Mobile Robotics Assignment 3

Team ID 24

Roll No. 2019702002 Nivedita Rufus

Roll No. 2019702008 Sravya Vardhani S

Q1) Stereo dense reconstruction

The task given at hand is to generate a 3D point cloud from the given stereo images from their corresponding disparity maps.

- **Generation of the disparity map:**

Approach:

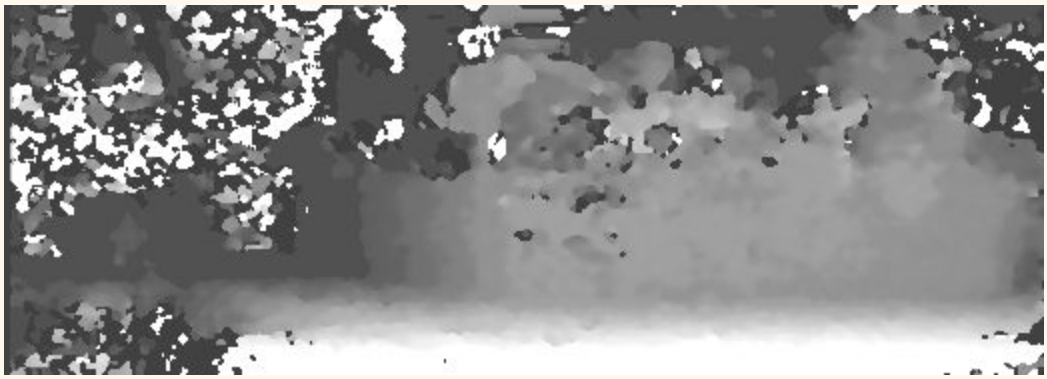
1. Calculate the disparity between the images, using the function **cv2.StereoSGBM_create()**.
2. Assign proper values to the parameters of the function to obtain smoother disparity maps by intuitive guessing.

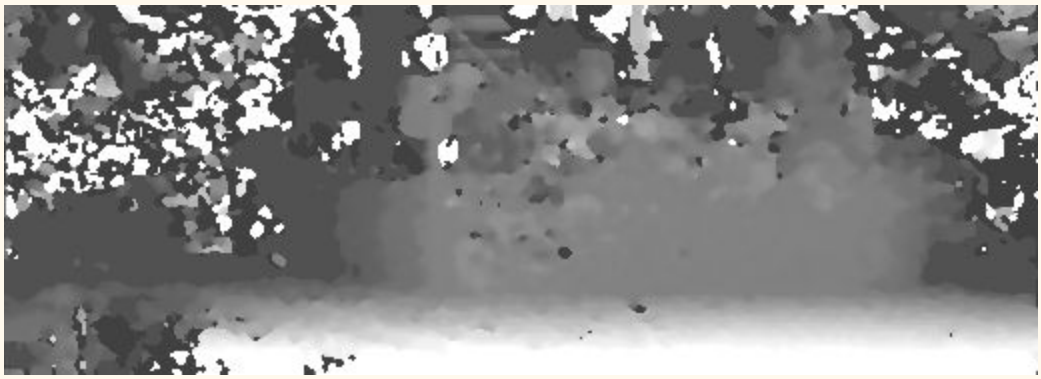
Code snippet:

```
stereo = cv2.StereoSGBM_create(minDisparity= 16,numDisparities = 80,blockSize = 5, uniquenessRatio = 10, speckleWindowSize = 1, speckleRange = 32, disp12MaxDiff = 1, P1 = 8*3*win_size**2, P2 =32*3*win_size**2)

disparity = stereo.compute(l_image,r_image)
```

Example:

Shown below are two of the 21 disparity maps we had obtained.





- **Generation of 3D points :**

The way to obtain the 3D points from the stereo images is given by this formula for a stereo normal case:

$$\begin{bmatrix} U \\ V \\ W \\ T \end{bmatrix} = \begin{bmatrix} B & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & Bc & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \\ p_x \end{bmatrix}$$

Where,

B = baseline distance

P_x = parallax/disparity in x

c = focal length of the camera

Approach:

1. Compute the the 3D points using the formula mentioned above .
2. For each of the pixels, record the colors of those pixels to have a color encode 3D point cloud.

Assign proper values to the parameters of the function to obtain smoother disparity maps by intuitive guessing.

Code:

```
q = np.array([[baseline, 0, 0, 0],[0, baseline, 0, 0],[0, 0, baseline*f,
0],[0, 0, 0, -1]])
      for j in range(disparity.shape[0]):
            for k in range(disparity.shape[1]):
                  temp =
np.array([k-(shape[1]/2),j-(shape[0]/2),1,disparity[j,k]])
                  temp1.append([k-(shape[1]/2),j-(shape[0]/2)])
                  temp = np.reshape(temp,(4,1))
                  temp = np.matmul(q,temp)
                  camera_pts.append(temp)
```

