TDDE05: Lab 2: Navigation

Cyrille Berger

February 4, 2020

The goal of this lab is to build a map of the environment and to use the map to plan a path for your robot.

1 Get the code for lab2

Get the skeleton code for lab2 from gitlab:

```
cd ~/TDDE05/catkin_ws/src/air_labs/
git pull --allow-unrelated-histories \
  https://gitlab.ida.liu.se/tdde05/air_labs.git master
```

2 TF

TF[1] is a package that allow to handle multiple coordinate frames over time, and their relationship. There is usually a root frame, corresponding to the origin of the world. In our labs, it is called `odom` (it can be sometime called `world` or `map`). All other frames need to express a transformation to `odom`, directly or indirectly.

On a robot, you usually have multiple frames:

- `husky0/base_footprint` corresponding to the center of all the contact points of the robot with the ground
- `husky0/base_link` corresponding to the center of gravity of the robot
- And one frame for each sensor:
  - `husky0/velodyne` corresponding to the velodyne laser
  - `husky0/imu` corresponding to the IMU sensor
  - ...

In ROS, the transformations are broadcasted on the `/tf` topic.

The *morse* simulator publishes the transformation from `husky0/base_link` to the sensor frames. It also publishes a transformation from `husky0/base_footprint` to `odom`. However, by default no transformation between `husky0/base_link` and `husky0/base_footprint` is published.

The TF tree can be shown using:

[1] `http://wiki.ros.org/tf`

```
rosrun rqt_tf_tree rqt_tf_tree
```

It can also be shown as a plugin in `rqt`.

We need to publish the missing transformation ourself, we can use ROS's `static_transform_publisher` to continuously publish a frame:

```
  rosrun tf static_transform_publisher  x y z qx qy qz qw
                                    source_frame target_frame hz
```

Where `x`, `y` and `z` correspond to the translation (in this case the `base_link` is $0.09947m$ above the `base_footprint`). `qx`, `qy`, `qz` and `qw` correspond to the rotation (as a quaternion, in this case there is no rotation so $0, 0, 0, 1$). `source_frame` and `target_frame` are the name of the frames for which we want to publish a transformation. `hz` is the frequency of publication ($100Hz$ is a good choice)

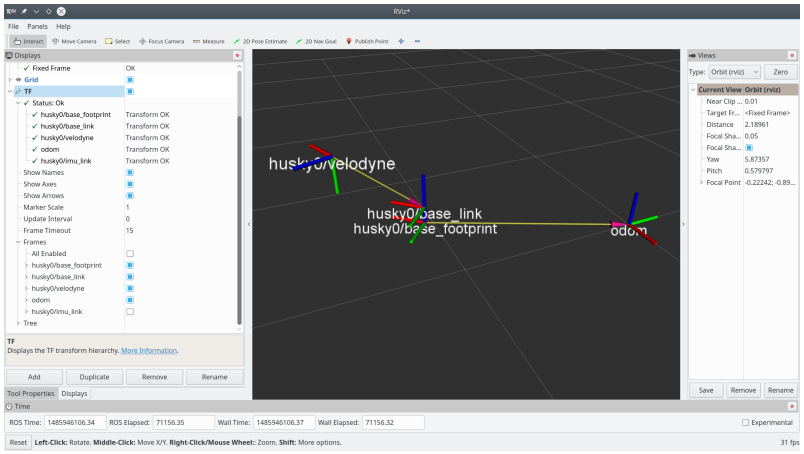

Figure 1: TF shown in RViz

You can also display TF in `rviz` (see figure 1). You simply need to add a TF display and in the global options, you need to set the *Fixed frame* to `odom`.

