Unit 2: Set Outdoors Navigation

Here you will learn how to access the GPS data and use it to navigate around without a mapping system. You will have to relay upon the lasers to detect any obstacles and navigate around them to get to your destination in GPS coordinates. You will also learn how to visualise real maps in RVIZ based on GPS data.

GPS basic concepts

So you have your Jackal Robot navigating with your map, thats great. But what happens when you want to move ouside your map? How do you localise yourself without AMCL system?
Well the answer is to use the **GPS data**.

It seems simple right? Well yes and no. GPS data has some limitations:

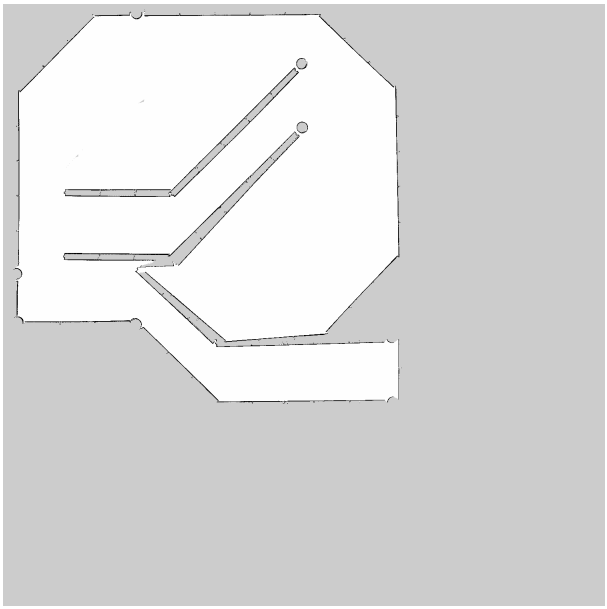
- GPS data is not very precise: GPS data oscilates in value and your position will move a lot. This is anavoidable and you will have to mitigate that error in your own way. **Odometry** data will mitigate some of these errors, but not all of them because Jackals wheels slide and this produces errors in the odometry. Bear that in mind.
- GPS gives you no orientation: GPS only gives you a point in space, no orientation. This means that you need to complement that with a **compas** of some kind. Here is where **IMU** comes in handy.

With all this in mind, you will be able to have a crude but usefull system to position and move your robot around an outdoors terrain without map. Of course, you can move around with GPS and while doing it, generate a new map, making bigger the area where map localization (much more precise) can be used next time.

Get The GPS navigation running

Step 0: Load an empty Map

Because we want to have the map, but we are going to move around based on the GPS, we load into the map server an empty map. For that just male a copy of the map you created in Unit1 and remove all the black areas leaving a totaly white (which means empty) map.



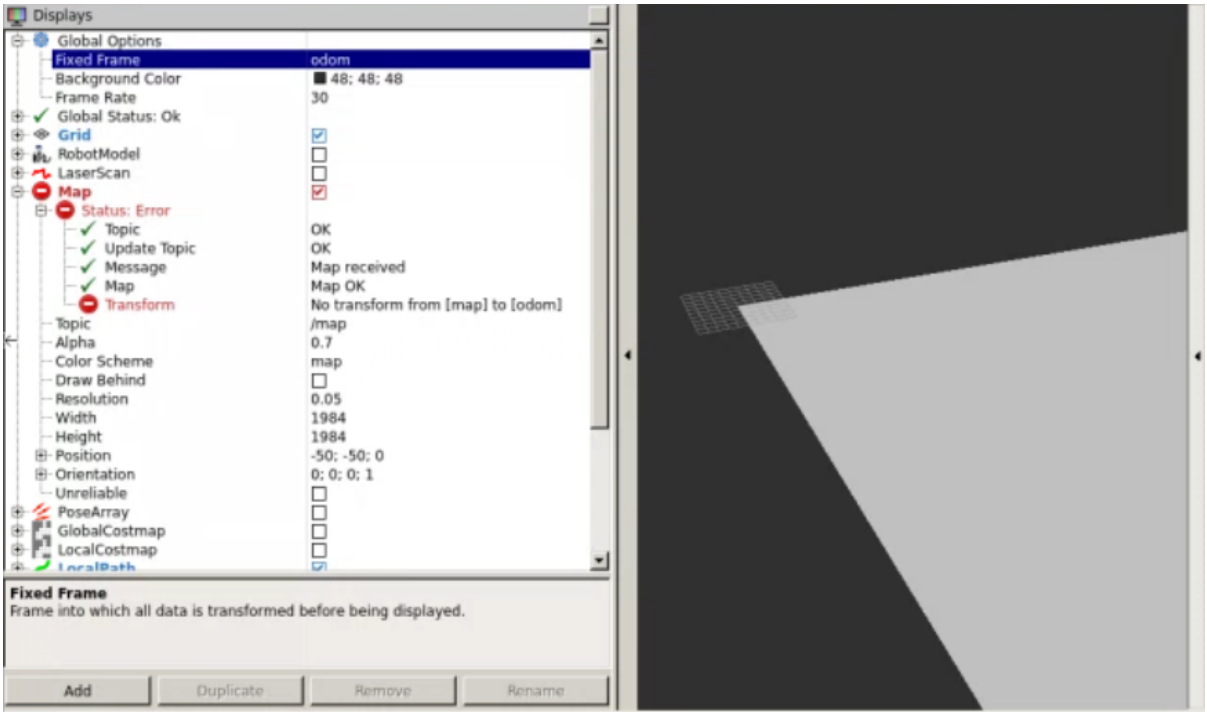
Create a map server launch to be able to test that works:

start_map_server.launch

In [ ]:

```
<launch>
    <node name="map_server" pkg="map_server" type="map_server" args="$(find my_jack
</launch>
```

Now the map server will load the new map into the param server and it should be seen in RVIZ. When you launch the map server you should see something like this:



Step 1: Disconnect AMCL

Of course the first thing you have to do retrieve the **map based navigation launch** which if you followed the named conventions in this course, it should have a name like **start_navigation_with_map.launch**. Create a copy of it with the name **start_navigation_with_gps_ekf.launch**.

start_navigation_with_gps_ekf.launch

In [ ]:

```
<launch>

    <!-- Run the map server -->
    <include file="$(find my_jackal_tools)/launch/start_map_server.launch" />

    <!--- Run Move Base -->
    <include file="$(find my_jackal_tools)/launch/with_map_move_base.launch" />


    <!-- Start RVIZ for Localization -->
    <include file="$(find my_jackal_tools)/launch/view_robot.launch">
        <arg name="config" value="localization" />
    </include>

</launch>
```

As you can see you only have to remove the **amcl.launch** and add the launch of RVIZ with the file **localization.rviz**. The rest is exactly the same.

