HALMSTAD UNIVERSITY

Master of Science in Engineering

MASTER THESIS

Monte-Carlo Tree Search in Continuous Action Spaces For Autonomous Racing

F1-Tenth

Jonatan Jönsson & Felix Stenbäck

Intelligent Systems and Digital Design
Supervisor: Slawomir Nowaczyk and Zahra Taghiyarrenani

Halmstad University, June 14, 2020–version 1.0

ABSTRACT

Autonomous cars involve problems with control and planning. In this paper, we implement and evaluate an autonomous agent based on a Monte-Carlo Tree Search in continuous action space. To facilitate the algorithm, we extend an existing simulation framework and use a GPU for faster calculations. We compare three action generators and two rewards functions. The results show that MCTS converges to an effective driving agent in static environments. However, it only succeeds at driving slow speeds in real-time. We discuss the problems that arise in dynamic and static environments and look to future work in improving the simulation tool and the MCTS algorithm. See code, https://github.com/felrock/PyRacecarSimulator

ACKNOWLEDGEMENTS

We would like to thank our supervisors Slawomir Nowaczyk and Zahra Taghiyarrenani for guiding us through this process. We thank our team apex members, Anton Olsson, Harald Lilja and Felix Rosberg. We also thank Rahul Mangharam and Hongrui Zheng at Pennsylvania University for welcoming us into the F1-tenth community. Finally we would like to thank Mikael Hindgren for allowing us to pursue this project.

Jonatan Jönsson & Felix Stenbäck

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AV	Autonomous Vehicle
IMU	Inertia Measurement Unit
CPS	Cyber-Physical System
DARPA	Defense Advanced Research Projects Agency
NN	Neural Network
FTG	Follow The Gap
FOV	Field of View
MLP	Multilayer Perceptron
MCL	Monte-Carlo Localization
MCTS	Monte-Carlo Tree Search
CPO	Contrained Policy Optimizing
MPC	Model Predictive Controller
ROS	Robot Operating System
ESC	Electronic Speed Controller
EKF	Extended Kalman Filter

1 INTRODUCTION

In this chapter we state the problem definition, the constraints we are subjected to and our contributions.

1.1 PROBLEM DEFINITION

A bird's eye view of the problem is the following, have a small car drive on a track without crashing and finishing laps as fast as possible. This is to be done on a real miniature-sized car. The car format is part of an on-going intelligent vehicles competition called F1-Tenth, and it is one-tenth of the size of an everyday car. The vehicle will be used to evaluate the effectiveness of agents and how fast actions are computed. Effectiveness is measured by comparing the performance of the different agents. The speed of action computation is limited to the hardware available on the car. The driving task is to navigate in static and dynamic environments.

For an agent to perform well in a race, it needs to make good decisions on what action to take in any given scenario. The actions are speed and steering angle. Our solution to finding good actions is to mirror the current state in a simulation that runs on the on-board computer, then simulates different actions and chooses one. To simulate different actions, we use a method called Monte-Carlo Tree Search(MCTS). We also implement Follow the gap(FTG), Policy driver and RRT* for comparison and to use as MCTS policy. Two reactive methods and one method that has a global planner. The car has LiDAR and odometry data available as input. To our knowledge, a real-time MCTS has not been used as a self-driving car solution.

The agent evaluates a given position and state of the car and output an action to perform. Monte-Carlo Tree Search(MCTS) is combined with a neural network policy which will be trained on a dataset. The policy is used to reduce unnecessary action sampling. The dataset has LiDAR points and steering angle corresponding to the input and output of the neural network. Preparations need to be made in order to facilitate MCTS. A birds eye view structure of the track, represented as a 2D map. The map is retrieved using a SLAM method, as the car is being driven by a reactive agent. Localizing while driving is also necessary for MCTS, here we use a particle filter. For the policy we used a MLP trained on driving data from several tracks. Each part is explained more in depth in Chapter 3.

1.2 BACKGROUND

Today's research in Cyber-Physical Systems(CPS) is rapidly increasing, specifically autonomous vehicles(AV). The modern full-scale platforms are not just expensive, they're also some of the most complex designed CPS. A broad majority of the experiments are done in very isolated environments, in simulations or on very restricted hardware designs.

What the creators of F1/10[1] are presenting is an autonomous racing cyber-physical platform and it's enabling the ability to address research and education for future autonomous systems [20]. It's cheap and it fits indoor. The project started out through a co-operative engagement among three universities in USA and Italy. Currently it's used in more than twenty institutions world wide and increasing.

1.3 RESTRICTIONS

1.3.1 *Contest*

The organizers of the competition have set a number of restrictions on the hardware for the vehicle.

1. A 1/10 scale rally car chassis equivalent to the Traxxas model 74054 type is allowed.
2. Only the use of stock tires, or equivalent - in size and profile, is allowed.
3. Use of Nvidia Jetson TX2 or an equivalent capability processor or anything of lower spec is allowed.
4. Use of Hokuyo 10LX or an equivalent LiDAR range sensor or anything with a lower specs allowed.
5. There are no restrictions on the use of cameras, encoders, or custom electronic speed-controllers.
6. Use of Brushless DC motor equivalent to Velineon 3500 or anything of lower spec is allowed.
7. It is up to the teams to demonstrate that they meet the above specifications.

1 http://f1tenth.org/race.html

1.3.2 *Environment*

The competition environment and our experiment environment will have the following characteristics.

- Very small amount of reflective surface.
- At least a width of 1.5m.
- 20-30cm rim along the edges of the course.

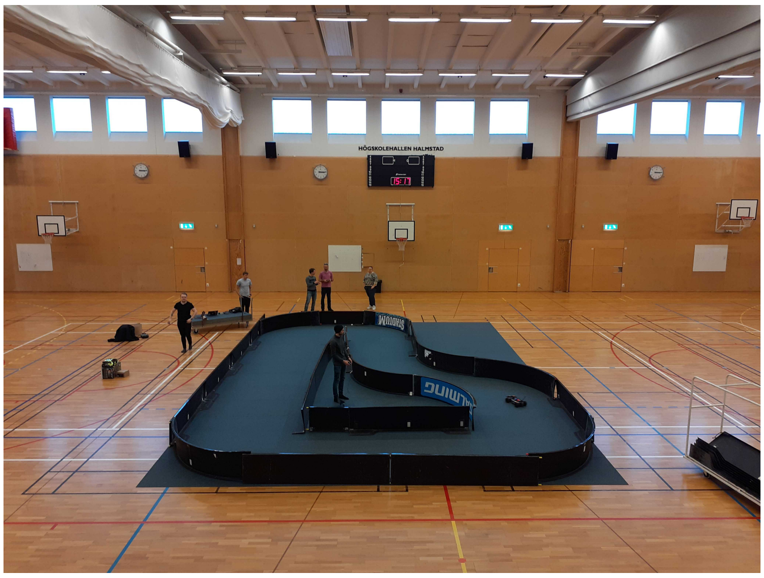


Figure 1: A race track build at Halmstad university sport center.

1.4 PURPOSE

According to World Health Organization(WHO)[2], 1.35 million people and increasing die every year in road traffic accidents. A vast majority of the accidents are caused by the state of the driver, e.g. angry or sleepy. The sensors of a self-driving car are eliminating these faults and autonomous driving is the key aspect toward safer roads. There have been a few incidents with AVs and in one particular case in early 2018 it led to the death of a pedestrian. By utilizing the F1/10 test-bed, there is only material damage risks and more challenging algorithms can be tested. Pushing autonomous driving forward has been done in the past with other contests, such as the DARPA Grand Challenge[26].

2 https://www.who.int/violence_injury_prevention/road_safety_status/2018/en/

1.5 LIMITATIONS

The limitations that are being faced in this project is mostly the computing power. All software is going to be running on a Nvidia Jetson TX2 module. Since it's going to be on a car, the calculation time is required to be minimal in order to act in time.

1.6 CONTRIBUTION

With this work we contribute with implementing and evaluating a reinforcement learning method that has showed very promising results on games. The evaluating process will extend to testing simulation speeds and effectiveness of driving. Effectiveness of driving is measure by visually logical driving actions and average lap times.

Our contribution with this work are the following,

- Implement and evaluate a MCTS with a policy network trained on LiDAR data for self-driving planning.
- Compare MCTS with different reactive methods, computation speeds and effectiveness of driving.
- Extending current F1-tenth simulation environment, enabling parallelization for faster computations in static environments.

2 LITERATURE SURVEY

In this section we provide information about existing research that is relevant to our problem definition.

