Laboratory assignment 3

Bag analysis, laser sensor visualization and mapping

Name: Sérgio Miguel Rosa Estêvão **JMBAG:**

Preparation and instructions

- **Do not: consult, read or examine ANY materials or solutions for assignments from the previous years' editions of this course, or from your colleagues. The assignment solution must *fully* be your own work. In case signs of plagiarism are detected, you will get a zero score and may be subject to be reported to the Faculty's Ethics committee.**
- **Do: consult the teaching staff (for this assignment, teaching assistant Juraj Oršulić) via Teams DM, if you have any problems, if anything is unclear, or if you need any help with any part of the assignment.**
- Review the lecture slides about `rosbag`, coordinate system (a.k.a. frame) and transformations. Examine how to define and use new ROS message types.
- 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 have `__main__`.
- Write clean, readable, easy to understand code. Give meaningful names to variables. Code quality will impact your grade. **Any deviation from the defined topic names, file names or the output format will result in points being deduced.**
- 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).
- When playing bags or running Stage, make sure to run `rosparam set use_sim_time true` before running any nodes (or RViz) which will subscribe to topics from bags. The bags should be played with `rosbag play --clock`.
- Keep a separate terminal window/tab open running `roscore`. This is considered good practice in general when developing and testing with ROS.
  If you use `roslaunch` without first having a `roscore` running separately, one will be started up automatically for you. However, if you afterwards run other nodes and then exit the initial `roslaunch` session, these ROS nodes will be orphaned since their `roscore` will be shut down with the initial `roslaunch` session.
- Create a Catkin package named `lab3_YOURJMBAG` for this assignment that will contain your solution.
- According to the best practices when writing a ROS node, the `__main__` part of the Python script should have three lines, and nothing else:
  - calling `rospy.init_node`
  - declaring an instance of the node class
  - calling `rospy.spin()`.

  Perform all other setup steps in the node class constructor method.

  Do not use `while not rospy.is_shutdown()` loops, `rospy.Rate`, sleep, or catch `ROSInterruptException` exceptions when calling `rospy.spin()`.
- In order to receive marks for tasks 2 and 4, it is necessary to submit the screenshots `map_fer.png` and `map_stage.png`.

Assignment

Task 1: Bag analysis

For this task, use the bags of turtle chases that you recorded in Lab assignment 2.

a) Examine the sample script `process_bag_example.py` which demonstrates how to loop through all messages in a bag and write them into another bag without any changes.

b) Write a Python script (which, like `process_bag_example.py`, is *NOT* a ROS node, but an *ordinary* Python script!) called `process_turtle_chase.py`. The script must take in **two** command-line arguments: the input bag filename, and the chase target pose topic name. The script should perform the following:

   i) Calculate and display the distance covered by the chaser turtle (you can assume that `turtle1` is the chaser turtle). To perform the covered distance calculation, sum up all Euclidean distances between sequential turtle positions, available in messages on the Turtlesim pose topic of the chaser turtle.
   Note: Be careful – do not set the initial position of the chaser turtle to (0, 0), and avoid adding a distance of "teleportation" from the origin to the first position read from the bag. At the beginning, you should initialize with `None` to indicate that you have not yet read a chaser turtle pose.

   ii) Calculate and display the *chase session duration*, defined by the difference between the serialization timestamps of the *first* and *last* `pose` topic messages of the chaser turtle in the bag. (See the comment in `process_bag_example.py` for an example of calculating time differences using `rospy.Time`.)

   iii) Calculate and display the average velocity for the chaser turtle by simply dividing the covered distance with the chase session duration.

   iv) In your Catkin package for the assignment, create a new ROS message type called `ChaserStatus`, consisting of these two `float64` message fields, in this order:
   `distance_travelled` and `distance_to_target`.

   v) Write and count the messages in a new bag called `<original filename>_processed.bag`, which will contain two topics:

      i. the chaser turtle pose topic, with all the original chaser pose messages written under a new topic named `/chaser/pose`;

      ii. a new synthesized `ChaserStatus` topic called `/chaser/status`, with one `ChaserStatus` message for each chaser turtle pose message.
      Make sure you use the timestamp of the chaser turtle pose message as the timestamp for the `ChaserStatus` message when writing it into the new bag.
      The field `distance_travelled` should contain the distance traveled so far by the chaser turtle, as described in i). Verify correctness of your calculated status messages: the first message should have distance zero, while the last message should contain the same distance as displayed in task i).
      The field `distance_to_target` should contain the current distance from the turtle to the current (last received) target position on the target pose topic – the length of the vector **d** as defined in Figure 3 of Lab assignment 2. If there hasn't been a target position message yet in the bag, set this field to -1.

