Assignment 3 : Calibration

Please edit the cell below to include your name and student ID #

name: Joseph Pariser

SID: 51698290

NOTE: In this assignment we will build on top of your **project** and **triangulate** functions from previous assignments. If you weren't happy with your implementation from assignment 2, please consult with the professor or your classmates to get things cleaned up before starting this assignment.

1. Parameterizing 3D Rotations

In order to optimize over the camera rotation during calibration, we need a way to parameterize the space of 3D rotations. There are many different ways to do this and each comes with different tradeoffs, but for our purposes we will adopt a simple approach of building a rotation by a sequence of rotations around the X, Y and Z axes (so called *Tait-Bryan angles*, see https://en.wikipedia.org/wiki/Euler_angles (https://en.wikipedia.org/wiki/Euler_angles) for more discussion)

1.1 Implement [15pts]

Write a function **makerotation** which takes as input three angles **rx,ry,rz** and returns a rotation matrix corresponding to rotating by **rx** degrees around the x-axis, followed by a rotation of **ry** degrees around the y-axis, followed by a rotation of **rz** degrees around the z-axis.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
import matplotlib.patches as patches
from mpl_toolkits.mplot3d import Axes3D
import visutils
%matplotlib inline
```

In [2]:
```python
def makerotation(rx,ry,rz):
    """
    Generate a rotation matrix

    Parameters
    ----------
    rx,ry,rz : floats
        Amount to rotate around x, y and z axes in degrees

    Returns
    -------
    R : 2D numpy.array (dtype=float)
        Rotation matrix of shape (3,3)
    """

    rx = np.deg2rad(rx)
    ry = np.deg2rad(ry)
    rz = np.deg2rad(rz)

    cos = np.cos(rx)
    sin = np.sin(rx)
    Rx = np.array([[1,   0,    0],
                   [0, cos, -sin],
                   [0, sin,  cos]])

    cos = np.cos(ry)
    sin = np.sin(ry)
    Ry = np.array([[cos, 0, sin],
                   [0,   1,    0],
                   [-sin, 0,  cos]])

    cos = np.cos(rz)
    sin = np.sin(rz)
    Rz = np.array([[cos, -sin, 0],
                   [sin,  cos, 0],
                   [0,   0,    1]])

    R = Rz@Ry@Rx

    return R
```

1.2 Testing [5pts]

Work out by hand what a 90 degree rotation should look like around each axis. Then execute the test examples below and add some tests (asserts) to make sure your code passes.

Find a way to achieve the same rotation as **makerotation(90,90,0)** but without using rotation around the x-axis. That is, determine some angles so that **makerotation(0,?,?) == makerotation(90,90,0)**

In [3]:
```python
#
# test your function on some simple examples
#
np.set_printoptions(precision=4,suppress=True)

print(makerotation(90,0,0), '\n')

print(makerotation(0,90,0), '\n')

print(makerotation(0,0,90), '\n')

print(makerotation(90,90,0), '\n')

ry = 90
rz = -90
print(makerotation(0,ry,rz))

# figure out what ry,rz values are needed in order to pass this test
assert((makerotation(90,90,0)-makerotation(0,ry,rz)<1e-9).all())
```

```
[[ 1.  0.  0.]
 [ 0.  0. -1.]
 [ 0.  1.  0.]]

[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [-1.  0.  0.]]

[[ 0. -1.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]]

[[ 0.  1.  0.]
 [ 0.  0. -1.]
 [-1.  0.  0.]]

[[ 0.  1.  0.]
 [-0.  0. -1.]
 [-1.  0.  0.]]
```

2. Reprojection Error [20pts]

We will now specify a function which computes the reprojection error. This is the function that we will later optimize when calibrating the camera extrinsic parameters. Take a look at the documentation for **scipy.optimize.leastsq**. The optimizer expects that our function should take a vector of parameters and return a vector of residuals which it will square and sum up to get the total error. For this reason, we will structure our code in the following way.

First, write a member function for the Camera class called **update_extrinsics** which takes a vector of 6 parameters (rx,ry,rz,tx,ty,tz). The function should keep the same intrinsic parameters (f,c) but update the extrinsic parameters (R,t) based on the entries in the parameter vector.

Second, implement a function named **residuals** which computes the difference between a provided set of 2D point coordinates and the projection of 3D point coordinates by specified camera. The residuals function takes as input the 3D points, the target 2D points, a camera with specified intrinsic parameters, and an extrinsic parameter vector. You should use **update_extrinsics** to update the extrinsic parameters, compute the projection of the 3D points with the updated camera and return a 1D vector containing the differences of all the x and y coordinates.

In [4]:
```
class Camera:
    """
    A simple data structure describing camera parameters

    The parameters describing the camera
    cam.f : float   --- camera focal length (in units of pixels)
    cam.c : 2x1 vector  --- offset of principle point
    cam.R : 3x3 matrix --- camera rotation
    cam.t : 3x1 vector --- camera translation

    """

    def __init__(self,f,c,R,t):
        self.f = f
        self.c = c
        self.R = R
        self.t = t

    def __str__(self):
        return f'Camera : \n f={self.f} \n c={self.c.T} \n R={self.R}

    def project(self,pts3):
        """
        Project the given 3D points in world coordinates into the spec

        Parameters
        ----------
        pts3 : 2D numpy.array (dtype=float)
```

```
        Coordinates of N points stored in a array of shape (3,N)

    Returns
    --------
    pts2 : 2D numpy.array (dtype=float)
        Image coordinates of N points stored in an array of shape

    """
    assert(pts3.shape[0]==3)

    # p cam = R-1(PWorld - t) = R-1*PWorld - R-1*t
    f, cx, cy, R, t = self.f, self.c[0][0], self.c[1][0], self.R,

    K = np.array([[f, 0, cx],
                  [0, f, cy],
                  [0, 0,  1]])
    R_3x4 = np.concatenate((R.T, R.T@-t), axis=1)
    C = K@R_3x4
    P = np.concatenate((pts3, np.ones((1, pts3.shape[1]))), axis=0
    pts2 = C@P
    if (pts2[2][0]!=0):
        pts2 = pts2[:2] / pts2[2]
    else:
        pts2 = pts2[:2]

    assert(pts2.shape[1]==pts3.shape[1])
    assert(pts2.shape[0]==2)

    return pts2


def update_extrinsics(self,params):
    """
    Given a vector of extrinsic parameters, update the camera
    to use the provided parameters.

