Project 5: SLAM

In this project you will implement a (rather primitive) SLAM system and run it on a real robot to build a map of AVW 4th floor. The project will require you to create several packages - put them all in the folder ***<my_directory_id>_project_5*** (do not rename the packages themselves!), put your .bag files and report there as well and compress everything to .zip.

Most of the guidelines (as well as starter code) are designed for Python. C++ developers will get some additional extra credit (+20%, as usual) for their implementations.

DELIVERABLES:

- ***<my_directory_id>_project_5*** - folder with your packages
- *.bag* file(s) with a robot performing SLAM and map screenshots
- A short report which will describe your results

So until now we were fooling around with primitive packages running on a simulator, but SLAM requires more - SLAM (simultaneous localization and mapping) requires a fully configured robot - all transforms should be known, all sensors should be properly publishing and all modules thoroughly tested separately. Fortunately, there are some best practices which will allow us (you) to implement the project relatively painlessly. We will start somewhat from afar - we will create a thing called 'URDF model' of our robot to ensure all transformations are correct (how can you localize, if you do not even know where your wheels are?!).

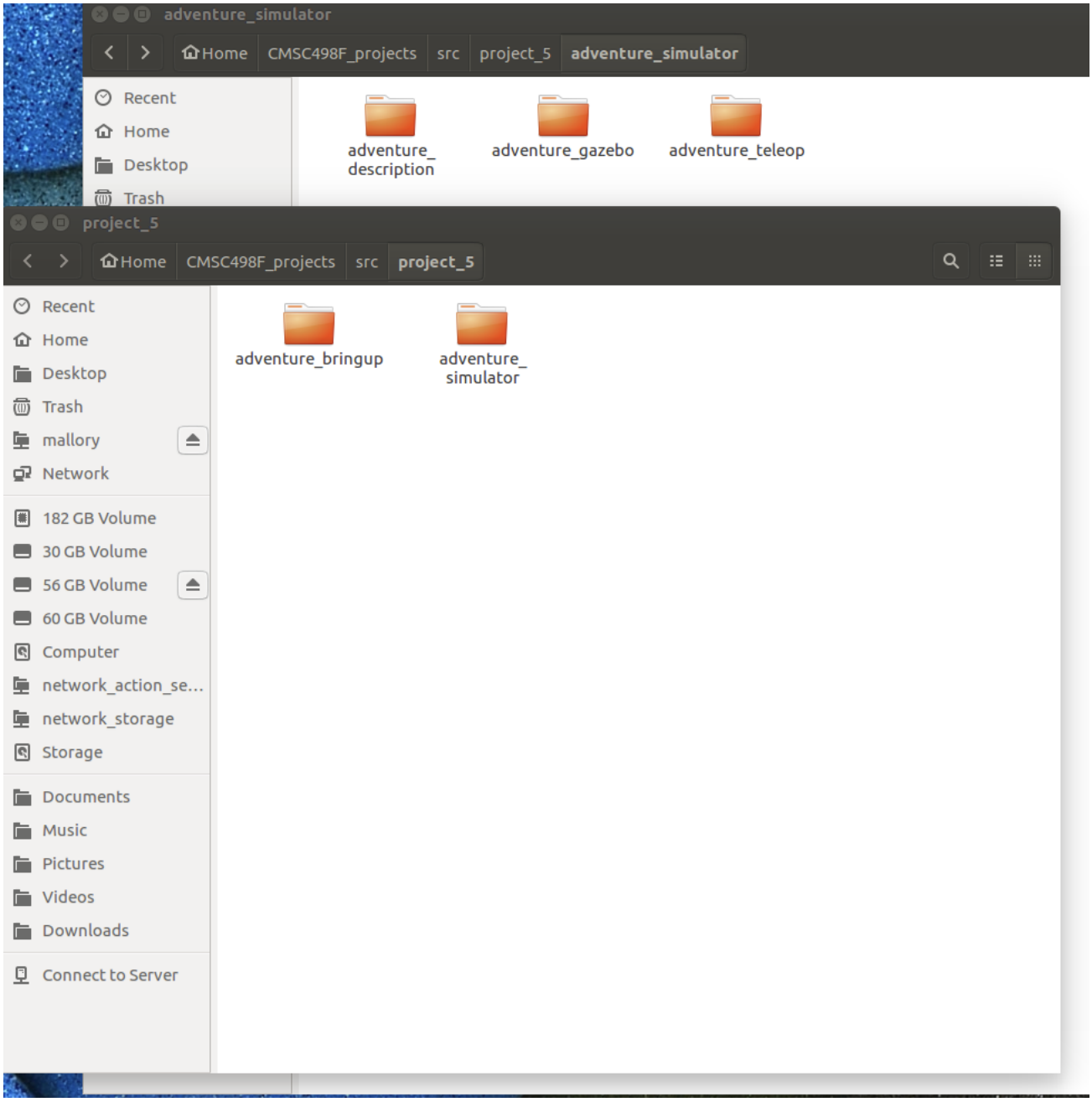
1. TF intro, URDF and robot setup!