2.1 AUTONOMOUS DRIVING

Self-driving cars are complex, and creating a general solution driver has been a prominent research topic in quite some time now. Back in the '80s, D. Pomerleau [22] showed that a vehicle could be controlled with a neural network end-2-end solution, ALVINN. DARPA hosted an off-road self-driving challenge [19], the competition resulted in pushing the research forward with autonomous cars such as Stanley[25] and BOSS[31] two cars that won the race. Bojarski et al[4] presented a CNN end-2-end driver, where raw data is linked to steering angles. Do et al[7] is an example of the method Bojarski et al[4] presented, tested on a small RC-car controlled with a Raspberry pi following lines similar to guidelines on real roads. Bansal et al[3] presented an imitation learning framework called ChauffeurNet, using a mid-level representation of data. Nordmark et al[18] used MCTS for decision making in a driving scenario for highway driving. They use fused sensor data as an input to make a decision and then create a trajectory. The MCTS expansion test different routes, and when the iteration is done, the best path in the tree is chosen.

2.2 SIMULATION

Cars are an expensive commodity, and there are limitations to testing complex driving scenarios that could occur, many driving tasks are therefore simulated. Simulations need to be realistic. Some of the open-source driving simulations are TORCS[29] and CARLA[8]. Where TORCS emulate a racing environment and CARLA for everyday driving, see Figure 2, 3. Game agents have been shown to learn complex problems in simulated environments, for example, AlphaZero[24], where a game agent learned to play the game of Go, Shogi, and Chess only with self-play. AlphaZero using an MCTS combined with a policy in self-play, and the previous agent AlphaGo used a policy that was trained on recorded games. Reinforcement learning models have also shown to learn strategy in complex games with incomplete information, such as AlphaStar[27], where they pre-train the agents with supervised learning and then self-play to improve. Pan et al[21] presented a framework of encoders and decoders to bridge the gap between simulation and reality, and train a reinforcement learning agent using this method.

Figure 2: TORCS

Figure 3: CARLA

2.3 CONTINUOUS ACTIONS SPACES FOR MCTS

Robot navigation and other problems with similar dynamics often have a continuous action space, for example, controlling a servo. For controlling a self-driving car, the action space is continuous for both steering and speed control. Moerland et al[17] proposed a method that builds on previously mentioned AlphaZero[24]. Where their proposed neural network outputs a continuous density instead of a discrete action space using a progressive widening and limiting the output bound. Yee et al.[30] extends MCTS to continuous actions space using kernel regression. Where their proposed method, KR-UCT prioritizes nodes that have more visits, and actions are shared through the kernel. Continuous action space has an advantage over discrete action space, which most of the papers brought up in the previous section used. Discrete action spaces for continuous problems are prominent in stuttering in navigation while continuous are smoother.

2.4 POLICY LEARNING

Finding a general driving *policy* equal to a human's ability is a hard task. Since there are many different types of driving, for example, highway driving and urban driving. The driving types have different *heuristics* for the human driver, e.g., on the highway, it's vital to keep a good distance between cars, and in an urban environment, it is important to keep an eye out for pedestrians. Achiam et al[1] proposed a method called CPO(Constrained Policy Optimizing), which optimizes a policy given some constraints for a reinforcement learning task. They showed this with a simulated robot locomotion task, where the constraints are safety. LeCun et al[11] is introducing a way to learn policies on observational data only. Their proposed way is to train a policy by unrolling a learned model of the environment dynamics while specifically penalizing two costs. The cost the policy wants to optimize, and an uncertainty cost is a deviation from the states it's trained on. They're evaluating their approach on a dataset taken from traffic cameras of a highway. Highway traffic is very dense with unpredictable actions. Their best model achieved $74.8 \pm 3.0\%$ success rate, where 100% is human driving. The evaluation consisted of two measures: whether the controlled car reaches the end of the road segment without collision or driving off the road. The distance traveled before the episode ends. Crankshaw et al[16] proposed a meta-policy network, for deep reinforcement learning. The meta-policy, using previously learned policies with similar structure, achieved 2.6x rewards than the next best policy, in a driving simulation. These policy methods are trained using reinforcement learning, while our proposed method is supervised learning. However, using constraints for not crashing is applicable and would increase the vehicle's overall safety.

2.5 LOCALIZATION

Being able to localize a vehicle in any given map is crucial for path planning, there are numerous ways of doing this. In the f1tenth testbed case, there are computation limits as well. The extended Kalman filter (EKF), which was first introduced in 1974 by A. Gelb [10], was considered a standard in the theory of nonlinear state estimation, navigation systems and global positioning systems (GPS). It has a low computational cost, but it assumes Gaussian nature of noise. What the EKF is trying to do is to find a simplified model of the problem and then find an exact solution, but with complex models, sometimes that's not enough. Therefore particle filters were introduced, it uses the whole complex model, but the solution is an estimation. P. Del Moral introduced it in 1996 [6]. Both EKF and particle filters belong to the Bayesian filters. A Bayesian filter is a general term used to estimate a state in a dynamic system from sensor measurements by predicting a future value given past and current observations.

In robot localization, it's crucial to have multi-modal hypotheses about where the robot might be. The EKF can only handle one hypothesis, and if that is incorrect, it's hard to recover. Particle filters can have the same amount of hypothesis as there are particles. In practice, the EKF typically requires the starting point of the robot to be known. The main idea of particle filters in localization is based on Monte-Carlo methods. It can handle non-Gaussian problems discretizing each data point into a particle where each particle is representing different states. It was introduced in 1999 by D. Fox et al. under the name Monte-Carlo localization (MCL) [9].

2.6 PREVIOUS F1/10 PROJECTS

F1/10th is a competition format that has been held by many organizations before, both for urban driving and racing. The competition starts with a course following the build, simple drive implementations, and ultimately a race. TunerCar[2] is a previous F1/10 project, controlled by an MPC(Model Predictive Controller), wherein each time-step, several simulations of actions are tested. Kungliga Tekniska Högskolan(KTH) has also participated in F1/10 competition[5]; the car used a one and two-layer MPC.

Other projects that use the 1/10th format are [12], where the main focus is driving correctly in an urban environment, lane follows, such as [7] which is also in the 1/10th format.

3 METHODOLOGY

In this chapter we provide information about the approach, algorithms and frameworks that are used in this project.

3.1 APPROACH

3.1.1 *Expanding on the Problem Definition*

For path planning, some problems arise. The planned path should be limited to physical movements and the size of the car. The action space for the vehicle is ample, trying all actions is an impossible task. To be able to search for a path in a simulation, the simulation needs to approximate the physicality of the car and be able to simulate LiDAR scans. Simulating LiDAR is costly, and it needs to be done for each update in the simulation to validate each state. For a static environment, it's possible to perform LiDAR scans in parallel since the map does not change with time. However, for a dynamic environment, it's not possible to do in parallel, and it makes simulating LiDAR in a dynamic environment more costly than in a static one.

3.1.2 *Deployment*

Understanding how everything will fit together, an explanation of what must be done for the vehicle to drive around the track will be provided in this section, also see Fig. 4. The individual parts will be described in more detail in further sections.

Initially, the race track needs to be mapped, and it is done through a technique called simultaneous localization and mapping (SLAM). The vehicle is driven by a reactive method algorithm around the track, and once it is done, the map is stored. The next step in the process is to create a global path for the AV to follow. The path is captured by driving the car around the track manually, and the coordinates are logged to a file.

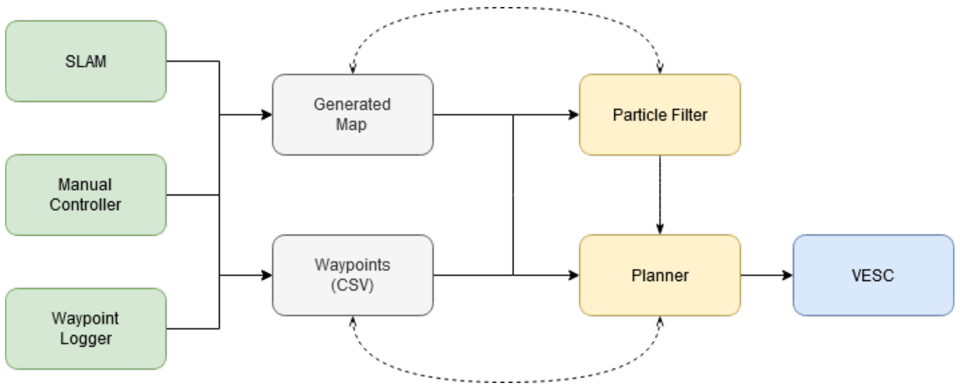


Figure 4: The deployment workflow.

3.2 ALGORITHMS AND CONCEPTS

3.2.1 *Simultaneous Localization And Mapping(SLAM)*

Hector SLAM is available as a package for ROS[14]. It allows a vehicle to map a region while simultaneously localizing within that region. Hector SLAM needs no odometry and uses low computation resources. Hector SLAM is launched as a ROS node and publishes a map and current pose data.

