The Development and Comparison of Algorithms towards an Autonomous F1/10-Car

Jens de Hoog*, Thomas Huybrechts*§, Ben Bellekens*§, Peter Hellinckx*§
jens.dehoog@student.uantwerpen.be, thomas.huybrechts@uantwerpen.be,
ben.bellekens@uantwerpen.be, peter.hellinckx@uantwerpen.be

*Faculty of Applied Engineering, Electronics-ICT, University of Antwerp, Belgium
§University of Antwerp - imec, IDLab, Faculty of Applied Engineering, Belgium

***Abstract*—Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.**

I. Introduction

The original intent of the F1/10-car was to develop an autonomous race car which would compete against other cars, each with other algorithms. The University of Pennsylvania [9] published multiple tutorials, lessons and exercises in order to set up an F1/10-car from the very beginning. Additionally, the university proposes several algorithms which can be used to drive on a race track. Though, these algorithms are optimised for racing on fixed tracks at high speeds.

II. Algorithms

A. Global outline

In order to navigate in a particular environment, multiple methods of navigation exist. In this paper, three approaches are discussed. A small literature study is made for each approach, along with an explanation of the algorithm itself and a discussion of the obtained results.

In the first method, the car drives on a fixed trajectory which was calculated in advance. The second approach deals with waypoints. Thus, the car has to drive along these points in order to reach the destination. It is worth noting that no path is calculated between the waypoints. Otherwise, the first method is approached. Only a destination is given in the third and last process such that the car has to find the destination itself.

B. 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

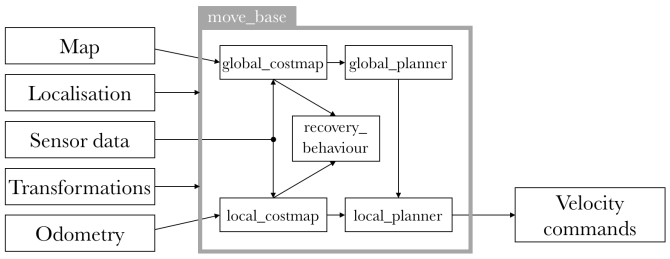

Fig. 1. Overview of the navigation stack with its input and output nodes, along with the plugins inside the *move_base*-node [7].

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].

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

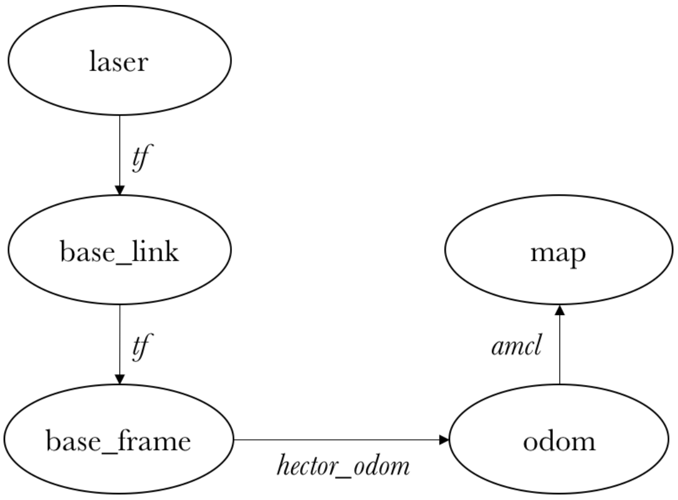

Fig. 2. Created transformation frames for the navigation stack. The *tf*-links between *laser*, *base_link* and *base_frame* are static transforms. Next, the *hector_odom*-link is created by a node which translates the odometry of *hector_mapping*. The *amcl*-transform is created by the localisation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

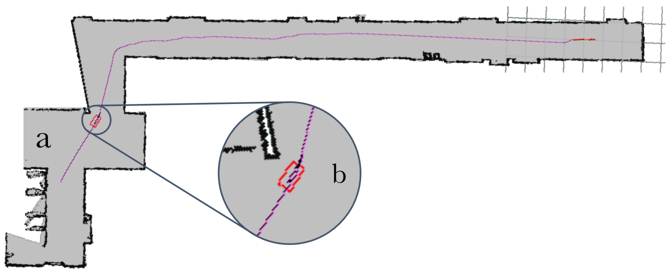

Fig. 3. a) shows the route which is planned by the navigation stack. The map is built with *hector_mapping* [5]. b) illustrates a close-up of the car moving on the route.

guess, the node also needs odometry data, along with the map, transformations and LiDAR-information [3] [11]. The pose guess is relative to the given map; thus, this node provides the transformation between the odometry and the map.
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 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

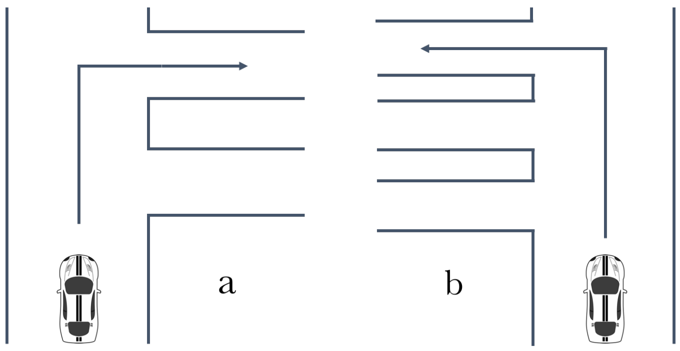

Fig. 4. a) illustrates a waypoint which is described as (*right*, 2). b) shows the instruction (*left*, 3).

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.