The output of `process_turtle_chase.py` must have **EXACTLY** this format (the values listed are placeholder numbers for illustration purposes only and should not be considered meaningful).

```
$ ./process_turtle_chase.py chase_second_turtle_2023-11-05-14-24-49.bag /turtle2/pose
Processing input bag: chase_second_turtle_2023-11-05-14-24-49.bag
Target pose topic: /turtle2/pose
Chaser turtle statistics:
  Covered distance: 12.34 TurtleSim units
  Average velocity: 56.78 TurtleSim units/s
Chase session duration: 34.56 s
Wrote 1234 messages to chase_second_turtle_2023-11-05-14-24-49_processed.bag
```

Store the reprocessed Lab assignment 1 bags from Task 1 to your assignment solution Catkin package, in the directory `bags/`.

The task of the rest of this assignment is to write two nodes: one to visualize the robot trajectory, and one which will visualize the laser data. Then, you will upgrade the laser visualization node to transform the points from the laser sensor frame to the fixed frame (the global, world coordinate system) and create a map.

You will visualize laser sensor data from two sources: a bag captured by a real robot, and the Stage simulator. The bag provided with this assignment, `fer-bc-ground-floor.bag`, contains a set of laser data measurements from a real robot's mission around the Faculty building and 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 *global fixed coordinate frame*) throughout the duration of the mission.

Task 2: **Visualizing the trajectory**

a) Play the provided bag, `fer-bc-ground-floor.bag`. Make sure to use the `--clock` option.

Hint: You should skip the first 25 seconds in the bag when the robot is stationary with `--start <seconds>`. You can play back the bag with a faster speed factor using `--rate <factor>`. Use `--loop` to keep playing the bag repeteadly until you quit with Ctrl-C. Use the space key to pause/resume playback.

While the bag is playing, execute `rosrun rqt_tf_tree rqt_tf_tree`. Which frames are in the tree? Which is the root (world) frame in the tree, what is its child frame, and which frame is the leaf (i.e. the frame without children frames)?

In this TF tree, map is the root or "world" frame, pioneer/base_link is a child of map, and pioneer/laser is a child of pioneer/base_link.

map->pioneer/base_link->pioneer/laser

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).

Translation: [0.140, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]

Hint: you can use `rosrun tf tf_echo source_frame target_frame` to echo current values of transformations between frames.

c) Start RViz. Set the fixed frame setting in RViz to the correct world frame you determined in a). Add the TF display in RViz (Add -> TF). In the TF Display options, under *Frames*, show only the global world frame and the base link frame. Observe how the robot is moving with respect to the world frame.

d) Write a ROS node class `TrajectoryVisualization` in a script called `trajectory_visualization.py`. The node should have **two private** ROS parameters called `fixed_frame_id` (**default value:** `map`) and `robot_frame_id` (**default value:** `base_link`).

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

```
Starting the trajectory visualization node.
fixed_frame_id: map
robot_frame_id: base_link
```

f) Add a `tf2_ros.Buffer` to your node, and a `tf2_ros.TransformListener`. Pass the `Buffer` instance to the constructor of `TransformListener`.

g) Add a 30 Hz `rospy.Timer` to your node.

Note: When creating the `Timer`, pass `reset=True` to the `Timer` constructor to make it well-behaved when the simulation time resets.

In the timer callback function, look up the transform which indicates the pose of the given robot frame in the given fixed world frame using the `lookup_transform` method of `Buffer`.

In this task, you can pass to `lookup_transform` the a default-constructed `rospy.Time()` as the lookup time argument (3rd argument). This returns the *last known pose of the robot.*

Note: `lookup_transform` can throw an exception, so you should wrap the call to this function in a `try-except` block. In case of catching an exception, print `Tf exception: <the exception>`, and `return` early from the timer callback.

If the robot position **has changed** since the last call of the timer callback, create a new `geometry_msgs/Point` message and set its contents to the robot position.

Then, append the new `Point` into the points array in a persistent `visualization_msgs/Marker` message. (See the provided sample code `circle.py` for an example of creating a marker.)

