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*—The original purpose of the F1/10-car, an F1-car with scaling factor 1/10, was to organise race competitions. However, this car is an ideal test bed for real autonomous cars. Therefore, this paper implements and compares three different navigation algorithms. The first algorithm involves the Navigation Stack from the ROS-framework (Robot Operating System), which allows the car to drive on a fixed path. The second algorithm processes waypoints in order to reach the destination and the last method implements an optimisation algorithm. Due to the limited time frame, the implementation of the last algorithm has not finished yet. Thus, the paper only compares two methodologies. The first one has a low maximum speed but an accurate goal description, whereas the second algorithm maintains a high speed, but the goal is described in a more abstract way, making the destination inaccurate.**

I. Introduction

These days, robotics are deployed in many applications in different domains and their popularity is still growing. Especially autonomous robots are a captivating topic due to the versatility of these machines. For example, interplanetary missions, which are almost impossible for humans, are ideal for autonomous robots. Self-driving cars are also an example. This is an enthralling topic in which the interest is also increasing. Major companies, such as Uber, Waymo and Tesla, Inc., are pushing the limits of autonomous vehicles further. The technology has even evolved so far that the car of Waymo has approximately one disengagement (an action where the driver has to take over control) in every 5,000 drived miles [4]. Additionally, real autonomous vehicles will be very practical in smart city environments that deploy Mobility as a Service (MaaS). When using such a service, the user pays for the ride he is taking instead of paying for the car he or she had bought. The combination of MaaS and self-driving cars provides many benefits such as reducing traffic congestions and overall transportation costs [30].

In 2016, the university of Pennsylvania started a project known as *F1/10-car* [26]. This car is a scaled model of an F1-car with scaling factor 1/10. The purpose of this project is to start a competition of F1/10-cars, which involves designing, building and testing of the car. Whereas many algorithms have been developed in order to race with this car, little research has been conducted in navigation. This topic is interesting as the car can serve as test bed for real autonomous cars and MaaS-systems.

Due to this lack of research, this paper implements and compares three different types of navigation algorithms. First, a fixed path is considered. I.e., the software calculates a path onto which the car will drive in order to reach the destination. The second approach are waypoints that describe the route to be driven. By navigating from point to point, the destination is reached. The last method involves an optimisation algorithm so that an optimal route can be derived from previous routes.

We have organised the rest of the paper in the following way. The remainder of this paper will further discuss the related work on these algorithms and how we incoperated this knowledge in our autonomous car. In section III, an elaboration for each algorithm is introduced. Section IV reveals the test setup that was used in order to obtain the results. Section V presents the obtained results for each method. In section VI a comparison of the three methodologies is presented. Finally, section VII suggests topics for further research and VIII forms the conclusion of this paper.

II. Related work

A. Fixed Path

In the state-of-the-art, considerable research has been conducted in planning and navigating on a particular 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). This method provides a solution for building a map and localising on that map at the same time, which is a challenge in robotics. *Hector_SLAM* is an implementation and provides information based on laser data from a LiDAR-sensor (Light Detection And Ranging) [11], [17]. Another approach is *gmapping*, which solves the SLAM-challenge by combining laser data and odometry [13]. Since the F1/10-car does not provide odometry data, this approach is not suitable for our research.

Henson et al. [6] 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 with-

out the requested orientation, the car starts to manoeuvre until its position is correct. The Ackermann-principle guarantees that the outer wheel turns less sharply than the inner wheel when turning the wheels of a car. This results in both wheels turning around the same pivot point, thus minimising the risk of slipping sideways. [15].

Theodosis suggests multiple methods, all of which use waypoints [31]. 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. [21].

In the ROS-framework (Robotic Operation System), a complete navigator package is available, which is also known as the *Navigation Stack* [22] [25]. 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 [9]. 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 [20]. 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 [6] [31] 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 [7]. 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 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

Although optimisation problems are challenging for both humans and computers, they are able to provide very useful data. For example, finding a path in a complex environment is one of the capabilities of an optimisation algorithm. These path planning problems are also solvable by algorithms such as Dijkstra or A*. However, they are very time consuming in comparison with optimisation systems [12].

