Computer Vision
Mini-Project 4 - Stereo Vision

Team names:

Mohamed Abdelwahed	201801978
Ibrahim Abdalaal	201800224

Introduction

In this project ,we have gone through three main parts :

- **Part1: in which we calculate the projection matrix from the 2d image points and their corresponding world 3d points.**
- **Part2 :calibrating two cameras using a chessboard pattern forming stereovision.**
- **parts 3: performing sift detection and matching between two of our captured images and applying epipolar constraint to increase the accuracy of the matches**

Part1

We calculated the projection matrix through the given 2d and 3d correspondences by solving System of equations in the form **AU=B** using least square where U is the projection matrix then we calculated the camera center through the following steps :

- `calculate_projection_matrix`

1. constructing the matrix A from the given 2d and 3d points
2. Constructing matrix B by reshaping the given 2d points
3. Solving using the linear technique where **U=(A.T*A)^-1 *A.T*B**

student.py > calculate_projection_matrix

```python
def calculate_projection_matrix(Points_2D, Points_3D):

    print('Randomly setting matrix entries as a placeholder')
    M = np.array([[0.1768, 0.7018, 0.7948, 0.4613],
                  [0.6750, 0.3152, 0.1136, 0.0480],
                  [0.1020, 0.1725, 0.7244, 0.9932]])
    ##################
    #Solving using least square  AU=B
    #dimensons :
    # A->  (N*2,12)  |||| B->(N*2,1)   ||| U -> (12,1) to be rehsaped
    print(Points_2D)
    N=Points_3D.shape[0]          ## N
    A=np.zeros((N*2,11))
    B=np.zeros((N*2,1))
    i=0                       ## counter to iterate over marix
    #constructing A
    for j in range (N):
        ##buiding x row
        A[i,0:3]=Points_3D[j,:]   #first 3 elemnts in each row of A
        A[i,3]=1                  #fourth elemet in A
        last_elemnts=Points_2D[j,0]*Points_3D[j,:]  ##last 3 elements of each x row
        A[i,8:11]=-1*last_elemnts
        #A[i,11]=-1*Points_2D[j,0]

        ##Building y row
        A[i+1,4:7]=Points_3D[j,:]   #first 3 elemnts in each row of A
        A[i+1,7]=1                  #fourth elemet in A
        last_elemnts=Points_2D[j,1]*Points_3D[j,:]  ##last 3 elements of each x row
        A[i+1,8:11]=-1*last_elemnts
        #A[i+1,11]=-Points_2D[j,1]
        i+=2
    #print(A)
    B=np.reshape(Points_2D,(-1,1))
    AT_A=np.matmul(A.T,A )   # A^T *A
    inverse =np.linalg.inv(AT_A)
    AT_B=np.matmul(A.T,B)
    U=np.matmul(inverse,AT_B)
    U=np.append(U,1)
    M=np.reshape(U,(3,4))
    return M
```

- **compute_camera_center(M)**

    1. M=[Q|m4]
    2. c=Q^-1*m4

```
def compute_camera_center(M):

    Center = np.array([1, 1, 1])
    Q=M[:,0:3]
    m4=M[:,3]
    Center=np.matmul(-1*np.linalg.inv(Q),m4)


    return Center
```

Results of part1

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

The projection matrix is:
 [[ 0.76785834 -0.49384797 -0.02339781  0.00674445]
 [-0.0852134  -0.09146818 -0.90652332 -0.08775678]
 [ 0.18265016  0.29882917 -0.07419242  1.        ]]