If the robot position since the last call has **not changed**, **do not** create a `Point` and **do not append** the same `Point` into the marker.

Hint: Because it should persist (i.e. be saved after the callback has finished), the `Marker` message is a good candidate for being a member variable of your ROS node class.

Note: Create a new `geometry_msgs/Point` object for each position before appending it to the trajectory marker. Do not reuse an existing point message, or you may end up modifying a single object every time due to the nature of Python names and their interaction with mutable objects.

h) After appending the received position, publish the updated marker as a `LINE_STRIP` marker (see `https://wiki.ros.org/rviz/DisplayTypes/Marker`) on the topic `robot_positions`. Use the frame id and stamp (i.e. the entire header) from the obtained `TransformStamped` for the marker.

i) In the callback, if the timestamp of the currently received transform is older than the previously published marker message timestamp, clear the points array of the persistent marker message. This means the bag playback or the simulation has been restarted, so we should clear the visualized trajectory. Print the following message when this happens:
`Timestamp has jumped backwards. Clearing the trajectory marker.`

Hint: at the beginning, `rospy.Time(0)` (which is the same as a default-constructed `rospy.Time()`) can be used as the initial marker message stamp, which will always be *earlier* than the first received transform stamp.

j) Set the marker `color` to use a favourite colour of yours. Also make sure to set marker `color.a` to 1.0, `pose.orientation.w` to 1.0, and `scale.x` to your liking (make sure the marker is thick enough so it is well visible in RViz).

Hint: You should set up the marker type, color and thickness in the trajectory visualization node constructor.

k) Display your trajectory marker in RViz. (Read Task 2, subtask a) for more details on adding topic displays in RViz.) Rename the marker display to an appropriate name, e.g. "Robot trajectory positions".

Run your trajectory visualization node, and use the information from subtask a) to set the correct frame id parameters of the node. Verify that the TF display axes corresponding to the robot's base link frame are matching with your marker for visualizing the trajectory. See Figure 1 for an example.

What is the full command line to run the trajectory visualization node to work with the provided bag?

rosrun lab3_0036563035 trajectory_visualization.py _fixed_frame_id:=map
_robot_frame_id:=pioneer/base_link

l) After setting up everything RViz, press Ctrl-S or *File -> Save Config* to save the current settings as `lab3.rviz` in `rviz/` in your Catkin package for the assignment solution.

m) Now try your trajectory visualization node with the Stage simulator robot, and you controlling the robot using `teleop_twist_keyboard.py`. What is the full command line to run your node in order to visualize the trajectory from the robot in Stage?

rosrun lab3_0036563035 trajectory_visualization.py _fixed_frame_id:=odom
_robot_frame_id:=base_laser_link

Hint: Make sure to change the fixed frame in RViz to the world frame used by the Stage simulator.

n) **See Task 4.h) for instructions on taking screenshots. Submission of screenshots is mandatory to receive marks for this task.**

Task 3: Laser sensor visualization

In this task, you will learn how to visualize range data from a laser sensor (lidar). We will use real-world 2D laser sensor data captured in the Faculty building.

a) Run and thoroughly examine the provided ROS node script, `circle.py`. This script is sample code you are allowed (and encouraged!) to use to help you get started in solving this task.

The `circle.py` script calculates a set of points from a circle. The circle radius is animated only to help illustrate that we are trying to visualize data which will change in real time. The points are stored in the `points` field of a message of type `visualization_msgs/Marker` and published on the topic `points`.

The reason for why a parameterized circle was chosen for visualization in sample script is because it is closely related to the task of visualizing the laser rangefinder data, as it will soon become apparent.

To visualize the circle points in RViz, run RViz (`rviz`), set the camera type to *TopDownOrtho* in the right *View* panel. In the left panel, in *Global options*, set the *Fixed frame* to match the frame id in `header.frame_id` in the points marker message published in `circle.py` (you may have to type it by yourself!). Finally, add a *Marker* display using the *Add* button on the bottom of the left panel.

If adding display by type (first tab in the *Add* window), you need to set the topic to `points` afterwards in the left panel, under the options of the `Marker` display. Alternatively, in the *Add* window, you can go to the second tab (*By topic*) and select the topic directly, which will automatically determine the display type (a `Marker` display).

b) Study the definition of the `sensor_msgs/LaserScan` message.

