Using Reinforcement Learning to Make Optimal Use of Available Power and Improving Overall Speed of a Solar-Powered Boat

R.L. Feldbrugge

August 2010

Master Thesis
Artificial Intelligence
Dept of Artifical Intelligence,
University of Groningen, The Netherlands

Internal supervisors:

•Prof. Dr. L.R.B. Schomaker	(Artificial Intelligence, University of Groningen)
•Dr. M.A. Wiering	(Artificial Intelligence, University of Groningen)

university of groningen / faculty of mathematics and natural sciences / artificial intelligence

Abstract

In preparation of the Frisian Solar Challenge 2010 (the world cup for solar powered boats), the Hanzehogeschool and RuG are designing a hydrofoiling solarboat. Hydrofoils are under water wings, providing lift, capable of raising a boat out of the water. This reduces drag significantly, enabling higher speeds with a decrease of power consumption. However a lot of energy has to be spent in order to get the boat out of the water. These large amounts of energy were previously not available through solar power, making hydrofoils state of the art in solar powered boats. The hydrofoils of the solarboat are retractable, meaning that during the race the solarboat has the ability to switch between states (sailing hullborne or foilborne). Deciding when to switch is a difficult task which has to take into account many different variables. We show how Reinforcement learning can be used to learn an optimal policy in a simulative environment which aims at finishing the race as fast as possible with a limited amount of energy. We use Artificial Neural Networks as function approximators for estimating the value for each action in an arbitrary state. The learned policy will have an advisory role to the pilot of the boat during the race. We set up several experiments, iteratively increasing the complexity of the model. First the partially observable Mountain car problem was modeled, our results show that through the use of artificial neural networks the value can be predicted more accurately than with a traditional tabular approach. This resulted in a significantly better performance than with the standard method. Next we modeled the solar boat in a simple race situation, competing on a linear track without an environment. We show how our algorithm is able to optimize the time required for the solar boat to reach the finish line. Finally we modeled the solar boat within his environment, taking into account position on the track, cornering, sun energy, changing weather and shadows. Two different policies were trained, the first was trained on one single track (Specific Policy Algorithm, SPA), the other was trained on all tracks (Generalizing Policy Algorithm, GPA). There was no significant difference between the performance of the SPA and the GPA. Both algorithms show a gradual decrease in time required to reach the goal, optimizing energy in a situation where energy is limited and unpredictable.

Contents

1 Introduction

Throughout many aspects of Artificial Intelligence, there is a recurring need to automate sequences of decisions. Whether the goal is to construct agents, control systems, machine vision or solve planning problems, many core issues remain the same. When making a decision, different alternatives (actions) have to be evaluated. This evaluation has to take into account the current world situation, and what the world will look like when a chosen action is performed. This requires more than just evaluating the immediate effects of actions, but also the long term effects have to be taken into account. These long term effects are not always as easy to define, especially when the outcome of actions are subject to uncertainty. Sometimes there are actions that can be performed, resulting in poor immediate results, but score much better over the long term. Choosing the optimal action has to make a trade-off between these two situations (the immediate reward and future gains).

The motivation for this research comes from the Frisian Solar Challenge, the world cup for solar powered boats. The race follows the 'Elfstedentocht' which can be translated as the 'Journey of Eleven Cities', an ice speed skating competition held in the province of Friesland (the Netherlands). The tour has a length of 220km and is conducted on canals, rivers and lakes between eleven Frisian cities. The race is divided into six stages. The goal of this race is to complete the race as fast as possible given a limited amount of energy (only a 1Kh battery and 1750W of solar panels are allowed). The Hanze Solar Team won the Frisian Solar Challenge in 2008 in the A-class with a mono hull configuration in a record time of 17 hours 39 minutes and 37 seconds.

In preparation of the Frisian Solar Challenge 2010, the Hanzehogeschool and RuG are designing a hydrofoiling solar boat. Hydrofoils are under water wings, similar to airplane wings. These wings provide lift, capable of raising a boat out of the water. This reduces drag significantly (see figure 1), enabling higher speeds with a decrease in power consumption. However before reducing drag a lot of energy has to be spent in order to get the boat out of the water, these

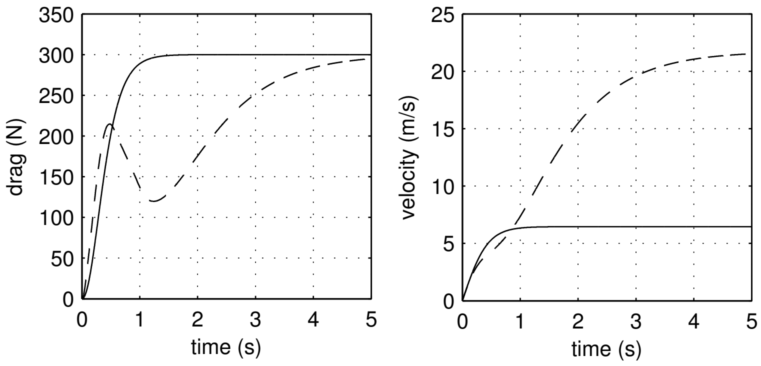


Figure 1: Drag comparison between a normal boat (solid line) and a hydrofoiling boat (dashed line) with 300N Thrust. The hydrofoil will reach a four fold top speed with this thrust level.

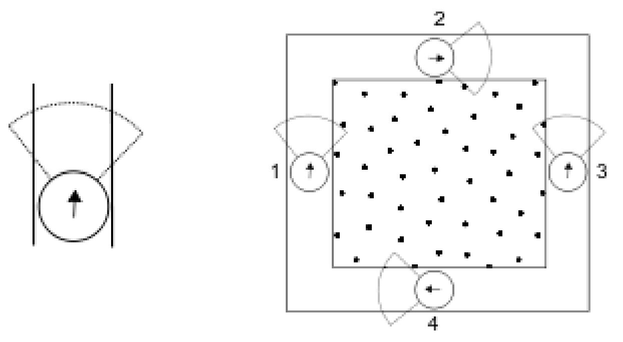
amounts of energy were previously not available through solar power. For this reason hydrofoils can be considered as state of the art in solar powered boats.

The new boat is going to be equipped with one set of retractable hydrofoils, giving it the ability to reduce drag by lifting the boat out of the water when possible. When there is no sufficient energy available, maintaining a high velocity on hydrofoils is impossible. Sailing with the hydrofoils extended beneath the boat at low speeds requires considerably more energy than without hydrofoils (figure 1, the drag of the foilborne boat is higher than the hullborne boat until $t = 0.5s$. The difference is the greatest at $t = 0.4s$ where the drag of the hydrofoil is 200N, against 100N for the hullborne boat). For this reason the choice has been made to make the hydrofoils retractable. This means we can choose during the race to sail at a low speed (saving energy) without the drag of the hydrofoils and at a high speed (spending a lot of energy) with the hydrofoils extended.

The transition from hullborne to hydrofoil requires a lot of energy. Figure 1 shows the total drag of both a hullborne and a foilborne boat, at low speeds the drag of the hydrofoiling boat is higher than the hullborne boat due to the resistance of the foils in the water. When the boat takes off the drag reduces, facilitating a higher overall speed. Other variables also have to be taken into account when making the decision whether or not the hydrofoil is to be deployed. One important variable is water depth, this influences the performance of both the hydrofoil and the mono-hull in a different way [25]. But also the expected amount of energy to be received during the remainder of the race and the current state of charge of the battery are important variables. It might even be advantageous to store energy in the battery when there is the possibility that in the future there is a higher energy demand. These are just a few variables, the number of variables to be taken into account depends on how detailed we want our model to be.

Because we have to deal with a dynamic model, making decisions is hard. One does not know beforehand how to act in the world (when to deploy the hydrofoils, or how fast to go), there is no straightforward solution for this. Being researchers in the field of Artificial Intelligence we saw an opportunity for Machine Learning [26] to solve this difficult problem. Machine learning is a research field that is concerned with the design and development of algorithms that give computers the ability to evolve behaviors based on empirical data (e.g. sensor readings or data from databases). In our case we apply Reinforcement Learning [26]. Reinforcement Learning techniques aim at learning a policy (guidelines of how to act in the environment) that maximizes some notion of a cumulative reward. This reward can be given directly, or with a certain time delay. The design of the reward function is very important because it steers the behavior of the agent towards a certain goal.

Reinforcement learning algorithms [26] attempt to find a policy that maps states of the world to actions which the agent should take in those states. Reinforcement learning problems are typically formulated as finite state Markov Decision Processes (MDPs) [4]. MDPs provide us with a mathematical framework for modeling decision-making in situations where outcomes can be influenced by a decision maker (e.g. an agent/policy), but still suffer from a certain amount of randomness. This means that when an action is taken, given a starting world state, it is not always certain that the next world state will always be reached given the same starting conditions. MDPs assume that the system



(a) Perceptual field of the robot (b) Rectangular world, ambiguous localization

Figure 2: Ambiguous localization for an MDP problem

can always fully observe its environment, in other words: The system always knows where it is and all that it needs to know about itself and its environment at each moment in time. This however is not always the case. One might think that a way of solving this is providing more sensory input, ambiguity in the sensory information however will still produce situations in which the system cannot know in what state it is. This is shown in figure 2, the robot perceives the hallway through his sensors (figure 2a), however when looking at the world in which the robot acts (figure 2b), it cannot discriminate between being in situation 1, 2, 3 or 4 (and even if it could it would not know in which direction it was traveling). The problem of ambiguity is also called partial observability. This cannot be dealt with within the traditional MDP framework. Extending the MDP framework in such a way that it can deal with these kinds of problems results in modeling the problem as a Partially Observable Markov Decision Process (POMDP) [26, 13]. In a POMDP it is assumed that the current state is hidden, instead of mapping states to actions in an MDP, a POMDP maps observations to actions.

An impressive application of using (PO)MDPs is the research done in the Stanford University Autonomous Helicopter Project [1] in which they use PEGASUS (Policy Evaluation of Goodness and Search Using Scenarios) to reduce the computational complexity of their (PO)MDP [18]. PEGASUS uses scenarios (predefined sequences of actions in a simulated world) to estimate the value of a policy. This reduces the computational complexity significantly [18]. It can be described as a policy search algorithm. PEGASUS reduces the problem of policy search in an arbitrary POMDP to one in which all the transitions are deterministic. In [19] the authors show convincingly how PEGASUS can be used to learn the difficult task of autonomous helicopter flight, and even extreme aerobatics.

The decision making process of when to switch from mono-hull to hydrofoil, when to store energy and how fast to go in a solar boat race can be modeled as a POMDP. This problem is not a toy-problem, meaning that solving the POMDP will take a considerable amount of computation time. Building a numerical model of the boat will also give us the ability to improve our design because different types of boat-configurations can be tested.

Our project focuses on exploring reinforcement learning methods that make optimal use of available power and optimizing the overall speed in applications with limited energy.

1.1 Research Question

The global research question could be defined as follows: *Can we use reinforcement learning to make optimal use of available power, optimizing the overall speed of a solar powered boat?*. This research question however is too broad, our research aims at exploring reinforcement learning methods that can be used for these kinds of applications (not just for solar boats). This resulted in a more specific research question: *Which reinforcement learning methods can be used for making optimal use of available power, optimizing the overall speed in application with limited energy?*. This research will explore a new application for machine learning techniques in the field of sustainable energy. A solar powered boat in this case will act as an application with limited energy. It will not be possible to explore all possible reinforcement learning algorithms, therefore we focus on the ones which we think are most likely to be successful.

1.2 Outline

This paper begins with explaining the theoretical background of reinforcement learning [26], different aspects of this research field will be discussed, relevant to our project. We will also explain the (Partially Observable) Markov Decision Processes in greater detail. Next we discuss how we modeled the solar boat, explaining all equations relevant to the physical dynamics of the boat (with and without hydrofoils). Also the modeling of the race itself (weather and track) are discussed in this section. We also discuss how we match our model with data gathered from the boat in the real world. After these sections we move on to the experiments, first the Mountain Car problem [14] is explained. The results from these experiments are used to guide our research, iteratively increasing its complexity. The next step after this subproblem is the problem of the power consumption optimization of a solar powered boat for a small race (for a linear track without taking into account several environmental parameters like weather and events along the way). After this section we reached the final experiment, the power consumption optimization of a solar powered boat (for the final race), followed by an overall discussion and conclusion.

2 Reinforcement Learning Theory

Reinforcement Learning [26] is a field of machine learning [26] in which an agent learns to take actions in an environment in such a way that it maximizes its performance. The performance is measured in terms of rewards that the agent gathers while acting in the world. The aim of a reinforcement learning algorithm is to find a policy that maps states of the world to actions that the agent should take when being in these states.

2.1 Reinforcement Learning

In Reinforcement learning no 'tutor', explaining the agent what it does good or bad, is present. This means that the agent has to interact with his environment through actions. At each point in time, the agent performs an action and observes how this changes his environment. This observation yields costs and/or rewards, the underlying dynamics for this cost function is not known to the agent. The aim of the agent is to discover a policy that provides a mapping from observations to actions that minimizes the long-term cost function. Reinforcement learning problems are often formulated as (Partially Observable) Markov Decision Processes ((PO)MDPs).

2.1.1 Reinforcement Learning vs. Supervised Learning

Supervised learning is a machine learning technique that tries to fit a function to training data. The training data is built up by pairs of input (usually vectors) and desired outputs. The job of the supervised learning algorithm is to learn a mapping from the input to the desired outputs. The ideal mapping has to generalize the training data so that it can correctly classify new data that has not been seen before (and has no label assigned to it). In reinforcement learning no input-output pairs are present, the algorithm only receives feedback from the world about how good the result of an action was but not if it made an error in taking that action. It is also not always the case that the agent receives a reward immediately after taking the action, but there can be a delay (e.g. one particular sequence of actions with a lower reward might lead to a much higher reward in the future).

Looking at the problem of optimizing the energy consumption of the solar boat we cannot define beforehand when we would like the hydrofoils to be deployed or at what speed we want to travel. If we knew this beforehand, this project would not be of any use. Several actions can be taken during the race, the boat will be able to speed up or slow down; but also lower or raise its hydrofoil. These four actions all have different impact on the energy consumption. Slowing down means less energy is wasted, however the finish line will not be reached. One solution might be to increase speed, spending a lot of energy. This sounds like a good action, however the amount of energy is limited. Depleting the battery when there is not enough sun-energy will also hinder the boat from reaching the finish line. Then there are the hydrofoils, making the transition from hullborne to foilborne means spending a lot of energy, before lifting the boat out of the water it has to overcome the added resistance of the foils in the water. This means that when making the transition from hullborne to foilborne the boat has to be able to keep maintaining foilborne for a reasonable amount of time before it actually saves energy. Taking into account external factors (changing weather, shadow areas with low sun radiance, bridges which cannot be taken foilborne and reducing speed in sharp corner) make the system highly dynamic, there is no simple solution to this problem.

The nice thing about reinforcement learning with respect to supervised learning is that no knowledge about what kind of behavior is required is needed, the environment shapes the learned behavior through the gathered rewards (defined by the reward function, discussed in section 2.1.2). Reinforcement learning can be used to estimate the outcome of any arbitrary action. The outcome in this case is a reward, provided to the agent through the environment. If this reward is different from the prediction, the model adjusts itself towards the perceived reward. Learning therefore means that the agent acts in its environment, learning about the feedback that it gets through the reward function and updates itself accordingly.

In the case of the solarboat we could define a reward function that aims at reaching the finish line as fast as possible. We can use this function to learn a policy that maximizes its reward; and therefore minimizes the time required to reach the finish line.

Because we do not know beforehand what sequences of actions we want the system to take, we are unable to define the input-output pairs which make up the training-set for a supervised learning algorithm. This forces us to solve the problem through reinforcement learning.

2.1.2 Designing a Reward Function

The beauty of reinforcement learning is that the agent has to gather rewards through interactions with the world, the downside of this is that how these rewards are defined influences the performance of the system dramatically. Take for example a simple mobile robot with an onboard processor in a small and empty world, which contains only one recharging station. One way to define the reward is the amount of energy that this agent has in his internal battery. Driving around will provide the robot with a negative reward, it costs energy to drive the motors. One solution would be to stay put, not moving at all. However the onboard processor will still use up a small amount of energy, resulting in a small negative reward. The only way for the agent to decide to move towards the recharging station is by designing the reward function in such a way that the tradeoff between not moving at all and moving towards a socket (gathering a lot of negative rewards, but at the socket receiving a large positive reward) will result in the behavior that we desire (note that in this case we have a bias towards what the desired behavior should be, namely reaching the socket and not dying a slow death). One way of implementing this is by assigning a second reward function to the agent that delivers a substantially low reward to dying. This way the agent will learn that after performing no action at all, this will result in a very low total reward. The downside of this approach is that the way in which the negative dying-reward is defined will influence the resulting behavior.