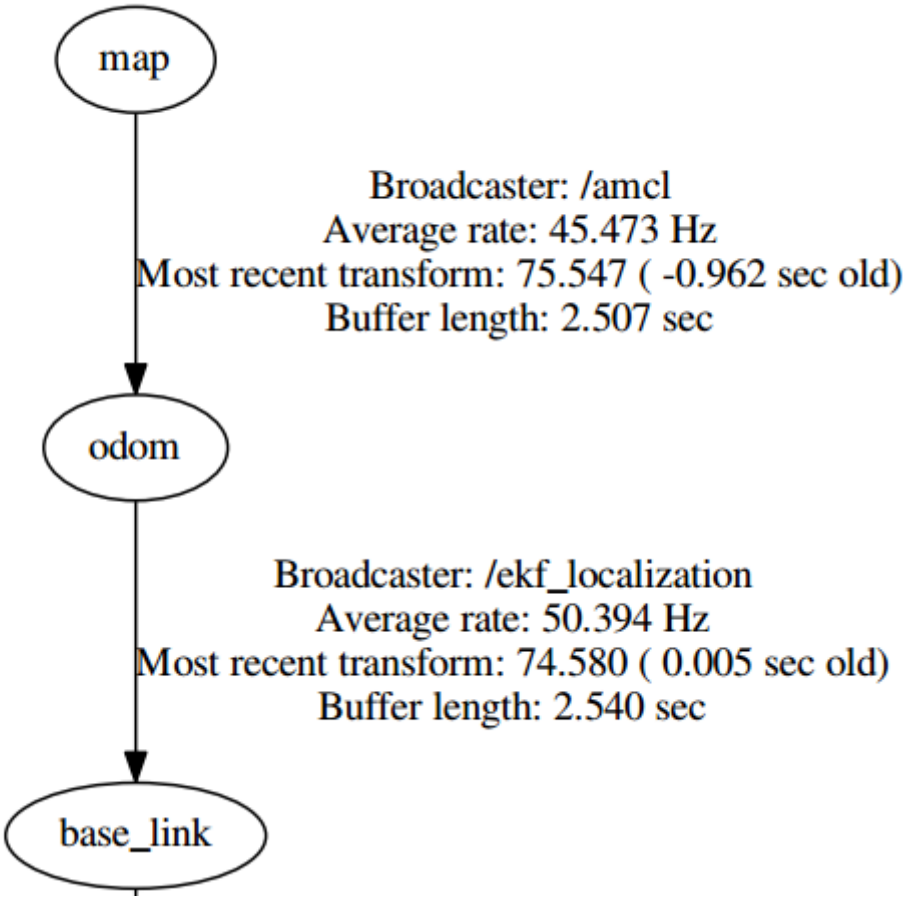
with_map_move_base.launch

In [ ]:

```
<launch>

  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="s

    <rosparam file="$(find my_jackal_tools)/params/costmap_common_params.yaml" comm
    <rosparam file="$(find my_jackal_tools)/params/costmap_common_params.yaml" comm

    <rosparam file="$(find my_jackal_tools)/params/map_nav_params/local_costmap_par
    <rosparam file="$(find my_jackal_tools)/params/map_nav_params/global_costmap_pa

    <rosparam file="$(find my_jackal_tools)/params/base_local_planner_params.yaml"
    <rosparam file="$(find my_jackal_tools)/params/move_base_params.yaml" command="

    <param name="base_global_planner" type="string" value="navfn/NavfnROS" />
    <param name="base_local_planner" value="base_local_planner/TrajectoryPlannerROS

    <remap from="odom" to="odometry/filtered" />
  </node>

</launch>
```

So what implication has removing **amcl.launch**?
Well the main function of **AMCL** is to publish the transform of **/map** frame to **/odom** frame.
Check out this frame tree with **amcl** running:

TF tree with amcl running in the navigation with map (extra_files/frames_with_amcl_map_to_odom.pdf)



Here you see that the node responsible of conecting **map** frame and **odom** frame is **AMCL**. So how will you do it now?
Have a look at **who is connecting odom to base_link**. Its a node called **ekf_localization**. That node is part of a package called **robot_localization** that merges sensor data through Kalman Filters to publish more stable TFs. You are going to use it to publish also the map --> odom TF with GPS, IMU and odom data merged.

Step2: Convert the GPS data into Odometry type data

Introducing the ROS package robot_localization (http://docs.ros.org/kinetic/api/robot_localization/html/index.html).
This package allows you to localise a Robot based on infinite number of inputs. You can merge the data from IMU, GPS, Vision.. To get the best localization through Kalman Filters.
You are going to learn the basics of how to set it up and use some extra tools GPS related like navsat_transform_node (http://docs.ros.org/kinetic/api/robot_localization/html/integrating_gps.html).

The **navsat_transform_node** alows you to convert the GPS **latitude** and **longitude** into **XY** coordinates that can be represented in space. Here is the example of the launch file you will need to launch it:

start_navsat.launch

In [ ]:

```
<launch>
 <!-- -->
  <node pkg="robot_localization" type="navsat_transform_node" name="navsat_transfor

    <param name="magnetic_declination_radians" value="0"/>
    <param name="yaw_offset" value="0"/>
    <param name="zero_altitude" value="true"/>


    <param name="broadcast_utm_transform" value="false"/>
    <param name="publish_filtered_gps" value="false"/>


    <param name="use_odometry_yaw" value="false"/>
    <param name="wait_for_datum" value="false"/>


    <remap from="/imu/data" to="/imu/data" />
    <remap from="/gps/fix" to="/navsat/fix" />
    <remap from="/odometry/filtered" to="/odom" />

  </node>

</launch>
```

Some element to comment on:

In [ ]:

```
<param name="magnetic_declination_radians" value="0"/>
<param name="yaw_offset" value="0"/>
<param name="zero_altitude" value="true"/>
```

These paraemeters are just to adjust some offsets you might have in your sensors IMU and so on. You also can consider Null the altitude.

In [ ]:

```
<param name="broadcast_utm_transform" value="false"/>
<param name="publish_filtered_gps" value="false"/>
```

- broadcast_utm_transform: This is in case you want to publish the static TF of the UTM (Origin of the GPS system) in the TF tree.
- publish_filtered_gps: No mystery here, if you want it to publish an improved by odometry and IMU GPS data.

In [ ]:

```
<param name="use_odometry_yaw" value="false"/>
<param name="wait_for_datum" value="false"/>
```

- use_odometry_yaw: This is for when you have a very reliable odometry turning system. In case of robots outdoors , the turning data in odometry is reliable enough because they skid once in a while , specially in rough terrain, thats why its turned OFF here.

- wait_for_datum: This is to give the system directly the datum, what is considered as the origin of our GPS localization. Normally its the GPS that directly gives this data and thats why its normaly turned off. If you need to give it for soe reason, you will have to add in this launch the following param setting:

In [ ]:

```
<rosparam param="datum">[55.944904, -3.186693, 0.0, map, base_link]</rosparam>
```

Or through the service **set_datum service and using the robot_localization/SetDatum service message**.

In [ ]:

```
<remap from="/imu/data" to="/imu/data" />
<remap from="/gps/fix" to="/gps/fix" />
<remap from="/odometry/filtered" to="/odom" />
```

This is the most important parameters. These are the topics from which it gets the GPS, IMU and Odometry data. With it it generates the Pose equivalent.

How to know what GPS data means and where it is in real life?
Well in the case of this simulated GPS, it starts located in the coordenates: latitude = 49.9, and longitude = 8.9.
In real life ofcourse it would have the readings of the real place.
You can get the conversion from address to lat-longitude and the otherway round in this Google page (https://www.gps-coordinates.net/).

