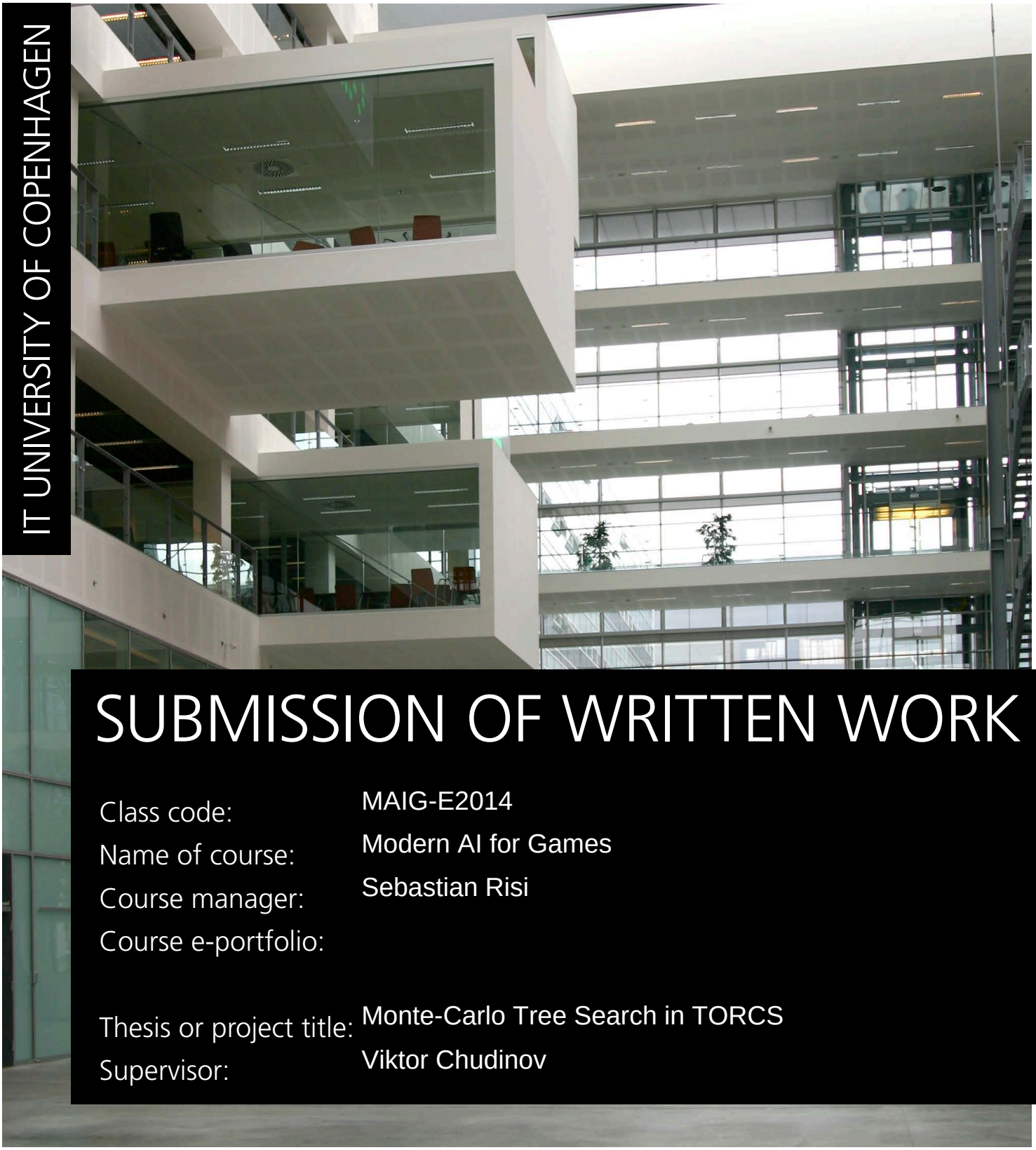
IT UNIVERSITY OF COPENHAGEN



# SUBMISSION OF WRITTEN WORK

| | |
|---|---|
| Class code: | MAIG-E2014 |
| Name of course: | Modern AI for Games |
| Course manager: | Sebastian Risi |
| Course e-portfolio: | |
| Thesis or project title: | Monte-Carlo Tree Search in TORCS |
| Supervisor: | Viktor Chudinov |

| | Full Name: | Birthdate (dd/mm-yyyy): | E-mail: |
|---|---|---|---|
| 1. | Jacob Fischer | 10/06-1991 | jaco@itu.dk |
| 2. | Nikolaj Korsbæk Falsted | 07/04-1991 | nifa@itu.dk |
| 3. | Mathias Beenfeldt Vielwerth | 27/02-1990 | mvie@itu.dk |
| 4. | | | @itu.dk |
| 5. | | | @itu.dk |
| 6. | | | @itu.dk |
| 7. | | | @itu.dk |

# MAIG Group Project
# Monte-Carlo Tree Search in TORCS

Jacob Fischer (jaco@itu.dk), Nikolaj Falsted (nifa@itu.dk) & Mathias Vielwerth (mvie@itu.dk)