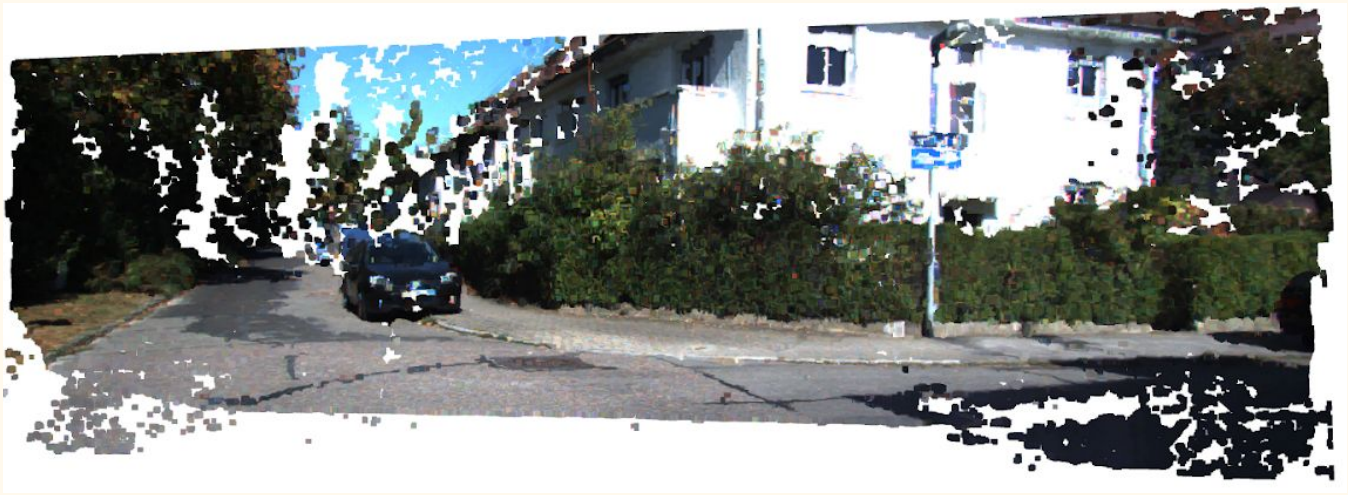
Visualising the point cloud

We remove certain invalid values of disparity and record the color information for each point. This is then visualised using open 3D.

```
mask_map = disparity > disparity.min()
      colors = cv2.cvtColor(l_image, cv2.COLOR_BGR2RGB)
      output_colors = colors[mask_map]
      mask_map = mask_map.ravel()
      output_points = camera_pts[mask_map]
      temp1 = temp1[mask_map]
```

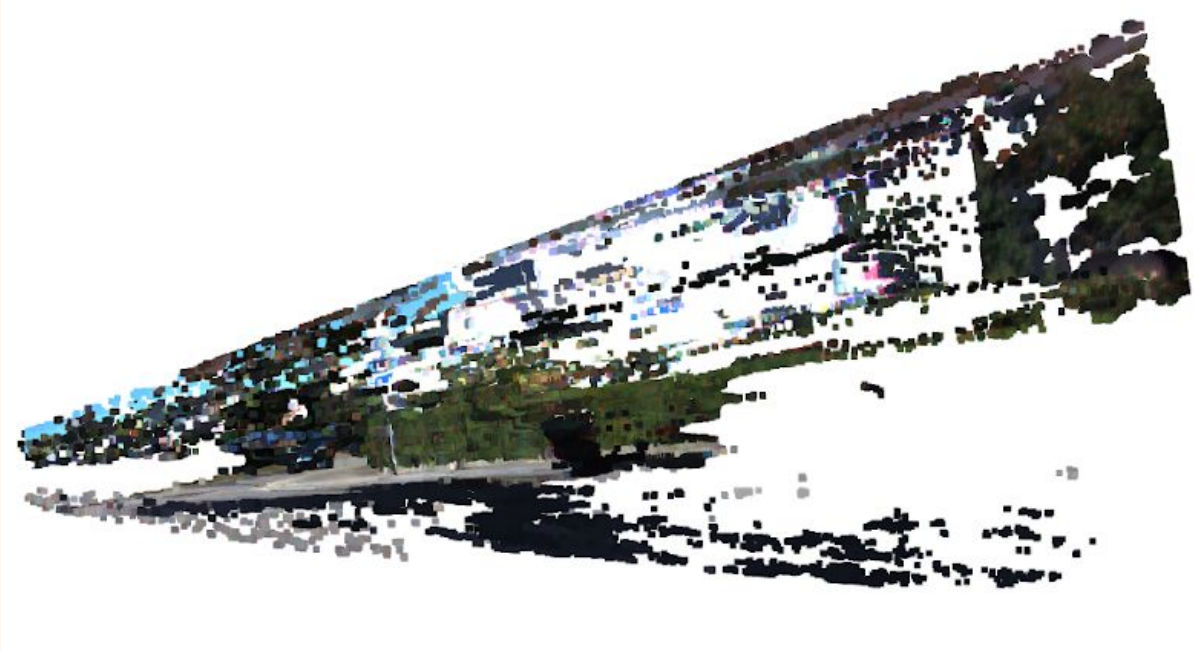
Example:

Shown below are the multiple views of the point cloud generated for a pair of stereo images in camera points. (First pair of images).

