CS/RBE 549 Computer Vision : Fall 2019
Project Report

Feature based 3D - 2D Visual Odometry

Team Members

Baladhurgesh Balagurusamy Paramasivan
Madhan Suresh Babu
Nikhil Jonnavithula
Sabhari Natarajan

Abstract

In this ever developing world of technology, mobile robotics have gained huge popularity. In the near future we would see a fleet of driverless cars on the roads. One of the important aspects of mobile robotics, especially the autonomous ones is its ability to estimate its own motion i.e. odometry. Using imaging devices such as cameras to estimate the motion has been greatly researched on as it uses simple sensors and limited resources. This kind of odometry is termed as Visual Odometry

In this project we implement feature based Stereo Visual Odometry (VO). This algorithm works based on tracking features between Frame at Time 'T' and 'T+1'. As, we are using a stereo camera, the 3D world coordinates of the features can be calculated. The coordinates of these features are used to find the pose of Frame 'T+1' wr.t Frame 'T'. The pose is incrementally calculated every time instant as combined to build the final odometry of the mobile robot.

An improvised algorithm was also tested to improve the accuracy of the visual odometry. The Stereo VO algorithm is tested on the widely used KITTI dataset (project by Karlsruhe Institute of Technology and Toyota Technological Institute). Using semantic segmentation for improving Stereo VO has been discussed briefly in the project. The Stereo VO generated using this algorithm are also shown in comparison to the ground truth provided in the KITTI dataset.

Table of Contents

List of Figures

1. Introduction

The most traditional method to get odometry is to use wheel encoders. This is a simple method, but these can only be used in ground vehicles. Even in ground vehicles, these provide inaccurate readings whenever there is wheel slippage due to muddy or loose soil kind of terrains. These inaccuracies build up over time and the odometry estimate drifts proportionally to the distance travelled. Another method commonly used for odometry is the GPS. This performs well in conditions where the signals can be received. For cases like indoor, underwater navigation where signals cannot be received this fails to help.

VO comes to the rescue here. Research suggests that VO are more accurate than wheel encoders (relative position error 0.1 - 2 %) as these are not affected by wheel slips. VO can be used to complement other sensors like Inertial Measurement Units (IMUs), SONAR, LIDAR. In regions with lack of GPS, VO can play a crucial role.

1.1 Problem Definition and Objective

VO is the process of estimating the egomotion of an agent by examining the changes that motion induces on the images of its single or multiple cameras. It is divided into 2 types: Monocular VO and Stereo VO. As the name suggests, Monocular VO is the process of calculating odometry of vehicle using one camera and Stereo VO is by using two cameras. The advantage of using Stereo VO is that we can calculate the scale of the odometry.

Fig. 1.1 Monocular VO Setup (Left) and Stereo VO Setup (Right)

Our objective in this project is to “Determine 6-D pose of the camera by using a stream of stereo image sequences”.

1.2 Assumptions

Before proceeding with Visual Odometry process, few assumptions were considered:

- Sufficient Illumination on the environment, to detect and track features.
- Dominance of static scene over moving objects, so that we calculate odometry with static frame of reference.
- Enough texture to allow apparent motion to be extracted, there should be enough features to detect in image, we cannot find enough texture on infinity corridor with the same color, or scene with full of snow.
- There should be sufficient overlap between subsequent frames so that we will be able to track features from 1 frame to another.

Fig. 1.2 Scenarios not ideal for VO

2. Methodology

2.1 Stereo VO Pipeline

Following flowchart shows the various steps followed by us for implementing the Stereo Visual Odometry. Each of these sections is explained in detail in the upcoming sections.

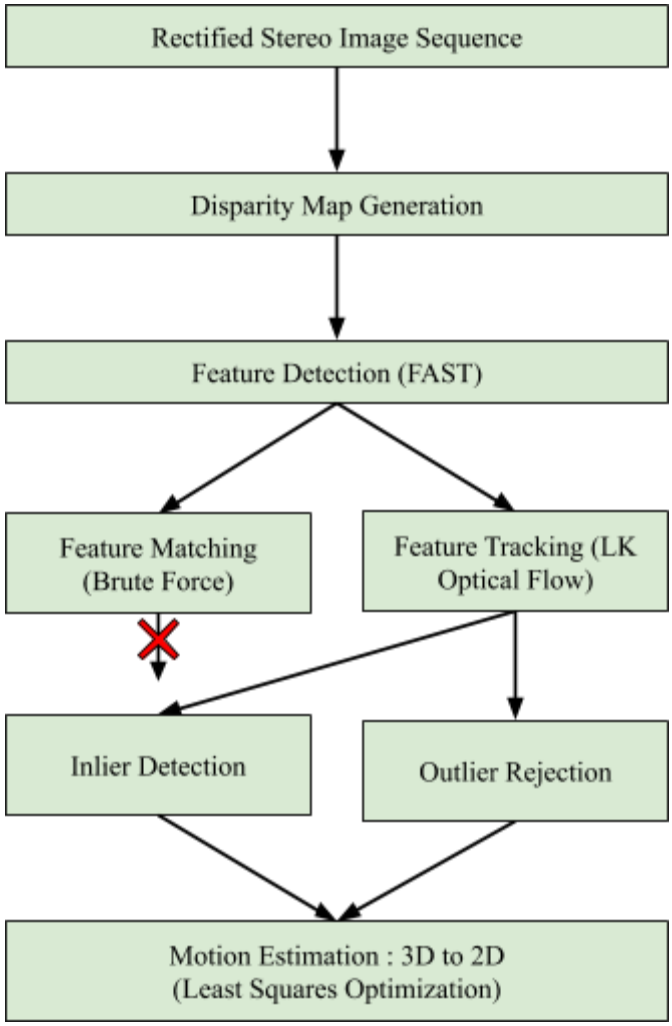

Fig. 2.1 Stereo VO Pipeline

2.2 Disparity Map Generation

For a point closer to the camera, the difference between the pixel coordinates in Left and Right frame will be higher than for a point far away from the camera as shown in Figure 2.1. This difference is known as disparity. In an undistorted and rectified set of images this difference in pixel coordinates will be only along the 'x' direction.

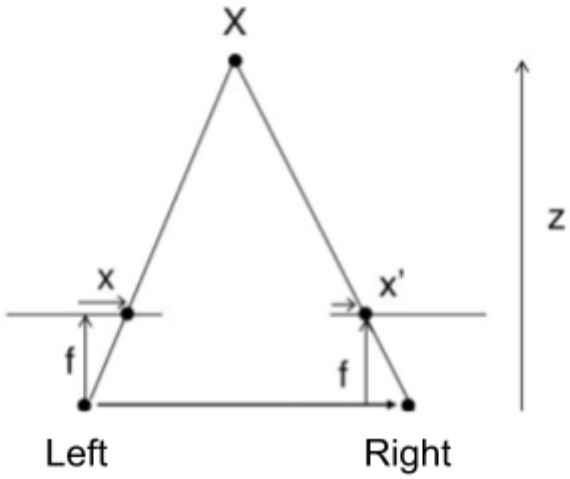

Fig. 2.2 Projection of 3D point in Image

Hence, we used the Block Matching Algorithm with a window size of 15x15 to find the disparity map at a given time 'T'. This algorithm takes a window from left image, slides (only along 'x') it onto the right image. It computes the Sum of Absolute Differences (SAD) for each disparity value and returns the value with least SAD. The left camera is used as the reference.

Further, we build a disparity mask from the disparity map, which will be used during the feature detection process. At higher depths stereo degenerates to a monocular case i.e. loses its depth perception. So, using features at higher depths will lead to inaccurate results. Therefore, we use a depth constraint of 3m to 25m to build the mask. Also, we trim small amount of pixel from all four sides, because, either these sections of image will not have good features or the same feature will not be available in the next frame. Figure 2.3 shows the disparity map and disparity mask.