Examine the provided bag `fer-bc-ground-floor.bag` using `rosbag info`. What is the name of the laser scan topic in the provided bag?

The laser scan topic in the provided bag (fer-bc-ground-floor.bag) is "/pioneer/scan"

Helpful tips for using `rostopic echo`: you can use `rostopic echo` to examine only parts of a message. For example, to view timestamps in the header of a `LaserScan` message on a topic named `scan`, you can use the following command: `rostopic echo scan/header/stamp`. To print only a single (1) message, you can pass `-n1`. To print messages straight from a bag without having to play it using `rosbag play`, you can pass `-b bagname.bag` to `rostopic echo`.

c) In the following subtasks, let j be the last two digits of your JMBAG + 100. For example, if your JMBAG is 0036465831, $j = 131$.

$j =$ 135

d) Write the full `rostopic echo` command for printing the `angle_min` field of the `LaserScan` message, for **first** j messages in `fer-bc-ground-floor.bag`. Note: all angles are in radians.

rostopic echo /pioneer/scan/angle_min -b fer-bc-ground-floor.bag -n 135

What is the value of `angle_min`?

-2.356194496154785

What is the value of `angle_increment`?

0.008726646192371845

The formula for calculating the angle of the range measurement with an index i (where the first measurement has the index of 0) is: $angle = angle\_min + i * angle\_increment$. What would be the angle for a range measurement with the index j as defined in c)?

−1.17809726

e) Note: **before running any nodes (or RViz)** which will be visualizing data **from a played bag or the Stage simulator**, make sure you run `rosparam set use_sim_time true` to use the simulated clock from the bag/Stage instead of the wall (system) clock. Make sure you have a `roscore` running in a separate tab.

(For running the sample `circle.py` from subtask a) which animates the circle based on the system clock, `use_sim_time` should be set to `false`, which is the default value after after starting up `roscore`.)

f) Write a script `laserscan_to_points.py` for a ROS node `LaserScanToPoints` based on the sample `circle.py` script. This ROS node must subscribe to a `sensor_msgs/LaserScan` topic named `scan` (**not directly to the topic from `fer-bc-ground-floor.bag`**).

Note: `LaserScanToPoints` should not use a `rospy.Timer` – only a subscriber to `LaserScan` messages.

Instead of calculating the points of a circle, calculate the Cartesian (x, y) points using the range measurements in received laser scans. Subtask d) illustrates how to calculate the angles of range range measurements.

Set the frame id in the `Marker` marker header to the frame id from the received `LaserScan` message. (**Do not hardcode it, read it from the received message.**) Also copy the timestamp from the received `LaserScan` message. You can accomplish both of these at once by simply copying the entire `header`.

g) Which topic remapping will you use when running `laserscan_to_points.py` to visualize the scans from `fer-bc-ground-floor.bag`? Enter the full command line.

rosrun lab3_0036563035 laserscan_to_points.py scan:=/pioneer/scan

Start the `laserscan_to_points.py` node with the command line above. Play the provided bag using the command `rosbag play --clock fer-bc-ground-floor.bag`.

Hint: you can pass `--loop` to keep looping the bag until you quit with Ctrl-C. Your solution must work with looping.

h) What is the frame id (in the header) of the `LaserScan` messages in `fer-bc-ground-floor.bag`?

"pioneer/laser"

How did you examine this?

rostopic echo /pioneer/scan/header/frame_id -b fer-bc-ground-floor.bag -n1

Start RViz with the setup **as described in a)**, and change the *Fixed frame* to the above frame id.

Finally, in RViz, add a `LaserScan` display for the `LaserScan` topic.

Adjust the points marker scale and color in the script to your liking, as well as `LaserScan` display colors in the left panel in RViz. Be sure you can tell them apart. It is recommended to set the *Style* property to *Points* for the `LaserScan` display.

