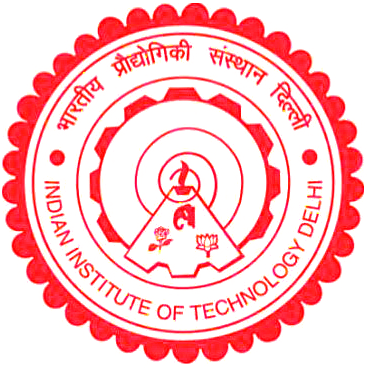
­­





|  |
| --- |
|  |
| **ROS DLIVE SUMMER 2018** |
|  |
| **INTERNSHIP REPORT** |

**meCHANICAL DEPARTMENT , IIT DELHI**

Duration – 7thMay ’18 – 25th June ’18

Author – Sree Aslesh Penisetty

ROS DLIVE SUMMER 2018

INTERNSHIP REPORT

# INTRODUCTION –

Upon introduction to the scope of autonomous navigation and its various prospects I was learning and researching under Dr. Sunil jha as a research intern in the ROS DLIVE team of IIT Delhi. This is the first time I have worked on a project as elaborate, detailed and involved as this. We were given the task of improving localization to support the navigation of the car using various sensors and data. The sensors provided to us were -

* GPS (ublox M8N)
* IMU (invensense 9255)
* Odometry source (ZED stereo camera)

This report is to present a detailed description of our findings, research , codes and methods to overcome errors we faced.

# WEEK 1 –

Most of the week went about adjusting to and familiarizing ourselves with the codes and how much work had been done on the car. I learnt about PCAN and the way we send commands to the car. We also started all over right from Ubuntu when we got here as there was some problem in the system. We were going about understanding various parts of the codes and also installing various drivers such as Nvidia graphic drivers and Cuda along with ZED.

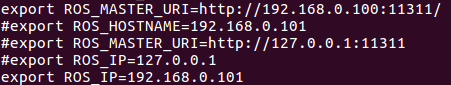
Also we were studying about the fusion of sensors and how to use them with other Odometry data and we found about using the Kalman filter which worked greatly to the advantage of improving the data upon fusion of the data. UKF was also another method to improve the sensor data.

While reviewing all of this we found out that in all of these there was no data which was closely related to a tradeoff between accuracy and continuity so we let the idea of improving the data rest aside while we worked on getting data which was accurate as well as continuous.

We had a bit of trouble accessing the raspberry pi in the car which is used to get GPS and IMU data and publish those topics over to the main computer. We had to change some configuration of ROS parameters such as

* ROS\_MASTER\_URI
* ROS\_HOSTNAME
* ROS\_IP

After that we could successfully access and also ping the raspberry pi, get the topic published over the Ethernet connection. Topics we are obtaining from the raspberry pi are –



* # indicates the commented out ones which we used previously, uncommented marks the ones which we are currently using

# Week 2 –

We found out that the zed stereo camera had a node which had a topic and it was publishing an Odometry message of type nav\_msgs/Odometry of its own. So we updated the zed ros wrapper and started working on checking whether that Odometry was good enough for us to work with or not. While working with this we have to keep in mind that the axis of zed stereo cam is different as compared to the axis we assume.

The following are the results we got -

# Week 3 –

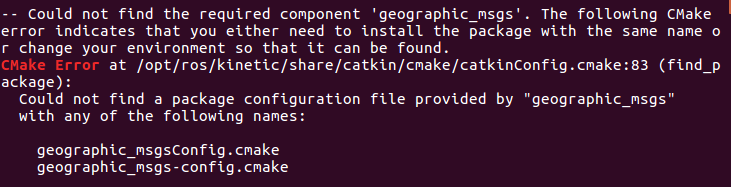
Based on the results we obtained, verifying the visual representation of the graph and as well as the log file it was concluded that we needed a better method for the Odometry.

So we started searching and reading about other methods of localizations using visual perception. In that process we found –

1. Rtabmap
2. Orb slam
3. Gslam
4. Lsd slam
5. Rgbd slam

These were some of the main methods of slam which we found and I particularly started working on ORB SLAM2 authored by Raul Mur. There were some issues encountered while building the package and the specific work around are specified in these following links –

Another problem we encountered were the unavailability of packages, something in the neighborhood of –



Then all I had to do was

* sudo apt-get ros-kinetic-geographic-msgs

And that will download the missing library, also make note that \_ is to be changed to – before entering the sudo apt-get command.

We started using the orb slam in its default mode and checked for the loop closure, although here we had relative values there were no specific values of x,y,z which were recorded or tabulated , so we had to do that on our own. The first test of orb slam was done using a monocular camera which was not very successful. We were just verifying how accurately orb slam2 would work and the inferences of the monocular slam are as follows

1. Took a lot of time to get the required number of matches
2. No absolute value of scale
3. Quickly lost mapping
4. Too varied with speed
5. Too varied with the environment lighting conditions as well
6. Takes too long to initialize

So with all these points on the table we moved on to stereo ORB SLAM2 hoping for better results

# Week 4 –

After a lot of working around with the orb slam we started verification of the loop closure and all the various other things included in the orb slam. We checked for correctness in the data when we figured out there was more to it and we had to tune the camera parameters.

The camera parameters such as fx and fy are mentioned also various other parameters are mentioned as well in the config file of the zed camera named as SN11268.conf.

[cx,cy] — Optical center (the principal point), in pixels.

(fx,fy)— Focal length in pixels.   
*fx*=*F*/*px*  
*fy*=*F*/*py*  
*F* — Focal length in world units, typically expressed in millimeters.  
(Px,Py)— Size of the pixel in world units.

s — Skew coefficient, which is non-zero if the image axes are not perpendicular.  
s=fytanα

This can be found at the path – {root dir}/{zed dir}/ config/--

**SN11268.conf**

[LEFT\_CAM\_2K]

fx = 1401.76

fy = 1401.76

cx = 1159.16

cy = 625.376

k1 = -0.168312

k2 = 0.0216054

[RIGHT\_CAM\_2K]

fx = 1397.95

fy = 1397.95

cx = 1164.75

cy = 620.326

k1 = -0.171194

k2 = 0.0250973

[LEFT\_CAM\_FHD]

fx = 1401.76

fy = 1401.76

cx = 1015.16

cy = 544.376

k1 = -0.168312

k2 = 0.0216054

[RIGHT\_CAM\_FHD]

fx = 1397.95

fy = 1397.95

cx = 1020.75

cy = 539.326

k1 = -0.171194

k2 = 0.0250973

[LEFT\_CAM\_HD]

fx = 700.881

fy = 700.881

cx = 666.082

cy = 360.688

k1 = -0.168312

k2 = 0.0216054

[RIGHT\_CAM\_HD]

fx = 698.977

fy = 698.977

cx = 668.877

cy = 358.163

k1 = -0.171194

k2 = 0.0250973

[LEFT\_CAM\_VGA]

fx = 350.44

fy = 350.44

cx = 348.541

cy = 187.844

k1 = -0.168312

k2 = 0.0216054

[RIGHT\_CAM\_VGA]

fx = 349.489

fy = 349.489

cx = 349.938

cy = 186.581

k1 = -0.171194

k2 = 0.0250973

[STEREO]