***Abstract*— Tree search methods are rarely applied in the effort to control racing cars with artifical intelligence, as such an approach first requires the development of an efficient and fairly accurate forward model. This paper details our approach to this challenge, where we have applied the Monte-Carlo Tree Search method to the open-source racing simulator TORCS.**

**A functioning and efficient forward model was developed, consisting of a number of sub-tasks capable of constructing it, transforming to it and simulating in it. This framework allows a tree-search based racing controller to make use of the Monte-Carlo method for best-first search decision-making in a discretized action space, where an action is represented by pressure on the gas pedal and rotation of the steering wheel.**

**We conclude that it is possible to implement an able racing controller based on this method, which generalizes to most road tracks as long as a warm-up period is provided. Deficiencies of the forward model, however, prevent it from competing on certain tracks.**

## I. Introduction

TORCS is an open-source racing simulation game, with an API for programmatically controlling the racing cars [1]. This makes it an ideal test bed for racing AIs. Competitions are held to pin these AI against each other, with a deal of academic research having been produced in the perpetual search for better racing AIs.

For official competitions in TORCS, the controller for a car is typically written in C/C++. Unofficial competitions do however take place, and these may use their own subset of the framework that TORCS exposes. The Simulated Car Racing competition has been arranged for the past couple of years and uses a TORCS car controller that implements a server [2]. Connecting to this server allows other clients written in any language to send input to the car and receive a state of the world. The competition provides a Java package containing the nessesary client connections to send and receive data from the TORCS controller server. We call this the Java Client API.

Monte-Carlo Tree Search is a best-first search method that builds a search tree iteratively, and bases its decision-making in statistical analysis of simulated play-outs [3]. It has received significant attention for its ability to make "educated guesses" in situations where the search space is too large for exhaustive search methods to be applicable.

In order to apply any tree search method to an AI, two things are needed: a forward model capable of predicting future game states based on what actions are taken, and an evaluation function to probe the desirability of a given game state. In the Java client API, we only have access to data that is relative to the car's position on the racing track at any given time step, which is not sufficient for simulating future time steps, requiring static knowledge of the track's shape. Fortunately, the rules of the competition [2] allow for a warm-up period before the race, which we utilize to record the track in advance.

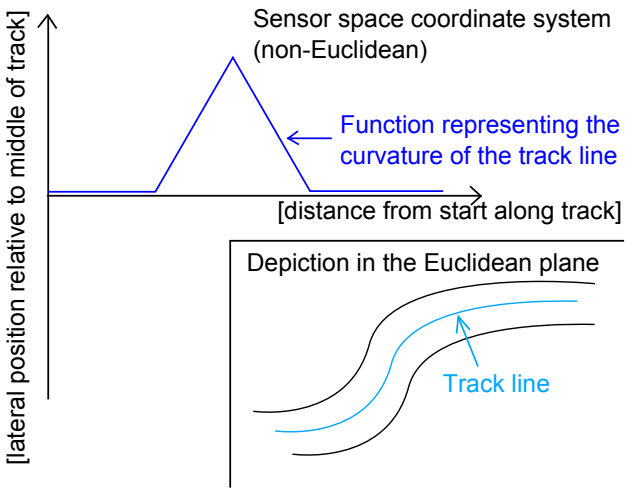



Fig. 1. Sensor space vs. Euclidean space. $y = 0$ represents the middle of the track. In the sensor space this line is straight despite the track line's curvature in Euclidean space. A straight line through a turn in the Euclidean plane will appear curved in the sensor space and vice versa.

Furthermore, the sensor state space mapped out by this data is not Euclidean [10], making it less ideal for applying a forward model intended to approximate a car racing physics engine. This notion is illustrated in figure 1. Hence, we map the recorded data of the track to a 2-dimensional Euclidean space, and a vector based forward model implemented by deference to the laws of classical physics is used to simulate play-outs.

Section II will explain our motive for this project, and section III will give a brief overview of the technical pre-conditions provided by TORCS and the Java Client API.

Section IV will examine how Monte-Carlo Tree Search can be used to control a car in TORCS, assuming an accurate forward model is given. Then, we will explain our approach to providing such a forward model, focusing on the three essential parts of this endeavor (section V):

- Recording the track during warm-up and mapping it to a Euclidean space
- Transforming any given state between the sensor space and the Euclidean space (and back)
- Simulating play-outs with a vector based forward model representing a simplification of the internal engine

## II. Background

The annual scientific competitions [4][5] held by IEEE offer a unique challenge over other competitions held in TORCS, as they limit the information the controller receives about the track to a range of sensors relative to the car, rather than providing absolute information about the game world. Having no absolute track data makes planning algorithms hard to implement, as was also evident in their absence from both the 2008 [4] and 2009 [5] competitions.

In racing, a race line [9] is a path along the track that a driver may follow to complete it. Finding the optimal race line in TORCS can be attained by applying a genetic algorithm to data read from the internal track files [6]. This is used to great success, beating the allegedly best controller "Simplix" [17]. The IEEE competitions do not give access to the track files, however, making this method difficult to apply. Creating a model of the track strictly from sensor data has been done before [7], where it is argued that having the entire track available would allow for planning algorithms to work. Research into modelling a forward function has also been done [8], but here its application is directly in the sensor space and not combined with a track model, limiting its use with regards to planning.

