TECHNISCHE UNIVERSITÄT WIEN

Autonomous Racing Cars
191.119 (VU 4,0) Semester: 2022S

Lab 2: ROS Basics & Safety Node

2021-03-15

Preface

Read all the instructions below carefully before you start working on the assignment, and before you make a submission. All sources of material and ressources must be properly cited (this also includes datasheets).

- Completeness of solution: A complete solution of a task also includes knowledge about the theory behind.
- Exercises are to be solved in teams. All team members must be indicated on the submission protocol. However, every team member must be able to explain the handed in solution. Grading is on an individual basis. Upload your solution (one per team) in TUWEL until 2021-03-29 23:59.
- For this assignment there is no exercise interview. However if submissions are unclear, students might be invited for exercise interviews.
- One team will additionally present their work at the *Lab 2 presentation*.

Learning outcomes

The following fundamentals should be understood by the students upon completion of this lab:

- Understand the directory structure and framework of ROS, implement publishers and subscribers in ROS.
- Understanding Cmake lists, package.xml files and dependencies in ROS.
- Working with launch files, RViz and Bag files.
- Use the LaserScan, Odometry, AckermannDriveStamped and other messages in ROS.
- Work with Time to Collision (TTC) calculation.

Deliverables and Submission

Write a exercise report and submit it as PDF file in TUWEL until 2021-03-29 23:59. Use the provided latex template and do not forget to fill in the parts marked with "TODO". If you do not use the provided template, make sure to include the same information. In any case the report must meet an adequate level for layout and readability that is appropriate for the academic context.
If you are using any program or script (e.g. *Matlab*, *Python*, *C/C++*) to solve an assignment, you must add the full sourcecode in the appendix of your PDF file and reference it in your description of the solution.

Transparency of Contribution

Describe in your submission protocol briefly how you worked together. How did you structure your work distribution and collaboration? Who contributed how much effort to which part of the work? (If one, or more team members, are not able to work on this assignment, you must also transparently state this here.)
(Please do not understand this preamble wrong to somehow exaggerate your contribution estimation: If you are working together well in your team, it should anyway be no problem to briefly describe how you worked together.)

1 ROS Basics

As introduced in the lecture, ROS nodes exchange messages based on so-called topics. This task will deal with this type of communication.

1.1 ROS Workspace and package

In task (3.) of the first exercise sheet you have already created a workspace (in directory `~/catkin_ws`). Read the tutorials on how to create a workspace [Sta22c] and how to create a ROS package [Sta22b].
Answer the following questions:

a.) What is a `Cmakelist.txt`? Is it related to a `Makefile` used for compiling C++ objects? If yes, then what is the difference between the two?

b.) Are you using `Cmakelist.txt` for python in ROS? Is there a executable object being created for python?

c.) Where would you run the command `catkin_make`? In which directory?

d.) The command `source` was used in the task (3.) of the first exercise sheet (and is also described in [Sta22c]). What is the significance of sourcing the setup files? What is the difference of the used files in this command (e.g. `/opt/ros/noetic/setup.bash` and `devel/setup.bash`)?

1.2 ROS Messages

This section deals with ROS messages, their data and how to show/analyse them in the terminal.
For this, start the simulator with command `roslaunch f1tenth_simulator simulator.launch` in one terminal window, and execute the further commands in another terminal window:

a.) The command `rostopic list` gives you a list of topics which are currently published. Include this list in your submission document.

b.) What is the ouput of the command `rostopic info /scan`?
What information does it give? Describe the information.

c.) Use the command `rostopic echo` to get the content of messages published on the topic `/scan`.
What is the exact command to print these messages continously?
What is the exact command to print just one of these messages?
(Hint: `rostopic echo --help`)

d.) How does the ouput of the previous task (1.2.c.) relate to the task (1.a.) of the first exercise sheet?

e.) The command `rostopic info` gives you the message type of a topic.
What is the type for the topic `/scan`?
Use `rosmsg show` to see the structure of this type. Include this in your submission document.
(Hint: `rosmsg show --help`)
You might also have a look at the data structure documentation [Sta22f] and the tutorial how to create custom structures for messages [Sta22a].