Once you have this launch ready, when launched, it will publish into the topic /odometry/gps.

Execute in WebShell #1

In [ ]:

```
roslaunch my_jackal_tools start_navsat.launch
```

Execute in WebShell #2

In [ ]:

```
rostopic info /odometry/gps
```

As you can see here, the message type is nav_msgs/Odometry

In [ ]:

```
## nav_msgs/Odometry

std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

As you can see,its a standard Odometry message. This means that now you have the position of the Jackal based only on its GPS, Odometry and IMU. So the obvious step now is convert this data into a TF that positions the robot in the correct spot.

Create TF publisher for MAP to Odom frame using EKF node

So now that you have a new source of odometry information based on the GPS, its time to merge that data with extra data from IMU and the classical odometry, to generate the TF **map -> odom**.

The first step is knowing that in this simulation the **efk** is already running to publish the **odom** to **base_link** TF. The only thing that you have to do is to start another instance of **ekf_localization_node** that publishes the **map** to **odom** TF.

Take a look on the one running to have a basic idea of what it could be doing and how it works:

Execute in WebShell #1

In [ ]:

```
rosnode list | grep ekf
```

You will get:

In [ ]:

```
user:~/catkin_ws/src$ rosnode list | grep ekf
/ekf_localization
```

You can see what the original **ekf_localization** is subscribed to and publishing

Execute in WebShell #1

rosnode info /ekf_localization

In [ ]:

```
user:~/catkin_ws/src$ rosnode info /ekf_localization
--------------------------------------------------------------------------------
Node [/ekf_localization]
Publications:
 * /odometry/filtered [nav_msgs/Odometry]
 * /tf [tf/tfMessage]
 * /diagnostics [diagnostic_msgs/DiagnosticArray]
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /tf [tf/tfMessage]
 * /clock [rosgraph_msgs/Clock]
 * /set_pose [unknown type]
 * /jackal_velocity_controller/odom [nav_msgs/Odometry]
 * /tf_static [tf2_msgs/TFMessage]
 * /imu/data [sensor_msgs/Imu]
```

As you can see it reads from sonsor topics like : **/jackal_velocity_controller/odom** (odometry) and **/imu/data** (IMU).
It the publishes : **/odometry/filtered** (merged odometry from basic odometry and imu) and **/tf** (where the TF odom to base_link is published).

Now have a look at the updated version of the **start_navigation_with_gps_ekf.launch**

UPDATED **start_navigation_with_gps_ekf.launch**

In [ ]:

```
<launch>
    <!-- Run navsat gps to odometry conversion-->
  <include file="$(find my_jackal_tools)/launch/start_navsat.launch" />


    <!-- Run the ekf for map to odom config -->
    <node pkg="robot_localization" type="ekf_localization_node" name="ekf_localizat
    <rosparam command="load" file="$(find my_jackal_tools)/config/robot_localizatio
  </node>


    <!-- Run the map server -->
    <arg name="map_file" default="$(find my_jackal_tools)/maps/mymap_empty.yaml"/>
    <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file

    <!--- Run Move Base -->
    <include file="$(find my_jackal_tools)/launch/with_map_move_base.launch" />


    <!-- Start RVIZ for Localization -->
    <include file="$(find my_jackal_tools)/launch/view_robot.launch">
        <arg name="config" value="localization" />
    </include>

</launch>
```

So concentrate in the part reffered to **ekf_localization**:

In [ ]:

```
<!-- Run the ekf for map to odom config -->
    <node pkg="robot_localization" type="ekf_localization_node" name="ekf_localizat
    <rosparam command="load" file="$(find my_jackal_tools)/config/robot_localizatio
  </node>
```

EKF needs to have a different nae from the one already running , otherwise ROS wont let you launch it for obvious reasons. Thats why this instance of this node is called **ekf_localization_with_gps**.

This node need to load a bunch of parameters that will configure how many sensors are used to calculate the TF map --> odom. This is loaded through the file **robot_localization_with_gps.yaml**.

robot_localization_with_gps.yaml

In [ ]:

```
#Configuation for robot odometry EKF
#
frequency: 50

odom0: /jackal_velocity_controller/odom
odom0_config: [false, false, false,
               false, false, false,
               true, true, true,
               false, false, true,
               false, false, false]
odom0_differential: true

imu0: /imu/data
imu0_config: [false, false, false,
              true, true, true,
              false, false, false,
              true, true, true,
              false, false, false]
imu0_differential: false

odom1: /odometry/gps
odom1_config: [false, false, false,
               false, false, false,
               true, true, true,
               false, false, true,
               false, false, false]
odom1_differential: false

odom_frame: odom
base_link_frame: base_link
world_frame: map
map_frame: map

process_noise_covariance": [0.05, 0,    0,    0,    0,    0,    0,     0,     0,
                            0,    0.05, 0,    0,    0,    0,    0,     0,     0,
                            0,    0,    0.06, 0,    0,    0,    0,     0,     0,
                            0,    0,    0,    0.03, 0,    0,    0,     0,     0,
                            0,    0,    0,    0,    0.03, 0,    0,     0,     0,
                            0,    0,    0,    0,    0,    0.06, 0,     0,     0,
                            0,    0,    0,    0,    0,    0,    0.025, 0,     0,
                            0,    0,    0,    0,    0,    0,    0,     0.025, 0,
                            0,    0,    0,    0,    0,    0,    0,     0,     0.04,
                            0,    0,    0,    0,    0,    0,    0,     0,     0,
                            0,    0,    0,    0,    0,    0,    0,     0,     0,
                            0,    0,    0,    0,    0,    0,    0,     0,     0,
                            0,    0,    0,    0,    0,    0,    0,     0,     0,
                            0,    0,    0,    0,    0,    0,    0,     0,     0,
                            0,    0,    0,    0,    0,    0,    0,     0,     0,


initial_estimate_covariance: [1e-9, 0,    0,    0,    0,    0,    0,    0,    0,
                              0,    1e-9, 0,    0,    0,    0,    0,    0,    0,
                              0,    0,    1e-9, 0,    0,    0,    0,    0,    0,
                              0,    0,    0,    1e-9, 0,    0,    0,    0,    0,
                              0,    0,    0,    0,    1e-9, 0,    0,    0,    0,
                              0,    0,    0,    0,    0,    1e-9, 0,    0,    0,
                              0,    0,    0,    0,    0,    0,    1e-9, 0,    0,
                              0,    0,    0,    0,    0,    0,    0,    1e-9, 0,
                              0,    0,    0,    0,    0,    0,    0,    0,    1e-9,
```