Planning algorithms in general, but tree search methods in particular seem very under-represented in the competitions, which are mainly dominated by neural networks and genetic algorithms. We feel there is an apparent niche here to explore, where a tree search in the space of possible paths through the track represents a different approach to approximating the optimal raceline. Inspired by the success of previous work, we will attempt to derive our own method of modelling the track and forward-stepping the game world, for use in this mostly unexplored territory of tree search in a racing game.

## III. Game Mechanics

In the Java Client API for controlling a car in TORCS, an interface is exposed that has a number of sensors defined, providing data from the game world at every tick [2]. Specifically, they contain information about how far along the track line the car is, how much off it is from the middle of the track, what its angle to the track is, and its velocity in each of the three dimensions (longitudinally, laterally and vertically). For simplicity, we do not concern ourselves with the vertical axis.

The other part of the sensor space includes a set of "range finders", i.e. a number of sensors arrayed in front of the car at different angles, providing metric data about the distances to the edge of the track. We use this data exclusively for building our track model.

An action in the Java client API, returned to the server at the end of every tick, allows the control of a number of factors, but we have limited these to acceleration, braking and steering. Transmission is controlled by a deterministic algorithm based on RPM, and acceleration/braking is further reduced to a single value *gas* for simplicity, with a negative sign representing brake.

## IV. Monte-Carlo Method

### A. Tree Search Car Controller

The *tick* or time step in the competition which provides the Java Client API is a mere 20 milliseconds long [2]. This makes sure the car is updated with a relevant view of the game world fairly often, but it also puts a limit on the runtime of the decision-making algorithm.

Monte-Carlo Tree Search is not exhaustive, and can be polled for an answer after any number of iterations. The longer it gets to run, however, the more likely it is to converge on an optimal solution. For this reason, we have designed our tree search controller to cache its search between ticks, making its decisions within an extended time interval. The car controller will record the current state at time $t$, and apply the forward model once to obtain a state at time $t + t_{step}$, where $t_{step}$ is the configurable decision time step. While driving towards this state in-game by repeatedly applying the best decision from the previous search, the controller will utilize its time budget searching for a decision from $t+t_{step}$.

In addition, utilizing tree search is intended as a mechanism for planning ahead, so it is natural that our approach involves a longer period of time between the steps at which new decisions are considered. In other words, a good decision should not have to be reconsidered immediately.

### B. Action Space

We define an action as the combination of two values: one representing gas or brake pressure, and one describing steering. Both are continuous values, and so are discretized in the interest of having as small a branching factor as possible. The average number of actions available from all states equals the average branching factor of the search tree. A smaller branching factor is always preferable, as long as the available actions allow for optimal play. Deciding which actions are needed for optimal play is not trivial, and is one of the many factors that can be tweaked (see section VII, table III) to increase performance.

We discretize both gas pressure and steering into intervals and allow only selection of neighbouring values to the previous action. E.g. with a current gas pressure of 0.4 and an interval of 0.2, it will only be possible to take actions with a gas pressure of either 0.2, 0.4 or 0.6. This method of filtering allows for a finer granularity, while still having a low branching factor, but at the cost of being able to make sudden changes in action. In that sense, having only neighbouring actions available seems very natural and also serves as a way of reducing skidding, which is currently not modelled in our forward function. We also do not allow braking and steering at the same time, as this almost always leads to skidding.

### C. Evaluation Function

Equation (1) displays the formula we use for evaluating the desirability of a given state. Our state space does not contain successful terminal states, since our benchmark (see section VI) is based on getting as far as possible within an unlimited

number of laps. Unsuccessful terminal states (denoted $\perp$) occur when the car is outside the boundaries of the track.

Our aim was for values to fall within the range $[0; 1]$. A heuristic that normalizes the distance travelled along the track line by the total length of the track is not ideal, since this yields values that grow increasingly larger as the game progresses. The $C_P$ constant of the UCT formula [3], which weighs exploration vs. exploitation in the Monte-Carlo tree search, has to be configured based on the interval of the value range. Thus, the formula in equation (1) yields a value relative to the *local* outset of the search. 0 means the car has been motionless for the full duration of the time difference $t$ between the local outset and the current state. 1 means the car has accelerated as much as possible, following the track line perfectly for the full duration. The intention is for the heuristic to be admissible, i.e. never overestimating the theoretical maximum, which is bounded by the maximum acceleration $a_{max}$. Since the distance is measured along the middle of the track, however, and not along the optimal race line [9], we cannot make this claim.

$$k = \frac{d - d_0}{\frac{1}{2}a_{max}t^2 + v_0 t} \qquad (1)$$
$$k_\perp = k^2 * P$$

where $d$ is the distance along the track line in the given state, $d_0$ is the distance along the track in the state at the root of the search tree, $v_0$ is the velocity at the root, $t$ is the time difference, $k$ is the value in a non-terminal state and $k_\perp$ is the value in a terminal state. $P$ is a configurable constant between 0 and 1 (the penalty factor). The bottom of the value fraction is the equation of motion [15].

Terminal states are penalized by being squared. Since values themselves fall between 0 and 1 (ideally), this penalizes high-valued terminal states less than low-valued ones. For configurability, this is further modified by the paremeter $P$.