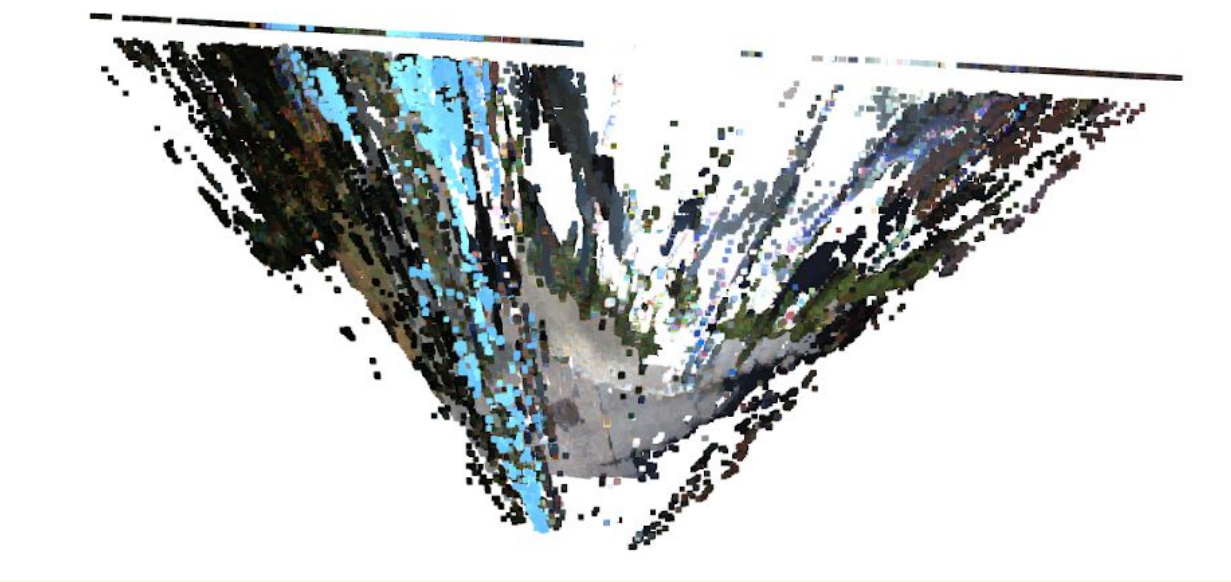


Front view of the point cloud

Slightly rotated front view of the point cloud



Side view of the point cloud



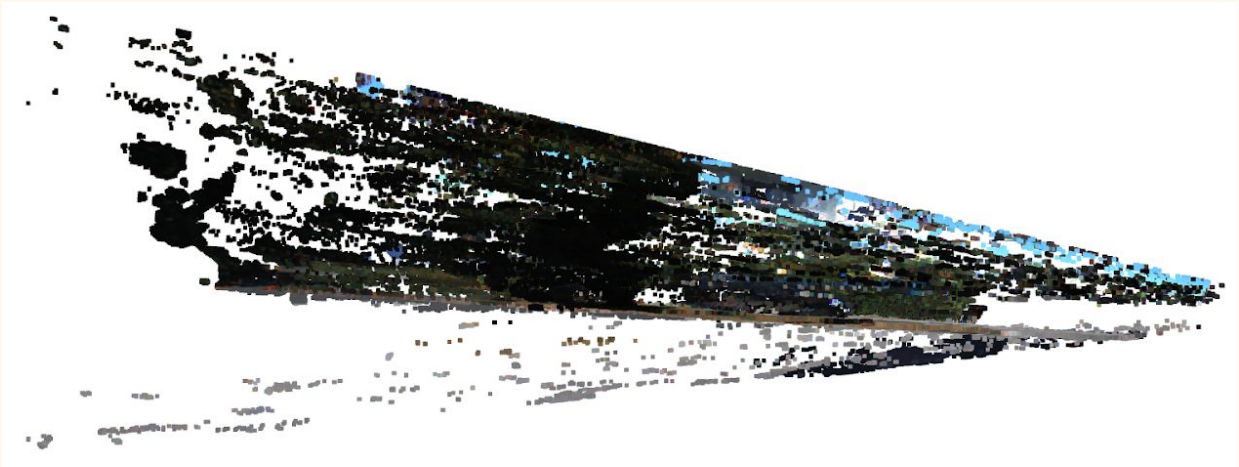
Top view of the point cloud

Example:

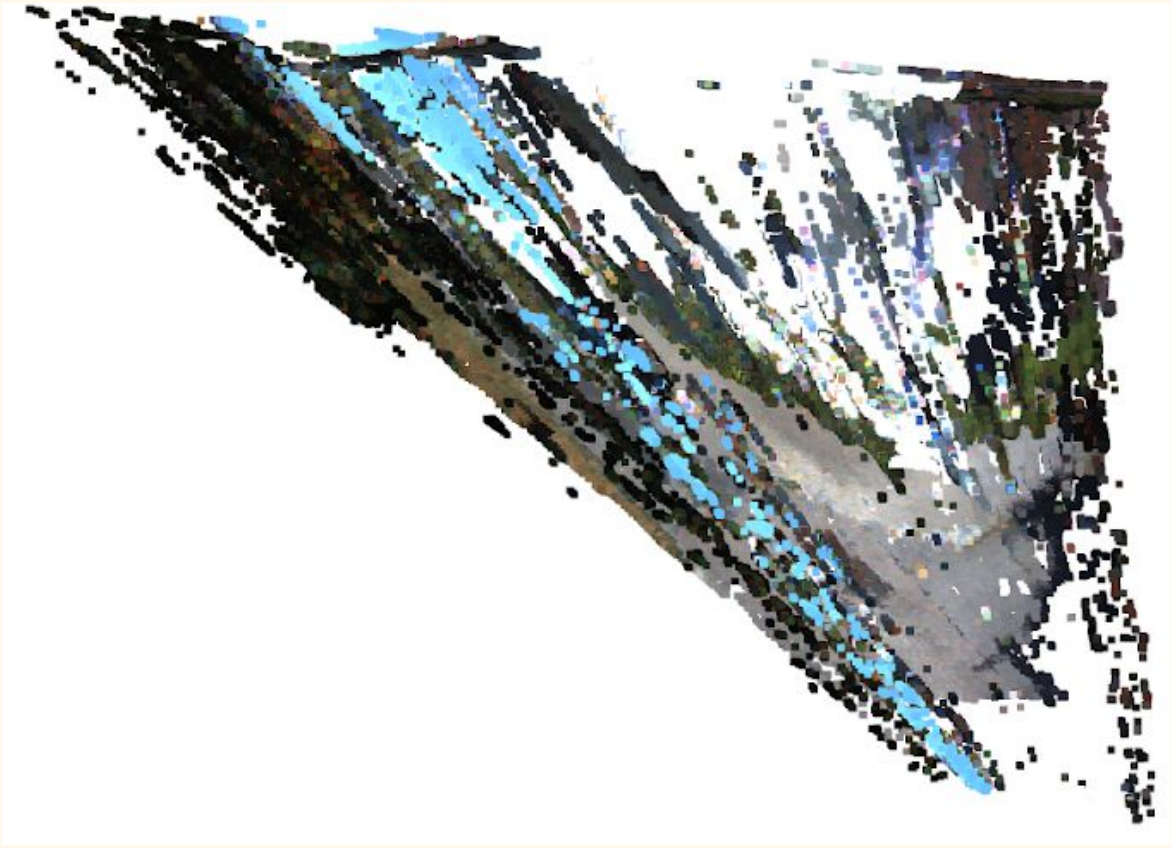
- **Registration of the 3D point cloud the world frame :**

Shown below are the multiple views of the point cloud generated for a pair of stereo images in world coordinate points,(First pair of images).

Front view of the point cloud



Side view of the point cloud



Top view of the point cloud

Approach:

1. To obtain the 3D point cloud in the world frame, we make use of the ground truth poses give to us.
2. Using the ground truth poses given to us for transforming the image into the world coordinate system, we obtained the transformed points by multiplication with the corresponding transformation matrix.