```
      0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,
```

This node ekf has many more parameters that can be set and for sure you will be able to upgrade the performance of the system by tuning the values. These are the basic:

Essentially what you have to set up is all the sources of data that will be merged to generate th TF. In this case you are going to use:

- Basic Jackal Wheel Odometry: this is published in the topic **/jackal_velocity_controller/odom**.
- Jackals IMU: this data isn published in the topic **/imu/data**.
- Odometry generated by the GPS to Odometry conversion made by the navsat_transform_node: This data will be published in the topic **/odometry/gps**.

All this data is set by stating the type of input (odom or imu), followed by a number. Thats why you have **odom0** and **odom1**, but **imu0** only. There you state the topic to read from.

Then you state the configuration. This indicates what data you will take into account an which one you will ignore.

odomX_config: [true, true, false, --> x, y, z Position Values

false, false, false, --> roll, pitch, yaw Orientation values

false, false, false, x linear-velocity, y linear-velocity, z linear-velocity

false, false, true, roll velocity, pitch velocity, yaw velocity

false, false, false] x accel., y accel., z accel.

In this example, you are stating that you take into account:

- X and Y Position.
- Pitch velocity

What you consider depend on whether that data is more or less reliable or not. In the case of your file **robot_localization_with_gps.yaml** you state that:

- In the **odometry sensors** you only consider the **linear velocities** and ignore the rest.
- In the case of the **imu** you only consider all the Roll-Pitch-Yaw orientation data (orientation and speed).