## V. Simulation Framework

### A. Recording the Track

Gathering of the track points is done by using the sensors of the car. Even though these are generally relative to the car itself, we do get information about how far along the track it is. We can use this information later to map the points into a Euclidian coordinate system. The most essential part of the gathering step is to find the curvature of the track in a single point relative to a global origin. In order to extract this data from the track, the array of sensors of the car is configured to have three sensors pointing slightly ahead of the car on both sides. One points in a 90 degree angle to the side of the car, presenting the distance to the edge of the track. The next one degree sharper ahead of the car, and the last one degree ahead of that. This gives us three points on the side of the track, and with these we can create a circle [11]. With a circle defined, we know that its curvature equals the inverse of its radius [12]. This, at the same time, describes the curvature of either the interior or exterior side of the track, as illustrated in figure 2. We need both sides

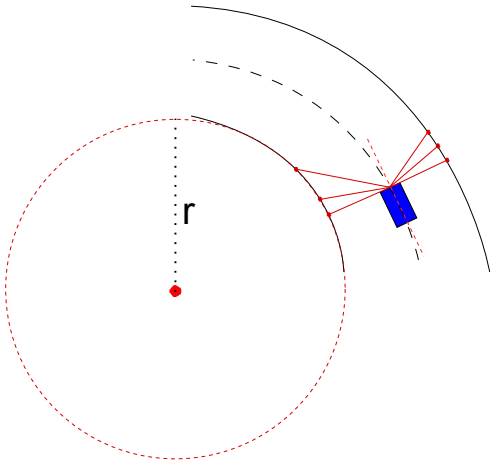



Fig. 2. Curvature of the track line extrapolated from three points measured in the sensor space.

to find the middle of the track. As all tracks are based on mathematical formulas (see appendix I for an example), we can assume that the average curvature of the two sides will equal the curvature of the middle of the track.

In practice, this yields results that are off by a slight error margin for every track. The data is only as good as the readings, and keeping the car running at a pace that will not get it disqualified by the game (and thus instantly removed from the track), while getting steady readings is not always possible. As such, we do minor corrections to the data after recording the track. The biggest change is making sure that the collective angles of all the points sum up to 360 degrees, as all tracks in TORCS begin and end in the same point.

### B. Euclidean Representation

*1) Construction:* The track data from the warm-up period contains a set of track points $D$ in the sensor space, which are all assumed to fall on the track line. A single point $d_i \in D$ has a payload of 4 values:

- its distance from the start line $|d_0 d_i|$ (equivalent to the arc length of the track line segment between $d_0$ and $d_i$)
- the angle $d_i \angle d_{i+1}$ between the tangent to the track at $d_i$ and at the following point $d_{i+1}$
- the width of the road $w^+(d_i)$ in the positive direction
- the width of the road $w^-(d_i)$ in the negative direction

Using this data, it is possible to map the track to a representation in Euclidean space. This approach involves mapping $d_0$ to the origin $(0, 0)$, and then displacing every following Euclidean point $p_i$ by the angle and distance from $d_i$ to $d_{i+1}$, thus obtaining the coordinates for $p_{i+1}$.

This workflow is illustrated in algorithm 1, where the track model is constructed by assigning to the functions $D(x)$ and $P(d_x)$ (lines 7 and 8), where $D(x) = d_x \Rightarrow x = |d_0 d_x|$ and $P(d_x)$ yields the Euclidean point $p$ corresponding to the track point $d_x$.

*2) Transformation:* We wish to distinguish between states in the sensor space and those in the Euclidean space. We

**Algorithm 1** Track Model Construction

$p \leftarrow (0, 0)$
**for all** $d_0..d_{n-1} \in D$ **do**
    $l \leftarrow |d_i d_{i+1}|$
    $\alpha \leftarrow d_0 \angle d_{i+1}$
    $\vec{v} \leftarrow \begin{pmatrix} l \cos\alpha \\ l \sin\alpha \end{pmatrix}$
    $p \leftarrow p + \vec{v}$
    $D(|d_0 d_i|) \leftarrow d_i$
    $P(d_i) \leftarrow p$
**end for**

define a state $\sigma$ as being represented in the sensory model, and a state $s$ as being represented in the vector model. An action $a$ can be applied to either representation in order to advance the game.[1]

A state $\sigma$ represents a subset of data from the sensory data model from [2], named as follows:

- $\delta$ – distance from start along the track line (m)
- $\tau$ – normalized distance to the closest point on the track line ($[-1; 1]$)
- $\theta$ – angle between the yaw of the car and the gradient of the track line at the closest point (radians)
- $\gamma$ – speed of the car in the direction of its yaw (km/h)

Note the absence of lateral velocity. Between actions, we always consider states to be quiescent, having no acceleration and no lateral velocity.[2]

A state $s$ is represented by the following two components:

- the position of the car, given as a point $p = (x, y)$
- the velocity of the car, given as a vector $\vec{v}$

A state $\sigma$ and a state $s$ is related by the functions in equation (2), allowing us to transform a representation in the sensory model to one in the vector model, and vice versa.

$$\begin{aligned} T_v(\sigma) &= s \\ T_s(s) &= \sigma \end{aligned} \tag{2}$$

Theoretically, reflexivity (3) is intended to hold for the above definitions. Note, however, that our internal model can only approximate this relation, so there is a slight loss of accuracy when performing the transformations in practice. See section V-D for measurements of this error.

$$\begin{aligned} T_v(T_s(s)) &\approx s \\ T_s(T_v(\sigma)) &\approx \sigma \end{aligned} \tag{3}$$

Algorithms 2 and 3 display in pseudo-code the implementations of these functions, where the symbols $\delta, \tau, \theta, \gamma$ for states $\sigma$ and $p, \vec{v}$ for states $s$ are used as defined above.

