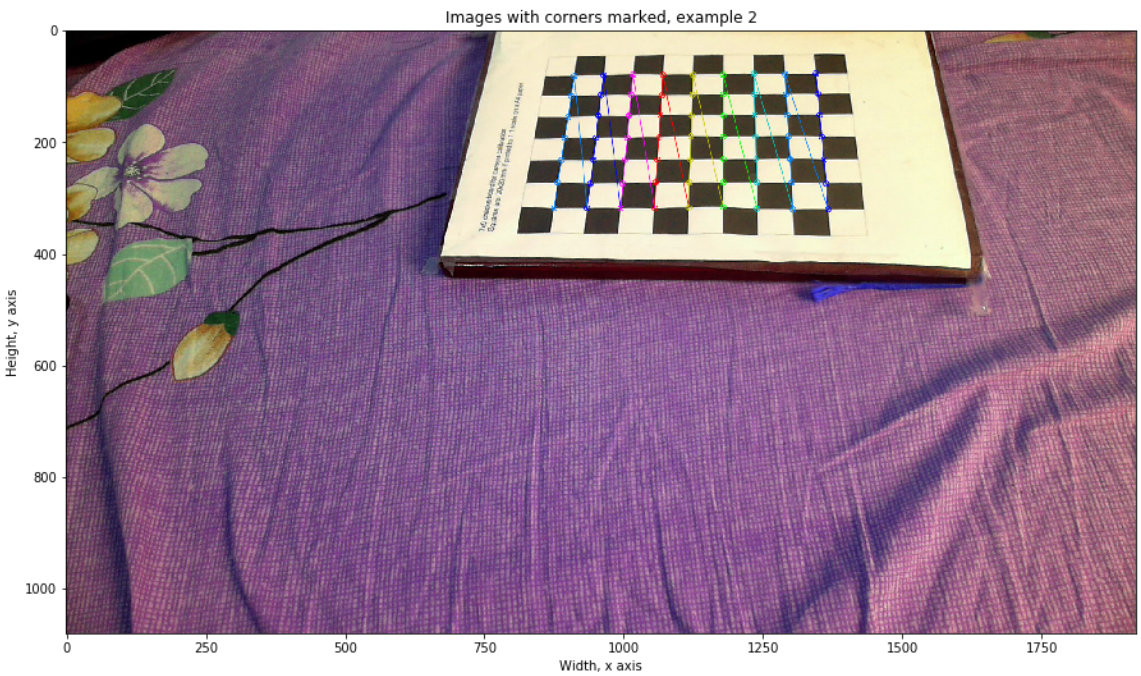
- Taking your observed error, is this lay inside the error bounds produced by the calibration algorithm?

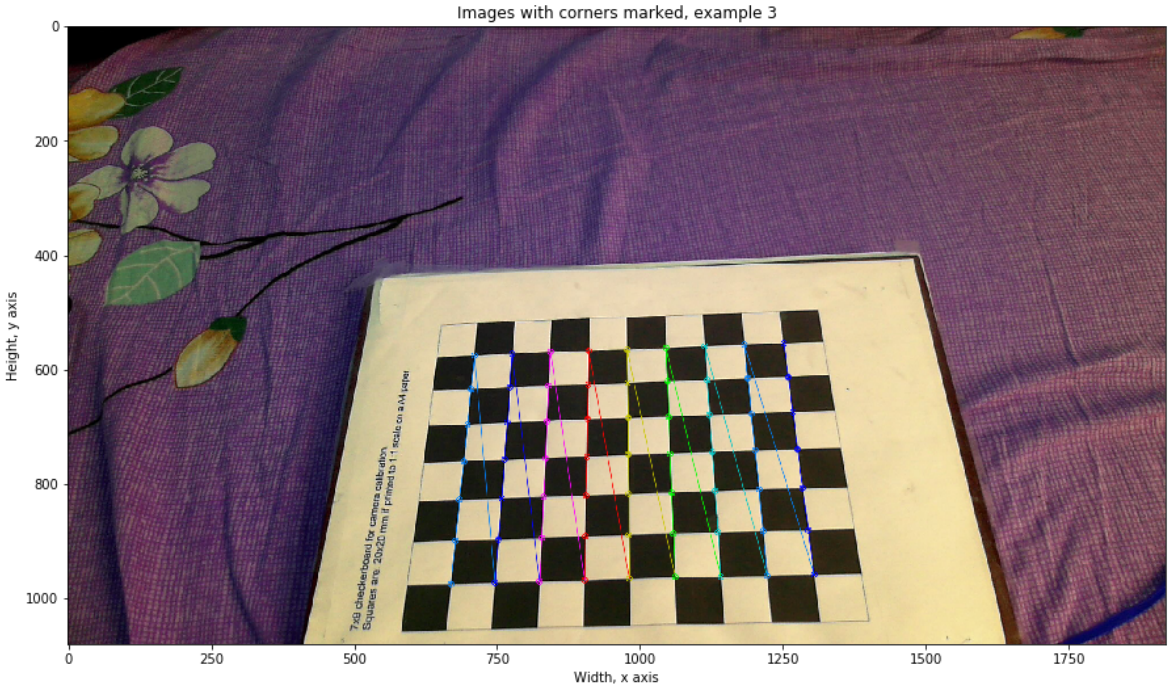
In [12]:

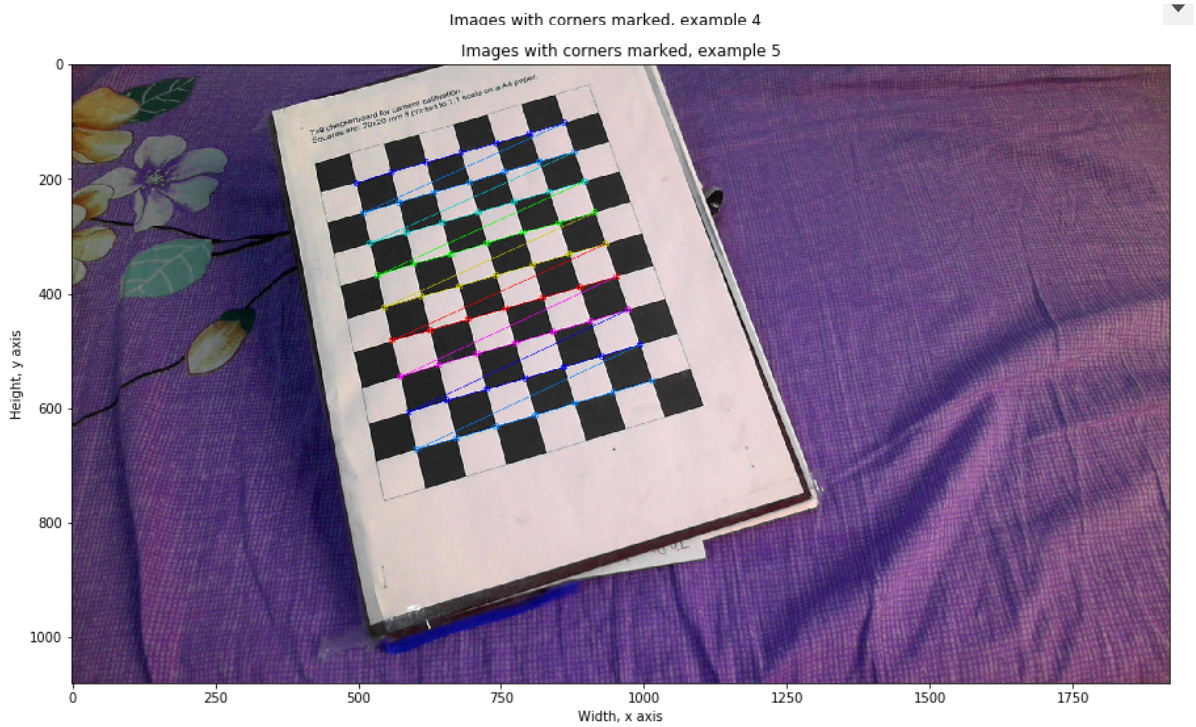
```
import os
import numpy as np
import cv2
import glob
from matplotlib import pyplot as plt
from copy import deepcopy
import rawpy

def get_points(images_path,pattern_shape):
    # termination criteria
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

    # prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
    objp = np.zeros((pattern_shape[0]*pattern_shape[1],3), np.float32)
    objp[:,:2] = np.mgrid[0:pattern_shape[0],0:pattern_shape[1]].T.reshape(-1,2

    # Arrays to store object points and image points from all the images.
    objpoints = [] # 3d point in real world space
    imgpoints = [] # 2d points in image plane.

    images = glob.glob(images_path+'*.jpg')
    count = 0
    for fname in images:
        img = cv2.imread(fname)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray, (pattern_shape[0],patter
        # If found, add object points, image points (after refining them)
        if ret == True:
            objpoints.append(objp)

            corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
            imgpoints.append(corners2)
            # Display only five images
            if count<5:
                plt.figure(figsize=(15,10))
                # Draw and display the corners
                img = cv2.drawChessboardCorners(img, (pattern_shape[0],pattern_
                plt.imshow(img)
                plt.title('Images with corners marked, example {}'.format(count
                plt.xlabel('Width, x axis')
                plt.ylabel('Height, y axis')
                plt.show()
            count+=1

    return objpoints, imgpoints, gray.shape[::-1]

def camera_calibration(objpoints,imgpoints,image_shape):
    return cv2.calibrateCamera(objpoints, imgpoints, image_shape,None,None)

def get_newmatrix(mtx,dist,image_shape):
    w = image_shape[0]
    h = image_shape[1]
    return cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))

def undistort_image(image,mtx,dist,newcameramtx,roi):
    # undistort
    dst = cv2.undistort(image, mtx, dist, None, newcameramtx)

```

```
    # crop the image
    x,y,w,h = roi
    dst = dst[y:y+h, x:x+w]
    return dst


images_path = "./webcam_2/"
pattern_shape = [7,9]
objpoints,imgpoints,image_shape = get_points(images_path,pattern_shape)
ret, mtx, dist, rvecs, tvecs = camera_calibration(objpoints,imgpoints,image_sha

# Here the focal lengths are in pixels.
newcameramtx, roi = get_newmatrix(mtx,dist,image_shape)

# Conversion from pixels to mm scale for focal length
camera_sensor_width = 5.85
camera_sensor_height = 3.27
image_width = image_shape[0]
image_height = image_shape[1]
new_camera_matix = deepcopy(mtx)
# F(mm) = F(pixel) * sensorwidth/image_width
new_camera_matix[0][0] = new_camera_matix[0][0] * camera_sensor_width /image_wi
new_camera_matix[1][1] = new_camera_matix[1][1] * camera_sensor_height /image_h


print("Camera matrix (Focal length in mm):\n",new_camera_matix)
# print("Rotation vector:\n",rvecs)
# print("Translation vector:\n",tvecs)
print("Distortion coefficients:\n",dist)
# print("New camera matrix:\n",newcameramtx)

with open("camera_parameter_file.txt",'w',encoding = 'utf-8') as f:
    f.write("Camera matrix:\n")
    f.write("(Focal length, fx, fy are in mm)\n\n")
    for item in new_camera_matix:
        f.write("%s\n" % item)
    f.write("\nDistortion matrix\n\n")
    for item in dist:
        f.write("%s\n" % item)
```




Images with corners marked, example 2
Height, y axis
Width, x axis
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


Images with corners marked, example 3
Height, y axis
Width, x axis
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750

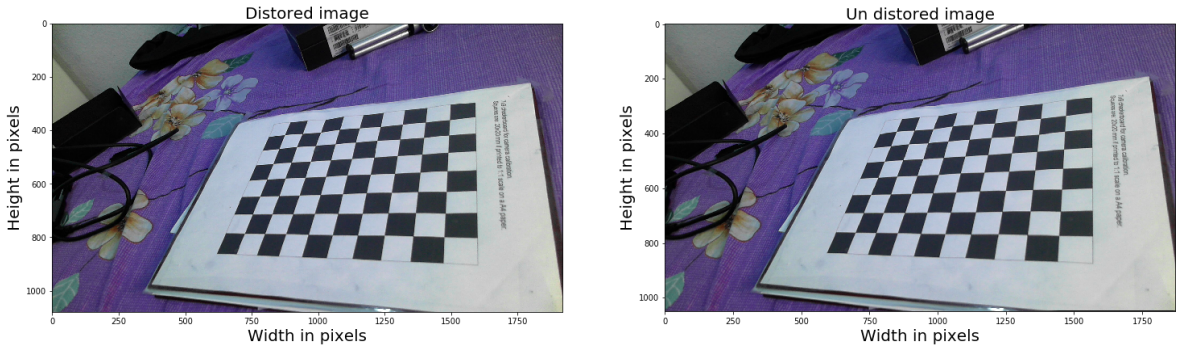



```
Camera matrix (Focal length in mm):
 [[  4.82046007   0.         964.61858884]
 [  0.           4.75060496 522.84961464]
 [  0.           0.           1.        ]]
Distortion coefficients:
 [[-0.01846476  0.12160998  0.01312348  0.00331914 -0.30123076]]
```

In [13]:

```
projection_error = 0
for i in range(0,len(objpoints)):
    imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, di
    error = cv2.norm(imgpoints[i],imgpoints2, cv2.NORM_L2)/len(imgpoints2)
    projection_error += error

print ("Reprojection error: ", projection_error/len(objpoints))
image = cv2.imread("test_images/my_photo-1.jpg")
undistorted_image = undistort_image(image,mtx,dist,mtx,roi)
fig,a =  plt.subplots(1,2,figsize=(25,20))
a[0].set_title("Distored image",fontsize=20)
a[0].set_xlabel("Width in pixels",fontsize=20)
a[0].set_ylabel("Height in pixels",fontsize=20)
a[0].imshow(image)
a[1].set_title("Un distored image",fontsize=20)
a[1].set_xlabel("Width in pixels",fontsize=20)
a[1].set_ylabel("Height in pixels",fontsize=20)
a[1].imshow(undistorted_image)
plt.show()
```

```
Reprojection error:  0.10773373208512961
```