Baseline = 120

CV\_2K = 0.0106479

CV\_FHD = 0.0106479

CV\_HD = 0.0106479

CV\_VGA = 0.0106479

RX\_2K = 0.00989681

RX\_FHD = 0.00989681

RX\_HD = 0.00989681

RX\_VGA = 0.00989681

RZ\_2K = -0.000339115

RZ\_FHD = -0.000339115

RZ\_HD = -0.000339115

RZ\_VGA = -0.000339115

The topics published by zed are -

**TOPICS PUBLISHED BY ZED**

* left camera
  + **/zed/rgb/image\_rect\_color** : Color rectified image (left RGB image by default).
  + **/zed/rgb/image\_raw\_color** : Color unrectified image (left RGB image by default).
  + **/zed/rgb/camera\_info** : Camera calibration data.
  + **/zed/left/image\_rect\_color** : Color rectified left image.
  + **/zed/left/image\_raw\_color** : Color unrectified left image.
  + **/zed/left/camera\_info** : Left camera calibration data.
* Right camera
  + **/zed/right/image\_rect\_color** : Color rectified right image.
  + **/zed/right/image\_raw\_color** : Color unrectified right image.
  + **/zed/right/camera\_info** : Right camera calibration data.
* Depth and point cloud
  + **/zed/depth/depth\_registered** : Depth map image registered on left image (by default 32 bits float, in meters).
  + **/zed/point\_cloud/cloud\_registered** : Registered color point cloud.
* Visual odometry
  + **/zed/odom** : Absolute 3D position and orientation relative to zed\_initial\_frame.

The topics highlighted in yellow are the topics which are being used as input for the orb slam data

### Working with ORB\_SLAM2 (Stereo) -

Orb slam works by extracting ORB features for the purpose of tracking, mapping and localization. This extraction procedure of features is based on a light contrast based algorithm which is applied on almost every pixel to figure out the features which are the most dominant. Once these features are extracted they are tracked to obtain the relative data as compared to the previous frame. There are two datasets in the orb slam and for both, their own relative settings files.

1. Euroc dataset
2. Kitti dataset

These files are found at the path –

* ~/{ORB\_SLAM dir}/Examples/Stereo

These 2 datasets are the ones we used and tested our data on.

These settings file contain the calibration and rectification parameters required to process the images we obtain from zed ros wrapper.

// rqt graph zed ros wrapper

# Euroc Dataset –

This dataset is used when rectification of images is needed as this contains the information holding the rectification parameters which we can change according to our requirement. A .yaml file means that is the configuration or settings files.

|  |
| --- |
| LEFT.height: 480 |
|  | LEFT.width: 752 |
|  | LEFT.D: !!opencv-matrix |
|  | rows: 1 |
|  | cols: 5 |
|  | dt: d |
|  | data:[-0.28340811, 0.07395907, 0.00019359, 1.76187114e-05, 0.0] |
|  | LEFT.K: !!opencv-matrix |
|  | rows: 3 |
|  | cols: 3 |
|  | dt: d |
|  | data: [458.654, 0.0, 367.215, 0.0, 457.296, 248.375, 0.0, 0.0, 1.0] |
|  | LEFT.R: !!opencv-matrix |
|  | rows: 3 |
|  | cols: 3 |
|  | dt: d |
|  | data: [0.999966347530033, -0.001422739138722922, 0.008079580483432283, 0.001365741834644127, 0.9999741760894847, 0.007055629199258132, -0.008089410156878961, -0.007044357138835809, 0.9999424675829176] |
|  | LEFT.P: !!opencv-matrix |
|  | rows: 3 |
|  | cols: 4 |
|  | dt: d |
|  | data: [435.2046959714599, 0, 367.4517211914062, 0, 0, 435.2046959714599, 252.2008514404297, 0, 0, 0, 1, 0] |
|  |  |
|  | RIGHT.height: 480 |
|  | RIGHT.width: 752 |
|  | RIGHT.D: !!opencv-matrix |
|  | rows: 1 |
|  | cols: 5 |
|  | dt: d |
|  | data:[-0.28368365, 0.07451284, -0.00010473, -3.555907e-05, 0.0] |
|  | RIGHT.K: !!opencv-matrix |
|  | rows: 3 |
|  | cols: 3 |
|  | dt: d |
|  | data: [457.587, 0.0, 379.999, 0.0, 456.134, 255.238, 0.0, 0.0, 1] |
|  | RIGHT.R: !!opencv-matrix |
|  | rows: 3 |
|  | cols: 3 |
|  | dt: d |
|  | data: [0.9999633526194376, -0.003625811871560086, 0.007755443660172947, 0.003680398547259526, 0.9999684752771629, -0.007035845251224894, -0.007729688520722713, 0.007064130529506649, 0.999945173484644] |
|  | RIGHT.P: !!opencv-matrix |
|  | rows: 3 |
|  | cols: 4 |
|  | dt: d |
|  | data: [435.2046959714599, 0, 367.4517211914062, -47.90639384423901, 0, 435.2046959714599, 252.2008514404297, 0, 0, 0, 1, 0] |

Those are the rectification parameters used in the orb slam when we enable the rectification.