[1]Note that applying an action $a$ to a state $\sigma$ is only possible in real-time by actually taking that action in the game. Simulations require the use of the vector space representation and forward model.

[2]This allows us to keep the transformation logic strictly concerned with states, and the forward model strictly concerned with the application of actions to states. Acceleration and lateral velocity is related to gas and steering, both components of actions, the exclusive domain of the forward model.

Thus, lines 7 and 8 of algorithm 2 assign to a new state $s$, where lines 2, 3, 4, 8 and 10 of algorithm 3 assign to a new state $\sigma$.

At line 7 of algorithm 3, the vector determinant is used to find the orientation of $p$ relative to the track line (left- or right-hand side). This is analogous to using the cross-product in 3-dimensional space [13] for the same purpose.

**Algorithm 2** Transformation $T_v$

$\alpha \leftarrow D(0) \angle D(\delta)$
**if** $\tau > 0$ **then**
    $t \leftarrow \tau * w^+(D(\delta))$
**else**
    $t \leftarrow \tau * w^-(D(\delta))$
**end if**
$p \leftarrow P(D(\delta)) + \begin{pmatrix} t \cos\alpha \\ t \sin\alpha \end{pmatrix}^\perp$ ▷ Perpendicular vector
$\vec{v} \leftarrow \begin{pmatrix} \gamma/3.6 \\ 0 \end{pmatrix} + \alpha + \theta$ ▷ Conversion to m/s

**Algorithm 3** Transformation $T_s$

$d_x \leftarrow P^-(p)$ ▷ Inverse of function P (closest track point to $p$)
$\delta \leftarrow |d_0 d_x|$
$\theta \leftarrow \angle\vec{v} - d_0 \angle d_x$
$\gamma \leftarrow |\vec{v}| * 3.6$ ▷ Conversion to km/h
$m \leftarrow P(d_x)$
$n \leftarrow P(d_{x+1})$
**if** $det(\vec{mn}, \vec{mp}) > 0$ **then** ▷ Vector determinant
    $\tau \leftarrow |mp|/w^+(d_x)$
**else**
    $\tau \leftarrow -|mp|/w^-(d_x)$
**end if**

### C. Forward Model

Equation (4) displays the forward step function, which takes a state $s$ in the vector model and applies an action $a$ to it, yielding a state $s'$, which represents $s$ projected a fixed time interval into the future, with the car having held the same $a_{gas}$ and $a_{steering}$ for the duration of the interval.

$$F(s, a) = s' \tag{4}$$

The implementation of the forward step function relies on assumptions about the relationship between pressure on the gas pedal and acceleration, and between rotation of the steering wheel and angular velocity. We have approximated these relationships through experimentation and regression analysis on recorded sets of data.

*1) Gas:* The relationship between pressure on the gas pedal and acceleration of the car is dependent on various physical properties of the car and thus data from the given car is needed in order to construct a model of the relationship. To gather this data, a track with a long straight stretch is needed for high speeds. A simple controller, that keeps constant gas

pressure while changing gears at high RPM, is driven down the stretch while recording data about its speed, gear and acceleration. At the end of the stretch the controller resets itself and runs again with a different gas pressure.

A custom script for RapidMiner [18] was used to do regression analysis on the data, to produce an equation for each interval of gear and gas, totaling 42 equations with 6 gears and 10 different gas intervals. Higher gear levels do not have data with low throttle as this makes it impossible to reach those gears. The equations are indexed after gas and gear so that they can be accessed by the controller. If the controller requests an equation that does not exist, an equation with a lower gas pressure is used instead. As data was only collected for accelerating speeds, any form of deceleration will be inaccurate. How this affects the controller is discussed in section VII.

*2) Steering:* To model the relationship between velocity, steering and the car's future position in Euclidean space, it is convenient to apply the concept of *angular velocity* [14][16]. Through experimentation, we found that the steering property of an action in TORCS, together with the car's current speed, is directly related to the radius $r$ of the circle around which the car will rotate (drawing it to the side as it travels forward). The steering is inversely proportional to the radius (the more steering, the smaller the circle along which the rotation happens), while the speed is directly proportional (the higher the speed, the bigger the circle). Equation (5) displays our radius approximation function $R(a_s, \bar{v})$. The values for the formula have been assigned by regression analysis on experimental data.

$$R(a_s, \bar{v}) = 176.8927 * 0.000272^{|a_s|} * 1.0099^{\bar{v}} \quad (5)$$

where $a_s$ is the steering property of the action $a$ and $\bar{v}$ is the average velocity of the car in the given forward step.