The design of a reward function is challenging, as researchers we do not want to bias our system to what we think the correct behavior should be. However the reward function should not be to 'loose' so that the system might optimize its reward function, but displays behavior that is not desirable. This means that there is a trade off in the design of the reward function. Looking at the solarboat at a glance we would say that the boat has to be as conservative with energy as possible. However minimizing power consumption leads to not taking any actions (like described above). Thinking about this notion showed us that the action power optimization of the boat is not to save energy, but use it in such a way that it is entirely depleted at the end of the race. This way no energy is wasted by not using any. A reward function that encodes this will however also not behave as we would like. It could spend all of its energy at the beginning of the race, reaching the finish line on only the power coming from the solar panels and still receive a high reward even though it might finishes last with this policy. The reward function has to encode both energy and speed. The simplest way of implementing this is by giving a negative reward for each timestep that the agents needs to finish the race. This way if it depletes its energy too soon it will not reach the finish line, resulting in a large negative reward. If it uses its energy wisely it should optimize its speed, which is the goal of our race, reaching the finish line as fast as possible with limited energy. Note that the reward function is programmed (hard coded) by the researcher, the reward function is not learned by any of the Reinforcement Learning algorithms. The question we as researchers ask ourselves (not to be mistaken with the research question) is: How can we initialize/update the reward function so as to induce best possible world utility? This, and the reward function for the solarboat, will more elaborately be discussed in section 5.2.2.

2.1.3 Markov Decision Processes

Markov Decision Processes (MDPs) [4] provide a mathematical framework for modeling decision-making under uncertainty. The MDP model assumes that the next state is solely determined by the current state (the Markov assumption). It also assumes that the state that the model is in is completely observable. This means that the current state has to be completely known at all times.

A Markov Decision Process can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$, where:

- S is a finite set of world states.
- A is a finite set of available actions.
- $T : S \times A \mapsto \Pi(S)$ is the state transition function, a probability distribution over world states for a given world state and agent action defined by the policy Π. $T(s, a, s')$ denotes the probability of ending up in state s', given the current state s and action a. T also represents the Markov assumption, the next state depends solely on the current observed state and action.
- $R : S \times A \mapsto \Re$ is the reward function. This function gives the expected reward gained by taking each action in each state. $R(s, a)$ can be written for the expected reward for taking action a in state s.
- $\gamma \epsilon [0, 1]$ is the discount factor, used in the case of an infinite horizon, weighing the reward function in such a way that rewards in the near future are of a greater influence than rewards later in time.

In many cases not all necessary state information is directly available. Consider a game of cards (e.g. Poker) where some of the cards are known, but other cards are hidden, even as the strategies (policies) of the opponents. The player must develop a so called Belief Function about the state of the world. For this tasks Partially Observable Markov Decision Processes [26] were developed.

2.1.4 Partially Observable Markov Decision Processes

In most real world situations, observing the environment (and taking readings from sensors) includes noise. This makes this kind of environment partially observable, the real world state cannot be perceived with absolute certainty. An MDP cannot deal with these kinds of problems. Partially Observable Markov Decision Processes (POMDPs) extend the MDP framework, giving it the ability to deal with partial observability. With this extension, modeling larger and more interesting classes of problems is possible.

POMDP algorithms are much more computationally intensive than MDP solvers, this is a result of the uncertainty about the true state of the model. This induces a probability distribution over the model states, whereas MDPs only have to deal with a finite set of states. The problem of finding optimal policies for finite-horizon POMDPs has been proven to be PSPACE-complete [20]. It however must be said that running a solved POMDP requires far less computational resources than at the learning stage, and is therefore very quick at run-time.

A finite Partially Observable Markov Decision Process can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O, \gamma \rangle$, in which:

- S is a finite set of world states with an initial state distribution b_0.
- A is a finite set of available actions.
- $T : S \times A \mapsto \Pi(S)$ is the state transition function, a probability distribution over world states for a given world state and agent action defined by the policy Π. $T(s, a, s')$ denotes the probability of ending up in state s', given the current state s and action a. T also represents the Markov assumption, the next state depends solely on the current unobservable state and action.
- $R : S \times A \mapsto \Re$ is the reward function. This function gives the expected reward gained by taking each action in each state. $R(s, a)$ can be written for the expected reward for taking action a in state s.
- Ω is a finite set of observations that can be received from the world.
- $O : S \times A \to \Pi(\Omega)$ is the observation function, a probability distribution over possible observations for a given action and resulting state. $O(s', a, o)$ can be given for the probability of making observation o, given action a and resulting state s'.
- $\gamma \, \epsilon \, [0, 1]$ is the discount factor, used in the case of an infinite horizon, weighing the reward function in such a way that rewards in the near future are of a greater influence than rewards later in time.

It is important to point out that a Markov Decision Process can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$, this means that algorithms used to solve POMDPs can also be used to solve MDPs (but not the other way around).

A policy is a mapping $\pi : S \mapsto A$. The value function of a policy π is also a mapping $V^{\pi} : S \mapsto \Re$. $V^{\pi}(s)$ gives the expected (discounted) sum of rewards for executing π from state s. An optimal policy is known to always exist in the discounted ($\gamma < 1$) case with bounded immediate reward [10]. POMDP policies are often computed using a value function over the belief space. This means that for different belief vectors, different policies can be chosen. Computing policies for every belief vector requires considerably more calculations.

2.1.5 PEGASUS Transformation from Stochastic to Deterministic

Consider a POMDP $M = \langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O, \gamma \rangle$ with initial state s_0 and a class Π of policies $\pi : S \mapsto A$. The goal is to find a policy in Π with a high utility. This stochastic model M can be transformed to a deterministic POMDP $M' = \langle \mathcal{S}', \mathcal{A}, T', R', \Omega, O', \gamma \rangle$ using a deterministic simulative model g for M [18].

The transformation is done as follows. The action space and discount factor remain the same. The state space for M' is represented as a vector $(s, p_1, p_2, \ldots)$ in which $s \,\epsilon\, S$, followed by an infinite sequence of real numbers in [0,1]. Now given $(s, p_1, p_2, \ldots)$ we can use the simulative model g to calculate $s' = g(s, a, p_1)$. The new state becomes $(s', p_2, p_3, \ldots)$, so $p_1$is used to generate s' from the correct distribution. For each policy $\pi \,\epsilon\, \Pi$, there will be an equivalent $\pi' \,\epsilon\, \Pi'$, in which $\pi'(s, p_1, p_2, \ldots) = \pi(s)$. Similar goes for the reward function $R'(s, p_1, p_2, \ldots) = R(s)$. Only observing s instead of $(s, p_1, p_2, \ldots)$ results in the original POMDP M. This means that the search for an optimal policy $\pi' \,\epsilon\, \Pi$ will produce a state sequence that will do equally well in the original POMDP M. This means that searching for an optimal policy in a stochastic POMDP can be reduced to searching for an optimal policy for an equivalent deterministic POMDP, which is much simpler.

The PEGASUS algorithm uses the fact that for computers to simulate stochasticity they have to generate a random number p and then use this value to calculate s' as a deterministic function of the input s, a. PEGASUS exploits this by pre-sampling a limited set of random numbers p in advance and fixing them for each π. The algorithm starts by drawing a sample $\left\{s_0^1, s_0^2, \cdots, s_0^m\right\}$ of m initial states according to an initial state distribution. Fixing the stochastic variables means that $\hat{V}(\pi)$ becomes a deterministic function.

The original idea of PEGASUS is that transforming a stochastic POMDP towards a deterministic equivalent enables the use of scenarios (fixed sequences/feeds of random numbers for the simulation) to compare policies to one another. This is needed when for example evolutionary algorithms are used in which a large number of policies have to be compared to one another. Because of the scenarios, and comparing policies on the same scenarios, less samples from the dynamic (simulated) environment have to be taken, increasing the speed in which the model learns. In our research we do not deal with a large number of policies, therefore the original idea of PEGASUS is abandoned. However we adopt the use of scenarios in our model. Weather scenarios are created, simulating the environmental parameters (e.g. solar radiation). Weather data is dynamic, there is some form of structure present within the recordings. However the underlying mechanisms are not always clear. Creating weather scenarios from recorded data allows us to simulate a lot of different weather types (bypassing the underlying mechanisms) and giving us the ability to estimate a more accurate performance of the policies.

2.2 Temporal Difference Learning

One way of solving reinforcement learning problems is through the use of Temporal Difference (TD) learning [26]. TD learning is a machine learning approach that learns how to predict a certain quantity that depends on future values of a given signal. The name originates from the use of changes/differences in the signal that can be used to predict values in a future time-step. It is a combination of Monte Carlo and dynamic programming (DP) ideas [26]. TD learning samples its environment according to some policy, hence the Monte Carlo resemblance. Dynamic programming is relevant because TD learning approximates its current estimation based on previous estimates, similar to dynamic programming. TD algorithms are often used in reinforcement learning for predicting the value function or total amount of reward expected in the future.

2.2.1 On-Policy versus Off-Policy Methods

In on-policy methods exploration is performed by following a policy that provides a mapping between state-action pairs. This means that the policy that is being optimized is also used as means of exploring the world. One example of this is the ϵ-greedy algorithm [26], this method will choose the action that has the highest estimated reward value, but does this with a certain probability. So in most of the cases ϵ-greedy will take the optimal action, but in some cases it will explore its neighboring states by performing a non-maximal action. Observations that are made after performing an action are used to improve its policy. In off-policy methods the policy that is learned is not the same as the policy that is being followed when exploring the world. For example Exploration-data can be gathered by Monte Carlo [26], this data will be used to learn the final policy. This policy can then be used by selecting the actions completely greedy (ϵ-greedy algorithm with $\epsilon = 0$).

2.2.2 Q-Learning (Off-Policy)

Q-learning was introduced by Watkins [28, 29], being independent of the policy being followed this learning algorithm directly approximates the optimal action value function. Consider a world in which an agent can perform an action a ($a \in A$), which allows the agent to move from state s ($s \in S$) to a new state s'. Q-learning is a reinforcement learning technique that maps a state and action value (s, a) to an estimated Quality (Q) of taking action a in state s following a greedy policy thereafter.

$$Q : S \times A \rightarrow \Re \tag{1}$$

This table will be updated each time s changes and a reward r is provided. This is done by value iteration, it updates the old value according to the new information:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2}$$

α Is the learning rate ($0 < \alpha \leq 1$) and determines the rate at which new information will override the old Q-value. γ Is the discount factor $0 < \gamma \leq 1$ which decreases the estimated Q value for states in the future.

Algorithm 2.1 Q-learning (Off-Policy)

```
Initialize $Q(s, a) = 0$ for all $a$
Repeat forever:
$s \leftarrow InitialState$
for each episode step do
    Select $a$, based on $s$
    Take action $a$, observe $r, s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
    if $s == terminal$ then
        Break
    end if
end for
```

Q-learning will keep on converging to a better solution as long as all state-action pairs continue to be updated. The Q-learning algorithm is shown in algorithm 2.1.

2.2.3 SARSA (On-Policy)

Another way of learning the Quality of an action is by the means of SARSA. The acronym SARSA stands for State-Action-Reward-State-Action and was first introduced by Rummery and Niranjan [23] as a modified Q-learning algorithm. The underlying principles are similar, SARSA however updates $Q_\pi(s, a)$ for the policy (π) that it's actually executing. This makes SARSA an on-policy algorithm. The Q-value update depends on the state of the agent s, the chosen action in that state a, the reward r received when taking action a in state s, the state that the agent will be in (s') after performing action a, and the action a' that the agent will take in state s'. Summarizing this results in a tuple (s, a, r, s', a'). The Q-value will be updated using formula 3.

Algorithm 2.2 SARSA-learning (On-Policy)

```
Initialize $Q(s, a) = 0$ for all $a$
Repeat forever:
$s \leftarrow InitialState$
Select $a$, based on $s$
for each episode step do
    Take action $a$, observe $r, s'$
    Select $a'$ for state $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
    $a \leftarrow a'$
    if $s == terminal$ then
        Break
    end if
end for
```

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)) \tag{3}$$

The Q value is updated through interactions with the environment, updating the policy depends on the taken actions. The Q-value for a state-action pair is not directly updated, but gradually adjusted with learning rate α. As with Q-learning SARSA will also keep on improving its policy to a better solution as long as all state-action pairs continue to be updated. The SARSA-learning algorithm is shown in algorithm 2.2.

2.3 Exploration

Exploring a new environment means that actions have to be taken. Always performing actions with a fixed policy will lead to a solution, however this solution will most likely not be the optimal solution. This means that instead of always taking the optimal action (one type of action selection that is favored in the final model), sometimes choosing a non-optimal action will sample the utility landscape of the environment. Learning from this sampling will guide the algorithm to a better solution, and eventually even to the best solution (given the amount of exploration). Too much exploration does have a downside, it will lengthen the time needed to learn the utility values. It might also lead to the algorithm being 'stuck' in the environment, not reaching the goal (if this is required). There is a tradeoff between selecting the optimal action and performing exploratory actions. There are numerous ways of selecting non-optimal actions. We will discuss two popular selection methods in the following sections.

2.3.1 ε -greedy Selection

One of the most simple ways of introducing exploration is by letting the agent explore its environment in a greedy manner, but instead of always taking the optimal action it selects a non-optimal action with probability ε. This selection rule is called ϵ -greedy Selection [26]. By selecting a non-optimal action the agent explores different state-action pairs and reaches states that have not been seen before, providing the agent with more knowledge about the world. Fixing ε means that throughout the interaction with the world the exploration rate remains the same, when starting however it is essential that the exploration rate is high. However when a model of the environment takes form taking exploratory actions would decrease the total reward of the agent. Decreasing ε after each time the agent interacts with the environment with a certain decrease-rate (dr_ε) will counteract this. This is called an ε - decreasing strategy and can be formulated as follows:

$$\varepsilon_t = \frac{\varepsilon_0}{1 + t * dr_\varepsilon} \tag{4}$$

2.3.2 Boltzmann Exploration

ϵ - Greedy strategies select other actions with equal probability. This means that taking the worst possible action (according to the agent's knowledge of the world) will have the same probability of being chosen as the second best action.

It might be beneficial to more often choose the action that is second best. By increasing the probability of being chosen for actions with a higher quality we arrive at Boltzmann exploration [26]. Boltzmann exploration is not a two-fold process like in ε - greedy where the agent flips a coin and decides whether or not it should perform a random or optimal action, but uses a Boltzmann distribution:

$$
\begin{aligned}
P(a_i) &= \frac{e^{(Q(s_t,a_i)/\tau)}}{\sum_{j=1}^{n} e^{(Q(s_t,a_j(t))/\tau)}} \qquad (5)\\
n &= \text{ number of actions}\\
\tau &= \text{ temperature with } \ (\tau \in \Re, \tau >= 0)
\end{aligned}
$$

The temperature τ controls the amount of exploration. With a higher temperature the amount of exploration is larger than with a lower temperature. The term temperature is still being used because the Boltzmann distribution was originally formulated to explain the crystallization of cooling materials. As with ε - greedy in a new world the agent would like to explore more than when it has gathered some experience. This is realized by decreasing the temperature as the experiment goes on. This can be done in a similar way as with the ε - greedy algorithm (formula 4), with ε replaced by τ.

2.4 Function Approximation (Artificial Neural Networks)

In the real world we, as humans, approximate many functions. One good example is when you get up in the morning you estimate how long it would take you to get to work. This estimation takes into account several factors (eg. weather, traffic, means of transportation).

Function approximation problems can be split into two classes:

- Classification: Discrete output, a given input is classified as belonging to a discrete group. An example of this is for example face identification, object recognition and the classification of handwritten text.

- Regression: Continuous output, the required output is a real value. An example of this is the problem described at the beginning of this paragraph, the estimation of our travel time in which some input parameters have to be mapped towards a real value, namely time.

In theory the differences between regression and classification problems are not large. However a classification problem often has a one binary output parameter for each possible class the input can belong to. The mapping from an input vector towards the binary encoded classification is the task for the function approximator. In regression problems however the output of the function approximator is no binary encoding but a real value, the function approximator will mimic a complex algorithmic function that depends on parameters, defined by the input vector. The strength of a suitable function approximator is that it should be able to approximate the output for any given input-output pairs, after having seen only a limited set of examples. In other words, the function approximator should be able to generalize to an extent that it predicts the outcome of unseen input data correctly.

