TDDE05: Lab 2: Navigation

Cyrille Berger

February 22, 2021

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.

On a side note, you should not expect your robot to perfectly avoid the obstacles and perfectly follow the motion plan, but the plan you generate should be avoiding the obstacles.

1 Get the code for lab2

Get the updated 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 Simulator with obstacles

You now should run the simulator with obstacles:

```
rosrun air_simple_sim simple_sim.py _world:=world_1 __ns:=/husky0
```

You can update your screen file, if needed. On a side note, the simulator does not simulate colision, so your robot will go through the obstacles.

3 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

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

- 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, and you should see something similar to figure **??**:

```
rosrun rqt_tf_tree rqt_tf_tree
```

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

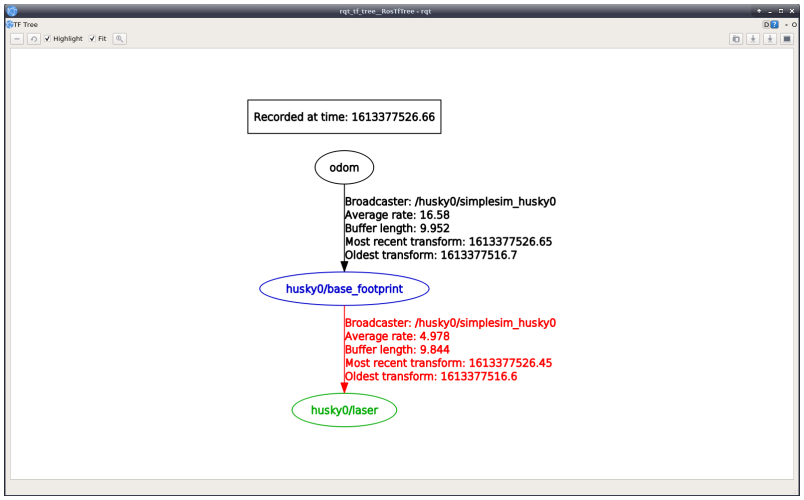

Figure 1: TF shown in rqt

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`.

4 Lidar scan message

The robot is equipped with a simulated 2D lidar sensor, which generates a message on the `/husky0/lidar` topic using the `sensor_msgs/LaserScan`[2] message. That message contains some meta information about the scan, which are traced from the sensor from `angle_min` to `angle_max` in increment of `angle_increment`. The `ranges` field contains the distance between the sensor and an obstacle. A `NAN` value indicates that the lidar didn't detect an obstacle. `range_max` indicates the maximum distance at which the lidar can detect obstacles. Figure 2 shows the most important value.

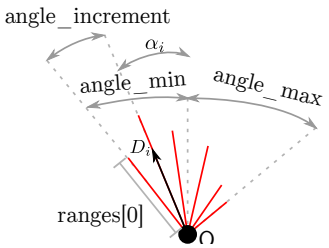

Figure 2: Laser scan, red lines indicate laser ray.

You can use RViz to visualize the lidar scan like in figure 3. You can add a `LaserScan` display and set the topic to `/husky0/lidar`. You can increase the value for `Decay Time` to see multiple frames at the same time. And `size (m)` to view bigger points.

[2] `http://docs.ros.org/en/api/sensor_msgs/html/msg/LaserScan.html`

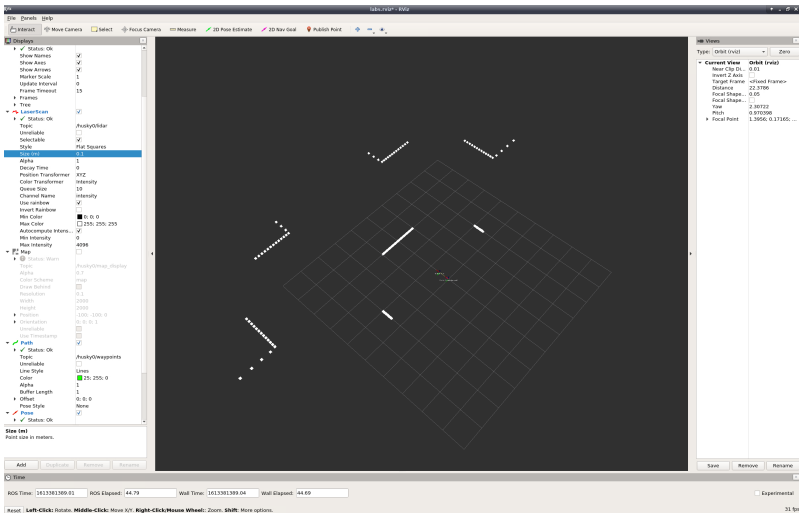

Figure 3: Lidar scan shown in RViz

5 General architecture

The general architecture of the motion planning system that we will develop in this lab is presented in figure 4.