As for the **odomX_differential** , you state False if you want to consider the values as they are, or you consider only their diference in time. Its advisable that you combine both if you have more than one sensor to have more stable readings. You normally put the one thnat has maybe a constant error in differential = True to remove that error.

Exercise U2-1

Implement all the launch files and configuration files to be able to launch it and have Jackal appera in RVIZ, fully localised and with the global and local map working. It should be able to move around using **2D Pose** signals in RVIZ. Just like using map navigation.

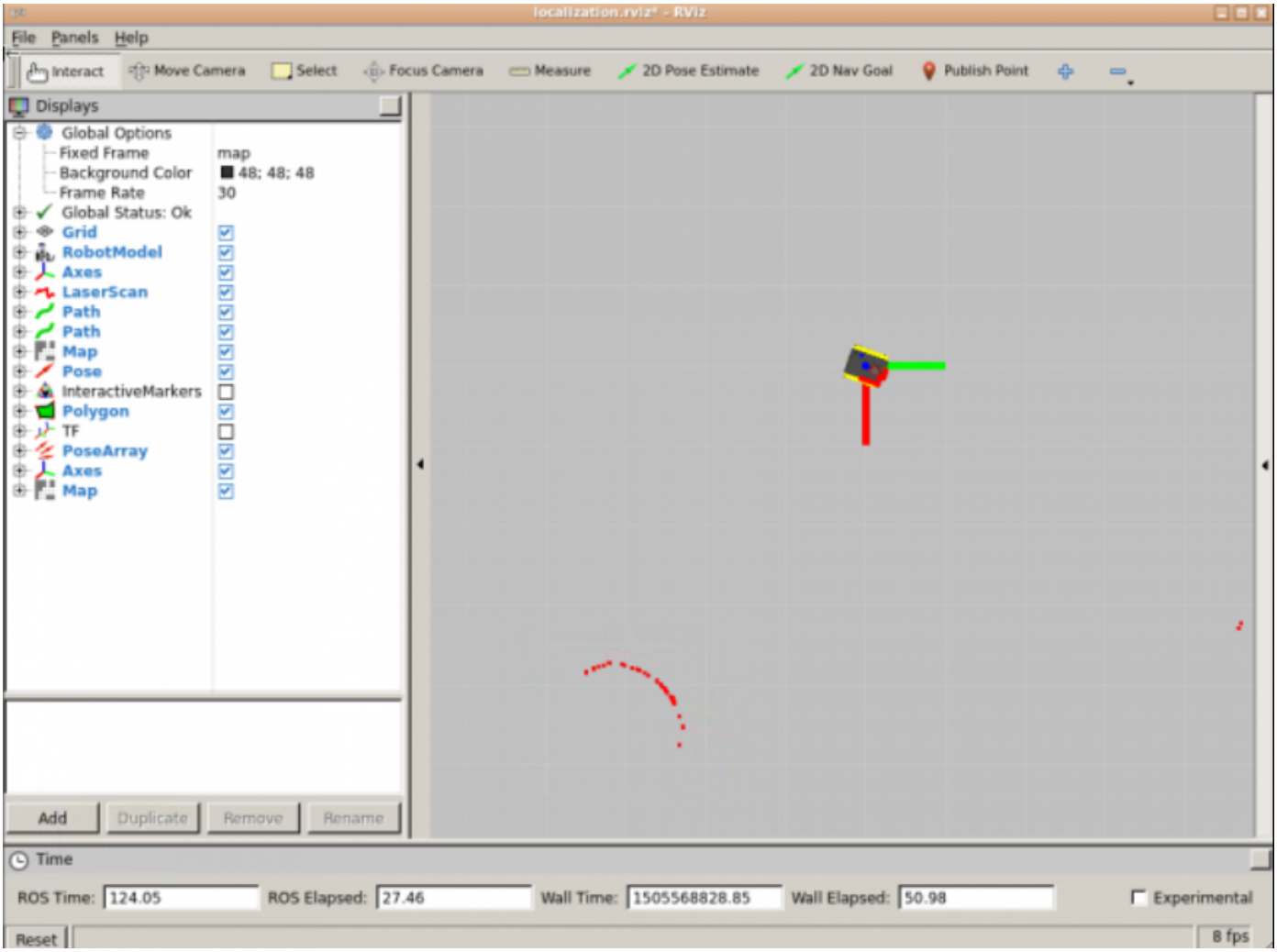
Steps to follow:

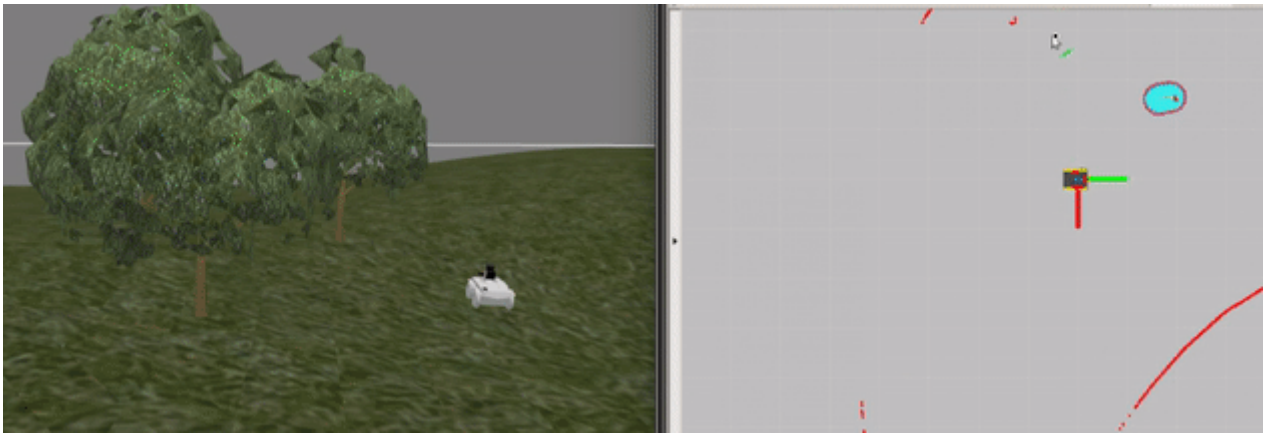
- Create the **start_map_server.launch** and check that the empty map you created is ok.
- Create the **start_navsat.launch** and check it works and makes the conversion. Move the Jackal robot around to see thet the values change accordingly.
- Create Updated version of **start_navigation_with_gps_ekf.launch** with the included ekf launching and navsat.
- Launch it and check that Jackal can navigate.

Bare in mind that this outdoors environment has hills and trees, so check out how the navigation performs when detecting these obstacles.

END Exercise U2-1

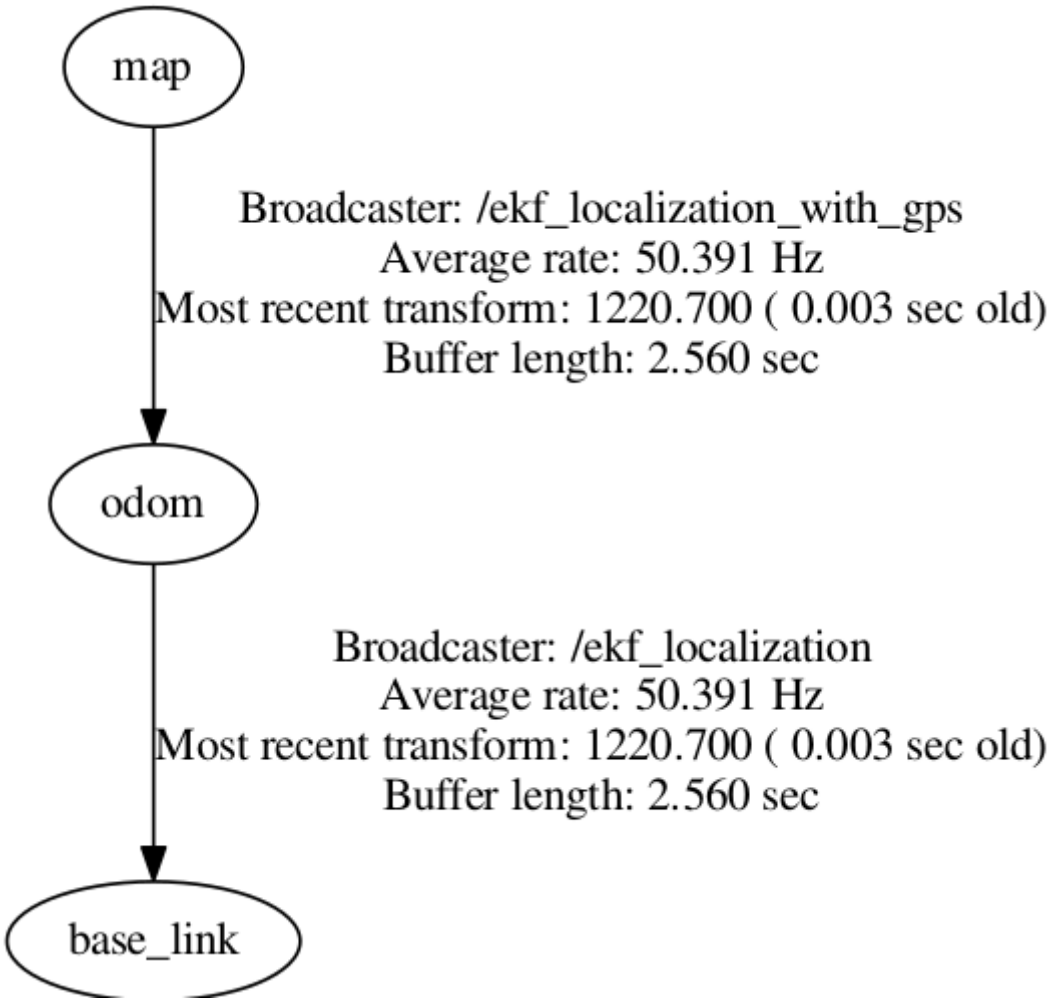
You should get something similar to this in RVIZ:





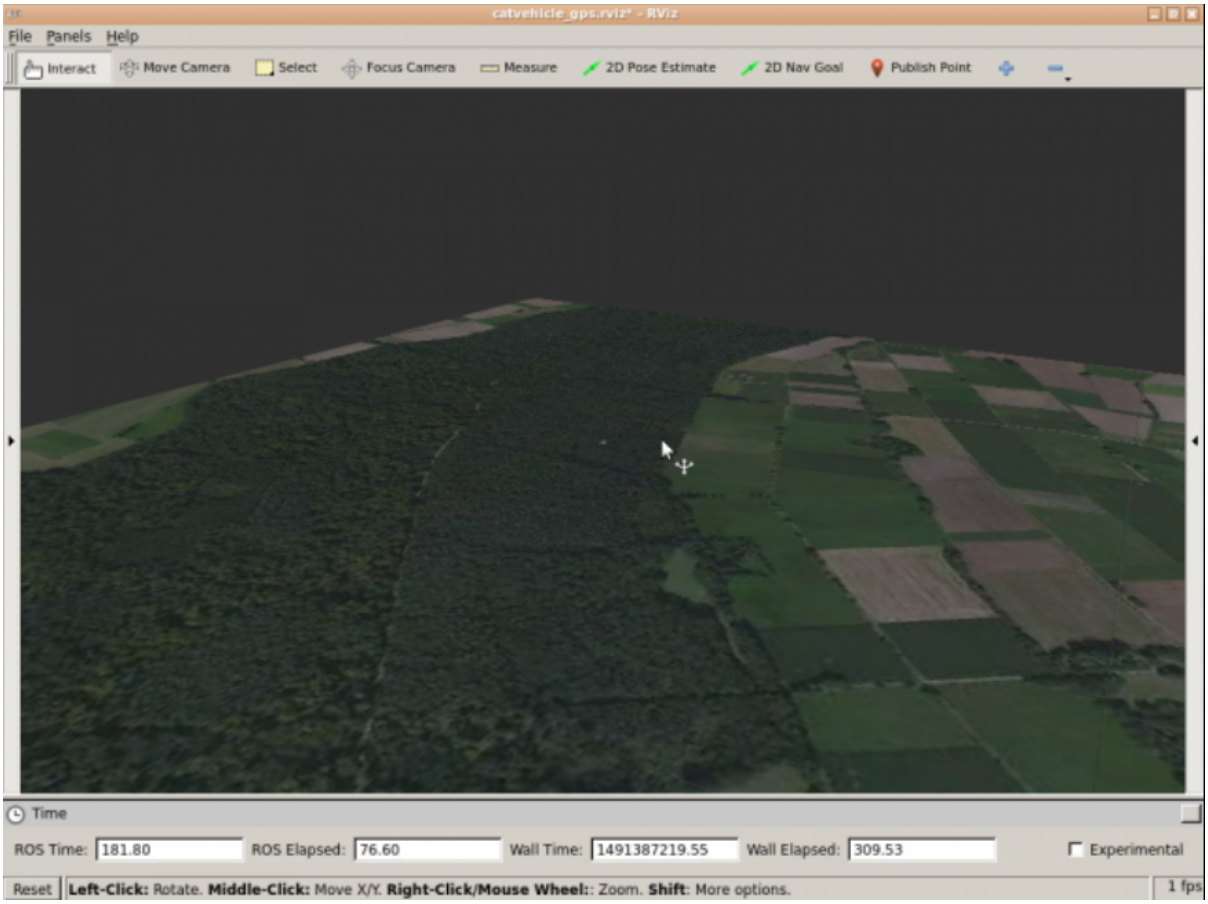
Now check the TF tree that you have. You should see that the **map** to **odom** is published by your **ekf_localization_with_gps** node.

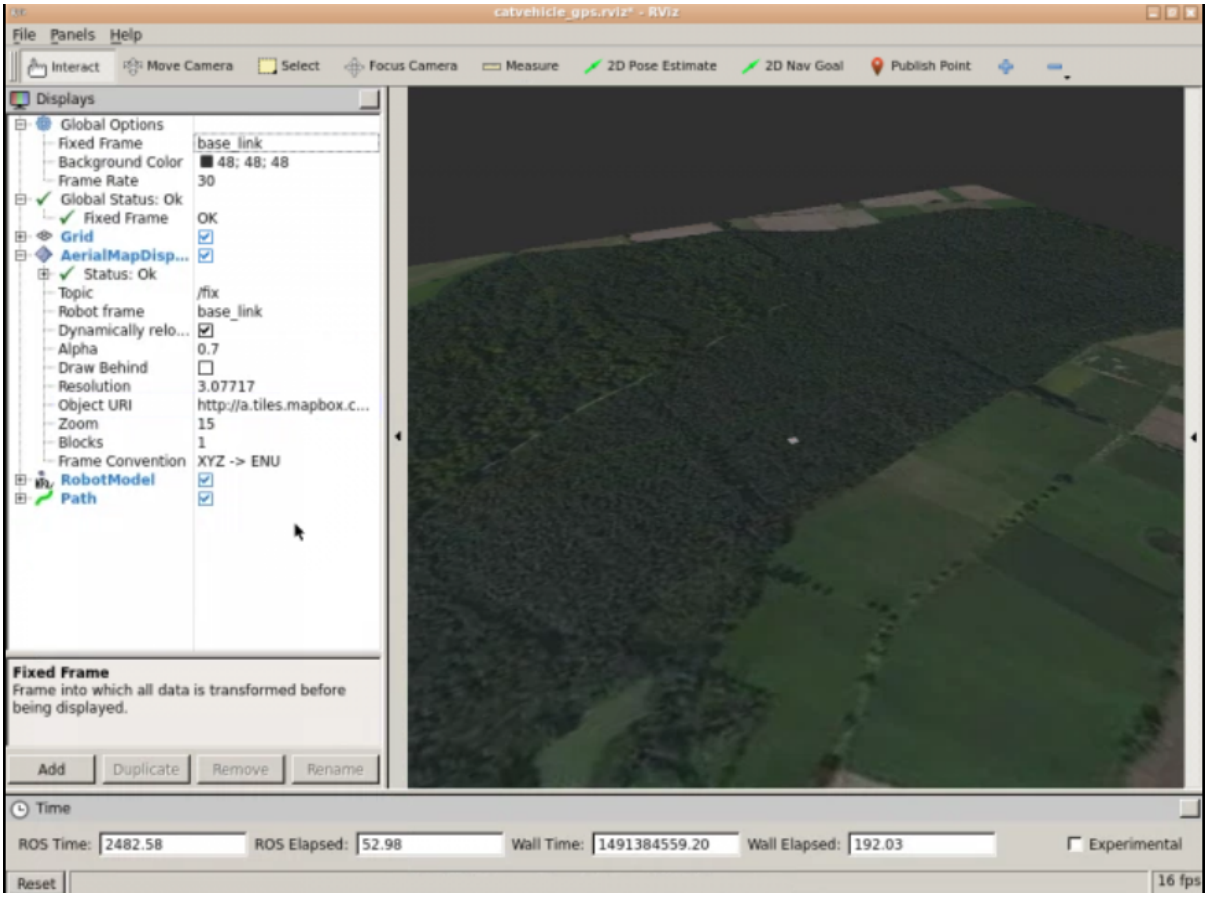
TF tree with ekf running in the navigation with empty map (extra_files/frames_with_ekf_map_to_odom.pdf)



Represent the MAP position in RVIZ

Because you have now GPS data, you can represent the robot in the real place where the GPS is positioning it. Getting results like:





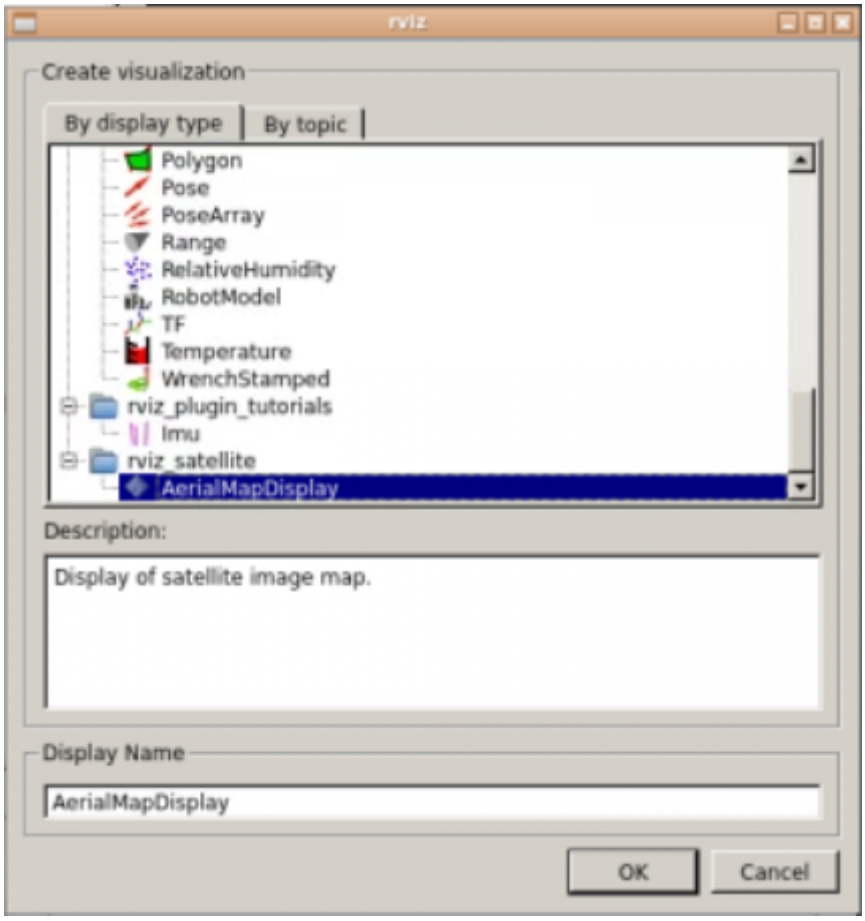