3.2.2 *Localization*

For localization, we use a particle filter, as explained in section 2.5. We are utilizing a fast Monte-Carlo localization algorithm developed at MIT by C. Walsh and S. Karaman in 2017 [28]. They are proposing a method that is accelerating the ray casting in a two-dimensional occupancy grid. It is robust against unmapped obstacles. In simplicity, the algorithm is casting rays from different positions in the map and then compare it with the real LiDAR data to find the location.

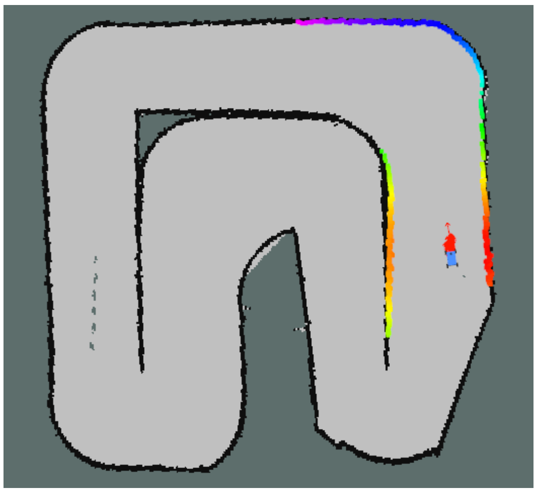


Figure 5: Red particles simulates the vehicles position, visualized with Rviz.

Ray Marching

There are numerous ways of finding edges with ray casting, but from the results presented in Walsh's and Karaman's paper, the RMGPU outperformed the other methods. First off, it's utilizing the GPU but also using a technique called ray marching. Ray marching is iteratively trying to find an intersection by using a circular search space. The radius represents the distance to the nearest obstacle, see Fig. 6.

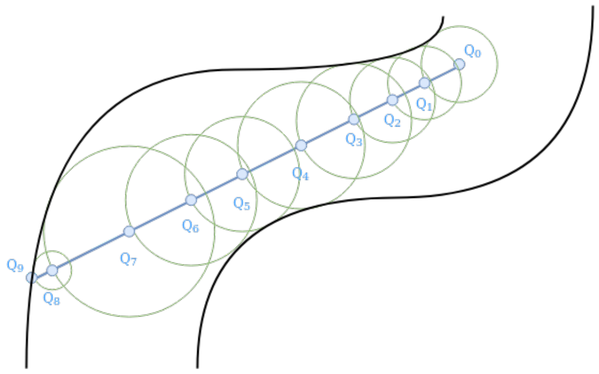


Figure 6: Visualization of ray marching where Q_0 is the hypothesized position.

3.2.3 *Motion Planning*

Motion planning is a term frequently used in the field of robotics. It's used to find a sequence of actions for a robot to move from one state to another. It will use its input to produce actions and, in this case, send it to the ESC. The actions are speed and steering angle. When working with motion planning, there are four types of workspaces, configuration, free, obstacle, and target space, see Fig. 7.

- **Configuration space**, describes the position and direction of the robot, but also the set of all possible configurations, represented as (x, y, θ).
- **Free space**, is the set of configurations that will avoid any collision with an obstacle.
- **Obstacle space**, is the space the robot can't move to.
- **Target space**, is the subspace of free space which the goal of the robot is. In the global perspective the target space is always observable, but in the local perspective it isn't always the same case. It's solved by generating sub-goals in the observable space.

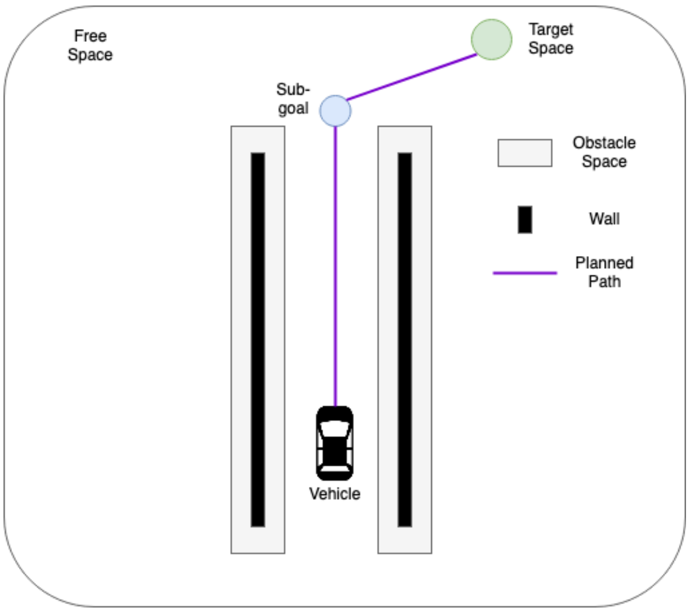


Figure 7: Motion planning in the different work spaces.

In high dimensional motion planning the sampled-based algorithms are currently considered the state-of-the-art, e.g. A*.

3.2.4 *Monte Carlo Tree Search*

Monte Carlo Tree Seach(MCTS) is an algorithm that evaluates decisions by exploring states in a tree structure. This requires a simulation tool to evaluate each state. The algorithm has four phases, *Selection*, *Expansion*, *Simulation* and *Backpropagation*.

Node **selection** is made with Upper Confidence bounds applied to Trees(UCT), where nodes are selected for visitation based on their reward, total visitations in the tree, and node-specific visitations, see Eq. 1. It is using the uncertainty in the action estimation to balance exploration and exploitation. *Exploitation* is benefiting in the short-term by selecting a greedy action to get the most reward. *Exploration* is benefiting in the long-term by improving the knowledge about each action.

$$UCT_i = X_i + C \times \sqrt{\frac{\ln(n)}{n_i}} \tag{1}$$

Where:

- X_i is the reward over visitations of the child.
- n is the number of visitations of the parent.
- n_i is the number of visitations of the child.
- C is an exploration constant.

Most selection functions, including UCT, require every action to be simulated once, this is not applicable in continuous action spaces. A solution to this is progressive widening. It is limiting the number of actions evaluated in a node based on the visits. Once the best available action is estimated, it will consider additional actions. The tree grows deeper in the promising parts according to UCT, and progressive widening assures it grows wider in the same part through **expansion**. The decision is determined by Eq. 2. The decision is based on keeping the number of actions bounded to the number of visits. Nodes with high visitations are more likely to expand. If the child hasn't expanded before, the state is passed to the policy network to produce an action. The other children do not use the policy network because the neural network will predict the same action for every child. Therefore an approach where the action is generated from an interval created from the initial prediction is used.

$$\sqrt{\sum_{a \in A} n_a} < |A| \tag{2}$$

Where *A* is actions considered in state, and n_a is the number of visitations of a child.

Simulation or *rollout* is when a new node has been created, and to validate its state; random actions are done. A rollout is a set of actions that are uniform random moves. In discrete problems, e.g., chess, a rollout is finished until the game is decided, win, loss or draw. In our case, that is not possible; therefore, the rollout is set to a fixed number of actions. A rollout is done either if the node is terminal or until the specified iteration count is fulfilled. It is terminal if the vehicle has crashed.

Backpropagation is used to calculate a branch's total score by adding every node's reward. It is then later used to determine the best possible action.

Figure 8 is an illustration of a theoretical scenario for an MCTS, where S_i are states and leaves of the graph are actions. Following the graph, the MCTS is selecting a node based on UCT and then expanding by generating a new action using the neural network. Next up is to do a rollout to evaluate the action and the reward achieved from the rollout is added to the parents through backpropagation. This compiles one MCTS iteration. The number of iteration is limited to a computational budget, which is the time the MCTS is allowed to use to predict an action.

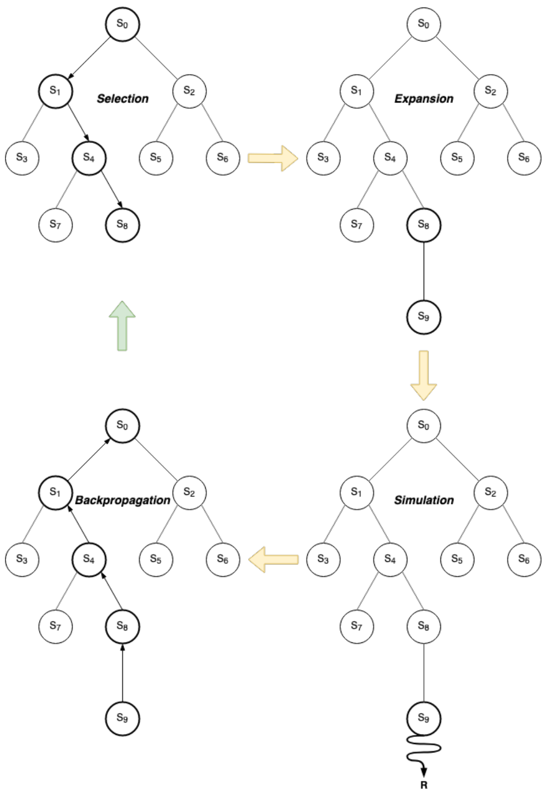


Figure 8: S_i are states in the simulator and each has a reward.