Several path finding algorithms have been developed over the past few years. First, Garcia et al. propose an algorithm based on *ant colony optimisation* [8]. This type of optimisation relies on the behaviour of real ants, which spread a pheromone when walking on a path in search of food [19]. As the pheromones evaporate after a period of time, a shorter path contains more chemical substances. As a result, the path with the most pheromones is the shortest path to the destination. Garcia et al. considers such an ant as a robot, searching for its destination. By assigning costs to each path, which are based on effort and distance, and iterating over the algorithm, the shortest path is found.

Saska et al. suggest a path planning algorithm for robotic football, using *particle swarm optimisation* (*PSO*) and Ferguson Splines [10]. In PSO, the particles in this algorithm are suggested solutions. Each particle knows its local best position, which affects the movement of that particle. The best position in the search space also affects this movement. This combination results in particles moving to a position which represents the global minimum, thus the most optimised point. A path containing these Ferguson Splines is considered as a smooth path, which is feasible for a car-like robot. A major drawback of this algorithm is the lack of dynamic updates. When a change in the environment occurs, the algorithm has to restart, which results in a high calculation time. Nasrollahy and Javadi suggested a methodology for using PSO in a dynamic environment [2].

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 provide a map of the environment. 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 discussed in the next paragraph. Other nodes outside the package provide the inputs and outputs

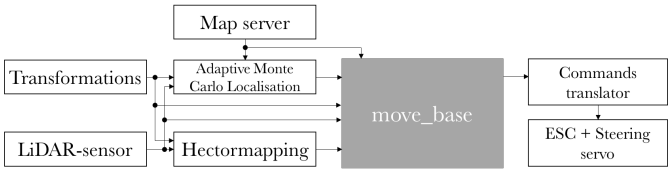

Fig. 1. Illustrates the integration of the *move_base*-node inside the existing environment

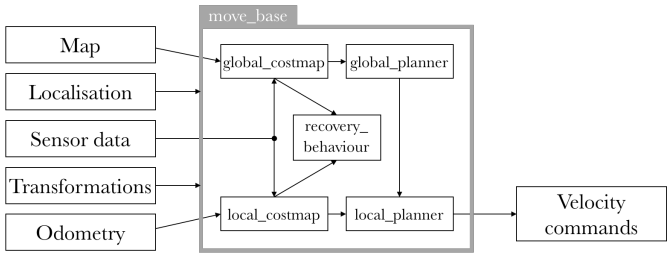

Fig. 2. Overview of the navigation stack with its input and output nodes, along with the plug-ins inside the *move_base*-node [22].

of the *move_base*-node. The required inputs are the odometry data, transformations, localisation information, the map on which the navigation takes place and the data from a certain rangefinder. In our setup, a LiDAR-sensor serves as rangefinder. The commands by which the car will 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 [22].

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. Another 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* [24]. The planner is realised using Trajectory Rollout algorithm and Dynamic Window Approach [3], [5]. Though, another implementation is available for car-like robots. This planner is known as *teb_local_planner* and it implements a methodology called *timed elastic band* [27]. 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 [23], [33].

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 [18] [32].

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. [16].

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 [14] [28]. 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

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 $(\mathit{right}, 2)$. I.e., the car should take the second exit on the right. The instruction of the second example is described as $(\mathit{left}, 3)$.

The following paragraph clarifies the algorithm. When the car is driving and detects a corner on one of its sides, the software checks if this is the corner that is specified by the waypoint instruction. If not, the car continues driving while setting the steering position to a neutral state, 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. When corners on both sides are detected, but neither of them is specified by the

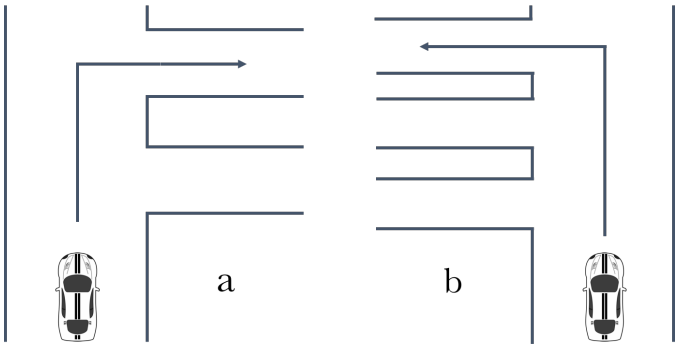

Fig. 3. a) Illustrates a waypoint which is described as (*right*, 2). b) Shows the instruction (*left*, 3).