   Issues faced:

   We had issues while transforming the coordinates because of the scale factor of the metrics obtained disparity and given translation leading to wrong projection in the

world coordinate system. We tried to rectify that by dividing by an optimum scale factor which was guessed by by trial and error.

Code:

```
def get_ground_truth():
    file = 'poses.txt'
    return np.genfromtxt(file, delimiter=' ',dtype=None)

def get_transformation_matrix(T34):
    M = np.empty((4, 4))
    M[:3, :3] = T34[:3, :3]
    M[:3, 3] = T34[:3, 3]
    M[3, :] = [0, 0, 0, 1]
    return M
T= get_transformation_matrix(ground_truth[i].reshape(3,4))
camera_pts = np.dot(T,camera_pts.T)
```

Q2) Motion estimation using iterative PnP

The Perspective-n-Point problem's goal is to estimate the position and orientation of a camera for the given 3D to 2D point correspondences, where 3D point is the world point and 2D is its image projection.

R and T matrix :

The R and T matrix correspond to the rotation and translation is called pose of the camera. It has 6 degrees of freedom 3 rotations and 3 translations, therefore it is ideal to get at least 3 pairs of correspondences to solve for the pose. Although 3 pairs of points are enough to compute R and T, it is always better to use more points to refine our matrix.

Procedure to find pose:

Step 1 : Obtain correspondances

Obtain 2D 3D correspondances from the previous question and import the data from the "*.ply*" file which contains information about both the three 3D coordinate and the RGB values of that particular coordinate. The corresponding two points are stored in a text file as well.

```
def get_camera_points(plydata,image_no):

    ply_cam = plydata.elements[0].data
    Xc = np.zeros(ply_cam.shape[0])
    Yc = np.zeros(ply_cam.shape[0])
    Zc = np.zeros(ply_cam.shape[0])

    for i in range(ply_cam.shape[0]):
        Xc[i] = ply_cam[i][0]
        Yc[i] = ply_cam[i][1]
        Zc[i] = ply_cam[i][2]
    Xc = Xc.reshape([ply_cam.shape[0],1])
    Yc = Yc.reshape([ply_cam.shape[0],1])
    Zc = Zc.reshape([ply_cam.shape[0],1])

    threeD = np.concatenate((Xc,Yc),axis=1)
    threeD = np.concatenate((threeD,Zc),axis=1)
    np.savetxt("Cam3D"+str(image_no)+".txt", threeD, delimiter=' ')
```

Step 2 : Calculate total reprojection error

The difference between the image point and its projected point. Ideally this difference should be zero, but in practice there is an error which is called as reprojection error and it is defined as follows.

$$r = \sum_i ||xi - PXi||^2$$

Code:

```
def rx(ptxy,ptXYZ,P):
    temp = np.dot(P,ptXYZ)
    rx = (ptxy[0] - temp[0])/temp[2]
    return rx

def ry(ptxy,ptXYZ,P):
    temp = np.dot(P,ptXYZ)
    ry = (ptxy[1] - temp[1])/temp[2]
    return ry

def error(ptxy,ptXYZ,P):
    r = []
    rdash = 0
    r.append(rx(ptxy,ptXYZ,P))
    r.append(ry(ptxy,ptXYZ,P))
    r = np.asarray(r)
    r = r.reshape([2,1])
    for i in range(r.shape[0]):
        rdash += r[i]**2
    return rdash

def tot_error(xy,XYZ,P):
    rdas  = 0
    for i in range(10000):
        ptxy = xy[i]
        ptXYZ = XYZ[i]
        rdas += error(ptxy,ptXYZ,P)
    return rdas
# Squared sum of return value of tot_error is the final value.
```

We can use this to analyse the stopping point or the number of points after which the loss starts increase, so that we can adjust out number of epochs.

Step 3 : Solving for pose using Gauss- Newton method

1. The algorithm starts with an intuitive initialisation of the projection matrix P and setting a tolerance limit which we set as 10^{-3}

2. The next step was to calculate the residue for the randomly selected 2D-3D correspondences(we had used 10,000 correspondences) and the current P(projection matrix)
3. We then calculated the Jacobian matrix,

$$\mathbf{J(r_j) = \partial r_j / \partial P}$$

$$\textbf{Where, } r = \sum_i ||xi - PXi||^2$$

$$P=[[p_{11}, p_{12}, p_{13}, p_{14}],$$

$$[p_{11}, p_{12}, p_{13}, p_{14}],$$

$$[p_{11}, p_{12}, p_{13}, p_{14}]]$$

4. Calculation of gradient is done by,

$$\mathbf{\nabla G = J^T(r)}$$

5. Now the new projection matrix is updated by,

$$\mathbf{\delta = (J^T J + \lambda I)^{-1} J^T(r)}$$

$$\mathbf{P_{new} = P_{old} - (\delta)}$$

(we use λ to maintain the non-singularity of the matrix)