3.3 FRAMEWORKS

3.3.1 *Testbed*

The test bed is an RC-car with a computation module, designed by F1-Tenth. The complete guide to building is available online and is part of a contest format called F1-Tenth. The car is running a Ubuntu 18.04 arm version on an Nvidia Jetson TX2, with sensors such as Hokyuo 10LX, Intellisense depth camera, and an IMU. A speed controller, VESC6, is used to control velocity and steering. All these components are controlled in software by ROS, where each part is a ROS-node. See Figure 9.

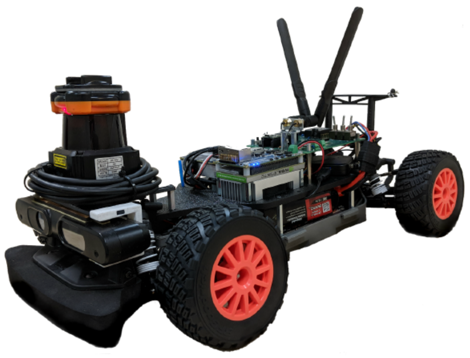


Figure 9: F1-Tenth racing car

3.3.2 *Robot Operating System*

Robot Operating System(ROS) is a collection of tools and libraries. It provides a lot of useful functionalities such as message passing and package management. The recommended parts of the F1-Tenth format are supported by ROS.

3.3.3 *Simulation*

The simulation framework is called racecar simulator; it's available on Github[1]. It's written in C++ and is built to work as a ROS node. The simulation needs a map to be able to simulate LiDAR points, the map is stored in a vector, and the map resolution is 2048x2048. LiDAR points are simulated using a ray-tracing technique, and each LiDAR point is a ray-trace. Configurable parameters are, weight and size of the car, LiDAR points to simulate, and field of view. These parameters are configured to our car, and the field of view is set to 270 degrees centered in front of the vehicle.

1 https://github.com/mit-racecar/racecar_simulator

4 RESULTS

In this chapter we provide results from the development experiments conducted.

4.1 ALGORITHMS

In this section we provide information about the algorithms that the self-driving agents use.

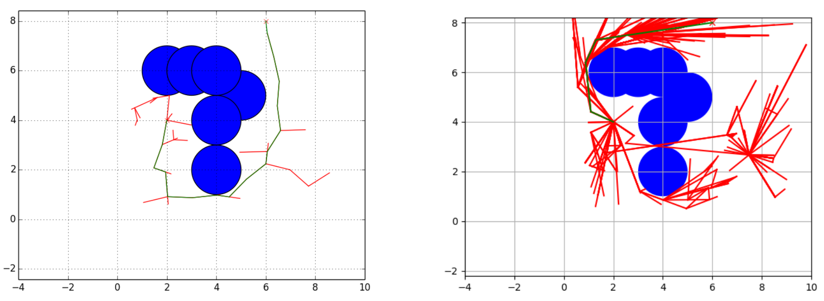
4.1.1 *Follow The Gap*

FTG is a well known reactive method that can avoid obstacles in a map. It has been used in previous f1tenth competitions where in 2018 they won the competition. It's searching for the closest point and creates a "bubble" with a fixed radius and points within are set to zero to indicate that they are out of the equation. Then it's measuring gaps in the lidar data, a gap is a sequence of measurements of which it's higher than a certain threshold, the gap with the largest sequence is the gap to follow. There are numerous approaches of where in this gap the vehicle should go, e.g. straight in the middle or the point which has the largest measurement. Following the middle showed superior performance. This method is used to gather the map with SLAM.

4.1.2 *Rapidly Exploring Random Tree (RRT)*

RRT is a sampling-based path planning algorithm developed in 1998 by S. Lavalle [15]. It's a fairly quick search algorithm which is generating points randomly and connects to the closest node and it creates a tree-like structure until it has reached the goal region. The major problem with RRT is that the path is often not optimized.

Therefore an optimized version called RRT* came along but not until 2011 [13]. The first key addition to the algorithm is a cost feature, which is the distance each vertex has traveled relative to the parent. As the closest node is found the RRT* is utilizing a neighborhood around the node and if a node with a lower cost is found, the node gets replaced. That is the second key addition, rewiring of the tree. See Fig. 10 for a comparison of the different techniques developed in python for demonstration purposes.



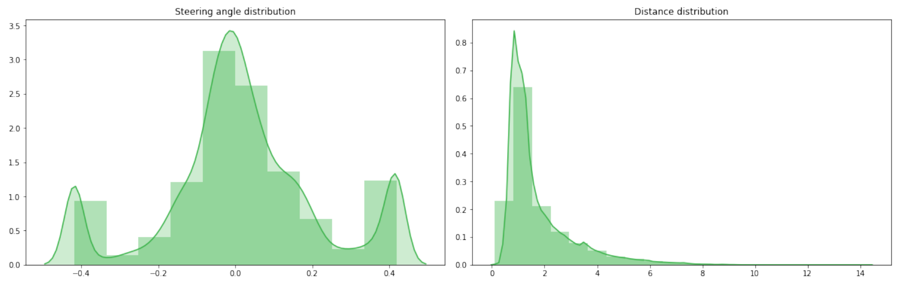
(a) RRT (b) RRT*

Figure 10: Path planning with max 500 iterations.

The RRT* algorithm is implemented as the local planner with a dynamic occupancy grid, enabling it to navigate through a map that has objects which are not present in the original map. An occupancy grid is a binary mapping of obstacles. It is following the principles mentioned in section 3.2.3. A value of 0 represents *free space* and 1 is for *obstacle space*, which can be the original map and also new obstacles. It is created from the lidar scan callback. At each measurement in front of the vehicle, a bounding box of obstacle space is created. Any trajectory calculated by the RRT* that intersects with the obstacle space will be declined. The occupancy grid is dynamic to solve the issue with dynamic objects leaving their mark on the grid. Meaning after a certain amount of boxes, it removes them in a first-in, first-out (FIFO) manner.
The RRT* algorithm was implemented as the planner together with an occupancy grid explained in section 3.2.3. The occupancy grid is updated through the LiDAR scan callback. A demonstration in simulation can be seen in this video, RRT*.

4.2 DATASET

The dataset was gathered during multiple sessions on different tracks. It consists of lidar data and actions taken by the VESC, in this case, steering angle and velocity. The vehicle was driven either by manual control or a reactive method, such as FTG. The steering angle is measured in radians and velocity is in meters per second, see Fig. 11 for the distribution of the steering angle and lidar measurement distances. Because of the limitations on the physical aspects of the vehicle the steering angle has a limit of ± 0.42 radians. The lidar data consists of 1081 data points which demonstrates a distance in meter to the obstacle, with a max distance of 15m.



(a) Steering angle (b) Distance

Figure 11: Data distribution of steering angle and lidar distance. Steering angles above zero is left and vice versa.

Looking at the steering angle distribution and mean distance (fig. 12), some similarities indicate a small bias. The dataset contains more left actions and also the mean distance value for left *(index = 900)* are lower than right *(index = 180)*. This means it prefers to stay closer to the left wall.

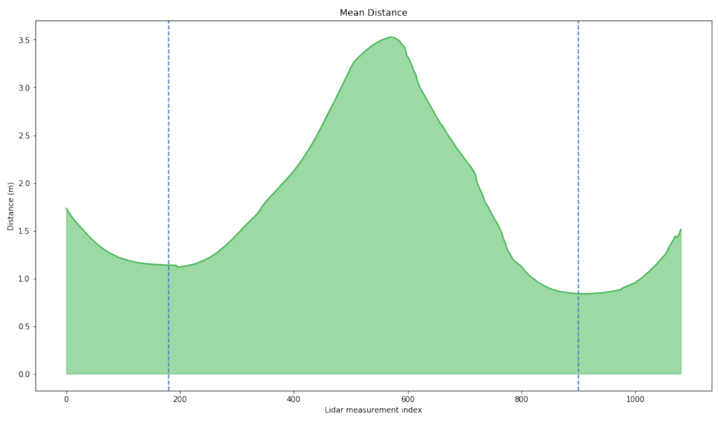


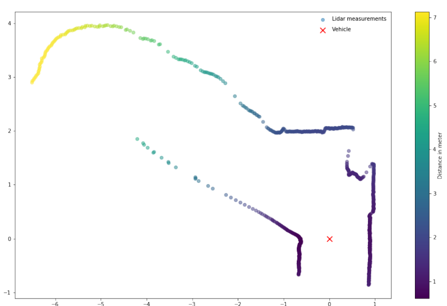
Figure 12: The mean distance for each lidar measurement. Blue lines are the points 90 degrees to the sides.