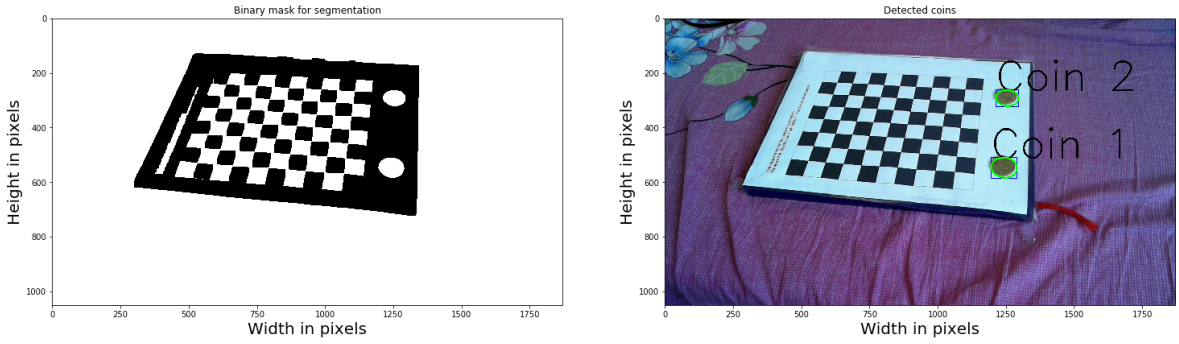


In [14]:

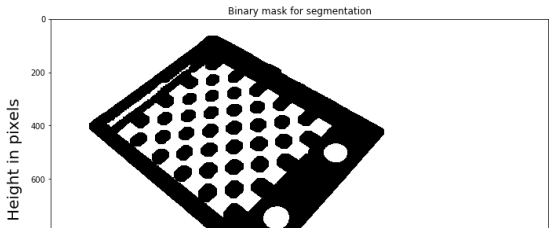
```
import imutils
folder_path = "./planarmeasure/webcam/"
img_paths = glob.glob(folder_path+"*jpg")

rvec = np.mean(rvecs,axis=0)
tvec = np.reshape(np.mean(tvecs,axis=0),(1,3))
r_mat,_ = cv2.Rodrigues(rvec)


def project_to_world_coordinated(point,r_mat,tvec,mtx):
    '''
    Function to project pixel coordinates to world coordinates
    params:
        point: 2d point (u,v,1)
        r_mat: rotation matrix
        tvec: translation vector
        mtx: camera matrix
    return:
        world_point: 3d point (x,y,z)

    '''
    point_wrt_center = np.array([point[0],point[1],1])
    point_wrt_center = np.reshape(point_wrt_center,(1,3))
    point_camera_coordinates = np.dot(point_wrt_center,np.linalg.pinv(mtx))
    world_point = np.dot((point_camera_coordinates- tvec),np.linalg.pinv(r_mat)
    return world_point
def distance_between_points(point1,point2):
    '''
    Function to calculate eculidean distance
    params:
        point1: world cordinates
        point2: world coordinates
    return:
        distance: distance between given points
    '''
    return np.linalg.norm(point1[:,:2]-point2[:,:2])
coin_diameters = []

# Looping over all the test images
for f in img_paths:
    #Read the image
    img = cv2.imread(f)
    # Undistort the image with calculated calbration parameters
    img = undistort_image(img,mtx,dist,newcameramtx,roi)
    pixx,pixy,_= img.shape
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Get binary mask fo image
    thresh = cv2.threshold(gray, 0, 255,cv2.THRESH_BINARY_INV| cv2.THRESH_OTSU)
    # Morphological operations to remove noise and get appropriate blobs
    kernel = np.ones((4, 4), np.uint8)
    closing = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel, iterations=7)
    kernel = np.ones((3, 3), np.uint8)
    opening = cv2.morphologyEx(closing, cv2.MORPH_OPEN, kernel, iterations=2)
    cont_img = opening.copy()

    # Find contours on fine tuned mask
    cnts = cv2.findContours(cont_img, cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(cnts)
#     loop over the contours
```

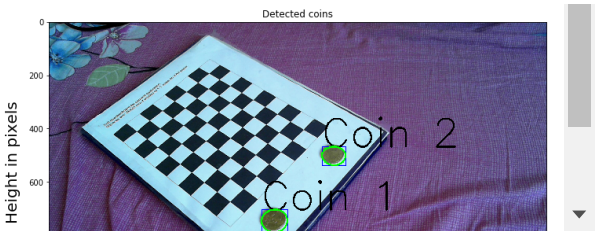
```
    coin_count = 0
    for (i, c) in enumerate(cnts):

        #Get number of sides of contour and area
        approx = cv2.approxPolyDP(c,0.01*cv2.arcLength(c,True),True)
        area = cv2.contourArea(c)

        # Filtering large blobs and small blobs
        if len(approx) in range(10,15) and (pixx*pixy/200) > area > (pixx*pixy/
            # draw the contour
            x, y, w, h = cv2.boundingRect(c)
            cv2.rectangle(img, (x, y), (x + w, y + h), (255, 0, 0), 2)
            # Getting world coordinates
            point1 = project_to_world_coordinated([x,y],r_mat,tvec,mtx)
            point2 = project_to_world_coordinated([x+w,y],r_mat,tvec,mtx)
            coin_count+=1

            # Get distance betw two given points
            diameter = np.round(distance_between_points(point1,point2),2)
            coin_diameters.append(diameter)
            print("Diameter of coin ",coin_count," is:",diameter)
            cv2.putText(img,"Coin "+str(coin_count), (x,y), cv2.FONT_HERSHEY_SI
            cv2.drawContours(img, [c], -1, (0, 255, 0), 5)
    fig,a =  plt.subplots(1,2,figsize=(25,20))
    a[0].imshow(cv2.cvtColor(closing,cv2.COLOR_BGR2RGB))
    a[0].set_title("Binary mask for segmentation")
    a[0].set_xlabel("Width in pixels",fontsize=20)
    a[0].set_ylabel("Height in pixels",fontsize=20)
    a[1].imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
    a[1].set_title("Detected coins")
    a[1].set_xlabel("Width in pixels",fontsize=20)
    a[1].set_ylabel("Height in pixels",fontsize=20)
    plt.show()
gt_diameter = 24.25 #in mm
mean_diameter = np.mean(coin_diameters)
error_in_diamter = gt_diameter - mean_diameter
print("Ground diameter of coin is: ",gt_diameter," mm")
print("Average diameter of coins is: ",np.round(mean_diameter,2)," mm")
print("Error in diameter of coins is: ",np.round(error_in_diamter,2)," mm")

```

```
Diameter of coin  1  is: 23.24
Diameter of coin  2  is: 20.3
```

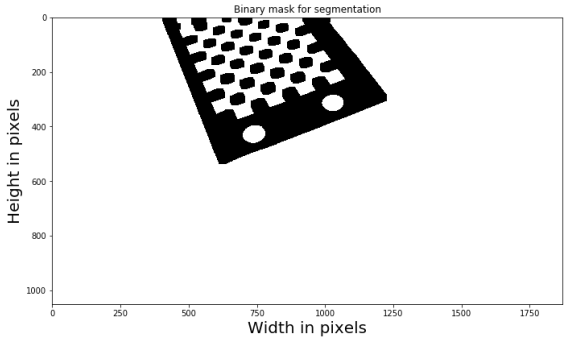



```
Diameter of coin  1  is: 23.73
Diameter of coin  2  is: 21.53
```






```
Diameter of coin  1  is: 20.3
Diameter of coin  2  is: 19.32
```




```
Ground diameter of coin is:  24.25  mm
Average diameter of coins is:  21.4  mm
Error in diameter of coins is:  2.85  mm
```

- Ground truth diamter of the coin used is 24.25mm and taken from https://en.wikipedia.org/wiki/50_euro_cent_coin (https://en.wikipedia.org/wiki/50_euro_cent_coin).
- Average diameter of the coins used in above experiments is 21.40mm.
- Error in measurement of diameter is 2.85mm.