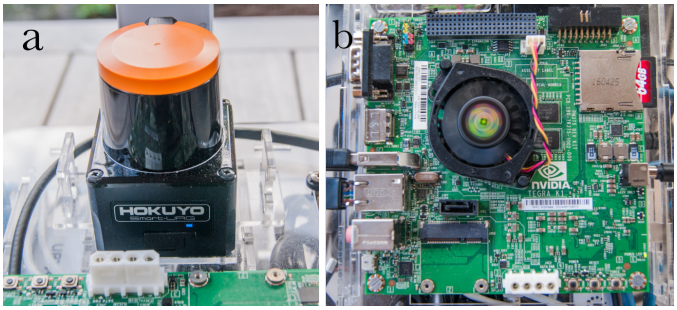

Fig. 4. a) Shows the Hokuyo UST-10LX as LiDAR-sensor. In b), the development board NVIDIA Jetson TK1 is shown.

waypoint, the steering position is also set to neutral in order to keep driving straight until a wall on either side is detected.

The actual implementation of the algorithm is done by B. Smits and S. Maes for their bachelor's thesis [29]. An elaborated discussion is found in their paper. It is also worth noting that we have to assume the car drives in the middle of the road. Smits and Maes implemented multiple ROS-nodes which adjust the wheel position of the car with given LiDAR-data, thus regulating the distance with respect to the wall while driving.

C. Optimisation Algorithm

Due to the limited time frame of this thesis, the implementation of an optimisation algorithm has not finished yet. Though, as mentioned in the previous section, the Ferguson Splines used in [10] are suitable for the F1/10-car.

IV. Test Setup

A. F1/10-car

The car used in this research is driven by an NVIDIA Jetson TK1 development board. This board contains a System on a Chip (Soc), which is known as Tegra K1 from the same manufacturer. The SoC consists of a quad core CPU, which is made of four ARM Cortex-A15-cores. The CPU is assisted by 192 CUDA-cores as graphical unit.

The used operating system is an Ubuntu distribution, i.e. version 14.04. The ROS-ecosystem (Robot Operating System) serves as middleware application. This open-source framework provides different types of libraries and packages in order to develop custom robot applications. Different ROS-nodes communicate with each other via the Publish-Subscribe-methodology.

Additionally, multiple sensors are present. 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 of both fast and accurate measurements. Second, an IMU-sensor (Inertial Measurement Unit) is available. This sensor consists of a magnetometer, gyroscope and accelerometer, all of which have three axes, thus having nine degrees of freedom in total. The output of both of these sensors is handled by different ROS-nodes, which in turn publish the information to other nodes.

The body of the car itself is made by Traxxas. The Velineon VXL-3s serves as motor controller or ESC and is connected to a Teensy 3.2 development board, containing an ARM Cortex-M4 processor. The steering servo is also connected to this development board. The microcontroller generates two PWM-signals (Pulse Width Modulation) that are sent to the ESC and servo in order to control the behaviour of the car. This microcontroller also communicates with other nodes via ROS.

Figure 4 shows the used LiDAR-sensor from Hokuyo and the NVIDIA Jetson TK1 development board.

B. Prerequisites

Before the car was able to navigate in indoor environments, certain prerequisites had to be made. First, the car was unable to drive at a low speed. This low speed was needed to avoid heavy crashes when debugging the software. The general approach is to modulate the ESC-signal with a square wave, such that the motor controller is turned on and off, thus regulating the speed. The first solution was to modulate the signal with another PWM-signal. Thus, the duty cycle could be adjusted in order to regulate the motor power. Several problems arose when using this method. The speed controller does not accept high frequency signals with a low duty cycle. Additionally, the M4-processor could not generate block signals with a low frequency and high duty cycles.

Due to this problems, we were urged to develop a new solution. In this approach, the frequency varies while the duty cycle is fixed. For example, the motor has less time to accelerate when increasing frequency of the square wave. Obviously, the acceleration time increases as the frequency decreases, resulting in a smoother driving experience.