The dataset is divided into two groups, real and simulated. To generalize a problem, a vast amount of data is needed, and due to time constraints, the data couldn't only consist of real scenarios, hence the simulated group. The simulated group was created by driving a car in the simulation environment used by the MCTS. It was driving on maps that is not included in subgroup real. The primary purpose of the dataset is to train a neural network that will act as a policy for the MCTS. The policy network is operating in a simulated copy of the real scenario, and therefore it will be acceptable to have a simulated part of the dataset. The total amount of data points is 31 963.

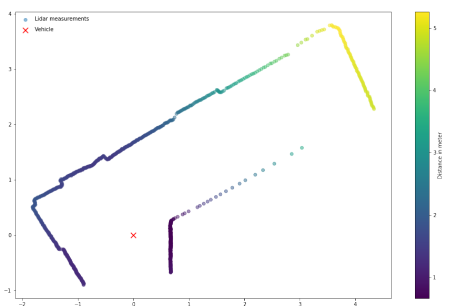
Table 1: F_n being a lidar point's distance in meter. Angle and velocity is the action taken by the vesc.

		Input				*Output*	
Type	**#**	**F_1**	**F_2**	**...**	**F_1081**	**Angle**	**Velocity**
Real	1	0.768	0.768	...	1.912	0.370	2.0
	2	0.759	0.773	...	1.896	0.381	2.0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	15 660	1.598	1.592	...	0.860	-0.045	5.05
	15 661	1.627	1.623	..	0.855	-0.049	5.05
Simulated	1	1.243	1.245	...	0.812	-0.121	3.5
	2	1.279	1.282	...	0.824	-0.135	3.5
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	16 301	0.353	0.361	...	1.729	0.391	3.0
	16 302	0.332	0.342	..	1.791	0.401	3.0

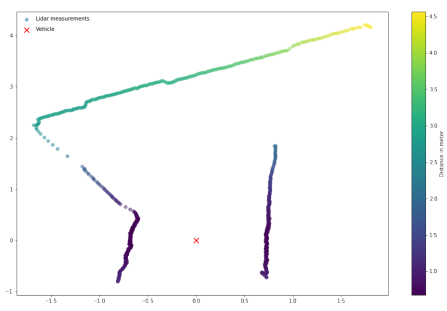
Different scenarios are demonstrated in Fig. 13 from subgroup, real. The vehicle is driving on a map collected in a hallway at Halmstad University. The colored points in the graphs are the 1081 lidar measures with the distance weighted in color. In each scenario, the action taken at that time is discretized for demonstration purposes. In scenario a) and d), there also exists an obstacle, which is the anomaly along the right wall. Obstacles are present to avoid any possible bias to completely straight walls.



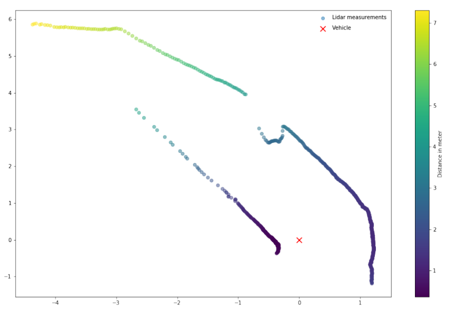
(a) Full left turn



(b) Full right turn.



(c) Straight



(d) Slight left turn

Figure 13: Example scenarios from the dataset.

4.3 NEURAL NETWORK

The neural network that will act as the policy will be of type regression, since it's in a continuous action space. The purpose of the neural network is to predict the steering angle. It will be used to reduce the search space of actions. The network is constructed using the open-source library Keras[1], which is running on top of Tensorflow[2]. From the original dataset 720 lidar points were used. Originally it has a 270° view but after reduction, the NN only uses 180°. It is only using 720 points because the rest of the points are behind the vehicle and therefore not of interest. The final model is a multilayer perceptron(MLP) with four hidden layers, see Fig 15.

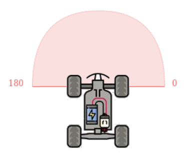


Figure 14: NN FOV

1 https://keras.io/
2 https://www.tensorflow.org/

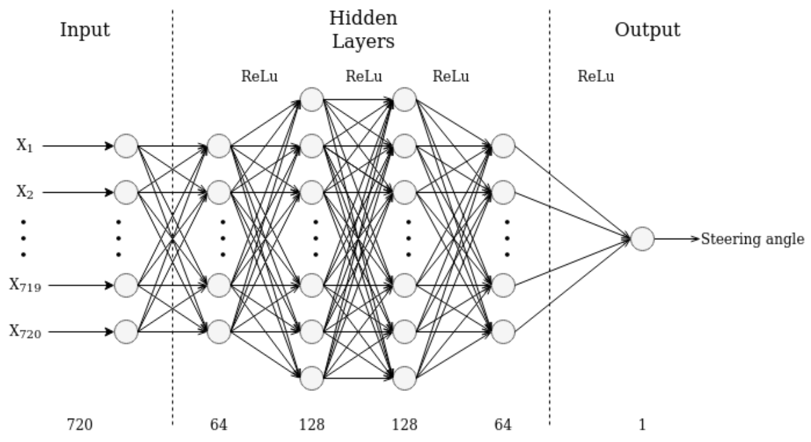


Figure 15: Proposed MLP architecture.

The proposed model was used together with Adam optimizer, a learning rate of 0.001, the batch size was set to 24 and 100 epochs with mean squared error as the loss function. The dataset was split accordingly, training = 23 092, validation = 4076 and testing = 4795. The input is then normalized and the model is trained in a supervised manner with steering angle as output. The validation set is used during training to validate each epoch. With early stopping, the model can avoid overfitting, and this is done whenever the validation error starts increasing above a certain threshold. Overfitting will increase the generalization error of the model, and must be avoided.

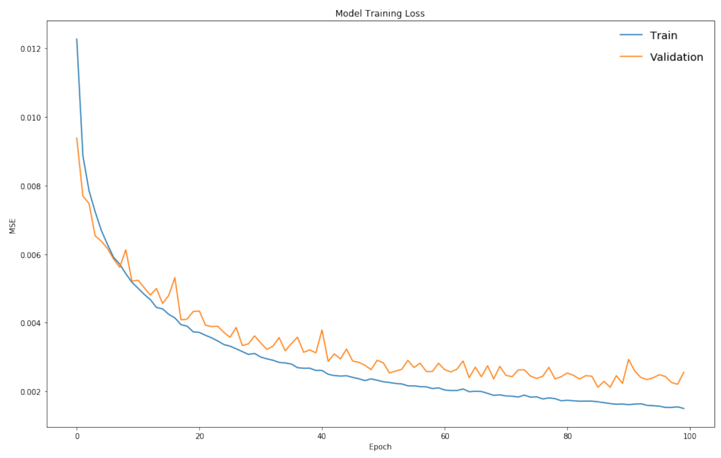
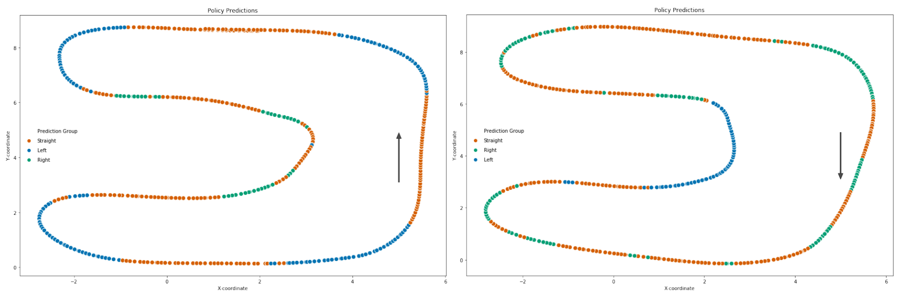


Figure 16: Training over 100 epochs.

On the test set, the model achieved a mean absolute error (MAE) of 0.0324. But this is on data that is very similar to the training data. Thus new data was collected.

4.3.1 *Generalization*

A generalization set was collected, it consists of 3958 data samples. It was collected on maps that exist in the training dataset but were gathered at a later occasion, which gives a slight difference in the positioning. To visualize the performance of the policy network, a FTG agent was driving around a map and then let the policy network predict the steering angle. It was done on a u-shaped map built at Halmstad University, see Fig. 19.



(a) *MAE = 0.129 radians.* (b) *MAE = 0.103 radians.*

Figure 17: A FTG agent driving around a map two ways with the policy output grouped and added to demonstrate what the policy is predicting.

A mean absolute error of 0.116 radians on the generalization test indicates that the policy network can distinguish whether it's a left or right turn well. The model will be used as a policy for the MCTS.

Libraries

Keras is an exstension of Tensorflow but with inference, it comes with some overhead; therefore, the model was converted to a TensorFlow type model called protocol buffer to increase inference speed. It's done by freezing the graph, which is a way to identify and save the required information, such as the graph and weights. The inference time decreased from 1.5 ms to 0.9 ms.

4.4 simulation

In this section we are providing results from the simulations.

