Chair of Cyber-Physical Systems in Production Engineering
Department of Mechanical Engineering
Technical University of Munich

F110 ROS simulator

denis.hoornaert@tum.de

August 19, 2021

Objective

The present section aims at introducing the reader on how to start the simulator, how the folder is structured, what is the minimal configuration and show a small demo.

5.1 Project structure

Once the previous step have been performed, you should end up with the directory tree shown in Figure 1. With the previous commands, you have automatically generated `build/`, `devel/` and `src/CMakeLists.txt`. These directories and files do not have to be modified manually. The interesting folders are actually contained in `src/f110_simulator/`.

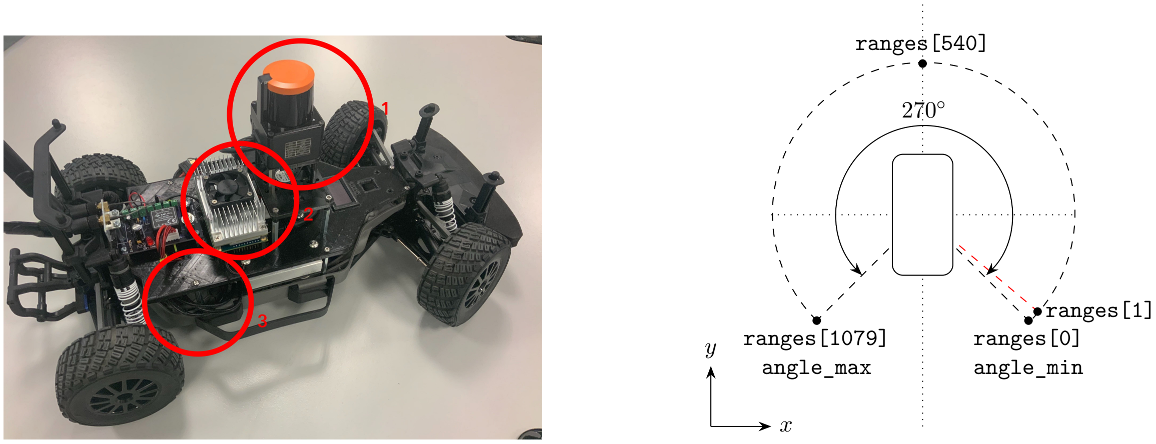
```
simulator/
├── build/
├── devel/
└── src/
    ├── CMakeLists.txt
    └── f110_simulator/
        ├── CMakeLists.txt
        ├── launch/
        ├── src/
        ├── include/
        ├── maps/
        └── node/
```

Figure 1: Structure of the F110 ROS simulator

In fact, the `launch/` directory actually contains `.launch` files that are in charge of specifying what nodes to start for a given command. The provided file called `simulator.launch` contains all the configuration required to start the simulator in itself and provides guidelines on how to include any new custom node. The file can then be used to launch at once all the specified nodes with the following command:

```
roslaunch f110_simulator simulator.launch
```

In the latter command, `roslaunch` start the nodes specified in `simulator.launch` which is located in the project `f110_simulator`.
By convention, the nodes that compose a project are located in the folder `node/`. For instance, you will find by default the source code of the node `keyboard.cpp`.
The content of the `node` directory is being taken into account during the build with the command `catkin_make` thanks to the following lines in `src/CMakeLists.txt`:



(a) Different components of the F110 car

(b) Representation of the Lidar's measurements

Figure 2

```
...
# Add the nodes
file(GLOB NODE_SRC_FILES node/*.cpp)
foreach(_node_file ${NODE_SRC_FILES})
  get_filename_component(_node_name ${_node_file} NAME_WE)
  add_executable(${_node_name} ${_node_file})
  target_link_libraries(${_node_name} ${LIBS})
  add_dependencies(${_node_name}
      ↪ f110_simulator_generate_messages_cpp)
endforeach()
...
```

5.2 Environment

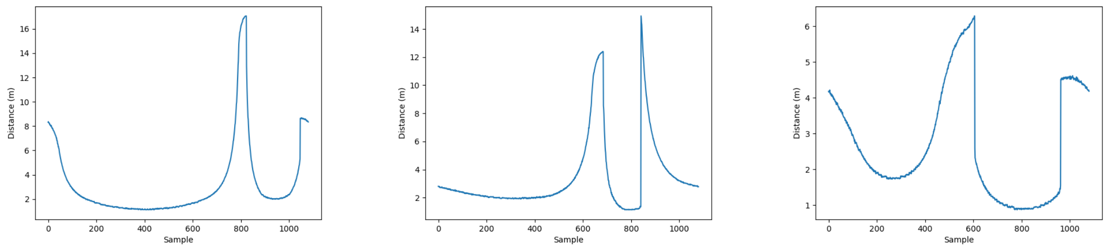
For these lessons, the environment used is simple and only composed of one actuator and one sensor. Indeed, the car receive data from a Lidar and react according to the latter's outcome by acting on the speed and/or the steering angle of the car.
The car's Lidar, as shown in Figure 2a, is a device capable to measure the relative distance between itself and its surrounding by sending light beams and measuring the time required to bounce and come back. For instance, as illustrated in 2b, the Lidar scans 75% its surrounding in one dimension, meaning that the sensor produces an array of distances where each element is a specific distance between the car and the nearest object present at a given angle of the car. See subsection 5.3.3 for further details about the message structure.

5.3 Messages

The nature of each sensor and actuator varying, logically, ROS defines different type of messages. Int he present subsection we present the ones required for the completion of the project, however, note that ROS defines a lot more messages. You can find an exhaustive list of the standard messages at `http://docs.ros.org/melodic/api/std_msgs/html/index-msg.html`.