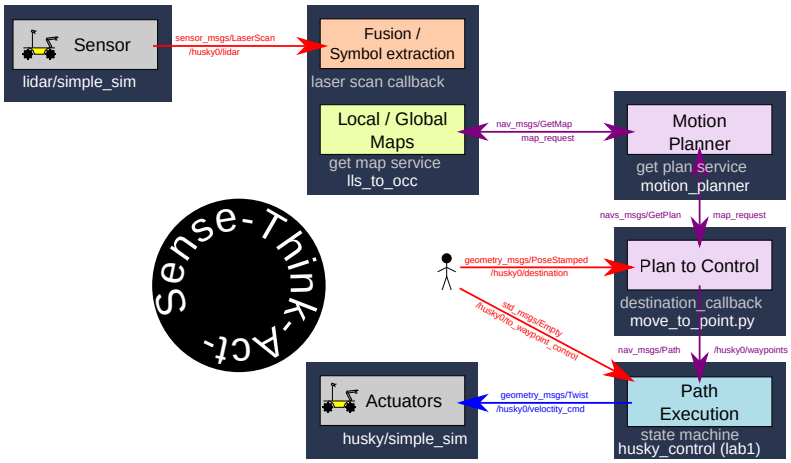

Figure 4: Architecture of the motion planning system, from sensing, to building a map, to planning, to execution

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

- `ls_to_occ` a node that takes a lidar scan from a sensor and build an occupancy mapo. 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 occupancy map 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:

- `occ_to_display` a program that allows to display the occupancy map in Rviz.
- `traversability_to_display` a programm that will transform the occupancy grid 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.

6 Generate Occupancy Grid

A OCC class is provided in `air_lab2/occ.h`, it contains functions for updating the OCC and for filling the `nav_msgs/GetMap` service answer. Some documentation is provided in the header.

You should create a new ROS node called `ls_to_occ` (called `ls_to_occ.cpp` file. You cann use the following base structure for your node:

```
#include "air_lab2/occ.h"

#include <ros/node_handle.h>
#include <ros/service.h>
#include <ros/subscriber.h>

#include <atomic>

#include <sensor_msgs/LaserScan.h>

class LStoOCC {
public:
  LStoOCC(const ros::NodeHandle& _nodeHandle)
    : m_nodeHandle(_nodeHandle), m_occ(nullptr),
      m_cell_size(0.1), m_robot_size(0.0)
  {
    // Fill in
  }


  void laserScanCallback(const sensor_msgs::LaserScanPtr& _message)
  {
    // Fill in
  }
private:
```

```
  ros::NodeHandle m_nodeHandle;
  OCC* m_occ;
  double m_cell_size, m_robot_size;
  // Fill in
};

int main(int argc, char** argv)
{
  ros::init(argc, argv, "ls_to_occ");
  ros::NodeHandle n;

  LStoOCC ptd(n);

  ros::spin();
  return 0;

}
```

Add the node in `CMakeLists.txt`, where there are other `add_executable`, near the bottom of the file:

```
add_executable(ls_to_occ src/ls_to_occ.cpp)
target_link_libraries(ls_to_occ
${catkin_LIBRARIES}
)
```

Do not forget to run `catkin build` every time you make a change to your C++!

6.1 Params

The `OCC` class requires two 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:

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

Do the same for the robot size (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 OCC generation algorithm

In the constructor of `LStoOCC` you need to subscribe to the topic in the constructor (check lecture 02 fore more details):

```
m_lsSub = m_nodeHandle.subscribe("scan", 1,
          &LStoOCC::laserScanCallback, this);
```

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

1. Get the transformation $\mathscr{T}$ from the sensor frame (given by `msg.header.frame_id`) to `odom` using the TF listener (see section 3)
2. Call the `ensureInitialise` function of the `OCC` class with $\mathscr{T}$ as arguement.
3. For all direction of the laser, compute the origin of the ray, as $O = \mathscr{T} * \{0,0,0\}$, the direction of the ray as $D_i = (\mathscr{T} * \{cos(\alpha_i), sin(\alpha_i), 0\} - O)$ (make sure D_i is normalised, you can use the `normalize` function from a `tf::Vector3`). Look at figure 2.

   Then call the function `rayTrace` of `OCC` where `_skip` is the size of the robot, `_length` is the range and `_obstacle` indicate if the ray has hit an obstacle. If the range of the current ray is `NAN`[3] give `range_max` as argument to `ray\_trace` and `\_obstacle` should be `false`.