    Parameters
    ----------
    params : 1D numpy.array of shape (6,) (dtype=float)
        Camera parameters we are optimizing over stored in a vecto
        params[:3] are the rotation angles, params[3:] are the tra

    """
    rx, ry, rz = params[:3]
    self.R = makerotation(rx, ry, rz)
    self.t = np.array([params[3:]]).T
```

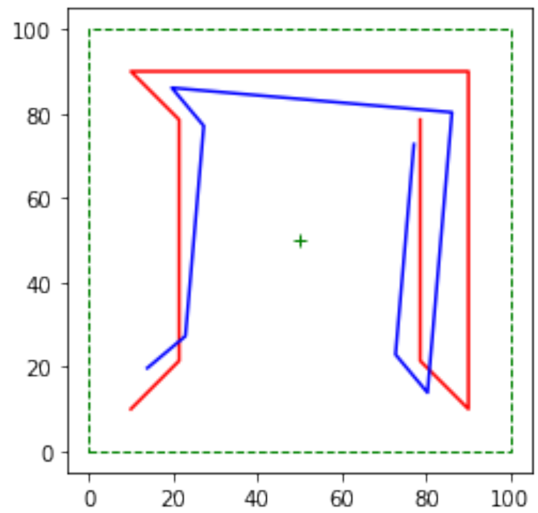
In [5]:
```python
def residuals(pts3,pts2,cam,params):
    """
    Compute the difference between the projection of 3D points by the
    with the given parameters and the observed 2D locations

    Parameters
    ----------
    pts3 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (3,N)

    pts2 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (2,N)

    params : 1D numpy.array (dtype=float)
        Camera parameters we are optimizing stored in a vector of shap

    Returns
    -------
    residual : 1D numpy.array (dtype=float)
        Vector of residual 2D projection errors of size 2*N

    """
    cam.update_extrinsics(params)
    temp = pts2 - cam.project(pts3)
    return temp.flatten()
```

In [6]:

```
#
# Test the residual function to make sure it is doing the right thing.
#

# create two cameras with same intrinsic but slightly different extrin
camA = Camera(f=200,c=np.array([[50,50]]).T,t=np.array([[0,0,0]]).T, R
camB = Camera(f=200,c=np.array([[50,50]]).T,t=np.array([[0,0,0]]).T, R

paramsA = np.array([0,0,0,0.5,0.5,-2.5])
paramsB = np.array([0,0,5,0.5,0.5,-3.0])
camA.update_extrinsics(paramsA)
camB.update_extrinsics(paramsB)

print(camA)
print(camB)


# create a test object (corners of a 3D cube)
pts3 = np.array([[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,0,0],[1,0,


# visualize the two projections
pts2A = camA.project(pts3)
pts2B = camB.project(pts3)

plt.plot(pts2A[0,:],pts2A[1,:],'r')
plt.plot(pts2B[0,:],pts2B[1,:],'b')


#visualize the sensor frame assuming our sensor is centered around the
plt.gca().add_patch(patches.Rectangle((0,0),2*camA.c[0,0],2*camA.c[1,0
plt.plot(camA.c[0],camA.c[1],'g+')
plt.axis('square')
plt.show()


# double check that the residuals are the same as the difference in th
print("\n residuals of camB relative to camA")
print(np.all( residuals(pts3,pts2A,camB,paramsB) == (pts2A-pts2B).flat
# print(pts2A-pts2B)

print("\n residuals of camA relative to camB")
print( np.all( residuals(pts3,pts2B,camA,paramsA) == (pts2B-pts2A).fla
# print(pts2B-pts2A)
```

```
Camera :
 f=200
 c=[[50 50]]
 R=[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
 t = [[ 0.5  0.5 -2.5]]
```

```
Camera :
 f=200
 c=[[50 50]]
 R=[[ 0.9962 -0.0872  0.     ]
 [ 0.0872  0.9962  0.     ]
 [ 0.      0.      1.     ]]
 t = [[ 0.5  0.5 -3. ]]
```




```
 residuals of camB relative to camA
True

 residuals of camA relative to camB
True
```

3. Camera Pose Estimation

We are now ready to estimate camera pose using optimize. Implement a function **calibratePose** which takes as input the 3D coordinates of a calibration object, the observed 2D coordinates in the image, and an initial guess of the camera. Your function should use **scipy.optimize.leastsq** to optimize the extrinsic parameters in order to minimize the reprojection error. Since the **residuals** function takes additional arguments and **leastsq** expects a function which only takes the parameter vector as input, you should use Python's **lambda** function to wrap **residuals**, subistituting in the parameters that are fixed during the optimization. Once you have determined the optimum parameters, update the extrinsic parameters to the optimum and return the resulting camera.

3.1 Implementation [30pts]

In [7]:
```
def calibratePose(pts3,pts2,cam,params_init):
    """
    Calibrate the provided camera by updating R,t so that pts3 project
    as close as possible to pts2

    Parameters
    ----------
    pts3 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (3,N)

    pts2 : 2D numpy.array (dtype=float)
        Coordinates of N points stored in a array of shape (2,N)

    cam : Camera
        Initial estimate of camera

    params_init : 1D numpy.array (dtype=float)
        Initial estimate of camera extrinsic parameters ()
        params[0:3] are the rotation angles, params[3:6] are the trans

    Returns
    -------
    cam : Camera
        Refined estimate of camera with updated R,t parameters

    """

    efun = lambda x : residuals(pts3, pts2, cam, x)

    params_opt, _ = scipy.optimize.leastsq(efun, params_init)

    cam.update_extrinsics(params_opt)

    return cam
```

3.2 Synthetic Test Example and Failure Cases [5pts]

Use the code below to check that your calibrate function works. Add some code to also visualize the point locations in 3D and the location and orientation of the camera (i.e., using the 3D plotting functions from Assignment 2)

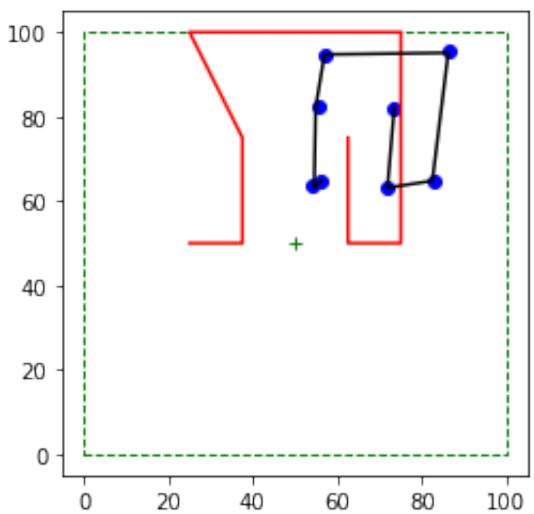
Once you are confident that your calibration function is behaving correctly, you should experiment with changing the initial parameters. Find a set of initial parameters which yields a ***wrong*** solution (i.e. where the Final Camera is not similar to the True Camera). In the text box below indicate what bad initialization you used and the resulting set of camera parameters after the optimization. Give a brief explanation of where this bad camera is located and what direction it is oriented in.

In [8]:

```
# 3D calibration object
pts3 = np.array([[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,0,0],[1,0,

# true camera
cam_true = Camera(f=50,c=np.array([[50,50]]).T,t=np.array([[-0.25,-0.2

print("\n True Camera")
print(cam_true)

# image of calibration object with some simulated noise in the 2D loca
pts2 = cam_true.project(pts3)
noiselevel = 0.50
pts2 = pts2 + noiselevel*np.random.randn(pts2.shape[0],pts2.shape[1])

# initial guess of camera params
cam = Camera(f=50,c=np.array([[50,50]]).T,t=np.array([[0,0,0]]).T, R=m
params_init = np.array([0,0,0,0.5,0,-1])
cam.update_extrinsics(params_init)

print("\n Initial Camera")
print(cam)
pts2init = cam.project(pts3)

# now run calibration
cam = calibratePose(pts3,pts2,cam,params_init)

print("\n Final Camera")
print(cam)
pts2final = cam.project(pts3)

#
# Plot the true, initial and final reprojections
# The final reprojection should be on top of the true image
#
plt.plot(pts2[0,:],pts2[1,:],'bo')
plt.plot(pts2init[0,:],pts2init[1,:],'r')
plt.plot(pts2final[0,:],pts2final[1,:],'k')
plt.gca().add_patch(patches.Rectangle((0,0),2*cam.c[0,0],2*cam.c[1,0],
plt.plot(cam.c[0],cam.c[1],'g+')
plt.plot(cam_true.c[0],cam_true.c[1])
plt.axis('square')
plt.show()

#
# Add some additional visualiztion here to show the points in 3D and t
# of cam_true and cam.  You can either use a 3D plot or show multiple
# and side views)
#
```

```
 True Camera
Camera :
```

```
f=50
 c=[[50 50]]
 R=[[ 1.      0.      0.    ]
 [ 0.      0.9848 -0.1736]
 [ 0.      0.1736  0.9848]]
 t = [[-0.25 -0.25 -2.  ]]

 Initial Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
 t = [[ 0.5  0.  -1. ]]

 Final Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[ 0.9997 -0.0067 -0.0223]
 [ 0.0022  0.9808 -0.195 ]
 [ 0.0232  0.1949  0.9805]]
 t = [[-0.1909 -0.1837 -2.0084]]
```



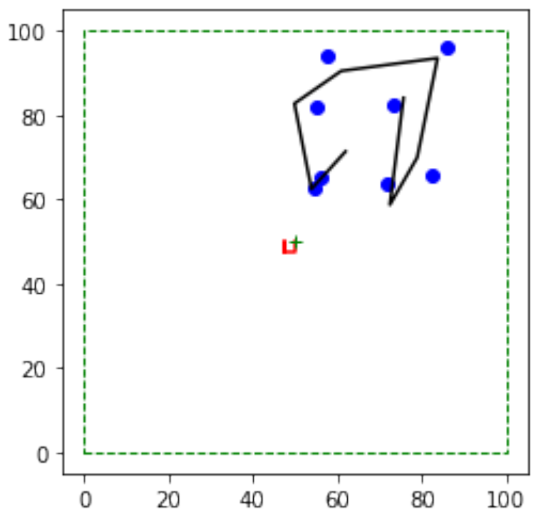
In [9]:

```
#
# Now repeat the calibration but with a setting for params_init that r
# in the optimization finding a poor solution (a bad local minima)
#

# 3D calibration object
pts3 = np.array([[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,0,0],[1,0,

# true camera
cam_true = Camera(f=50,c=np.array([[50,50]]).T,t=np.array([[-0.25,-0.2

print("\n True Camera")
print(cam_true)
```

```
# image of calibration object with some simulated noise in the 2D loca
pts2 = cam_true.project(pts3)
noiselevel = 0.5
pts2 = pts2 + noiselevel*np.random.randn(pts2.shape[0],pts2.shape[1])

# initial guess of camera params
cam = Camera(f=50,c=np.array([[50,50]]).T,t=np.array([[0,0,0]]).T, R=m
params_init = np.array([0,0,0,0,0,20])
cam.update_extrinsics(params_init)

print("\n Initial Camera")
print(cam)
pts2init = cam.project(pts3)

# now run calibration
cam = calibratePose(pts3,pts2,cam,params_init)

print("\n Final Camera")
print(cam)
pts2final = cam.project(pts3)

#
# Plot the true, initial and final reprojections
# The final reprojection should be on top of the true image
#
plt.plot(pts2[0,:],pts2[1,:],'bo')
plt.plot(pts2init[0,:],pts2init[1,:],'r')
plt.plot(pts2final[0,:],pts2final[1,:],'k')
plt.gca().add_patch(patches.Rectangle((0,0),2*cam.c[0,0],2*cam.c[1,0],
plt.plot(cam.c[0],cam.c[1],'g+')
plt.plot(cam_true.c[0],cam_true.c[1])
plt.axis('square')
plt.show()



#
# Add some additional visualiztion here to show the points in 3D and t
# of cam_true and cam.  You can either use a 3D plot or show multiple
# and side views)
#
# Visualize the resulting bad solution.
#
```

```
 True Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[ 1.      0.      0.    ]
 [ 0.      0.9848 -0.1736]
 [ 0.      0.1736  0.9848]]
 t = [[-0.25 -0.25 -2.  ]]
```

```
 Initial Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
 t = [[ 0  0 20]]

