# Laboratory class #2: Low level control of a mobile robot in ROS

Let us install the following package,

```
$ sudo apt-get install ros-noetic-tf2-tools
$ sudo apt-get install ros-noetic-rqt
$ sudo apt-get install ros-noetic-rqt-graph
```

## 1. Transformations in ROS (tf)

When working with a robotic environment there exist different local coordinate frames, for example one for each robot and it is very common the need of performing transformation between different frames. ROS has a tool which permits transformation between frames, called **tf**.

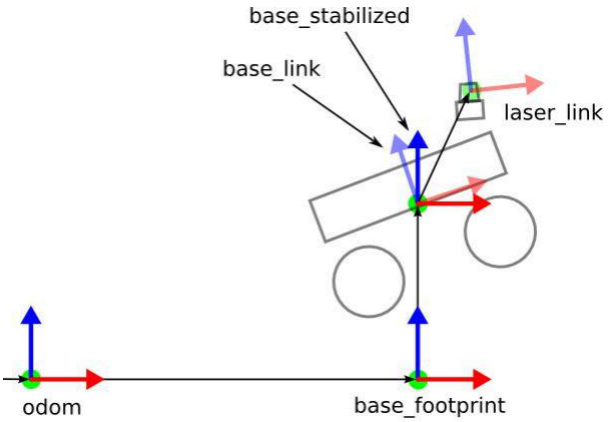


Figure 1. Different 3D coordinate frames of stage of a robot

The tf package is a tool to track multiple 3D coordinate frames over time. It maintains the relationship between coordinate frames in a tree structure buffered in time. Additionally, it allows the user to transform points and vectors between the coordinate frames at desired time. It is implemented as a publisher/subscriber model on the topics /tf and /tf_static.

Figure 1 shows the different frames generated by Stage when our example world is launched. The main frame is odom with the (0,0,0) coordinate in the center of the map. Frames base_footprint and base_link are identical for us (there is an identity transformation between them) since in our 2D environment the robots always are on ground. Finally, base_laser is the frame corresponding to the laser sensor. We may also have a camera frame if such a sensor exists on the robot.

TF builds a tree of transforms between frames. This tree can be obtained by using the following commands:

```
$ rosrun tf2_tools view_frames.py
$ evince frames.pdf
```

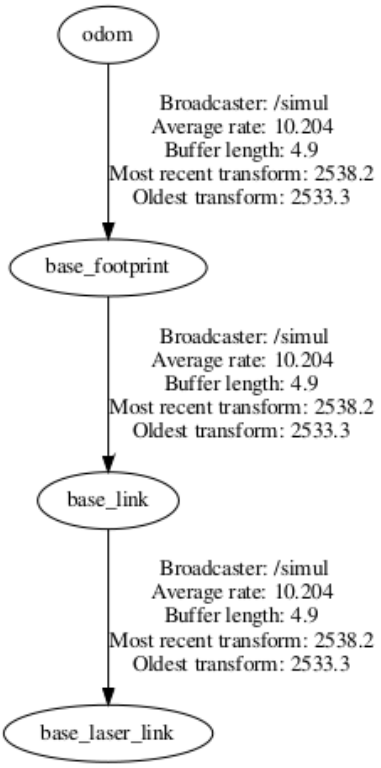


Figure 2. An example of a tf tree

Figure 2 shows an example of a tf tree in the case of one robot obtained by running the previous command once the launch file that starts ROS is executed.

## *Exercise 1.*

For this exercise it requires the same teleop_twist_keyboard_cpp node used in laboratory 1 (to move the robot with the keyboards). The source code of the node is included in the package that we prepared for you for this lab and can clone in your ~/catkin_ws/src folder directly from Visual Studio Code (by pressing Ctrl + Shift + P -> Git: Clone) from the following URL:

https://github.com/luisriazuelo/pXX_arob_lab2.git

After cloning the repository, you have to modify the name of the following folder and the content of the following files according to the pair number:

Folders:

- pXX_arob_lab2

Files:

- CMakeList.txt
- package.xml
- launch/start.launch

Change pXX prefix, where pXX corresponds to the assigned pair number (e.g. p00, p01 .....) for this course. The assigned pair number can be found in the Moodle document "Laboratories Pairs".

