



## Laboratory exercise 5

# Mapping with known poses

Name:

JMBAG:

### Preparation and helpful instructions

- Review the lecture slides about coordinate system (a.k.a. frame) transformations and lidar sensors.
- Remember to add an appropriate shebang (`#!/usr/bin/env python3`) at the beginning of your Python scripts, to make them executable using `chmod +x my_script.py`, and to add the appropriate `__main__` wrappers around your script.
- Write clean, readable, easy to understand code. Give meaningful names to variables. Use classes or functions where you see fit. Code quality will impact your grade.
- Format your code according to the PEP 8 style guide. (Your IDE, e.g. PyCharm, Visual Studio Code, usually has a *Format code* command for this).
- Keep a separate terminal window/tab open running `roscore`. This is considered good practice in general when developing and testing with ROS.
- Make sure to run `roscpp set use_sim_time true` before running any nodes (or RViz) which will subscribe to topics from bags played with `roscpp play --clock`.
- *Hint*: to play a bag e.g. at 3 times faster rate and to skip the first 40 seconds, you can use `roscpp play --clock -r 3 -s 40`.
- Examine how to use ROS parameters in Python.  
[http://wiki.ros.org/roscpp\\_tutorials/Tutorials/Parameters](http://wiki.ros.org/roscpp_tutorials/Tutorials/Parameters).  
Example of reading private parameters and specifying them on command line when running the node:

```
roscpp.init_node('my_node')
# ~ denotes a private parameter.
# The second parameter is the default value and is optional.
robot_name = roscpp.get_param('~robot_name', 'alpha')
explode_on_startup = roscpp.get_param('~explode_on_startup', False)

$ ./my_node.py _robot_name:=delta _explode_on_startup:=true
```

### Assignment

#### Task 1: Warmup – Visualizing the trajectory

- a) What is your favourite colour? (Tip: do not select a dark colour.)

Find the RGB representation of your favourite colour. E.g. `r, g, b = (0, 0, 255)` for blue.  
Your favourite search engine has a tool for this if you search for *colour picker*.

- b) Write a ROS node called `trajectory_visualizer.py`. The node should have **two private** ROS parameters called `frame_id` (default value: `map`) and `child_frame_id` (default value: `base_link`).

- c) Print the values of the two parameters on startup, like so:

```
Starting the trajectory visualizer node.
frame_id: map, child_frame_id: pioneer/base_link
```

- d) Subscribe to messages of type `tf2_msgs/TFMessage` ([http://docs.ros.org/en/noetic/api/tf2\\_msgs/html/msg/TFMessage.html](http://docs.ros.org/en/noetic/api/tf2_msgs/html/msg/TFMessage.html)) on the `/tf` topic.
- e) In the `TFMessage` callback, go through all `geometry_msgs/TransformStamped` ([http://docs.ros.org/en/api/geometry\\_msgs/html/msg/TransformStamped.html](http://docs.ros.org/en/api/geometry_msgs/html/msg/TransformStamped.html)) in the `transforms` array of the received `TFMessage`. For those `TransformStamped` whose `header.frame_id` and `child_frame_id` match the appropriate private node parameters, copy the robot position from `transform.translation` and **append** it as a `geometry_msgs.Point` to the `points` array in a persistent `visualization_msgs/Marker` message.
- Hint:* Because it should persist (i.e. be saved after the callback has finished), the marker message is a good candidate for being a class member.
- Note:* Create a new `geometry_msgs.Point` object for each point before appending it to the marker. Do not reuse an existing one, or you may end up modifying a single point object every time due to the nature of Python names and their interaction with mutable objects.
- f) After appending the received position, publish the marker as a `LINE_STRIP` ([http://docs.ros.org/en/noetic/api/visualization\\_msgs/html/msg/Marker.html](http://docs.ros.org/en/noetic/api/visualization_msgs/html/msg/Marker.html)) marker on the topic `robot_positions`. Use frame id and stamp (i.e. the header) from the received transform.
- g) In the callback, if the timestamp of the currently received transform is older than the previously published marker timestamp, clear the `points` array in the marker. This means the bag playback has been restarted, so we should clear the trajectory. Print the following message when this happens: **Timestamp has jumped backwards, clearing the trajectory**.
- Hint:* at the beginning, `rospy.Time(0)` can be used as the initial marker stamp, which will always be older than the first received transform stamp.
- h) Display the marker in RViz. Set `map` as the global frame. Set the marker color (in your script) to use your favourite colour from a). Also adjust the scale (line thickness) to your liking. Start up the node with parameters given in c), and play back the bag provided with this exercise. Remember to follow the advice from the Preparation section of the exercise.
- Hint:* Don't forget to set `marker.color.a` to 1., `pose.orientation.w` to 1., and `scale.x` to your liking, otherwise the marker may not appear.
- i) Add the TF display in RViz (Add -> TF). Verify that the axes corresponding to the robot are matching with your marker for visualizing the trajectory.

## Task 2: Mapping with known poses

The task of this exercise is to write a node which will transform the points from the laser sensor frame to the global coordinate system, `map`.