Q-learning and SARSA are both tabular reinforcement learning algorithms. This means that they map a state, action pair towards the Quality value. This mapping is stored in a table, containing a Q-value for each possible state action pair. The consequence of this is that we have to determine in which state we are. Discretizing the world around us is a way of solving this problem but induces several problems:

- The world is not a discrete place, we can use the tabular algorithms but are then forced to quantize the state-action space of the system. Discretizing the world means downscaling its complexity to a level that our algorithms can manage. Instead of adjusting our algorithms to deal with an infinite number of states we adjust the perceptional detail of the world in such a way that we do not have this problem anymore. The algorithms will search for optimal actions in this discrete system, while these might not be optimal actions for the continuous system.
- We introduce a strong bias towards what we as researchers think are good states. This means that we design the state-space of our agent in such a way that we introduce states that, we think, are important and necessary for the system to act optimally. This eliminates possible other states that might also be beneficial for the agent, but simply did not come up in the researcher's mind.
- The number of states that can be defined is finite, but for large numbers of states the complexity of updating the Q-table becomes increasingly more computationally intensive.

One popular way of solving this problem is by replacing the Q-table with a function approximator, in our case an Artificial Neural Network (ANN) [7, 8]. ANNs are networks of interconnected artificial neurons. These artificial neurons mimic the behavior of biological neurons and can be used to model complex relationships between in- and output pairs, finding patterns in data and function approximation. By using a function approximator instead of a table we bypass the problem of defining states, directly providing our sensory data (which encodes the world, and therefore also the state we are in) towards the network.

2.4.1 Multilayer Feed-Forward Neural Networks

The Artificial Neural Network is an interconnected assembly of simple processing elements, neurons, whose functionality is loosely based on the animal neuron. It was first introduced by McCulloch and Pitts [15] in 1943. Each neuron has an internal activation function, which is based on the input that the neuron receives. The processing ability of the network is stored in the inter unit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns [6]. One of the most commonly used network architectures is the Multilayer Feed Forward Neural Network which consists of multiple layers of neurons with one input and one output layer. The layers are connected to each other, allowing the neurons to propagate their activation towards the following layer. There are no recurrent connections in feed forward networks, networks that do have these kind of connections are called

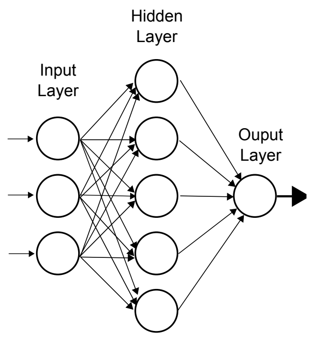


Figure 3: Example of a three layer neural network

Recurrent Neural Networks. Each connection in the network has a weight w_i, these weights are randomly initialized between 1 and -1, and are used to weigh all the incoming signals x_i (equation 6). Next the activation of the output of the neuron is calculated using the weighted input y, this can be done with for example a sigmoidal function (equation 7), a hyperbolic tangent (equation 8) or a simple linear activation function (equation 9), this result is forwarded to all neurons connected to that specific node. Note that a multilayer network with only linear activation functions has an equivalent single layer network that will have the same performance. An example of a simple multilayer feed forward neural network is shown in figure 3.

$$y = \sum_{i=0}^{n} w_i x_i \tag{6}$$

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{7}$$

$$\sigma(y) = tan^{-1}(y) \tag{8}$$

$$\sigma(y) = y \tag{9}$$

2.4.2 Training Algorithms

The most popular way of training a feed forward neural network is by the means of backpropagation. The heart of backpropagation is the backwards propagation of errors, hence its name. Backpropagation was introduced by Bryson and Ho in 1969 [5, 24] but at first did not gain much popularity. Only until the mid 1980s the real power of the backpropagation algorithm was discovered [22]. Backpropagation is a gradient descent supervised learning method, for learning the weights of both the hidden and output units of a neural network. Based on the delta rule (equation 10), it requires an input pattern together with the desired outcome of the network. The algorithm propagates the error that the network produces at its output backwards from the output neurons to the hidden neurons. This gradient will be used to modify the weights in such a way that this error is minimized. In order for the weights of the neurons in the hidden layer to be modified correctly it is necessary to calculate how much the hidden neuron contributes to the error in the next layer. Equation 11 describes the backpropagation algorithm.

$$\Delta w_{ji} = \alpha(t_j - \sigma(y_j))\sigma'(y_j)x_i \tag{10}$$

α is the learning rate

t_j is the target output

y is weighted input of the neuron

x_i is the i^{th}input

$$\begin{aligned} &\text{Learning rule:} && \Delta w_{ji} = \eta\delta_j x_i \\ &\text{Output units:} && \delta_j = (t_j - \sigma(y_j))\sigma'(y_j) \\ &\text{Hidden units:} && \delta_j = [\Sigma_k \delta_k w_{kj}]\sigma'(y_j) \\ &&& \text{(k=index unit next layer)} \end{aligned} \tag{11}$$

With the standard backpropagation algorithm, for each element in the training set the weights are updated towards each single element. This means that the weights can oscillate between training-iterations. By adding a small amount of the previous weight change we can counteract this oscillation, the magnitude of this amount is controlled with the momentum coefficient [27]. Introducing momentum also has the advantage that when the network weights are adjusted in the right direction, this direction is sustained. This smooths out irregularities in the training data, increasing the speed of convergence and providing a way for the network to escape local minima.

There a two ways of training the network, online and offline. In online training each time a new observation is made the networks weights are updated towards this single observation (sample by sample). In offline training the observations are stored, and training is done on a large batch of observations (also called batch training). The advantage of online training is that training is done while the agent is interacting with its environment, in offline training this is done afterwards. Using offline training however has the advantage that more sophisticated training algorithms can be used that converge faster and are more reliable than online gradient descent methods [21].

2.5 SARSA with Neural network

Using SARSA with a neural network has the big advantage that it is capable of learning the Q-function, while not keeping track of the Q-table. This approach has the advantage that it is able to deal with large state-spaces, but suffers from a longer learning stage. When using SARSA the network is used to act in the world. Actions are selected using e.g. the ϵ-greedy algorithm or Boltzmann-exploration, after each step the agent gets a reward, the SARSA value is calculated. This value can be compared to the value that the network outputs, this error can be backpropagated and the network will learn from this experience (providing an output that resembles the SARSA value more the next time it is in that state). In this case there is no test set, the performance of the network can not be monitored in terms of correctly classified/calculated outputs. The performance will reveal itself through the reward gathered by the agent, acting in the world. The hyperbolic tangent (equation 8) is nonlinear, but (contrary to the sigmoidal activation function, equation 7) equals zero in the origin, and its derivative equals one. This means that for small weight values the unit will resemble the behavior of a linear unit [8]. Linear function approximators have shown to converge to a good solution [26]. Using the hyperbolic tangent now introduces us with a unit that starts out analogue to a linear unit (when initialized with small weights), changing into a non-linear unit when this is necessary.

2.5.1 Network Topology

In general when combining a function approximator with Q-learning or SARSA it is common to construct one network for each possible action [7, 8]. In the case of n possible actions, n different neural networks are used. This has the advantage that learning the weights for a network associated with one specific action will not interfere with the learned weights of the remaining $(n-1)$ actions. It however introduces a little bit more complexity into the model, requiring more computations.

One single network to calculate the Q or SARSA value can be used. An example with three sensors and two action nodes is provided in figure 4. The network has action inputs, 'telling' the network for which action the Q or SARSA value has to be calculated. In the case of one single input node only a maximum of three actions can be encoded $(-1, 0, 1)$. In cases with more than three actions we have chosen to add one input node for each action, the action that is chosen gets the value 1 while all others get the value 0. This binarized action vector A also allows multiple actions to be chosen at one time step, extending the standard Q-learning and SARSA algorithms (we will not go into details as this

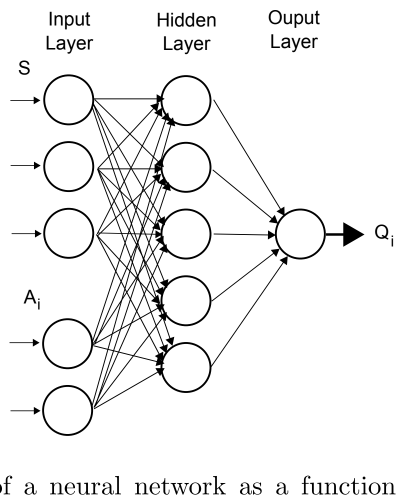


Figure 4: Topology of a neural network as a function approximator for the SARSA value (for an application with three sensors and two action nodes). S is the sensory input vector, encoding the system state. A_i is a list of all possible permutations of the binarized action vector. The network outputs the Q (or SARSA) value for permutation i.

is not within the scope of this thesis). All possible permutations A_i of the action vector are presented to the neural network, resulting in a list of quality values Q_i. The action vector with the highest Q value ($argmax_i(Q_i)$) is chosen (also taking into account $\epsilon - greedy$ action selection).

A pilot experiment has shown that for the mountain car using the binarized action input network compared to multiple networks had the advantage that it converged faster to a good solution. Because of this we have chosen to go for binarized action inputs for all of our experiments.

2.5.2 Training Algorithms

As a training algorithm we use a modified version of the backpropagation algorithm. In our case we calculate the output of the network (let's call this O), act in the world according to the outcome of O, gather our reward and calculate the SARSA value (Q). The standard backpropagation algorithm requires us to deliver the input and preferred outcome of the network. It then calculates the output of the network, the error and all Δs, in the case with SARSA we already have the output O, we can therefore skip the first part of the Backpropagation algorithm, and directly backpropagate the error without unnecessary calculations.

2.5.3 Implementation

We implemented the neural network with SARSA in Matlab. Matlab offers fast matrix calculations. This can be efficiently exploited for the calculation of neural networks [17]. Matlab however has as a downside that if and for-loops are slow. We tried to avoid these in our implementation. However this disadvantage is outweighed by the fast programming time. This project required some exploratory research, having a high level language (like e.g. Matlab) increased our realization speed, but is in the end slower at run-time.

The input values of the network are scaled in such a way that they all fall in the same range, this is needed because else the network will focus, in the learning stage, on the inputs that have the largest value. This is simply because even though the weights might be small, these larger values will influence the outcome of the network in a larger extent than smaller input values.

2.6 Neural Fitted Q Iteration (NFQ)

The Neural SARSA algorithms is an online approach. Another approach is offline, batch-training. Riedmiller [21] introduced the Neural Fitted Q Iteration (NFQ) as a model-free RL algorithm, which models the Q -function by a Multilayer Feed Forward Network (similar to Neural SARSA). However, instead of updating the network's weights after each timestep, a large number of samples (experiences) is recorded at runtime and the network is trained afterwards (offline). The used network topology is similar to the Neural SARSA algorithm, described in section 2.5.1.

The original NFQ algorithm described in [21] is a little bit too simple for our application, it starts out by calculating the Q values for all states. This table is used to act in the world, gathering more experiences (the rewards for all encountered state-action pairs). The list of experiences grows after each time-step. After a fixed amount of iterations all the Q-values are calculated according to the experiences, and the neural network is trained on this data. The trained network is used to calculate the new Q-values for all states and the loop starts over again. This approach has two downsides, the first is that we do not have the possibility to calculate the Q-values for all states (we avoid defining states in our model). Furthermore the increasing size of the experience-record can pose a problem on the computational demands of the model (e.g. in the case of a maximum of $m = 5000$ steps per episode, training for $N = 10000$ episodes results in an experience record of length $m * N = 50000000$. Computers can deal with these large tables, however storing the data has to be done on the hard-drive: resulting in a very slow running model). Considering these two

Algorithm 2.3 Neural Fitted Q Iteration

```
Initialize Neural Network (NN)
Initialize Pattern Set $P$
**Repeat N times:**
$s \leftarrow InitialStateVector$
**for each episode step i do**
    **select** $a$ **according to** $\epsilon - greedy(ForwardPropagate(NN, s), \epsilon))$
    **Take action** $a$**, observe** $r, s'$
    $s \leftarrow s'$
    **Store Experience:**
    $P.s^i \leftarrow s$
    $P.s'^i \leftarrow s'$
    $P.r^i \leftarrow r$
    $P.a^i \leftarrow a$
    **if** $s == terminal$ **then**
        **Break**
    **end if**
**end for**
**Train the batch after every** $k^{th}$ **epoch:**
**if** $mod(N, 5) == 0$ **then**
    **Initialize Goal** $G$
    **Initialize Input Patterns** $I$
    **for number of experiences j do**
        $G^j \leftarrow ForwardPropagate(NN, P.s^j, P.a^j) + \alpha(P.r^j + \gamma max_b ForwardPropagate(NN, P.s'^j, b) - ForwardPropagate(NN, P.s^j, P.a^j))$
        $I^j \leftarrow P.s^j, P.a^j$
    **end for**
    **while error < threshold do**
        **for number of experiences j do**
            $NN, error \leftarrow Backpropagate(NN, I^j, G^j)$
        **end for**
    **end while**
    **Reset Pattern Set** $P$
**end if**
```

points we decided to modify the existing NDQ algorithm in such a way that we avoid these problem. This is done by calculating the Q-values for a state only when needed (and the input vector that encodes the state is known), and the batch-size is limited. This limited size means that after each training run the batch is emptied (no history is kept, requiring considerably less data storage). The final algorithm is described in algorithm 2.3. Training is performed via the Backpropagation algorithm, described in section 2.4.2.

In the next section details of the Solar Boat will be explained, required for modeling the dynamic behavior and control (section 3). In section 4, we will return to the Reinforcement Learning algorithms with a pilot study (the mountain car problem) as a precursory to the problem of the Solar Boat Race.

3 Modeling the Solar Boat

For our computer simulation a descriptive model of the solar boat is constructed. This is done because the actual boat was not ready yet at the beginning of the project. Designing the computer simulation also aided in the mechanical design process, revealing some important aspects of the behavior of the hydrofoil in conjunction with the catamaran. After the modeling was completed and the boat was taking shape, several towing-tests were performed so we could confirm and refine our theoretical model to better match the real behavior of the solar boat.

3.1 Equations

Our mathematical model is mainly based on the findings of [25], along with some basic physics we came to the following set of equations, describing the behavior of our boat. For the simulations we used a time-step. A large time-step is desired to reduce the computation-time of our model, a small time-step is desired for a higher accuracy of our model. The value of the time-step was determined using a pilot experiment in which we started with a time-step of 0.01s, increasing it until the system was not able to simulate the boat correctly anymore. This resulted in a dt which was set at a value of $0.1s$.