4.4.1 *Building a Simulator with Cython and Range_libc*

We have implemented the LiDAR and the kinetic simulations using an existing library called range_libc and porting a frequently used kinetic simulator developed by C. Walsh and S. Karaman [28]. Range_libc is a library with a few different implementations but mainly used for running ray-tracing algorithms. LiDAR and Kinetic simulation code are written in C++, ported with Cython, and put together with Python.

4.4.2 *UCT with Racecar Simulator*

The extension of the simulation tool is primarily to facilitate an MCTS algorithm. The most computationally expensive operations of the simulation are inference with the policy network and applying LiDAR simulation, inferring takes approximately *1ms*, and lidar scan a third of that. Consequently, the expensive operation of the MCTS algorithm becomes the rollout. To accelerate the rollout stage, we first simulate only the kinetics and pass all the states as a large chunk to the GPU. It's faster by a large margin, as can be seen in fig. 18. *RMGPU* and *RM* is the proposed simulation environment with and without GPU. *UPENN* is an existing simulation environment built at the University of Pennsylvania[3], which is an improvement of the initial racecar simulation mentioned above. Figure 18 primarily shows in increase time performance in simulated LiDAR scans run, where running simulations in parallel on the GPU vastly outperform the other options.

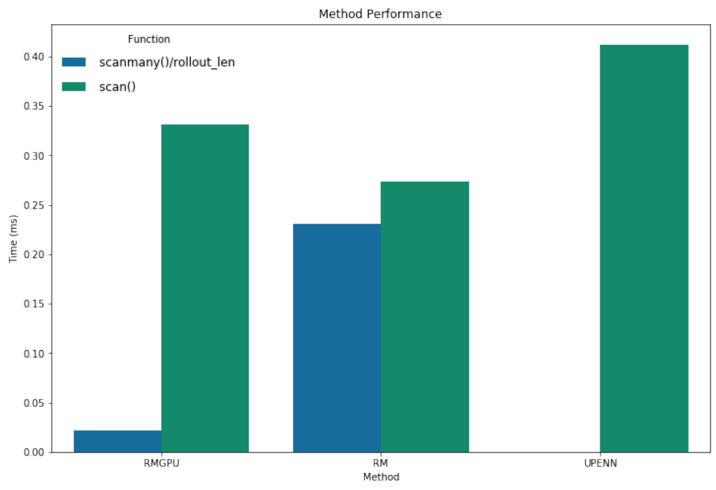


Figure 18: Time performance of ray-tracing between simulations.

4.4.3 *Monte-Carlo Tree Search*

The MCTS is implemented as described in Section 3.2.4. The proposed racecar simulator together with range_libc for simulating lidar scan with ray marching was used as the simulator to validate each action. Each mcts phase is demonstrated in code below, see Algorithm 1. This is done for every action the vehicle is taking. The following tests were performed with a Intel i7 Skylake CPU with a GTX 1070

3 https://github.com/mlab-upenn/racecar_simulator

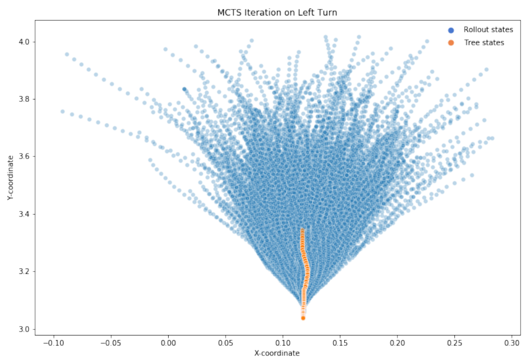
graphics card.

Algorithm 1 MCTS for Continuous Action Spaces

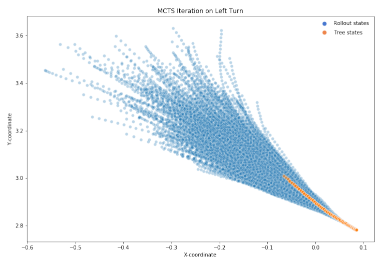
```
1:  Create root node
2:  while time < budget do
3:      procedure ITERATION(node, expanded=False)
4:          action ← $\mathrm{argmax}_a \bar{v}_a + C\sqrt{\frac{\log \sum n_b}{n_a}}$        ▷ Selection
5:          if $\sqrt{\sum n_b} < \|A\|$ then
6:              Iteration(action, expanded=False)        ▷ Widening
7:          end if
8:          if not expanded then        ▷ Expansion
9:              newAction ← generateAction()        ▷ Policy
10:             add newAction to tree
11:             if not terminal then
12:                 $rv$ ← rollout        ▷ Simulation
13:             end if
14:         end if
15:         Update tree        ▷ Backpropogation
16:     end procedure
17: end while
18: Find best child
```

Action Generators

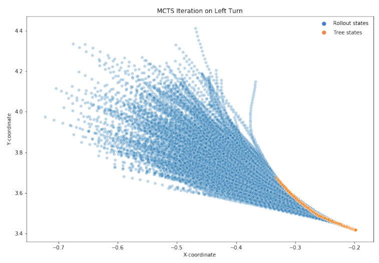
The MCTS used three action generators for the expansion phase. The neural network, FTG and a randomly sampled action. A selected node that has expanded prior will generate the same state if passed to the action generator. So instead, a random sample in a specified bound given by the first generated action. The span for the Neural network is 2.3 degrees. The span is based on the mean error derived from the neural network training. The same span is used for the other action generators as well. In Figure 19, an MCTS searched for 10 seconds, placed roughly in the same place at a left turn on our test track starting at 0 velocities. The blue trails are roll-out states x, y coordinates, and the orange trail is the path of the tree structure. The random action generator (a) does not seem to converge to actions taking a left turn, while both the neural network(b) and FTG(b) favors actions that are to the left. Both FTG, and the Neural network generates more samples to the left, while the random action generator does not.



(a) *Random*



(b) *Neural Network*



(c) *FTG*

Figure 19: Tree and rollout state coordinates generated by MCTS with a computational budget of 10 seconds.

Reward functions

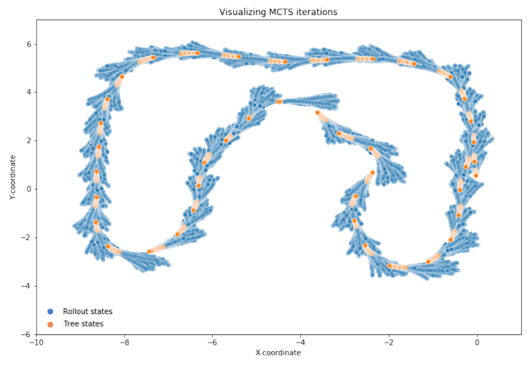
Two different reward functions were tested with the MCTS. The reward functions role is to estimate the value of a node after a rollout is performed.

$$\sum_{r \in R} V_r \qquad (3) \qquad\qquad \sum_{r \in R} \frac{V_r}{||D_r||} \qquad (4)$$

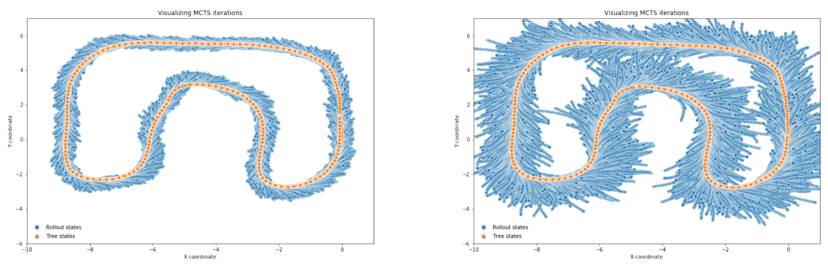
The more straightforward reward function seen in Eq 3 is a total velocity accumulator until crash or end of rollout iterations V_r is the state's current velocity. The other uses predefined waypoints and takes the speed over the distance to the closest waypoint, seen in Eq 4, $||D_r||$ is the distance to the waypoint.

Computational budget

Computational budget for the MCTS has some clear trade-offs, choosing more time to estimate a good path will result in slower actions, and the estimation will inherently become worse because the state of the car keeps changing. In the simulation, it is possible to change the frequency of the car state changes. Thus, lowering the frequency results in more computation for the algorithm without the trade-off for slower actions.



(a) Budget at 0.01 seconds



(b) Budget at 0.05 seconds (c) Budget at 0.1 secondss

Figure 20: Tree and rollout state coordinates generated by MCTS for a single lap on our test track.

In Figure 20 one lap on the test track is done in simulation. For (a), the budget is set to 0.01, which is the same as the update rate of the simulation, and rollout max iteration is set to 50. The search of the MCTS with the defined budgets performed 7, 43, and 109 iterations. For $(b), (c)$ the budget is increases 5 and 10 fold, and the update rate is lowered to equal the budget. For (c) rollout, max iterations were increased to 100.