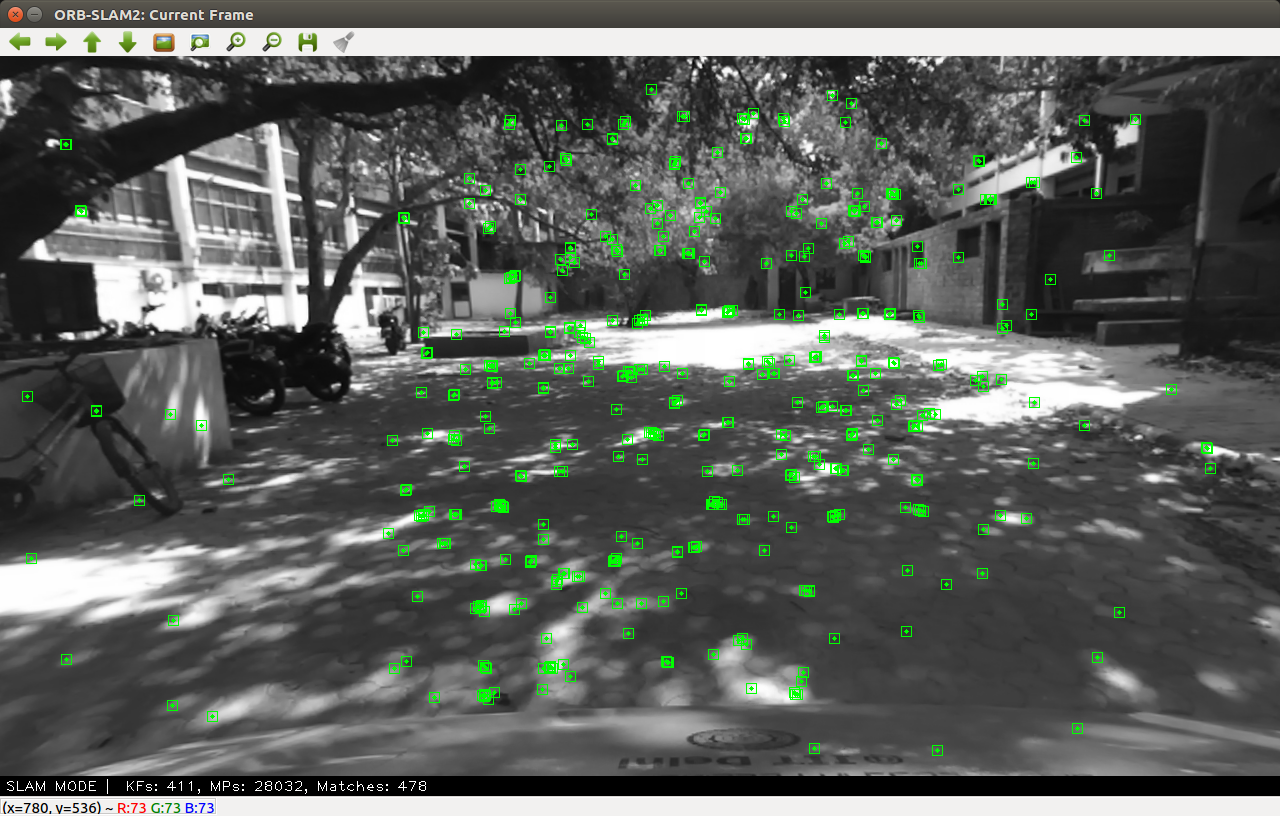
Disadvantages –

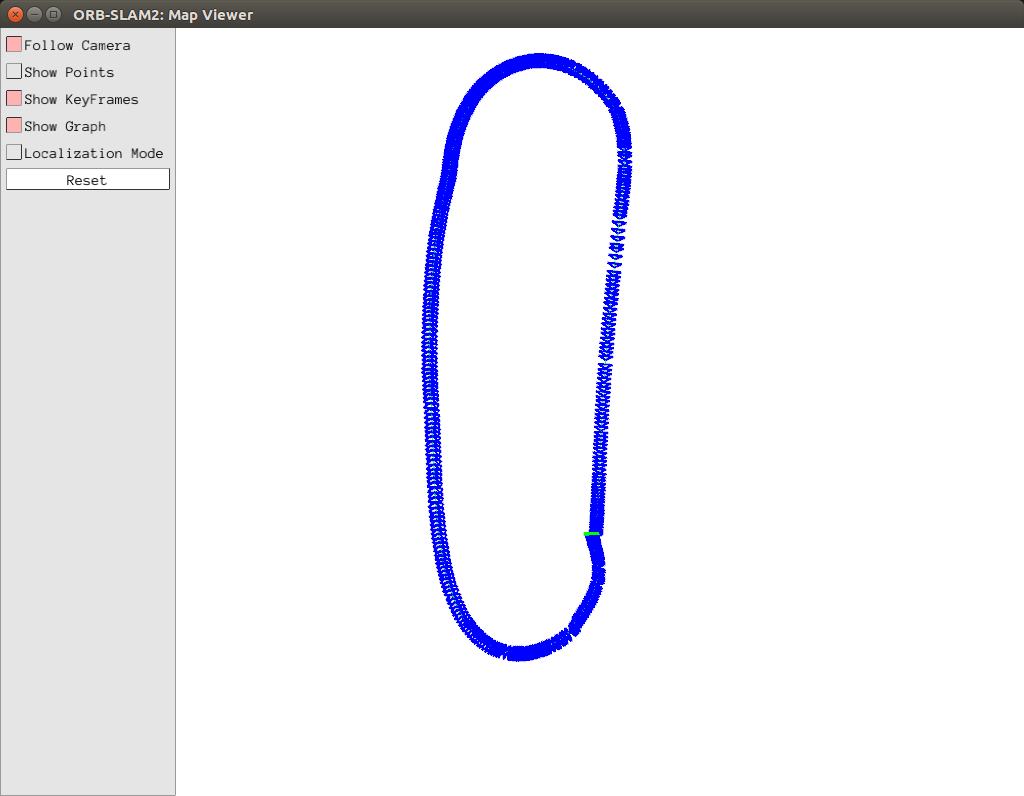
1. Rectification increases the computation time per loop
2. Increase of the computation highly affects the frames processed
3. Due to this there is a loss in frames sometime what appears to be a lag
4. This in some cases has been pertinent to the loss of tracking ORB features.

As this was resulting to be a huge interference to the overall computation, we had an option to use KITTI dataset which required pre-rectified images which we got from the zed ros wrapper.

Using this significantly increased the computation speed and reduced the frequency of lost features.

And the result we obtained from the datasets are as follows –





As we can see the loop closure was taking place very well. The only problem we had here was that we did not know the scale of every movement.

# Week 5 –

As the loop closure was happening correctly we move onto the scale, as however the relative data may be, it will always remain an assurance that the method used results in substantial data but it is always the absolute data that we have to consider. So we use the static map we have to plot that data. But before that we have to extract Odometry data from the orb slam so we had to change a part of the code to make this happen

One of the files – ROS\_STEREO it examples/ros/orb\_slam2/src folder has a part of code which refers to

**167** mpSLAM->TrackStereo(cv\_ptrLeft->image,cv\_ptrRight->image,cv\_ptrLeft->header.stamp.toSec());

This line of code returns a Tcw matrix which is not being used here so we change that part of the code to

**167** pose = mpSLAM->TrackStereo(cv\_ptrLeft->image,cv\_ptrRight->image,cv\_ptrLeft->header.stamp.toSec());

Once we did this we have a tcw matrix which is a transformation matrix from world frame to camera frame from which we are to obtain twc which will give us position and orientation in the world frame.

## Approach 1 –

Using the above pose we can calculate the translation and rotation in the world co-ordinates in the following way –

**/\* global left handed coordinate system \*/**

**static cv::Mat pose\_prev = cv::Mat::eye(4,4, CV\_32F);**

**static cv::Mat world\_lh = cv::Mat::eye(4,4, CV\_32F);**

**// matrix to flip signs of sinus in rotation matrix, not sure why we need to do that**

**static const cv::Mat flipSign = (cv::Mat\_<float>(4,4) << 1,-1,-1, 1,**

**-1, 1,-1, 1,**

**-1,-1, 1, 1,**

**1, 1, 1, 1);**

**//prev\_pose \* T = pose**

**cv::Mat translation = (pose \* pose\_prev.inv()).mul(flipSign);**

**world\_lh = world\_lh \* translation;**

**pose\_prev = pose.clone();**

**tf::Matrix3x3 tf3d;**

**tf3d.setValue(pose.at<float>(0,0), pose.at<float>(0,1), pose.at<float>(0,2),**

**pose.at<float>(1,0), pose.at<float>(1,1), pose.at<float>(1,2),**

**pose.at<float>(2,0), pose.at<float>(2,1), pose.at<float>(2,2));**

**tf::Vector3 cameraTranslation\_rh( world\_lh.at<float>(0,3),world\_lh.at<float>(1,3), - world\_lh.at<float>(2,3) );**

**//rotate 270deg about x and 270deg about x to get ENU: x forward, y left, z up**

**const tf::Matrix3x3 rotation270degXZ( 0, 1, 0,**

**0, 0, 1,**

**1, 0, 0);**

**static tf::TransformBroadcaster br;**

**tf::Matrix3x3 globalRotation\_rh = tf3d;**

**tf::Vector3 globalTranslation\_rh = cameraTranslation\_rh \* rotation270degXZ;**

**tf::Quaternion tfqt;**

**globalRotation\_rh.getRotation(tfqt);**

**double aux = tfqt[0];**

**tfqt[0]=-tfqt[2];**

**tfqt[2]=tfqt[1];**

**tfqt[1]=aux;**

**tf::Transform transform;**

**transform.setOrigin(globalTranslation\_rh);**

**transform.setRotation(tfqt);**

**br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "camera\_pose"));**

**in this the transform published is the same as the Odometry as in our case the initialization is always at (0,0,0) and the reference from the map frame is always linked with the values same as Odometry.**