 Final Camera
Camera :
 f=50
 c=[[50 50]]
 R=[[-0.7995  0.2993  0.5209]
 [ 0.2225 -0.6578  0.7195]
 [ 0.558   0.6912  0.4593]]
 t = [[1.4863 1.5957 2.878 ]]
```



[[**describe the failure mode here... how is the camera located and oriented for the bad local minima?**]]

params_init = np.array([0,0,0,0,0,20])

So camera is facing points from opposite side, very far away which makes a bad local minima

4. Calibration from real images [25pts]

There is a provided set of calibration images (images of a planar checkerboard) along with stereo pair depicting an object. In order to calibrate the intrinsic camera parameters we will use the OpenCV library which includes functionality for automatically detecting corners of the checkerboard and estimating intrinsic parameters. To install OpenCV python libraries in your Anaconda environment. You can do this from the terminal via the command **conda install opencv** or via the Anconda Navigator gui.

I have provide a standalone script **calibrate.py** which uses OpenCV to carry out calibration of the camera intrinsic parameters for a series of checkerboard images. Read through the provided script to understand the code and modify file paths as necessary in order to compute the intrinsic camera parameters from the set of provided calibration images.

4.1 Implementation

Fill in the code snippet below to carry out the following steps.

0. Run the **calibrate.py** script to estimate the intrinsic camera parameters.
1. Load in the intrinsic parameter calibration data saved by the script in *calibration.pickle*. Since our camera model assumes that the focal length is the same in the x and y axes, you can set your f to be the average of the two estimated by the script.
2. Load in the test images *Left.jpg* and *Right.jpg* and use the **cv2.findChessboardCorners** function in order to automatically get the 2D coordinates of the corners in the image.
3. Specify the true 3D coordinates of the 6x8 grid of checkerboard corners. The squares are 2.8cm x 2.8cm.
4. Use your **calibratePose** function to estimate the R,t for each camera. You will likely need to experiment with selecting the initial parameters in order to get a good solution (e.g., translate so the cameras have positive z coordinates and rotate so they are looking down on the checkerboard).
5. Finally, as a consistency check, once you have the calibrated pose for each camera, you can use your triangulate function to estimate the 3D coordinates of the checkerboard corners based on the 2D points in the left and right camera. The re-triangulated points should be close to the specified true 3D coordinates.

In [10]:
```
def triangulate(pts2L,camL,pts2R,camR):
    """
    Triangulate the set of points seen at location pts2L / pts2R in th
    corresponding pair of cameras. Return the 3D coordinates relative
    to the global coordinate system


    Parameters
    ----------
    pts2L : 2D numpy array (dtype=float)
```

```
    pts2L : 2D numpy.array (dtype=float)
        Coordinates of N points stored in an array of shape (2,N) seen

    pts2R : 2D numpy.array (dtype=float)
        Coordinates of N points stored in an array of shape (2,N) seen

    camL : Camera
        The first "left" camera view

    camR : Camera
        The second "right" camera view

    Returns
    -------
    pts3 : 2D numpy.array (dtype=float)
        (3,N) array containing 3D coordinates of the points in global

    """

    #
    # Your code goes here.  I recommend adding assert statements to ch
    # sizes of the inputs and outputs to make sure they are correct
    #
    assert(pts2L.shape[0]==2)
    assert(pts2R.shape[0]==2)
    assert(pts2L.shape==pts2R.shape)

    pts3 = np.array([[0, 0, 0]]).T

    for i in range(pts2L.shape[1]):

        # get x, y camera coordinates
        xl, yl = pts2L[:2, i]
        xr, yr = pts2R[:2, i]

        # subtract principle points to get image coordinates

        xl, yl = xl-camL.c[0][0], yl-camL.c[1][0]
        xr, yr = xr-camR.c[0][0], yr-camR.c[1][0]

        # get rotation mtx
        Rl, Rr = camL.R, camR.R

        # get focal length
        fl, fr = camL.f, camR.f

        # get translation vector
        T = camR.t - camL.t

        # calculate vector q / focal length
        qL_over_f = np.array([[ xl/fl, yl/fl, 1 ]]).T
        qR_over_f = np.array([[ xr/fr, yr/fr, 1 ]]).T

        A = np.hstack((Rl@qL_over_f, -Rr@qR_over_f))
```

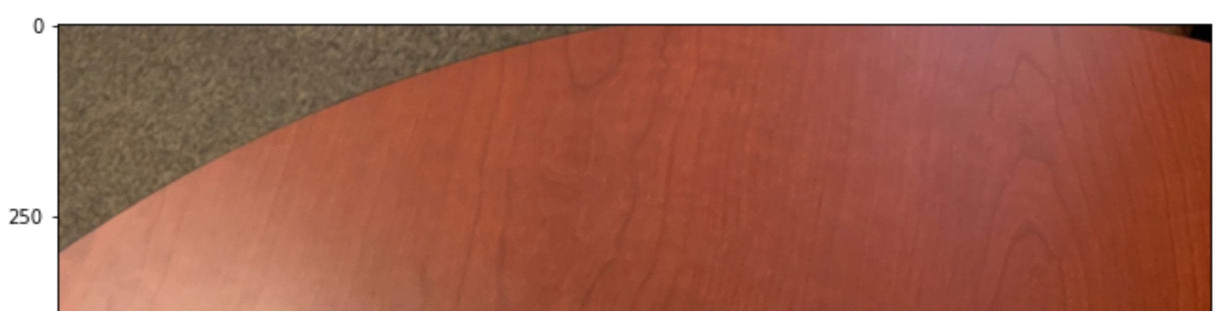
```
        # least squares
        z = np.linalg.lstsq(A, T, rcond=None)[0]

        zL = z[0][0]
        Pl = zL * qL_over_f
        P1 = Rl@Pl + camL.t

        zR = z[1][0]
        Pr = zR * qR_over_f
        P2 = Rr@Pr + camR.t

        P = .5*(P1+P2)

        pts3 = np.hstack((pts3, P))

    pts3 = pts3[:,1:]