6.3 OCC request service

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

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

```
bool mapService(nav_msgs::GetMapRequest& _req,
                nav_msgs::GetMapResponse& _resp)
{
  // Fill in
}
```

You can use the `requestMap` function of the `OCC` class to fill the response. You can create a service the following way (check lecture 02 fore more details):

```
m_mapRequest = m_nodeHandle.advertiseService("map_request",
               &LStoOCC::mapService, this);
```

6.4 Run the OCC generation and display

You can run the OCC:

```
rosrun air_lab2 ls_to_occ __ns:=/husky0 scan:=lidar
```

To display it:

[3] You should use the `std::isnan(x)` function and not comparison to `NAN`

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

Then in Rviz add a display for `map` and set it to the topic `/husky0/map_display`. You should see something like on figure 5.

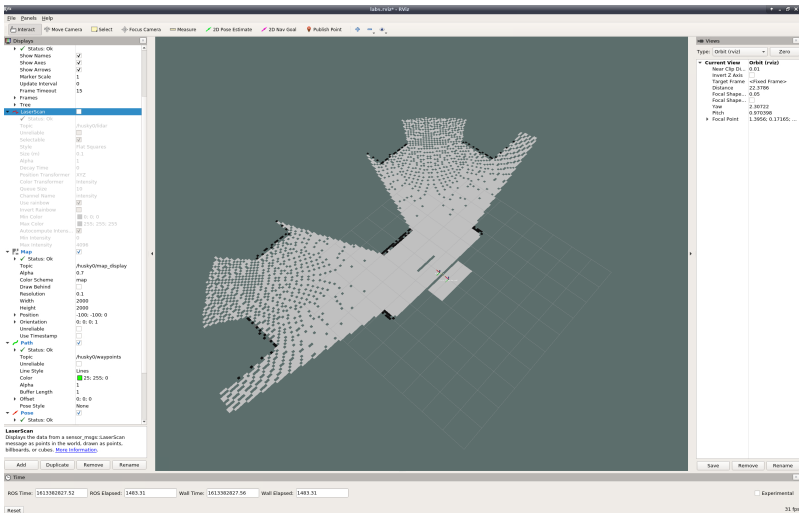

Figure 5: Rviz with a map. The grey part shows no obstacle, the greenbluish color represent *unknown* area, and black obstacles.

6.5 Initialisation

If on figure 5, there is an unknown area around the robot, this is because the laser cannot see close. This will cause problem for the motion planning, to work around the problem, make sure that the `initial_size` of your robot is correctly set (you can use a value of 1.0).

7 Path planning

In the second part of the lab you should create a program that will plan motion path using the OMPL[4] 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`).c Follow a similar model as you did for `ls_to_occ.cpp`.

In `CMakeLists.txt`, you should link with the `OMPL` library, using the following line:

```
target_link_libraries(motion_planner
${catkin_LIBRARIES} ${OMPL_LIBRARIES}
)
```

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. You should call it like this:

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

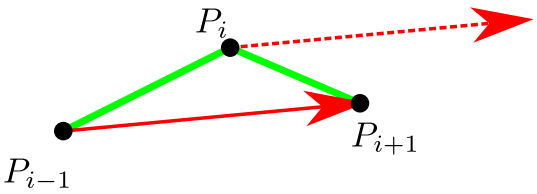

Figure 6: Compute the orientation around P_i to be parallel to the vector P_{i-1}, P_{i+1}

```
... = motion_planner_interface->planPath<nav_msgs::GetMap>(...);
```

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 6 (for the first and last point, follow the direction of the path).

A few C++ tips:

- To access elements of a `std::pair`, you can use `first` and `second`:

```
1       std::pair<double, double> coord(4,3);
2       std::cout << "x= " << coord.first
3                 << " y= " << coord.second << std::endl;
```

- To add elements to a vector, the best is to use `push_back`:

```
1       geometry_msgs::PoseStamped p;
2       ...
3       poses.push_back(p);
```

- The `geometry_msgs/PoseStamped` message needs quaternion, you are going to compute the angle as euler, you can use the `tf::Quaternion` class to compute the quaternion value:

```
1       tf::Quaternion orientation;
2       orientation.setRPY( 0, 0, yaw_angle);
3
4       geometry_msgs::PoseStamped p;
5       p.pose.orientation.x = orientation.getX();
6       ...
```

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 `ls_to_occ`, `occ_to_display`, `motion_planner` and `move_to_point.py`!