Another prerequisite we had to make, was the ability to drive at a constant speed. First, we opted for speed regulation via the IMU-sensor. Though, it appeared that this method was unable to derive the speed from the output data. We tried many methods such as a Kalman-filter, complementary filter or derivation of speed from raw accelerometer data, but all of the obtained resuls were not as expected. Therefore, an RPM-sensor from Traxxas is integrated in the body and is connected to the Teensy development board. The sensor is based on the

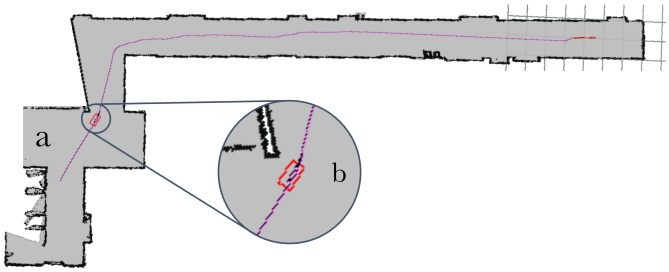

Fig. 5. a) shows the route which is planned by the navigation stack. The map is built with *hector_mapping* [17]. b) illustrates a close-up of the car moving on the route.

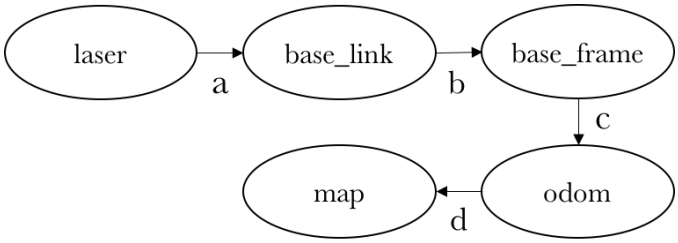

Fig. 6. Created transformation frames for the navigation stack. a) and b) are static transformations. The transformation in c) is provided using the odometry data from *hector_mapping*. d) is provided by the *AMCL*-node, which links the map with the pose of the robot.

Reed-principle. Every 100 ms, the Teensy board counts how many rotations occurred. This value is published via ROS. A speed regulator, which uses this RPM-counter, is implemented in the bachelor's thesis of Smits and Maes [29].

V. Results

A. Fixed Path

The navigation stack is installed on the NVIDIA Jetson. In order to control this navigator, we utilised an external virtual machine (VM) running Ubuntu 16.04. This VM provides the *rviz*-environment, which is a visualisation tool in ROS. Using this tool, we are able to send locations to the navigation stack, such as pose estimates and goals. These locations are specified by a position in the XYZ-space and a quaternion as orientation. The units are respectively meters and radians. When using the navigation stack, the *rviz*-tool visualises the position of the car, the calculated paths and the provided map. This visualisation is shown in figure 5. Figure 6 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 lot of computing power. The onboard NVIDIA Jetson TK1 was unable to calculate paths at a requested rate of 2 Hz. Instead, each calculation lasted approximately three seconds. Even a virtual machine with high specifications was unable to meet the requirements. 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 limitation 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.

A final adjustment is made in the goal tolerance. Because of the Ackermann-steering mechanism, chances are that the goal is never reached due to the constraints of a car-like robot. To overcome this problem, a certain tolerance is set for both position and orientation. A margin of one meter is set for the position, whereas an orientation margin of 2π is active such that any orientation is possible in order to reach the goal.

B. Waypoints

With the first implementation of the waypoints 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. After a few adjustments, the detection rate of false positives decreased drastically.

A major drawback of this algorithm is the use of raw LiDAR-data. When the car is driving along a black object, the laser beam does not return a reflection, which results in the maximum range of the sensor. Thus, the algorithm detects a corner and the car starts to manoeuvre. This problem is partially solved by taking multiple measurements on different angles. However, the problem still occurs when using large black objects. Though, none of these problems occur in environments without black objects.

An advantage of this method is the ability to maintain a high speed. By relying on raw LiDAR-data instead of location tracking, the refresh rate of the algorithm is much higher.

C. Optimisation Algorithm

As mentioned in section III-C, no implementation and results are available yet. Though, we expect that parameter optimisations are needed and many iterations of the PSO-algorithm are required in order to navigate the car using the calculated paths.

VI. Comparative Study

The first method was to navigate on a fixed path. Using this method, an accurate destination can be reached as it is specified in very precise values. Though, a tolerance is set in order to reach the goal properly with a car-like robot. In case of a holonomic robot or a robot with a differential steering

mechanism, this tolerance is not required. A drawback of this navigation stack is the slow refresh rate. Therefore, only low speeds are possible. If a faster speed is applied, the module is unable to keep up with the fast location updates. These advantages and disadvantages make this approach suitable for applications where a low speed is not really a drawback, but an exact destination is required.