### Approach 2 –

**Similar to the previous but to decrease computation and increase the accuracy we use a method of hamilton’s quaternion.**

**tf::Quaternion hamiltonProduct(tf::Quaternion a, tf::Quaternion b) {**

**tf::Quaternion c;**

**c[0] = (a[0]\*b[0]) - (a[1]\*b[1]) - (a[2]\*b[2]) - (a[3]\*b[3]);**

**c[1] = (a[0]\*b[1]) + (a[1]\*b[0]) + (a[2]\*b[3]) - (a[3]\*b[2]);**

**c[2] = (a[0]\*b[2]) - (a[1]\*b[3]) + (a[2]\*b[0]) + (a[3]\*b[1]);**

**c[3] = (a[0]\*b[3]) + (a[1]\*b[2]) - (a[2]\*b[1]) + (a[3]\*b[0]);**

**return c;**

**}**

**void ImageGrabber::GrabImage(const sensor\_msgs::ImageConstPtr& msg)**

**{**

**// Copy the ros image message to cv::Mat.**

**cv\_bridge::CvImageConstPtr cv\_ptr;**

**try**

**{**

**cv\_ptr = cv\_bridge::toCvShare(msg);**

**}**

**catch (cv\_bridge::Exception& e)**

**{**

**ROS\_ERROR("cv\_bridge exception: %s", e.what());**

**return;**

**}**

**cv::Mat pose = mpSLAM->TrackMonocular(cv\_ptr->image,cv\_ptr->header.stamp.toSec());**

**if (pose.empty())**

**return;**

**//Quaternion**

**tf::Matrix3x3 tf3d;**

**tf3d.setValue(pose.at<float>(0,0), pose.at<float>(0,1), pose.at<float>(0,2),**

**pose.at<float>(1,0), pose.at<float>(1,1), pose.at<float>(1,2),**

**pose.at<float>(2,0), pose.at<float>(2,1), pose.at<float>(2,2));**

**tf::Quaternion tfqt;**

**tf3d.getRotation(tfqt);**

**double aux = tfqt[0];**

**tfqt[0]=-tfqt[2];**

**tfqt[2]=tfqt[1];**

**tfqt[1]=aux;**

**//Translation for camera**

**tf::Vector3 origin;**

**origin.setValue(pose.at<float>(0,3),pose.at<float>(1,3),pose.at<float>(2,3));**

**//rotate 270deg about x and 270deg about x to get ENU: x forward, y left, z up**

**const tf::Matrix3x3 rotation270degXZ( 0, 1, 0,**

**0, 0, 1,**

**-1, 0, 0);**

**tf::Vector3 translationForCamera = origin \* rotation270degXZ;**

**//Hamilton (Translation for world)**

**tf::Quaternion quaternionForHamilton(tfqt[3], tfqt[0], tfqt[1], tfqt[2]);**

**tf::Quaternion secondQuaternionForHamilton(tfqt[3], -tfqt[0], -tfqt[1], -tfqt[2]);**

**tf::Quaternion translationHamilton(0, translationForCamera[0], translationForCamera[1], translationForCamera[2]);**

**tf::Quaternion translationStepQuat;**

**translationStepQuat = hamiltonProduct(hamiltonProduct(quaternionForHamilton, translationHamilton), secondQuaternionForHamilton);**