    assert(pts3.shape == (3, pts2L.shape[1]))
    return pts3
```

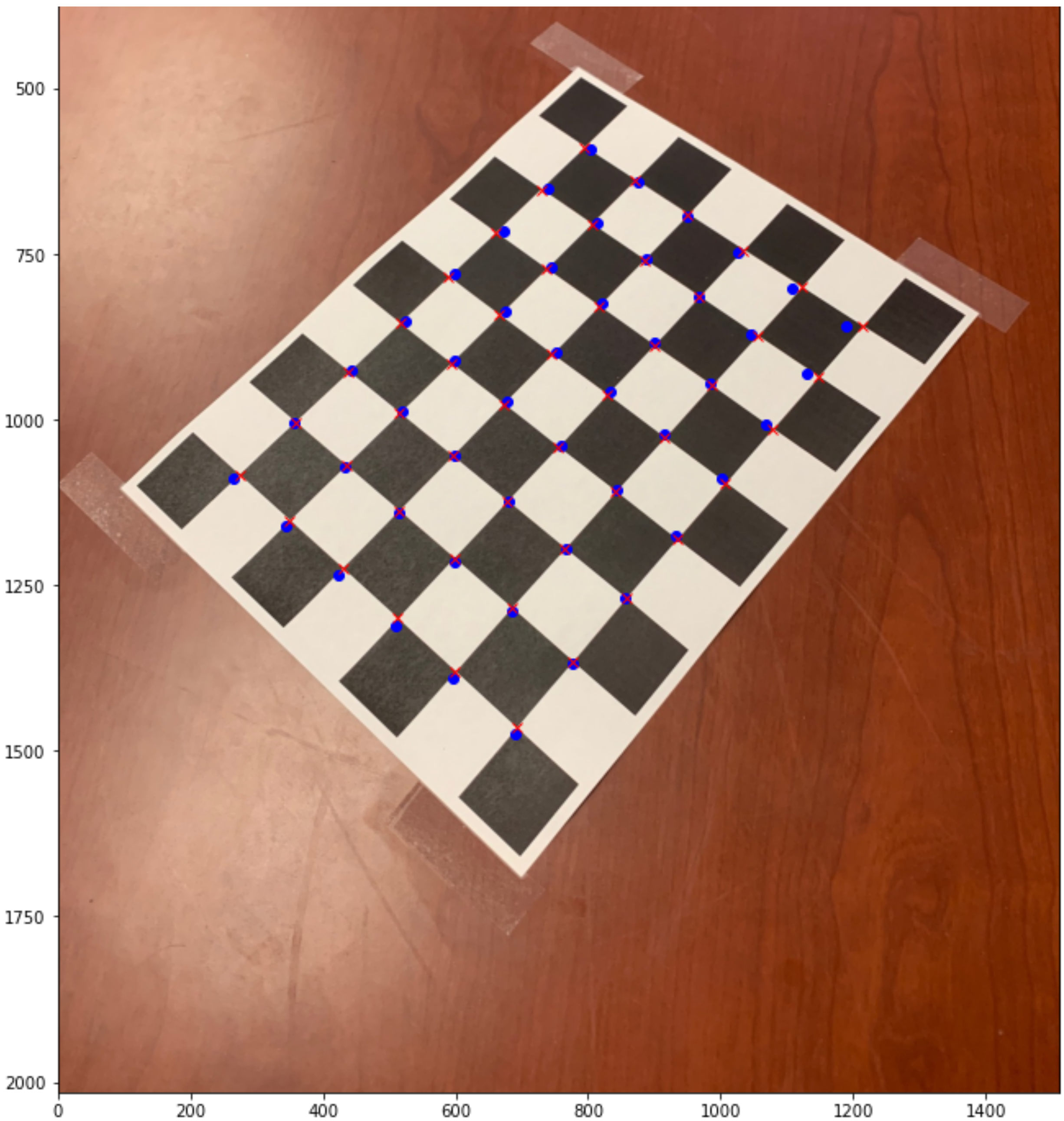
In [11]:

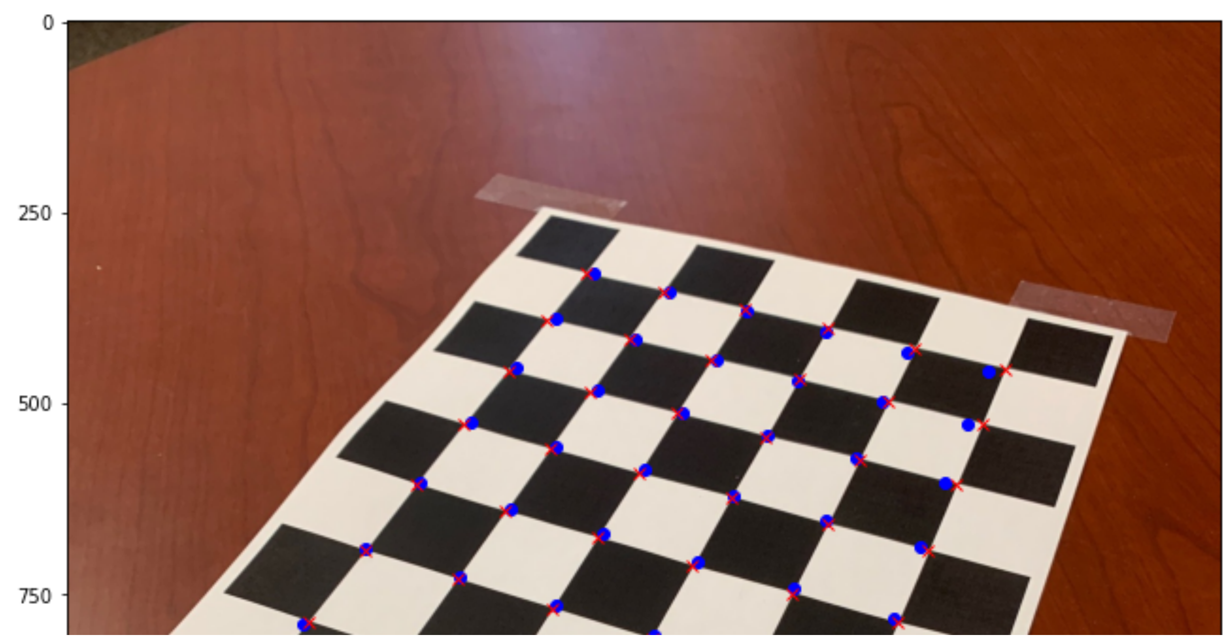
```
import cv2
import pandas as pd

# load in the intrinsic camera parameters from 'calibration.pickle'
obj = pd.read_pickle('calibration.pickle')

fx, fy, cx, cy = obj.get('fx'), obj.get('fy'), obj.get('cx'), obj.get(
f = (fx + fy) / 2.0

# create Camera objects representing the left and right cameras
# use the known intrinsic parameters you loaded in.
camL = Camera(f=f, c=np.array([[cx,cy]]).T, t=np.array([[0,0,0]]).T, R
camR = Camera(f=f, c=np.array([[cx,cy]]).T, t=np.array([[0,0,0]]).T, R

