TDDE05: Lab 2: Navigation

Cyrille Berger

February 6, 2017

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

1 Get the code for lab2

Get the skeleton code for lab1 from gitlab:

```
1 cd ~/TDDE05/catkin_ws/src/air_labs/
```

```
2 git pull git@gitlab.ida.liu.se:tdde05/air_labs.git master
```

WARNING: if you try to build at this point, you are very likely to get an error, you need to define some messages and services first.

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.

¹http://wiki.ros.org/tf

The TF tree can be shown using:

1 rosrun rqt_tf_tree rqt_tf_tree

1 2 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 0001). 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
 1
 2
  // your program is running
   tf::TransformListener m_tfListener;
3
 4
 5
   // In one of your callback or elsewhere:
   // We want to know the transformation from source_frame_id to
 6
 7
   // destination_frame_id at time time_stamp
 8
9
   tf::StampedTransform transform;
10
   trv
11
    {
12
     // Define a 1s timeout
13
     ros::Duration timeout(1.0);
     // First, lets wait a bit to make synchronize our TF tree and
14
     // make sure we have received the transform
15
     m_tfListener.waitForTransform(destination_frame_id,
16
17
                                     source_frame_id, time_stamp,
18
                                     timeout);
19
      // Then lets get the transformation
20
      m_tfListener.lookupTransform(destination_frame_id,
21
                                    source_frame_id, time_stamp,
22
                                    transform);
    } catch(tf::TransformException& ex)
23
24
    {
25
      ROS_ERROR_STREAM( "Failed to get the transformation: "
                        << ex.what() << ", quitting callback");
26
27
      return;
28
   }
      You can transform the coordinate of a point using:
      tf::Vector3 v = transform * tf::Vector3(x, y, z);
 1
```

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/PointCloud2² 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:

 $^{^{2} \}tt{http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html}$

```
1 sensor_msgs::PointCloud2ConstIterator<float> it(message, "fieldname");
2
3 while(it != it.end())
4 {
5 ROS_INFO_STREAM("Value is " << *it);
6 ++it;
7 }</pre>
```

 $sensor_msgs::PointCloud2ConstIterator$ 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.

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



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

- 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.
- 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:

- 1 std_msgs/Header header
- 2 nav_msgs/MapMetaData info
- 3 float32[] data

```
And a new service GetFloatGrid:
```

1 -

2

```
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

To help with the DEM, we provide you with a template class (in the air_lab2/extensible_grid.h header):

6.1 extensible_grid

```
1 template<typename _T_, int _SubSize_ = 1000>
2 class extensible_grid;
```

This class provides an implementation of a grid that can be expanded to the infinite. It works by storing the cells in subgrids whose size is given by _SubSize_. _T_ is the type of the grid cell. It is a sparse structure, and subgrids are only instantiated if needed, see figure 4.

You can create an extensible_grid with:

```
1 struct cell_type { /* fields */ };
```

```
2 extensible_grid<cell_type> grid(resolution);
```

resolution is the size of a cell, I sugget to use 0.1m, you should have it set using a ROS parameter.

You can access a value in the grid with (*x* and *y* are expressed in meters):

```
cell_type& cell = grid.get_value_ref(x, y);
 1
      You can iterate over all the different subgrids:
    for(auto cit = grid.cbegin(); cit != grid.cend(); ++cit)
 1
 2
    {
 3
      // Not all the subgrids are instantiated, check if this one is:
 4
      if(cit.is_valid())
 5
      {
 6
        for(int y = 0; y < cit.get_size(); ++y)</pre>
 7
        {
8
           for(int x = 0; x < cit.get_size(); ++x)</pre>
9
           {
10
             const cell_type& c = cit(x,y);
11
          }
12
        }
13
      }
14
15
   }
```



Figure 4: extensible_grid and its associated functions.