The total residual is:
 0.0445349939493l366

The estimated location of the camera is:
 [-1.51263977 -2.35165965  0.28266502]
```

These results are the same with the expected ones reported in the project document.
Total residual is very good and acceptable.

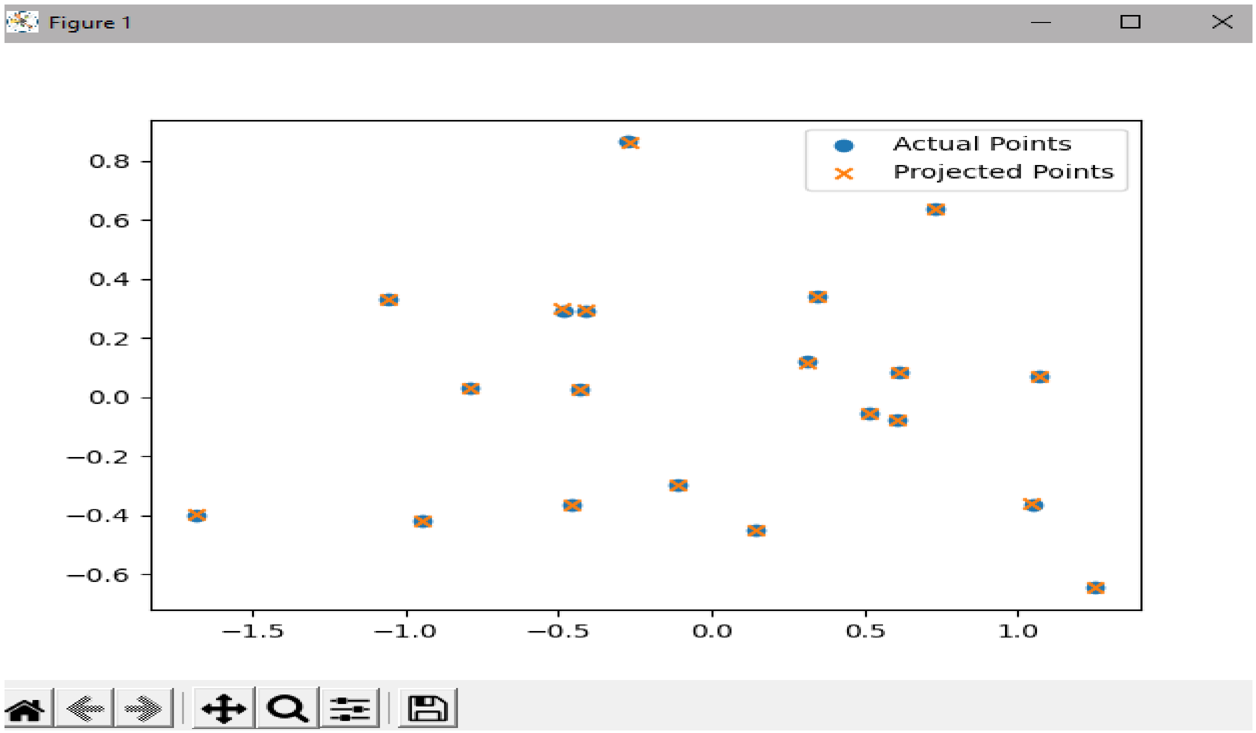


Figure 1
Actual Points
Projected Points
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−1.5
−1.0
−0.5
0.0
0.5
1.0

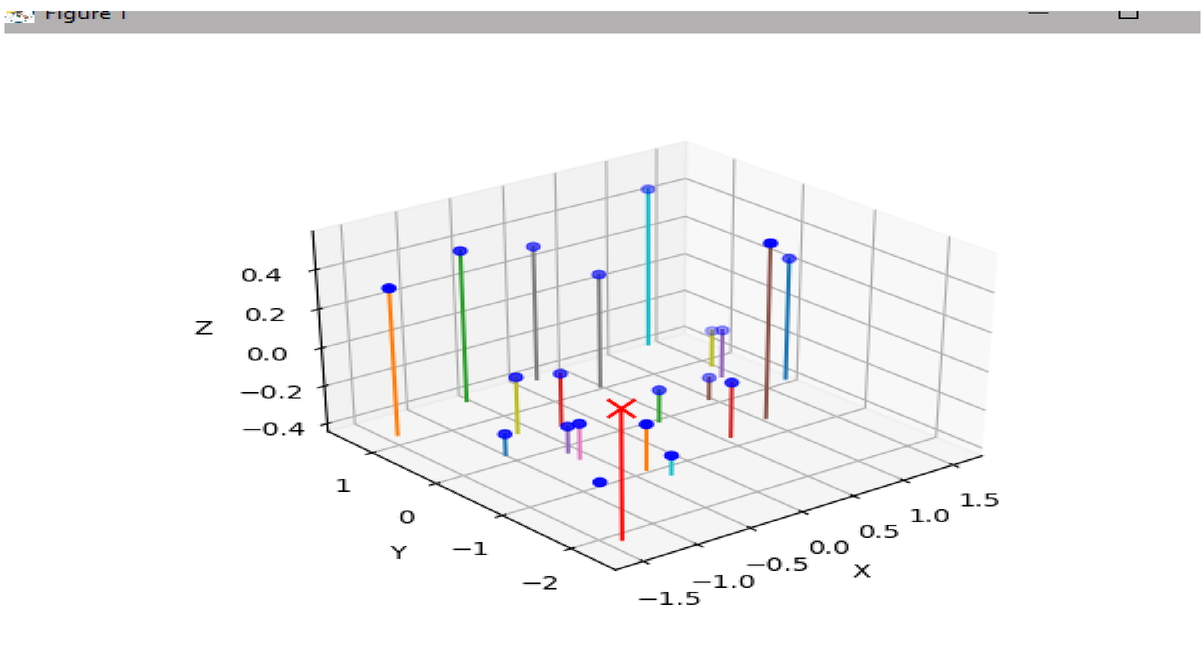


Figure 1
Z
0.4
0.2
0.0
−0.2
−0.4
Y
1
0
−1
−2
X
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Part2

In this part we calibrated our own cameras using chessboard pattern and then we estimated the essential matrix using stereocalibrate function :

- **CamerCalibrate() [function implementation using chessboard pattern]**
    1. **Starting by defining the chessboard dimension which is (6,8)**
    2. **Then defining the world coordinate for the 3d points of the chessboard**
    3. **For each image ,find the corners in the chessboard using findchessborad function**
    4. **If the corners found ,then append the image points and and object points**
    5. **Calibrate the camera using Camera calibrate function**
    6. **Calculate rotation matrix using opencv Rodrigues**
    7. **Concatenating the rotation matrix and translation vector (RT)**
    8. **Getting the projection matrix using by multiplying the camera matrix with RT**