Task2: Compare the MATLAB calibration process and the respective openCV calibration using Python, spell out in detail :

- Similarities / differences.

- Will both tools produce IDENTICAL results for IDENTICAL cameras?

- What makes the difference?

In [6]:

```
# YOUR CODE HERE
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('./webcam_2/*.jpg')

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (7,9),None)

    if ret == True:
        objpoints.append(objp)
        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)
        # Draw and display the corners
        img = cv2.drawChessboardCorners(img, (7,9), corners2,ret)
#         plt.imshow(img)

# cv2.destroyAllWindows()
imgpoints = np.array(imgpoints,'float32')



pattern_size = (7, 9)
pattern_points = np.zeros( (np.prod(pattern_size), 3), np.float32)
pattern_points[:, :2] = np.indices(pattern_size).T.reshape(-1, 2).astype(np.flo
pattern_points = np.array(pattern_points,dtype=np.float32)

ret, matrix, dist_coef, rvecs, tvecs = cv2.calibrateCamera([pattern_points], [c

print(matrix)
print(dist_coef)
```

```
[[1.63336621e+03 0.00000000e+00 1.08854095e+03]
 [0.00000000e+00 1.76382081e+03 6.83461309e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
[[ 0.37108656 -1.21584908  0.02733278  0.00880263  1.63530882]]
```

In [7]:

```
# YOUR CODE HERE
# The camera calibration is done using Caltech toolbox and the results are as f
# Focal Length:          fc = [ 1587.20251   1572.95907 ] +/- [ 32.41292   28.5
# Principal point:       cc = [ 972.03474   516.89158 ] +/- [ 9.88424   25.4343
# Skew:             alpha_c = [ 0.00000 ] +/- [ 0.00000  ]   => angle of pixel
# Distortion:            kc = [ 0.00634   -0.04359   0.01339   0.00525  0.00000
# Pixel error:          err = [ 0.40559   0.77478 ]
```

Both process uses the same algorithm presented by Zhang here (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr98-71.pdf).

But, The main differences are seen in

1) Corner detection method :

. Matlab uses the pre-entered corner and the squares side length inorder to extract the next corner, This needs manually marking the four corners of the checkerboard
. OpenCV uses color gradient changes to extract the corners.

2) Optimization techniques :

. MATLAB uses the Levenberg-Marquardt non-linear least squares algorithm for the optimization
. OpenCV performs optimization based on Maximum Likelihood criterion of gradients.

From the results above they produced minimally different results for an identical camera

Task3: Read the slide set read the subsection on "Computation of F" (OR check the book Chapter 11). Implement a Normalized 8 point algorithm.

Use the following signature:

`fNorm8Point = estimateFundamentalMatrix(inlierPts1, inlierPts2)`

where the inliers denote the pixel coordinates of corresponding points in left / right image.

Image material can be found under images, and also some point arrays
(`FMatrixCorrPointsStereo.txt`) for corresponding points in left and right image of image number 7
(`images/7.picture-24.jpg`)

In [8]:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import typing
from copy import deepcopy
import re
import matplotlib.patches as mpatches
import time

def correspondence_parser(points_file: str) -> typing.Tuple[np.ndarray]:
        '''
        Reading inputs
        :param file_to_parse: File name containing corresponding points in lef
        :return final_image_point: Tuple of lists containing points of type fl
        '''
        image_points = list()
        final_image_point = list()

        f = open(points_file)

        lines = f.readlines()
        image_points.append(list(lines[4:65]))
        image_points.append(list(lines[70:131]))

        for points in image_points:
            float_point = list()
            for point in points:
                point_string = re.findall(r'[0-9\.]+', point)
                point_ = list()
                for p in point_string:
                    point_.append(float(p))
                float_point.append(point_)
            final_image_point.append(float_point)

        image1_points = np.array(final_image_point[0]).T
        image2_points = np.array(final_image_point[1]).T

        return image1_points, image2_points


class Compute_Fundamental_matrix():

    def __init__(self):
        np.set_printoptions(suppress=True)


    def normalize(self, x: np.ndarray) -> np.ndarray:

        '''
        Normalizes a given matrix, x by performing similarity transformation.
        Now the centroid of the new points has (0,0)^T and the average distance
        from the origin is square.root(2)
        :param x: Points in an image to be normalized.
        :return x_bar: Normalized points in the image.
                T: Similarity transformation matrix.
        '''

        x_bar = np.asarray(x)
        m = np.mean(x,1)
```

```
60
61          deno = list()
62
63          for i in x.transpose():
64              deno.append(list(i-m))
65          deno = np.square(np.array(deno))
66          total = 0
67          for i in deno:
68              total = sum(i)**(1/2.) + total
69
70          s = 2**(1/2)/(total/len(deno))
71
72          T = np.array([[s, 0, -s*m[0]], [0, s, -s*m[1]], [0, 0, 1]])
73
74          x = np.vstack((x, np.ones((1,x.shape[1]))))
75
76          x_bar = np.dot(T, x)
77
78          return x_bar, T
79
80      def Fundamental_matrix(self, x, x_bar, to_P2 = False):
81
82          '''Computing Fundamental matrix F
83          :param x: left image points.
84                 x_prime: right image points corresponding with x points.
85                 to_P2: Whether point should be converted to P2.
86          : return F: Fundamental matrix mapping x points to epiline in right ima
87          '''
88
89          if to_P2:
90              x = np.vstack((x, np.ones((1, x.shape[1]))))
91              x_bar = np.vstack((x_bar, np.ones((1, x_bar.shape[1]))))
92
93          # Initializations for A matrix
94          A = list()
95
96          # Forming A matrix as provided in Hartley and zisserman multiple view g
97
98          # x1'x1, x1'y1, x1', y1'x1, y1'y1, y1', x1, y1, 1
99
100         for i,j in zip(x.transpose(), x_bar.transpose()):
101             row = [j[0]*i[0], j[0]*i[1], j[0], j[1]*i[0], j[1]*i[1], j[1], i[0]
102             A.append(row)
103
104         A = np.array(A)
105
106         # Ah = 0 ;Minimizing ||Ah|| such that ||h||=1
107         # Solution by SVD
108         U,s,Vt = np.linalg.svd(A)
109         V=Vt.T
110         F_cap=V[:,-1] # Last column of V corresponding eigen vector to the sma
111
112         F_cap = np.reshape(F_cap,(3,3))
113
114         return F_cap
115
116     def enforce_singularity(self, F_cap, r_to_l):
117
118         '''
119         Ensure det(F) = 0 because F is a rank 2 matrix.
120         :param F_cap: F to be made singular
```