Make sure that your laser scan visualization node is working correctly and that the points published by `laserscan_to_points.py` on the `points` topic match the `LaserScan` display in RViz. **If it looks different, you have incorrectly solved the subtask f) and will not receive marks for Task 3!**

Save a copy of the current version of `laserscan_to_points.py` as `laserscan_to_points_task3.py` before moving on to Task 4.

i) As in Task 2, try out your node in real-time with Stage. Which topic remapping will you use when running `laserscan_to_points.py` to visualize the scans from the robot simulated in Stage? Enter the full command line.

rosrun lab3_0036563035 laserscan_to_points.py scan:=/base_scan

Task 4: **Mapping with known poses**

a) Start with your solution of Task 3, which you will be modify to solve this task. Add a **private ROS parameter** to your node named `fixed_frame_id`, with the default value `map`.

b) Add a `tf2_ros.Buffer` to your node, and a `tf2_ros.TransformListener`. Pass the `Buffer` instance to the constructor of `TransformListener`.

c) In the `LaserScan` subscriber callback, look up the transform describing the world pose of the laser sensor using the `lookup_transform` method of `Buffer`, where `fixed_frame_id` is the **target frame**, while the **source frame** should be read from the header of the received `LaserScan` message.

   Make sure you look up the pose of the robot exactly at the timestamp stored in the laser scan message header! Recall that `lookup_transform` takes in the lookup time as the 3rd argument.

   Pass a lookup timeout value (the 4th argument to `lookup_transform`) of 0.2s, i.e., `rospy.Duration(0.2)`.

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

   Note: In the laser scans in the provided bag, invalid range measurements (where the sensor did not register a return beam) have range of *exactly* zero. They will appear right at the origin of the laser frame, and will leave a trail along the trajectory. Discard/filter these points, as well as points with range larger than `range_max` from the `LaserScan` message.

   Note: In case you use `+=` to calculate the measurement angle in the loop, do not forget to perform the increment, even if you are skipping processing these points.

d) The transformation from `lookup_transform` is returned as a geometry_msgs/TransformStamped, which contains 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 θ radians around an axis given by unit vector **u** is:

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

   In our use case, rotations within the 2D xy plane are yaw-only rotations around the z axis, i.e. $\mathbf{u} = \mathbf{k}$:

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

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

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

   i) Recover the orientation (yaw) angle θ from the quaternion in the transform returned by `lookup_transform`.

   ii) After calculating the Cartesian laser scan points (x, y) in Task 3, transform them from the laser sensor frame to the world fixed frame.

   Hint: there are a couple options. The first option is to construct a `numpy` or `PyKDL` transformation matrix as described in the lectures for the angle θ and the translation from the obtained transform. Then, convert the Cartesian point of the laser measurement into a vector and apply the transform by multiplying the transformation matrix with the vector.

   The other (easier) option in this case to simply add the global orientation θ to the previously calculated range measurement angle, and after calculating the sine/cosine, to simply add the translation to the calculated point.

iii) In order to indicate that points in the marker are now expressed in the global world fixed frame, change **the marker frame id in the header to the global world fixed frame given by the ROS parameter added in subtask a)**.

e) Make the points marker a persistent variable (i.e. a node class member). For now, clear the points array in the marker each time you receive a new scan and enter the callback function. Also, at the beginning of the laser scan callback, insert the following code (adjusted for your variable names):

```
if received_message.header.stamp < self.points_marker.header.stamp:
    print('Timestamp has jumped backwards. Clearing the buffer.')
    self.points_marker.header.stamp = received_message.header.stamp
    self.points_marker.points.clear()
    self.tf_buffer.clear()
    return
```

This code will ensure your node keeps working after restarting the bag playback (e.g. with `--loop`). Make sure this does work correctly after playback is restarted.

f) Start RViz, ensure that the fixed frame is set to `map` and that the points marker and `LaserScan` displays have been added, and play the provided bag. Verify the correctness of your code by making sure your published points marker matches with the RViz `LaserScan` display. (Toggle the two displays on and off and make them have different colours to help you compare). **If you do not solve this subtask correctly, you will not receive marks for Task 4.**

Store the updated RViz configuration in `lab3.rviz`.

g) Add two additional **private** ROS parameters: `accumulate_points` (default value: `False`) and `accumulate_every_n` (default value: 50). Print the values of the parameters on startup, like so:

```
Starting the LaserScan to points node.
fixed_frame_id: map
accumulate_points: True
accumulate_every_n: 50
```

If `accumulate_points` is true, in the scan callback function, do not clear the marker points each time you receive a scan, but keep (accumulate) them instead.

However, when `accumulate_points` is true, **process only every `accumulate_every_n`-th scan**. (For all other scans, return early from the `LaserScan` callback, immediately after the `if` from subtask e).

Hint: Add a counter to the node class which will persist between calls of the laser scan callback function.