URDF is a format which will allow you to virtually 'assemble' your robot out of pre-defined primitives. You will create a number of configuration files and 'load' them into ROS when you start the robot. As a result, ROS will be aware of all transforms between the different robot components - TF will automatically read URDF and start publishing them!

Wait, but we have already used TF before (in the previous project - remember */odom* frame?) and it worked - why do we need to change the URDF description now?! - That is right, but the problem is - we have used URDF from official TurtleBot (which was done for you) but our 'adventure' TurtleBots are custom - the camera position is different (actually, the camera is upside down…) and we have a gripper! Also, we have an additional camera *on* the gripper! TF worked well before because the position of the wheels with respect to the center of the robot was the same (we use the same 'kobuki' platform), but now this is not enough - we also need to know the correct position of the Asus Xtion RGB-D camera! That is why we need to have our own URDF description.

First of all, to efficiently maintain a large number of configuration files required for the robot startup, a certain layout is widely used by ROS developers. Everything will be described step-by-step later, but for a reference of style, layout and how the robot description should be done: link, link and link (the last link is an example of a poor layout, but it has the most complicated URDF description).

This is how your directory tree will approximately look like (there will be one more package for the slam):



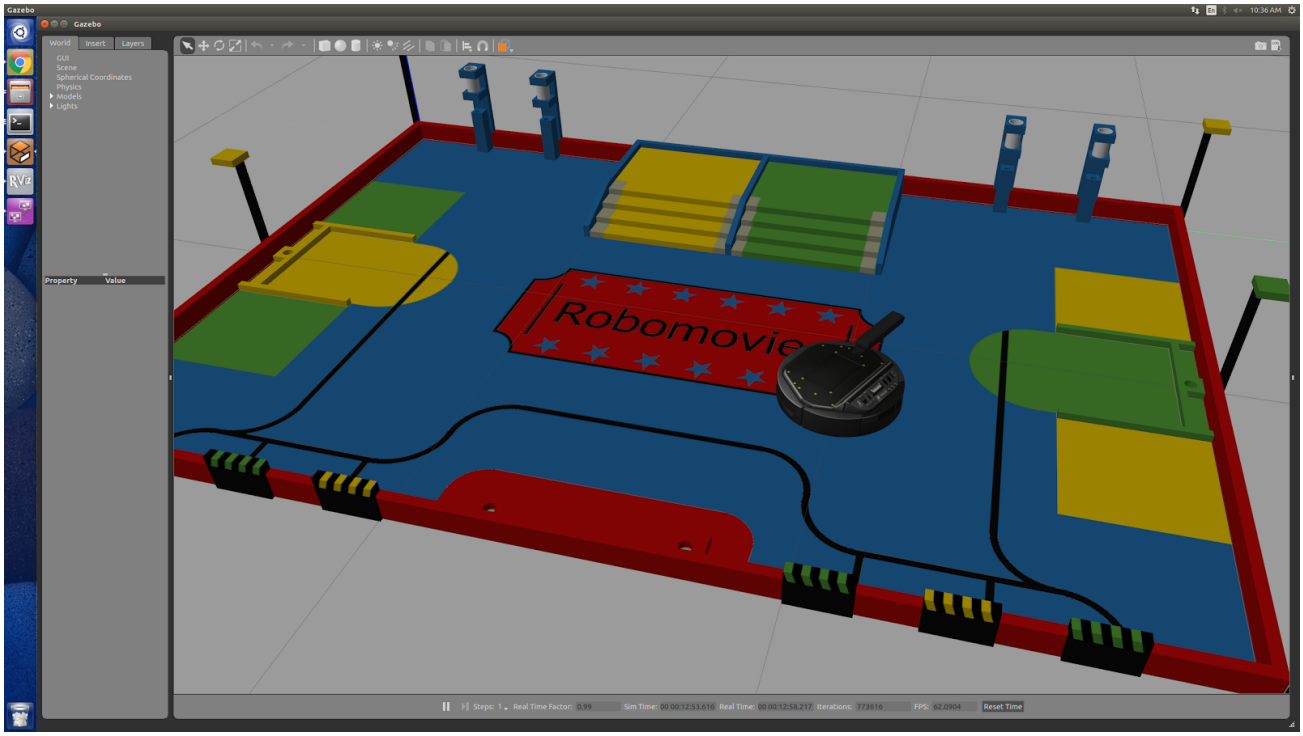
By the way, if you are still not comfortable with roslaunch and .launch files - this is the time to read up on them a little bit: link. There are also plenty of examples in the internet.