But this is not what you should see, because if you retrieve the address from the Google page based on the origin latitude and longitude (olat = 49.9 olon = 8.9), you should get a map similar to the one shown in Google maps:

So to get this in RVIZ you have to:

- Represent the GPS Current Position in RVIZ using a plugin: This plugin allows you to position the summit XL based in the GPS data published in /gps/fix topic in the corresponding real place on Earth https://github.com/gareth-cross/rviz_satellite (https://github.com/gareth-cross/rviz_satellite).

  You can copy a rviz file that is already set for you in The path is **/home/simulations/public_sim_ws/src/all/jackal/jackal_tools/jackal_tools/rviz/gps.rviz**

- You can set it manually as follows:



Then set the topic where the GPS data is published (For example: /gps/fix) , the RobotFrame (for example /odom)and set the URL where to fetch the MapTiles. This URL has to have the following structure: http://a.tiles.mapbox.com/v4/mapbox.satellite/{z}/{x}/{y}.jpg?access_token=YOUR_ACCESS_TOKEN (http://a.tiles.mapbox.com/v4/mapbox.satellite/%7Bz%7D/%7Bx%7D/%7By%7D.jpg?access_token=YOUR_ACCESS_TOKEN)
The Only thing that has to be changed is the YOUR_ACCESS_TOKEN which its given by MapBox https://www.mapbox.com/install/js/cdn-add/ (https://www.mapbox.com/install/js/cdn-add/).

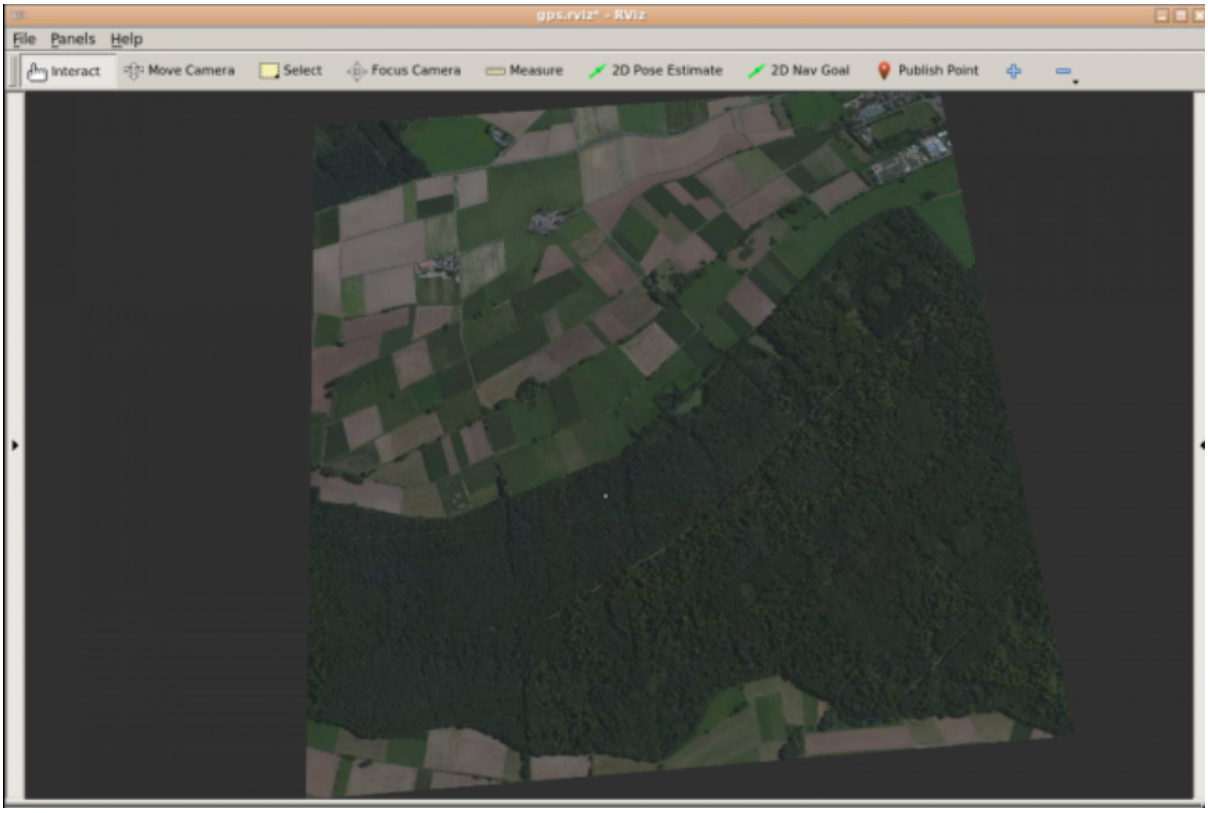
In this case the correct topic candidates for the GPS data are:

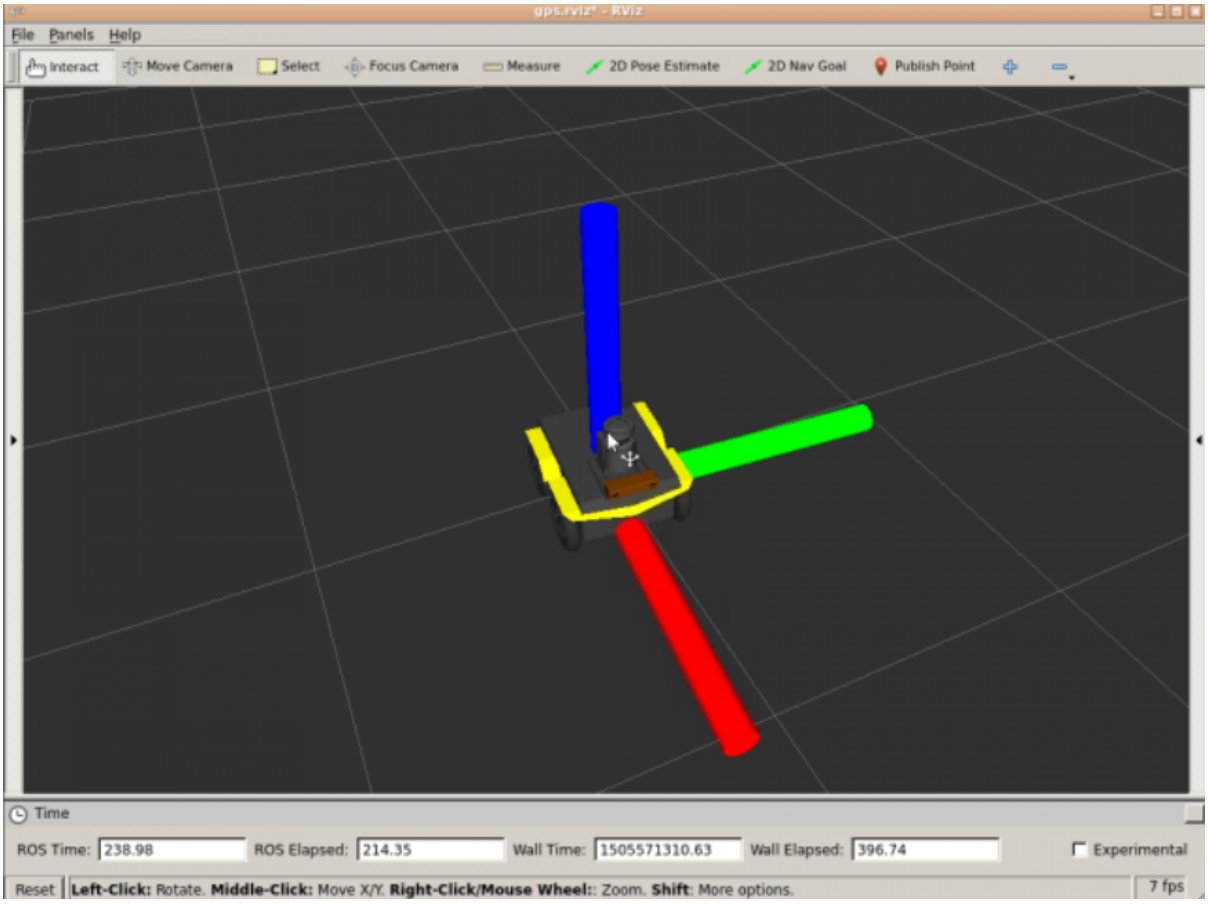
- /navsat/fix: This is the GPS data published directly by the GPS.
- /gps/filtered: This topic is published by the Nav_sat_stranform node. This means that is the GPS data published by **/navsat/fix** but merged with the odometry and imu data, so it should be more precise. So the logical choice has to be the **/gps/filtered**.

As for the Robot frame, **odom** is a good choice.

Also note that you have to select as **fixed_frame** in the global_options the **map** frame to see when the robot moves around the map.