Equation (6) displays the formula we use to determine angular velocity, where $r$ is the radius, $\bar{v}$ is the velocity in m/s, $\omega$ is the angular velocity (radians/s) and $\theta$ is the angular distance travelled (radians) around a circle of size $r$ in $t$ seconds, $t$ being the time interval of the forward step.

$$\begin{aligned} r &= R(a_s, \bar{v}) \\ \omega &= \bar{v}/r \\ \theta &= \omega * t \end{aligned} \quad (6)$$

In practice, steering is limited by other factors such as road grip, causing the car to skid if it attempts sharp turns at high velocities. This is a shortcoming of our model.

### D. Model Accuracy and Performance

In order to properly review the performance of our car controller, it is important first to establish the impact of our simulation framework on the accuracy of the controller's decision-making. As stated in section V-B.2, the transformations between states in the sensor model and vector model are off by a slight margin of error. We have measured this error, as shown in table I. The second column displays the average discrepancy between the input state $\sigma$ and output state $\sigma'$ for the given sensor after the transformation $T_s(T_v(\sigma)) = \sigma'$, measured every tick for 5 seconds, totalling 250 measurements. During our tree search, states are not transformed back and forth multiple times, so a discrepancy of just 5 centimeters is acceptable, and unlikely to be the cause of suboptimal benchmarks of the controller as a whole.

TABLE I
AVERAGE TRANSFORMATION ERROR

| Sensor | Avg. discrepancy |
|---|---|
| Distance from start | 0.053 m |
| Track position | 0.027 m |
| Angle to track | $2.2 * 10^{-5}$ radians |

TABLE II
SIMULATION FRAMEWORK PERFORMANCE IMPACT

| Measurement | No. of iterations | Fraction |
|---|---|---|
| Isolated tree search | 1937.7 | 100% |
| Only transformation | 1413.8 | 73% |
| Only forward step | 1242.3 | 64% |
| Full framework | 842.2 | 43% |

Table II gives an idea of how much of an impact our simulation framework has on the runtime performance of our tree search algorithm. It compares the number of iterations possible with a timeout of 10 milliseconds on an isolated tree search with no simulation framework being invoked, to the same benchmark with transformations and/or forward stepping being applied in the search. It can be seen that having the full framework present cuts the number of iterations to 43% compared to its absence.

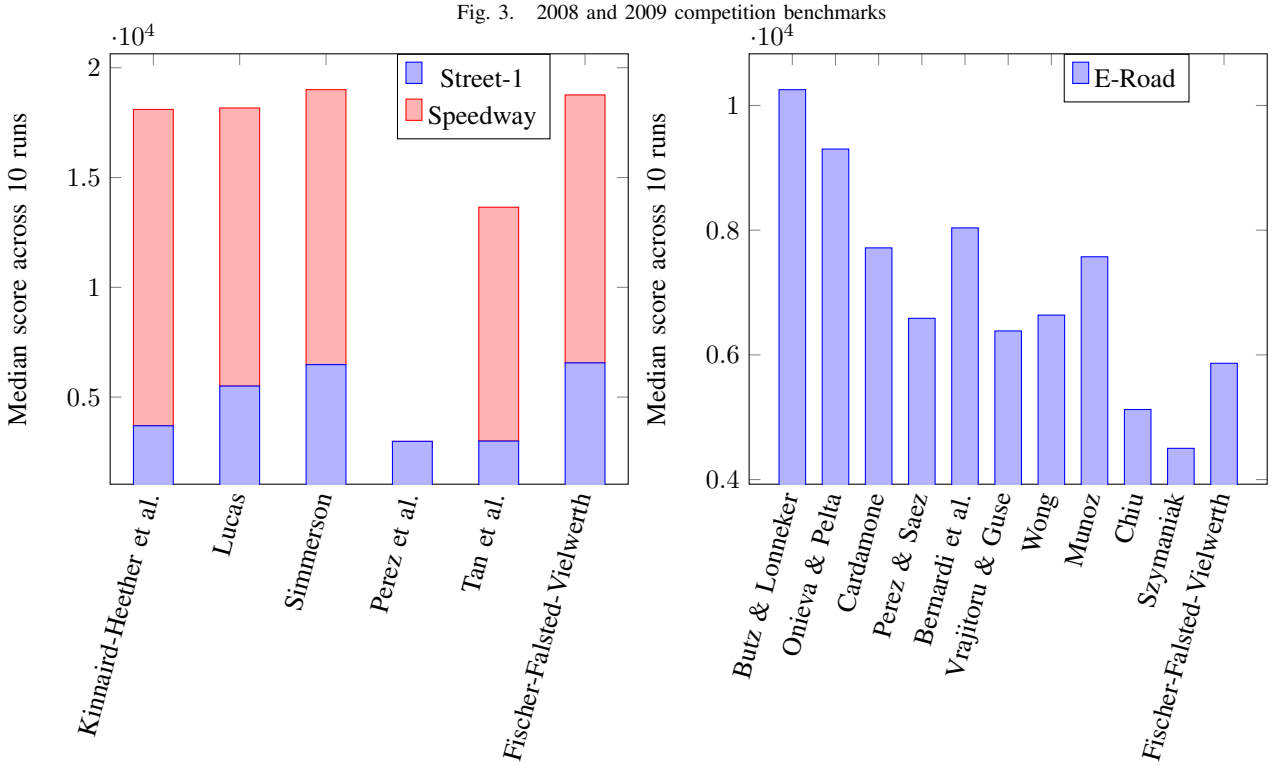
## VI. RESULTS

In this paper, we are comparing ourselves to the competitions held in 2008 [4] and 2009 [5]. Figure 3 displays how our own controller, labelled Fischer-Falsted-Vielwerth, compares to the results from those years. For the competition, a set of three tracks is selected and each controller is run ten times on each track for 10,000 game ticks. The median of the distance raced is then used as a score. We were unfortunately unable to use results from later competitions, as they are run on generated tracks rather than the default TORCS tracks.