5.3.1 Standard messages

The only standard messages that must be described here are the `header` and `String` messages. Basically, the former simply provide timing information about the message, whereas the latter



(a) Location 1 (b) Location 2 (c) Location 3

Figure 3: Examples of the Lidar's measurements at three different locations

simply encapsulate a classic `C` String structure. A quick description of the messages can be found here-under with the associated link for further information. `http://docs.ros.org/jade/api/std_msgs/html/msg/Header.html`

```
Header {
    uint32 seq
    time stamp
    string frame_id
}
```

`http://docs.ros.org/melodic/api/std_msgs/html/msg/String.html`

```
String {
      string data
}
```

5.3.2 Controlling the motor

The messages to control the motor (i.e. the ones to be published on the `\drive` topic) follow the `AckermannDrive` message structure described here-under.

```
AckermannDrive {
    float32 steering_angle
    float32 steering_angle_velocity
    float32 speed
    float32 acceleration
    float32 jerk
}
```

In our case, we are only interested in two of the five fields; namely `steering_angle` and `speed`. More accurately:

- `steering_angle` is expressed in **radian**
- `steering_angle_velocity` denotes the speed at which the steering angle change. A value of zero means that transitions are instantaneous.
- `speed` is expressed in $\frac{m}{s}$
- `acceleration` is expressed $\frac{m}{s^2}$. A value of zero means a instantaneous acceleration.
- `jerk` is expressed in $\frac{m}{s^3}$. A value of zero means a instantaneous jerk.

For ease of use, we will always assume a value of zero for `steering_angle_acceleration`, `acceleration` and `jerk`. For further details, the reader is invited to visit the following link:
`http://docs.ros.org/jade/api/ackermann_msgs/html/msg/AckermannDrive.html`
Finally, a more complete alternative deemed as "Stamped" is provided and used by the `\drive` topic. This variation simply consists in the `AckermannDrive` message and a `Header` as described in subsection 5.3.1. For further details, the reader is invited to visit the following link:
`http://docs.ros.org/jade/api/ackermann_msgs/html/msg/AckermannDriveStamped.html`

```
AckermannDriveStamped {
    Header          header
    AckermannDrive  drive
}
```

5.3.3 Sensing distances

As described in subsection 5.2, distances between the car and its environment are measured by a Lidar. The measurements of the latter are posted on the topic `\scan` and have the following structure:

```
LaserScan {
    Header header
    float32 angle_min
    float32 angle_max
    float32 angle_increment
    float32 time_increment
    float32 scan_time
    float32 range_min
    float32 range_max
    float32[] ranges
    float32[] intensities
}
```

In our case, we are particularly interested in a subset of the fields:

- `angle_min` expressed in **radian**, which denotes the angle where the measurements start
- `angle_max` expressed in **radian**, which denotes the angle where the measurements end
- `angle_increment` expressed in **radian**, it denotes the angular distance between two contiguous measurements
- `ranges` is an array containing all the measurements of the Lidar. Each measurement is expressed in meter

Remark:

For further information, the reader is invited to visit the following link:
`http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html`.
More generally, ROS defines a large variety of sensor message types that can be found here:
`http://docs.ros.org/melodic/api/sensor_msgs/html/index-msg.html`

5.4 Managing the F110 Simulator project

5.4.1 Bootstraping the project

All the students will receive the lab package (named `package.zip`) containing the F110 ROS simulator, the VNC clients and the lab handouts by the teaching assistant via Zoom. If the students decide to use the remote lab computers, the package will already be available in the home directory of the account they have been assigned to.

As mentioned before, the provided package is a `.zip` file. Thus, you must first extract its content using the following command line:

```
unzip package.zip
```

Doing so, will provide you with two folders: `simulator/` contains the F110 ROS simulator. The folder is further described in Fig. 1 and `labs/` contains all the lab handouts.

Once the package content is extracted, you need to build the proejct for the very first time. Move to the `simulator/` folder. For the time being, the folder only contains `src/` (type `ls`). In order to end up with the directory tree shown in Fig. 1, you should first set the ROS environment variables by typing:

```
source /opt/ros/melodic/setup.sh
```

Thereafter, you can build the project by typing:

```
catkin_make
```

Upon the completion of the command, the current folder (i.e. `simulator/`) should be populated with `build/` and `devel/` as shown in Fig. 1.

> Remark:
>
> You will get and error saying that catkin is not installed, it means that you have not source ROS!

The project being built, we can finally start. However, you must first set the ROS project environment variable. Similarly as what we have done for ROS, the command is:

```
source devel/setup.sh
```

Finally, you can start the simulator as follows:

```
roslaunch f110_simulator simulator.launch
```

5.4.2 Development cycle

Each time you close a terminal, both the ROS and the simulator environment variables are discarded, This implies that each time you open a new termianl, you must `source` ROS and the project. In order to avoid such a hassle, you are invited to append

```
source /opt/ros/melodic/setup.sh
```

to the end of your `.bashrc` file. The latter being located in your home directory (you can move there with `cd ~`). **Make sure to only append the line and not to erase anything else!**

Throughput the development of your project, you will use the aforementioned command in the following order:

```
catkin_make
source devel/setup.sh
roslaunch f110_simulator simulator.launch
```

Remark:

Catkin is stubborn and big changes such as file renaming or the introduction of a new set of files might not be taken into account. In order to fix this issue, we invite you to either use

```
catkin_make clean
```

Or more radically to delete the `build/` folder

```
rm -r build/
```

make sure not to delete the other folders!