3.1.1 Foil Drag

The drag of the hydrofoil is calculated using the following set of formulas:
Density of the water:

$$\rho_w = 1000 \ \ [kg/m^2] \tag{12}$$

Constants for the spray and interference drag (empirically defined, dimensionless):

$$C_{spr} = 0.24 \tag{13}$$

$$C_{int} = 0.1 \tag{14}$$

Dynamic viscosity of water:

$$\mu = 1.14 \cdot 10^{-3} \ \ [Pa \cdot s] \tag{15}$$

Number of struts piercing the water:

$$N_s = 1 \tag{16}$$

Number of corners:

$$N_j = 1 \tag{17}$$

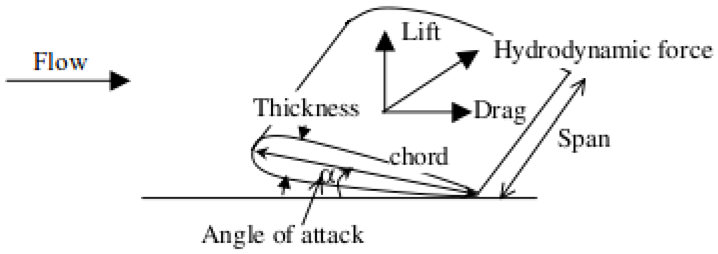


Figure 5: Dimensions of a hydrofoil

Influence of foil depth:
h = depth in chords
l = aspect ratio

$$E = 0.85 + \frac{0.16}{\sqrt{h/l}} = 0.98 \tag{18}$$

Angle of attack of the foil (see figure 5):

$$\alpha\,(rad) \;= \alpha_{front} = \alpha_{back} \tag{19}$$

Width of the foils (see figure 5):

$$b_{front} = 0.498 \;\; [m] \tag{20}$$

$$b_{rear} = 0.498 \;\; [m] \tag{21}$$

Chord length (see figure 5):

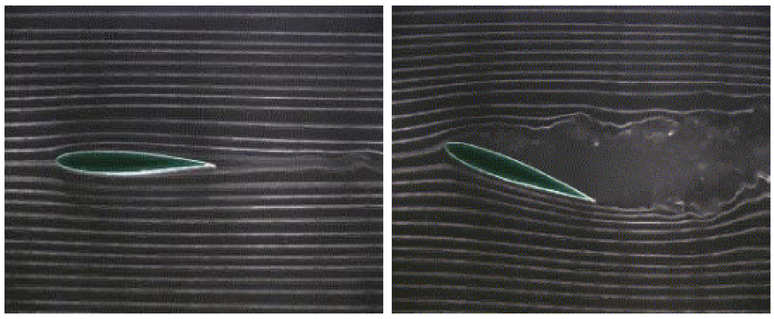
$$c = 0.12 \;\; [m] \tag{22}$$

Foil thikness (see figure 5):

$$t = 0.0125 \;\; [m] \tag{23}$$

Lift, alpha cannot exceed 7°or else the flow will separate from the wing, resulting in stall (see also figure 6):

$$L(\alpha, b, v_f) = \alpha \cdot \rho_w \cdot {v_f}^2 \cdot b \cdot c \cdot 2 \cdot \pi \;\; N \quad \text{with} \quad \alpha \in [0^\circ, 7^\circ] \tag{24}$$



(a) Viscous Flow ($\alpha = 0°$) (b) Viscous Flow ($\alpha = 22°$)

Figure 6: Experimental flow around a hydrofoil

Dynamic Pressure:

$$q_w(v_f) = \frac{1}{2} \cdot \rho_w \cdot {v_f}^2 \ \ [N/m^2] \tag{25}$$

Friction Drag, this is the drag produced by the viscosity of the water meaning the pressure of the water against the submersed part of the foil, traveling at a certain speed:

$$D_{fri}(\alpha, b, v_f) = q_w(v_f) \cdot 2 \cdot b \cdot c \cdot \left(\frac{0.075}{({}^{10}log(\frac{\rho_w \cdot v_f \cdot c}{\mu}) - 2)^2}\right) \ \ [N] \tag{26}$$

Interference Drag, when two submerged bodies are close to each other the amount of drag that arises is higher than the sum of the drag of both bodies, this drag is produced by the pressure/wave interference between submersed bodies and can be defined as follows:

$$D_{int}(\alpha, b, v_f) = q_w(v_f) \cdot C_{int} \cdot N_j \cdot t^2 \ \ [N] \tag{27}$$

Spray Drag, the amount of drag produced when an object pierces the water. This creates a spray that in its turn produces drag:

$$D_{spr}(\alpha, b, v_f) = q_w(v_f) \cdot C_{spr} \cdot N_s \cdot t^2 \ \ [N] \tag{28}$$

Induced Drag, the amount of drag as a result of the lifting force. A foil creates lift by bending flow. Because a pressure difference arises due to the difference in flow velocity at the top and the bottom of the wing, water (or air) will flow sideways, towards the tips (where the pressure difference is low). This

Figure 7: Wake vortex on an airplane, source: NASA Langley Research Center (NASA-LaRC)

creates turbulent flow, also called the wake vortex (see for an example figure 7). Creating this turbulent flow requires energy, but the swirling motion also causes air to move down behind the wing (called downwash), resulting in drag:

$$D_{ind}(\alpha, b, v_f) = (\frac{2}{\pi \cdot E \cdot \rho_w \cdot b^2}) \cdot (\frac{L(b)^2}{v_f^2})\ [N] \tag{29}$$

The total drag of the hydrofoil can now be defined as:

$$D_{total}(\alpha, b, v_f) = D_{fri}(\alpha, b, v_f) + D_{int}(\alpha, b, v_f) + D_{spr}(\alpha, b, v_f) + D_{ind}(\alpha, b, v_f)\ [N] \tag{30}$$

Figure 8 shows $D_{total}(\alpha, b_{front}, v_f) + D_{total}(\alpha, b_{back}, v_f)$ for $\alpha = 2°$, a reasonable

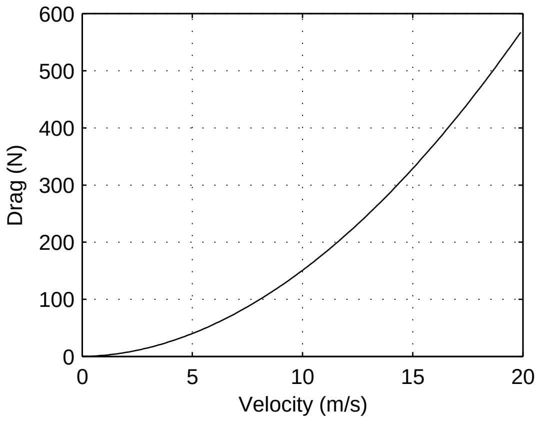


Figure 8: Foil drag characteristics for $\alpha = 2°$

angle of attack also used in moth sailboats [16], and v_f ranging up from 0 till 15m/s.

3.1.2 Boat Elevation

Lifting the boat out of the water means that we have to simulate the elevation of the boat. This also means that, as the boat gets higher out of the water the drag of the hull decreases.
Mass:

$$m = 180 \;\; [kg] \tag{31}$$

Gravitation Force:

$$F_z = m * g \tag{32}$$
$$g = 9.81 \;\; [m/s^2]$$
$$F_n = F_z (\text{in rest}) \tag{33}$$

We now introduce a lifting force F_{foil}, this force is in the same direction as the F_n (the normal force), we can rewrite equation 33 to 34:

$$0 = F_n + F_{foil} - F_z \tag{34}$$

As the foil is producing lift and F_z is a fixed value the increase of F_{foil} will decrease F_n. The depth of the hull when stationary (so the foil is not providing any lift) is $-0.09m$, at this point $F_n = F_z$. At depth $0cm$, $F_n = 0N$ (the water does not provide any buoyancy anymore), meaning that the entire weight of the boat has to be carried by the foils ($F_{foil} = m * g$). The transition from $-0.09m$ to $0m$ introduces a decrease in F_n, this relation is dependent on the shape of the hull (its volumetric size). We assumed that this relation, and its effect on F_n, and therefore its effect on the resulting force F_{up} can be described with equations 35 and 36 in which factor is a non-linear value between 0 and 1, with 1 = fully buoyant and 0 = on surface.

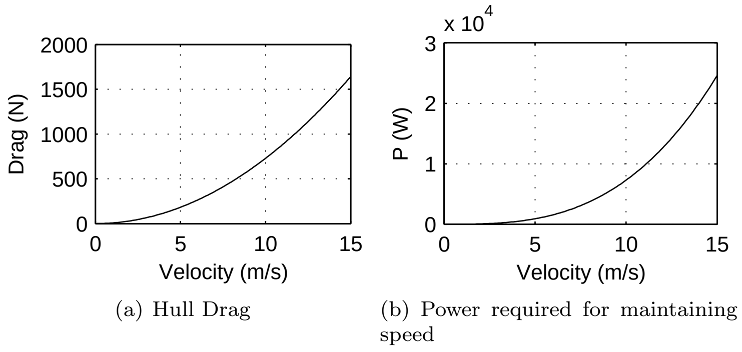
The height of the boat is now calculated using algorithm 3.1, which combines all previous formulas.

$$factor = (\frac{depth}{-0.09})^{0.4} \tag{35}$$

$$F_{up} = F_{foil} + (F_n \cdot factor) - F_z \;\; [N] \tag{36}$$

Algorithm 3.1 $elevation(F_{foil}, height, v_{up})$

```
$height \in \Re$
$F_{up} = F_{foil} + (F_n \cdot (\frac{depth}{-0.09})^{0.4} - F_z$
$a = \frac{F_{up}}{m}$
$v_{up} \Leftarrow v_{up} + a \cdot dt$
$height \Leftarrow height + v_{up} \cdot dt$
if $height > strut\_length$ then
    $height \Leftarrow strut\_length$
end if
```



(a) Hull Drag (b) Power required for maintaining speed

Figure 9: Hull characteristics for a fixed water depth of -9cm

3.1.3 Hull Drag (Water)

Figure 9(a) shows the water drag of the hull for a fixed water depth (-9 cm). The power that is needed for the boat to remain at the same velocity, given the drag of the hull is shown in Figure 9(b).

We estimated the frontal area of the submersed part of the boat (37) and the form factor (38) by fitting the performance of our previous boat, the sunlimited, to our equation. The sunlimited was able to travel at a speed of $7m/s$ with $2500W$ of power. This resulted in the factors for our new boat:

Wetted surface area of the boat:

$$S_w = 0.3 \ [m^2] \tag{37}$$

Form factor (based on the drag coefficient, dimensionless):

$$C_w = 0.04859 \tag{38}$$

Water Drag:

$$D_{water} = \frac{1}{2} \cdot S_w \cdot \rho_w \cdot C_w \cdot {v_f}^2 \ [N] \tag{39}$$

The previous formulas are for a normal boat, when the depth is varied the resistance will be different. As the boat gets lifted out of the water the drag will decrease. We estimate that this will happen similar to the change in buoyancy (formula 35):

$$D_{water}(depth) = \frac{1}{2} \cdot C_w \cdot S_w \cdot \rho_w \cdot {v_f}^2 \cdot (\frac{depth}{-0.09})^{0.4} \ [N] \tag{40}$$

Figure 10 shows the result of equation 35 for a water depth of -9cm up till 0cm (where the boat is completely lifted out of the water). Lifting the boat out of the water will decrease the drag significantly, this means that even when not completely hydrofoiling a hydrofoil can support the boat by decreasing its drag as long as the drag of the submersed foil does not exceed the amount of reduced drag of the hull.

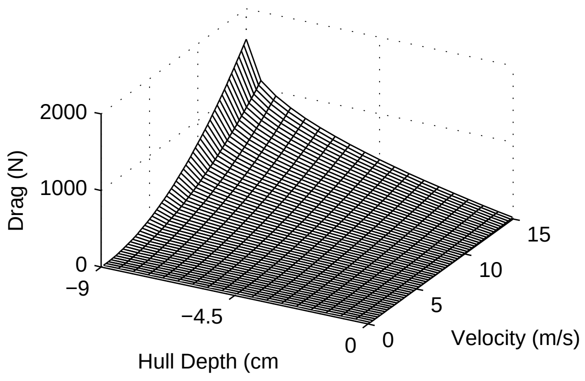


Figure 10: Drag as a function of the Hull's depth

3.1.4 Hull Drag (Air)

Air Density:

$$\rho_a = 1.25 \ \ [kg/m^2] \tag{41}$$

Frontal area of the boat:

$$S_a = 0.35 \ \ [m^2] \tag{42}$$

Form factor (based on the drag coefficient, dimensionless):

$$C_a = 0.4 \tag{43}$$

Dynamic Pressure:

$$q_a(v_f) = \frac{1}{2} \cdot \rho_l \cdot {v_f}^2 \ \ [N/m^2] \tag{44}$$

Air Friction:

$$D_{fri\_air}(v_f) = q_a(v_f) \cdot S_a \cdot C_a \ \ [N] \tag{45}$$

Figure 11 shows the result $D_{fri\_air}(v_f)$ for $v_f = 1$ up til 15m/s

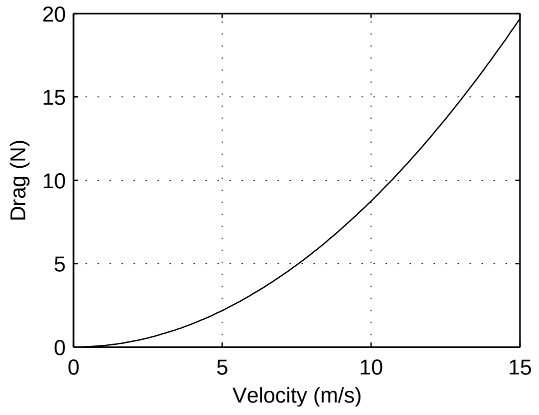


Figure 11: Air drag as a function of the velocity

3.1.5 Thrust

We have described numerous drag-coefficients, these are all forces that slow the boat down. Now lets look at propulsion. This can simply be described as a force F_{thrust}, in opposite direction of all drag coefficients. The acceleration of an object with mass m is defined as:

$$a = \frac{F}{m} \tag{46}$$

The velocity of the object depends on the velocity that it already has, and the timestep (dt) at which we sample our simulation:

$$v \Leftarrow v + a \cdot dt \tag{47}$$

Our absolute traveled distance depends on the velocity, and again the timestep (dt):

$$x \Leftarrow x + v \cdot dt \tag{48}$$

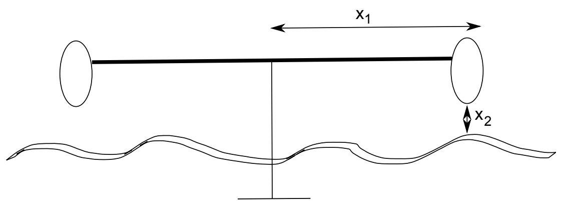
Substituting F with F_{res}, the total of all horizontal forces ($F_{res} = F_{thrust} - (D_{fri\_air}(v_f) + D_{water}(depth + D_{total}(\alpha, b, v_f)))$) acting on the boat now gives us a way of simulating our boat.

For our simulation we want our amount of Thrust to be dependent on the Power input, delivered to the motor. The amount of thrust is determined using Javaprop [12]. Our used propeller is imported in Javaprop's simulation, delivering us with a lookup table containing the amount of power going into the Propeller and the resulting amount of thrust that can be delivered for one single speed. This simulation is done for range of speeds (from $v = 0$ up till $v = 23$ in twenty steps (defined by Javaprop)) and Power ($P = 0$ up til $P = 3000$ in steps of 100W). The resulting lookup table can be used to determine the amount of thrust in Newtons given the current speed and Power input. A discount factor is used to include the efficiency of the motor with controller and all involved mechanics, this is a simplification and the size of this discount factor is roughly estimated at an 0.6% efficiency. Future work should investigate the efficiency of all the electronic and mechanical systems on the boat, this however is beyond the scope of this research project and should be performed externally.

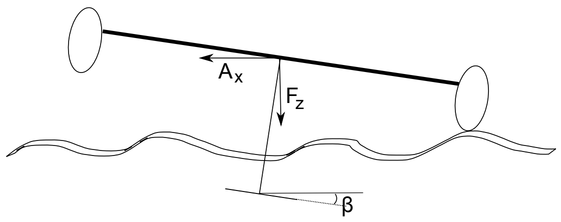
3.1.6 Cornering

Traveling at a high velocity means that care has to be taken when taking corners. The speed at which a corner can be taken on hydrofoils is dependent on the sailing height (the clearance between the hull and the water), the width of the boat and the mass. Given a total mass m of 180kg of weight gives $F_z = 1765.8N$, $x_1 = 1.1m$ and $x_2 = 0.3m$ respectively being the distance from the center of the boat to the outside of the hull and the height of the hull above the water (see figure 12.a). The maximum angle (between the horizontal plane and the deck) that the boat can tilt before one of the hulls hits the water is defined as $\beta = tan^{-1}(\frac{x_1}{x_2})$.

The maximal centrifugal force A_x can be defined as being $A_x = \frac{x_2}{x_1} \cdot F_z = 481.6N$. This maximum velocity for a given radius can now be calculated using



(a) Dimensions



(b) Resulting forces when cornering

Figure 12: Cornering with a hydrofoiling catamaran

formula 49. The result of this equation is shown in figure 13. The inverse of equation 49 is given by 50, with this equation we can calculate for a given speed what the maximum corner radius is that the boat can handle.

$$v_{corner}(r_{corner}) = \sqrt{\frac{A_x \cdot r_{corner}}{m}} \tag{49}$$

$$r_{corner}(v) = \frac{m \cdot v^2}{A_x} \tag{50}$$

It is hard to measure beforehand the radius of every corner for the solar race track, it is also not possible to deduce this from the GPS data (which are

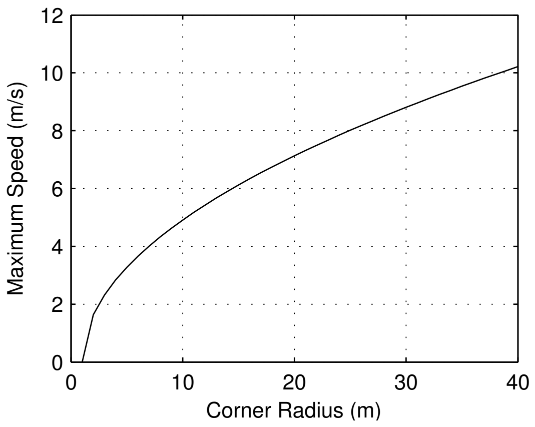


Figure 13: Maximum velocity given a corner radius

Relative Angle (°)	**Corner Radius**	**Max Velocity**	**Occurrences**
0 - 40	> 40m	none	571
40 - 80	> 40m	none	190
80 - 120	≈ 20m	7m/s	44
120 - 180	> 40m	none	1

Table 1: Correlation between Corner Radius and Relative Angle

merely way-points). They can be used to determine the relative angle between two track-segments. We calculated the angles for every turn in the entire race, random measurements show that there is a correlation between the corner radius and the relative angle of the corner. These measurements have provided us with the information shown in table 1 (with the angle meaning the change in bearing, 0° means no change in bearing while 180° means going in the opposite direction). The maximum angle that is encountered during the race is 174.4°.