1.3 ROS Bags

This section deals with how to record and play back data with ROS bags. Read the tutorial on how to work with ROS bags [Sta22j].

Answer the following questions:

a.) Where does the bag file get saved? How can you change where it is saved?

b.) What is the exact command to record just the messages from the topics `/scan` and `/odom`?

2 Automatic Emergency Brake (AEB) with Time to Collision (TTC)

The goal of this task is to develop a safety node for the race cars that will stop the car from collision when travelling at higher velocities. We will implement TTC using the `LaserScan` messages in the simulator.

Relevant Messages

LaserScan Message

Recall from the previous task that each `LaserScan` message (topic `/scan`) contains several fields that will be useful to us. You'll need to subscribe to the scan topic and calculate TTC with these `LaserScan` messages.

Odometry Message

Both the simulator node and the car itself publish `Odometry` messages [Sta22e]. Within its several fields, the message includes the car's position, orientation, and velocity. (Some of them, like position, are not filled with proper values on the hardware race car.) You'll need to explore this message type in this task.

AckermannDriveStamped Message

`AckermannDriveStamped` is the message [Sta22d] type that we'll use throughout the course to send driving commands to the simulator and the car. Note that we won't be sending driving commands to the car from this node, we're only sending the brake commands. By sending an `AckermannDriveStamped` message with the velocity set to 0.0, the simulator and the car will interpret this as a brake command and hit the brakes.

The TTC calculation

Time to Collision (TTC) is the time it would take for the car to collide with an obstacle if it maintained its current heading and velocity. Between the car and its obstacle, we can calculate it as:

$$TTC = \frac{r}{[-\dot{r}]_+}$$

where r is the distance between the two objects and $\dot{r}$ is the time derivative of that distance. $\dot{r}$ is computed by projecting the relative velocity of the car onto the distance vector between the two objects. The operator $[]_+$ is defined as: $[x]_+ := \max(x, 0)$.
You'll need to calculate the TTC for each beam in the laser scan. Projecting the velocity of the car onto each distance vector is very simple if you know the angle between the car's velocity vector and the distance vector (which can be determined easily from information in the `LaserScan` message).

2.1 ROS Node: safety_node

For this lab, you will make a Safety Node that should halt the car before it collides with obstacles. To do this, you will make a ROS node that subscribes to the `LaserScan` and `Odometry` messages. It should analyze the `LaserScan` data and, if necessary, publish an `AckermannDriveStamped` with the velocity field set to 0.0 m/s, and a `Bool` message [Sta22g] set to `True`. The `AckermannDriveStamped` message will be received by the Mux node and the `Bool` message will be received by the Behavior Controller, which will then tell the Mux node to select the appropriate Drive message.

Note the following topic names for your publishers and subscribers (also included in the skeleton code):

- LaserScan: `/scan`
- Odometry: `/odom`
- Bool message: `/brake_bool`
- Brake message: `/brake`

In ROS nodes can be implemented either in C++ or Python. We want you to be able to understand both Python and C++ because it might happen that third-party nodes are written in either the one or the other programming language. For this exercise sheet it is required to implement both a Python and C++ node. Later labs may be solved either in C++ or Python (details will follow on this).

a.) Create a new ROS package in your Workspace with name `safety_node1`. (Refer to section *1.1 ROS Workspace and package* of this exercise sheet how to create a ROS package.) Set up the required files and copy the skeleton file `safety_node.cpp` in the respective source folder. The skeleton file is provided in the archive `exercise2_report_template_and_data.tar.bz2` in TUWEL.
Implement the automatic emergency brake as described using C++ in this file.
Add your `safety_node.cpp` in the appendix of your submission document.

b.) Add a launch file [Sta22k],[Sta22l] in the package `safety_node1` which starts the simulator (using the launch file `simulator.launch` in package `f1tenth_simulator`) and your node which is implemented in file `safety_node.cpp`.
Add your launch file in the appendix of your submission document.