```
def CameraCalibrate(o,ffname):

    ##inputs is :
    #1- o : number of image of wanted projection matrix
    #2- ffname : name of the folder containing the images
    print("Calibrating Camera ....")
    #Defining the dimensions of checkerboard
    CHECKERBOARD = (6,8)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

    # Creating vector to store vectors of 3D points for each checkerboard image
    objpoints = []
    # Creating vector to store vectors of 2D points for each checkerboard image
    imgpoints = []


    # Defining the world coordinates for 3D points
    objp = np.zeros((1, CHECKERBOARD[0]*CHECKERBOARD[1], 3), np.float32)
    objp[0,:,:2] = np.mgrid[0:CHECKERBOARD[0], 0:CHECKERBOARD[1]].T.reshape(-1, 2)
    prev_img_shape = None

    # Extracting path of individual image stored in a given directory
    path='./'+ffname+'/*.jpg'
    images = glob.glob(path)

    #p#rint(images)
    p=0
    for fname in images:
        #print(fname)
        img = cv2.imread(fname)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

```python
        # Find the chess board corners
        # If desired number of corners are found in the image then ret = true
        ret, corners = cv2.findChessboardCorners(gray, CHECKERBOARD,cv2.CALIB_CB_ADAPTIVE_THRESH+
            cv2.CALIB_CB_FAST_CHECK+cv2.CALIB_CB_NORMALIZE_IMAGE)
       # print(ret)
        """
        If desired number of corner are detected,
        we refine the pixel coordinates and display
        them on the images of checker board
        """
        if ret == True:
            objpoints.append(objp)
            # refining pixel coordinates for given 2d points.
            corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

            imgpoints.append(corners2)
            if p==o:
                points_3d= np.array(objp).reshape(-1,3)
                points_2d=np.array(corners2).reshape(-1,2)


            # Draw and display the corners
            img = cv2.drawChessboardCorners(img, CHECKERBOARD, corners2,ret)
            p+=1

    cv2.destroyAllWindows()
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],None,None)
    print("Reprojection Error: \n",ret)
    R = (cv2.Rodrigues(rvecs[o]))[0]
    t = tvecs[o]
    Rt = np.concatenate([R,t], axis=1) # [R|t]
    M = np.dot(mtx,Rt) # A[R|t]
```

```python
    print("Done Calibrating.....")
    return M,points_3d,points_2d,objpoints,imgpoints,mtx,dist,rvecs, tvecs,gray
```

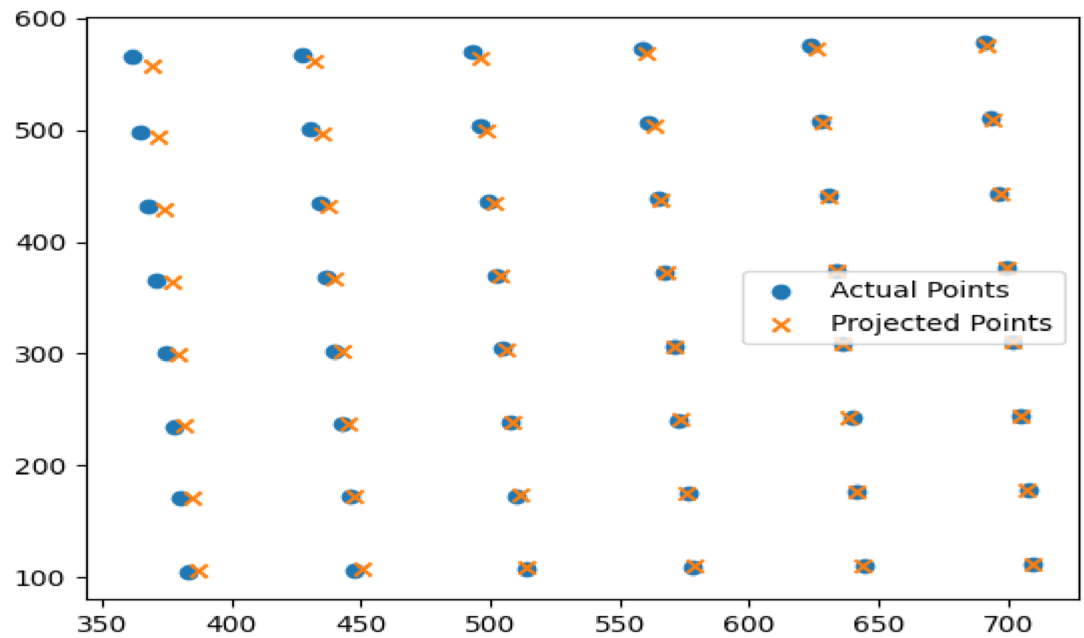
- **Stereo camera**

  Using stereoCalibrate funcion ,we were able to compute the essential matrix to be used later