The three selected tracks in the 2008 competition consist of one hill track, one tarmac track, and one oval track. For the 2009 competition, the tracks consist of one hill track, one dirt track and one tarmac track. As our forward model is based on data from tarmac tracks, we are unable to compete on dirt and hill tracks. Our scores were consistently lowest on those tracks, so we have omitted them from our charts.

Compared to the 2008 controllers, we make first place on Street-1 and hit the top three on Speedway. The 2009 controllers drive significantly better, so we only achieve scores comparable to the bottom three controllers.

Fig. 3. 2008 and 2009 competition benchmarks




## VII. Discussion

Our controller competes well on tarmac roads, and by fine-tuning the configurable constants we saw an increase in our controller's performance on the benchmarks. The driving behaviour observed align in some cases with human behavior, such as accelerating on straight sections and braking before turns, compared to earlier in the project, where driving speed was generally limited by the lowest speed possible on the track.

The three tracks chosen for the competitions usually contain at least one track that is either a dirt track or a hill track. A dirt track has low traction, which causes high speed on acceleration to spin or skid the car. A hill track has steep climbs and descends, making the car accelerate slower or faster than on a flat track. Both these types changes the basic premises for how the car handles on the track. Since this is not part of the calculations in our forward model, we are at this point unable to race these tracks at competitive speeds. In the future, changes to the forward model together with tests of traction during the warm-up is likely to let us achieve results on dirt and hill tracks, similar to those obtained on tarmac. A further improvement would be to collect actual data about how the car brakes, instead of the hand tuned solution that describes deceleration at the moment.

As we have limited the amount of actions to not contain heavy braking and steering at the same time, positioning of the car before a corner plays an important role. Having the car on the wrong side of the track before a corner forces the car to change position before being able to hit the brake. This can lead to anything, from not taking the corner optimally to not taking the corner at all. An example of this is how the car handles turn 11 on the Street 1 track. Turn 11 is a very sharp hairpin. The speed of the car needs to be very low, but the turn ends the longest straight on the track. The time it takes to brake is therefore high, as it enters the braking zone at high speed. If the car is not positioned well when the search reaches the upcoming turn, it needs to adjust its position to make a proper entry. The time it takes to make this adjustment makes it unable to complete the deceleration in time, and the car either comes to a complete halt or leaves the road. We attribute this behavior to the relatively low depth of the search tree. With a greater depth, the controller might realize that the turn is coming up sooner, thus repositioning the car before entering the brake zone.

Since the focus of this project has been to implement and test Monte-Carlo Tree Search, we decided not to augment the controller with special case handling such as ABS (Anti-lock braking system), ESP (Electronic stability program) and off track recovery. These augmentations would likely have increased our scores, but we felt it would pollute the results of the actual tree search, overshadowing potential faults.

TABLE III

CONFIGURABLE CONSTANTS

| General | |
|---|---|
| Decision time step $[s]$ | 0.25 |
| Timeout $[ms]$ | 15 |
| Gear limit[3] | 2/6 |
| **Evaluation** | |
| Off course boundary | 5.3 |
| Off track boundary | 0.7 |
| Penalty factor | 0.4 |
| Max. acceleration $[m/s^2]$ | 10 |
| **Monte-Carlo** | |
| $C_P$ value | 0.7 |
| Simulation depth limit | 3 |
| **Action Space Definition** | |
| Gas interval size | 0.2 |
| Steering interval size | 0.1 |
| Gas discretization factor | 5 |
| Steering discretization factor | 10 |

## VIII. CONCLUSIONS

Throughout this paper, we have presented our work with Monte-Carlo Tree Search in the TORCS racing game. We have successfully discretized the action space of the car and attained better driving capabilities by removing actions with undesirable outcomes. The Monte-Carlo Tree Search is tuned to maximize the distance raced within the allotted time. We can successfully build a track model and perform simulations in it, but the forward model used to simulate these steps was found, under certain circumstances, to have deficiencies compared to what actually plays out. The implementation fares good compared to results from 2008 and 2009, but fails when driving on dirt or hill tracks, as the forward model is optimized for tarmac roads. By looking at the scores compared to the other cars, we find that Monte-Carlo Tree Search presents a viable option to control a racing car at competetive speeds.

## REFERENCES


[1] B. Wymann, "TORCS, The Open Racing Car Simulator". http://torcs.sourceforge.net/

[2] D. Loiacono, L. Cardamone, P. L. Lanzi, "Simulated Car Racing Championship Competition Software Manual", April 2013.