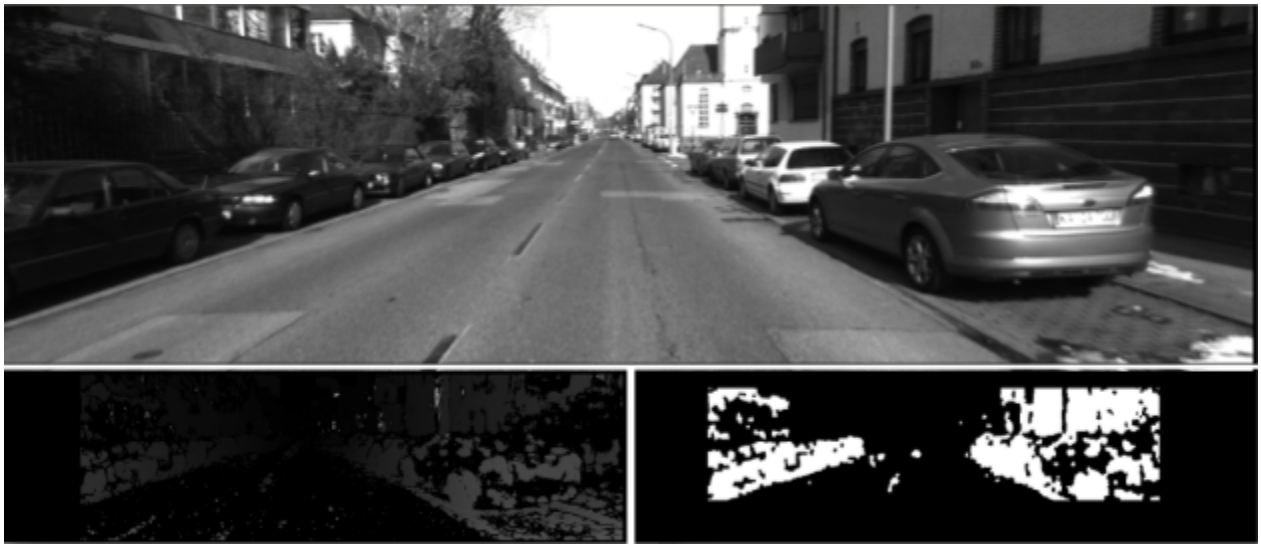

Fig 2.3 (Clockwise from top) Input Left Image, Disparity Mask, Disparity Map

2.3 Feature Detection

In order to perform VO we need to identify key points or features in each frame so that they could be used as a reference for the motion estimation of the robot. To identify the features in each image we used 'Features from accelerated segment test' (FAST) algorithm. FAST is a corner detection algorithm.

FAST features identified ~8000 features in our road scene images. We only require the features for which we can find the 3D coordinate and also doesn't have larger depths. This is where the disparity mask found in previous step is used. This gives ~1400 features. Here the features are dense in some regions and scarce on other. Using features very close to each other will not give us accurate results.

So we add a step called feature binning, which divides the image into grids (20x20 windows) for feature detection, and select only one best feature from each bin. This resulted in ~200 evenly spread out features in each frame.

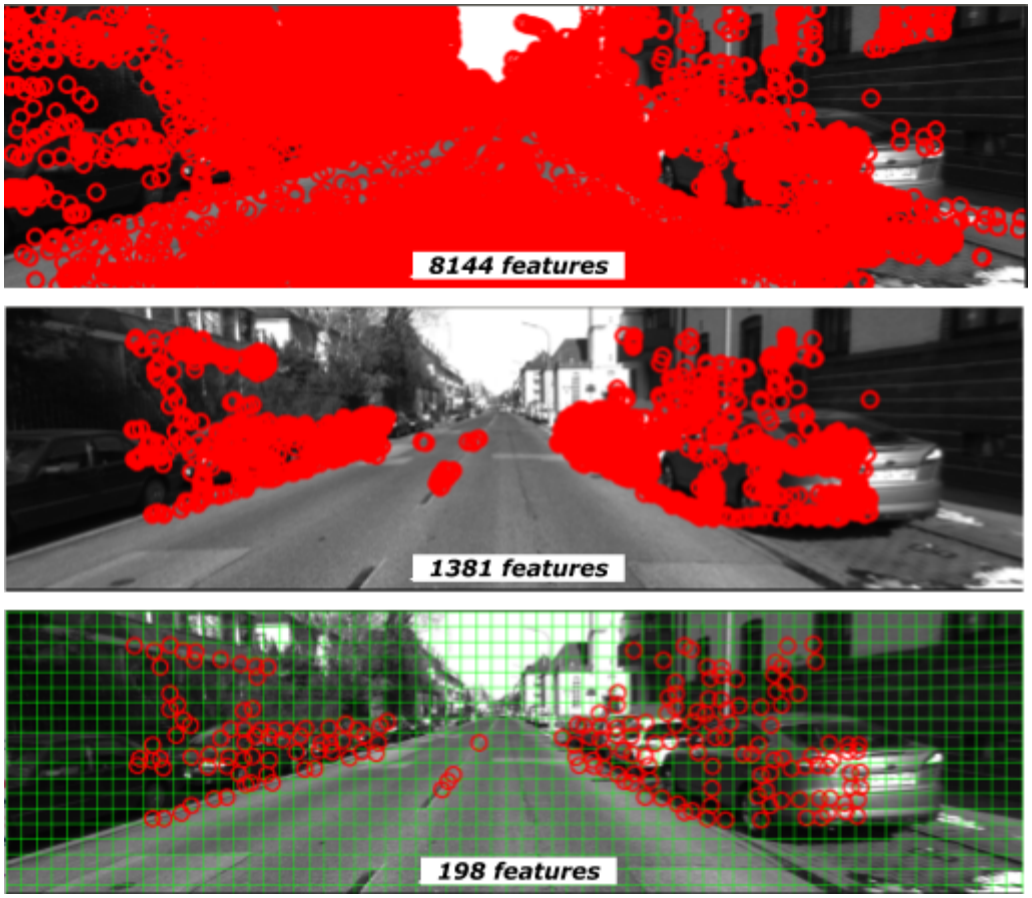

Fig. 2.4 FAST features detected : Full Image, Adding Disparity Mask, Adding Feature Binning

2.4 Feature Matching and Tracking

2.4.1 Feature Matching

Feature Matching as the name suggests is the process where features in two different image are matched, in this case between Frame at Time ‘T’ and ‘T+1’. For this the features in both the frames are found suing the previous method. To find matches, we used the Brute Force matcher that which takes the descriptor of one feature in time ‘T’ and is matched with all other features time ‘T+1’ using some distance calculation and the closest one is returned. This feature matching method was computationally very expensive due to comparison between every pair of features, hence was not used for correlating the features.

2.4.2 Feature Tracking

Another method to co-relate the features, is feature tracking. Here, we find features in time ‘T’ and it is tracked onto time ‘T+1’ using the Lucas–Kanade Optical Flow. This is much faster than feature matching as number of computations are only equal to the number of features found. This method assumes that the flow is essentially constant in a local neighbourhood of the pixel/feature under consideration, and solves the basic optical flow equations for all the pixels in that neighbourhood, by the least squares criterion. It is also less sensitive to image noise than pointwise methods. Feature tracking is done only on the left frame. From the ~200 features tracked ~40 best features tracked are selected for further processing.

Fig. 2.5 Features tracked from Frame ‘T’ to ‘T+1’

2.5 Determining Static Features

In a typical road scene there are both static or stationary objects and dynamic objects. The odometry of the vehicle would be accurate only if we calculate its translation and rotation with respect to a set of stationary points as this is equivalent to finding the motion of the robot with respect to a stationary world frame. Hence, we extract a set of stationary point from the given set of features that were tracked and use it for motion estimation of the robot.