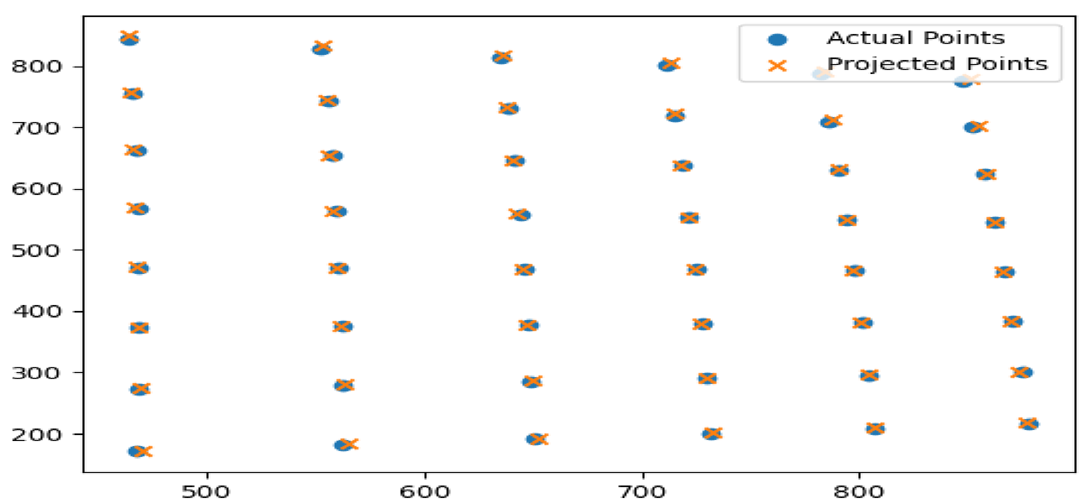
```python
retStereo, newCameraMatrixL, distL, newCameraMatrixR, distR, rot, trans, essentialMatrix,fundamentalMatrix=cv2.stereo
imgpoints1,imgpoints2,mtx1,dist1,mtx2,dist2,gray1.shape[::-1],stereocalib_criteria,flags)
```

Part2_results

- First camera :
  1. Reprojection Error: 0.5689131935306923
  2. The total residual is: 133.69416547523434



- Second camera
  1. Reprojection Error: 0.664144274461620
  2. The total residual is: 98.63853631015635



```
essentialMatrix:
 [[ 2.14139108 -0.1042873   2.89049652]
 [-5.16517039 -2.06069005 -8.31391141]
 [-3.88897355  9.81566736  0.33695337]]
```

Comments on part2 results

Although the total residual is little high for both cameras but the reprojection error is good (in subpixels accuracy) ,also the visualization is not bad

Part3:

In this part we calibrated our own cameras as done before. Then we captured images of a selected object.