6.2 DEM generation algorithm

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

- Get the transformation *T* from the sensor frame (given by msg.header.frame_id) to odom using the TF listener (see section 2)
- 2. For all points P_s in the point cloud \mathscr{P} , compute its coordinates in the odom frame as $P_g = \mathscr{T} * P_s = (x_g, y_g, z_g)$. z_g correspond to the altitude of the point, x_g and y_g to its coordinate in the DEM map. This allow you to update the altitude of the point, using a simple averaging technique:

$$elevation_at_cell(x_g, y_g) = \frac{elevation_at_cell(x_g, y_g) + z_g}{samples_at_cell(x_g, y_g) + 1}$$
(1)

$$samples_at_cell(x_g, y_g) = samples_at_cell(x_g, y_g) + 1$$
(2)

See section 3 to access the points in the point cloud.

6.3 DEM request service

Now you need to give access the DEM through service calls. We will use a ROS service call 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 }
```

In _resp you will need to fill:

- the header field:
 - frame_id with the frame of the map, in this case odom
 - stamp the current time (you can use ros::Time::now())
- the info field:
 - resolution in m, the size of the cell
 - width, height, origin with width, height and origin of the map (you can get them from the extensible_grid)

Then the tricky part is to fill the data field. First you will need to resize the grid with:

```
1 __resp.grid.data.resize(_resp.grid.info.width*_resp.grid.info.height, NAN);
You can access pixel (x,y) with:
```

1 __resp.grid.data[x + y * _resp.grid.info.width]

Then you need to fill the data. We suggest that you use the subgrid_iterator_impl from the extensible_grid class to iterate over the subgrids and fill the resulting data structure. In figure 5 you can see a mapping between the coordinate (xg,yg) in the FloatGrid and the coordinate in a subgrid.



Figure 5: Coordinates for copying a subgrid to a FloatGrid.

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:
```

1 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 6.



Figure 6: Rviz with a map. The grey part show the altitude, the greenbluish color represent *unknown* area.

6.5 Initial altitude

As you can see on figure 6, 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.

7 Path planning

In the second part of the lab you should create a program that will plan motion path using the OMPL³ library. To do this you should create a new ROS node called motion_planner.

You will need the following headers from OMPL:

```
1 #include <ompl/base/Goal.h>
```

```
2 #include <ompl/base/spaces/SE2StateSpace.h>
```

```
3 #include <ompl/geometric/SimpleSetup.h>
```

```
4 #include <ompl/geometric/planners/rrt/RRTstar.h>
```

```
5 #include <ompl/geometric/planners/rrt/RRTConnect.h>
```

We will assume some namespace aliases to make our life easier:

```
1 namespace ob = ompl::base;
```

```
2 namespace og = ompl::geometric;
```

The main challenge for using OMPL is to create a ob::StateValidityChecker which is a class that allow to check if a state is a valid position in the environment. We will consider that the Husky has four points of contact, as shown on figure7, you can assume that width is 60*cm* and length is 80*cm*.



Figure 7: Husky's footprint.

We will consider that a position is valid according to the following algorithm:

- Fit a plane *P* through the four contact points A, B, C, D. You can use gte::ApprOrthogonalPlane3 in the air_lab2/Mathematics/GteApprOrthogonalPlane3.
 h.
- Check that the distance between the points *A*, *B*, *C*, *D* and plane \mathcal{P} is not bigger than 10cm.
- Check that the normal of the plane \mathscr{P} does not have a angle with the vertical bigger than 30°.
- Check that there are no point whithin the robot footprint that goes higher than the ground clearance of the robot which is 15*cm*. See figure 8 for a few example of good and bad configurations.

³http://ompl.kavrakilab.org/



Figure 8: Some collision configuration.

This node should provide a service plan_path of type nav_msgs/GetPlan, with a callback function that looks like: 1 bool planPath(nav_msgs::GetPlanRequest& _req, 2 nav_msgs::GetPlanResponse& _resp) 3 {

4 5 }

. . .