The second algorithm, on the other hand, is capable of maintaining high speeds as no location tracking is needed. As mentioned in the previous section, the refresh rate of this algorithm is much higher. Each time the LiDAR-sensor publishes a message, the algorithm is executed. Hence, the refresh rate is 40 Hz. A drawback of this methodology is the inaccuracy of the goal. The algorithm is not capable of reaching an exact destination point. Instead, the goal is an abstract instruction. This waypoint approach is suitable for applications where the maximum speed is really important and the destination point does not need to be exact. An example of such an application is an indoor race track with many possible passages.

According to [10], the optimisation algorithm is able to calculate a path for a car-like robot in a complex environment. Though, a real-time update of the path is not possible yet without restarting the algorithm.

VII. Further Research

First of all, the optimisation algorithm, developed by Saska [10], needs to be implemented and compared to the other algorithms in order to finish the research.

As mentioned in the previous paragraph, the algorithm of Saska has a slow update speed. However, if we are able to combine the work of Saska with the algorithm of Nasrollahy [2], which suggests a PSO-method for using in a dynamically changing environment, then we have path planner for car-like robots which updates in real-time. This would be an advantage in comparison with the navigation stack of our first method. The only drawback would be the location tracking. Therefore, only a low speed can be maintained.

By optimising the location tracker, which is used in the navigation stack and the optimisation algorithm, we are able to increase the update rate, thus resulting in a higher maximum driving speed.

Additionally, research could be conducted in optimising the navigation stack for car-like robots.

VIII. Conclusion

In order to explore the navigation capabilities of an F1/10-car, this paper has introduced three different types of navigation algorithms. I.e., navigating by a fixed path, via waypoint instructions and via an optimisation algorithm. The paper implemented two algorithms instead of three, due to the limited time frame. These two algorithms have been compared with each other. The drawback of the first method is the limit to low driving speeds, while the accurate specification of a goal makes the algorithm suitable for exact navigation commands. The second algorithm is able to maintain higher speeds, but a goal is specified in a more abstract way. Thus, the algorithm is ideal in environments where speed is more important than the goal. The last algorithm is listed in further research, as it needs to be implemented and tested to finish the 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] A. Z. Nasrollahy, H. H. S. Javadi. Using Particle Swarm Optimization for Robot Path Planning in Dynamic Environments with Moving Obstacles and Target. *2009 Third UKSim European Symposium on Computer Modeling and Simulation*, (3):60–65, 2009.

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

[4] C. Mui. Waymo Is Crushing The Field In Driverless Cars. https://www.forbes.com/sites/chunkamui/2017/02/08/waymo-is-crushing-it, 2017-02-8. [Online; last accessed May 2017].

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

[6] 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.

[7] 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.

[8] M. A. P. Garcia, O. Montiel, O. Castillo et al. Path planning for autonomous mobile robot navigation with ant colony optimization and fuzzy cost function evaluation. *Applied Soft Computing*, 9(3):1102–1110, 2009.

[9] 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.

[10] M. Saska, M. Macas, L. Preucil et al. Robot Path Planning using Particle Swarm Optimization of Ferguson Splines. *2006 IEEE Conference on Emerging Technologies and Factory Automation*, 19(January 2007):833–839, 2006.

[11] 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.

[12] W. Dang, K. Xu, Q. Yin et al. A Path Planning Algorithm Based on Parallel Particle Swarm Optimization. In D.S. Huang, V. Bevilacqua, and P. Premaratne, editors, *Intelligent Computing Theory: 10th International Conference, ICIC 2014, Taiyuan, China, August 3-6, 2014. Proceedings*, pages 82–90. Springer International Publishing, Cham, 2014.

[13] 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.

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

[15] 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.

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

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

[18] 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].

[19] M. Dorigo, M. Birattari, T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[20] 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.

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

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

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

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

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

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

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

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

[29] B. Smits, S. Maes, and P. Hellinckx. Self-Driving F1/10 Car. Bachelor's thesis, University of Antwerp, 2017.

[30] Great Speculations. Self-Driving Cars: The Building Blocks of Transportation-as-a-Service. https://www.forbes.com/sites/greatspeculations/2016/09/20/self-driving-cars-the-building-blocks-of-transportation-as-a-service/, 2016-09-20. [Online; last accessed May 2017].

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

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

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