So let us begin!

1) Let's start from something simple - from the package named *'adventure_gazebo'*. This package is in charge of launching Rviz, Gazebo and the robot controllers. Examine carefully the 'launch' directory. Here everything is done for you, except for one thing - the map is wrong! The default one is from the Eurobot competition - this is a good example of how to add a custom map to your simulation, but this map is obviously bad for SLAM. Replace it with a Willow Garage map. If you want to see how the map looks like in the simulator (without your robot), run:

   roslaunch gazebo_ros willowgarage_world.launch

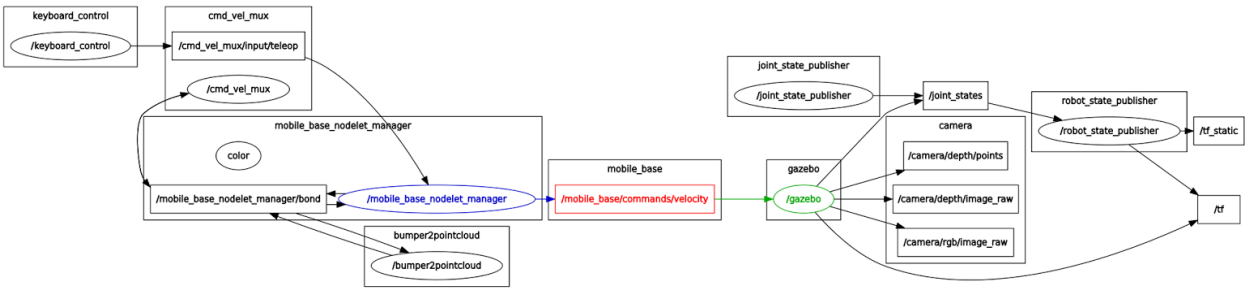
This is how the default map looks like. Note that the TurtleBot Xtion sensor is not in place:



2) Now we need to fix the *'adventure_description'* package. As you may already understand (by reading launch files from *'adventure_gazebo'*, especially the spawner launch file), the way simulation works - Gazebo will run, and then some *spawn_<smth>.launch* will read *.urdf.xacro* descriptions and load them into Gazebo to create a robot model there. The same spawner launch script will run some core nodes to publish sensor data and TF. Now launch