# load in the left and right images and find the coordinates of
# the chessboard corners using OpenCV
imgL = plt.imread('calib1/Left.jpg')
ret, cornersL = cv2.findChessboardCorners(imgL, (8,6), None)
pts2L = cornersL.squeeze().T

imgR = plt.imread('calib1/Right.jpg')
ret, cornersR = cv2.findChessboardCorners(imgR, (8,6), None)
pts2R = cornersR.squeeze().T

# generate the known 3D point coordinates of points on the checkerboar
pts3 = np.zeros((3,6*8))
yy,xx = np.meshgrid(np.arange(8),np.arange(6))
pts3[0,:] = 2.8*xx.reshape(1,-1)
pts3[1,:] = 2.8*yy.reshape(1,-1)


# Now use your calibratePose function to get the extrinsic parameters
# for the two images. You may need to experiment with the initializati
```

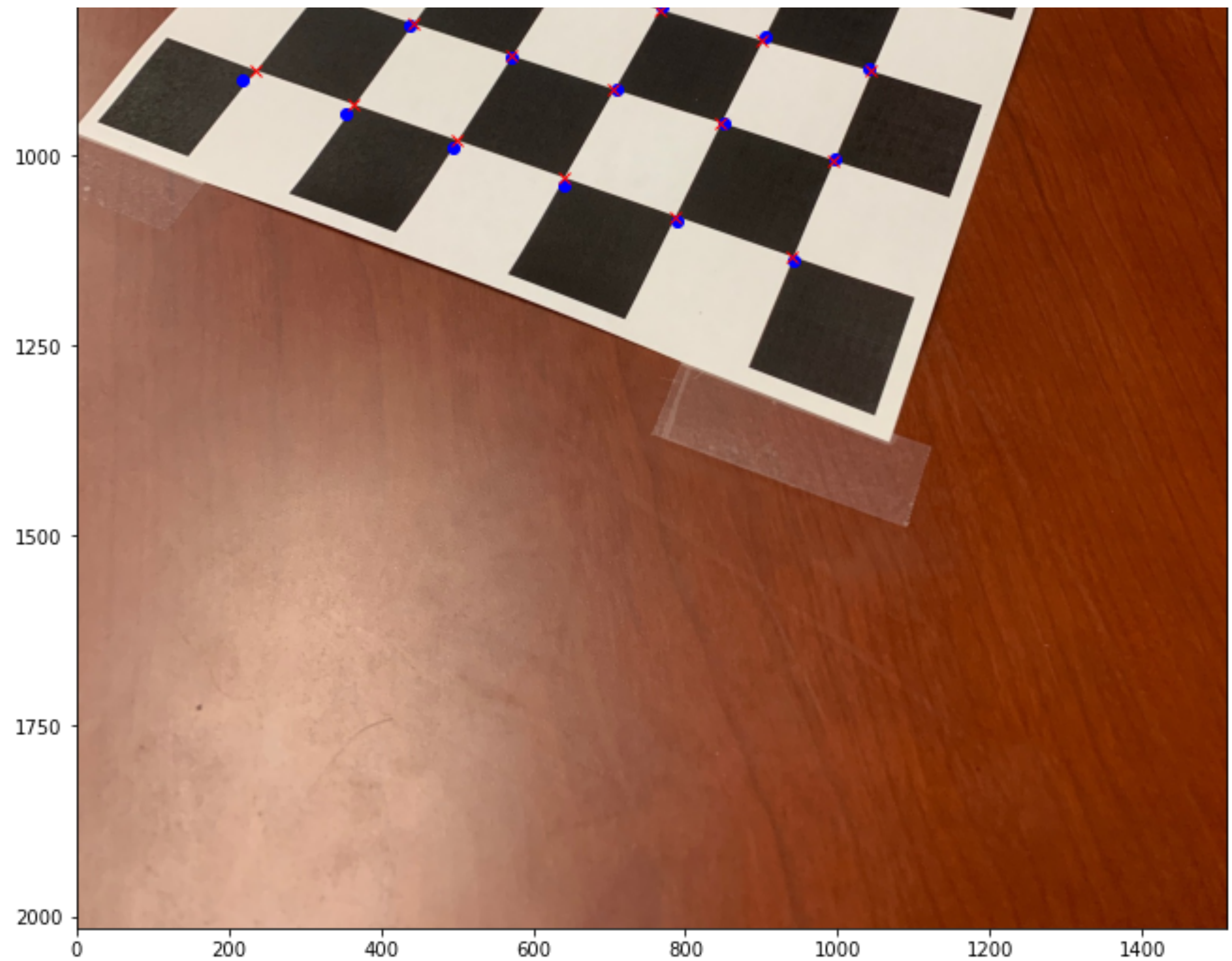
```python
# in order to get a good result

...
params_init = [0, 0, 0, 1, 1, -1]
camL = calibratePose(pts3,pts2L,camL,params_init)

params_init = [0, 0, 0, -1, 0, -1]
camR = calibratePose(pts3,pts2R,camR,params_init)

print(camL)
print(camR)

# As a final test, triangulate the corners of the checkerboard to get
pts3r = triangulate(pts2L,camL,pts2R,camR)

# Display the reprojected points overlayed on the images to make
# sure they line up
plt.rcParams['figure.figsize']=[15,15]
pts2Lp = camL.project(pts3)
plt.imshow(imgL)
plt.plot(pts2Lp[0,:],pts2Lp[1,:],'bo')
plt.plot(pts2L[0,:],pts2L[1,:],'rx')
plt.show()

pts2Rp = camR.project(pts3)
plt.imshow(imgR)
plt.plot(pts2Rp[0,:],pts2Rp[1,:],'bo')
plt.plot(pts2R[0,:],pts2R[1,:],'rx')
plt.show()
```