**tf::Vector3 translation(translationStepQuat[1], translationStepQuat[2], translationStepQuat[3]);**

**//Scaling**

**if(m\_numScales > 0) {**

**translation = m\_scale \* translation;**

**}**

**//Set world**

**m\_currentQ = tfqt;**

**m\_currentT = translation;**

**translation = translation - m\_worldT;**

**tfqt = tfqt \* m\_worldQ.inverse();**

**//Creates transform and populates it with translation and quaternion**

**tf::Transform transformCurrent;**

**transformCurrent.setOrigin(translation);**

**transformCurrent.setRotation(tfqt);**

**//Publishes transform**

**static tf::TransformBroadcaster br;**

**br.sendTransform(tf::StampedTransform(transformCurrent, ros::Time::now(), "world", "camera\_pose"));**

**/\* global left handed coordinate system \*/**

**static cv::Mat pose\_prev = cv::Mat::eye(4,4, CV\_32F);**

**static cv::Mat world\_lh = cv::Mat::eye(4,4, CV\_32F);**

**// matrix to flip signs of sinus in rotation matrix, not sure why we need to do that**

**static const cv::Mat flipSign = (cv::Mat\_<float>(4,4) << 1,-1,-1, 1,**

**-1, 1,-1, 1,**

**-1,-1, 1, 1,**

**1, 1, 1, 1);**

**//prev\_pose \* T = pose**

**cv::Mat translation = (pose \* pose\_prev.inv()).mul(flipSign);**

**world\_lh = world\_lh \* translation;**

**pose\_prev = pose.clone();**

**tf::Matrix3x3 tf3d;**

**tf3d.setValue(pose.at<float>(0,0), pose.at<float>(0,1), pose.at<float>(0,2),**

**pose.at<float>(1,0), pose.at<float>(1,1), pose.at<float>(1,2),**

**pose.at<float>(2,0), pose.at<float>(2,1), pose.at<float>(2,2));**

**tf::Vector3 cameraTranslation\_rh( world\_lh.at<float>(0,3),world\_lh.at<float>(1,3), - world\_lh.at<float>(2,3) );**

**//rotate 270deg about x and 270deg about x to get ENU: x forward, y left, z up**

**const tf::Matrix3x3 rotation270degXZ( 0, 1, 0,**

**0, 0, 1,**

**1, 0, 0);**

**static tf::TransformBroadcaster br;**

**tf::Matrix3x3 globalRotation\_rh = tf3d;**

**tf::Vector3 globalTranslation\_rh = cameraTranslation\_rh \* rotation270degXZ;**

**tf::Quaternion tfqt;**

**globalRotation\_rh.getRotation(tfqt);**

**double aux = tfqt[0];**

**tfqt[0]=-tfqt[2];**

**tfqt[2]=tfqt[1];**

**tfqt[1]=aux;**

**tf::Transform transform;**

**transform.setOrigin(globalTranslation\_rh);**

**transform.setRotation(tfqt);**

**br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "camera\_pose"));**

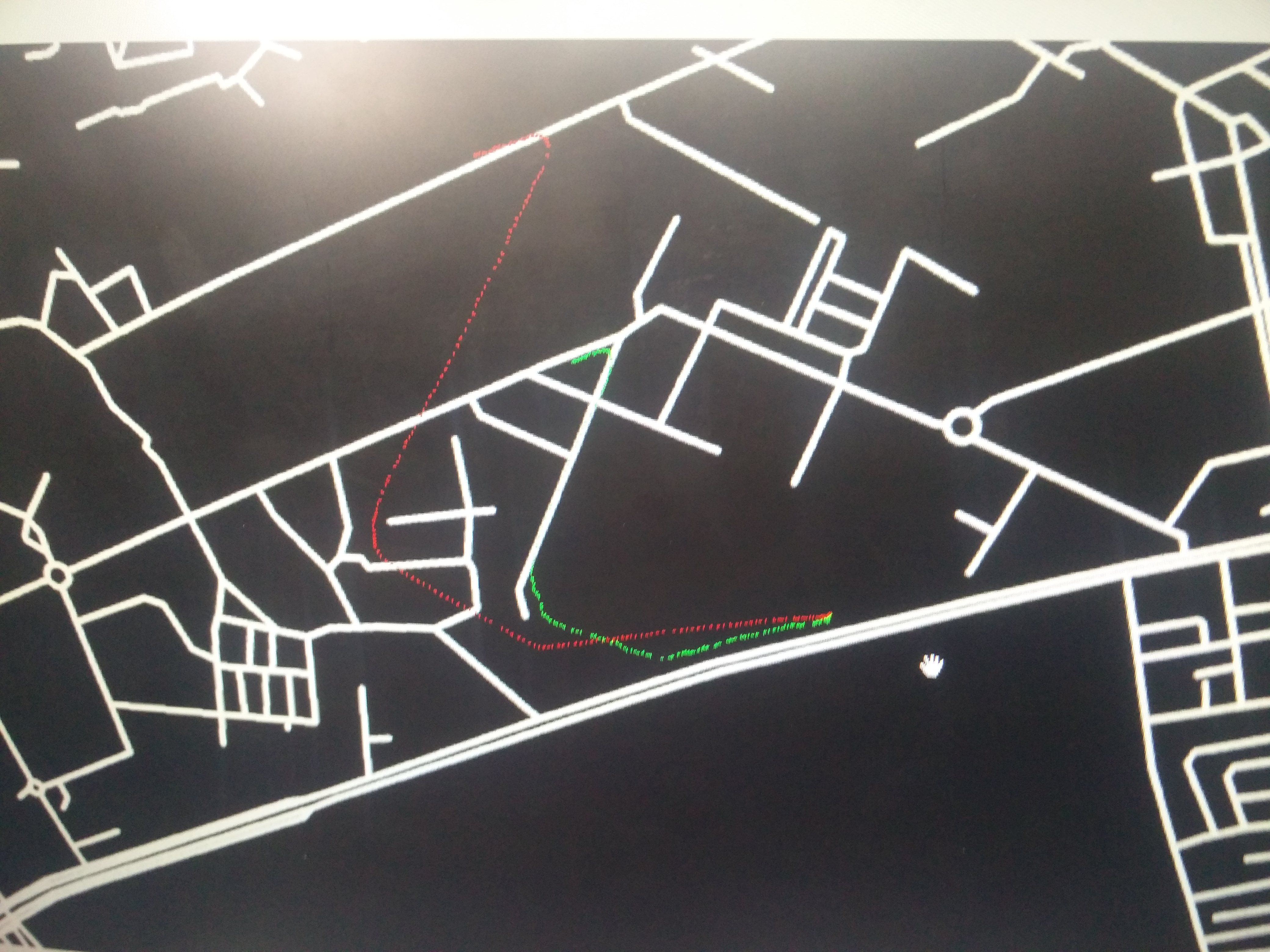
**}**

Similar to the previous case here as well we use the values of the transform as the Odometry data. (Reason already given)

We can find the explanation of quaternions and its Hamilton products here

[https://en.wikipedia.org/wiki/Quaternion#Hamilton\_product](https://en.wikipedia.org/wiki/Quaternion%23Hamilton_product)

Once we got the data –the first time we tested we got this result when we plotted it on the open street map -



From this we realized that there was a scaling error which we discussed about and then the rest of the week we tried reducing the parameters, altering functions and one which had a better output that the rest was obtained by changing this part of the code , although redundant and of no use , it still had a bit of an effect –

**//Scaling**

**if(m\_numScales > 0) {**

**translation = m\_scale \* translation;**

**}**

### Week 6 –

I was trying to improve the data when we tried to fuse the data of gps and orb slam using an averaging and mean filter