The basic principle used here is that for a pair of static points, the euclidean distance (d) between then at "t" should be the same as that of at "t+1". If d is not same, then it is either:

1. An error in 3D coordinate estimation of at least one of the two features.
2. One of the two features is dynamic.

We performed this step in two different methods :

1. Inlier Detection : Selection best features
2. Outlier Rejection : Rejecting bad features

2.5.1 Inlier Detection

This method makes use of graph theory and is popularly known as the Maximum Clique problem. In this method the idea is to determine the set of points that are all stationary with respect to each other. The Inlier Detection methods involves the following steps :

2.5.1.1 Calculating the Pairwise Absolute Distance Error Matrix :

	f_1	f_2	f_3	...	f_n
f_1	0	e_{12}	e_{13}	...	e_{1n}
f_2	e_{21}	0	e_{23}	...	e_{2n}
f_3	e_{31}	e_{32}	0	...	e_{3n}
⋮	⋮	⋮	⋮	0	⋮
f_n	e_{n1}	e_{n2}	e_{n3}	...	0

Fig 2.6 Pairwise Absolute Distance Error Matrix

For the selected features, 3D coordinates are found using the disparity value and camera parameters (provided by KITTI).

Using the 3D coordinates of the features from time T and T+1 we calculate the distance between every pair of features in both the time instants (d) and use them to find the change in distance between the features over time. These changes in distances are the entries of the Pairwise Absolute Distance Error Matrix , i.e., $e_{12} = | d_{12}^{t} - d_{12}^{t+1}|$.

2.5.1.2 Graph formation and Adjacency matrix calculation :

Next, we form a graph of the features with the number of nodes in the graph equal to the number of keypoints that were tracked. Then the Adjacency matrix of the graph (indicates the connections between nodes in the graph) is calculated by thresholding the Pairwise Absolute distance error matrix, i.e., nodes 1 and 2 are connected if e_{12} in the Pairwise Absolute Distance Error matrix is less than a threshold value. The thresholding operation results in a graph that has the stationary nodes connected to each other. The best feature is the one which has maximum number of nodes connected to it i.e. the row which has maximum values less than the threshold. If there are multiple choices available the first one in the matrix is selected

2.5.1.3 Finding the largest fully connected subgraph :

Since the graph connections indicate stationarity of features with respect to each other, finding the largest subgraph that is fully connected results in a set of points that are all stationary with respect to each other based on the best feature found. The result is also called as the Maximum Clique, where a clique is a fully connected subgraph. The points of the maximum clique are used in the Optimization step for motion estimation of the robot.

2.5.2 Outlier Rejection

The inlier detection method relies heavily on the value of threshold to determine the stationarity of the nodes. Choosing a high value for the threshold would lead to incorrect results and choosing a very small value might not find stationary points at all. So in order to avoid the dependence on an arbitrary value of threshold to identify stationary points, we performed the Outlier detection method, which directly uses the Pairwise Absolute Distance Error matrix instead of thresholding.

Here we eliminate the features that are the most dynamic or non-stationary. We find the most dynamic feature by finding the total sum of each row in the Pairwise Absolute Distance Error matrix. The row with the largest value (largest error) would be the most dynamic point. We perform elimination of the most dynamic point and remove the feature's row and column in the Pairwise Absolute Distance Error matrix. This elimination process of the most dynamic point is performed until we get to the required number of points for the Optimization step.

2.6 Optimization for Motion Estimation

Using the static set of features from the Inlier detection or outlier rejection, we feed it to the optimization routine. In the optimization routine, we use Levenberg-Marquardt nonlinear least squares minimization for finding the translation and rotation matrix. We are using 3D-2D motion estimation, so our Objective function looks like,

$$\epsilon = \sum_{\mathcal{F}^t, \mathcal{F}^{t+1}} (\mathbf{j_t} - \mathbf{PTw_{t+1}})^2 + (\mathbf{j_{t+1}} - \mathbf{PT^{-1}w_t})^2$$

Where,

- P - Projection matrix (From camera parameters)
- T - Homogeneous Transformation Matrix which needs to estimated
- w_t , w_{t+1} - 3D World point at time instant t and t+1
- j_t , j_{t+1} - 2D feature points at time instant t and t+1

Here the objective function is minimizing the reprojection error. For a feature, TW_{t+1} gives the transformed 3D world point at time 'T', using projection matrix (P) we reproject the 3D point to 2D image frame.The error between the 2D feature point and the reprojected 2D point is calculated. In the same way, the world point at 'T' is reprojected to 2D image frame at time 'T+1'. We are using 10 features for this minimization process.

The Levenberg-Marquardt algorithm combines two minimization methods: the gradient descent method and the Gauss-Newton method. In the gradient descent method, the sum of the squared errors is reduced by updating the parameters in the steepest-descent direction. In the Gauss-Newton method, the sum of the squared errors is reduced by assuming the least squares function is locally quadratic, and finding the minimum of the quadratic. The Levenberg-Marquardt method acts more like a gradient-descent method when the parameters are far from their optimal value, and acts more like the Gauss-Newton method when the parameters are close to their optimal value. This algorithm allows us to find the optimal 6D pose vector in a quicker way as it uses the advantages of both, gradient descent and Gauss- Newton method.

2.7 Improvement in Inlier Detection

As discussed earlier, the Inlier detection method depends on the threshold value selected which can make or break the VO. A large value of threshold would incorrectly classify the dynamic point as a static point. Hence for a more robust method of estimation of the stationary features, we developed a variant of the Inlier detection method that starts with a small value for threshold and makes small increments to the threshold value until we obtain a feature that is static w.r.t ~10 other features. This reduces the probability of classifying dynamic features as static features.

Optical Flow tracking does not perform well when the features in image have moved larger distances. Features at lesser depths move large distances in the image as compared to distant features. So, tracking the features at lesser depths resulted in incorrect tracking. To solve this ,we updated the lower threshold for disparity mask from 3m to 10m. The number of feature

points after feature binning reduced but there were sufficient points for tracking. If the number of features went too low, the lower threshold was reduced dynamically.

This method provided better results compared to the original Inlier detection and outlier rejection method due to identification of a better set stationary points.

2.8 Extended Work : Semantic Segmentation

Estimating the stationary points through the Inlier detection or the Outlier detection works as long as there exists a dominance of static key points over dynamic ones. Our methods fail when the whole view of the cameras are covered by a moving object, for example when a vehicle crosses at an intersection in front of the robot. A better method to identify the stationary points would be to determine the class of each pixel (trees, buildings, other vehicles). When pixels are identified as belonging to certain classes we could then easily conclude that features of the class trees and buildings for example are the stationary points.

Semantic segmentation is the method of classifying each pixel of an image to certain classes. We worked on building the pixel-wise classifier using a popular semantic segmentation architecture called SegNet. It is a deep CNN architecture with an encoder network and a decoder network. The novelty in this architecture is the use of max-pooling indices from the encoders for up-sampling in its corresponding decoder layer. We were able to achieve a global class accuracy of 82.27% .