   roslaunch adventure_gazebo adventure_world.launch

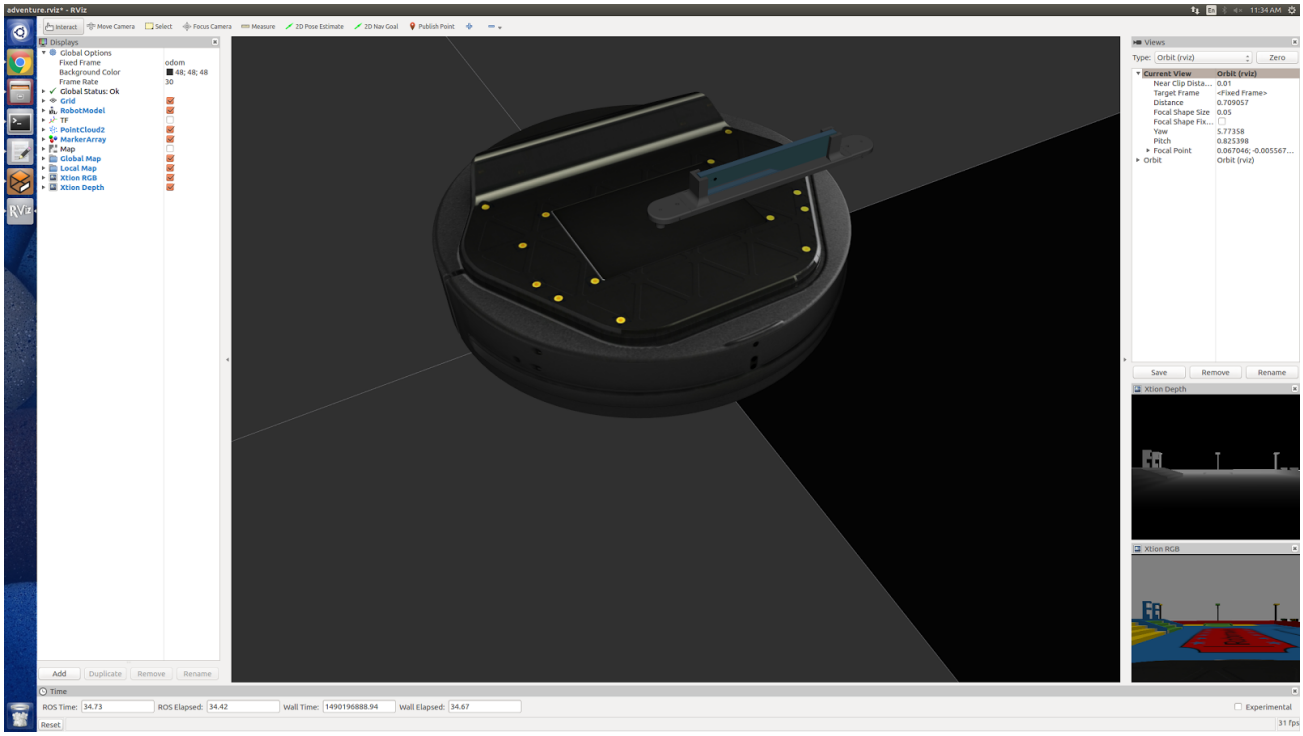
Try to understand how different nodes communicate with each other. Here is a picture to help:



The mobile base nodelet manager (blue) talks to a number of helper nodes, like 'keyboard teleop' or 'cmd_vel_mux' on the left side of the graph and then through the /mobile_base/commands/velocity (red) talks to Gazebo (green). The Gazebo produces a variety of simulated sensor outputs, like 'camera', 'tf' and 'joint_state_publisher' - you will subscribe to those topics later.

3) Ok, our problem is simple - the URDF model of our robot is incomplete! Particularly, the camera is wrong (it is RealSense R200, and we are using Asus Xtion) and the camera position is incorrect! Moreover, there is something terribly wrong with the Asus Xtion URDF file (which is provided to you), so you cannot put it in the desired location. Replace the camera, fix the error and place the camera properly (it should be exactly the same as on a real adventure-TurtleBot). **Do not** add any new URDF links!

This is what you are given initially - this guy has a wrong sensor in a wrong location. Fix it!