```
Camera :
 f=1561.0139694131512
 c=[[1021.1465  755.8365]]
 R=[[ 0.7928  0.5479 -0.2671]
 [ 0.6058 -0.66    0.4443]
 [ 0.0671 -0.514  -0.8552]]
 t = [[20.2533 -0.3845 37.7861]]
Camera :
 f=1561.0139694131512
 c=[[1021.1465  755.8365]]
 R=[[ 0.9506  0.271  -0.1511]
 [ 0.3029 -0.7042  0.6421]
 [ 0.0676 -0.6562 -0.7516]]
 t = [[ 17.4715 -11.9475  24.0513]]
```




500
750
1000
1250
1500
1750
2000
0
200
400
600
800
1000
1200
1400


0
250
500
750

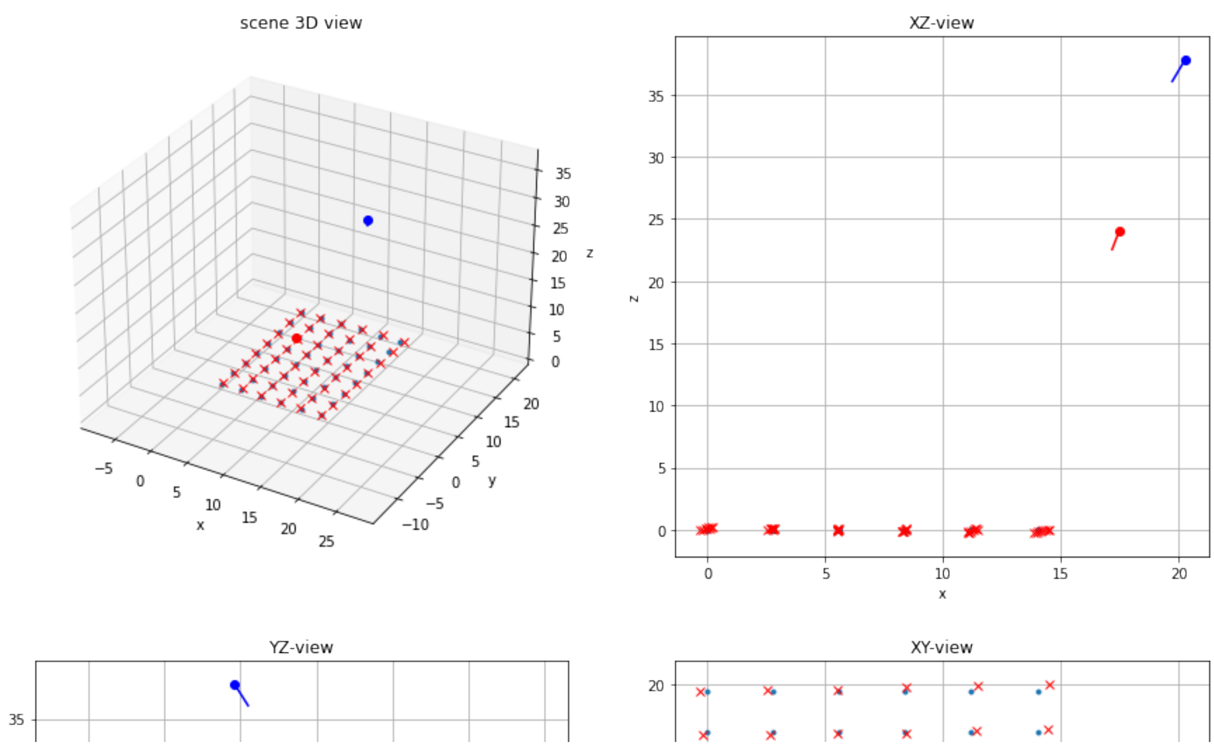


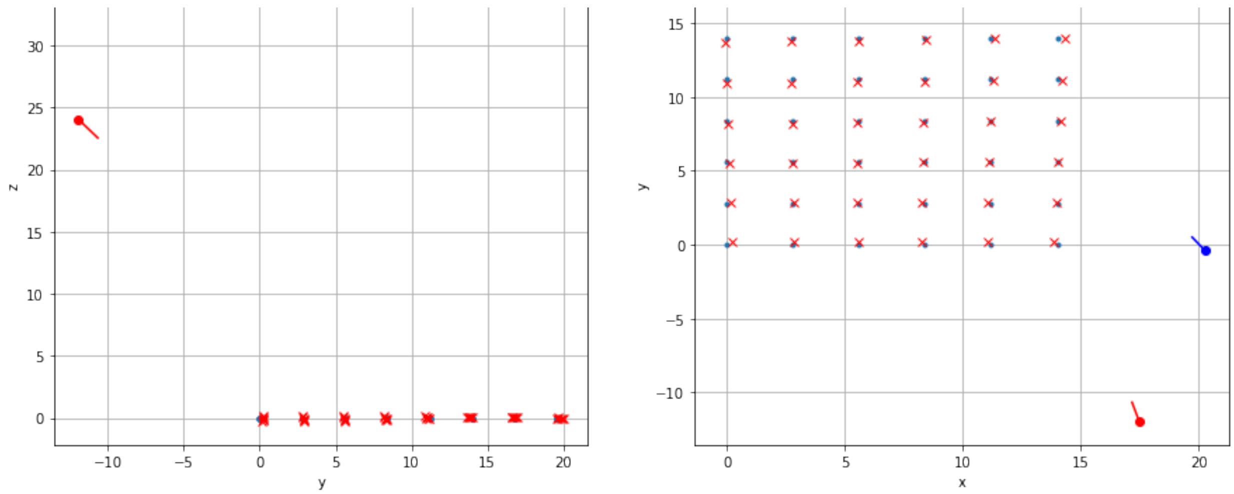
The code below provides a visualization of the estimate camera positions relative to the checkerboard.

In [12]:
```
# generate coordinates of a line segment running from the center
# of the camera to 3 units in front of the camera
lookL = np.hstack((camL.t,camL.t+camL.R @ np.array([[0,0,2]]).T))
lookR = np.hstack((camR.t,camR.t+camR.R @ np.array([[0,0,2]]).T))

# visualize the left and right image overlaid
fig = plt.figure()
ax = fig.add_subplot(2,2,1,projection='3d')
ax.plot(pts3[0,:],pts3[1,:],pts3[2,:],'.')
ax.plot(pts3r[0,:],pts3r[1,:],pts3r[2,:],'rx')
ax.plot(camR.t[0],camR.t[1],camR.t[2],'ro')
ax.plot(camL.t[0],camL.t[1],camL.t[2],'bo')
ax.plot(lookL[0,:],lookL[1,:],lookL[2,:],'b')
ax.plot(lookR[0,:],lookR[1,:],lookR[2,:],'r')
visutils.set_axes_equal_3d(ax)
visutils.label_axes(ax)
plt.title('scene 3D view')

ax = fig.add_subplot(2,2,2)
ax.plot(pts3[0,:],pts3[2,:],'.')
ax.plot(pts3r[0,:],pts3r[2,:],'rx')
ax.plot(camR.t[0],camR.t[2],'ro')
```