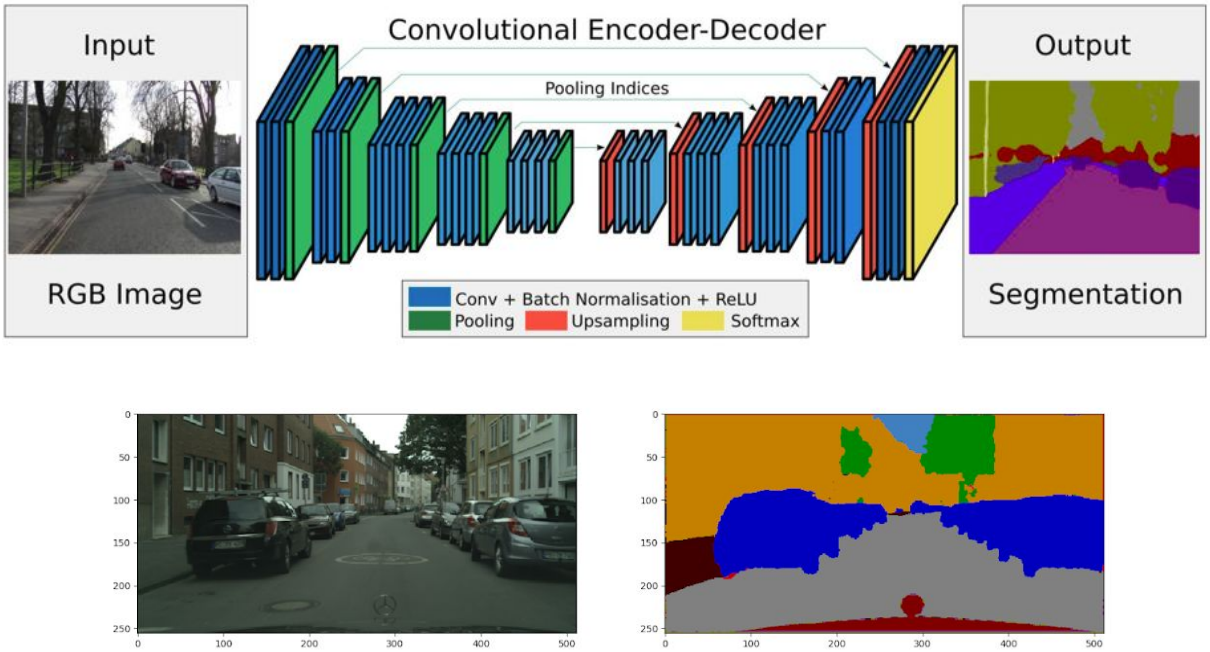

Fig.2.7 SegNet architecture and a test result

We also implemented a modified version of the SegNet that contains *strided 2* convolutional layers instead of the max-pooling layers for performing the down sampling operation, and skip connections from each encoder layer to its corresponding decoder. The modified architecture provided improved segmentation results with a global class accuracy of 86.1%.

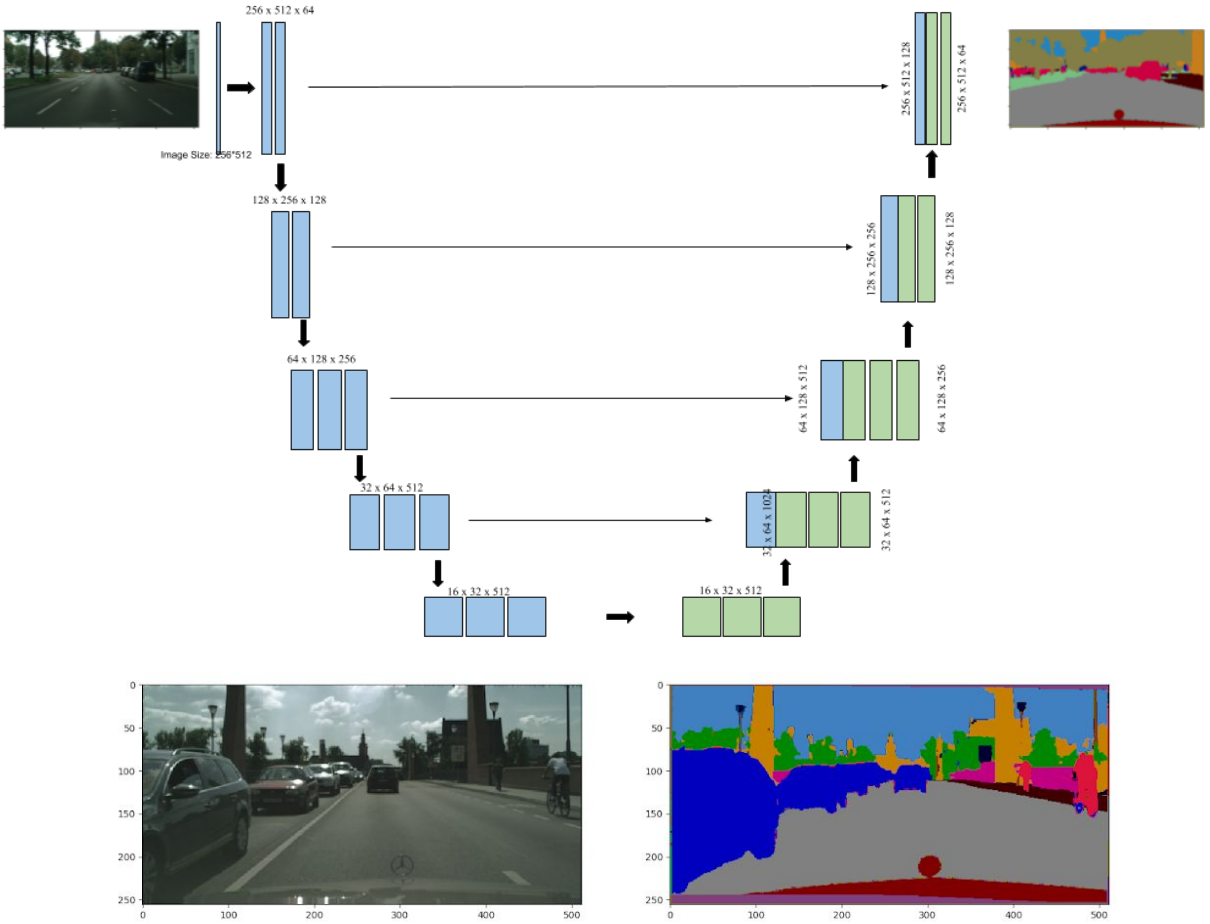

Fig.2.8 Modified SegNet architecture and a test result

3. Results

3.1 Inlier Detection vs. Outlier Rejection

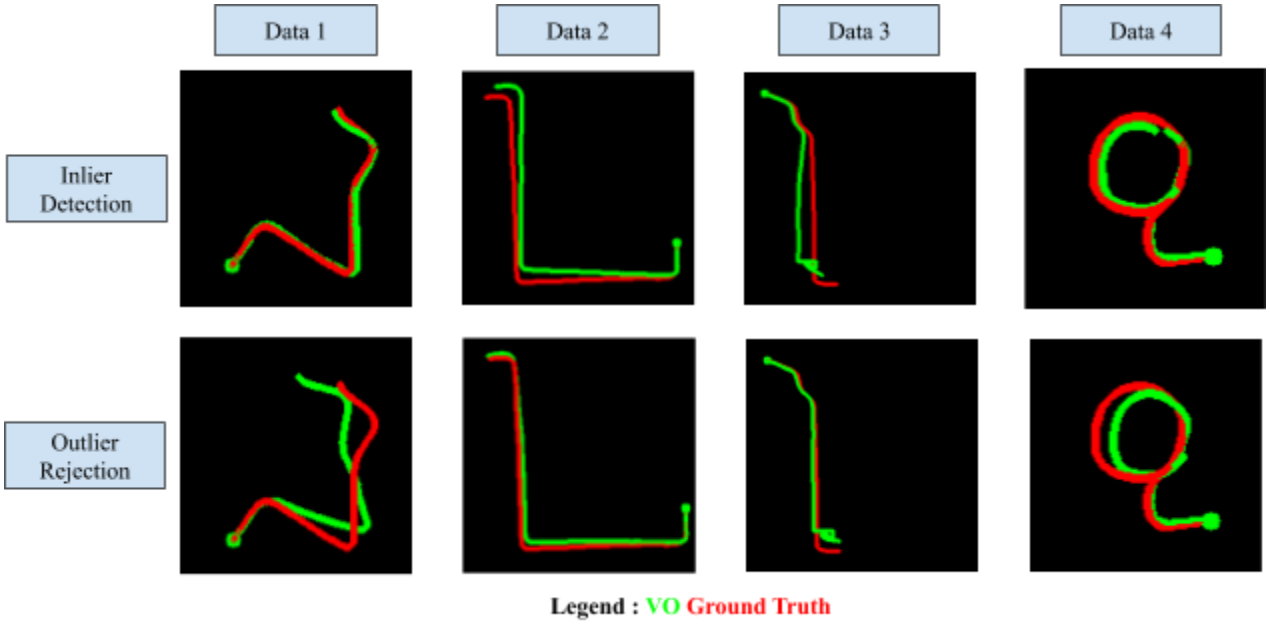

Fig 3.1 Inlier Detection vs. Outlier Rejection Comparison

In data set 1 & 4, we see in this data set, Inlier detection out performs Outlier rejection. It is the first right turn where the Outlier rejection fails and that error is carrier forward throughout. This is because at few instances it had chosen a set of features which belong to a moving vehicle. All the features on the vehicle are static w.r.t each other. When we have less features tracked and more features are from a single dynamic object like a car, this system fails.