4) Once again, run

   roslaunch adventure_gazebo adventure_world.launch

   and make sure everything is correct. You are provided with a custom *'adventure_teleop'* package - try to control your robot in Gazebo, make sure Rviz shows sensor output.

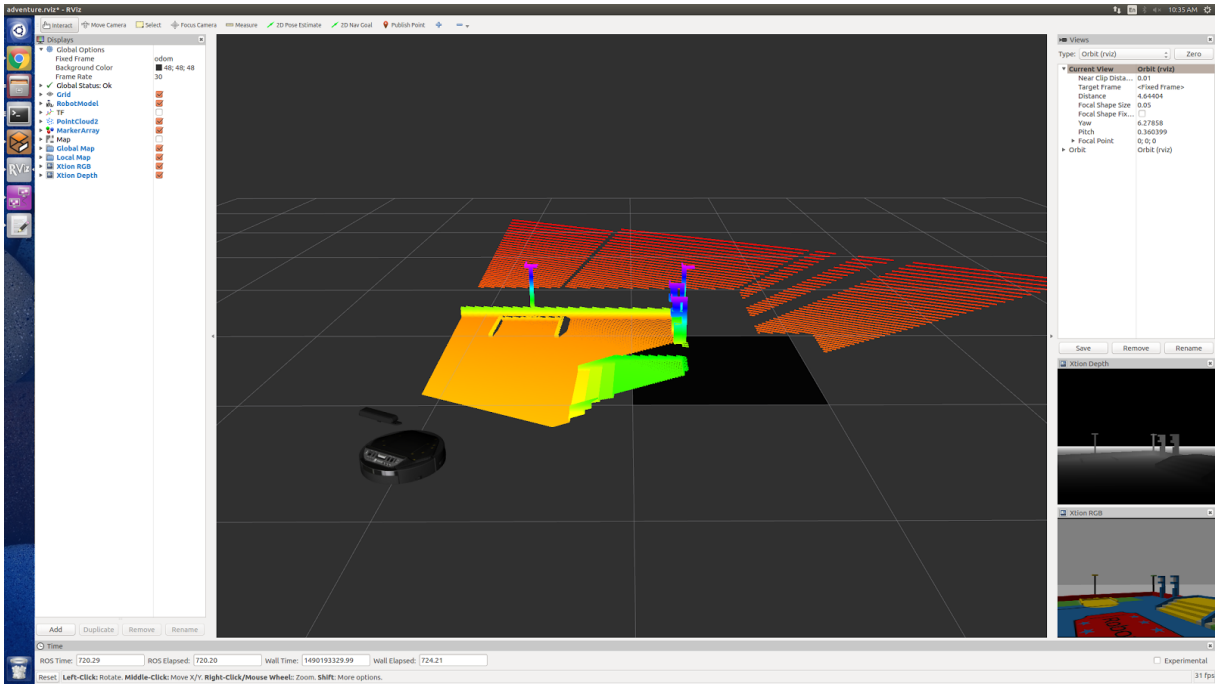
   roslaunch adventure_teleop adventure_teleop.launch

5) Now let's remember why we were doing all this stuff in the first place - to provide TFs for a SLAM on a real robot. From the ROS perspective, the only difference between a real robot and a simulation is that there is a robot driver running instead of Gazebo. Totally symmetrically to 'adventure_gazebo' there is an *'adventure_bringup'* package which runs the robot driver, as well as loads the URDF model into the parameter server to provide TF transforms. Fill in *mobile_base.launch.xml* , which is in charge of running hardware drivers (such as Xtion driver or Kobuki driver). To test everything, run:

   roslaunch adventure_bringup minimal.launch

   You should see real sensor output in Rviz, be able to teleoperate the robot, and the robot model in Rviz should look like the adventure-TurtleBot (and not the original one).

That is what you will see in Rviz (if you have not changed a map and/or configured the sensor position):



2. Line fitting

We will implement our SLAM in a very simple manner - we will assume that most of what we can see are walls which we can easily be approximated via lines. Our Xtion RGB-D sensor will provide us with a pointcloud which will be automatically converted (you do not have to implement this) to the 'laserscan' message by means of this node. So, obviously, the first step should be detecting these lines from the data in the 'laserscan' message and displaying what you have found in Rviz.