Furthermore, if we passed the checks and are about to process the received scan, do one more check: if the pose received from `lookup_transform` is identical to the pose of the previously processed scan (if there is one), treat this scan as already processed and also return early from the laser scan callback.

h) Accumulate the scans from the entire bag to create a map. Adjust the points marker scale and colour to your liking. Display the trajectory marker from Task 2 as well. Take a screenshot in RViz and name it `map_fer.png`. See Figure 1 for an example.

After creating the marker points map from laser scans in the provided bag, repeat the same with the Stage simulator. Drive your robot around `simple_rps.world` while your trajectory and laser visualization nodes are running. Take a screenshot in RViz and name it `map_stage.png`. See Figure 2 for an example. Save the RViz configuration for Stage as `lab3_stage.rviz`.

While exploring in Stage for this subtask, record a bag with **only** the `/tf` topic and the `LaserScan` (NOT marker!) topic, with the bag prefix `stage_exploration`. Store the bag in the Catkin package for the assignment.

Hint: You will need to adjust the frame id parameters and topic remappings for creating a map with the robot in Stage, as well the RViz fixed frame. You should be able to use a much lower value (e.g. 1) than the default 50 for `accumulate_every_n` for Stage.

This subtask (creating screenshots) is mandatory to receive marks for Task 2 (trajectory visible on screenshots) and Task 4 (accumulated points in the marker visible in the screenshots).

