A. *Fixed path*

In the state-of-the-art, considerable research has been done in planning a particular route and navigating on that route. Henson et al. [1] propose an algorithm which processes waypoint instructions by calculating a route between each point. This algorithm is optimised for cars with an Ackermann-steering mechanism. Thus, if the waypoint is reached but without the requested orientation, the car starts to manoeuvre until the positioning is correct.

Theodosis [12] suggests multiple methods, all of which use waypoints. These checkpoints are connected with racing lines such that a car-like robot can easily navigate without impossible pivot points.

In the ROS-framework, a complete navigator package is available, which is also known as the *Navigation Stack*. This module calculates a path and velocity commands based on given information, such as odometry and data from rangefinders. A major drawback of this navigator is the fact that it is created for robots with holonomic and differential steering mechanisms. Though, it is usable for car-like robots as there exist implementations for Ackermann-steering mechanisms.

For this research, the navigation stack has been implemented. It is easy to set up on the F1/10-car as the software on the car is already running a ROS-environment. The module is realised using multiple nodes and packages. One of these nodes is the *move_base*-node, which has a major role in the system. This node provides the linkage between the global and local planning algorithms that are used to calculate a path from the start location to the destination. Other nodes outside the package provide the inputs and outputs of the *move_base*-node. The needed inputs are the odometry data, transformations, localisation information, the map on which the navigation would take place and the data from a certain rangefinder. In our setup, a LiDAR-sensor (Light Detection and Ranging) serves as rangefinder. The commands by which the car would drive, are the generated outputs of the central navigation node. Figure 1 shows an overall overview of the different nodes in the navigation stack [7].

Inside the *move_base*-node, multiple calculations take place in order to navigate. First, a plugin called *global_costmap* produces a costmap, which is used to plan the route from start to destination. This is done by the plugin *global_planner*. The default implementation of this planner is *navfn*. In our implementation, we opted for the more advanced planner *global_planner* because of the ability to customise multiple parameters. Second, the central node produces a local costmap which is used by the local planner. By using this particular costmap and planner, the navigation stack is able to plan a route around an unexpected obstacle that is not present on the map. The default local planner is *base_local_planner*. Though, an other implementation is available for car-like robots. This planner is known as *teb_local_planner* [10]. As a safety feature, the *move_base*-node includes a recovery plugin which plans a secondary route when the car is stuck and thus unable to follow the planned route. Figure 1 shows an abstract overview of the navigation stack with its nodes and plugins [8], [14].

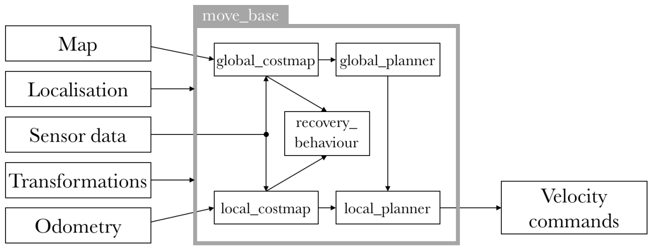

Fig. 1. Overview of the navigation stack with its input and output nodes, along with the plugins inside the *move_base*-node [7].

Before the car is able to drive using the navigation stack, one has to make sure a map of the environment is present. The map used in this research is built by using *hector_mapping*. This package is an implementation of the SLAM-methodology (Simultaneous Localisation And Mapping), which only uses data from the LiDAR-sensor. Therefore, this is the most suitable implementation for our F1/10-car due to the lack of odometry data. An elaborated discussion on *hector_mapping* can be found in [2] and [5].

In order to use the *move_base*-node properly, certain precautions have to be made. First, the transformation tree has to be correct such that the laser data can be transformed to the center of the robot. To build such a tree, two *tf*-nodes are created. The first node transforms the LiDAR-data to a link that connects all types of sensors. The second node transforms the linking point to the center of the frame. Since only one sensor is present, we locate the linking point to the center of the frame; thus, the second transformation is one-to-one. Though, this second transform is needed as the navigation stack needs the *base_frame*-transformation [6] [13].