Now the first step is a little bit tricky - we want to use PCL for RANSAC line fitting - it is actually an overkill, but PCL is a really useful tool and you should be comfortable with using it. So let's install PCL!

a) Python
   - *sudo apt-get install libpcl-**
   - *sudo apt-get install ros-indigo-pcl-ros*
   - *sudo apt-get install cython*
   - *git clone https://github.com/strawlab/python-pcl*
   - *cd python-pcl*
   - *sudo python setup.py install*

b) C++
   - *sudo apt-get install libpcl-**
   - *sudo apt-get install ros-indigo-pcl-ros*

   For C++ that should be enough, but you will need to configure your *CMakeLists.txt* and *package.xml* properly - use this link as an example: link - this package is basically the implementation of this project, so if you can figure out how it works, feel free to copy pieces of code (This is to compensate for the lack of the starter code)! The most important file for you here is *lines.cpp*.

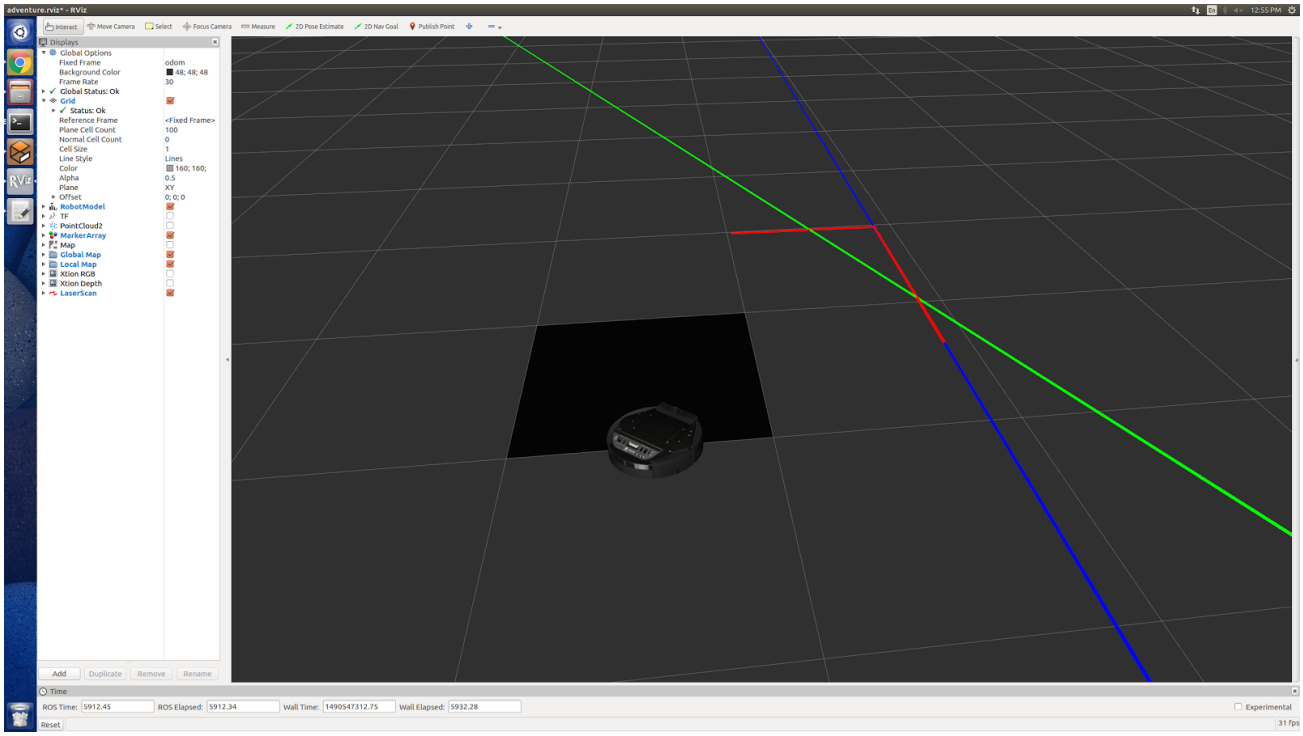
Do not forget to checkout this tutorial, as well as other cool PCL tutorials!