In this data set 2 & 3, outlier rejection performs better than Inlier detection, this shows us that both of these methods are not consistently working good on varied datasets. In data set 3 towards the end of VO we see a distortion. This is due to the complete scene being covered by a tram (dynamic object). This was also one of our assumptions that there should be more static features than dynamic ones. When this assumption fails, it leads to incorrect VO results.

To correct this, in case of autonomous cars, we can add a non-holonomic constraints i.e. the dominant direction of motion for the vehicle should be forward.

3.2 Improved Inlier Detection

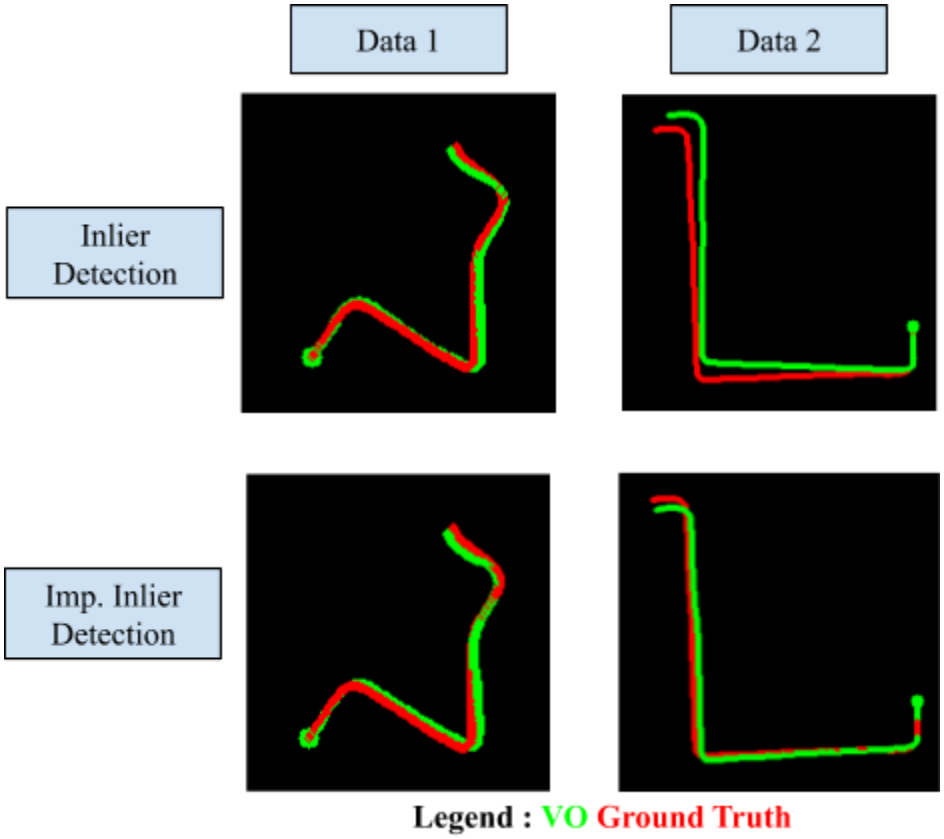

Fig 3.2 Inlier Detection vs. Improved Inlier Detection

Since in our results from Inlier detection and outlier rejection did not work consistently on all the datasets, we tried to improve the Inlier detection method which is explained in Section 2.7. After improvement the algorithm worked better than the previous methods and was also consistently over different data sets.

4. Conclusion

From the methods tested and the results seen we arrived at the following conclusion:

1. Addition of Disparity Mask helped reduce the number of features and combining it with feature binning, it gave us evenly spread out features. Number of features also reduced by 97% (~8000 to ~200), which helped in faster computation time.
2. Feature Tracking is more efficient that Feature Matching in terms of computation time.
3. Inlier Detection and Outlier Rejection gave considerably good performance, but was not consistent over different data sets.
4. The improved Inlier Detection Algorithm performed better than both the previous algorithms and was also consistent over different data sets as it was able to select a better set of static features.

This processing was done on a Core i7 processor and following were the computational time observed for the 3 methods tested:

Computational rate comparison for the VO Algorithms		
Inlier Detection	Outlier Rejection	Improved. Inlier Detection
15 fps	10 fps	14 fps

The computation speed can be further improved if we use GPU for processing.

The further scope for the project include the following:

1. Adding Non-Holonomic constraint
2. Using the Semantic Segmentation results
3. Implementing Bundle Adjustment to reduce drift over time

References

1. KITTI Dataset - http://www.cvlibs.net/datasets/karlsruhe_sequences/
2. Stereo Visual Odometry Blog http://avisingh599.github.io/vision/visual-odometry-full/
3. Visual Odometry Tutorial by Davide Scaramuzza(Univ of Zurich) http://rpg.ifi.uzh.ch/visual_odometry_tutorial.html
4. Segnet - https://arxiv.org/pdf/1511.00561.pdf