[3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods", *IEEE Transactions on Computational Intelligence and AI in Games*, 2012, pp. 1–49.

[4] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds, and Y. Saez, "The WCCI 2008 Simulated Car Racing Competition", *IEEE Symposium on Computational Intelligence and Games*, 2008, pp. 119–126.

[5] D. Loiacono, P. Lanzi, J. Togelius, E. Onieva, D. Pelta, M. Butz, T. Lnneker, L. Cardamone, D. Perez, Y. Sez, M. Preuss, and J. Quadflieg, "The 2009 Simulated Car Racing Championship", *IEEE Transactions on Computational Intelligence and AI in Games*, 2009, pp. 131–147.


[3]Gear limit is set to 2 on the Street-1 track.


[6] L. Cardamone, D. Loiacono, P. Lanzi, and A. Bardelli, "Searching for the optimal racing line using genetic algorithms", *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 2010, pp. 388–394.

[7] J. Quadflieg, M. Preuss, O. Kramer, and G. Rudolph, "Learning the track and planning ahead in a car racing controller", *IEEE Symposium on Computational Intelligence and Games*, 2010, pp. 395–402.

[8] M. Butz and T. Lonneker, "Optimized Sensory-motor Couplings plus Strategy Extensions for the TORCS Car Racing Challenge", *IEEE Symposium on Computational Intelligence and Games*, 2010, pp. 317–324.

[9] B. Beckman, "The Physics of Racing, Part 5: Introduction to the Racing Line", 1991. http://www.miata.net/sport/Physics/05-Cornering.html

[10] S. Christopher, and E. W. Weisstein, "Euclidean Space", *MathWorld*–A Wolfram Web Resource. http://mathworld.wolfram.com/EuclideanSpace.html

[11] E. W. Weisstein, "Circumcircle", *MathWorld*–A Wolfram Web Resource. http://mathworld.wolfram.com/Circumcircle.html

[12] E. W. Weisstein, "Curvature", *MathWorld*–A Wolfram Web Resource. http://mathworld.wolfram.com/Curvature.html

[13] E. W. Weisstein, "Cross Product", *MathWorld*–A Wolfram Web Resource. http://mathworld.wolfram.com/CrossProduct.html

[14] E. W. Weisstein, "Angular Velocity", *MathWorld*–A Wolfram Web Resource. http://mathworld.wolfram.com/AngularVelocity.html

[15] R. Nave, "Description of Motion", *HyperPhysics*–Department of Physics and Astronomy, Georgia State University. http://hyperphysics.phy-astr.gsu.edu/hbase/mot.html

[16] R. Nave, "Rotational Quantities", *HyperPhysics*–Department of Physics and Astronomy, Georgia State University. http://hyperphysics.phy-astr.gsu.edu/hbase/rotq.html

[17] "TORCS Robot Development — Simplix". http://www.simplix.wdbee-aorp.de/SIMPLIX/SimplixDefault.aspx

[18] "RapidMiner". https://rapidminer.com/

## APPENDIX I
## TRACK DEFINITION EXAMPLE

```
<!--*******************************-->
<!--     Segment 4                 -->
<!--*******************************-->
<section name="curve 20">
  <attstr name="type" val="lft" />
  <attnum name="arc" unit="deg" val="40.0" />
  <attnum name="radius" unit="m" val="90.0" />
  <attnum name="end radius" unit="m" val="115.0" />
  <attnum name="profil steps length" unit="m" val="4.0" />
  <attnum name="z end" unit="m" val="-1.5" />
  <!--Left part of segment-->
  <section name="Left Side">
    <attnum name="start width" unit="m" val="4.0" />
    <attnum name="end width" unit="m" val="2.0" />
    <attstr name="surface" val="tr-road1" />
  </section>
  <section name="Left Border">
    <attnum name="width" unit="m" val="1.0" />
    <attnum name="height" unit="m" val="0.1" />
    <attstr name="surface" val="curb-aa-bw-l" />
    <attstr name="style" val="curb" />
  </section>
  <!--End of left part-->
  <!--Right part of segment-->
  <section name="Right Side">
    <attnum name="start width" unit="m" val="2.0" />
    <attnum name="end width" unit="m" val="2.0" />
    <attstr name="surface" val="tr-road1" />
  </section>
  <section name="Right Border">
    <attnum name="width" unit="m" val="0.0" />
  </section>
  <!--End of right part-->
</section>
```

## APPENDIX II
## TORCS SETUP GUIDE

Download and install TORCS 1.3.4 for Windows.
**https://sourceforge.net/projects/torcs/files/torcs-win32-bin/**

Download the patch files and copy them into the TORCS folder. Overwrite any conflicting files.
**http://sourceforge.net/projects/cig/files/SCR%20Championship/Server%20Windows/2.0/patch.zip/download**

Dowload the controller that acts as a UDP server. Copy the content into the drivers subfolder.
**http://sourceforge.net/projects/cig/files/SCR%20Championship/Client%20Java/2.0/scr-client-java.tgz/download**

Further instructions can be found at
**http://arxiv.org/pdf/1304.1672.pdf**

## APPENDIX III
## VIDEO

A short video showing how the controller handles a track and how it gathers data about the track.
**https://www.youtube.com/watch?v=rKcSIDMPoVg**