The first step is to get the map from pc_to_dem ROS node. You can look at dem_to_display to see how to make a service call to pc_to_dem to access the map.

```
1
   class StateValidityChecker : public ob::StateValidityChecker
2
   {
 3
   public:
 4
      StateValidityChecker(const air_lab2::FloatGrid& _grid,
 5
                       ob::SpaceInformation *si)
        : ob::StateValidityChecker(si), m_grid(_grid)
 6
 7
      {}
 8
      virtual bool isValid(const ob::State* state) const
 9
      {
10
        const ob::SE2StateSpace::StateType* ss = state->
                                      as<ob::SE2StateSpace::StateType>();
11
        // Coordinate of the state are given by ss->getX(), ss->getY()
12
        // and ss->getYaw()
13
        // Check the footprint of the robot for collision
14
15
        return ???;
      }
16
      double elevation_at(double _x, double _y) const
17
18
      {
19
        int x = (_x - m_grid.info.origin.position.x) / m_grid.info.resolution;
        int y = (_y - m_grid.info.origin.position.y) / m_grid.info.resolution;
20
21
        // Modify this function to look in neighbourgh cell for an
22
        // altitude if std::is_nan(elevation_safe_at(x, y))
23
24
        return elevation_safe_at(x, y);
25
      }
26
      double elevation_safe_at(int _i, int _j) const
27
      Ł
28
        if(_i \ge 0 and
           j \ge 0 and
29
30
           _i < m_grid.info.width and
31
           _j < m_grid.info.height)
        {
32
          return m_grid.data[ _i + _j * m_grid.info.width ];
33
34
        } else {
35
          return NAN;
36
        }
37
      }
38
   private:
39
      const air_lab2::FloatGrid m_grid;
40 };
      Then in your planPath, after getting the map you can do the path planning.
      Create an instance of the state space, we will use a 2D space (x,y,yaw):
```

1 ob::StateSpacePtr space(new ob::SE2StateSpace());

We need to set the bounds according to the map bounds:

```
ob::RealVectorBounds bounds(2);
1
2 bounds.setLow(0, left coordinate of the map);
3 bounds.setHigh(0, right coordinate of the map);
4 bounds.setLow(1, top coordinate of the map);
  bounds.setHigh(1, bottom coordinate of the map);
5
  space->as<ob::SE2StateSpace>()->setBounds(bounds);
6
     Setup the valididy checker:
   ob::SpaceInformationPtr si(new ob::SpaceInformation(space));
1
2
  ob::StateValidityCheckerPtr svc(
3
       new StateValidityChecker(req.response.grid, si.get()));
4
  si->setStateValidityChecker(svc);
5
     Set the start and goal position according to _req:
   ob::ScopedState<> start(space);
1
2 start[0] = start x;
3 start[1] = start y;
4 start[2] = start yaw;
5
6 ob::ScopedState<> goal(space);
  goal[0] = start x;
7
8 goal[1] = start y;
  goal[2] = start yaw;
9
     Create an instance of ompl::base::ProblemDefinition and set the start and goal states
  for the problem definition.
  ob::ProblemDefinitionPtr pdef(new ob::ProblemDefinition(si));
1
2
3 pdef->setStartAndGoalStates(start, goal);
```

We will use the RRT* planner:

```
ob::PlannerPtr planner(new og::RRTstar(si));
 1
 2
   planner->setProblemDefinition(pdef);
3 planner->setup();
4
5
   ob::PlannerStatus solved = planner->solve(1.0);
6
7
   if (solved)
8
   {
9
      og::PathSimplifierPtr psp(new og::PathSimplifier(si));
10
      og::PathGeometric pg = *static_cast<og::PathGeometric*>(
11
12
                                pdef->getSolutionPath().get());
     psp->simplify(pg, 1.0);
13
14
15
      _resp.plan.header.frame_id = req.response.grid.header.frame_id;
16
17
     // Convert the plan
18
     for(ob::State* s : pg.getStates())
19
      {
20
        const ob::SE2StateSpace::StateType* ss =
21
                          s->as<ob::SE2StateSpace::StateType>();
22
        // You can use ss->getX(), ss->getY() and ss->getYaw()
        // to get the coordinate
23
24
      }
25
   }
```

8 Running the motion planner

You can run the motion planner this way:

3

```
1 rosrun air_lab2 motion_planner __ns:=/husky0
```

```
2 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.