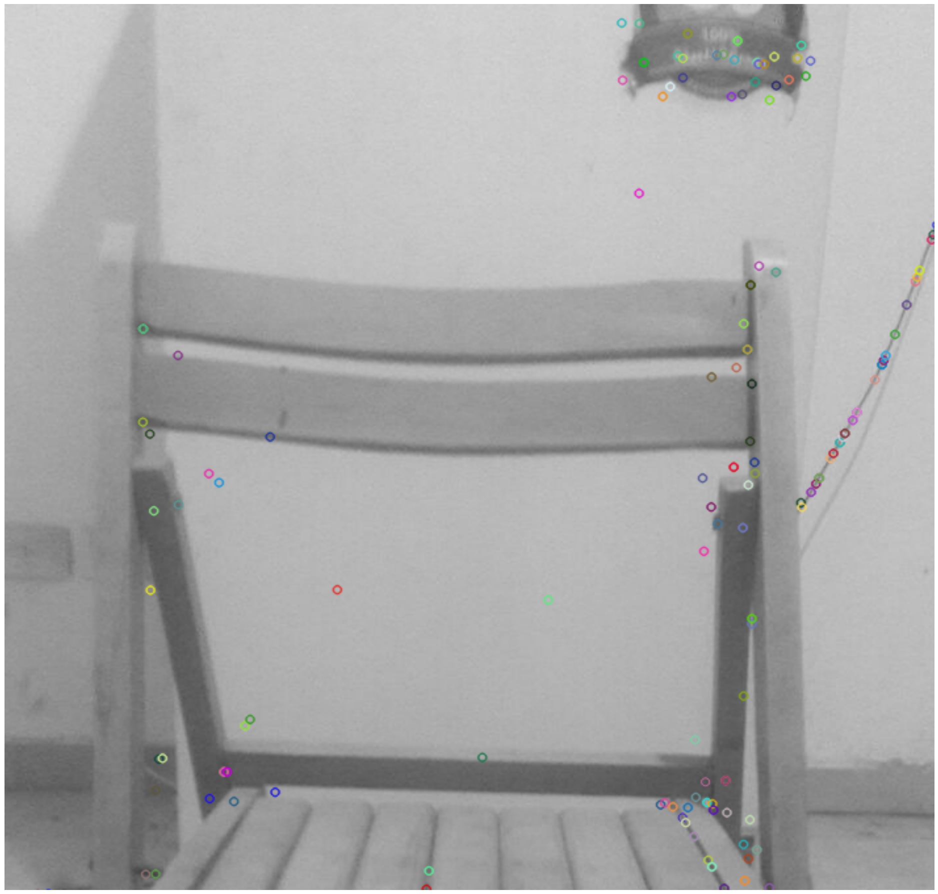
- **Then we used sift_create.detect and compute to calculate keypoints and descriptors.**
- **We used flannbasedmatcher then we applied knnmatch to find the matches**
- **After that we applied the ratio test to detect the uncertainties**
- **We found x and y coordinates, then we improve the matches by calculating the epipolar constraint=0 but we find a threshold between 0.1 and -0.1**
- **After that we used the image points after applying the epipolar constraints in triangulatepoints, and we plot the point in a 3d view.**
- **We compared the length after the reconstruction with the length between the main corners in the object, and we found that they are near each in some lengths.**

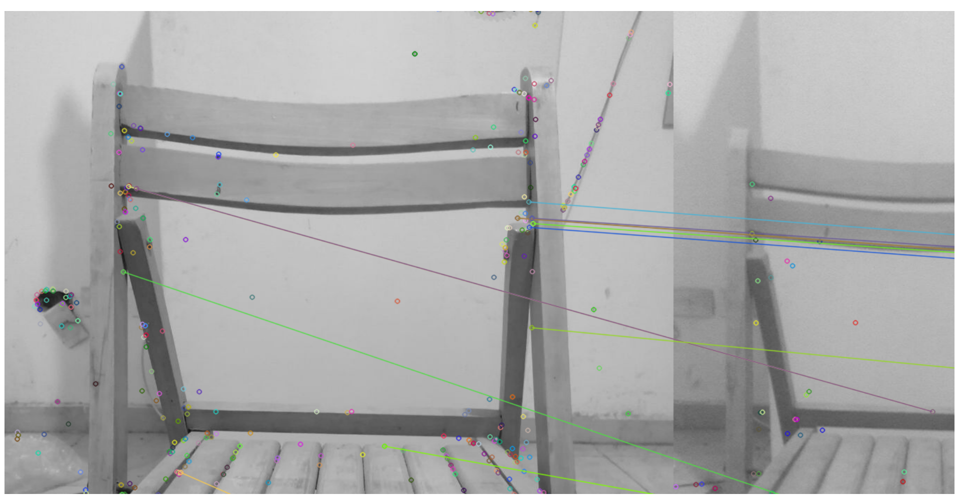
At last, we recommend using a better camera than ours to help better with the matching and the calibration.