```
                r_to_l: whether the mapping is from right to left image.
        :return F_cap_prime: singular F.
        '''

        U, s, Vt = np.linalg.svd(F_cap)
        s[2] = 0
        s = np.diag(s)
        if r_to_l:
            F_cap_prime = np.dot(U,np.dot(s, Vt)).T
        else:
            F_cap_prime = np.dot(U,np.dot(s, Vt))

        return F_cap_prime


    def desired_F(self, T_bar, T_prime_bar, F_normalized):

        '''
        Used for finding F in unnormalized original co-ordinates from normalize
        F = transpose(T') F_normalized T.
        '''
        desired_F = np.dot((T_prime_bar.T),(np.dot(F_normalized, T_bar)))

        desired_F = desired_F/desired_F[-1][-1]

        return desired_F


def estimateFundamentalMatrix(x, x_prime):

    '''
    Computes fundamental matrix, F.
    :param x: left image points.
           x_prime: right image points corresponding with x points.
    : return F: Fundamental matrix mapping x points to epiline in right image.
    '''

    F = Compute_Fundamental_matrix()

    # Normalize x to overcome the numerical calculation issues.
    x_bar, T_bar = F.normalize(x)

    # Normalize x' to overcome the numerical calculation issues.
    x_prime_bar, T_prime_bar = F.normalize(x_prime)

    # Compute Fundamental matrix.
    F_cap = F.Fundamental_matrix(x_bar, x_prime_bar)

    # Enfore singularity constraint det(F_cap) = 0.
    F_cap_prime = F.enforce_singularity(F_cap, False) # Set to l_to_r always.

    # Desired F unnormalized by using the similartity transformations done to
    F = F.desired_F(T_bar, T_prime_bar, F_cap_prime)

    return F


corresponding_points = '../FMatriximages/FMatriximages/7/FMatrixCorrPointsStere
# Get initial corresponding points.
inlierPts1, inlierPts2 = correspondence_parser(corresponding_points)
print('\n********** OUR IMPLEMENTATION**********\n')
```

```
start = time.time()
fNorm8Point = estimateFundamentalMatrix(inlierPts1, inlierPts2)
end = time.time()
F = fNorm8Point
print("Fundamental matrix computed by our implementation, F:\n",F)
print("F is computed by our implementation in {} seconds.". format(round(end-st
corresponding_points = '../FMatriximages/FMatriximages/7/FMatrixCorrPointsStere

print('\n********** OPENCV IMPLEMENTATION**********\n')
start = time.time()
F_cv, mask = cv2.findFundamentalMat(inlierPts1.T, inlierPts2.T, cv2.FM_8POINT)
end = time.time()
print("Fundamental matrix computed by OpenCV implementation, F_cv:\n",F_cv)
print("F is computed by OpenCV implementation in {} seconds.". format(round(end
```

```
********** OUR IMPLEMENTATION**********

Fundamental matrix computed by our implementation, F:
 [[ 0.00000003 -0.00008342  0.03308724]
 [ 0.00008593  0.000009    0.90938887]
 [-0.03374484 -0.91580868  1.        ]]
F is computed by our implementation in 0.00623 seconds.

********** OPENCV IMPLEMENTATION**********

Fundamental matrix computed by OpenCV implementation, F_cv:
 [[ 0.00000003 -0.00008342  0.03308732]
 [ 0.00008593  0.000009    0.90937261]
 [-0.03374492 -0.91579242  1.        ]]
F is computed by OpenCV implementation in 0.00195 seconds.
```

In [9]:

```
def test_case1(inlierPts1, inlierPts2):
    '''
    Checking if the Fundamental matrix satisfy the transpose property.
    If F is the fundamental matrix of the pair of cameras (P, P'), then
    F^T is the fundamental matrix of the pair in the opposite order: (P', P).

    Instead of using the camera matrix,
    the direction of mapping points has been used to check the property.
    :param inlierPts1: Points in left image.
           inlierPts2: Points in right image corresponding to left image points
    '''

    F_L2R = estimateFundamentalMatrix(inlierPts1, inlierPts2)
    F_R2L = estimateFundamentalMatrix(inlierPts2, inlierPts1)
    difference = F_L2R - F_R2L.T
    assert np.all(difference < 0.05),'Test case 1 failed'
    print('Test case 1 passed')
def test_case2(inlierPts1, inlierPts2):
    '''
    Checking with the benchmark OpenCV Fundamental matrix.
    :param inlierPts1: Points in left image.
           inlierPts2: Points in right image corresponding to left image points
    '''

    F = estimateFundamentalMatrix(inlierPts1, inlierPts2)
    F_cv, _ = cv2.findFundamentalMat(inlierPts1.T, inlierPts2.T, cv2.FM_8POINT)
    difference = F - F_cv
    assert np.all(difference < 0.05),'Test case 2 failed'
    print('Test case 2 passed')

def test_case3(inlierPts1, inlierPts2):
    '''
    Checking x'^T F x = 0 for any random corresponding point.
    :param inlierPts1: Points in left image.
           inlierPts2: Points in right image corresponding to left image points
    '''

    F = estimateFundamentalMatrix(inlierPts1, inlierPts2)
    point = np.random.randint(0,inlierPts1.shape[1])
    x_prime = np.array([[inlierPts2[0][point]], [inlierPts2[1][point]], [1]])
    x = np.array([[inlierPts1[0][point]], [inlierPts1[1][point]], [1]])
    value = x_prime.T @ F @ x
    assert value[0][0]<0.05, 'Test case 3 failed'
    print('Test case 3 passed')

test_case1(inlierPts1, inlierPts2)
test_case2(inlierPts1, inlierPts2)
test_case3(inlierPts1, inlierPts2)
```

```
Test case 1 passed
Test case 2 passed
Test case 3 passed
```

The first cell contains the implementation of computing F. The second cell contains three test cases to verify the implementation of F by testing the F matrix computed. The test case tries to evaluate whether F satisfy the properties of F.

Task4: Draw the respective epipoles on top of the images based the matrix F and corresponding points you found in task4

In [10]:

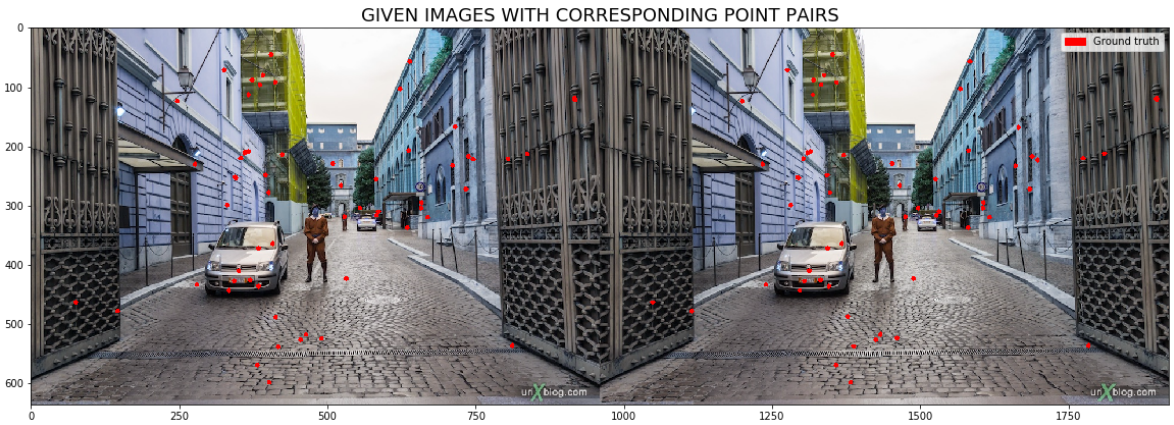
```
def find_epipole(F) -> typing.Tuple[np.ndarray]:
    '''Compute left and right null vector for finding the epipoles
    :param F: Fundamental matrix