**#include <ros/ros.h>**

**#include "geometry\_msgs/Quaternion.h"**

**#include <nav\_msgs/Odometry.h>**

**#include <sensor\_msgs/NavSatFix.h>**

**#include <math.h>**

**#include <iostream>**

**using namespace std;**

**int flag=0;**

**double prev\_lat=0,prev\_lon=0,curr\_lat=0,curr\_lon=0;**

**double lat1,lat2,long1,long2;**

**double gps\_x=0,gps\_y=0;**

**double delta\_x=0,delta\_y=0;**

**double prev\_x=0,prev\_y=0;**

**double R =6371.0;**

**double a,c,d,dlong,dlat,ang;**

**double vx,vy;**

**const double pi = 22/7;**

**void get\_dist()**

**{**

**lat1 = curr\_lat;**

**long1 = curr\_lon;**

**lat2 = prev\_lat;**

**long2 = prev\_lon;**

**lat1 \*=pi/180;**

**lat2 \*=pi/180;**

**long1\*=pi/180;**

**long2\*=pi/180;**

**dlong = (long2 - long1);**

**dlat = (lat2 - lat1);**

**ang = atan2(dlong,dlat);**

**// Haversine formula:**

**a = sin(dlat/2)\*sin(dlat/2) + cos(lat1)\*cos(lat2)\*sin(dlong/2)\*sin(dlong/2);**

**c = 2 \* atan2( sqrt(a), sqrt(1-a) );**

**d = R \* c;**

**gps\_y+=(d\*cos(ang)\*2.8);**

**gps\_x+=(d\*sin(ang)\*2.8);**

**}**

**void get\_gps\_data(sensor\_msgs::NavSatFix msg)**

**{**

**cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* GPS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;**

**if(flag==0)**

**{**

**prev\_lat=msg.latitude;**

**prev\_lon=msg.longitude;**

**}**

**else**

**{**

**curr\_lat=msg.latitude;**

**curr\_lon=msg.longitude;**

**get\_dist();**

**}**

**cout << "latitude - " << msg.latitude << " longitude - " << msg.longitude << endl;**

**flag=1;**

**cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;**

**}**

**void get\_odom\_data(nav\_msgs::Odometry msg)**

**{**

**delta\_x = msg.pose.pose.position.x;**

**delta\_y = msg.pose.pose.position.y;**

**}**

**int main(int argc, char\*\* argv)**

**{**

**ros::init(argc, argv, "fuse\_data");**

**ros::NodeHandle n;**

**ros::Publisher odom\_fused = n.advertise<nav\_msgs::Odometry>("odom\_fused", 50);**

**ros::Subscriber sub\_gps = n.subscribe ( "/gps\_new", 10, get\_gps\_data );**

**ros::Subscriber sub = n.subscribe ( "/odom\_cam", 1, get\_odom\_data );**

**ros::Time current\_time, last\_time;**

**current\_time = ros::Time::now();**

**last\_time = ros::Time::now();**

**ros::Rate r(1.0);**

**while(n.ok()){**

**ros::spinOnce(); // check for incoming messages**

**current\_time = ros::Time::now();**

**//compute odometry in a typical way given the velocities of the robot**

**double dt = (current\_time - last\_time).toSec();**

**nav\_msgs::Odometry odom;**

**//set the position**

**// ROS\_INFO("gps - x is %f and y is %f",gps\_x,gps\_y);**

**// ROS\_INFO("delta - x is %f and y is %f",delta\_x,delta\_y);**

**cout << "gps \_X = " << gps\_x << " Y = " << gps\_y << endl;**

**cout << "delta \_X = " << delta\_x << " Y = " << delta\_y << endl;**

**cout << " distance - " << d << endl;**

**d=0;**

**odom.pose.pose.position.x = (delta\_x + gps\_x)/2;**

**odom.pose.pose.position.y = (delta\_y + gps\_y)/2;**

**odom.pose.pose.position.z = 0.0;**

**//odom.pose.pose.orientation = odom\_quat;**

**vx = odom.pose.pose.position.x/dt;**

**vy = odom.pose.pose.position.y/dt;**

**//set the velocity**

**odom.twist.twist.linear.x = vx;**

**odom.twist.twist.linear.y = vy;**

**odom.twist.twist.angular.z = 0;**

**//publish the message**

**odom\_fused.publish(odom);**

**last\_time = current\_time;**

**r.sleep();**

**}**

**}**

As we couldn’t extract the exact direction we are heading in from the gps without using any other sources, the value of this always resulted in a lesser value than the lower most of the mean filter itself so the result was poor.

Once this did not work, I thought of improving the data by just getting the speed of the car and differentiating it to get the values of relative positions with respect to the previous points from the imu by using the following node-

**#include <math.h>**

**#include <iostream>**

**#include <ros/ros.h>**

**#include "sensor\_msgs/Imu.h"**

**#include "std\_msgs/Float32.h"**

**using namespace std;**

**double time\_prev,time\_curr,delta;**

**float linear\_accel\_x=0,linear\_accel\_y=0,linear\_accel\_z=0;**

**float linear\_vel\_x=0,linear\_vel\_y=0,linear\_vel\_z=0;**

**void imu\_callback(sensor\_msgs::Imu msg)**

**{**

**linear\_accel\_x=msg.linear\_acceleration.x;**

**linear\_accel\_y=msg.linear\_acceleration.y;**

**linear\_accel\_z=msg.linear\_acceleration.z;**

**time\_curr=ros::Time::now().toSec();**

**delta = time\_curr - time\_prev;**

**linear\_vel\_x = (linear\_vel\_x + linear\_accel\_x \*delta)\*5/18;**

**linear\_vel\_y = (linear\_vel\_y + linear\_accel\_y \*delta)\*5/18;**

**linear\_vel\_z = (linear\_vel\_z + linear\_accel\_z \*delta)\*5/18;**

**}**

**int main(int argc, char \*\*argv)**

**{**

**ros::init(argc, argv, "speed\_node");**

**ros::start();**

**time\_prev =ros::Time::now().toSec();**

**ros::NodeHandle nh;**

**ros::Subscriber imu\_sub=nh.subscribe("/imu\_new",1000,imu\_callback);**

**// ros::Publisher speed\_pub\_x = nh.advertise<std\_msgs::Float32>("speed\_x", 100);**

**// ros::Publisher speed\_pub\_y = nh.advertise<std\_msgs::Float32>("speed\_y", 100);**

**// ros::Publisher speed\_pub\_z = nh.advertise<std\_msgs::Float32>("speed\_z", 100);**

**ros::Rate r(30);**

**while(ros::ok())**

**{**

**// std\_msgs::Float32 speed\_x;**

**// std\_msgs::Float32 speed\_y;**

**// std\_msgs::Float32 speed\_z;**

**// speed\_x.data = linear\_vel\_x;**

**// speed\_y.data = linear\_vel\_y;**

**// speed\_z.data = linear\_vel\_z;**

**// speed\_pub\_x.publish(speed\_x);**

**// speed\_pub\_y.publish(speed\_y);**

**// speed\_pub\_z.publish(speed\_z);**

**cout >> "X - " >> linear\_vel\_x >> " Y - " >> linear\_vel\_y >> " Z - " >> linear\_vel\_z >> endl;**

**ROS\_INFO("x\_vel = %f, y\_vel= %f , z\_vel = %f", linear\_vel\_x,linear\_vel\_y,linear\_vel\_z);**

**ROS\_INFO(" z\_vel= %f ",linear\_vel\_z);**

**ros::spinOnce();**

**r.sleep();**

**}**

**return 0;**

**}**

// speed variations. Plot from ros

Which did give some results but not as precise as we expected, also the accumulation of error in each stage of the computation will cause a huge drift in between the actual value and the erroneous value.

### Week 7 (ROBOT\_LOCALIZATION ) –

After experimenting with many methods we decided that one person would continue work on improving the data while others should move on to further topics of requirement such as robot localization. In an instance where we get better data, the respective data will be merged into the robot localization as it has this provision. Also the robot localization package is always built in the catkin workspace.

How robot localization works ? –

//rqt graph of robot localization

Robot localization I basically used to get a better as well as a stable Odometry data combined using various Odometry sources from gps, imu and visual Odometry.

So before we go about using it we have to set it up first.