```
ax.plot(camL.t[0],camL.t[2],'bo')
ax.plot(lookL[0,:],lookL[2,:],'b')
ax.plot(lookR[0,:],lookR[2,:],'r')
plt.title('XZ-view')
plt.grid()
plt.xlabel('x')
plt.ylabel('z')

ax = fig.add_subplot(2,2,3)
ax.plot(pts3[1,:],pts3[2,:],'.')
ax.plot(pts3r[1,:],pts3r[2,:],'rx')
ax.plot(camR.t[1],camR.t[2],'ro')
ax.plot(camL.t[1],camL.t[2],'bo')
ax.plot(lookL[1,:],lookL[2,:],'b')
ax.plot(lookR[1,:],lookR[2,:],'r')
plt.title('YZ-view')
plt.grid()
plt.xlabel('y')
plt.ylabel('z')

ax = fig.add_subplot(2,2,4)
ax.plot(pts3[0,:],pts3[1,:],'.')
ax.plot(pts3r[0,:],pts3r[1,:],'rx')
ax.plot(camR.t[0],camR.t[1],'ro')
ax.plot(camL.t[0],camL.t[1],'bo')
ax.plot(lookL[0,:],lookL[1,:],'b')
ax.plot(lookR[0,:],lookR[1,:],'r')
plt.title('XY-view')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
```

Out[12]: Text(0, 0.5, 'y')





4.2 Recovered Pose

Using the provided calibration images, what are the recovered parameters for the left and right cameras? How far apart are the camera centers in centimeters (i.e. what is the baseline) ?

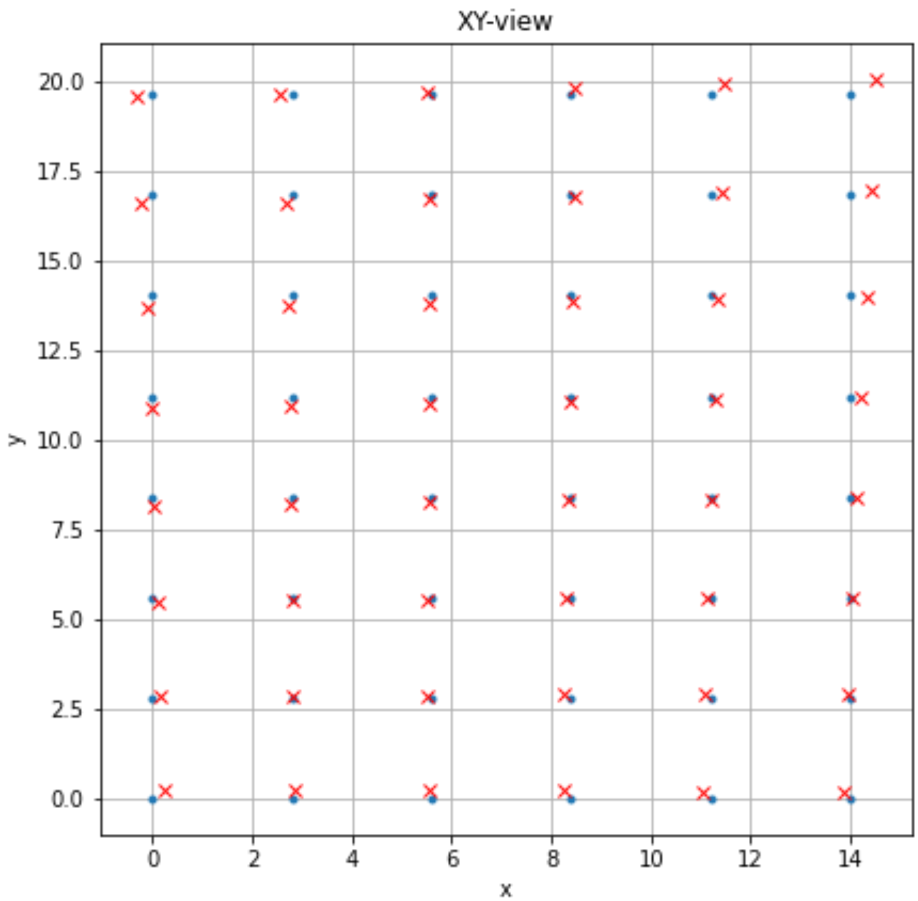
In [13]:
```
#    [ x              y              z           ]
tL = [20.25334604, -00.38445876,  37.78610728]
tR = [17.47153954, -11.94747213,  24.0512738 ]

x1, y1, z1 = tL[0], tL[1], tL[2]
x2, y2, z2 = tR[0], tR[1], tR[2]

# compute distance of 3D point
dist_sqare = (x1 - x2)**2 + (y1 - y2)**2 + (z1 - z2)**2
dist_sqrt = dist_sqare**.5


print("Baseline is", round(dist_sqrt,2), "cm")
```

```
Baseline is 18.17 cm
```

4.3 Reconstruction Accuracy

Using the estimated camL and camR and the 2D point locations, triangulate to get 3D locations. What is the average error (in cm) for your recovered 3D locations of the grid corner points relative to their true coordinates? Where might this error be coming from?

In [14]:
```
fig = plt.figure()
ax = fig.add_subplot(2,2,4)
ax.plot(pts3[0,:],pts3[1,:],'.')
ax.plot(pts3r[0,:],pts3r[1,:],'rx')
plt.title('XY-view')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.show()

merrorX = np.mean(np.abs(pts3r[0] - pts3[0]))
merrorY = np.mean(np.abs(pts3r[1] - pts3[1]))

print('mean error: ', merrorX + merrorY)
```




```
mean error:  0.2618530840435475
```

The likely cause of this error is

4.4 Focal Length

The checkerboard photos were taken with an iPhone Xs. Teardowns of this device reveal that the sensor 5.6mm x 4.2mm in size. Based on this and your recovered value for f, what was the focal length in millimeters? Explain how you computed this. Is the result you get a reasonable match to the published focal length of of 4.25mm?

The recovered principle points were cx=1021 and cy=756 as well as a focal length of f=1561, measured in pixels.

Multiplying cx and cy by 2 would give us the width and height of the sensor in pixels:

W=2042 and H=1512 measured in pixels.

Given that the sensor is 5.6mm x 4.2mm in size, we can determine by division the conversion from mm to pixels to be:

1mm = 360 pixels

Therefore, the focal length in mm must be 1561 / 360 = 4.34mm.

Depending on how we define a 'reasonalble' match to be, the percent difference is %diff = (4.34 - 4.25)/4.25 * 100% whcih is a percent differnce of 2.07%. I would say this is a 'reasonable' match given a low %diff such as 2.07%.