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

Jens de Hoog*, Thomas Huybrechts*§, Ben Bellekens*§, Peter Hellinckx*§

*Department of Applied Engineering, Electronics-ICT, University of Antwerp, Belgium
Email: jens.dehoog@student.uantwerpen.be
§University of Antwerp - imec, IDLab, Department of Applied Engineering, Belgium
Email: {thomas.huybrechts, ben.bellekens, peter.hellinckx}@uantwerpen.be

***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

II. Related work

A. Fixed path

Before navigation on a fixed path takes place, a the provision of a map and localisation on that map is needed. In the state-of-the-art, considerable research has been done in planning a particular route and navigating on that route. However, a map and localisation information have to be provided in the first place. To tackle this problem, multiple approaches have been introduced. One of these approaches is the *SLAM*-methodology (Simultaneous Localisation And Mapping). *Hector_SLAM* is an implementation and provides information based on laser data from a LiDAR-sensor (Light Detection And Ranging) [7], [12]. Another approach is *gmapping*, which solves the SLAM-problem by laser data and odometry [8]. Since the F1/10-car does not provide odometry data, this approach is not suitable for our research.

Henson et al. [4] 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. The Ackermann-principle makes sure that, when turning the wheels of a car, the outer wheel turns less sharply than the inner wheel. This results in both wheels turning around the same pivot point, thus minimising the risk of slipping sideways. [10].

Theodosis suggests multiple methods, all of which use waypoints [22]. These checkpoints are connected with racing lines such that a car-like robot can easily navigate without impossible pivot points. Racing lines are paths which provide the highest average speed when driving by combining maximal acceleration and grip on the road while minimising the travelled distance. [15].

In the ROS-framework, a complete navigator package is available, which is also known as the *Navigation Stack* [16] [19]. 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. Holonomic robots are able to move in certain degrees of freedom (*DoF*), while having an equal or larger number of controllable DoF [6]. In this way, these robots are able to move in any direction without repositioning the wheels. Differential steering mechanisms mostly consists of two wheels which have the same speed when moving forward, but different speeds when changing direction [14]. Though, it is usable for car-like robots as there exist implementations for Ackermann-steering mechanisms.

B. Waypoints

In the current state-of-the-art, many navigation algorithms process waypoints in order to reach the destination. E.g., the aforementioned algorithms [4] [22] 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.

Other studies process waypoints in different methods. For example, Erp et al. developed a vibrotactile waist belt which consists of a GPS-sensor and magnetometer [5]. The distance and direction of the next waypoint is provided to users by tactile instructions. May and Ross [1] described waypoints as landmarks, such as traffic lights, churches and other marks that are visible from a considerable distance. The user acquires *turn-by-turn*-instructions with these landmarks as navigation cues.

The aforementioned methodologies provide solutions for navigation with pedestrians or real cars. Though, these algorithms are not suitable for the F1/10-car as it lacks a GPS-sensor and an advanced vision system which is able to track landmarks. Also, adding a GPS-module to the car is not

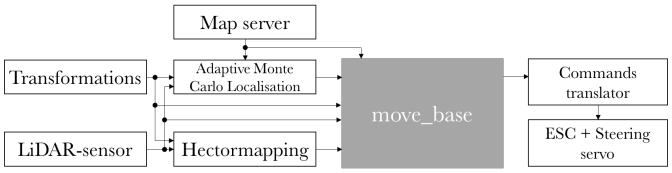

Fig. 1. Illustrates the integration of the *move_base*-node inside the existing environment

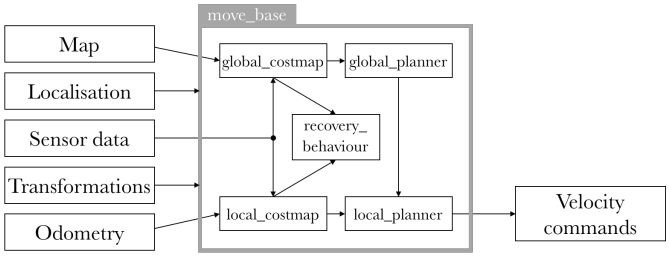

Fig. 2. Overview of the navigation stack with its input and output nodes, along with the plug-ins inside the *move_base*-node [16].

an option because the navigation will take place in indoor environments. Due to these shortcomings, this paper suggests a new methodology to approach the waypoint navigation problem.