We use this information to gradually reduce the speed of the solar boat in the simulation towards 7m/s when facing corners which fall in the 80-120°bin if the speed of the boat exceeds the safe limit. Appendix A.2 (figure 29) shows all the corners that fall into this bin on the entire track.

3.1.7 Weather

Weather data is taken from the KNMI-dataset [11], this dataset contains weather information from 1987 until now with a resolution of 1 hour. The weather station of choice is located in the city Leeuwarden (lon: 5.755 lat: 53.225), this city acts as both the starting and finishing point for the Frisian Solar Challenge. All the elements that are recorded are summarized in Appendix 6.

The global radiation, Q (in J/cm^2h^{-1}), during one hour can be converted to Watt using the following equation:

$$nJ/cm^2h^{-1} = \frac{n}{3600} \cdot 10000 \ [W/m^2] \tag{51}$$

As our initial global radiation model we averaged the global radiation data from the KNMI over the last ten years. This resulted in the mean radiation-estimation shown in figure 14. By introducing a factor we can scale this figure so that it matches the amount of power that the boat actually receives. This factor can change over time, which can be interpreted as changing weather (e.g. for the simulation of clouds).

We extended this simple model with a scenario based approach. We set up several scenarios for the weather (as discussed in section 2.1.5), all taken from the KNMI-dataset. The KNMI-dataset contained the global radiation data of 279 days (in Juli) with a one hour interval. For each simulation iteration we randomly choose a day as our scenario and run our experiment. This way the boat will learn to cope with all possible weather situations that were perceived in Leeuwarden over the last 9 years.

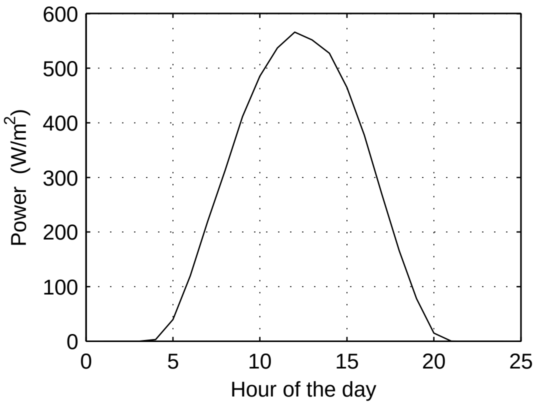


Figure 14: Global radiation, averaged over the last ten years

3.1.8 Solar Panels

In our model we incorporate the expected amount of energy coming from the solar panels. For this we are interested in the average sunshine duration, the percentage of maximum potential sunshine duration and the global radiation (respectively SQ, SP and Q in table 6). Given the starting time of each track and the GPS information, we can calculate the zenith and azimuth angle of the sun (see also figure 15). The projected area of any surface is given by equation 52, in which β is the angle between the surface and the line of sight, and A is the true area of the surface (with $\beta = 0$). For a Rectangle this can be simplified to equation 53 with L being the length of the rectangle and W the width.

$$A_{projected} = \int_A cos\beta \, dA \tag{52}$$

$$A_{projected-rectangle} = L \cdot W \, cos\beta \tag{53}$$

Calculating the projected area of our solar panels and taking into account the efficiency of the solar panels at different angles (to be added later), when

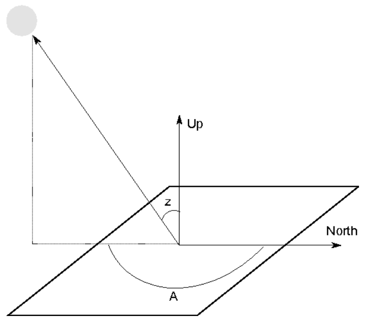


Figure 15: Azimuth (A) and Zenith (z) angle

given the solar radiance we can determine the total amount of energy that we receive from the sun.

3.1.9 Shadow

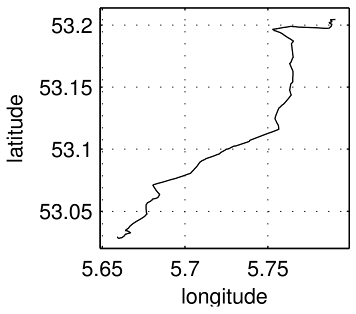
Shadow influences the amount of sun-power reaching the solar panels. For an accurate simulation all obstacles around the race-track and their heights have to be recorded. Given the height of the objects and the distance to the boat it is possible to calculate if the object is dropping a shadow on the boat, reducing its energy input. This however requires a lot of annotations (the locations and heights of the objects), but also calculations. Therefore we have chosen to simplify this issue. The simplification comes down to only taking into account very large objects, from which we know for sure that they will cast a shadow on the track. The GPS locations of all these objects should be recorded. Take for example a tree, it can be defined as one GPS coordinate (its root), however it casts a large shadow. We have defined that when the boat is within 5m from a shadow-location, it affects the solar input with a factor Z_{shadow}. This means that the tree will cast a shadow of 10m long. This is a simplification, not all trees will have a shadow that is 10m long, and also not all objects which cast shadows on the track are trees (there are are also buildings that influence the input from the sun). When there are objects which cast bigger shadows, more GPS coordinates have to be added (e.g. with lines of trees). Only recording the GPS coordinates however is not enough, one also needs to calculate the angle between the objects and the boat. If this angle is in the same quadrant as the azimuth angle, the object can cast a shadow on the boat. Only if this is true (and the boat is within 10m of the object) the sun intensity is decreased by a factor $Z_{shadow} = 0.6$ (also an estimation).

3.1.10 The Race

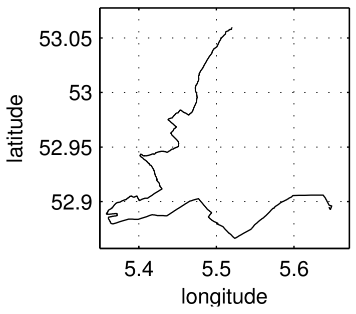
The Frisian Solar Challenge follows an ice speed skating competition with a length of 220km. The race is conducted on canals, rivers and lakes and divided into six stages. The GPS coordinates of the track are built in Google Maps, from which they were exported to a .kml file. We have build a Matlab function that is able to import this kml file into our simulative environment. The imported data

Day	**Stage**	**Km**	**Start Time (estimated)**
Monday 5 July	Leeuwarden - Sneek	26.0	11:00
	Sneek - Sloten	20.0	14:00
Tuesday 6 July	Sloten - Bolsward	58.0	10:00
Thursday 8 July	Bolsward - Harlingen	18.8	11:30
	Harlingen - Franeker	11.8	15:00
Friday 9 July	Franker - Bartlehiem	30.3	9:00
	Bartlehiem - Dokkum	13.0	14:00
Saturday 10 July	Dokkum - Grote Wielen	25.0	11:00
	Grote Wielen - Leeuwarden	5.4	15:00

Table 2: Race schedule Frisian Solarchallenge 2010



(a) Stage 1, Leeuwarden - Sneek



(b) Stage 3, Sloten - Bolsward

Figure 16: Two samples of the GPS information of the track used in our simulation

contained way-points, between these way-points lines can be drawn; enabling us to calculate the length of the track and the angle of all corners (see also section 3.1.6). A summary of the race is provided in table 2. Two samples of the tracks being used are shown in figure 16. All tracks can be found in Appendix A.2.

3.2 Data Gathering and Model Confirmation

As the boat is beginning to take shape it is important to validate our model. In our model we made several (large) assumptions, these represent themselves in three ways. First we assumed that we could describe the hydrofoil behavior according to the formulas given by [25]. Secondly we describe the behavior of the air and water resistance in a similar way. Thirdly some parameters for the formulas had to be estimated. Validating our model means that we first have to look at the most important factor, formulated as a hypothesis the research question becomes: "Can we describe the behavior of our hydrofoils and catamaran using our simple formulas?".

If this **is** the case the next hurdle becomes refining our parameters, tweaking the values until our recorded data resembles the curves provided by the computer model. More about this will be discussed in section 3.2.1.

If this is **not** the case our entire set of formulas can be set aside. In this case we have to find a way of recording the drag for the different input parameters that are important. More about this will be discussed in section 3.2.2.

3.2.1 Refining our Model

If we are allowed to refine our model we have to find the values for all our parameters, the parameters that have to be refined are:

- C_{spr} (equation 13)
- C_{int} (equation 14)
- E, h, l (equation 18)

- α_{front} and α_{back} (equation 19) At what angle is the foil set up, and what is the influence of steering the tip? Can this be translated in terms of changing the value of α.
- S_a (equation 42)
- S_w (equation 37)
- C_a (equation 43)
- C_w (equation 38)

Variables that have to be previously defined/measured, not following from the experiment (Difficulty: The dimensions of the bladerider are not uniform):

- b_{front} (equation 20)
- b_{rear} (equation 21)
- c (equation 22)
- t (equation 23)

During the experiments measurements that have to be taken are:

- **Velocity**. We have to measure the velocity of the boat, GPS is not accurate enough. For this we suggest logging the speed with a standard boat-speedometer. When this is difficult, measurements can also be taken from the winch. This however assumes that we do not have any slack in the pulling-cable. The RPM of the winch can be recorded, using the diameter of the winch we can calculate the speed at which the boat is traveling.
- **Drag**. Drag should be recorded by a digital force gauge between the pulling-rope of the boat and the boat itself. Using the velocity and the drag we can calculate how much energy is needed for the boat to travel at this speed.
- **Angle of Attack**. α should be recorded throughout the experiment, this value influences the drag significantly. Since the main foil is stationary we suggest measuring the angle of the sensor.
- **Height**. Height measurements should be taken, when the boat is foil borne the drag of the water on the hull is zero. When performing our data analysis it is important to know when the boat is hull borne or foil borne.
- **Water Depth**. The water depth influences the performance of both the hydrofoil and the hull, this means that recording the water depth gives us the ability to analyze these effects. After the first experiments we have to find another spot, where the water depth is sufficiently/significantly different. This will also be discussed in section 3.2.3.

For this refinement we have to do a series of towing-tests, from $0m/s$ we pull the boat through the water, in several configurations (hullborne and foilborne). The model has to be fitted to the drag curves that are recorded. As there are a lot of parameters, knowing which parameters have to be tweaked is difficult. In order for our job to be simpler we suggest simplifying our model (several factors are multiplied with each other), resulting in less variables (but also the variables have less 'meaning'). This will be done beforehand, hopefully resulting in less factors to be tuned, giving us the ability to quickly check if we can use our model or not.

The drag during the extending and retracting moves are difficult to capture in our model, we suggest recording the drag (as discussed in section 3.2.2) in this situation and skipping capturing this situation in the model.

The parameters of our model are fitted by hand, fine tuning the parameters of our simulation until they match the recorded data. This might seam tedious, but a lot of equations can be simplified in such a way that very few parameters have to be tuned.

When we are unable to fit our model to the measured data we must conclude that it is not possible to model the solarboat with our equations, we can overcome this by replacing all the equations with recorded values. This will be discussed in the following section.

3.2.2 Discarding our Model

If we are not allowed to refine our model we have more work to do. Instead of using formulas to define our model we now have to measure the drag of the boat/foils in all possible cases. We have chosen for a rotary suspension of our foil, extending and retracting the foil means that the speed of the boat decreases to a speed between 0 and $2m/s$, this depends on the thrust being applied during the transition. This is convenient for the data analysis, we now only have to measure the drag of the boat from $0m/s$ up till its top speed, including the phase transition. This test should also be performed for the boat without extending or retracting the foil (only the drag of the hull has to be determined).

We also have to determine how much drag is being produced when the boat is moving at velocity v and the foil is being extended or retracted. The velocity at which this should be performed for the extension move is the maximum hull borne velocity of the boat. The velocity at which the drag should be measured for the retracting move should be the speed at which the boat is no longer able to hydrofoil. This means that we first have to determine when hydro foiling, at which speed the boat cannot keep itself out of the water (transitioning between the foil borne to the hull borne state).

3.2.3 Future Work

In theory the model confirmation sounds straightforward, however in practice it is difficult to get good measurements. Difficulties with the boat and with the recording equipment cost a lot of time, resulting in the absence of reliable data; even two weeks before the race. Given that our final model has to train for quite some time (days, or even weeks), adjusting our model to fit the real data was not feasible given the limited amount of time before the race. This however is very important, without validation our model is of no use in the real world. If this project is to be continued in the future it is important that data gathering takes place at least a few months in advance of the race. In this particular case this was not possible due to a lot of setbacks, however after the race a lot of readings can still be done; improving our model for next years.

When all the drag curves, mentioned in the previous sections, are measured it is possible to focus on the influence of water-depth on our boat. Recording the water depth at our first test-site will give us one sample, next we have to find a spot where the water depth is sufficiently different (depending on the first test-site), but uniform (the depth should not change much throughout the experiment). Taking the same measurements, as described in the previous sections, we can investigate how much the influence of water depth is to the total drag of a bullhorn and foil borne boat and use this to come to a better simulation of the solar boat.

3.3 MATLAB

For our implementation we chose for MATLAB (MATrix LABoratory), a numerical computing environment developed by MathWorks. MATLAB is specifically optimized for matrix manipulations, but also for the plotting of functions and data and implementation of algorithms. User Interfaces can be created and it can be interfaced with other programs. Being a high level programming language, the resulting code is slower than for example C, C++ or Java, but because it is a high level language programs can be quickly developed. This is an advantage in our experiment, a lot of modeling has to be done, the easy plotting tools allow quick debugging and gives us the ability to come up with a functioning and flexible program quickly.

4 A Pilot Experiment In Reinforcement Learning: The Mountain Car

The Mountain Car problem is a reinforcement learning problem which acts as a step up towards the more complex problem of the solar boat. It is used in several articles discussing (PO)MDPs [3, 26, 9]. The mountain car task falls within the same range as another popular reinforcement learning problem, the inverted pendulum [2, 30]. Starting with a well-understood toy-problem gives us the ability to benchmark our algorithms, iteratively increasing its complexity until we end up at the simulation of the solar boat for the entire race.

4.1 Problem Description

Consider an underpowered car that has to be driven up a steep mountain road (figure 17). The gravity in this model is stronger than the car's engine, this means that the only solution for reaching the goal is first moving away from the goal up the opposite slope on the left. Then, by applying full throttle the car is able to build up enough kinetic energy to make it higher up the slope, this may be repeated several times. The beauty of this problem is that before reaching the goal, things have to get much worse (the distance between the car and the goal has to increase). In an MDP situation the car can perceive its position and velocity. When the current position of the car is unknown to the system the problem becomes a POMDP. The mountain slope can be discretized into states, including the goal states. This means that a tabular algorithm can be applied to this problem.

4.2 Implementation

There are several implementations of the mountain car problem available, we used Matlab for the implementation of our reinforcement learning framework. A Matlab implementation of the mountain car problem is given by José Antonio Martin [14]. This implementation is modified so it can be used with our reinforcement learning framework.

The mountain of the problem is defined as a cosine function, the car moves by subjecting it to a force ($force\_direction$). This force can be to the left, right or zero (when no force is being applied). These three actions are the only actions that the agent has access to. For every time-step that the car does not reach the goal it gets a penalty (a negative reward), when the goal is reached

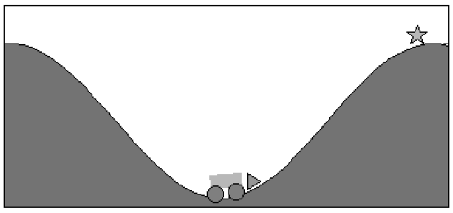


Figure 17: The Mountain Car Problem, the star denotes the goal [14].

the simulation is ended. A complete description of the simulation is given in algorithm 4.1.

Algorithm 4.1 *Simulate_MountainCar(force_direction, position, velocity)*

```
$force\_direction \in \{-1, 0, 1\}$
**if** $position < terminal\_position$ **then**
  $loss = 0.999$ (friction coefficient)
  $velocity \Leftarrow loss * (velocity + (0.0005 \cdot force) + (-0.0025 \cdot cos(3 \cdot position)))$
  $position \Leftarrow position + \Delta t \cdot velocity$
  $Reward \Leftarrow -1$
**else**
  Terminate Simulation
**end if**
```

Three types of algorithms were implemented:

- Random action selection: At each time-step the action that the policy chooses is completely random, this acts as a baseline for comparing the performances of our policy learning algorithms.
- Tabular SARSA (POMDP): A tabular implementation of the SARSA policy learning algorithm.
- NN-SARSA (POMDP): The neural network implementation of the SARSA policy learning algorithm.