Appendices

A1. Python Code SVO - Main

```
#!/usr/bin/env python

import sys
import cv2
import numpy as np
from scipy.spatial import distance
from scipy.optimize import least_squares
import math as m
import time
from HelperFunctions_v4 import *

code_proc_time = time.time()

Data_Num = 7              # 1-7
Data_Num = Data_Num - 1
disparity_type = "BM"     # "BM+WLS","BM"
select_type = "Inlier"    # "Inlier","Outler"
flag_record = 0           # 1-Map only, 2-Full Video
nPts_opt = 10             # Number of points for optimization

flag_display = 1

# Approx number of points to be returned after optical flow
nPts_track = 40
dist_thr = [10,25]        # Disparity Distance Threshold

Data = [ "/Data Set/2009_09_08_drive_0010",
         "/Data Set/2010_03_17_drive_0046",
         "/Data Set/2010_03_05_drive_0023",
         "/Data Set/2010_03_09_drive_0081",
         "/Data Set/2010_03_09_drive_0020",
         "/Data Set/2010_03_04_drive_0033",
         "/Data Set/2010_03_09_drive_0019"]
Data_dir = "/media/sabhari/Data/Educational/WPI/# Fall 2019/RBE 549
Computer Vision/Project" + Data[Data_Num]
Data_Frames = [1424, 967, 370, 341, 546, 399, 372]
Pause = False
```

```
# Reading the calibration file and getting the Projection Matrix
Calib_file = open(Data_dir + "_calib.txt",'r')
Calib_file_lines = Calib_file.readlines()
P1_roi = np.zeros(12)
P2_roi = np.zeros(12)
for j in range(12):
    P1_roi[j] = Calib_file_lines[12].split()[j+1]
    P2_roi[j] = Calib_file_lines[14].split()[j+1]
P1_roi = P1_roi.reshape(3,4)
P2_roi = P2_roi.reshape(3,4)
# Setting the camera parameters
f = P1_roi[0,0]
base = -P2_roi[0,3]/P2_roi[0,0]
cx = P1_roi[0,2]
cy = P1_roi[1,2]

# Read Ground Truth File and initialize Map
GT_file = open(Data_dir + '/insdata.txt','r')
GT_file_lines = GT_file.readlines()
StartGT = [0,0]
CurGT = [0,0]
StartGT[0] = float(GT_file_lines[0].split()[4])
StartGT[1] = float(GT_file_lines[0].split()[5])
Map = np.zeros((20,20,3), np.uint8)
centre = int(Map.shape[0]/2)
cv2.circle(Map,(centre,centre),5,color=(0,255,0), thickness = -1)

currID = 0      # Current Frame ID
prevID = 0      # Previous Frame ID
# Image Variables : Left Frame, Right Frame, Disparity Map, Disparity Mask
curr = [0,0,0,0]
bgr = [0,0,0,0]
prev = [0,0,0,0]

Position = [[1, 0, 0, 0],
            [0, 1, 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, 0]]
# Getting the initial pose of the vehicle
# So that VO can be compared with Ground Truth
CurGT[0] = float(GT_file_lines[1].split()[4])
CurGT[1] = float(GT_file_lines[1].split()[5])
```

```
GT_x = CurGT[0] - StartGT[0]
GT_y = CurGT[1] - StartGT[1]
theta = m.atan2(-GT_x,-GT_y)
IniRot = [[m.cos(theta) , 0, m.sin(theta), 0],
          [0            , 1, 0           , 0],
          [-m.sin(theta), 0, m.cos(theta), 0],
          [0            , 0, 0           , 1]]
Position = np.dot(Position,IniRot)

# Parameter for Save Video and the final Map
SaveName = str(Data_Num+1)+"_"+str(disparity_type)\
           +"_"+str(nPts_opt)+"_"+str(select_type)+"_new"
if flag_record == 2:
   fourcc = cv2.VideoWriter_fourcc('M','J','P','G')
   if Data_Num == 0:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1448,440))
   elif Data_Num == 1:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1438,415))
   elif Data_Num == 2:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1434,421))
   elif Data_Num == 3:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1426,418))
   elif Data_Num == 4:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1426,418))
   elif Data_Num == 5:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1432,421))
   elif Data_Num == 6:
      out = cv2.VideoWriter("Video_"+SaveName+".avi",\
                            fourcc, 30.0, (1426,418))

np.set_printoptions(suppress=True)

# Setting parameters based on disparity type selected
if disparity_type == "BM":
   left_matcher = cv2.StereoBM_create(\
       numDisparities=16*10, blockSize=15)
```

```
elif disparity_type == "BM+WLS":
   left_matcher = cv2.StereoBM_create(\
       numDisparities=16*10, blockSize=15)
   right_matcher = cv2.ximgproc.createRightMatcher(left_matcher)
   wls_filter = cv2.ximgproc.createDisparityWLSFilter(\
       matcher_left=left_matcher)
   wls_filter.setLambda(8000)
   wls_filter.setSigmaColor(0.8)
else:
   print ("Incorrect Disparity Method Selected")
   sys.exit()

i = 0
k = 0

if flag_display == 1:
    cv2.namedWindow("Stereo VO MAP")

print ("CODE STARTED")
# Starting the process
while(k!=27 and i<Data_Frames[Data_Num]):
   if Pause == False:
      currID = i

      # Reading LEft and Right Stereo Pair
      Left_Img_path = Data_dir + "/I1_" + str(i).zfill(6) + ".png"
      curr[0] = cv2.imread(Left_Img_path,0)

      Right_Img_path = Data_dir + "/I2_" + str(i).zfill(6) + ".png"
      curr[1] = cv2.imread(Right_Img_path,0)

      # Calculating Disparity
      if disparity_type == "BM":
         curr[2] = np.int16(left_matcher.compute(curr[0],curr[1]))
      elif disparity_type == "BM+WLS":
         displ = np.int16(left_matcher.compute(curr[0],curr[1]))
         dispr = np.int16(right_matcher.compute(curr[1],curr[0]))
         curr[2] = wls_filter.filter(displ, curr[0], None, dispr)

      # Normalizing Disparity Map for Visualization
      disp_view = cv2.normalize(curr[2], None, beta=0,\
                alpha=np.amax(curr[2])/16, norm_type=cv2.NORM_MINMAX);
```

```
disp_view = np.uint8(disp_view)

# Dynamic Disparity Mask Thresholding
temp_thr = dist_thr[0]
curr[3] = cv2.inRange(\
    disp_view,int(f*base/dist_thr[1]),int(f*base/dist_thr[0]))
while np.sum(curr[3]) < 20000000:
    temp_thr -= 1
    curr[3] = cv2.inRange(\
        disp_view,int(f*base/dist_thr[1]),int(f*base/temp_thr))
    if temp_thr == 3:
        break