Second, the navigation stack needs a source of odometry data. Since the car does not feature a suitable source, we have to generate such data from the LiDAR-sensor. *Hector_mapping* has the ability to provide odometry data which is derived from the laser scan. In order to use this data in the navigation stack, a particular node is needed which translates the odometry information to transformation data. The implementation of this node is available on the *f1tenth*-repository of Nischal K.N. [4].

Third, the robot must be able to guess its location on the given map. This is implemented by the *AMCL*-node (*Adaptive Monte Carlo Localisation*). In order to generate a suitable pose guess, the node also needs odometry data, along with the map, transformations and LiDAR-information [3] [11].
Final, the translation from the velocity commands of the navigator to instructions for the motor controller needs to be made. Figure 2 shows the transformations created by the aforementioned nodes.

As a result, the planned route is shown in figure 3, along with a close-up of the car navigating on that path.

A major drawback of the navigation stack is its lack

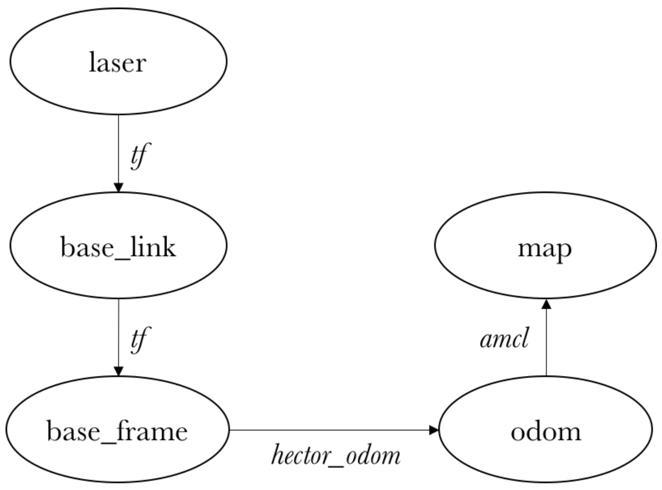

Fig. 2. Created transformation frames for the navigation stack. The *tf*-links between *laser*, *base_link* and *base_frame* are static transforms. Next, the *hector_odom*-link is created by a node which translates the odometry of *hector_mapping*. The *amcl*-transform is created by the localisation provider.

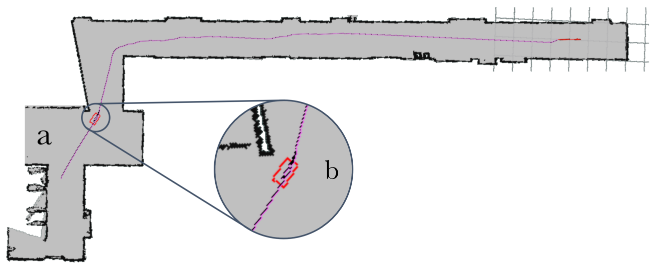

Fig. 3. a) shows the route which is planned by the navigation stack. The map is built with *hector_mapping* [5]. b) illustrates a close-up of the car moving on the route.

of optimisation for car-like robots. Instead, it is especially created for robots with holonomic or differential steering mechanisms. Therefore, the car is sometimes unable to follow the calculated path. Because of this problem, we implemented *teb_local_planner* such that the route would be more optimised for Ackermann-steering. It appeared that this planner needed a a lot of computing power. The onboard NVIDIA Jetson TK1 was unable to finish the calculations before the requested deadlines. Even a virtual machine with high specifications could not finish the process. Therefore, the default local planner is used for further usage. Although using this planner could result in pivot points and sharp curves, the car is perfectly able to navigate on the trajectory if the curves are almost straight and without turning points. Another drawback is the low driving speed at which the car has to drive in order to update its position. By increasing speed, a certain latency exists between the actual position and the calculated one. As a result, the navigation stack is too slow with processing a new local path and crashes could occur.