c.) Create a new ROS package in your Workspace with name `safety_node2`. Set up the required files and copy the skeleton file `safety_node.py` in the respective scripts folder. The skeleton file is provided in the archive `exercise2_report_template_and_data.tar.bz2` in TUWEL.
Implement the automatic emergency brake as described using Python in this file.
Add your `safety_node.py` in the appendix of your submission document.

d.) Add a launch file [Sta22k],[Sta22l] in the package `safety_node2` which starts the simulator (using the launch file `simulator.launch` in package `f1tenth_simulator`) and your node which is implemented in file `safety_node.py`.
Add your launch file in the appendix of your submission document.

e.) We are working on a system (testbench) that automatically checks your code for the automatic emergency brake. For this you need to upload a ZIP-File of your node containing the following files (use exactly these names for directories and files). As soon as the system is available in TUWEL we will update the assignment sheet with the relevant information. If the system might not be available soon, we will require you to make a video (similar to the first exercise sheet).

Directory structure for python implementation:

```
scripts
  safety_node.py
CMakeLists.txt
package.xml
```

Directory structure for C++ implementation:

```
src
  safety_node.cpp
CMakeLists.txt
package.xml
```

Note: Make sure to press `b` on your keyboard in the terminal window that launched the simulator. This will activate the AEB and allow the Behavior Controller to switch the Mux to Emergency Brake when the boolean is published as `True`. Pressing `b` again will toggle the AEB off.

Information: In this exercise sheet this task is in simulation only. There will be a sub-task in a later exercise-sheet to test your Safety Node on the hardware race car.

2.2 Data visualization and parameter adjustments

In ROS there are tools for data visualization (`rqt_plot` [Sta22h]) and parameter adjustments (`rqt_reconfigure` [Sta22i]) available. These are useful to analyze internals of your running nodes and adjust their parameters.

a.) Use `rqt_plot` to visualize the following data over time (you might need to publish additional topics in your node from task (2.1) for this):

- velocity of the car
- TTC
- minimum distance to any obstacle (minimum for all rays in `LaserScan` messages)
- information, if the emergency brake is engaged

Describe the needed modifications in your submission document and add a screenshot showing the plots.

b.) Adjust the node which you implemented in task (2.1) such that you can modify the threshold for TTC using `rqt_reconfigure`. Note that your node must also work if `rqt_reconfigure` is not used (e.g. use a reasonable default value). Describe the needed modifications in your submission document and add screenshots to show the behaviour for different TTC-thresholds.

Appendix

Grading

The following points can be achieved for each task of this exercise sheet:

Exercise	Points
1.1.a.	1
1.1.b.	1
1.1.c.	1
1.1.d.	2
Subtotal	5
1.2.a.	1
1.2.b.	2
1.2.c.	1
1.2.d.	2
1.2.e.	2
Subtotal	8
1.3.a.	2
1.3.b.	1
Subtotal	3
2.1.a.	23
2.1.b.	3
2.1.c.	23
2.1.d.	3
2.1.e.	15
Subtotal	67
2.2.a.	7
2.2.b.	10
Subtotal	17
Grand Total	100

FAQ

Q: How should the TTC threshold be determined?
A: We recommend trial and error on a variety of situations (e.g. driving down the hall vs straight at a wall) to minimize false positives while preventing crashes. You can get a good initial estimate using the car's deceleration ($8.26\ \frac{m}{s^2}$) and it's velocity for keyboard driving ($1.8\ \frac{m}{s}$).

Q: Should "false" be published on the `brake_bool` topic when the AEB is not in effect?
A: Yes. Publishing "false" will make sure that no Drive messages on the `brake` topic are sent to the Simulator, and will allow other controllers (e.g. keyboard) to take back control.

Q: If the AEB is not engaged, should a `AckermannDriveStamped` message be published on the `brake` topic?
A: No, the `brake` topic is meant to only have messages that bring the car to a stop, so there is no need to send anything in this case. Moreover, if the AEB is not engaged, messages on the `brake` topic will not be sent to the Simulator.