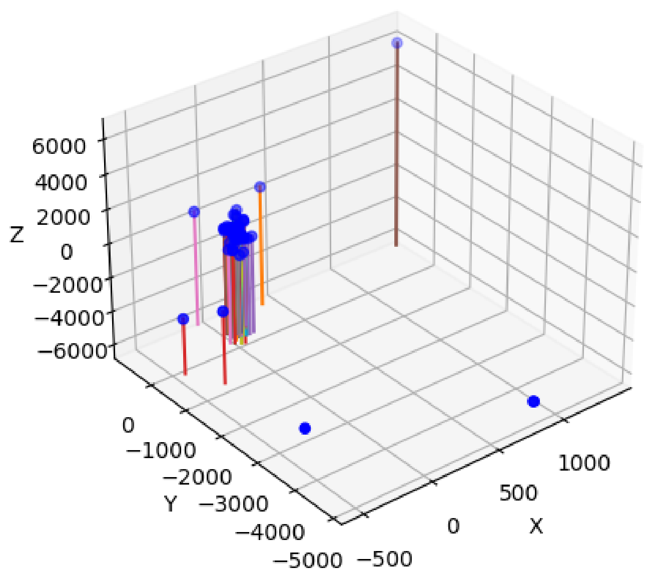
```python
def part_3 (folderName,E,f1,f2,M2,M3) :
    print("part3 start....  ")
    path='./'+folderName+'/*.jpg'
   # path='./hh/*.jpg'
    images = glob.glob(path)
    i=0

    for fname in images:
        img = cv2.imread(fname)
        # convert to greyscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        sift = cv2.SIFT_create()
        keypoints, descriptors = sift.detectAndCompute(img, None)
        #print(np.array(keypoints).shape)




        #print(int(str(keypoints[0][1]),16))
        if i==0:
            k1=keypoints
            d1=descriptors
            img1=img
            i+=1
        else:

     sift_image = cv2.drawKeypoints(gray, keypoints, img)
     # show the image
     cv2.imshow('image', sift_image)
     # save the image
    # cv2.imwrite("table-sift.jpg", sift_image)
     cv2.waitKey(0)
     cv2.destroyAllWindows()
         #sift detect

#sift detect
  bf = cv2.BFMatcher()
  # FLANN parameters
  FLANN_INDEX_KDTREE = 0
  index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
  search_params = dict(checks=50)

  flann = cv2.FlannBasedMatcher(index_params,search_params)
  matches = flann.knnMatch(d1,d2,k=2)
  #matches = bf.knnMatch(d1,d2,k=2)
```

```python
m1=np.linalg.inv(f1)
m2=np.linalg.inv(f2)
img1_points=[]
img2_points=[]
good = []
for m ,n in matches:
    pts1=np.zeros((1,3))
    pts2=np.zeros((1,3))
    (x1, y1) = k2[m.trainIdx].pt
    (x2, y2) = k1[m.queryIdx].pt
    img1_points.append([x1,y1])
    img2_points.append([x2,y2])
   # print(k2[m.trainIdx].pt)
    pts2=np.array([x2,y2,1])

    #pts2.append((f2))
    pts1=np.array([x1,y1,1])
    ##Normalizing two points
    v2=np.matmul(m2,pts1)
    v1=np.matmul(m1,pts2)
    #print(pts2)
    ##Eppolar cnstrains
    con=np.matmul(v2.T,E)
    consrain=np.matmul(con,v1)
    print("const",consrain)
    if (consrain>-.1 and consrain<.1):
```

```python
img1_points=np.array(img1_points).reshape(-1,2)

img2_points=np.array(img2_points).reshape(-1,2)
img3 = cv2.drawMatchesKnn(img1,k1,img,k2,good,None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
cv2.imshow('image', img3)
# save the image
# cv2.imwrite("table-sift.jpg", sift_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

##Reconstruction

points=cv2.triangulatePoints(M3,M2,img1_points.T,img2_points.T).T
#cv2.triangulatePoints(img2_points,M3,points_3d_2)
points = points[:, :3] / points[:, 3:]

plot3dview2(points)
```







Z
6000
4000
2000
0
−2000
−4000
−6000
Y
0
−1000
−2000
−3000
−4000
−5000
X
−500
0
500
1000