// screenshot of topics

From the above topics we use –

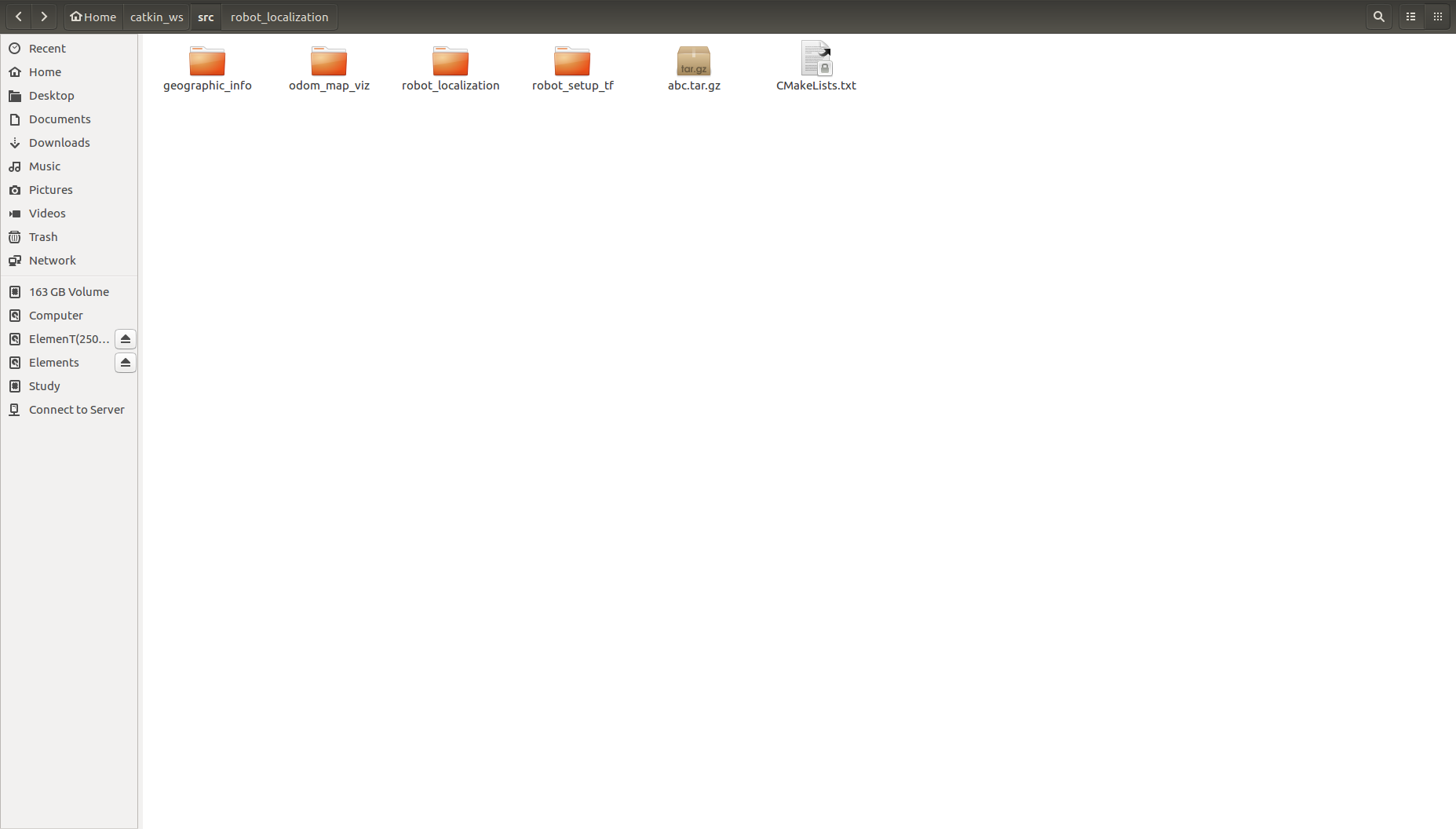
1. Odom\_cam
2. Imu\_new
3. Gps\_new

So set it up we have two methods and it depends on how we want to use the robot localization package –

1. Change topics in the Launch file
2. Change topics directly in the .cpp file we aim to use

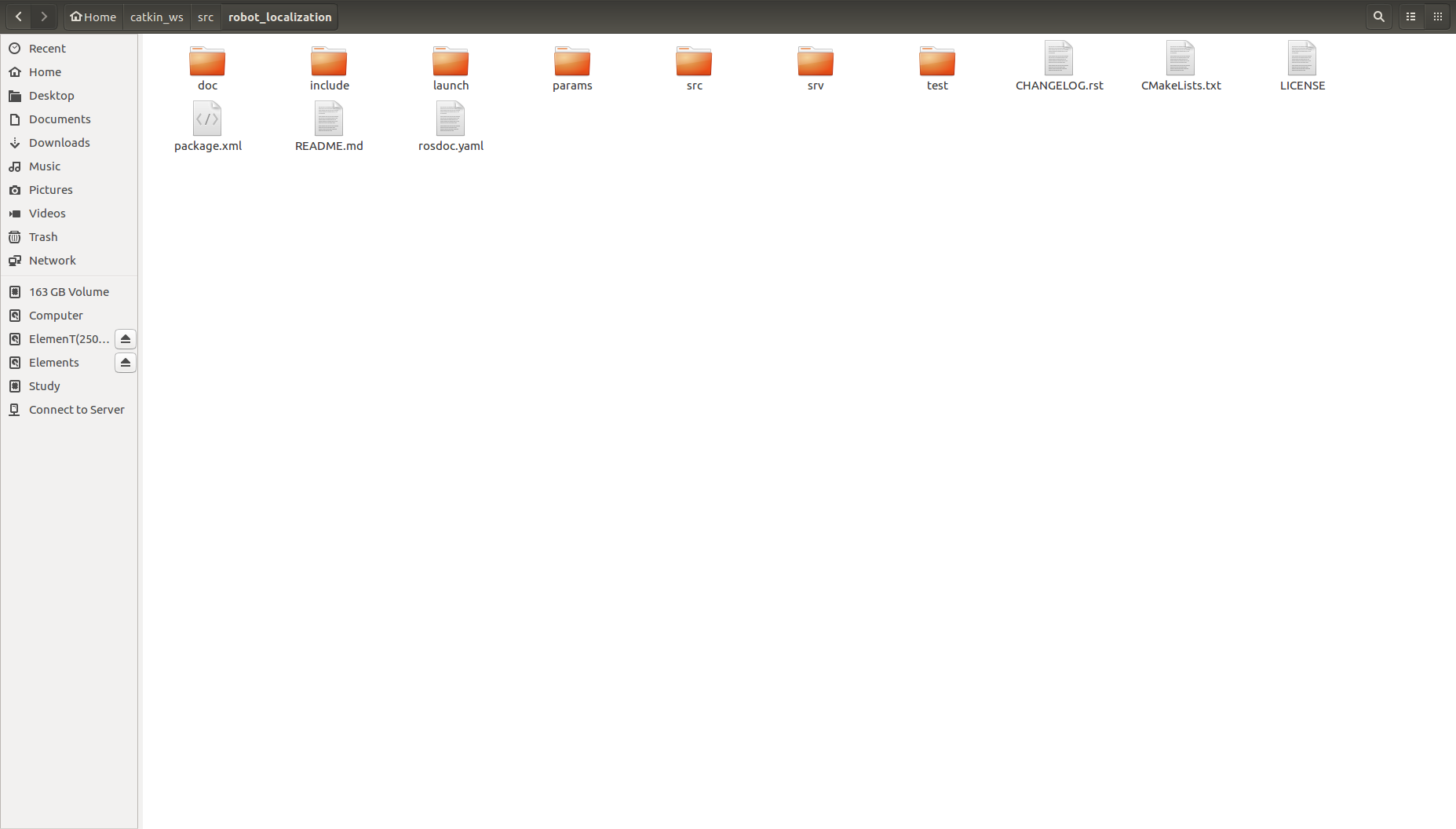
### Changing topics in the launch file –

To do this we first have to go to the src directory of our catkin workspace where the robot localization was placed and built from –