If you want to access TF information from your code, in C++, you can use a `tf::TransformListener`. This class will listen on `/tf` topic and allows you to lookup for transformation between two frames:

```
// Declare this as a class member! It needs to stay alive while
// your program is running
tf::TransformListener m_tfListener;

// In one of your callback or elsewhere:
// We want to know the transformation from source_frame_id to
// destination_frame_id at time time_stamp

tf::StampedTransform transform;
try
{
  // Define a 1s timeout
```

```
  ros::Duration timeout(1.0);
  // First, lets wait a bit to make synchronize our TF tree and
  // make sure we have received the transform
  m_tfListener.waitForTransform(destination_frame_id,
                                source_frame_id, time_stamp,
                                timeout);
  // Then lets get the transformation
  m_tfListener.lookupTransform(destination_frame_id,
                               source_frame_id, time_stamp,
                               transform);
} catch(tf::TransformException& ex)
{
  ROS_ERROR_STREAM( "Failed to get the transformation: "
                    << ex.what() << ", quitting callback");
  return;
}
```

You can transform the coordinate of a point using:

```
  tf::Vector3 v = transform * tf::Vector3(x, y, z);
```

For python usage, you can see an example in `air_lab2/src/move_to_point.py`.

3 Point cloud message

The robot is equipped with a simulated Velodyne sensor, which generates point clouds outputted on the `/husky0/velodyne` topic using the `sensor_msgs/PointCloud2`[2] message. That message is essentially a binary blob which can contains a list of objects with attributes. The attributes are described in the `fields` field of type `sensor_msgs/PointField`. During this lab, the simulator generates a point cloud with three fields `x`, `y` and `z` of type `float32` corresponding to the Cartesian coordinates of the points.

To access the data in the binary blob, ROS provides a set of convenient classes, in the `sensor_msgs/point_cloud2_iterator.h` header. For this lab, we are mostly interested in the `sensor_msgs::PointCloud2ConstIterator` class which allows to access the value of one field in the message:

```
sensor_msgs::PointCloud2ConstIterator<float> it(message, "fieldname");

while(it != it.end())
{
  ROS_INFO_STREAM("Value is " << *it);
  ++it;
}
```

[2] http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html

`sensor_msgs::PointCloud2ConstIterator` gives access to a single field, you therefore need to create one of those iterator per field (`x`, `y` and `z`) you want to access and increment them synchronously.

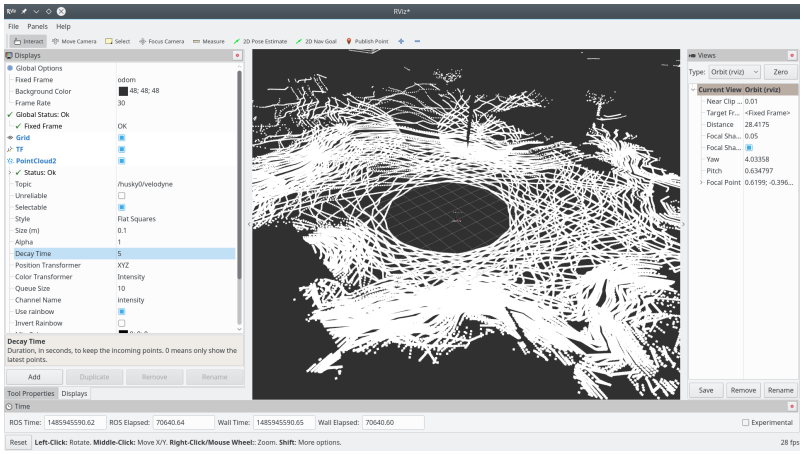

Figure 2: Points cloud shown in RViz

You can use RViz to visualize the point clouds like in figure 2. You can add a `PointCloud2` display and set the topic to `/husky0/velodyne`. You can increase the value for `Decay Time` to see multiple frames at the same time. And `size (m)` to view bigger points.

4 General architecture

The general architecture of the motion planning system that we will develop in this lab is presented in figure 3.