    :return epipole_left: Left camera epipole
            epipole_right: Right camera epipole
    '''

    U, s, Vt = np.linalg.svd(F)
    V = Vt.T
    epipole_left = V[:,-1]
    epipole_left = epipole_left/epipole_left[2]


    U, s, Vt = np.linalg.svd(F.T)
    V = Vt.T
    epipole_right = V[:,-1]
    epipole_right = epipole_right/epipole_right[2]

    return epipole_left, epipole_right

epipole = find_epipole(F)
print('Epipole for F computed by our implementation:\nLeft:\n{}\nRight:\n{}'.fo
epipole_cv = find_epipole(F_cv)
print('Epipole for F computed by OpenCV implementation:\nLeft:\n{}\nRight:\n{}'


```

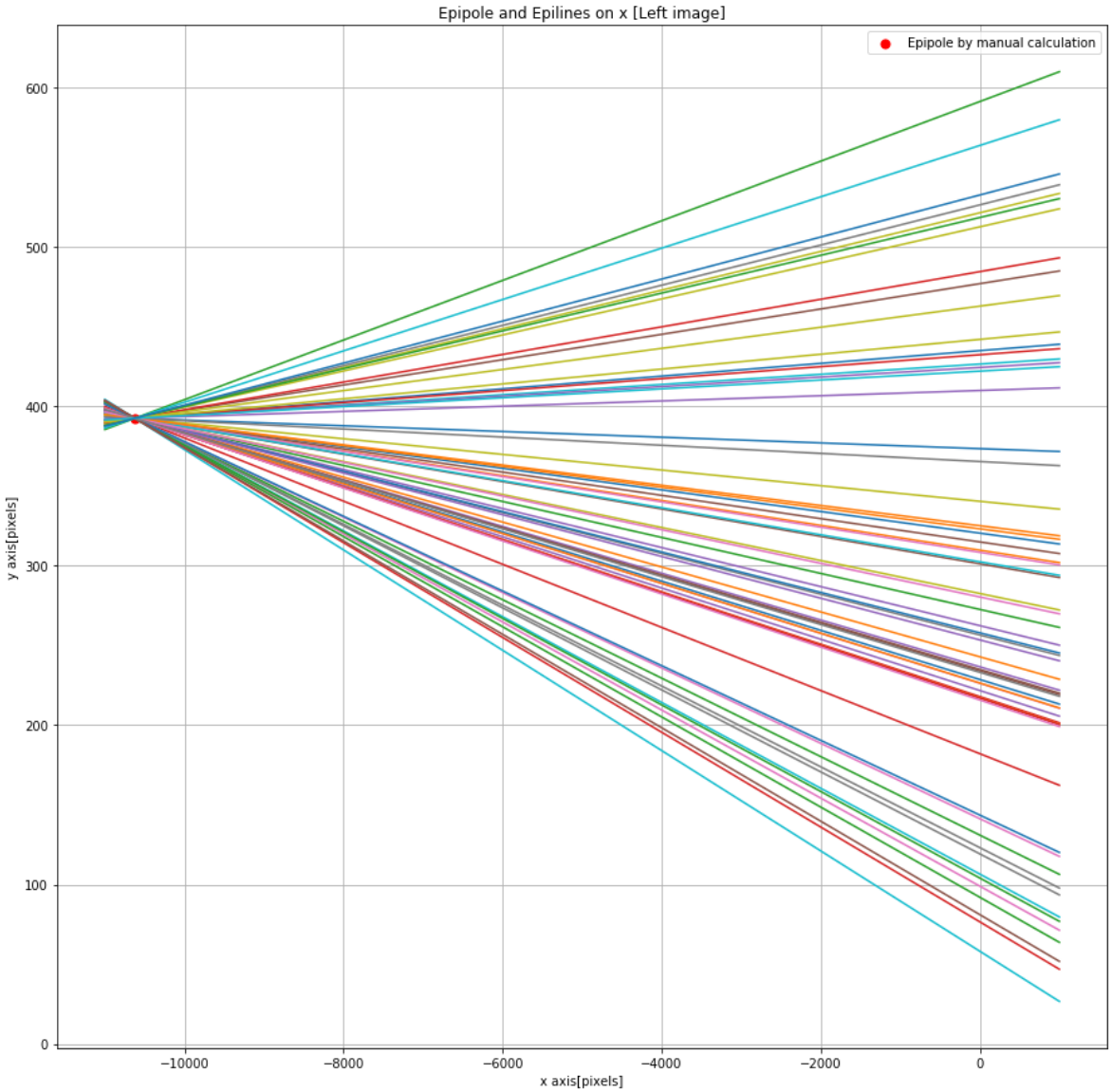
```
Epipole for F computed by our implementation:
Left:
[-10623.6595891     392.54229703      1.        ]
Right:
[-10935.60035238    396.7816619       1.        ]
Epipole for F computed by OpenCV implementation:
Left:
[-10623.33565852    392.5383261       1.        ]
Right:
[-10935.26518849    396.77754247      1.        ]
```

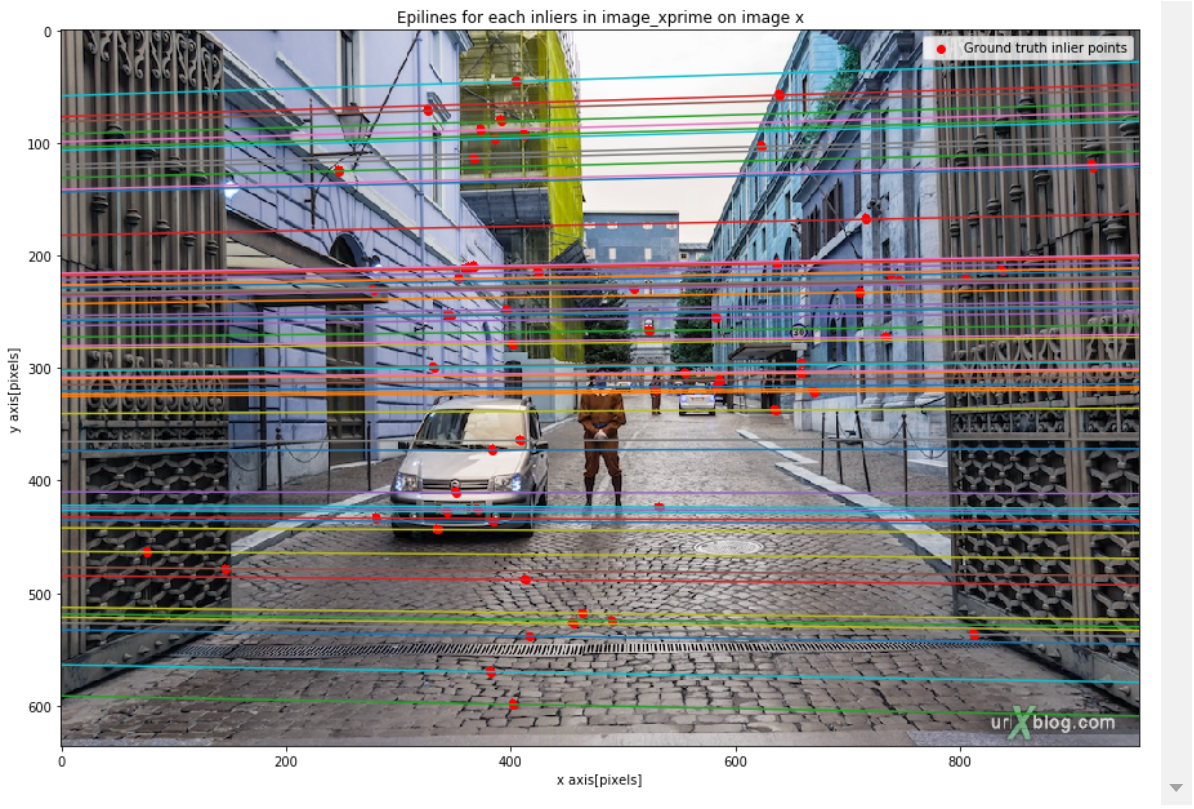
In [11]:

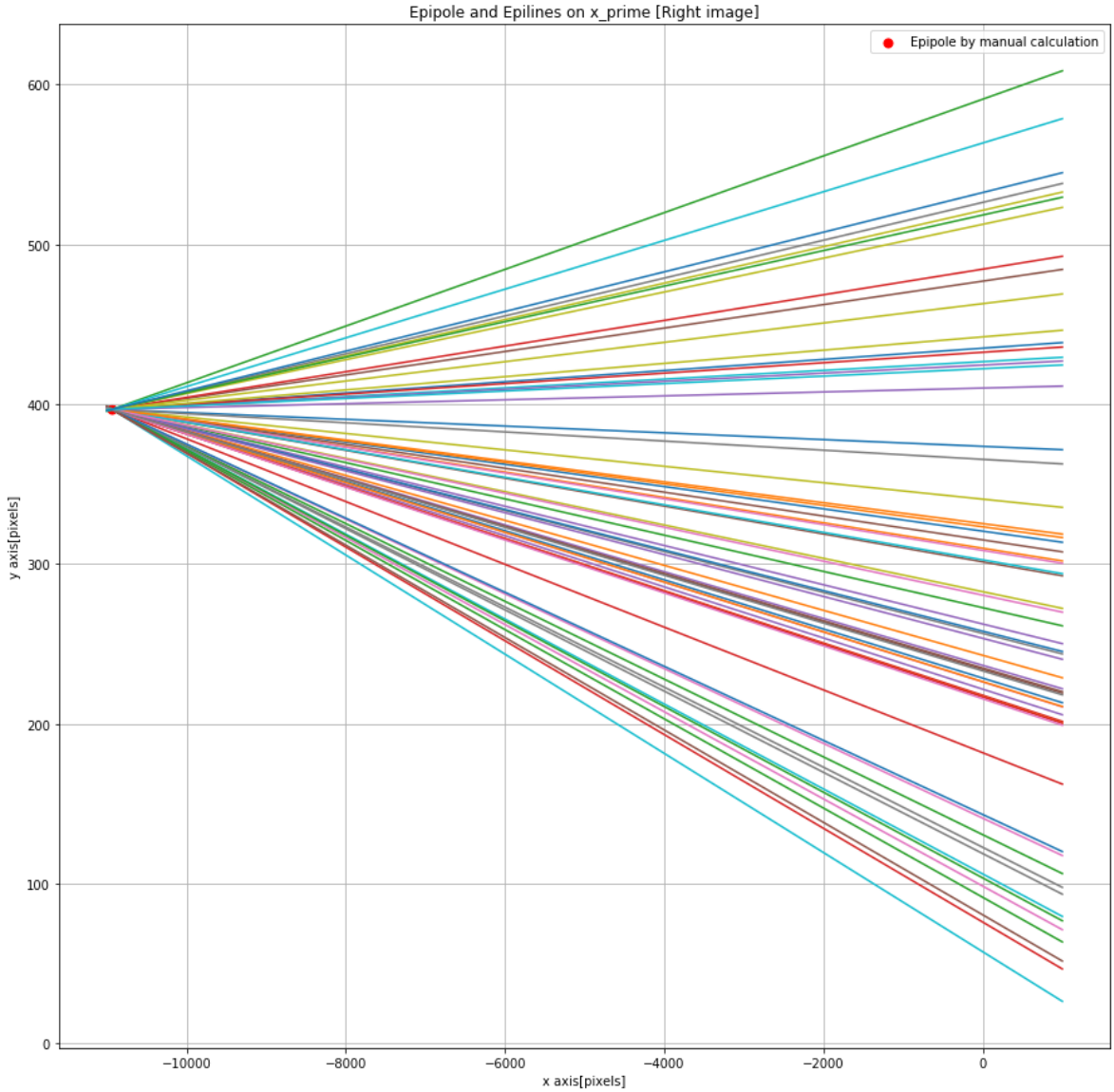
```
def draw_epilines(F, img1, img2, inlierPts1, inlierPts2):
    '''
    Draws epilines on another image for the points given in an image using F gi
    Images are plotted for illustration purposes.
    Epipoles and the corresponding epilines are plotted in the graph.
    Epilines are plotted in the image directly.