The bag provided with this exercise contains a solution for robot localization — a trajectory, which is a set of timestamped poses/transformations which tell us where the robot is in the *world* (i.e. in a *fixed global coordinate frame*) throughout the duration of the drive. The solution has been computed using an algorithm named *simultaneous localization and mapping* (SLAM).

- a) Play the provided bag and, while the bag is playing, execute `roslaunch rqt_tf_tree rqt_tf_tree`. Which frames are in the tree? Which is the root frame in the tree, what is its child frame, and which frame is the leaf (without children frames)?

- b) What is the value of the fixed transform between the vehicle base frame and the laser sensor frame? (Write the translation vector and the orientation quaternion).

*Hint:* you can use `rostopic echo /tf` or `roslaunch tf tf_echo source_frame target_frame` to examine transforms.

- c) Start with your solution of Task 2 from Exercise 4 (`laserscan_to_points.py`), which shall be modified to solve this task. Read a **private ROS parameter** named `global_frame` with the default value `map`.
- d) Add a `tf2_ros.Buffer` to your node. Also, add a `tf2_ros.TransformListener`. Pass the `Buffer` instance to the constructor of `TransformListener`.
- e) In the `LaserScan` callback, look up the transform describing the global pose of the laser sensor using the `lookup_transform` method of `Buffer` where `global_frame` (ROS parameter) is the **target frame**, while the **source frame** should be read from the header of the received `LaserScan` message.

*Note:* `lookup_transform` can throw an exception, so you should wrap it in a `try-except` block. In case of catching an exception, print `Tf exception`, and `return` early from the `LaserScan` callback.

- f) The transformation is returned as a `geometry_msgs/TransformStamped` ([http://docs.ros.org/en/api/geometry\\_msgs/html/msg/TransformStamped.html](http://docs.ros.org/en/api/geometry_msgs/html/msg/TransformStamped.html)) containing a `Vector3` of the translation part of the transformation, while the rotation part is described using a quaternion.

Quaternions are an extension of complex numbers which has three imaginary units: **i**, **j**, and **k**. They are a mathematical tool which can be used to represent rotations in 3D space. The general formula relating a quaternion **q** and rotation by  $\theta$  radians around an axis given by unit vector **u** is:

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})$$

Rotations within the 2D  $xy$  plane can be viewed as yaw-only rotations around the  $z$  axis, i.e. **u** = **k**:

$$\mathbf{q}_z = w + z\mathbf{k} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \mathbf{k}$$

If  $z$  and  $w$  — the sine and cosine of an angle  $\frac{\theta}{2}$  — are known, the original angle  $\theta$  can be recovered using the inverse trigonometric function `atan2`:

$$\frac{\theta}{2} = \text{atan2}(z, w) \implies \theta = 2 \text{atan2}(z, w)$$

- i) Recover the orientation (yaw)  $\theta$  from the quaternion in the transform returned by `lookup_transform`.
- ii) Apply the transformation to previously calculated  $(x, y)$  points in the laser sensor frame in order to transform their positions to the global frame.
- Hint:* there are two options. The first option is to construct a numpy transformation matrix as described in the lectures for the angle  $\theta$  and the translation from the obtained transform. Then, convert the laser point into a vector and apply the transform by multiplying the transformation matrix with the vector.
- The other (easier) option is to add the orientation  $\theta$  to the previously calculated range measurement angle, and afterwards add the translation to the calculated point.
- iii) Change **the marker frame id to the global frame, given by the ROS parameter**.
- g) Make the marker a persistent variable (e.g. a class member) if you haven't already done so. Clear the points in the marker each time you receive a new scan. Also, at the beginning of the laser scan callback, insert the following code (the persistent variables `marker` and `buffer` may be class members in your implementation, so add `self.` where appropriate):

```
if received_message.header.stamp < marker.header.stamp:
    print('Timestamp has jumped backwards, clearing the buffer.')
    marker.header.stamp = received_message.header.stamp
    marker.points.clear()
    buffer.clear()
    return
```

This code will ensure your node keeps working after restarting the bag playback.

- h) Start RViz, ensure that the global frame is set to `map` and that the marker and `LaserScan` displays have been added, and play the provided bag. Verify the correctness of your code by making sure it matches with the RViz native `LaserScan` display. (Toggle the two displays on and off and make them have different colours to help you compare).
- i) Finally, add two additional **private** ROS parameters: `accumulate_points` (default: `False`) and `accumulate_every_n` (default: 50). If `accumulate_points` is true, do not clear the marker points each time you receive a scan, but keep (accumulate) them instead. However, when `accumulate_points` is true, allow only every `accumulate_every_n`-th scan to be processed. (For all other scans, return early from the callback, after the `if` from subtask g).

Print the values of the two parameters on startup, like so:

Starting the laser scan visualizer node.

```
global_frame: map, accumulate_points: True, accumulate_every_n: 50
```

- j) Accumulate the scans from the entire bag to create a point cloud map. Adjust the marker point scale and colour to your liking. Display the trajectory marker from Task 1 as well. Take a screenshot in RViz and name it `map.png`.

### Exercise submission

---

Create a zip archive containing **this pdf with the filled out answers** and **all other exercise files** (`trajectory_visualizer.py`, `laserscan_to_points.py`, `map.png`) and upload it on Moodle.