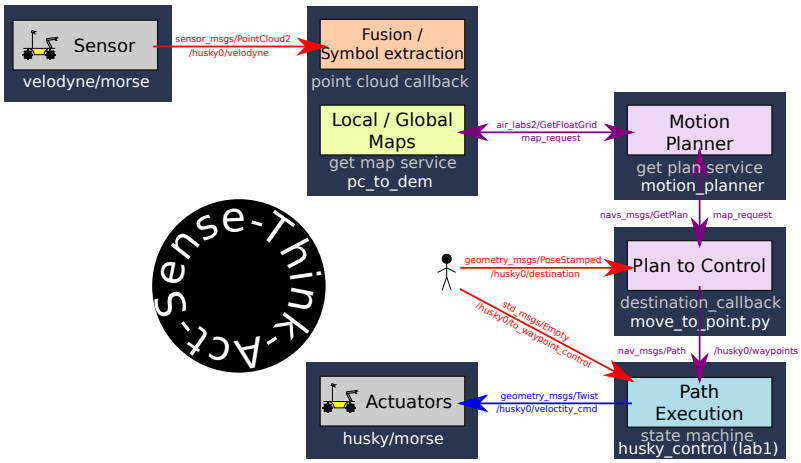

Figure 3: Architecture of the motion planning system, from sensing, to building a map, to planning, to execution

In this lab you will develop two new ROS nodes and use the state machine node you have developed during lab1:

- `pc_to_dem` a node that takes a point cloud from a sensor and build a digital elevation map (DEM). This node will also provide a service that return the map.

- `motion_planner` a node that provide as a service a motion plan from a point of origin to a specified destination. This node will use the DEM to compute the motion plan.

We provide you with three programs that will help you developing the functionalities for this lab, to get it into your project, run:

- `dem_to_display` a program that will transform the DEM that you build into a message of type `nav_msgs/OccupancyGrid` which can be display in Rviz.
- `traversability_to_display` a programm that will transfor the DEM that you build into a traversability map that will be published as message of type `nav_msgs/OccupancyGrid` which can be display in Rviz.
- `move_to_point.py` a program that will conveniently trigger the computation of a motion plan every time you select a destination in Rviz. It takes as input topic a `destination` of type `geometry_msgs.msg.PoseStamped` and it will call the planner to get a path that is then output on the `planned_path` topic. It uses a parameter called `robot_frame` which correspond to the robot frame in the TF tree

5 Messages and services

You will need to create a new message, `FloatGrid`:

```
std_msgs/Header header
nav_msgs/MapMetaData info
float32[] data
```

And a new service `GetFloatGrid`:

```
---
air_lab2/FloatGrid grid
```

The message allows to represent a DEM and the service define an interface to get the DEM and from the service.

6 Generate DEM

A `DEM` class is provided in `air_lab2/dem.h`, it contains functions for updating the DEM and for filling the `GetFloatGrid` service answer. Some documentation is provided in the header.

You should create a new ROS node called `pc_to_dem` (you can use `pc_to_dem.cpp` file and don't forget to update `CMakeLists.txt`), you can follow the same structure as the `simple_controller` node from the first lab.

6.1 Params

The DEM class requires three arguments, the best is to make them configurable on the command line, you can do so using the parameters system of ROS, you can access parameters in C++ this way:

```
1   double grid_cell_size    = 0.1;
2   ros::NodeHandle private_nodehandle("~");
3   private_nodehandle.getParam("grid_cell_size", grid_cell_size);
```

Do the same for the other two parameters of DEM (you can reuse `private_nodehandle`), then you can change the parameters from the command line the same way you did in lab1 to set the parameters of the PID controller.

6.2 DEM generation algorithm

In a callback triggered everytime a new point cloud $\mathscr{P}$ is received, follow those steps to generate a DEM:

1. Get the transformation $\mathscr{T}$ from the sensor frame (given by `msg.header.frame_id`) to `odom` using the TF listener (see section 2)
2. Call the `ensureInitialise` function of the DEM class with $\mathscr{T}$ as arguement.
3. For all points P_s in the point cloud $\mathscr{P}$, compute its coordinates in the `odom` frame as $P_g = \mathscr{T} * P_s = (x_g, y_g, z_g)$ and call the function `updateElevationAt` of DEM with P_g as argument.

6.3 DEM request service

Now you need to give access the DEM through service calls. We will use a ROS service call (named `map_request`) for that purpose using the `air_lab2/GetFloatGrid` service definition.

You will create a callback for the service function that looks like:

```
1  bool mapService(air_lab2::GetFloatGridRequest& _req,
2                  air_lab2::GetFloatGridResponse& _resp)
3  {
4  ...
5  }
```

You can use the `requestMap` function of the DEM class to fill the response.