Code: Jacobian is defined as follows:

```
def j1(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j1 = 2*ptXYZ[0]*((jn/jd)-ptxy[0])/jd
    return j1
def j2(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j2 = 2*ptXYZ[1]*((jn/jd)-ptxy[0])/jd
    return j2
def j3(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
```

```
    j3 = 2*ptXYZ[2]*((jn/jd)-ptxy[0])/jd
    return j3
def j4(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j4 = 2*((jn/jd)-ptxy[0])/jd
    return j4
def j5(ptxy,ptXYZ,P):
    j5 =0
    return j5
def j6(ptxy,ptXYZ,P):
    j6 = 0
    return j6
def j7(ptxy,ptXYZ,P):
    j7 = 0
    return j7
def j8(ptxy,ptXYZ,P):
    j8 =0
    return j8
def j9(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j9 = -2*ptXYZ[0]*((jn/jd)-ptxy[0])*(jn/((jd)**2))
    return j9
def j10(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j10 =-2*ptXYZ[1]*((jn/jd)-ptxy[0])*(jn/((jd)**2))
    return j10
def j11(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j11 =-2*ptXYZ[1]*((jn/jd)-ptxy[0])*(jn/((jd)**2))
    return j11
def j12(ptxy,ptXYZ,P):
    jn = sum(np.multiply(P[0], ptXYZ))
    jd = sum(np.multiply(P[2], ptXYZ))
    j12 =-2*((jn/jd)-ptxy[0])*(jn/((jd)**2))
    return j12

def g5(ptxy,ptXYZ,P):
```

```
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g5 = 2*ptXYZ[0]*((gn/gd)-ptxy[1])/gd
    return g5
def g6(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g6 = 2*ptXYZ[1]*((gn/gd)-ptxy[1])/gd
    return g6
def g7(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g7 = 2*ptXYZ[2]*((gn/gd)-ptxy[1])/gd
    return g7
def g8(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g8 = 2*((gn/gd)-ptxy[1])/gd
    return g8
def g1(ptxy,ptXYZ,P):
    g1 =0
    return g1
def g2(ptxy,ptXYZ,P):
    g2 = 0
    return g2
def g3(ptxy,ptXYZ,P):
    g3 = 0
    return g3
def g4(ptxy,ptXYZ,P):
    g4 =0
    return g4
def g9(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g9 = -2*ptXYZ[0]*((gn/gd)-ptxy[1])*(gn/((gd)**2))
    return g9
def g10(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g10 =-2*ptXYZ[1]*((gn/gd)-ptxy[1])*(gn/((gd)**2))
    return g10
```

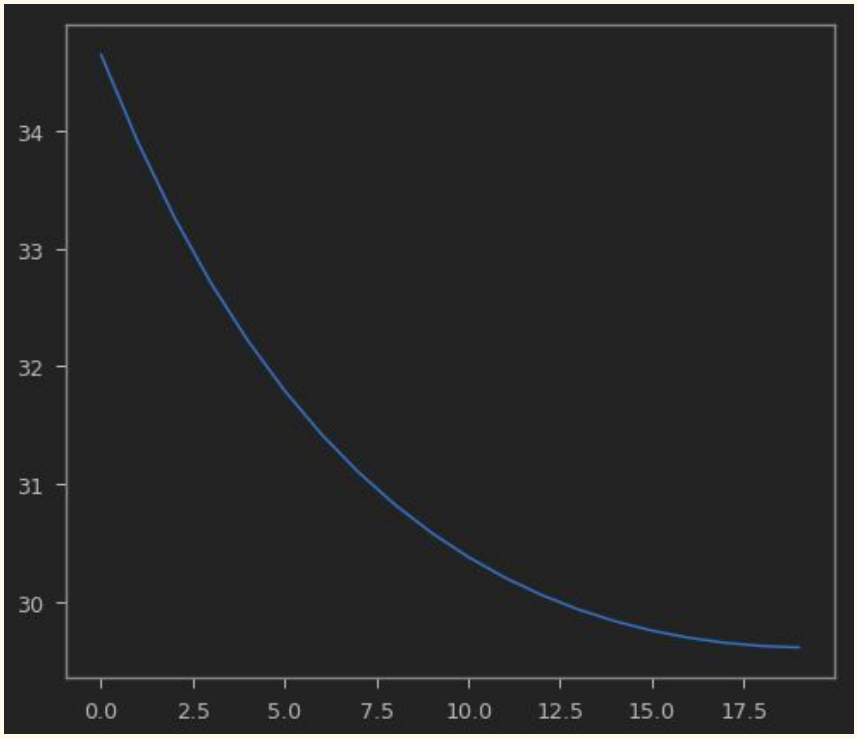
```
def g11(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g11 = -2*ptXYZ[1]*((gn/gd)-ptxy[1])*(gn/((gd)**2))
    return g11
def g12(ptxy,ptXYZ,P):
    gn = sum(np.multiply(P[1], ptXYZ))
    gd = sum(np.multiply(P[2], ptXYZ))
    g12 = -2*((gn/gd)-ptxy[1])*(gn/((gd)**2))
    return g12



def Jack(xy,XYZ,P):
    J = []

        for i in range(10000):
        ptxy = xy[i]
        ptXYZ = XYZ[i]
        J.append(j1(ptxy,ptXYZ,P))
        J.append(j2(ptxy,ptXYZ,P))
        J.append(j3(ptxy,ptXYZ,P))
        J.append(j4(ptxy,ptXYZ,P))
        J.append(j5(ptxy,ptXYZ,P))
        J.append(j6(ptxy,ptXYZ,P))
        J.append(j7(ptxy,ptXYZ,P))
        J.append(j8(ptxy,ptXYZ,P))
        J.append(j9(ptxy,ptXYZ,P))
        J.append(j10(ptxy,ptXYZ,P))
        J.append(j11(ptxy,ptXYZ,P))
        J.append(j12(ptxy,ptXYZ,P))
        J.append(g1(ptxy,ptXYZ,P))
        J.append(g2(ptxy,ptXYZ,P))
        J.append(g3(ptxy,ptXYZ,P))
        J.append(g4(ptxy,ptXYZ,P))
        J.append(g5(ptxy,ptXYZ,P))
        J.append(g6(ptxy,ptXYZ,P))
        J.append(g7(ptxy,ptXYZ,P))
        J.append(g8(ptxy,ptXYZ,P))
        J.append(g9(ptxy,ptXYZ,P))
        J.append(g10(ptxy,ptXYZ,P))
```

```
        J.append(g11(ptxy,ptXYZ,P))
        J.append(g12(ptxy,ptXYZ,P))


    J = np.asarray(J)
    J= J.reshape([20000,12])
    print(J.shape)
    return J

Jack(xy,XYZ,P)
```

We update P using the gradient as shown below,

```
def updated_P(xy,XYZ,P,zeta):
    J = Jack(xy,XYZ,P)
    R = getR(xy,XYZ,P)
    grad = np.matmul(J.T,R)
    norm = np.matmul(grad.T,grad)

    if (norm>zeta):
        dot = np.dot(J.T,J)
        temp = inv(dot+d4*np.eye(12))
        Sk = -np.dot(temp,grad)
        P = P.reshape([12,1])
        Pnew = P - Sk
        Pnew = Pnew.reshape([3,4])
        P = Pnew
    return P
```

Convergence of the reprojection error can be shown in the following plot v/s no of iteration



Reprojection Error Vs Number of Iterations

It can be observed that the algorithm converges after 20 iterations. Hence max iterations can be chosen around 20.