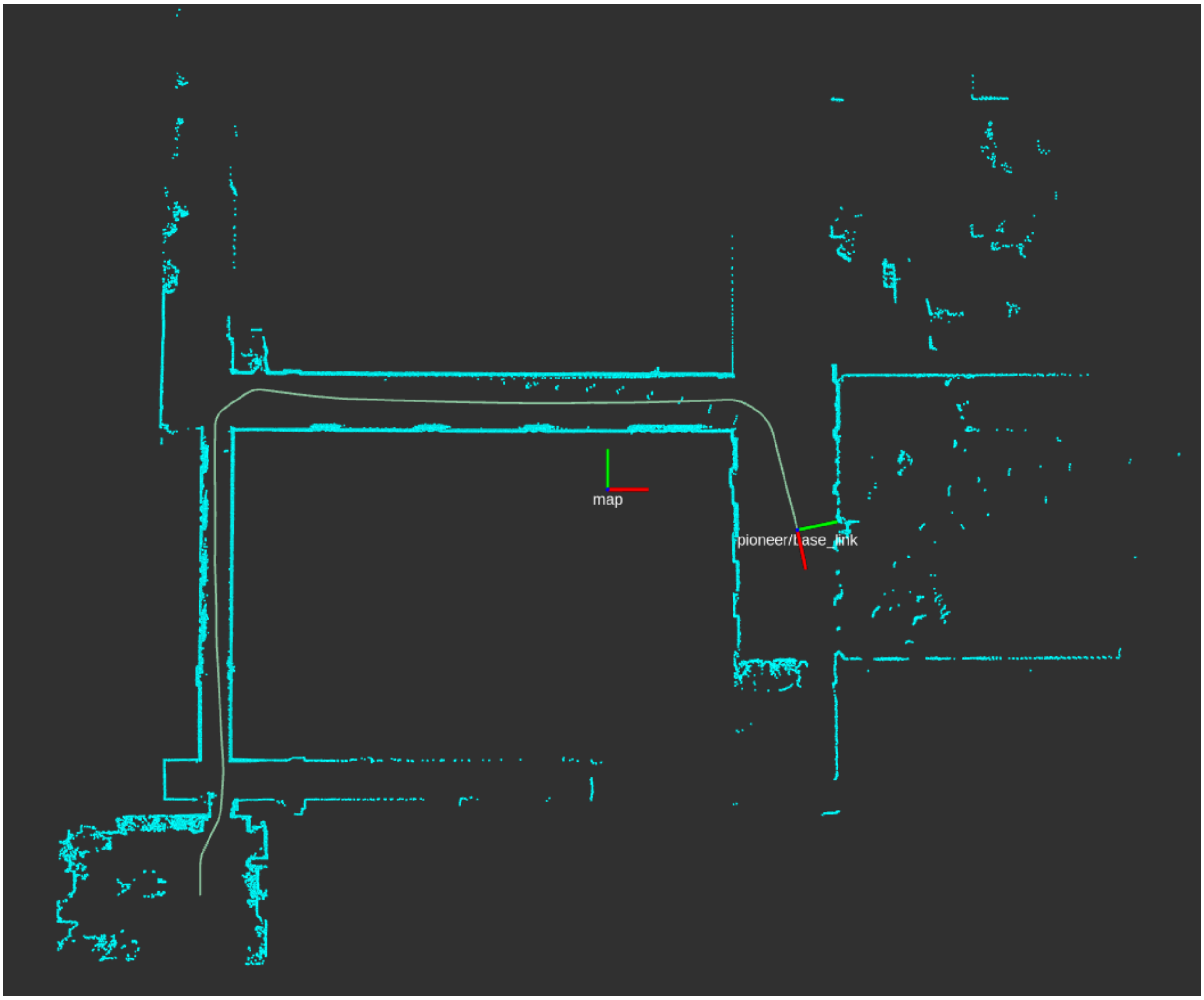

Figure 1: Example `map_fer.png` screenshot for subtask h), showing successful completion of Tasks 2, 3 and 4. **Submission of `map_fer.png` is mandatory for receiving marks for Tasks 2 and 4.**

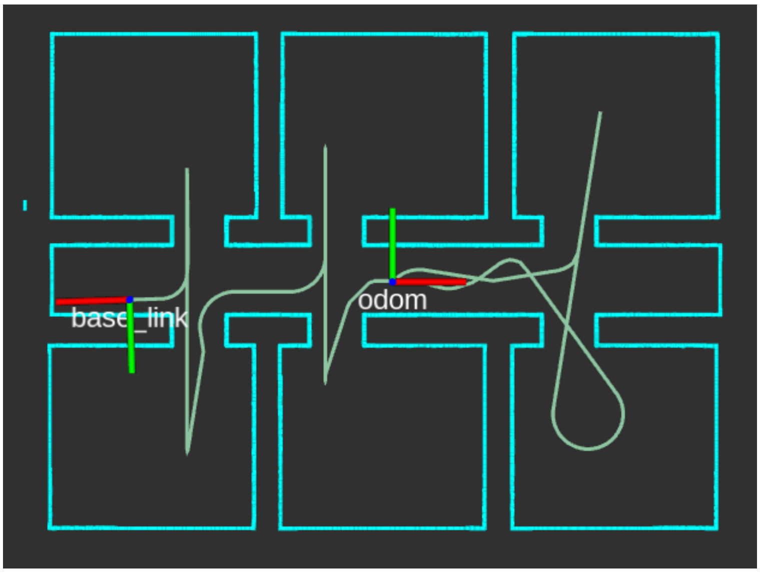

Figure 2: Example `map_stage.png` screenshot for subtask h), showing successful completion of Tasks 2, 3 and 4 with the Stage simulator. **Submission of `map_stage.png` is mandatory for receiving marks for Tasks 2 and 4.**

Some additional notes:

- The command line remapping specifier `:=` should not have spaces around it.
- Do not change the `points` topic name in the code.
- Until closed with Ctrl-C, trajectory and laser visualization nodes must keep working and clear the markers when bag playback/simulation are restarted.
- The points marker should be published from the laser scan message callback.
- The timer and the initial time are only used for animating the circle in the example. They should not be used in any way in the node for visualizing laser scans.
- **In case you need help with solving the assignment by yourself, have any difficulties or questions, contact the course staff and you will receive support with solving the assignment. Do not use solutions written by others.**

Assignment submission

Create and upload to Moodle a zip archive containing **this pdf with the filled out answers** and **all other assignment files inside the Catkin package** `lab3_YOURJMBAG`:
`process_turtle_chase.py`, `chase_second_turtle*_processed.bag`, `chase_mouse*_processed.bag`, `trajectory_visualization.py`, `laserscan_to_points.py`, `laserscan_to_points_task3.py`, `map_fer.png`, `map_stage.png`, `stage_exploration_*.bag`, `lab3.rviz`, `lab3_stage.rviz`.

Please note that it is mandatory to submit `map_fer.png` and `map_stage.png` in order to receive marks for Tasks 2 and 4.

Do not submit duplicate files, or place .zip inside .zip. Do not include the entire workspace, e.g. the `build` and `devel` folders; include only the Catkin package. In your Catkin package, scripts should be placed inside `scripts/`, RViz configurations inside `rviz/`, bags inside `bags/`, and images and the filled out assignment PDF inside `media/`.