Q: Does the order of messages published matter? That is, is there a difference between:

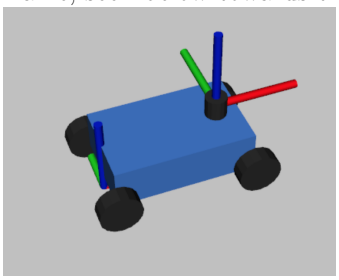
```
brake_publisher.publish(brake_msg);
brake_bool_publisher.publish(brake_bool_msg);
```

and

```
brake_bool_publisher.publish(brake_bool_msg);
brake_publisher.publish(brake_msg);
```

A: No. Due to latency between sending and receiving messages, this realistically won't matter.

Q: How are the coordinate frames of the `LaserScan` and `Odometry` data oriented with respect to the car?
A: The `Odometry` data is in the frame centered on the rear axle, seen below. The message gives the magnitude and sign of the car's velocity. It is always directed in the `+x` direction (red axis). The `LaserScan` data is in the LiDar's frame, seen below towards the front of the car.



Q: When testing driving forwards and in reverse, the car's distance from the wall after braking seems different between these two cases. Am I doing something wrong, or should I try and use different TTC thresholds?
A: The ranges in the `LaserScan` message are from the LiDar's frame (seen above) which is not in the center of the car. An ideal AEB system would subtract the distance from the LiDar to the edge of the car for each beam, and use that distance in the TTC calculation. A two-threshold system could work too, but for this assignment, as long as it doesn't crash during keyboard teleop, a single threshold value is good enough. More sophisticated solutions are always welcome!

Q: After the car is stopped by the AEB, if I drive the car towards the wall again, it will be stopped by the brake again. But if I repeat this, the car will move closer and closer to the wall until a collision occurs before the AEB can stop the car. Is this be acceptable or should the AEB prevent collisions in all circumstances?
A: A collision in this unique situation is fine. When the car is extremely close to the wall and begins to move, it is impossible to stop it in time.

References

[Sta22a] Stanford Artificial Intelligence Laboratory et al. *Creating a ROS msg and srv*. 2022. URL: http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv.

[Sta22b] Stanford Artificial Intelligence Laboratory et al. *Creating a ROS Package*. 2022. URL: http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

[Sta22c] Stanford Artificial Intelligence Laboratory et al. *Creating a workspace for catkin*. 2022. URL: http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

[Sta22d] Stanford Artificial Intelligence Laboratory et al. *Documentation: api/ackerman_msgs/AckermannDriveStamped*. 2022. URL: http://docs.ros.org/noetic/api/ackermann_msgs/html/msg/AckermannDriveStamped.html.

[Sta22e] Stanford Artificial Intelligence Laboratory et al. *Documentation: api/nav_msgs/Odometry*. 2022. URL: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html.

[Sta22f] Stanford Artificial Intelligence Laboratory et al. *Documentation: api/sensor_msgs*. 2022. URL: http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/.

[Sta22g] Stanford Artificial Intelligence Laboratory et al. *Documentation: api/std_msgs/Bool*. 2022. URL: http://docs.ros.org/noetic/api/std%5C_msgs/html/msg/Bool.html.

[Sta22h] Stanford Artificial Intelligence Laboratory et al. *Documentation: rqt_plot*. 2022. URL: http://wiki.ros.org/rqt_plot.

[Sta22i] Stanford Artificial Intelligence Laboratory et al. *Documentation: rqt_reconfigure*. 2022. URL: http://wiki.ros.org/rqt_reconfigure.

[Sta22j] Stanford Artificial Intelligence Laboratory et al. *Recording and playing back data*. 2022. URL: http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data.

[Sta22k] Stanford Artificial Intelligence Laboratory et al. *Roslaunch*. 2022. URL: http://wiki.ros.org/roslaunch.

[Sta22l] Stanford Artificial Intelligence Laboratory et al. *Roslaunch XML*. 2022. URL: http://wiki.ros.org/roslaunch/XML.

Acknowledgments

This course is based on F1TENTH Autonomous Racing which has been developed by the Safe Autonomous Systems Lab at the University of Pennsylvania (Dr. Rahul Mangharam) and was published under CC-NC-SA 4.0 license.