Step 4 : decomposition of P into R,t:

1. **Obtaining R from P:**

   The value of R is obtained using the formula given below:

$$R = K^{-1}P_{t3} \qquad P_{t3} : \text{First three columns of P.}$$

$$R_{+} = UV^{T} \text{ where } UDV^{T} = R$$

2. **Obtaining t from P:**

   The value of t is given by:

$$t = K^{-1}P_4 / \sigma_1 \quad \text{where } D = \text{diag}(\sigma_1, \sigma_2, \sigma_3)$$
$$P = K[R_+ \quad t]$$

Code:

```
P = P.reshape([3,4])
K = np.array([[7.070912e+02, 0.000000e+00, 6.018873e+02], [0.000000e+00,
7.070912e+02, 1.831104e+02], [0.000000e+00, 0.000000e+00, 1.000000e+00]])
Pdash = P[:,0:3]
print(Pdash)
Kinv = inv(K)

R = np.matmul(Kinv,Pdash)
print(R)
u, s, vh = np.linalg.svd(R)
print(u.shape,vh.shape)
Rdash = np.dot(u,vh)
print(s)
t = np.matmul(Kinv,P[:,3])/s[0]
print('RT',np.matmul(Kinv,P))
print("Rdash",Rdash)
print("T",t)
```

Case 1

Rt1 = [[0, -1, 0, 0],[1, 0, 0, 0],[0, 0,1, 0]]

Obtained result from Gauss Newton optimization

```
('Rdash', array([[ 0.02565167, -0.99827111, -0.05288471],
       [ 0.99923224,  0.02717161, -0.02822464],
       [ 0.02961281, -0.05212009,  0.99820167]]))
('T', array([13.58830941,  0.0272596 , -0.23296   ]))
```

Case 2

Rt2 = [[0.5, -0.866, 0, 0],[0.866, 0.5, 0, 0],[0, 0, 1, 0]]

Obtained result from Gauss Newton optimization

```
('Rdash', array([[ 0.49552562, -0.86656944, -0.05926013],
       [ 0.85197394,  0.49819417, -0.16106821],
```

```
       [ 0.16909985,  0.02932534,  0.98516256]]))
('T', array([-18.12433606,   9.80396228,   7.18330792]))
```

Verification of images:

We have verified the Gauss Newton optimisation by reprojecting the 3D points onto the image as shown in case 1,2 respectively by using,

$$x_i = PX_i$$

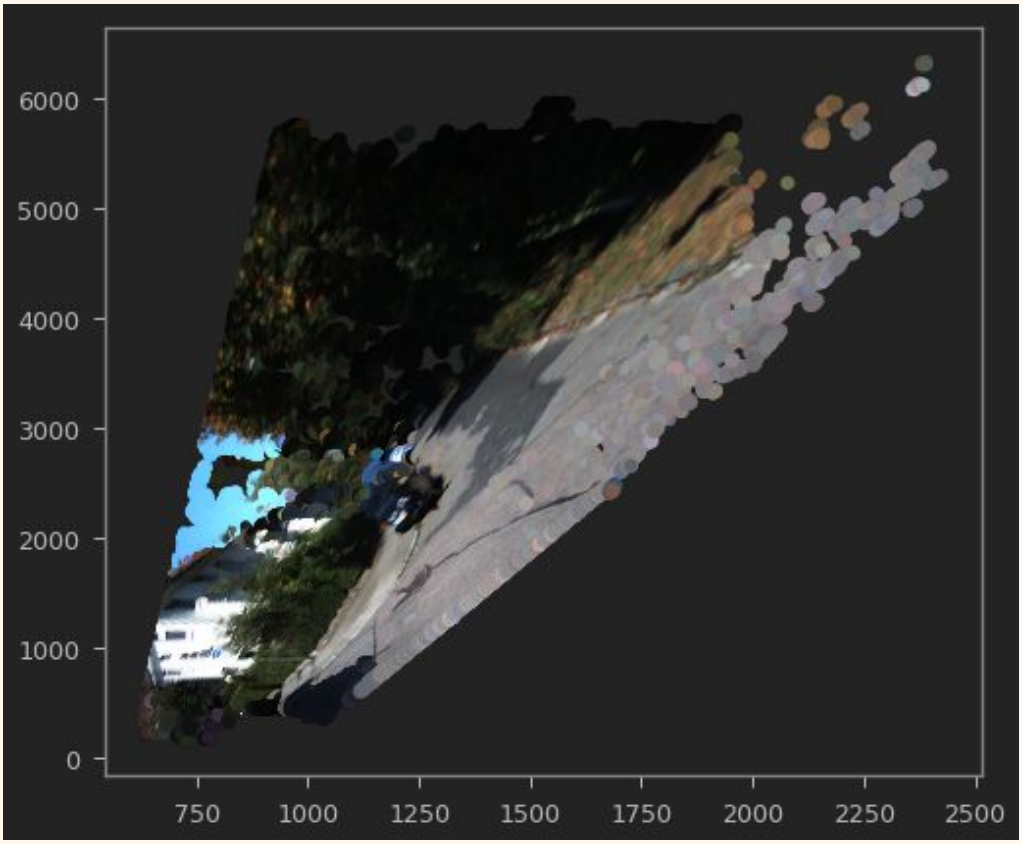
Where x_i is 2D coordinates and X_i is the 3D world coordinates.