# Trimming Bottom, Left, Right part in the mask because
# Bottom : Mostly Road
# Top : Mostly Sky
# Right : Features will not be there in next frame
rows = curr[3].shape[0]
cols = curr[3].shape[1]
curr[3][int(3*rows/4):,:] = 0
curr[3][:int(1*rows/10),:] = 0
curr[3][:,int(17*cols/20):] = 0

# Reading Ground Truth for comparison
CurGT[0] = float(GT_file_lines[currID].split()[4])
CurGT[1] = float(GT_file_lines[currID].split()[5])

bgr[0] = cv2.cvtColor(curr[0],cv2.COLOR_GRAY2BGR)
# bgr[1] = cv2.cvtColor(curr[1],cv2.COLOR_GRAY2BGR)
if currID == prevID + 1:
   # Finding fast features with Disparity Mask and Binning
   Pts_1 = find_keypoints(prev[0],prev[3],20,20,1)
   # Tracking the features detected
   Pts_1,Pts_2 = track_keypoints(\
       prev[0],curr[0],Pts_1,nPts_track)
   # Finding 3D Coordinates for features tracked
   Pts_1,Pts_2,Pts3D_1,Pts3D_2 = \
       Calc_3DPts(prev[2],Pts_1,curr[2],Pts_2,f,base,cx,cy,16)

   # Choosing method to select points for optimization
   if select_type == "Outlier":
       Pts_1F, Pts_2F, Pts3D_1F, Pts3D_2F = \
```

```
        find_bestPts_OR(Pts_1,Pts_2,Pts3D_1,Pts3D_2,nPts_opt)
elif select_type == "Inlier":
    clique = find_bestPts_ID(Pts3D_1,Pts3D_2,nPts_opt)
    Pts_1F = [Pts_1[i] for i in clique]
    Pts_2F = [Pts_2[i] for i in clique]
    Pts3D_1F = [Pts3D_1[i] for i in clique]
    Pts3D_2F = [Pts3D_2[i] for i in clique]
else:
    print ("Incorrect Feature Selection Method")
    sys.exit()

# Homogenizing the coordinates
homo = np.ones((len(Pts3D_1F),1))
Pts_1F = np.hstack((Pts_1F,homo))
Pts_2F = np.hstack((Pts_2F,homo))
Pts3D_1F = np.hstack((Pts3D_1F,homo))
Pts3D_2F = np.hstack((Pts3D_2F,homo))

# Running the optimization
dSeed = np.zeros(len(Pts3D_1F))
optRes = least_squares(mini, dSeed, method='lm', \
         max_nfev=200,args=(Pts3D_1F, Pts3D_2F, Pts_1F,\
         Pts_2F,P1_roi))

# Finding Rotation and Translation
Rmat = genEulerZXZMatrix(\
    optRes.x[0], optRes.x[1], optRes.x[2])
Trans = np.array(\
    [[optRes.x[3]], [optRes.x[4]], [optRes.x[5]]])

# Updating the odometry
newPosition = np.vstack(\
    (np.hstack((Rmat,Trans)),[0, 0, 0, 1]))
Position = np.dot(Position,newPosition)

# Processing Ground Truth for plotting
GT_x = CurGT[0] - StartGT[0]
GT_y = CurGT[1] - StartGT[1]

# Resizing map dynamically
while (centre+int(GT_x) >= Map.shape[0]-25 or\
       centre-int(Position[0,3]) >= Map.shape[0]-25 or\
```

```
            centre-int(GT_y) >= Map.shape[1]-25 or \
            int(Position[2,3])+centre >= Map.shape[1]-25):
        Map = np.insert(Map,len(Map[0]),0,axis=1)
        Map = np.insert(Map,len(Map),0,axis=0)
    while (centre+int(GT_x) <= 25 or\
            centre-int(Position[0,3]) <= 25 or\
            centre-int(GT_y) <= 25 or\
            int(Position[2,3])+centre <= 25):
        Map = np.insert(Map,0,0,axis=1)
        Map = np.insert(Map,0,0,axis=0)
        centre+=1

    # Plotting Ground Truth point in RED
    cv2.circle(Map,(centre+int(GT_x),centre-int(GT_y)),\
               2,color=(0,0,255),thickness = -1)
    # Plotting VO point in Green
    cv2.circle(Map, (centre-int(Position[0,3]),\
               int(Position[2,3])+centre),2,\
               color=(0,255,0), thickness = -1)

    # Drawing the tracked features
    for j in range(len(Pts_2)):
        point = Pts_2[j,:]
        cv2.circle(bgr[0], (int(point[0]), int(point[1])), 10,\
                   color=(0,0,255), thickness = 3)
    # Drawing the features for optimization
    for j in range(len(Pts_2F)):
        point = Pts_2F[j,:]
        cv2.circle(bgr[0], (int(point[0]), int(point[1])), 10,\
                   color=(0,255,0), thickness = 3)

elif prevID != 0:
    print ("Frames missed... Stopping execution")
    break

prev[0] = curr[0]
prev[1] = curr[1]
prev[2] = curr[2]
prev[3] = curr[3]
prevID = currID
i = i + 1
```

```
# Processing data for display purposes
if flag_display == 1:
    bgr[2] = cv2.cvtColor(disp_view,cv2.COLOR_GRAY2BGR)
    bgr[3] = cv2.cvtColor(curr[3],cv2.COLOR_GRAY2BGR)

    bgr[2] = cv2.resize(bgr[2],None,fx=0.5, fy=0.5,\
                        interpolation = cv2.INTER_CUBIC)
    bgr[3] = cv2.resize(bgr[3],None,fx=0.5, fy=0.5,\
                        interpolation = cv2.INTER_CUBIC)

    display_image = np.concatenate((bgr[2], bgr[3]),1)
    if display_image.shape[1]>=bgr[0].shape[1]:
       display_image = np.concatenate(\
           (bgr[0],display_image[:,0:bgr[0].shape[1],:]),0)
    else:
       display_image = np.concatenate(\
           (bgr[0][:,0:display_image.shape[1],:], display_image),0)

    display_image = cv2.resize(display_image,None,\
                fx=0.75, fy=0.75,interpolation = cv2.INTER_CUBIC)
    scale = float(display_image.shape[0])/float(Map.shape[0])
    display_map = cv2.resize(Map,None,fx=scale,fy=scale,\
                             interpolation = cv2.INTER_CUBIC)
    display = np.concatenate((display_image, display_map),1)
    cv2.imshow("Stereo VO MAP",display)

    if flag_record == 2 and Pause == False \
       and currID <= Data_Frames[Data_Num]-1:
       out.write(display)
       if currID == Data_Frames[Data_Num]-1:
          out.release()
          print("Recording closed******************")

    if flag_record >= 1 and currID == Data_Frames[Data_Num]-1:
        cv2.imwrite("Map_"+SaveName+".png",Map)

    k = cv2.waitKey(1)

    if k == ord('p'):
       Pause = not Pause
    elif k == 27:
       break
```

```
    else:
        if (i%100 == 1):
            print (100*i/Data_Frames[Data_Num], "% completed")

# Displaying the processing time
code_proc_time = time.time() - code_proc_time
print ("CODE COMPLETED")
print(i/code_proc_time, "FPS")
if i == Data_Frames[Data_Num]:
    cv2.waitKey(0)
cv2.destroyAllWindows()
```