6.4 Run the DEM generation and display

You can run the DEM:

```
1  rosrun air_lab2 pc_to_dem __ns:=/husky0 point_cloud:=velodyne
```

To display it:

```
rosrun air_lab2 dem_to_display __ns:=/husky0
```

Then in Rviz add a display for `map` and set it to the topic `/husky0/dem_display`. You should see something like on figure 4.

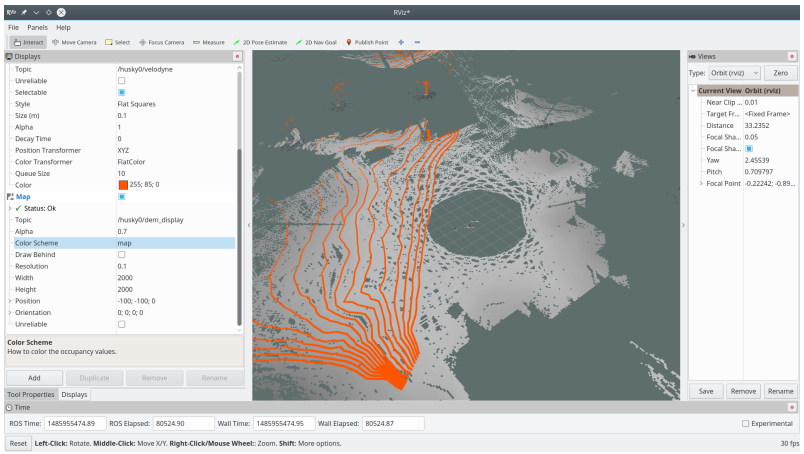

Figure 4: Rviz with a map. The grey part show the altitude, the greenbluish color represent *unknown* area.

In Rviz you can also add a display for `MarkerArray` and use the topic `/husky0/dem_markers_display` to display the map in 3D. However this view might be too computationally intensive for the labs computer and we recommend that you do not run it continuously.

6.5 Initial altitude

As you can see on figure 4, there is an unknown area around the robot because the laser cannot see close. This will cause problem for the motion planning, to work around the problem, you can set the altitude around the robot to the current altitude of the robot. Using the `initial_radius` and `initial_relative_elevation` (to guide you: the Lidar is 43cm above the ground).

7 Path planning

In the second part of the lab you should create a program that will plan motion path using the OMPL[3] library. To do this you should create a new ROS node called `motion_planner` (you can use `motion_planner.cpp` file and don't forget to update `CMakeLists.txt`).

A `MotionPlannerInterface` class is provided for you in `air_lab2/motion_planner_interface.h`, it contains a function that generate a path based on a start and end position.

In your `motion_planner` node, you should create a service call `plan_path` of type `nav_msgs/GetPlan` and then call the function `planPath` of `motion_planner`. You then need to convert the list of points returned to that function into a list of `geometry_msgs/PoseStamped`.

You will need to compute the orientation of the pose around each point of the path, to do so follow figure 5 (for the first and last point, follow the direction of the path).

[3] `http://ompl.kavrakilab.org/`

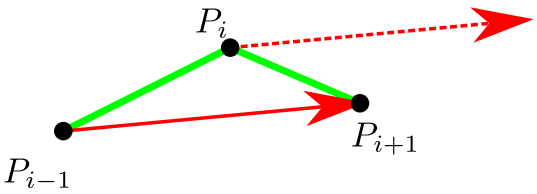

Figure 5: Compute the orientation around P_i to be parallel to the vector P_{i-1}, P_{i+1}

8 Running the motion planner

To run the motion planner you need to run two ROS nodes:

```
rosrun air_lab2 motion_planner __ns:=/husky0
rosrun air_lab2 move_to_point.py __ns:=/husky0 \
          _robot_frame:=husky0/base_footprint planned_path:=waypoints
```

You can then use Rviz to select the point (on topic `/husky0/destination`), display the path (on topic `waypoints`). Do not forget to put your Husky controller in `waypoints` control to see it move.

To help you debug, you can use the `traversability_to_display` programm to display a map of which cells are considered traversable (that program is really slow).

9 Launch file

Update your launch file to include `pc_to_dem`, `dem_to_display`, `static_transform_publisher`, `motion_planner` and `move_to_point.py`!