Your code should go to the *'adventure_slam'* package and there is some starter Python code provided to you. In contrast to other projects, this starter code is just an API demonstration and you will probably need to redesign the package considerably. Also please notice the usage of ROS markers - it is a huge help in the debugging process - you can see your lines in Rviz as they are fitted to the laserscan! There is a launch file which runs the simulation, Rviz, teleop and laserscan conversion for you:

roslaunch adventure_gazebo adventure_demo.launch

Later you will modify this file to also run your SLAM node.

The starter Python code (when run with Rviz) results in this picture. The blue line is the RANSAC line fitting. In addition, the green line shows the result of least square line fitting (which we will not use due to obviously poor results). The red line is the laserscan:



3. Making it clean!

Now you probably realize that we might have to tune a lot of parameters in the future in order to make our code work properly; previously we used the command line to pass arguments to the program but this time let's figure out how we can read the parameters from the launch files (that is the ROS way).

- Python users, read this; C++ users, read this. C++ users can also refer to the sample file (at the bottom) and both C++ and Python users can use this package as a roslaunch reference.
- Note that some parameters are specified directly in the *.launch* file, and some are in *.yaml* file. There is no difference in how and where the parameters are specified and both ways can be useful in different applications.

For this project you have to make launch files to run your nodes and you do not have to use *.yaml* files if you do not want to.

1) Create a *'launch'* directory in *'adventure_slam'* package and create a launch file which will run just your *adventure_slam* node with proper parameters.
2) Modify the *adventure_bringup/launch/adven ture.launch* and *adventure_gazebo/launch/adven ture_demo.launch* to *include* your slam launch file and (as a result) run your slam node on real robot and in simulation. Do not just copy the contents of the *.launch* file created in (1) - you need to include it!

4. Localization

Now that we can detect the lines on the laserscan (that is, we know where the walls are in respect to us), we can figure out our rotation and translation. Continue to work in '*adventure_slam*' package.

Math and geometry

1) The starter code is only able to detect one wall at a time, but you need all on them! This is easy - detect lines one by one and exclude the inliers of the current line from the laserscan points before moving on to the next line.
2) Ignore the lines which do not have enough inliers. You want to have a parameter in the launch/yaml file to control the minimum number (percentage) of inliers for the set of points to be considered a line.
3) Compare all the lines from the previous iteration with the lines detected on this iteration. If a pair of lines is almost parallel and the distance between them is small, consider the pair of lines as a match. Please, mind corner cases! If your line match does not work, you will get a very low grade for the whole project!
4) For each match compute the angle between corresponding lines. The average angle is going to be our rotation!
5) Consider the matches again, but now compute the shift of the robot in 2D space (be careful here, the lines are probably not parallel, so the distance between them is probably zero).

Infrastructure

In the future, we would like to use the robot_pose_ekf node - an extended Kalman filter which can fuse our wheel odometry and visual odometry. As a result, we need to publish *nav_msgs/Odome try* messages. We will also broadcast a tf transform to test our code without Kalman filter.

1) Here are tf tutorials (link) - you should be pretty familiar with tf at this point. All you need to do is to convert your rotation and shift to a tf message and to an Odometry message and then broadcast the tf and publish the Odometry message. Please, name your tf *'odom_visual'* and the message - *'vo'*.
2) Check how it works - launch the simulation and in Rviz choose *'odom_visual'* as fixed frame (Global Options -> Fixed Frame). Display the original *'odom'* frame. At first, the *'odom'* and *'odom_visual'* should align but then they might separate. Why could that happen?

5. Mapping

Probably the easiest part - the mapping should be done exactly as in the previous project (Project 4), but (as you might have noticed) we were not using transforms, we were using Odometry messages there. You

need to implement a separate node (there is no starter code for this) in the same *'adventure_slam'* package similar to the one in Project 4, but which will listen to the tf between either */odom* and */base_footprint* or */odom_visual* and */base_footprint* (Make that a parameter in the launch file) - you have listened to the tf before, in Project 3.