4.3 Results

We modeled the problem as a POMDP, meaning the mountain car can only perceive his own velocity, it does not know where it is on the slope. As a performance measurement we took the number of required iterations for the algorithm to reach the goal, with $\epsilon = 0$. This means that no random effects influence the performance of the agent.

We performed three experiments in which the same settings for the algorithm were used, for each experiment 100 trials were performed. The maximum number of steps for each episode/trial was set at 1000 steps.

The following SARSA settings were used for the tabular algorithm:

$$\begin{aligned} learning\_rate &= 0.5 \\ \gamma &= 1 \\ \epsilon\,(i) &= 0.02 \cdot 0.995^{i} \\ &\text{with } i \text{ being the number of successfully finished episodes} \end{aligned} \tag{54}$$

For the NN-SARSA algorithm we coarsely searched the parameter-space and came up with the following parameters (The action input node varies between -1 and 1, where -1 stands for steering to the left, 0 for performing no action and 1 steering to the right. See also section 2.5.1):

$$\begin{aligned} n_{sensor-inputs} &= 1(\text{velocity}) \\ n_{action-inputs} &= 1 \\ n_{hidden} &= 20 \\ n_{output} &= 1 \\ learningrate\, \eta &= 0.03 \\ momentum &= 0.1 \\ \gamma &= 1 \\ \epsilon\,(i) &= 0.02 \cdot 0.995^{i} \\ &\text{with } i \text{ being the number of successfully finished episodes} \end{aligned} \tag{55}$$

For the tabular SARSA implementation we tested four different situations with 20, 50, 100 and 200 velocity states, together with 3 actions this resulted in a Q-table of size 20*3 up till 200*3.

The results of the experiments are shown in figure 18. The Random action selection algorithm required an average of 903.5 steps to reach the goal with a standard deviation of 180.7. The Tabular SARSA algorithm with 20 states resulted in a mean performance of 897.4 steps with a standard deviation of 180.4. The best performance came of the Neural Network SARSA algorithm, requiring an average of 275.8 steps with a standard deviation of 20.2 to reach the goal. The minimum number of steps required was 211 steps.

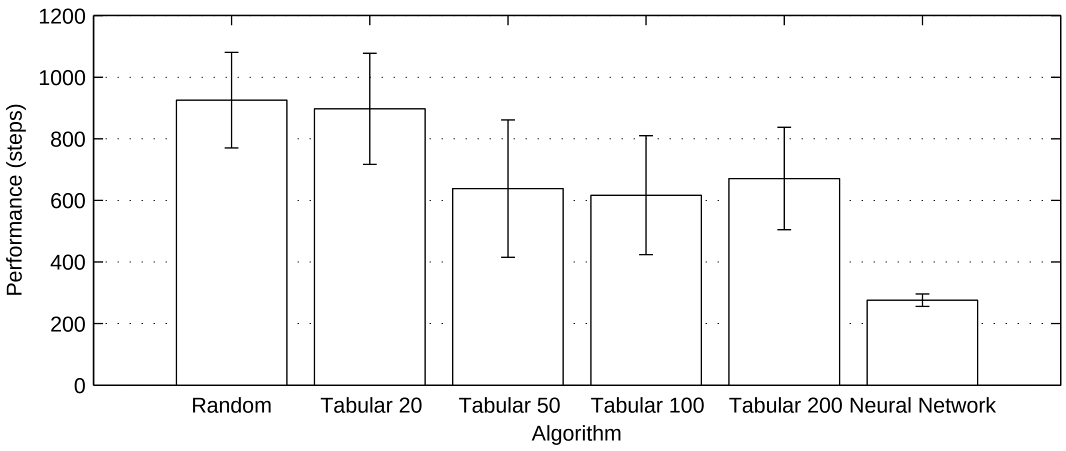


Figure 18: Results of the (POMDP) SARSA mountain car experiments. Lower equals a better performance (less steps required to reach the goal). Errorbars denote the standard deviations. The Neural Network SARSA algorithm outperforms the Tabular SARSA implementations and Random Action Selection significantly.

A one-way ANOVA was used to test for performance differences between the algorithms. Performance has been found significantly different across these six algorithms, F (2, 594) = 193.19, $p < 0.01$.

Post hoc comparison using the Tukey HSD test indicated that the performance of both the Random (M = 925.4, SD = 155.1) and the Tabular 20 (M = 897.4, SD = 180.4) algorithm were not significantly different. They however showed a significant difference towards the Tabular 50 (M = 638.3, SD = 222.7), Tabular 100 (M = 616.9, SD = 192.9) and the Tabular 200 (M = 671.0, SD = 166.5) algorithm (which show no significant difference between one another). The Neural Network algorithm (M = 275.8, SD = 20.2) proved to be the best algorithm, and showed a significant difference towards all other algorithms (all with a 95% confidence interval).

The NN-SARSA implementation outperformed both the Random Action Selection and Tabular SARSA algorithms significantly in the case of the POMDP mountain car problem.

4.4 Discussion

The mountain car is an easy problem in the sense it only has to figure out that when the car is moving to the left, it should apply force to the left and vice versa. This means that the simplest (and optimal) policy is always to apply force in the direction that the car is moving, this is very easy to learn for any function approximator. When the problem is modeled as an MDP, the mountain car can perceive its speed and position on the slope. Upgrading the problem towards a POMDP leaves out the perception of the position, this actually filters out trivial information (location). The remaining information (velocity) is the only variable that is required for reaching an optimal policy. The experiments however showed that the Tabular SARSA algorithm did not perform as well as the NN-SARSA algorithm. Our first assumption was that this was due to the number of states in the system, increasing the number of states should induce better performance. This was the case when comparing the Tabular SARSA with 20 states with the setup with 50 states, however increasing the number of states even further did not yield any improvement. Multilayer Feed Forward Neural networks can take a continuous variable as input, and provide a non-linear mapping from input to output states. This has several advantages, mainly the advantage is that no discrete number of states have to be defined (creating a continuous state-space). One might say that this could explain why the NN-SARSA implementation outperformed the tabular approaches, however this is not backed up by the results of increasing the number of states in the Tabular SARSA algorithm (no significant difference was observed when the state space becomes greater than 50 states). The good performance of the NN-SARSA algorithm is probably due to the nonlinearity and symmetry of the tanh activation functions of the neurons. The tanh function is negative for activations below the bias (and vice versa positive for activations above the bias). When randomly initializing the network with small weights between +0.15 and -0.15, there is a 50% chance that the input variable (the velocity in this case) will produce an activation with equal sign. Combining this with the input action node means that only a relation between the action node has the same sign as the velocity node, or vice versa. This can quickly be learned by a neural network. We conclude that the initial architecture of a non-linear Multilayer

Feed Forward Network is in favor of solving the mountain car problem. This however might not hold for other reinforcement learning problems (e.g. the solarboat). Using a Feed Forward Artificial Neural Network to estimate the Q-value of the SARSA algorithm still appears to be promising for POMDPs. The subproblem of the mountain car laid down the foundation for the much more difficult problem of making optimal use of available power in a solar powered boat which we will discuss in the next section.

5 Making Optimal Use of Available Power And Optimizing Overall Speed in a Small Solar Boat Race

After tackling the mountain car problem, we iteratively increased the complexity of our model. We first designed an algorithmic model of the solar boat, described in section 3 [25]. This model is at the heart of our following experiment, the simulation of a solar powered boat in a small race (without obstacles and events along the way).

5.1 General Problem Description

Consider a straight,linear track on which a speed race is held. The given amount of energy is not enough for the boat to reach the finish-line, giving full throttle the entire time. This means that for optimizing the overall-time in this race-situation, a policy has to be found which utilizes its energy in a smart way. Changing from hullborne to foilborne will also aid in the overall speed of the boat. Deploying the hydrofoil lowers the drag and increases the overall speed of the boat, giving it the ability to finish the race within time.

5.2 Implementation

The length of the race was set to 500m, the maximum number of steps was set to 5000 (with dt set at 0.1 the boat has a total of 500 seconds to complete the track). A Graphic User Interface was constructed so we could view the different variables change over time and analyze the boat's behavior. The interface is displayed in figure 19.

5.2.1 Modelling the Solar Boat

Our model is constructed around the physical formulas [25] provided in section 3. These formulas give us the ability to calculate the drag of the boat when sailing normally.

We did not implement the entire model, making only use of the following sub-models and leaving out environmental variables (weather, bridges, etc):

- Foil Drag, the drag caused by the submersed hydrofoil (section 3.1.1)
- Boat elevation, the influence of transitioning from hullborne to foilborne on the final drag (section 3.1.2)
- Hull Drag (Water), the drag of the submersed part of the hull (section 3.1.3)
- Hull Drag (Air), the drag of the remaining part of the hull through the air (section 3.1.4)
- Thrust, the influence of thrust on the velocity of the boat (section 3.1.5)

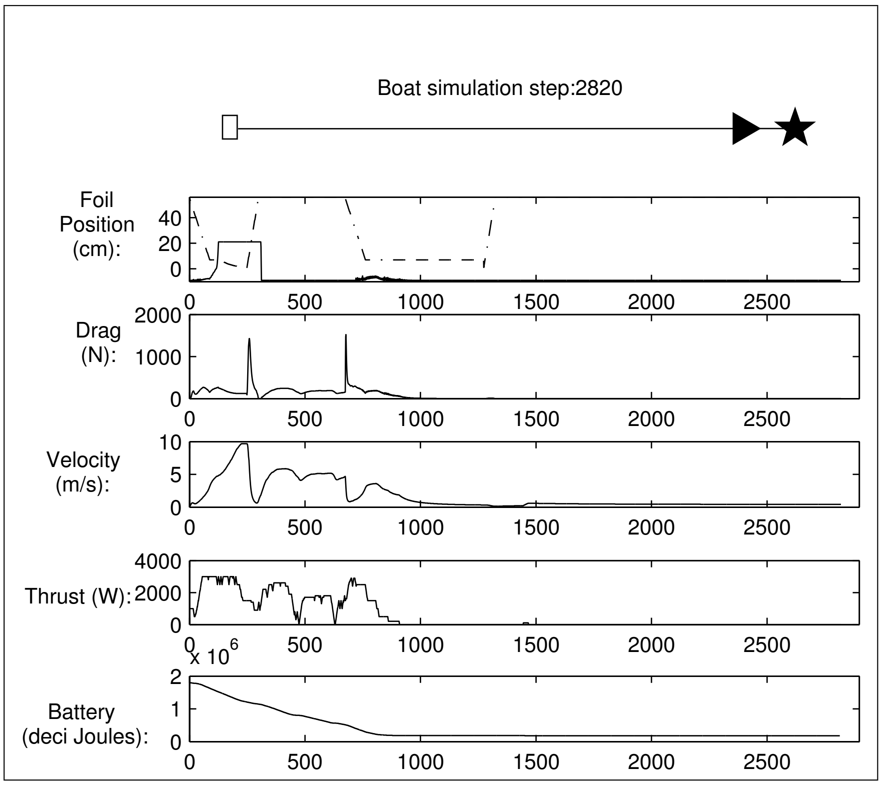


Figure 19: Simulation information displayed in a graphical interface, designed for the small race problem. The square denotes the starting point and the star the goal. The underlying graphs show several parameters, the dashed line in the Foil Position graph shows the position of the hydrofoil, the solid line shows the actual height of the boat with a level of 0 representing the water's surface.

The battery size was set at 3600000 dJ (100 Wh) for each episode. dJ is used, making it easy to calculate the energy consumption (the 0.1 factor originates from the time-step dt). The amount of thrust is given in Watt, when the battery is depleted the thrust becomes zero.

Epsilon (ϵ) is set to 0.02, meaning there is a 2% change for the algorithm to choose a random action via $\epsilon - greedy$ action selection. The value of epsilon is annealed at every successful run with a factor of 0.9999. This resulted in $\epsilon(i) = 0.02 \cdot 0.9999^i$ (with i being the number of successful finished runs), providing us with a very small value for ϵ at the end of the simulation (if all runs are successful) $\epsilon(10000) = 0.0074$.

Our policy has been allowed to perform the following actions:

- Increase Throttle: increase the throttle with 100W
- Decrease Throttle: decrease the throttle with 100W
- Deploy Foil: lowers the foil into the water
- Retract Foil: retracts the foil out of the water

- Nothing: taking no action

5.2.2 Reward Function

The focus at the beginning of this project was on the optimization of the energy consumption of the boat. Focusing mainly on using energy economically will result in unwanted behavior, as discussed in section 2.1.2. The competition that we are participating in is a competition of speed, the competitors will try to complete the race in the shortest amount of time. The amount of energy used is not a criterion in this competition. However the amount of energy that is available to the competitors is limited. Using energy wisely, not wasting it on trivial actions, will result in a better result in the race.

Designing the reward function becomes a simple matter of assigning a negative reward to the distance that the agent has to travel towards the end of the race. Given limited amounts of resources the agent will not be able to complete the race if going full throttle for the entire time, a policy has to be found that varies the amount of throttle and makes the decision to deploy (or retract) the hydrofoil in such a way that it optimizes its overall speed.

The resolution/time-step of our simulation was set at 0.1s, at each time-step the Q-values for all actions are calculated, an action is taken and the Q-values are updated towards the received reward. We are not able/allowed to directly control our boat, meaning that the driver has to interpret the outcome of our system and act accordingly. Providing the driver with an update every 0.1s puts a large mental strain on the driver, therefore we would like to penalize performing a lot of different actions within a short time interval. This is done by adding a negative reward for performing different actions shortly after one another.

A formal definition of the final reward function is given in equation 56. When only taking D_s into account the reward function provides a mapping from $D_s \rightarrow Q$, in other words, Q describes the integral of the percentile distance towards the goal with respect to time. However by adding a negative reward for an action taken shortly after one another (D_c), this description is not accurate anymore. D_c encodes the elapsed time between actions, as time between actions increases this value will tend towards zero. D_c is scaled so that it falls withing the range of $[0 < D_c < 0.1]$, since D_s falls in the range $[0 < D_s < 1$ $D_c]$ adds a small negative reward in the case of an action being taken within ϕ seconds. Taking the sum of D_s and D_c gives us a new reward function, taking into account both the distance that is has to travel, and the time elapsed between now and the last action. D_c provides a negative reward when a decision is taken within $\frac{\phi}{dt}$ seconds after the last decision proportional to the elapsed time. Making the decision to take no action is not counted as a decision.

The reward has to be normalized in such a way that it is within the scope of the output of the neural network. For example when the reward function can range from -1 up till 1 at each time-step, after n iterations the total reward r_{tot} that can be received will be $[(-1 \cdot n) \leq r_{tot} \leq (1 \cdot n)]$, given a hyperbolic tangent (equation 8) with a range between -1 and 1 the reward has to be divided by n for it not too exceed these limits (given discount factor $\gamma = 1$).

$$\begin{aligned}
D_s &= -\frac{s_{max} - s_{current}}{s_{max}} && (56)\\
&\quad \text{(with } s_{max} \text{ being the track distance,}\\
&\quad \text{and } s_{current} \text{ the current position)}\\
D_c &= -\frac{100 + (t_{decision} - t_{current})}{(\phi/dt) \cdot 10} && (57)\\
D_c &= Min(D_c, 0)\\
&\quad \text{(after } \phi \text{ seconds reward is limited to 0}\\
&\quad\ t_{current} \text{ is the current timestep}\\
&\quad \text{and } t_{decision} \text{ being the timestep of the last decision)}\\
D_{total} &= D_s + D_c
\end{aligned}$$

5.2.3 Neural Network

The neural network acts as a function approximator, estimating the SARSA value for any given state, action pairs. Since we have not defined states, the network takes input variables from which the state can be deduced. The weights of the network are randomly initialized between -0.15 and 0.15. Small weight-values simulate the behavior of a neural network at the beginning of the experiment, giving it the ability to transcend the linear behavior of the network towards non-linear when this is required. The network takes six inputs:

- Velocity (m/s)
- Position (m)
- Throttle (W)
- Drag (N)
- Battery Energy (dJ)
- Time elapsed after last action (0.1s)

The Inputs of the neural network are normalized. This ensures that they have about the same numerical range, bypassing the problem that the weights of the network will focus on the large input values at the beginning of the experiment, ignoring the smaller values which might also contain vital information. The maximum velocity that can be achieved by the boat is estimated, we defined the range a little bit larger and set the maximum velocity to $20m/s$. The track distance varies per track for the final model, but in our sub-experiment we only simulate a fixed linear track with a distance of 500m. $MaxThrust$ is set at 3000W. The maximum drag that can be perceived is set at 5000, this value is empirically defined, mainly based on the drag that is created while deploying the hydrofoil into the water. The battery capacity is set at 3600000 deci Joules, this is the same as 100Wh. 100Wh is not enough for the boat to reach the

Table 3: Neural Network Inputs

Input:	Normalization factor
Velocity (m/s)	$Max\,Velocity^{-1}$
Position (m)	$Track\,Distance^{-1}$
Throttle (W)	$Max\,Thrust^{-1}$
Drag (N)	$Mac\,Drag^{-1}$
Battery Energy (deci Joules)	3600000^{-1}
Time elapsed after the last action (0.1s)	100^{-1}

finish line within 5000 steps (500 seconds) with the throttle set at maximum. The time between the last action and the current time-step is also taken as an input, enabling the network to deal with the action frequency. After ten seconds there is no negative reward anymore, meaning that normalization can be done by performing a division by $10 * dt = 100$ steps. All normalization factors are shown in table 3.