Code:

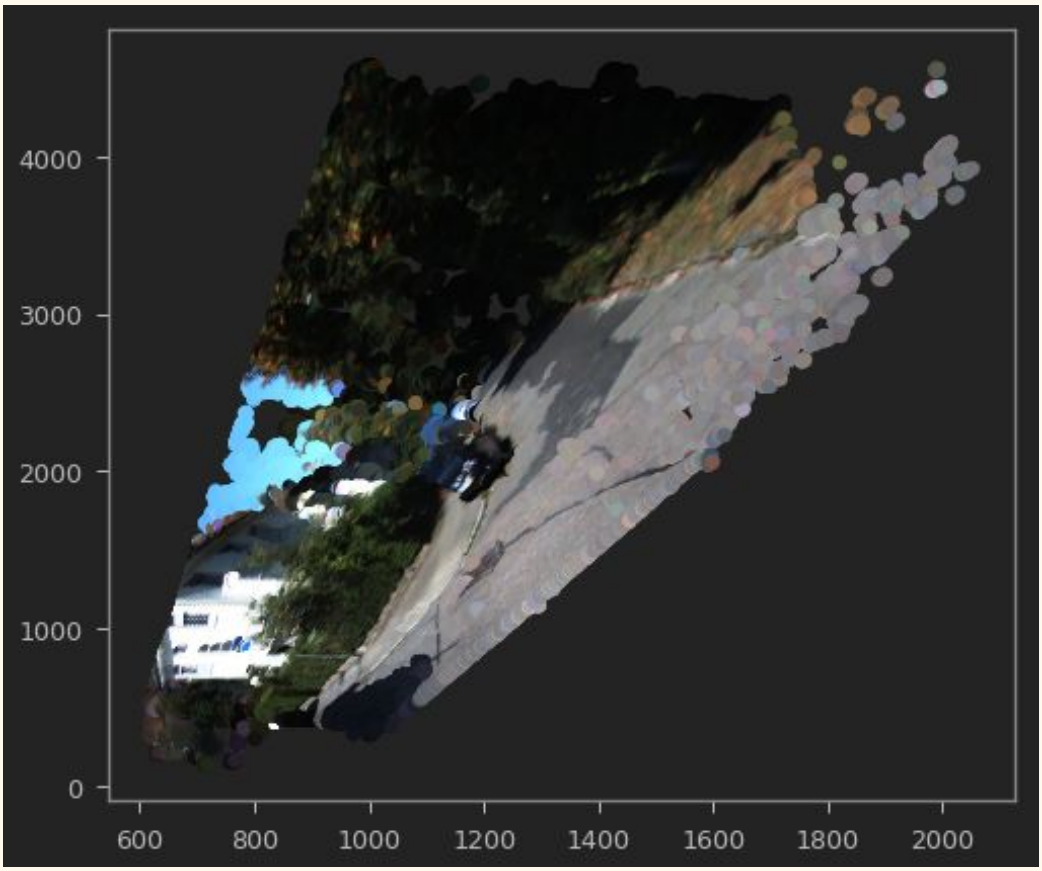
```
P1 = np.matmul(K,np.array([[0,-1,0,0],[1,0,0,0],[0,0,1,0]]))
t = t.reshape([3,1])
P21 = np.hstack((Rdash,t))
P2 = np.matmul(K,P21)
#image_no = 0
cam_points = np.loadtxt("Cam3D"+str(image_no)+".txt", dtype=float)
print(cam_points.shape)

one = np.ones([cam_points.shape[0],1])
XYZPt = np.hstack((cam_points,one))
for i in range(cam_points.shape[0]):
    x1pt[i] = np.dot(P1[0],XYZPt[i])/np.dot(P1[2],XYZPt[i])
    y1pt[i] = np.dot(P1[1],XYZPt[i])/np.dot(P1[2],XYZPt[i])
    x2pt[i] = np.dot(P2[0],XYZPt[i])/np.dot(P2[2],XYZPt[i])
    y2pt[i] = np.dot(P2[1],XYZPt[i])/np.dot(P2[2],XYZPt[i])
for i in range(ply_cam.shape[0]):
        R[i] = ply_cam[i][3]
        G[i] = ply_cam[i][4]
        B[i] = ply_cam[i][5]

RG = np.hstack((R,G))
RGB = np.hstack((RG,B))
plt.scatter(x1pt,y1pt,c = RGB/255.0)
```

The images obtained are shown below.