C. Optimisation Algorithm

III. Implementations

A. Fixed path

For this research, the navigation stack has been implemented as the software on the car is already running a ROS-environment.

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 a part of the *hector_SLAM*-implementation, which is mentioned in the previous section. As the car lacks a source of odometry data, another implementation such as *gmapping* is not suitable.

a) Overview: The navigation 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. The explanation of these calculations is found in the next paragraph. 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 the integration of the navigation module in the existing environment. Figure 2 illustrates an abstract overview of the navigation stack with its nodes and plug-ins [16].

Inside the *move_base*-node, multiple calculations take place in order to navigate.

b) Global planning: In order to make a global plan, a global costmap must be generated. This particular map is derived from the environment map, which in turn is made by *hector_mapping*. A costmap is an occupancy grid which specifies where the car is allowed to navigate. An other node named *global_planner* calculates a path from start to destination, using the global costmap. The default implementation of this planner is *navfn*, which uses Dijkstra as path finding algorithm, but we opted for the more advanced planner *global_planner* because of the ability to customise multiple parameters. Unfortunately, the name of the implementation is the same as the one of the node, thus making the whole structure a bit unclear.

c) Local planning: Second, the central node produces a local costmap which is used by the local planner. The costmap is made by obtaining LiDAR-information in a range of three meters around the car. In this way, unexpected obstacles are located on this costmap. Therefore, the local planner is able to plan a route around these obstructions. The default local planner is *base_local_planner* [18]. The planner is realised using Trajectory Rollout algorithm and Dynamic Window Approach [2], [3]. Though, an other implementation is available for car-like robots. This planner is known as *teb_local_planner* [20]. It implements a methodology called *timed elastic band* and is more optimised for car-like robots. As a safety feature, the *move_base*-node includes a recovery plug-in which plans a secondary route when the car is stuck and thus unable to follow the planned route. Figure 2 shows an abstract overview of the navigation stack with its nodes and plug-ins [17], [24].

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 transformation 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 [13] [23].

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. By matching sequential scan frames, the occurred motion can be derived, which in turn can be outputted as odometry data. In order to use this odometry information 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. [11].

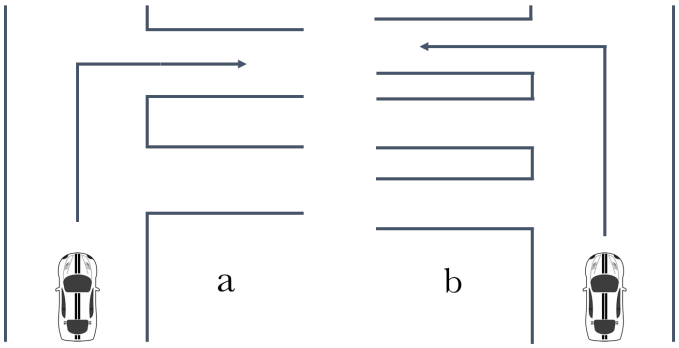

Fig. 3. a) illustrates a waypoint which is described as $(right, 2)$. b) shows the instruction $(left, 3)$.

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 [9] [21]. 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.

B. Waypoints

In this 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 3 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.

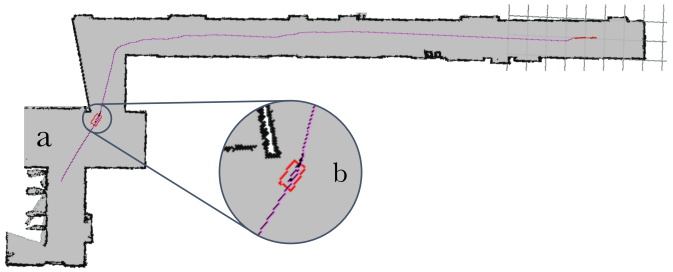

Fig. 4. a) shows the route which is planned by the navigation stack. The map is built with *hector_mapping* [12]. b) illustrates a close-up of the car moving on the route.