No energy from the sun is modeled, the only source of energy is the battery in this experiment.

Given the six inputs for simulated sensors, an additional five different input nodes are required for the binary encoding of the actions. This results in a total of eleven input nodes.

For the output layer we have chosen one single (tangent hyperbolic) output, which provides us with the estimated Q-value for each encoded action. Because the output is non-linear the actual size of the Q-value is not estimated, only its size with respect to all other state, action pairs is estimated. Estimating the actual Q-value (with a linear output node) is useful when the Q-value says something about the environment. For example if the reward value only gives a negative reward for each timestep, the Q-value will encode the time required for the agent to reach the finish line. In our case the reward function is more complex, no such translation can be made; so the actual size of the Q-value does not matter for us. A pilot experiment also showed that a network with a non-linear output converges faster to a good solution than a network with a linear output node, for this reason we have chosen to use the tangent hyperbolic activation function for our output node. More information about the network topology can be found in section 2.5.1.

For the neural network settings we coarsely searched the parameter-space, this resulted in the following parameters:

$$\begin{aligned} n_{sensor-inputs} &= 6 \\ n_{action-inputs} &= 5 \\ n_{hidden} &= 20 \\ n_{output} &= 1 \\ learningrate\ \eta &= 0.03 \\ momentum &= 0.1 \end{aligned} \tag{58}$$

5.2.4 SARSA

The discount factor γ is set at 1, meaning that no discounting is taking place. This value is chosen because we are situated in a world with a finite horizon, the number of steps for the simulation is limited and therefore the reward is bounded.

$$\begin{aligned} \gamma &= 1 \\ \epsilon\,(i) &= 0.02 \cdot 0.9999^{i} \end{aligned} \tag{59}$$

5.2.5 NFQ

For the Neural Fitted Q Iteration algorithm we used the same neural network as for the NN SARSA algorithm, however in batch mode, as is typical for NFQ (see also section 2.6). We experimented with the setting for k (the number of epochs that is used to gather data). The bigger the size of batch, the slower the system becomes. This tradeoff resulted in a value of $k = 5$, still allowing fast computations and a batch size that is reasonably large for the network to train on (multiplying with the maximum number of steps (5000), results in a maximum of 25000 observations in our batch). We train the network after each k^{th} epoch until the training error is less than 0.001.

5.3 Results

For the final experiment we made a comparison between the Random selection, NN SARSA and NFQ algorithm.

Environmental Settings:

$$\begin{aligned} E\_battery &= 36000000\,deci\,Joules \\ Max\,Thrust &= 3000\,Watt \\ Max\,Distance &= 500\,m \\ dt &= 0.1\,s \\ Max\,Steps &= 5000 \\ epochs_{training} &= 10000 \end{aligned} \tag{60}$$

Experiment 1: Random action selection: We took a large amount of 10000 samples for the random action selection algorithm, this amount of samples is large with respect to the NN SARSA algorithm (which has twenty samples), however this provides us with a better estimation of the performance of a random agent (because of the difficulty of this task it is hard for random action selection to finish). The amount of steps required for the random action selection was 4819.1 with a standard deviation of 743.8 steps. The fastest instance of the Random action selection algorithm finished the race in 631 steps (63.1 seconds).

Experiment 2: SARSA with neural network (NN SARSA): This experiment requires quite some computation time, each training epoch (of 10000 episodes) took between 8 and 20 hours to complete (the better the performance, the quicker the finish is reached, this therefore increases the speed in which all training epochs are completed). Twenty samples were taken, resulting in a mean performance of 929.1 with a standard deviation of 273.7 steps. The fastest instance of the NN SARSA algorithm finished the race in 521 steps (52.1 seconds).

Experiment 3: Neural Fitted Q Interation (NFQ): This experiment is slightly quicker than the NN SARSA algorithm because batch-training is used (offline). Twenty samples were taken, resulting in a mean performance of 665.2 with a standard deviation of 152.3 steps. The fastest instance of the NFQ algorithm finished the race in 517 steps (51.7 seconds).

The results are shown in figure 20, the best algorithm finished the race in 51.7 seconds (an instance of the NFQ algorithm).

Because of the different sample sizes an F-test for testing the equality of variance was used, this proved that the variances are significantly different, F(999,19) = 7.38, $p < 0.01$. Because of this difference Welch's t test was used to test the difference between the two algorithms. This test showed a significant difference in performance between the Random and the NN SARSA algorithm, t(10018)=23.36, $p < 0.01$.

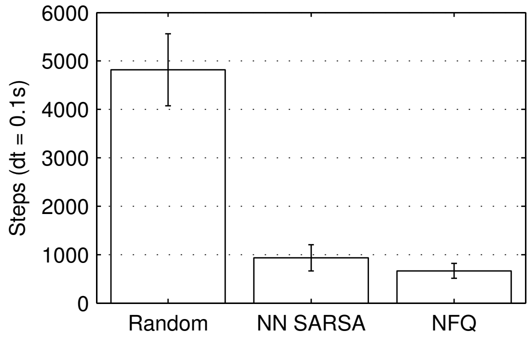
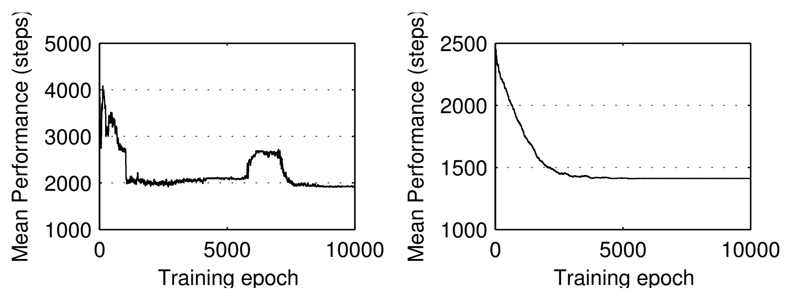


Figure 20: Small race results (500m), errorbars denote the standard deviations. Lower equals a better performance (less steps required to reach the goal). NN SARSA outperforms the Random algorithm. The NFQ algorithm outperformed both the NN SARSA and the Random algorithm ($p < 0.01$).



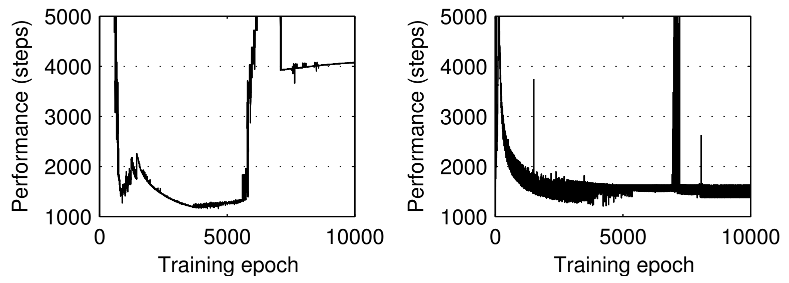
(a) Average (smoothed) convergence of the NN SARSA algorithm
(b) Average (smoothed) convergence of the NFQ algorithm

Figure 21: Convergence of both the NN SARSA and the NFQ algorithm, NFQ displays better (more stable) convergence.

Finally we compared the NN SARSA algorithm with the NFQ algorithm using the student t-test. This test showed a significant difference in performance between the NN SARSA and the NFQ algorithm, t(38)=3.86, $p < 0.01$.

5.4 Discussion

Figure 21.a shows the convergence for the NN SARSA algorithm, on most cases the algorithm converges nicely. However as seen in the graph sometimes a discontinuity in the decreasing performance is observed. In this specific case this periodic decrease in performance is caused by two runs (figure 22.a shows a decrease in performance after some time. Figure 22.a displays how after about 6000 training epochs the algorithm 'loses focus'. This decrease in performance is caused by randomly selecting an action that forces the algorithm towards a less optimal decision sequence, resulting in a decrease in performance. The network regains performance after about 1000 epochs but still performs worse than before. It however does not recover from this (within the limit of 10000 training epochs), a long period of bad performance also trains the network weights



(a) Algorithm looses its focus and is **not** able to recover quickly
(b) Algorithm looses its focus but **is** able to recover quickly

Figure 22: Two cases in which the NN SARSA algorithm loses focus on the simple race task.

towards this bad performance. This might overwrite knowledge gathered in previous epochs with a good performance, meaning that the longer the period of decreased performance, the longer the network takes before it recovers (generally spoken). Figure 22.b shows a training run that loses focus after 7000 training epochs, but quickly recovers. This short decrease in performance did not overwrite the network weights to forget the SARSA values that result in better performance. Both cases show how exploration in the system's environment is taking place, in most cases this exploration leads to a better performance.

Figure 21.b shows the convergence for the NFQ algorithm, the algorithm converges smoothly. It also shows that after 5000 epochs the algorithm shows no further improvement (on average). Therefore we can reduce the number of epochs used to learn a policy, a reduction in epochs means a significant reduction in time for the entire model. The NFQ algorithm showed the best converging behavior and is also faster than NN SARSA algorithm. NFQ is therefore to be preferred over NN SARSA.

All optimized policies displayed the use of hydrofoils to increase overall speed, the onset time at which the hydrofoils are deployed varied between policies; however the faster the hydrofoil was deployed, the faster the overall time. Figure 23 shows a test run of one of the learned NFQ policies without random action selection, it can clearly be seen that it reduces speed (the boat in the

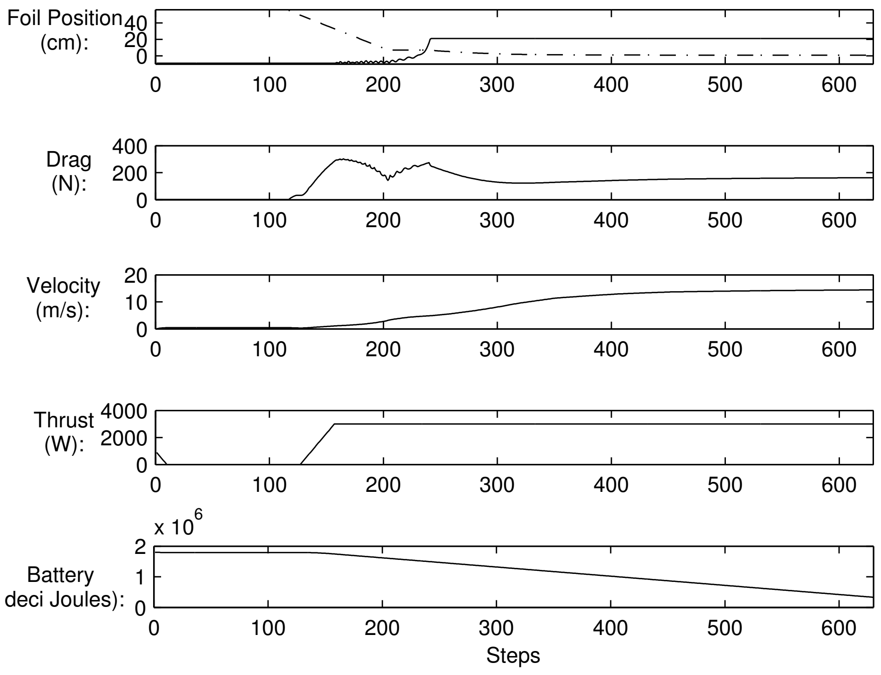


Figure 23: Simulation run of an instance of the NFQ algorithm (with $\epsilon = 0$). The top plot shows the position of the hydrofoil (dashed line), and the position of the boat (solid line) with respect to the water level. The policy learns do decelerate in the beginning ($t = 10$), deploy the hydrofoil ($t = 110$) and race to the finish line efficiently ($t = 630$).

simulation has an initial throttle setting), deploys the hydrofoil and during the deployment accelerates to full throttle, lifting the boat out of the water. It then reaches the finish line in 630 steps (63 seconds). In situations where the hydrofoil is not used the boat is not able to finish, stalling (because the battery is depleted) before the finish line.

The small race experiment is still a simplification of the power consumption optimization of a solar powered boat, the real problem. The small race experiment shows us that both the Artificial Neural Network SARSA and the Neural Fitted Q Iteration algorithm can be used to optimize the speed in a race situation, learning to make use of the hydrofoils and increase overall speed. The good results encourages us to move on to the next section where we look at optimizing the power and overall speed for the entire race (not just a small straight track).

6 Making Optimal Use of Available Power And Optimizing Overall Speed in a Solar Boat Race

In the previous experiment we considered a linear track on which a race was held. The amount of energy was not enough for the boat to reach the finish-line full throttle. Methods were found that learn a policy which optimizes the overall-time of a small race, changing from hullborne to foilborne when this is necessary. We now upgrade our problem, increasing the complexity by introducing a non-linear track and an expanded environment (with sun and events along the way (e.g. bridges which cannot be passed while hydrofoiling). For this experiment we will not consider the NFQ algorithm, this algorithm was added later on in the project. We unfortunately did not have enough time to incorporate (and train) it in our final model.

6.1 General Problem Description

The Frisian Solar Challenge 2010 follows the 'Elfstedentocht', the famous ice speed skating tour. This tour has a total length of 220 km, divided into six stages (going through canals, rivers and lakes). The goal is to complete the race as fast as possible with a limited amount of power (1kH fully charged battery and maximally 1750W coming from the solar panels).

How can we apply what is learned in the previous experiments to optimize the difficult task of the solar-race?

6.2 Implementation

For the solar-race we implemented a Graphic User Interface, so we could view the different variables change over time and analyze the boat's behavior. The interface is displayed in figure 24.

6.2.1 Modeling the Solar Race

The model of the solar boat is similar to section 5.2.1, this model describes the foil drag, elevation of the boat when hydrofoiling, drag of both air and water and the boat's propulsion (with a timestep dt = 0.1s). We extended our model by implementing the following items:

- Stages, GPS way-points for all six stages are used to simulate the track, including the starting times (section 3.1.10).
- Cornering, the behavior of the boat in corners is taken into account, reducing the speed of the boat in corners that are too sharp (section 3.1.6
- Bridges, all the bridges of the track are recorded, included in their information is the width and height of each individual bridge.
- Sun, given the time of day and position on the earth we can calculate how much power is to be received from the sun (section 3.1.8)

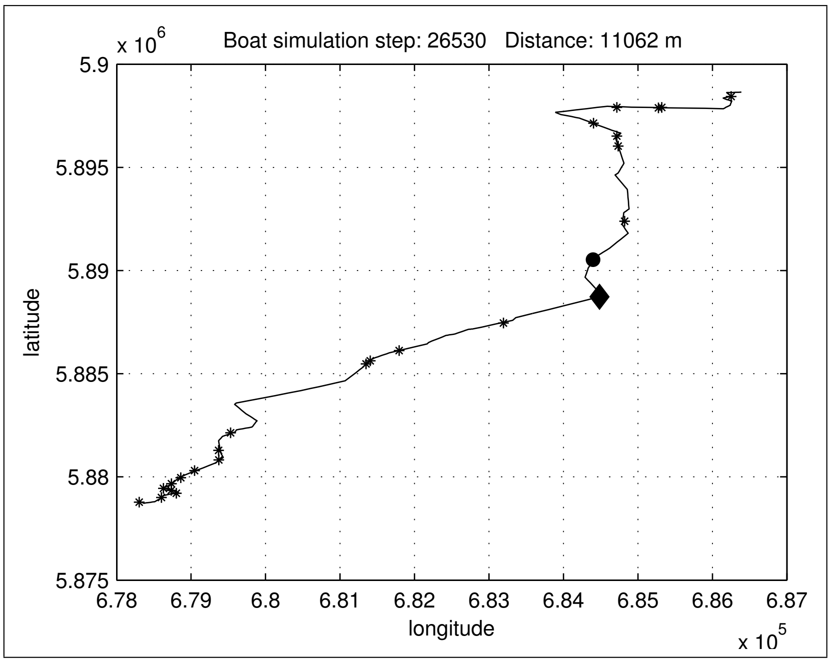


Figure 24: A schematical description of the solar race, displayed in a graphical interface developed for this problem. The circle shows the location of the boat, the in the actual simulation the color of the circle can be changed to show if the boat is foiling or not. Crosses show all events along the way and the diamond shows the nearest event.