After modifying the package, compile the package and execute the launch file (from the launch folder and(do not forget to execute `source devel/setup.bash`).

```
$ roslaunch pXX_arob_lab2 start.launch
```

The information about the current transform tree can be visualized also in the console by using the following command:

```
$ rosrun tf tf_monitor
```

You can try the commands presented before the exercise and see the tf tree as a pdf file. In order to print information about the transformation between two frames, the following command can be used,

```
$ rosrun tf tf_echo source_frame target_frame
```

If we consider the initial state of the robot, the following command

```
$ rosrun tf tf_echo odom base_link
```

will return the following,

```
- Translation: [-6.000, -6.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
            in RPY (radian) [0.000, -0.000, 0.000]
            in RPY (degree) [0.000, -0.000, 0.000]
```

Use the cpp code below and add a node called robot_location to package pXX_arob_lab2. Compile and execute the node to see the actual position of the robot (by using transformation between base_footprint and odom frames). Remember from lab1 that you need to update the CMakeList.txt file in the package folder (add_executable and target_link_libraries). The cpp code is the following:

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
using namespace std;
int main(int argc, char** argv){
        ros::init(argc, argv, "robot_location");
        ros::NodeHandle node;
        tf::TransformListener listener;
        ros::Rate rate(2.0);
        listener.waitForTransform("/base_footprint", "/odom", ros::Time(0),ros::Duration(10.0));
        while (ros::ok()){
                tf::StampedTransform transform;
                try {
                listener.lookupTransform("/base_footprint", "/odom", ros::Time(0),transform);
                double x = transform.getOrigin().x();
                double y = transform.getOrigin().y();
                cout << "Current position: (" << x << "," << y << ")" << endl;
         } catch (tf::TransformException &ex) {
         ROS_ERROR("%s",ex.what());
         }
         rate.sleep();
         }
 return 0;
}
```

### 2. rqt User Interface

rqt is a software of ROS that implements the various GUI tools in the form of plugins. One can run all the existing GUI tools as dockable windows within rqt but the tools can still run in a traditional standalone method. In order to run the rqt tools execute the following command

```
$ rqt
```

that shows a GUI where you can choose any available plugins on your system. If you want to run only a graph of nodes and topics, you can execute,

```
$ rosrun rqt_graph rqt_graph
```

## 3.- Defining the world and the objects in stage

To run simulations on stage, the environment/world should be given as an input parameter. The world is described in a text file called worldfile, usually with `.world` extension and in the world folder inside the package. Check the `simple.world` file from the world folder in the package for the second laboratory.

### 3.1. Defining and loading models

A worldfile contains a set of models that define all the elements needed for simulations. A standard definition of a model is done using the following format,

```
define model_name model
(
   # parameters
)
```

This is defining a model called “model_name” and between brackets there is a list of parameters. These parameters depend on the type of model that is defined. After the map model is defined it can be loaded and during this step, an image representation of the environment can be attached. This feature will be used in the following lab.

### 3.2. Stage simulation window and worldfile properties

In the worldfile it is also necessary to specify the window that will be used by Stage for the simulation. In our worldfile, the first lines are defining the stage window as

```
window
(
  size [ 635.000 666.000 ] # size of the window in pixels
  scale 36.995   # pixels per meter
  center [ -0.040  -0.274 ] # location of the center of the window in world coordinates
  rotate [ 0.000  0.000 ] # angle of rotation relative to straight up (degrees)
  show_data 1   # 1=on 0=off : show the laser view of the robot
)
```

The descriptions of all the properties of the considered worldfile are given in the text. It is important to notice that there are many other parameters and for a full list is possible to consult the following link:

https://rtv.github.io/Stage/group__worldgui.html

In the worldfile, different properties of the simulation may be specified. In our willow-erratic world, the following properties are set,

```
resolution 0.02  # set the resolution of the underlying raytrace model in meters

interval_sim 100   # simulation timestep in milliseconds
```

All properties can be found here: https://rtv.github.io/Stage/group__world.html

### 3.3. Objects: dynamic obstacles and target points

Following the definition of the window, in the worldfile, a model called "block" is defined that will be used to simulate a dynamic obstacle that can be moved with the mouse during the robot movement. If a model is defining an object not identifiable by the sensors the `ranger_return` property should be set to -1 (or any other negative number) . Furthermore, if this model is not an obstacle and the robots should pass through it without colliding, `obstacle_return` property should be set to 0. These two properties of the model can be used to define a target point to be visible on the map but that is not affecting the motion of the robots.

*Exercise 2.*

Update the world file simple.world by adding a new model of size 0,15 x 0,15 x 0,15 and yellow color defining a target point. Add such objects at the following poses: [2 -5 0 0], [4 -1 0 0], [-2 0 0 0] and [-1 5 0 0]. Move the robot with the keyboard and check that the robot is not colliding with this new model introduced but is colliding with the dynamic obstacle in the original worldfile.

## 4.- Typical messages in ROS

### 4.1. Standard messages (std_msgs)

During the first laboratory class we consider a typical type of message of type std_msgs::String. Many other types of std_msgs exist, and a list of all types can be found here: http://wiki.ros.org/std_msgs. Just as a reminder, let us perform the following commands:

```
$ roscore

$ rostopic list
```

In other terminal,

```
$ rostopic pub /hello std_msgs/Int8 10
```

Without pressing CTRL+C, split again one window and execute

```
$ rostopic list
```

You'll be able to see a new topic called `/hello`. On the same terminal window, execute

```
$ rostopic echo /hello
```

And you'll be able to see that on that topic there is the message "10".

### 4.2. Some ROS messages

Geometry_msgs (http://wiki.ros.org/geometry_msgs) provides messages for common geometric primitives such as points, vectors, and poses. They can be used to set the state of the robot, send velocities to the robot wheels or read laser scans.

- `geometry_msgs/PoseStamped.msg` can be used to define the state of the robot. A variable can be defined as this type using: `geometry_msgs::PoseStamped Goal` (having `#include <geometry_msgs/PoseStamped.h>`). In this case, the variable `Goal` has the following structure:
  - `Goal.pose.point.x; Goal.pose.point.y; Goal.pose.point.z;`
  - `Goal.pose.orientation.x; Goal.pose.orientation.y; Goal.pose.orientation.z; Goal.pose.orientation.w.`

  It is important to mention that this type of message also has a header in which it is possible to put time information and frame ids, useful for example if we have many nodes executing on different computers.

- `geometry_msgs/Twist.msg` can be used to send linear and angular velocities to the robot. A variable is defined as this type using: `geometry_msgs::Twist input` (having `#include <geometry_msgs/Twist.h>`). In this case, the variable input has the following structure:
  - `input.linear.x` - is the linear velocity;
  - `input.angular.z` - is the angular velocity.

  Run the launch file from arob_lab2 package and then execute the following command:

```
$ rostopic pub /cmd_vel geometry_msgs/Twist -r 100 '[10, 0, 0]' '[0, 0, 4]'
```

The robot should start moving with a linear velocity of 10 and an angular velocity of 4. The option "-r x" will publish the topic at a frequency of x Herts.

- `nav_msgs/Odometry.msg` can be used to get the actual position of the robot in the environment. A constant with the current odometry of the robot is defined as: `const nav_msgs::Odometry& msg`. The `msg` constant has the following structure:
  - `msg.pose.pose.position.x, msg.pose.pose.position.y` are the x and y position of the robot;

- `tf::getYaw(msg.pose.pose.orientation)` is the orientation (requiring to have `#include <tf/transform_broadcaster.h>`).

When a worldfile is loaded in stageros, the following **topics** published or subscribed by stageros can be used to read and control the robot:

- /odom (message type nav_msgs/Odometry) containing odometry data from the position model.
- /base_scan (message type sensor_msgs/LaserScan) scans from the laser model
- /base_pose_ground_truth (of type nav_msgs/Odometry)
- /ground truth pose image (of type sensor_msgs/Image) visual camera image
- /depth (of type sensor_msgs/Image) depth camera image
- /camera_info (of type sensor_msgs/CameraInfo) camera calibration info

The /odom topic gives simulated odometry, which is affected by settings in the .world file (noise can be considered) while the /base_pose_ground_truth topic always provides a perfect, globally referenced pose for the robot in the simulation, independent of .world file settings. In this laboratory class, for controlling the robot, the /odom topic is read.

### *Exercise 3.*

Complete the node lowcontrol that subscribes to the robot's /base_pose_ground_truth and a topic called /goal to receive the target point and publish the required velocities to control the robot to the goal. The program can be tested given sequentially the poses of the target objects defined in the worldfile in Exercise 1.

In order to publish manually a goal, you can use the following command,

```
$ rostopic pub /goal geometry_msgs/PoseStamped '{header: {stamp: now, frame_id: "odom"}, pose: {position: {x: 2, y: -5, z: 0.0}, orientation: {w: 0}}}'
```

---

### *Exercise 4.*

The following exercise consists of testing the code implemented in the previous exercise on a real platform. To do this, you should send the package "pXX_arob_lab2" to one of the robots available for real-world testing. The code will be loaded using the method indicated during the practical session. The execution of a command similar to the previous exercise will be tested, and as a result, the real platform will move to a target position.

To launch the code on the robot, you should use the launch file "real_robot.launch." To do this, you must understand the contents of the file and modify it according to your pair number.

---

### *Exercise 5.*

Complete the followTargets node such that it will read from the text file targets.txt the list of targets and publish them one by one to the topic /goal created in Exercise 3. The new goal is published when the robot is "sufficiently close" to the previous target.

---

## 5. References


- Stage worldfile: https://player-stage-manual.readthedocs.io/en/stable/WORLDFILES/
- ROS wiki Documentation: http://wiki.ros.org/Documentation