C. *Waypoints*

In the current state-of-the-art, many navigation algorithms process waypoints in order to reach the destination. E.g., the aforementioned algorithms [1] [12] use waypoints to describe the route, but the paths between those points are still calculated; thus these algorithms are suitable for the fixed path approach mentioned in the previous section. Due to the lack of research for this problem, no acceptable solutions are available yet. Therefore, this paper suggests a new methodology to tackle this problem.

In the algorithm, a waypoint is similar to an instruction of *turn-by-turn*-based navigation. A waypoint consists of two datafields: *direction* and *count*. The *direction*-field has three possible options: *straigt*, *left* and *right*. The *count*-variable describes which turn should be taken. Figure 4 shows two examples. In the first case, the waypoint is described as (*right*, 2). I.e., the car should take the second exit on the right. The instruction of the second example is described as (*left*, 3).

The actual implementation of the algorithm is done by B. Smits and S. Maes for their bachelor's thesis, but first, a few prerequisites have to be made. We have to make sure the car drives in the middle of the road in order to drive in a hallway. Smits and Maes implemented multiple ROS-nodes which adjust the wheel position of the car by given LiDAR-data, thus regulating the distance with respect to the wall while driving.

With this ROS-nodes, the implementation of the algorithm takes place. When the car is driving and it detects a corner on one of its sides, the software checks if this is the corner that is specified by the waypoint instruction. If false, the car continues driving while focusing on the opposite wall, such that the car stays in the middle of the road. If the result is true, the car will take the turn by adjusting the wheel position until the car is parallel with the wall of the corner. If the software detects corners on both of its sides, but neither of them is specified by the waypoint, the steering position is set to neutral. Therefore, the car drives straight until a wall on either side is detected. Thus, the aforementioned ROS-nodes can be resumed.

a) *Concept:*

D. *Optimisation Algorithm*

Literature

Ik leg verschillende algoritmes naast elkaar en maak een kleine vergelijkende studie. Ik leg ook uit waarom ik een bepaald algoritme heb gekozen.

Concept

Het gekozen algoritme wordt in meer in detail besproken.

Resultaten

De resultaten van dit algoritme wordt besproken.

III. Comparative study

De hierboven voorgelegde approaches worden met elkaar vergeleken en er worden verschillende toepassingen opgesomd waarin elk algoritme faalt of uitblinkt.

IV. Test setup

A. *F1/10-car*

Ik vertel over de wagen in het algemeen en welke sensoren eropzitten. Dit gebeurt heel kort. Ook vertel ik over de globale structuur van ROS en welke belangrijke nodes er gebruikt worden.

B. *Prerequisites*

In dit deel vertel ik over het traag rijden en het rijden tegen constante snelheid.

- Slow accurate driving: er wordt een beetje aandacht besteedt aan de twee approaches en waarom ze wel/niet werkten
- Ik vertel kort waarom het rijden op softwareniveau (met IMU) niet werkten en hoe het met de RPM-sensor is opgelost.

V. Conclusion

De conclusie wordt gemaakt.

VI. Further Research

In dit deel wordt kort besproken wat er allemaal nog gedaan kan worden van research. Er kan bijvoorbeeld gekeken naar worden optimalisaties voor de Navigation Stack en meer specifiek voor Ackermannsteering. Ook kan er gekeken worden naar betere herkenning van hoeken en inhammen voor het tweede algoritme. Als laatste zouden er betere optimalisatie-algoritmes onderzocht kunnen worden, en met of zonder neural network.

References

[1] G. Henson, M. Maynard, G. Dimitoglou et al. Algorithms and performance analysis for path navigation of ackerman-steered autonomous robots. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, PerMIS '08, pages 230–235, New York, NY, USA, 2008. ACM.

[2] S. Kohlbrecher, O. Von Stryk, J. Meyer et al. A flexible and scalable SLAM system with full 3D motion estimation. *9th IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2011*, pages 155–160, 2011.

[3] B. P. Gerkey. amcl - ROS Wiki. http://wiki.ros.org/amcl, 2016. [Online; last accessed May 2017].

[4] Nischal K.N. Repository: 1/10th Scale Autonomous Race Car. https://github.com/nischalkn/F1tenth, 2016. [Online; last accessed May 2017].

[5] S. Kohlbrecher. hector_mapping - ROS Wiki. http://wiki.ros.org/hector_mapping, 2012.

[6] S. Kohlbrecher. How to set up hector_slam for your robot - ROS Wiki. http://wiki.ros.org/hector{_}slam/Tutorials/SettingUpForYourRobot, 2012. [Online; last accessed April 2017].

[7] E. Marder-Eppstein. move_base - ROS Wiki. http://wiki.ros.org/move_base, 2016. [Online; last accessed May 2017].

[8] E. Marder-Eppstein. nav_core - ROS Wiki. http://wiki.ros.org/nav_core, 2016. [Online; last accessed May 2017].

[9] University of Pennsylvania. The Official Home of F1/10. http://f1tenth.org/, 2016. [Online; last accessed May 2017].

[10] Christoph Rösmann. teb_local_planner - ROS Wiki. http://wiki.ros.org/teb{_}local{_}planner, 2016. [Online; last accessed May 2017].

[11] S. Thrun, W. Burgard, D. Fox. *Probabilistic Robotics*. 1999.

[12] Paul Alan Theodosis. *Path Planning for an Automated Vehicle using Professional Racing Techniques*. Degree of doctor of philosophy, Stanford University, 2014.

[13] W. Woodall. Setting up your robot using tf - ROS Wiki. http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF, 2015. [Online; last accessed April 2017].

[14] K. Zheng. ROS Navigation Tuning Guide. Technical report, 2016.