In the end we want to get something like hector_slam package:

6. Build some maps!

The result of this project should be two map screenshots and two .bag files.

1) Run your code in the simulator and create a map of the Willow Garage by teleoperating the turtlebot from the keyboard. Create a screenshot of a resulting map (from Rviz). You should also record the mapping process to the *.bag* file - follow this tutorial to learn how to use *.bag* files.

   rosbag record -a

2) Run your code on a real turtlebot (use your *'adventure_bringup'* package to load all the drivers) and do the same thing as you did in (1).
3) Do not forget to run your *.bag* files and verify that your mapping is reproducible!

Note, that *.bag* files can be very big - if you have problems with submitting them, you can upload those files to a file sharing service and submit a link.

7. Extra credit

- ***Extended kalman filter (50)***
  The problem of or current approach is that we need to see at least two non-parallel walls all the time in order to localize. Yet, we also have wheels! The wheel odometry information drifts over time, but it always gives us some information about how we move. In contrast, visual odometry (laser scan) does not drift, but sometimes does not give us any information. Can we 'fuse' those two sensors to make localization more robust? - Yes!

  The answer is the Kalman filter! Do not worry, I will not make you implement a Kalman filter. Instead, you will be using the standard ROS node *'robot_pose_ekf'* - but for 50 points, you will have to figure out how to plug it in yourself and you will have to tune it! Check out this tutorial and also this launch file. Note, that your odom (wheel and visual) will need to publish pose covariances in order for Kalman filter to work, and covariance for the visual odometry is not constant (depends on how much lines you see). See this file as an example (it is C++, but Python is the same!

  The deliverables for this part are - repeat part 6 with Kalman filter and show that the result is much better.

- ***Complete URDF model (30)***
  This part is actually rather easy - finish the URDF model of our turtlebot! Add the gripper, the web camera to the tip of the gripper and (maybe) the 3d model of Intel NUC or a Laptop. Do not forget to put the screenshots of your model into your report!

8. Grading

Robot Model	35 points
Replace the map	15 points
Fit the line with RANSAC	40 points
Launch files and parameters	10 points
Localization - geometry	30 points
Publish tf and odometry for localization	20 points
Map building	35 points
Create a map of AVW 4th floor and Willow Garage	15 points
Extra credit	80 points

NB! The TA will have to read your code and this is a big project, so pay attention to code quality and overall project organization. For each occurrence of 'dead code', unused variables or files you will lose 15 points. If you miss the guidelines or do not follow 'best practices' (in terms of organization), for each occurrence you will lose up to 30 points (depending on severity). If you are not sure about something (like 'what is code quality' or 'best practices') please, **do not** hesitate to ask your TA! Here are some coding guidelines: C++, Python.

Also keep in mind, that many parts of this project are interdependent, so if you fail one part, you may also lose points for other parts!

9. Github stuff

This part is totally optional, but it might save you a lot of time in this course and in the future. You are going to create a Github account and get access to the student pack to get free private repositories and other useful things!

Github is a version control system which is *extremely* useful for teamwork and code sharing! You can store your code for free on Github (without being afraid of losing it ever), see all changes you have ever made and revert them if you wish. When working in a team, Github provides a marvelous environment for code review and parallel development. All code hosted on Github is visible to everyone, so if you want to keep your stuff private you have to pay for a 'private repository' - this is done to encourage open source development! Yet, all students (people who have university-affiliated e-mail addresses) can get access to private repositories free of charge. Here is a step-by-step guide on how to get started with Github and get free private repositories (in case you need them).

1) Create an account on Github website (if you do not have one) - keep your username short and simple!
2) Go here and follow the instructions to get a free student pack (to get private repositories)
3) Check out this page to see what else you can get for free!
4) Follow these instructions to get started with Git: link, link, link
5) Fork our class repo and start developing your projects on the top of it!