If all went well you should see something like this:





Exercise U2-2

Try different settings for the RVIZ **AerialMapDisplay**, see how it might change its visualization.

END Exercise U2-2

How to convert GPS Coordinates to Pose Data

This skill is very important because you need to be able to sent poses to the move base to move the robot. But normally when you move outdoors, you have GPS coordenates to go to. So you have to use the geonav_transform (http://wiki.ros.org/geonav_transform) package.
This apcakge allows you to convert Latitude and longitude to XYZ coordinates, based on the GPS origon that you set. In this case the GPS origin is:

- olat = 49.9 olon = 8.9

gps_to_xyz.py

In [ ]:

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

# Import geonav tranformation module
import geonav_transform.geonav_conversions as gc
reload(gc)
# Import AlvinXY transformation module
import alvinxy.alvinxy as axy
reload(axy)
import rospy
import tf
from nav_msgs.msg import Odometry


def get_xy_based_on_lat_long(lat,lon, name):
    # Define a local orgin, latitude and longitude in decimal degrees
    # GPS Origin
    olat = 49.9
    olon = 8.9

    xg2, yg2 = gc.ll2xy(lat,lon,olat,olon)
    utmy, utmx, utmzone = gc.LLtoUTM(lat,lon)
    xa,ya = axy.ll2xy(lat,lon,olat,olon)

    rospy.loginfo("#########  "+name+"  ###########")
    rospy.loginfo("LAT COORDINATES ==>"+str(lat)+","+str(lon))
    rospy.loginfo("COORDINATES XYZ ==>"+str(xg2)+","+str(yg2))
    rospy.loginfo("COORDINATES AXY==>"+str(xa)+","+str(ya))
    rospy.loginfo("COORDINATES UTM==>"+str(utmx)+","+str(utmy))

    return xg2, yg2

if __name__ == '__main__':
    rospy.init_node('gps_to_xyz_node')
    xg2, yg2 = get_xy_based_on_lat_long(lat=49.9,lon=8.9, name="MAP")
    xg2, yg2 = get_xy_based_on_lat_long(lat=50.9,lon=8.9, name="MAP")
```