A time trial in the simulation was done to compare reactive methods with their corresponding MCTS. The test was performed on four test tracks, two bigger ones, and two smaller ones, respectively. See Table 2 and 3. The number in the cell represents the average lap time in seconds, calculated over ten completed laps. The MCTS used 0.01, 0.05, 0.1, and 0.2 seconds as budget. Numbers marked with a * indicates that the agent crashed on its first lap. The maps are available in the Appendix Test Maps.

Table 2: Average lap time (seconds) with budgets(B). Instant action (IA) are none MCTS-agents.

U-shaped						Big F5				
Agent	IA	$B_{0.01}$	$B_{0.05}$	$B_{0.1}$	$B_{0.2}$	IA	$B_{0.01}$	$B_{0.05}$	$B_{0.1}$	$B_{0.2}$
FG	20.7	-	-	-	-	9.0	-	-	-	-
RRT*	**18.39**	-	-	-	-	8.57	-	-	-	-
NN	7.50*	-	-	-	-	5.8*	-	-	-	-
$MCTS_{FG}$	-	20.1	19.6	19.3	19.1	-	8.90	8.60	8.32	**8.30**
$MCTS_{NN}$	-	7.20*	8.40*	8.60*	8.80*	-	4.92*	6.05*	6.12*	6.16*
$MCTS_{RND}$	-	3.31*	3.52*	4.04*	4.06*	-	1.32*	1.35*	1.41*	1.44*

Table 3: Average lap time (seconds) with budgets(B). Instant action (IA) are none MCTS-agents.

No Box						D-shaped				
Agent	IA	$B_{0.01}$	$B_{0.05}$	$B_{0.1}$	$B_{0.2}$	IA	$B_{0.01}$	$B_{0.05}$	$B_{0.1}$	$B_{0.2}$
FG	9.05	-	-	-	-	6.91	-	-	-	-
RRT*	**7.78**	-	-	-	-	**5.82**	-	-	-	-
NN	2.51*	-	-	-	-	5.74*	-	-	-	-
$MCTS_{FG}$	-	7.21*	9.01	9.0	9.1	-	2.38*	2.41*	6.81	6.85
$MCTS_{NN}$	-	2.51*	2.5*	2.5*	2.51*	-	6.2	5.66*	6.1	6.1
$MCTS_{RND}$	-	1.2*	2.25*	2.23*	2.5*	-	2.4*	2.41*	2.42*	2.4*

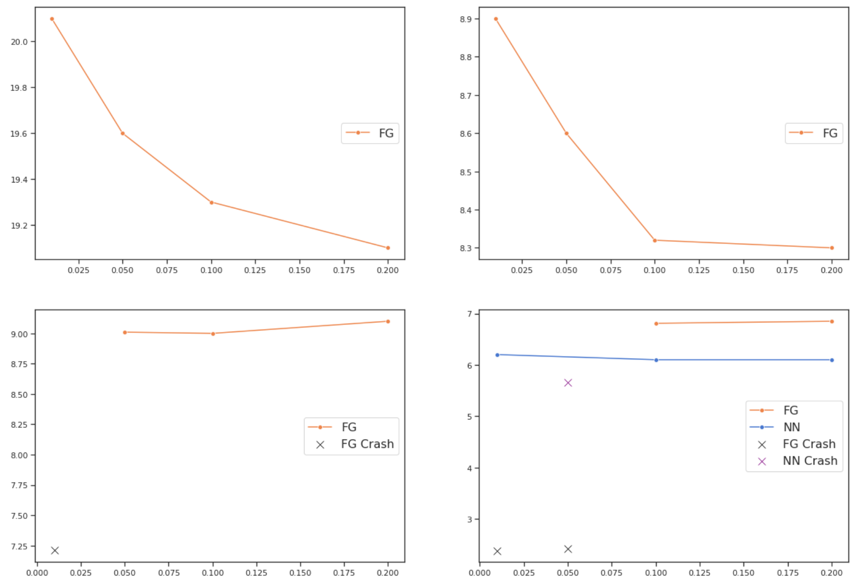


Figure 21: The performance change in seconds with budget of those MCTS agents that finished laps. *Graphical description of the tables above.*

4.5 testing mcts on f1-tenth car

Testing MCTS agent on the physical car only worked for low speeds, e.g., $2.0 m/s$. This is due to the computational constraints of the TX2. The TX2 is running, all the ROS Nodes, a particle filter, and then the MCTS agent on top of this. Running with a budget of 0.01 seconds yielded a iteration count 2-3, with FTG as an action generator. By using neural network policy as a generator with the same budget yielded 1-2 iterations. The car was able to drive several laps without crashing.

5 DISCUSSION

In this chapter we discuss our findings, problems our current solutions can not solve and applicability in other fields.

5.1 ANALYSIS OF THE RESULTS

In this section we analyze our results and make conclusions as to why we got the result.

5.1.1 *Simulation Extension*

During this work we have implemented a simulation environment based on racecar_simulator[1] by MIT. We also have made our driving agents publicly available with the simulation environment, it is published on a Github repository[2].

As shown in the bar plot, Fig 18, running the ray-tracing methods on the GPU gave a faster result. But we want to point out that this only works for static environments. Since the map and it's mapped objects do not move, the distance transformation does not need to re-calculated for each step. Maximum rollout iterations were set to 100 for the majority of the tests. Instead of evaluating one state and then scanning the next, we scan for the entire max iterations and later evaluate. Evaluating includes checking for crashes and calculating a reward.

5.1.2 *Neural Network Policy*

We ended up using a MLP because LiDAR is a 1-D array, another reason was that a smaller and simpler model has faster inference time. The structure of the model was iterated with a simple grid-search. The neural network could probably perform better with hyperparameter tuning.

A dataset consisting of lidar points and actions of the control unit was created. The dataset is in two equal-sized parts, one from the testbed and one from the simulation. The data from the testbed were collected from different tracks that were built on Halmstad University.

1 https://github.com/mit-racecar/racecar_simulator
2 https://github.com/felrock/PyRacecarSimulator

The simulation part was created in the f1tenth simulation environment. A variety of maps was used, from a simple circular track to a more complex shape such as a u-shape. With this dataset, a multilayer perceptron model with four hidden layers was trained and achieved a mean absolute error of 0.03 on the test set. A generalization test was also conducted to test the performance even further. The vehicle was placed in a map that had not been included during training, the model achieved an MAE of 0.116, which is $6-7°$. It concluded that the neural network could distinguish whether it is a right or a left turn, which is its purpose.

5.1.3 *MCTS Comparison*

It was shown in Figure 20 that with a higher budget for the MCTS the agent was less prone to over-steering and had smoother actions than agents with a lower budget, however, limited to our implementation and testing capabilities. To conclude that MCTS is better would need more investigation. The tree structured of the MCTS follow the actual path more accurately with budget set to 0.05 and 0.1.

For the time trial test, the difference in lap times was smaller between agents on the maps Map D and No box. This is because those maps are smaller U-shape and Big F5. The increase in budget for MCTS on map Big F5 and U-Shape showed a decrease in laps average, for FTG. Using randomly selected actions did not finish a lap on any of the tracks, and the time before the crash did not increase with a budget increase. The Neural Network however, performed a good lap time on Map D, with the exception of crashing with budget 0.05. Also selecting an action for the Neural network is slower than for FTG, which results in 2x lower iterations per budget. The RRT* implementation performed the best on the majority of the tracks, where it only performed worse than $MCTS_FG$ on Big F5. This shows our implementation of MCTS in it's current state falls short.

Finally a test on the F1-Tenth testbed was performed and it showed that the car could drive at low speeds, with low iteration count on MCTS. At these low iteration counts, MCTS becomes almost obsolete, and driving with purely follow the gap or the policy would be a better choice for driving at higher speeds. Since the first child is sampled directly from the chosen policy, and if that child is then selected, the driving mechanism is only of the policy.

5.2 Static and dynamic obstacles

For the task of planning a path with an environment that has both static and dynamic obstacles force a driving agent to have a mechanism to classify them. It is crucial to know what objects move to be able to plan. If that mechanism is available to the agent, there is still the problem of translating the dynamic object into the simulation and approximating the path of the dynamic object. The task of simulating LiDAR becomes more costly, because of approximating the path of the translated dynamic object, assuming it's an F1-tenth car. The method of ray-marching performs worse when the state of the map have incremental steps.

5.3 Real-time issues

The major problem the planner is facing is the computational constraints of an on-car processing unit, in our case, the Nvidia Jetson tx2. Which adds to the issue of acting in time, the faster the vehicle drives the quicker response time the driver needs to have. With a computational budget of 0.1s, it's able to perform 30 iterations with NN, which is too low for the MCTS to converge on an improved action. The policy is trained to reduce the search space and not drive the vehicle. In section 4.4.3, we demonstrated that by increasing processing performance, the planner converges to a better chosen path on the test track. Another approach would be to improve the policy, which would lead to the decreased necessity of alternate trajectories.