Case 1: Rotation by 90 degree

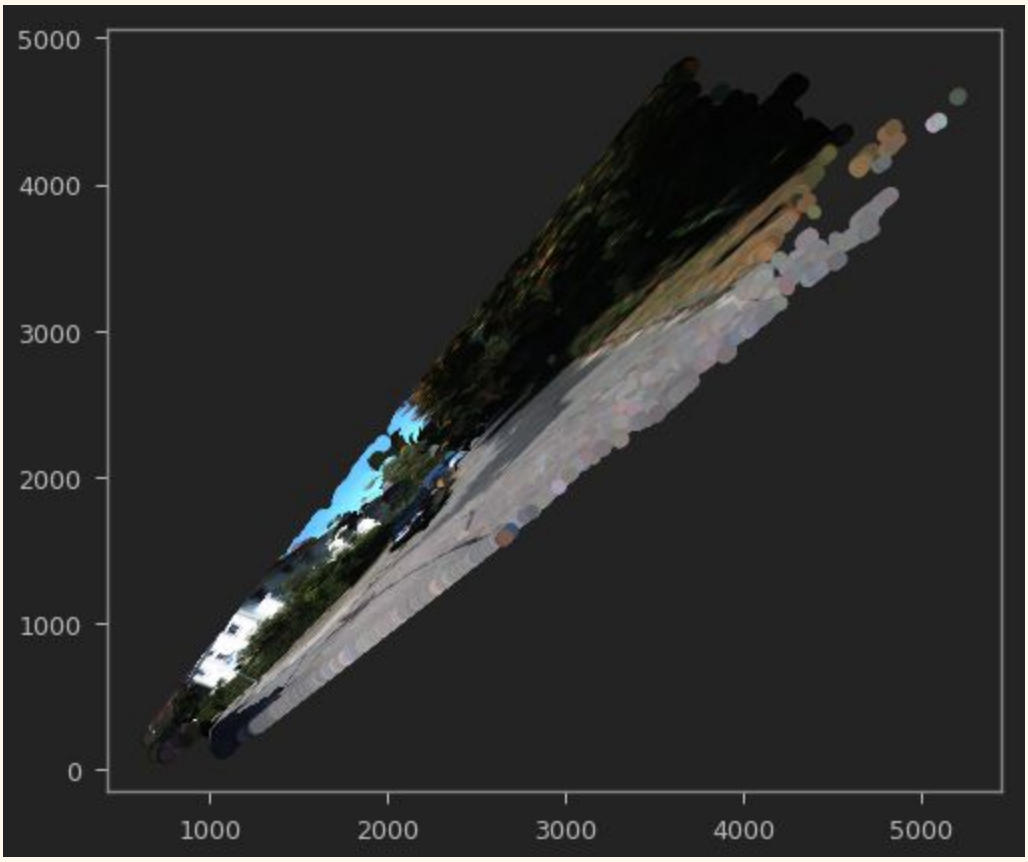


Projection with actual matrix

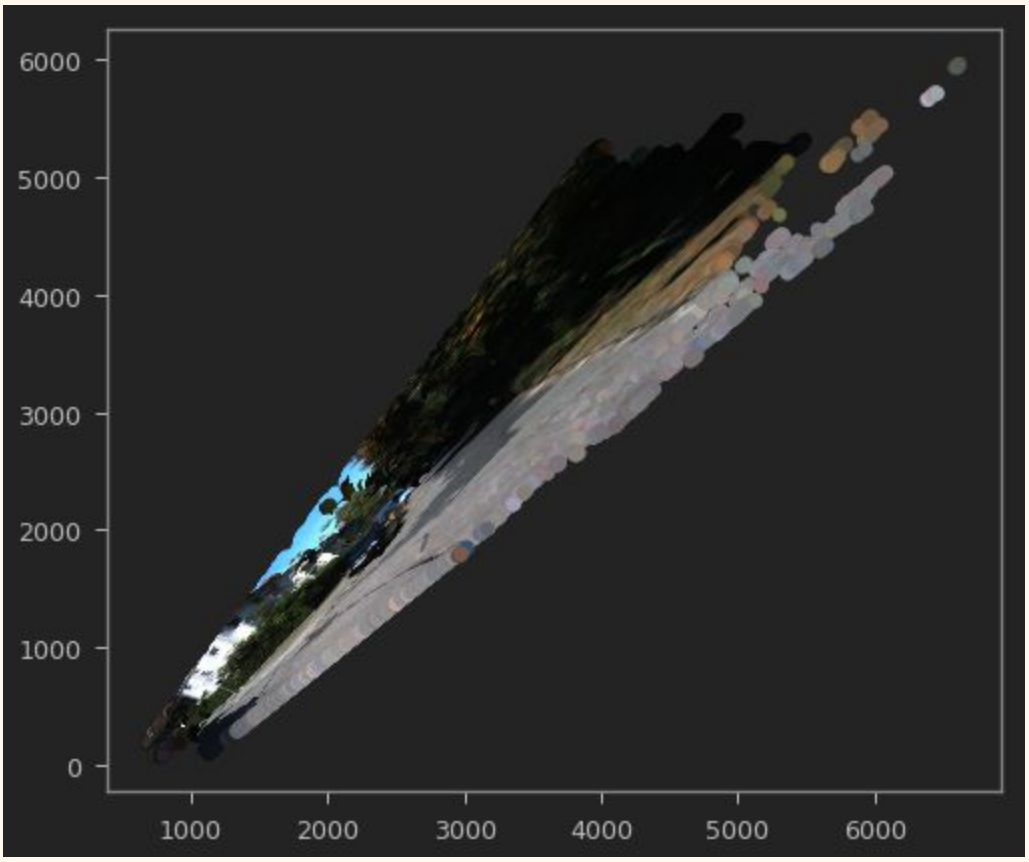


Projection using matrix obtained by GN optimisation

Case 2: Rotation by 30 degrees



Projection with actual matrix



Projection using matrix obtained by GN optimisation

Observations using different initializations

It was observed that different initializations converge to almost the same value but not necessarily in the same number of iterations. Three observations are shown below,

Actual T	Initialization	Obtained T
[[0.5, -0.866, 0, 0], [0.866, 0.5, 0, 0], [0, 0, 1, 0]]	[[0.5,-0.866,0.1,0.1], [0.866,0.5,0.1,0.1], [0.1,0.1,0.8,0.1]]	[[0.49552562, -0.86656944, -0.05926013,-18.12433606,], [0.85197394, 0.49819417, -0.16106821, 9.80396228], [0.16909985, 0.02932534, 0.98516256,7.18330792]]
[[0.5, -0.866, 0, 0],	[[0.5,-0.9,0.2,0.1], [0.6,0.5,0.1,0.4],	[[0.47910058, -0.9230605 , -0.0651311,-75.82254463,],

[0.866, 0.5, 0, 0], **[0, 0, 1, 0]]**	**[0.2,0.1,0.8,0.3]]**	**[0.91028823, 0.48465268, -0.15302825,19.35048808],** **[0.16630718, -0.00127497, 0.98607317,5.40863358]]**
[[0.5, -0.866, 0, 0], **[0.866, 0.5, 0, 0],** **[0, 0, 1, 0]]**	**[[0.6,-0.6,0.1,0.3],** **[0.7,0.5,0.3,0.4],** **[0.1,0.1,0.8,0.2]]**	**[[0.48425375, 0.91861678, -0.09215345,30.10035515],** **[0.91496847, -0.46559287, 0.17080557,-12.36931945,],** **[-0.12321422, 0.14995019, 0.98098583, 0.9341321]]**

We observed that we get different translations, owing to the scale ambiguity issue.

References

[1] Xiao Xin LU, A Review of Solutions for Perspective-n-Point Problem in Camera Pose Estimation https://iopscience.iop.org/article/10.1088/1742-6596/1087/5/052009/pdf

[2] Wen Huey Lai, Sie Long Kek and Kim Gaik Tay, Solving Nonlinear Least Squares Problem Using Gauss-Newton Method.

[3] Cyrill Stachniss - slides

Roles:

Since it was a complicated assignment, it was done and debugged together completely. So there is no distinct role division.