    :param F: Fundamental matrix mapping points in left image to lines in right
           img1: Left image (x image)
           img2: Right image (x_prime image)
           inlierPts1: Points in left image.
           inlierPts2: Points in right image corresponding to left image points
    '''

    x = np.vstack((inlierPts1, np.ones((1, inlierPts1.shape[1]))))
    x_prime = np.vstack((inlierPts2, np.ones((1, inlierPts2.shape[1]))))

    epiline_on_xprime = F@x
    epiline_on_x = F.T@x_prime

    span = np.linspace(-11000, 1000 , 61)
    span_img1 = range(img1.shape[1])
    span_img2 = range(img2.shape[1])

    plt.figure(figsize=(15,15))
    for n,i in enumerate(range(x.shape[1])):
        y = (-epiline_on_x[2][n] -epiline_on_x[0][n]*span)/ (epiline_on_x[1][n]
        plt.plot(span, y)
        plt.title('Epipole and Epilines on x [Left image]')
        plt.grid()
    plt.scatter(epipole[0][0], epipole[0][1], c ='red', s=50, label='Epipole by
    plt.legend()
    plt.xlabel('x axis[pixels]')
    plt.ylabel('y axis[pixels]')
    plt.show()

    plt.figure(figsize=(15,15))
    for n,i in enumerate(range(x.shape[1])):
        y = (-epiline_on_x[2][n] -epiline_on_x[0][n]*span_img1)/ (epiline_on_x[
        plt.plot(span_img1, y)
    plt.scatter(inlierPts1[0],inlierPts1[1], c='r', label='Ground truth inlier
    plt.imshow(img1)
    plt.legend()
    plt.title('Epilines for each inliers in image_xprime on image x')
    plt.xlabel('x axis[pixels]')
    plt.ylabel('y axis[pixels]')
    plt.show()

    plt.figure(figsize=(15,15))
    for n,i in enumerate(range(x_prime.shape[1])):
        y = (-epiline_on_xprime[2][n] -epiline_on_xprime[0][n]*span)/ (epiline_
        plt.plot(span, y)
        plt.title('Epipole and Epilines on x_prime [Right image]')
        plt.grid()
    plt.scatter(epipole[1][0], epipole[1][1], c ='red', s=50, label='Epipole by
    plt.legend()
    plt.xlabel('x axis[pixels]')
    plt.ylabel('y axis[pixels]')
    plt.show()
```

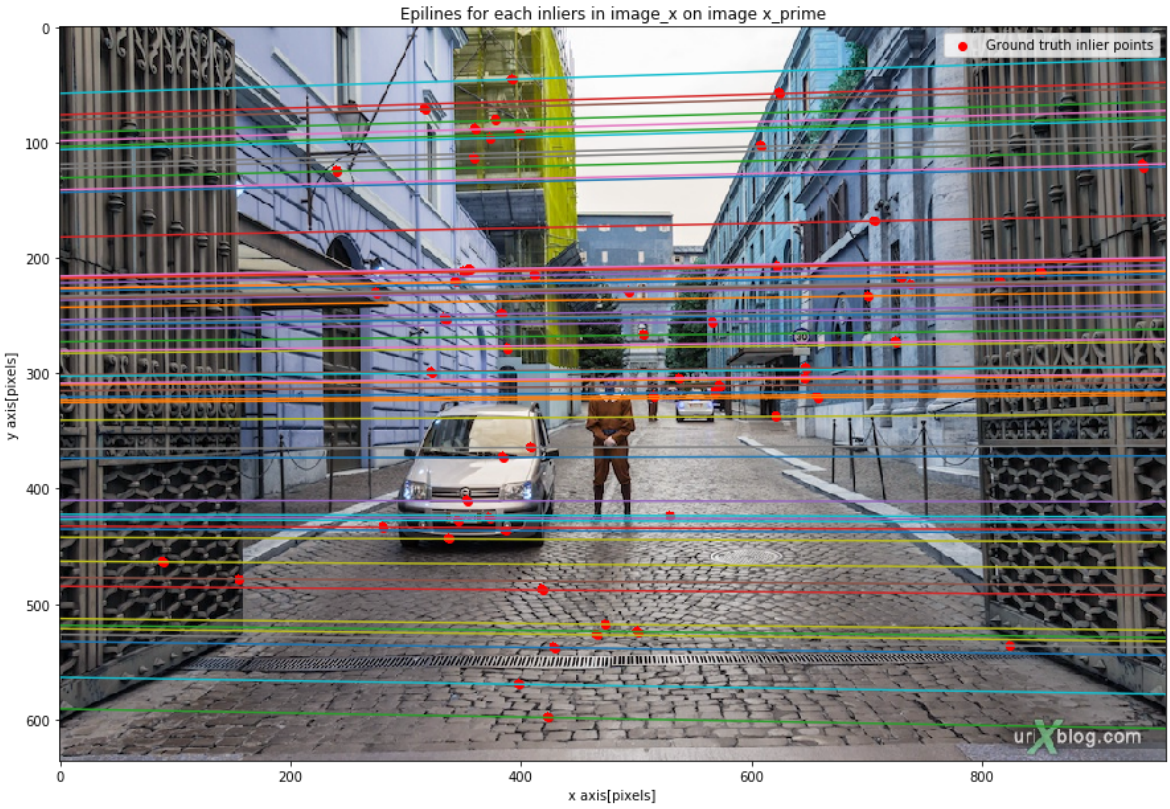
```
60
61      plt.figure(figsize=(15,15))
62      for n,i in enumerate(range(x_prime.shape[1])):
63          y = (-epiline_on_xprime[2][n] -epiline_on_xprime[0][n]*span_img2)/ (epi
64          plt.plot(span_img2, y)
65      plt.scatter(inlierPts2[0],inlierPts2[1], c='r', label='Ground truth inlier
66      plt.imshow(img2)
67      plt.legend()
68      plt.title('Epilines for each inliers in image_x on image x_prime')
69      plt.xlabel('x axis[pixels]')
70      plt.ylabel('y axis[pixels]')
71      plt.show()
72
73
74  corresponding_points = '../FMatriximages/FMatriximages/7/FMatrixCorrPointsStere
75  # Get initial corresponding points.
76  inlierPts1, inlierPts2 = correspondence_parser(corresponding_points)
77  fNorm8Point = estimateFundamentalMatrix(inlierPts1, inlierPts2)
78
79  img = cv2.imread('../FMatriximages/FMatriximages/7/picture-24.jpg')
80  img1 = img[:,:img.shape[1]//2]
81  img2 = img[:,img.shape[1]//2:]
82  # Displaying corresponding points
83  for i in zip(inlierPts1[0], inlierPts1[1]):
84      cv2.circle(img,(int(i[0]), int(i[1])),1,(255,0,0), 5)
85
86  for i in zip(inlierPts2[0], inlierPts2[1]):
87      cv2.circle(img,(int(i[0]) + 960, int(i[1])),1,(255,0,0), 5)
88  plt.figure(figsize=(20,20))
89  plt.imshow(img)
90  plt.title('GIVEN IMAGES WITH CORRESPONDING POINT PAIRS', fontsize=18)
91  red_patch = mpatches.Patch(color='red', label='Ground truth')
92  plt.legend(handles=[red_patch])
93  plt.show()
94
95  draw_epilines(fNorm8Point, img1, img2, inlierPts1, inlierPts2)
```




Epipole and Epilines on x [Left image]
Epipole by manual calculation
600
500
400
300
200
100
0
y axis[pixels]
−10000
−8000
−6000
−4000
−2000
0
x axis[pixels]


Epilines for each inliers in image_xprime on image x
Ground truth inlier points
y axis[pixels]
x axis[pixels]
0
100
200
300
400
500
600
0
200
400
600
800
urXblog.com


Epipole and Epilines on x_prime [Right image]
Epipole by manual calculation
y axis[pixels]
x axis[pixels]
0
100
200
300
400
500
600
−10000
−8000
−6000
−4000
−2000
0



The epilines in image x (left image) and image x_prime (right image) are obtained from the points in the other image using the fundamental matrix computed by our implementation. The epipoles for the left camera and right camera are computed using SVD decomposition of F and F^T, respectively. The two plots (epipole and epilines graphs) above illustrate the eplines on a image plane is extended to meet at an epipole for both images. The

epipoles match the manually computed epipoles (using SVD) for both left and right image. The epilines intersect a particular point on an image which depicts that this point is the corresponding point for the point whose epiline intersects the point.