5.4 Limitation of simulation

MCTS relies heavily on that the simulated actions are as close to as real scenario. In the case of testing the MCTS, the simulation that is running inside MCTS is the same as the actual world. So actions performed in both simulations work the same, however, evaluating against reality there exists an error between the simulated action and the real performed action. Since MCTS was quite slow we where unable to test this. The depth of the tree was only 1-2, which is only ~0.02 seconds simulated.

5.5 Approximating an optimal path

The initial idea for the MCTS approach was to use it only as a self-driving agent. But by allowing it to run during a more extended period of time, lap times will become better as the search depth grows. Once a lap is done, the expansion part would only allow expansion by widening. Then over time, an optimal path will be approximated with the physics acting on the car, however, limited to the simulations accuracy.

6 CONCLUSION

The topic of applying a Monte-Carlo Tree Search in continuous action spaces for autonomous racing has been explored in this study. It was implemented in the F1-tenth format, which is a vehicle one-tenth of the size of a real vehicle. With the on-board processing unit, it was shown that MCTS is not suitable for a real-time problem such as racing. But as the computational performance increased, the action it predicted got better. It's visible in Figure 20 that with an increase in budget the agent is less prone to over-steering. This also shows in the time trial test, where on the larger maps, a higher computational budget gave a lower average lap time, for Tabel 2 and Table 3. For the main reason for this project, of driving a car in real time, MCTS is not suitable with the current set up. The computing device on the car, combined with optimizing the software, should be considered to use this method. A better option would be using a method like RRT* for driving in the environments that we have tested.

6.1 FUTURE WORK

To proceed with this work a next step could be to implement a simulation environment that can handle multi agent problems. Dealing with the problems of doing ray-tracing with incremental updates of obstacles. Or applying these methods on an existing simulator such as CARLA, or even in games such as GTA 5.

An interesting topic to follow up is a way to translate objects into the simulation. By using senors such camera and LiDAR estimate the position on the map. Translating meaning, localizing and keeping track of dynamic objects in the race. Another possible enhancement is building a light weight simulation specific for multi agent racing. To be able to test different agents and how the behave in a dynamic setting.

The MCTS algorithm we are using is popular among reinforcement learning tasks. Such as agents that have outperformed humans is AlphaZero[24], AlphaStar[27] and more recently MuZero [23]. Where self-play is the key feature used that would be interesting to see done on a problem dealing with autonomous vehicles.

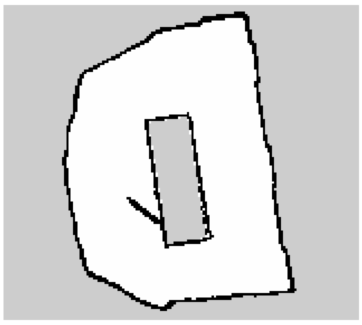
Furthermore, as we did not do any extensive testing of the hyper parameters of the MCTS or of the roll-out policy. They were set with values inspired from previous works. For the code to work better for real-time handling it will need to be optimized even further.

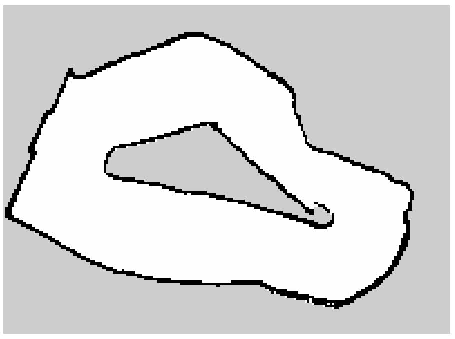
Part I

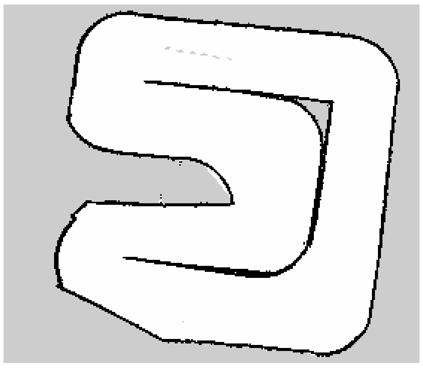
APPENDIX

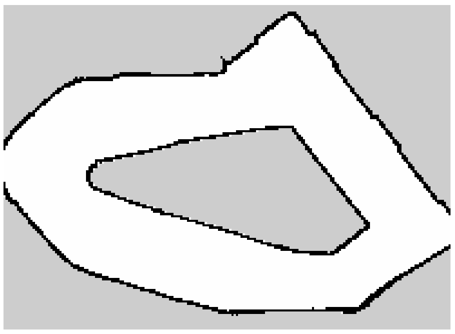
A

APPENDIX TEST MAPS


(a) Map D


(b) Nobox


(c) Map U


(d) F5 Big

BIBLIOGRAPHY

[1] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. *CoRR*, abs/1705.10528, 2017. URL http://arxiv.org/abs/1705.10528.

[2] J. Auckley, A. Jain, K. Luong, R. Mangharam, M. Okelly, and H. Zheng. Tech Report: TunerCar: A Superoptimization Toolchain for Autonomous Racing. Technical Report UPenn-ESE-09-15, University of Pennsylvania, September 2019.

[3] Mayank Bansal, Alex Krizhevsky, and Abhijit Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst, 2018.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

[5] Florian Curinga. Autonomous racing using model predictive control, 2017.

[6] Pierre Del Moral. Non linear filtering: Interacting particle solution. *Markov Processes and Related Fields*, 2:555–580, 03 1996.

[7] T. Do, M. Duong, Q. Dang, and M. Le. Real-time self-driving car navigation using deep neural network. In *2018 4th International Conference on Green Technology and Sustainable Development (GTSD)*, pages 7–12, Nov 2018. doi: 10.1109/GTSD.2018.8595590.

[8] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator, 2017.

[9] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. *AAAI/IAAI*, 1999(343-349):2–2, 1999.

[10] Arthur Gelb. *Applied optimal estimation*. MIT press, 1974.

[11] Mikael Henaff, Alfredo Canziani, and Yann LeCun. Model-predictive policy learning with uncertainty regularization for driving in dense traffic. *CoRR*, abs/1901.02705, 2019. URL http://arxiv.org/abs/1901.02705.

[12] Gökhan Karabulut. Mini autonomous car architecture for urban driving scenarios. 9 2019.

[13] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *CoRR*, abs/1105.1186, 2011. URL http://arxiv.org/abs/1105.1186.

[14] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

[15] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[16] Richard Liaw, Sanjay Krishnan, Animesh Garg, Daniel Crankshaw, Joseph E. Gonzalez, and Ken Goldberg. Composing meta-policies for autonomous driving using hierarchical deep reinforcement learning. *CoRR*, abs/1711.01503, 2017. URL http://arxiv.org/abs/1711.01503.

[17] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Aoc: Alpha zero in continuous action space, 2018.

[18] Anders Nordmark and Oliver Sundell. Tactical decision-making for highway driving. 2018.

[19] Defense Advanced Research Project Agency Information Processing Technology Office. Autonomous off-road vehicle control using end-to-end learning, 2004. URL http://net-scale.com/doc/net-scale-dave-report.pdf.

[20] Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, and Marko Bertogna. F1/10: an open-source autonomous cyber-physical platform. *CoRR*, abs/1901.08567, 2019. URL http://arxiv.org/abs/1901.08567.

[21] Xinlei Pan, Yurong You, Ziyan Wang, and Cewu Lu. Virtual to real reinforcement learning for autonomous driving, 2017.

[22] Dean Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D.S. Touretzky, editor, *Proceedings of Advances in Neural Information Processing Systems 1*. Morgan Kaufmann, January 1989.

[23] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap,

and David Silver. Mastering atari, go, chess and shogi by planning with a learned model, 2019.

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[25] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge: Research articles. *J. Robot. Syst.*, 23(9):661–692, September 2006. ISSN 0741-2223. doi: 10.1002/rob.v23:9. URL `http://dx.doi.org/10.1002/rob.v23:9`.

[26] C. Urmson and W. ". Whittaker. Self-driving cars and the urban challenge. *IEEE Intelligent Systems*, 23(2):66–68, March 2008. ISSN 1941-1294. doi: 10.1109/MIS.2008.34.

[27] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019.

[28] Corey H. Walsh and Sertac Karaman. CDDT: fast approximate 2d ray casting for accelerated localization. *CoRR*, abs/1705.01167, 2017. URL `http://arxiv.org/abs/1705.01167`.

[29] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6), 2000.

[30] Timothy Yee, Viliam Lisy, and Michael Bowling. Monte carlo tree search in continuous action spaces with execution uncertainty. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 690–696. AAAI Press, 2016. ISBN 9781577357704.

[31] William "red Whittaker, Dave Ferguson, and Michael Darms. Boss and the urban challenge.