A2. Python Code SVO - Helper Functions

```
#!/usr/bin/env python

import sys
import cv2
import numpy as np
from scipy.spatial import distance
from scipy.optimize import least_squares
from math import *
debug = 0

# Function to get rotation matrix
def genEulerZXZMatrix(psi, theta, sigma):
    c1 = cos(psi)
    s1 = sin(psi)
    c2 = cos(theta)
    s2 = sin(theta)
    c3 = cos(sigma)
    s3 = sin(sigma)

    mat = np.zeros((3,3))

    mat[0,0] = (c1 * c3) - (s1 * c2 * s3)
    mat[0,1] = (-c1 * s3) - (s1 * c2 * c3)
    mat[0,2] = (s1 * s2)
    mat[1,0] = (s1 * c3) + (c1 * c2 * s3)
    mat[1,1] = (-s1 * s3) + (c1 * c2 * c3)
    mat[1,2] = (-c1 * s2)
    mat[2,0] = (s2 * s3)
    mat[2,1] = (s2 * c3)
    mat[2,2] = c2

    return mat

# Function for least squares optimization
def mini(dof,random_3d_1, random_3d_2, random_2d_1, random_2d_2,P):
    Rmat = genEulerZXZMatrix(dof[0], dof[1], dof[2])
    translationArray = np.array([[dof[3]], [dof[4]], [dof[5]]])
    temp = np.hstack((Rmat, translationArray))
    perspectiveProj = np.vstack((temp, [0, 0, 0, 1]))
    forward = np.matmul(P,perspectiveProj)
    backward = np.matmul(P, np.linalg.inv(perspectiveProj))
```

```
    numPoints = len(random_2d_1)
    errorA = np.zeros((numPoints,3))
    errorB = np.zeros((numPoints,3))
    pred2d_1 =[]
    pred2d_2 =[]
    for i in range(len(random_3d_1)):
      pred2d_1.append(np.matmul(forward,random_3d_2[i]))
      pred2d_1[i] = pred2d_1[i]/pred2d_1[i][-1]
      pred2d_2.append(np.matmul(backward,random_3d_1[i]))
      pred2d_2[i] = pred2d_2[i]/pred2d_2[i][-1]
      error_1 = random_2d_1[i]-pred2d_1[i]
      error_2 = random_2d_2[i]-pred2d_2[i]
      errorA[i,:] = error_1.reshape(1,3)[0]
      errorB[i,:] = error_2.reshape(1,3)[0]
    residual = np.vstack((errorA,errorB))
    return residual.flatten()

# Function to find fast features in the image
# Disparity Mask + Feature Binning
def find_keypoints(Image,Mask,Tile_H,Tile_W,nFeatures):
    featureEngine = cv2.FastFeatureDetector_create()
    H,W = Image.shape
    kp = []
    for y in range(0, H, Tile_H):
        for x in range(0, W, Tile_W):
            Patch_Img = Image[y:y+Tile_H, x:x+Tile_W]
            Patch_Mask = Mask[y:y+Tile_H, x:x+Tile_W]
            keypoints = featureEngine.detect(\
                Patch_Img,mask=Patch_Mask)
            for pt in keypoints:
                pt.pt = (pt.pt[0] + x, pt.pt[1] + y)

            if (len(keypoints) > nFeatures):
                keypoints = sorted(keypoints,\
                                   key=lambda x: -x.response)
                for kpt in keypoints[0:nFeatures]:
                    kp.append(kpt)
            else:
                for kpt in keypoints:
                    kp.append(kpt)

    trackPts = cv2.KeyPoint_convert(kp)
```

```
    trackPts = np.expand_dims(trackPts, axis=1)

    global debug
    if debug == 1:
        print ("# Points Tracked : " + str(len(trackPts)))
    return trackPts

# Function to track keypoints from T to T+1 using LK Optical Flow
def track_keypoints(Image_1,Image_2,Pts_1,nPts):
    # Parameters for lucas kanade optical flow
    lk_params = dict( winSize  = (15,15),maxLevel = 3,\
                      criteria = (cv2.TERM_CRITERIA_EPS | \
                      cv2.TERM_CRITERIA_COUNT, 50, 0.03))
    Pts_2, st, err = cv2.calcOpticalFlowPyrLK(\
                     Image_1,Image_2, Pts_1, None,\
                     flags=cv2.MOTION_AFFINE, **lk_params)

    # separate points that were tracked successfully
    ptTrackable = np.where(st == 1, 1,0).astype(bool)
    TrkPts_1 = Pts_1[ptTrackable, ...]
    TrkPts_2 = Pts_2[ptTrackable, ...]
    TrkPts_2 = np.around(TrkPts_2)
    global debug
    if debug == 1:
        print ("Points successfully tracked: " + str(len(Pts_2)))

    error = 4
    errTrackablePts = err[ptTrackable, ...]
    errThreshPts = np.where(errTrackablePts < \
                            error, 1, 0).astype(bool)
    # Dynamically change threshold to get required points
    while np.count_nonzero(errThreshPts) > nPts:
        error = round(error - 0.1,1)
        errThreshPts = np.where(errTrackablePts < \
                                error, 1, 0).astype(bool)

    while np.count_nonzero(errThreshPts) < nPts :
        error = round(error + 0.1,1)
        errThreshPts = np.where(errTrackablePts < \
                                error, 1, 0).astype(bool)
        if error >= 8:
            if debug == 1:
```

```
                print ("Max Limit Reached... Exiting loop")
            break

    TrkPts_1 = TrkPts_1[errThreshPts, ...]
    TrkPts_2 = TrkPts_2[errThreshPts, ...]
    if debug == 1:
        print ("Points with error less than " \
               + str(error) + " : " + str(len(TrkPts_1)))

    return TrkPts_1,TrkPts_2

# Finding 3D coordinate of selected features using Disparity Map
def Calc_3DPts(DisparityA,PointsA,DisparityB,PointsB,\
               f,base,cx,cy,scale):
    Pts_3DA = []
    Pts_3DB = []
    Pts_2DA = []
    Pts_2DB = []
    for j in range(len(PointsA)):
        PtA = PointsA[j]
        PtB = PointsB[j]

        dA = DisparityA[int(PtA[1])][int(PtA[0])]/scale
        dB = DisparityB[int(PtB[1])][int(PtB[0])]/scale
        if dA > 0 and dB > 0:
            Pts_3DA.append([base*(PtA[0] - cx)/dA,\
                            base*(PtA[1] - cy)/dA, f*base/dA])
            Pts_3DB.append([base*(PtB[0] - cx)/dB,\
                            base*(PtB[1] - cy)/dB, f*base/dB])
            Pts_2DA.append(PtA)
            Pts_2DB.append(PtB)
    return np.asarray(Pts_2DA),np.asarray(Pts_2DB),\
           np.asarray(Pts_3DA),np.asarray(Pts_3DB)

# Finding best points using Improved Inlier Detection
def find_bestPts_ID(point_cloud1,point_cloud2,minReq) :
    dist_difference = 0.05
    max_node = -1
    max_count = 0
    point_cloud1 = np.asarray(point_cloud1)
    point_cloud2 = np.asarray(point_cloud2)
```

```
num_points = point_cloud1.shape[0]
W = np.zeros((num_points,num_points))
count = 0
point_clouds_relative_dist = np.zeros((num_points,num_points))
while max_node == -1:
    for i in range(num_points) :
        diff_nodes_t1 = point_cloud1 - point_cloud1[i,:]
        diff_nodes_t2 = point_cloud2 - point_cloud2[i,:]
        dist_nodes_t1 = np.linalg.norm(diff_nodes_t1,axis=1)
        dist_nodes_t2 = np.linalg.norm(diff_nodes_t2,axis=1)
        abs_dist = abs(dist_nodes_t1 - dist_nodes_t2)

        point_clouds_relative_dist[i] = \
            np.asarray(abs_dist).T
        wIdx = np.where(abs_dist < dist_difference)
        W[i,wIdx[0]] = 1
        count = np.sum(W[i,:])
        if count > max_count:
            max_count = count
            max_node = i
    if max_count < minReq and dist_difference < 0.5 :
        max_count = 0
        max_node = -1
    if max_node == -1:
        dist_difference += 0.01
count = 0
clique = [max_node]

while True :
    max_count = 0
    max_node = 0
    potentialnodes = list()
    Wsub = W[clique,:]
  # print(Wsub)
    for i in range(num_points) :
        sumclique = np.sum(Wsub[:,i])
        if sumclique == len(clique) :
            isin = True
        else :
            isin = False
        if isin == True and i not in clique :
            potentialnodes.append(i)
```

```
            max_count = 0
            max_node = 0
            for i in range(len(potentialnodes)) :
                Wsub = W[potentialnodes[i],potentialnodes]
                sumclique = np.sum(Wsub)
                if sumclique > max_count :
                    max_count = sumclique
                    max_node = potentialnodes[i]

            if max_count == 0 :
                if len(clique) >= minReq :
                    break
                else :
                    dist_difference += 0.05
                    for k in range(num_points) :
                        diff_nodes_t1 = point_cloud1 \
                                        - point_cloud1[k,:]
                        diff_nodes_t2 = point_cloud2 \
                                        - point_cloud2[k,:]
                        dist_nodes_t1 = \
                            np.linalg.norm(diff_nodes_t1,axis=1)
                        dist_nodes_t2 = \
                            np.linalg.norm(diff_nodes_t2,axis=1)
                        abs_dist = abs(dist_nodes_t1 - dist_nodes_t2)
                        point_clouds_relative_dist[k] = \
                            np.asarray(abs_dist).T
                        wIdx = np.where(abs_dist < dist_difference)
                        W[k,wIdx[0]] = 1

            if len(clique) >= minReq or dist_difference > 10  :
                break
            clique.append(max_node)
        return clique

    # Finding Best Points using Outlier Rejection
    def find_bestPts_OR(Pts_1,Pts_2,Pts3D_1,Pts3D_2,minReq):
        if len(Pts3D_1) < 6:
            print("ERROR : Less than 6 points")
            print(len(Pts3D_1))
            sys.exit()
        Compare3D = np.zeros((len(Pts3D_1),len(Pts3D_1)))
        for i in range(len(Pts3D_1)):
```

```
    for j in range(len(Pts3D_1)):
        Dis_1 = distance.euclidean(Pts3D_1[i],Pts3D_1[j])
        Dis_2 = distance.euclidean(Pts3D_2[i],Pts3D_2[j])
        Compare3D[i,j] = abs(Dis_1-Dis_2)

Sum3D = np.sum(Compare3D,axis = 1)
FinalIndex = np.argsort(Sum3D)
while len(Sum3D) > minReq:
    Compare3D = np.delete(Compare3D,FinalIndex[len(Sum3D)-1],0)
    Compare3D = np.delete(Compare3D,FinalIndex[len(Sum3D)-1],1)
    Pts_1 = np.delete(Pts_1,FinalIndex[len(Sum3D)-1],0)
    Pts_2 = np.delete(Pts_2,FinalIndex[len(Sum3D)-1],0)
    Pts3D_1 = np.delete(Pts3D_1,FinalIndex[len(Sum3D)-1],0)
    Pts3D_2 = np.delete(Pts3D_2,FinalIndex[len(Sum3D)-1],0)
    Sum3D = np.sum(Compare3D,axis = 1)
    FinalIndex = np.argsort(Sum3D)

return np.asarray(Pts_1),np.asarray(Pts_2),\
       np.asarray(Pts3D_1),np.asarray(Pts3D_2)
```