Note that you have different conversions, depending on the model used. You will use here the standard geonav transform **gn**. But you can als use the **AlvinXY** converter , which is a simple rectilinear conversion, or get the UTM values.

See the output of the two tests. Just changing in une unit in latitude, the efect in meters is huge. Obvious

because the distace between two lines of latitude is around **111 kilometers**. See that in AvinXY the result is exactly that: 111227.305636 meters or 111.227305636 Kilo metres. In the GN is not exactly that because is taking into acound more factors.

In [ ]:

```
[INFO] [WallTime: 1505572823.286398] [0.000000] #########  MAP  ###########
[INFO] [WallTime: 1505572823.286697] [0.000000] LAT COORDINATES ==>49.9,8.9
[INFO] [WallTime: 1505572823.286876] [0.000000] COORDINATES XYZ ==>0.0,0.0
[INFO] [WallTime: 1505572823.287080] [0.000000] COORDINATES AXY==>0.0,0.0
[INFO] [WallTime: 1505572823.287267] [0.000000] COORDINATES UTM==>492818.438732,552
[INFO] [WallTime: 1505572823.287644] [0.000000] #########  MAP  ###########
[INFO] [WallTime: 1505572823.287837] [0.000000] LAT COORDINATES ==>50.9,8.9
[INFO] [WallTime: 1505572823.289169] [0.000000] COORDINATES XYZ ==>149.529522407,11
[INFO] [WallTime: 1505572823.289821] [0.000000] COORDINATES AXY==>0.0,111227.305636
[INFO] [WallTime: 1505572823.290013] [0.000000] COORDINATES UTM==>492967.968254,563
```

Exercise U2-3

Create now a program that moves the robot to three different GPS positions in space. Steps to follow:

- Create a GPS to Pose conversor: This can be accomplished copying what you have just learned on how to use the GPS to coordinates conversor **gps_to_xyz.py**.
- Create a client of move_base : This client will be the one in charge of sending to the move_base server the poses. Make a system that allows it to know when it has reached its destination.

END Exercise U2-3

Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Here you have an example of move_base client that migh come in handy.

move_base_client.py

In [ ]:

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

import rospy
import time
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal, MoveBaseResult, MoveBa

"""
class SimpleGoalState:
    PENDING = 0
    ACTIVE = 1
    DONE = 2
    WARN = 3
    ERROR = 4

"""
# We create some constants with the corresponing vaules from the SimpleGoalState cl
PENDING = 0
ACTIVE = 1
DONE = 2
WARN = 3
ERROR = 4

"""
/move_base/goal
### PYTHON MESSAGE

rosmsg show move_base_msgs/MoveBaseGoal
geometry_msgs/PoseStamped target_pose
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w



/move_base/cancel
/move_base/cmd_vel
/move_base/current_goal
/move_base/feedback
"""

# definition of the feedback callback. This will be called when feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    rospy.loginfo(str(feedback))

# initializes the action client node
```

```
rospy.init_node('move_base_gps_node')

action_server_name = '/move_base'
client = actionlib.SimpleActionClient(action_server_name, MoveBaseAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = MoveBaseGoal()

goal.target_pose.header.frame_id = "/map"
goal.target_pose.header.stamp    = rospy.get_rostime()
goal.target_pose.pose.position.x = 0.0
goal.target_pose.pose.orientation.z = 0.0
goal.target_pose.pose.orientation.w = 1.0

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will be 1 if active
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2 if NO ERROR,
state_result = client.get_state()

rate = rospy.Rate(1)

rospy.loginfo("state_result: "+str(state_result))

while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result))

rospy.loginfo("[Result] State: "+str(state_result))
if state_result == ERROR:
    rospy.logerr("Something went wrong in the Server Side")
if state_result == WARN:
    rospy.logwarn("There is a warning in the Server Side")

#rospy.loginfo("[Result] State: "+str(client.get_result()))
```

Congratulations! You are now able to navigate with GPS to unmapped places.