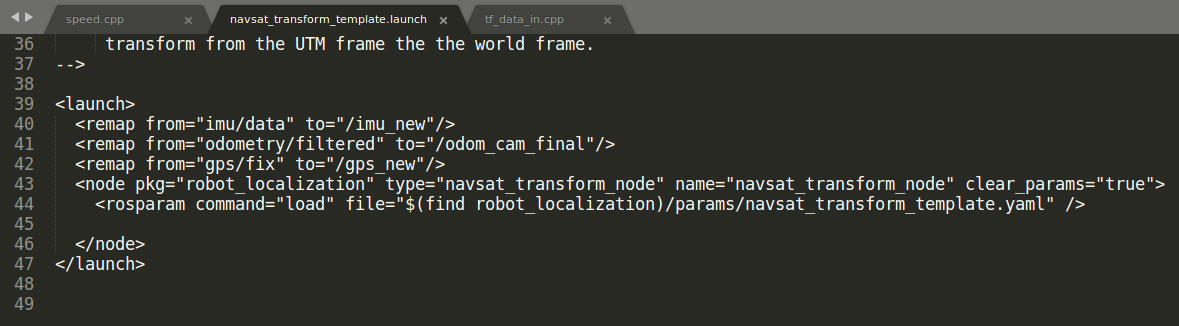
You will see something like this –

And in this if you wish to edit the launch file go to the launch folder



After this we used the file - navsat\_transform\_template.launch

In this file we edited the following lines –

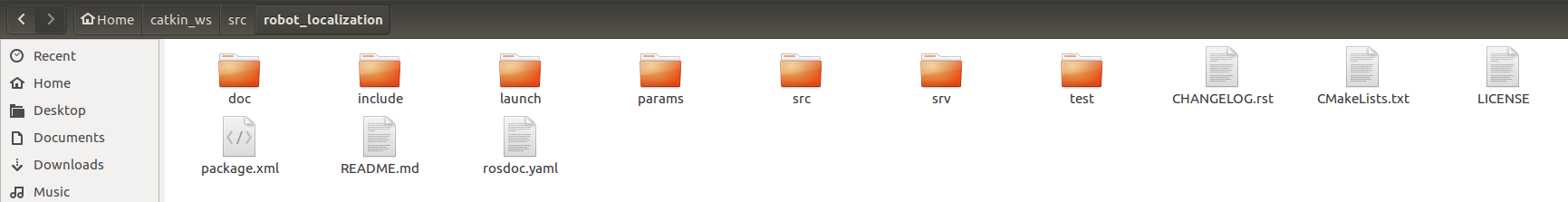


In this the lines 40, 41 and 42 were edited to change their topics to the ones which we are using as above. <remap “ ”/> function allows us to change the topics or to be more precise map them from one topics to another without the requirement of change of topic name entirely.

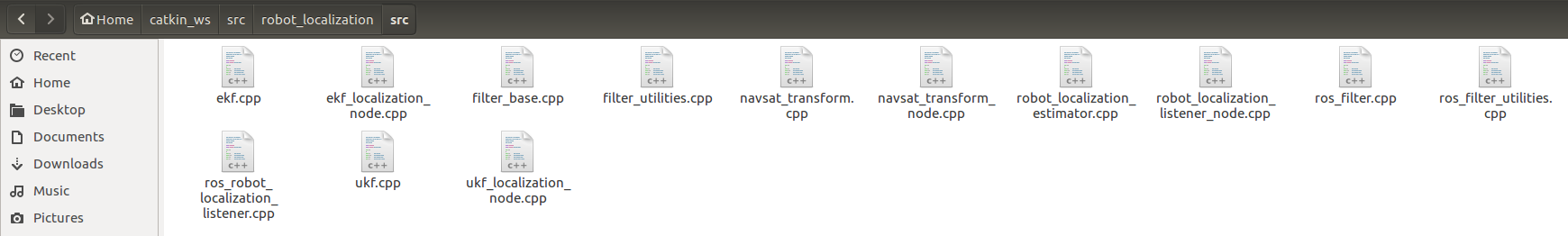
### Changing the topics in .cpp file

This is what we have done currently as we use the navsat transform node so we change the topics name directly. Also we don’t use a launch file so there is no way we can remap the topic names. So having no other choice we do it as –

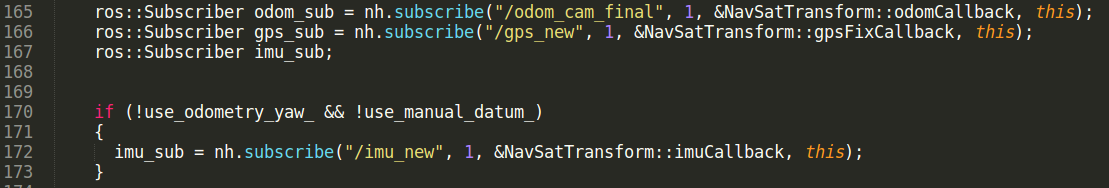
We go to the src folder of the robot localization package –



And once we enter the src directory we have many .cpp files , we need to edit navsat\_transform\_node.cpp as this contains the subscriber node



Open it with any text editor and edit the following



In this lines 165,166 and 172’s topics have been modified to the ones which we are using as shown above.

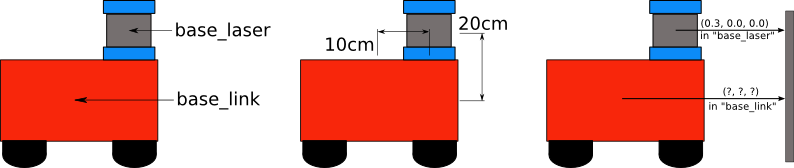
Once changing these we have successfully completed the topics part but we are note done with the setup just yet, we still have the frameand transform parts to take care of.

### Frame id and transforms –

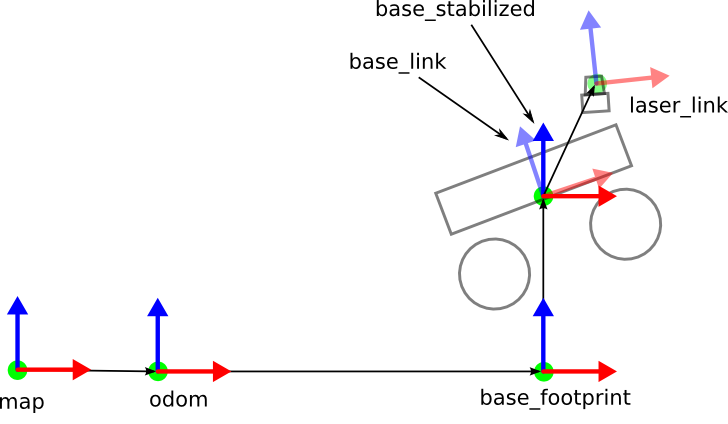
Frame id is the current frame in which the sensor is obtaining the data or the frame in which we are viewing the data. We have 4 frames basically

1. Map frame
2. Odom frame
3. Base\_link frame
4. Sensor\_link frame

Now mostly map frame is the same frame as odom frame as the frame in which we view the map will be the odom frame.

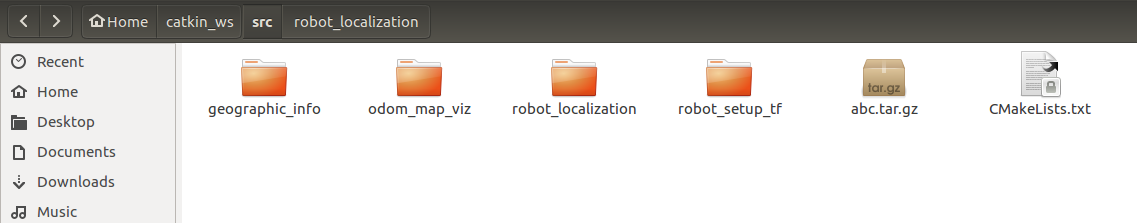


Similar to the example instead of base\_laser we have sensor\_link



As the image shown here map and odom frame are the same so they are co-incidental and we directly map it to base\_link without the use of base\_foot print so to do this I had to write a separate code and it is given below –

ROBOT\_SETUP\_TF



In the src folder of robot\_Setup\_tf there is a single code which is –

// code of robot setuptf

Once the setup is done accordingly we can run the robot\_localization package while simultaneously running the node in the robot\_setup\_tf package.

The code to run the packages are as follows –