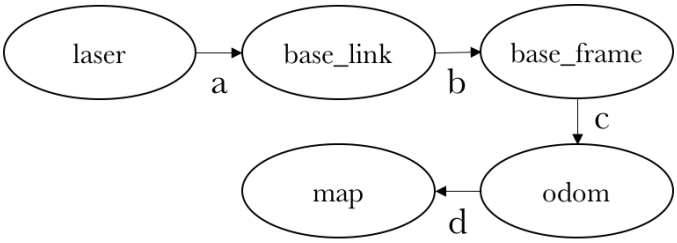

Fig. 5. Created transformation frames for the navigation stack. a) and b) are static transformations. The transformation in c) is provided using the odometry data from *hector_mapping*. d) is provided by the *AMCL*-node, which links the map with the pose of the robot.

C. Optimisation Algorithm

IV. Test Setup

A. F1/10-car

The car used in this research consists of multiple sensors. First, a Hokuyo UST-10LX serves as LiDAR-sensor on the robot. With a refresh rate of 40 Hz and an angle resolution of 0.25°over a range of 270°, this sensor is capable for both fast and accurate measurements.

Second, an IMU-sensor (Inertial Measurement Unit) is present. This sensor consists of a magnetometer, gyroscope and accelerometer, all of which have three axes, thus having nine degrees of freedom.

V. Results

A. Fixed path

Figure 5 shows the transformations created by the aforementioned nodes.

The planned route is shown in figure 4, along with a close-up of the car navigating on that path. Additionally, figure 5 illustrates the transformations which are created by the transformation nodes of the navigation stack

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 at a requested rate of 2 Hz. Instead, each calculation lasted approximately three seconds. Even a virtual machine with high specifications could not finish the process. The virtual machine was hosted by a macOS-host with Parallels as virtualisation software. The specifications were set to eight virtual cores and four gigabytes of RAM. Because of this latency, the default local planner was 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, so crashes could occur.

B. Waypoints

With the first implementation of the algorithm, the system detected many false positives. That is, the car turned left or right while a corner was absent. This phenomenon happened when, for example, the car was in the vicinity of a door.

C. Optimisation Algorithm

VI. Comparative study

VII. Conclusion

VIII. Further Research

References

[1] A. J. May, T. Ross. Presence and Quality of Navigational Landmarks: Effect on Driver Performance and Implications for Design. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 48(2):346–361, 2006.

[2] B. P. Gerkey, K. Konolige. Planning and Control in Unstructured Terrain. *ICRA Workshop on Path Planning on Costmaps*, 2008.

[3] D. Fox, W. Burgard, S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1):23–33, 1997.

[4] 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.

[5] J. B. F. Van Erp, H. A. H. C. Van Veen, Chris Jansen et al. Waypoint navigation with a vibrotactile waist belt. *ACM Transactions on Applied Perception*, 2(2):106–117, 2005.

[6] M. Mariappan, C. Chee Wee, K. Vellian et al. A navigation methodology of an holonomic mobile robot using optical tracking device (OTD). *IEEE Region 10 Annual International Conference, Proceedings/TENCON*, pages 1–6, 2009.

[7] 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.

[8] G. Grisetti, C. Stachniss, W. Burgard. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.

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

[10] Desmond King-Hele. Erasmus Darwin's Improved Design for Steering Carriages–And Cars. *Notes and Records of the Royal Society of London*, 56(1):41–62, 2002.

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

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

[13] 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].

[14] M. S. Saidonr, H. Desa, R. Md Noor. A differential steering control with proportional controller for an autonomous mobile robot. *Proceedings - 2011 IEEE 7th International Colloquium on Signal Processing and Its Applications, CSPA 2011*, (March):90–94, 2011.

[15] Neil Macfarland, Noah Weinthal, E Carr Everbach, and Matt Zucker. A Testbed for Autonomous Racecar Control Strategies. 2015.

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

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

[18] Eitan Marder-Eppstein. base_local_planner - ROS Wiki. http://wiki.ros.org/base{_}local{_}planner, 2017. [Online; last accessed May 2017].

[19] Eitan Marder-Eppstein. navigation - ROS Wiki. http://wiki.ros.org/navigation, 2017. [Online; last accessed May 2017].

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

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

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

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

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