- Weather, the KNMI dataset provided us scenarios of how the weather from the last ten years. Scenarios are selected from the KNMI dataset for each simulation epoch according to PEGASUS [18].
- Shadows, although not recorded due to restricted time, it is possible to import locations where objects next to the track could cast a shadow on the boat, resulting in better energy prediction (section 3.1.9). Due to limited time the GPS coordinates of large objects casting shadows on the track could not be added. For computational purposes we therefore skipped all relating calculations.

The maximum capacity of the battery was set at 36000000 dJ (1kwh). The amount of power going to the motor can maximally be 3000W.

Instead of learning an action sequence that leads to a certain throttle-position, we have simplified our model so that the policy is able to directly choose out of three throttle positions. This is a rather large simplification, however it gave our policy the opportunity to quickly sail with the boat (not spending expensive computation time on learning how to control the throttle).

Our policy has been allowed to perform the following actions:

- Low Throttle: setting the throttle at 1000W
- Medium Throttle: setting the throttle at 2000W
- High Throttle: setting the throttle at 3000W
- Deploy Foil: lowers the foil into the water
- Retract Foil: retracts the foil out of the water
- Nothing: taking no action

Epsilon (ϵ) is set to 0.02, meaning there is a 2% probability for the algorithm to choose a random action via $\epsilon - greedy$ action selection. The value of epsilon is annealed at every successful run with a factor of 0.999. This resulted in $\epsilon(i) = 0.02 \cdot 0.999^i$ (with i being the number of successful finished runs), providing us with a very small value for ϵ at the end of the simulation

6.2.2 Reward Function

For the reward function we choose the reward function described in section 5.2.2. This function resulted in a policy displaying the behavior that we desired, optimizing speed and reducing the number of actions that the pilot has to take. Therefore no changes to the reward function had to be made.

6.2.3 Neural Network

For the neural network the same network as in section 5.2.3 is used, acting as the function approximator which estimates the SARSA value. The weights of the network are randomly initialized between -0.15 and 0.15.
The Network takes as inputs:

- Velocity (m/s)
- Position (m)
- Throttle (W)
- Drag (N)
- Battery Energy (dJ)
- Solar Energy (W)
- Amount of times switched between foilborne and hullborne (N_{switches})
- Solar Radiance at this timestep (J/cm^2)
- Solar Radiance after one hour (J/cm^2)

Table 4: Neural Network Inputs

Input:	Normalization factor
Velocity (m/s)	$Max\,Velocity^{-1}$
Position (m)	$Track\,Distance^{-1}$
Throttle (W)	$Max\,Thrust^{-1}$
Drag (N)	$Mac\,Drag^{-1}$
Battery Energy (dJ)	36000000^{-1}
Solar Energy (W)	2000^{-1}
N_{switches}	1000^{-1}
Solar Radiance at this timestep (J/cm^2)	max radiance for scenario^{-1}
Solar Radiance after one hour (J/cm^2)	max radiance for scenario^{-1}

The Inputs of the neural network are normalized. This ensures that they have about the same numerical range. Table 4 shows the normalization factors for our inputs (most normalization factors are inherited from section 5.2.3 (table 3)). More information about the network topology can be found in section 2.5.1

Given the nine inputs for simulated sensors, an additional six different input nodes are required for the binary encoding of the actions. This results in a total of fifteen input nodes. For the output layer we have chosen one single (tangent hyperbolic) output. For the neural network we coarsely searched the parameter-space, this resulted in the following parameters:

$$\begin{aligned} n_{sensor-inputs} &= 9 \\ n_{action-inputs} &= 6 \\ n_{hidden} &= 24 \\ n_{output} &= 1 \\ learningrate\ \eta &= 0.03 \\ momentum &= 0.1 \end{aligned} \tag{61}$$

6.2.4 SARSA

The discount factor γ is set at 1, meaning that no discounting is taking place. This value is chosen because we are situated in a world with a finite horizon, the number of steps for the simulation is limited and therefore the reward is bounded.

$$\begin{aligned} \gamma &= 1 \\ \epsilon\left(i\right) &= 0.02 \cdot 0.9999^{i} \end{aligned} \tag{62}$$

6.3 Experimental Setup

Instead of running a large number of experiments (which would take a lot of time with the improved model) we have chosen to focus on learning **one** good policy. Since our policy will be used in a race situation we are not interested in developing a n algorithm that averagely finds a good policy. We are more interested in finding one single good policy that can be used in our race. We are posed with a design choice when learning our policy, we can either optimize one policy for each stage in the race or optimize one policy for all stages. The goal of our experiments is to find out whether there is a difference in performance between optimizing a policy for each singe stage (a specific policy), or optimizing it for the entire race (a generalizing policy). To figure this out we performed the following two experiments:

- Specific Policy Algorithm (SPA)
  - Optimized for one single track
  - Weather scenario chosen according to PEGASUS [18].
  - Starts with six different experiments
  - After 1500 training epochs the best performing experiment/policy is chosen and optimized until 5000 epochs
- Generalizing Policy Algorithm (GPA)
  - Optimized for all tracks
  - Weather scenario and track number chosen according to PEGASUS [18].
  - Starts with six different experiments
  - After 1500 training epochs the best performing experiment/policy is chosen and optimized until 5000 epochs

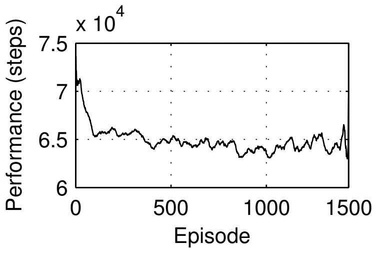
After the experiments we will analyze the difference in performance between the two policies on one particular track. We hypothesized that the SPA would outperform the GPA.

6.4 Results

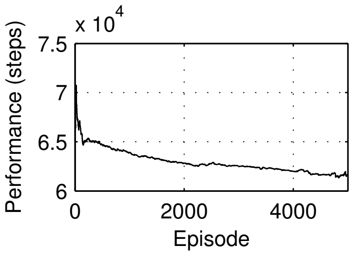
One episode for our model took an average of 177.5 seconds for the first 1500 episodes on six 2.52Ghz dual core PCs with each core dedicated to one MATLAB session. This means that running the initial experiment of 1500 episodes takes about 74 hours.

We start out by looking at the results for the Specific Policy Algorithm (SPA). Figure 25 shows the average convergence of all six experiments for the first 1500 episodes, the best of these six experiments was used for our analysis (and is allowed to keep on training for another 3500 epochs).

The convergence of the General Policy Algorithm (GPA) does not tell us much, because each episode a different scenario/track is chosen with different length and other parameters, performance will vary significantly between tracks. Instead of looking at the convergence we look at the mean performance of all six experiments. The experiment with the best mean performance after 1500 episodes was chosen to keep on training for another 3500 epochs (5000 in total).



(a) Average convergence of the six initial experiments for the first 1500 episodes



(b) Convergence of the best experiment for 5000 episodes

Figure 25: Convergence of the SPA algorithm. Smoothed with a moving average filter for clarity. Lower number of steps equals better performance.

After the policies were trained we took 100 samples for both the GPA and SPA on one single track (track 1 between Leeuwarden and Sneek). The results are shown in figure 26. A student t-test was conducted. We set up the null hypothesis to state that the samples come from populations with equal means. The t-test showed no significant difference in performance between the SPA and the GPA ($t(198) = 3.8171, p < 0.01$). All policies finished the race with an average speed of about 14.5km/h, the fastest policy had an average speed of 15.5km/h.

We also tested the SPA and GPA on all other tracks, these results are shown in table 5. Ten samples for each tracks were taken, the performance shown in the table is the mean performance over these ten instances. On all tracks the SPA finishes faster than the GPA (except on track # 3 where neither algorithm was able to reach the finish line successfully).

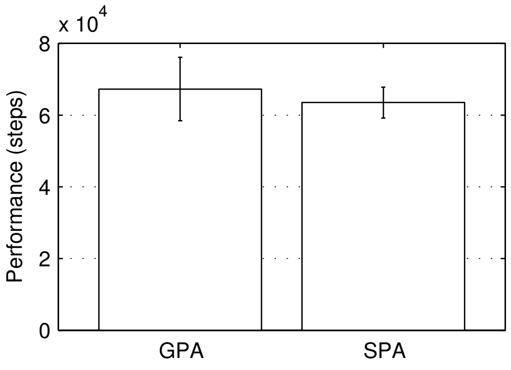


Figure 26: Results for the GPA and SPA, errorbars denote the standard deviations. Lower equals a better performance (less steps/time required to reach the finish).

Track#	SPA Performance (steps)	GPA Performance (steps)
1	61574.7	63817.8
2	44939.6	52211.5
3	108000.0	108000.0
4	43789.0	47510.9
5	27713.0	32878.3
6	72817.9	84213.7
7	30188.0	39287.4
8	53779.2	58821.0
9	12538.0	12659.0

Table 5: Average performance of the SPA and GPA algorithms on all tracks

6.5 Discussion

The SPA performed equally well as the GPA, this goes against our hypothesis that the SPA would outperform the GPA. The number of training epochs (5000) however is still low, we did not have time to run the model any longer and therefore made our comparison at 5000 epochs. Nevertheless the similar performance tells us a few things. The main thing is that it is apparently not necessary to train nine different SPA's (given that there are nine different stages divided over five days), but one could suffice with only one policy (SPA or GPA). The GPA takes considerable more time to train than the SPA because every epoch a different track has to be loaded into the memory. In practice the difference was about one day of training time for the SPA against a total of five days of training for the GPA. The longer training time is a downside and given that the SPA always finishes faster we can state that training on one single stage is preferred. The finding that the SPA always finished faster is interesting, this means that we do not have to spend a lot of time switching between tracks in the simulation. But more importantly this shows that training for a longer time on one single environment still enables the policy to generalize sufficiently to deal with unseen environments (unlike the GPA). This generalization is an important finding, showing the relevance of using neural networks in stead of tabular approaches.

Given that there was no significant difference between the GPA and the SPA on track 1 indicates that the policies do not get the right input variables. Training on one specific track should increase the performance of a SPA on that particular track with respect to the GPA. Further research should be done, investigating which sensory information is of use an which are not. The average speed of 14.5km/h might not seem large, however the computer simulation of the race was demanding. For example all 157 bridges were kept in the simulation, while actually not all are too low (these are only a handfull). Furthermore the simulations were performed with an estimated model of the solarboat, one could not say that this would be the actual speed of the solarboat since there was no time to validate our algorithmic model. The convergence of the performance of the policies however do show that our policies optimize the overall speed. Given enough training time the speed of the boat could increase even further.

7 Discussion

In this paper we asked ourselves the question: "Can we use reinforcement learning to make optimal use of available power, optimizing the overall speed of a solar powered boat?" and in more detail: "Which reinforcement learning methods can be used for this?". We have started out with the mountain car example problem (section 4), showing a significant performance improvement by using neural networks for estimating the SARSA value over the standard tabular approach. The mountain car problem is too simple, it can easily be solved by neural networks.

Comparing our algorithms on a more complex problem gave us a better understanding about the strength of using function approximators instead of tabular approaches. For this reason we moved on to the problem of making optimal use of available power in a small solar powered boat race (section 5). In this race (of 500 meters) a limited amount of energy is available, this amount is not sufficient to reach the finish line without using hydrofoils. The Neural Fitted Q Iteration (NFQ) algorithm performed best at this problem, followed by Neural Network SARSA (NN SARSA). NFQ also displayed better converging behavior (faster and more reliable) than NN SARSA. The learned policies maximized their reward functions, resulting in controllers that optimized the overall speed of the solar boat. Even though the NFQ algorithm is better than NN SARSA, we were not able to use this in the final problem (making optimal use of available power in a solar boat race, see also section 6). The addition of the NFQ algorithm was done at the end of this project, we unfortunately did not have any more time to implement it for the final model.

The final problem consisted of a model of the entire race, taking into account environmental parameters (like weather and corners). This problem is more difficult than the previous, requiring considerably more computation time. We tested two different approaches. A Generalizing Policy Algorithm (GPA) was trained on all tracks, while a Specific Policy Algorithm (SPA) was trained on one single track. The results showed no significant difference between the two. However the SPA always finished faster than the GPA, indicating that the SPA is able to generalize successfully. This showed the neural network's ability for dealing with unseen world and system states. It also tells us that we do not need to train a different policy for each track, one single (generalizing) policy is sufficient. This saves a lot of training-time. More research however has to be done to investigate which input variables are relevant for the network to act optimally. The results of the NFQ algorithm on the small race are promising. The results suggest that an implementation of the NFQ algorithm for the final race will increase the performance of the boat even further.

The NFQ algorithm trains offline using batches, in contrast to the NN SARSA algorithm with learns to predict the SARSA-value online. This can be a downside when a controller is needed on demand (instead of waiting for the NFQ algorithm to finish its current batch, the NN SARSA algorithm can be used directly). If time is limited the NN SARSA is the best option. When having sufficient training time, NFQ will increase the speed of the boat even further.

MATLAB has provided us with a nice environment in which we could quickly change/adapt our model and visually see our result. During the design process MATLAB has decreased the time needed to set up our framework. Computation time is quite a big issue in these experiments. This has several downsides, on the one hand there is feedback. If it takes a few days before the results of an experiment are known an iterative design process takes a lot of time. In the case of a bug this also sets us as researchers back a few days each time this occurs. On the other hand it is preferred that computing a new policy can be done quickly, if changes to the model are made during the race (e.g. mechanical changes which influence the performance, or additional environmental information is added) a new policy has to be computed as fast as possible. This is not possible with the current implementation. Speed can be increased by translating our project from MATLAB to a C++ implementation which can be done now that our model has proven to be successful.

In future work the mathematical model has to be matched/confirmed with the real boat. A series of towing test should be performed, providing information about the drag of both the hull and the foil at different speeds. Also the electrical and mechanical losses should be mapped so we could make a better estimation of the energy management of the boat. Another point is that additional environmental factors should be incorporated in our model, such as wind, opponents, water depth. Increasing the detail of our simulation results in a better energy prediction. The better the energy prediction, the better the learned policy will perform in the simulation; but also in the real world. Machine learning techniques can be used for learning a model of the solar boat if the mathematical approach does not succeed.

Besides the research which is described in this paper, this project shows how research, design, development and implementation can go hand in hand. One example of this is that the findings of our algorithmic model were used during the design process, providing the developers information about drag and stability. This information was used in this case to determine the size, location and design of both the hydrofoils and propeller.

Machine learning can also be used on several other interesting projects for the solar boat. One example is dynamic steering, with increasing speeds and complexity of the controls steering becomes increasingly difficult. Machine learning can be used to learn controllers that steer the hydrofoils more efficiently, possibly even allowing faster cornering under the influence of wind and other factors. Another example is path planning, determining not only how fast to go, but also giving the pilot advise where to go, taking into account environmental parameters and possibly even opponents.

This project has shown that we are able to make optimal use of available power while optimizing the overall speed by using reinforcement learning. This could not only be useful in solar-race applications but also in other fields like robotics (acting optimally with limited energy) and durable energy (e.g. using energy buffers to overcome periods with low amounts of energy). More research in this field could provide us with even more ways of dealing with decision making in situations were the amount of energy is limited and unpredictable, as it is with most sustainable energy sources.

References

[1] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *In Advances in Neural Information Processing Systems 19*, page 2007. MIT Press, 2007.

[2] C.W. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

[3] J. Andrew (Drew) Bagnell. Learning decisions: Robustness, uncertainty, and approximation. Master's thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2004.

[4] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6:679–684, 1957.

[5] Arthur E. Bryson and Yu-Chi Ho. *Applied optimal control : optimization, estimation, and control.* Hemisphere Pub. Corp. ; distributed by Halsted Press, Washington : New York :, rev. printing. edition, 1975.

[6] Rozaida Ghazali, Abir Jaafar Hussain, Nazri Mohd Nawi, and Baharuddin Mohamad. Non-stationary and stationary prediction of financial time series using dynamic ridge polynomial neural network. *Neurocomput.*, 72(10-12):2359–2367, 2009.

[7] Chun gui Li, Meng Wang, Shu-Hong Yang, and Zeng fang Zhang. Urban traffic signal learning control using sarsa algorithm based on adaptive rbf network. *Measuring Technology and Mechatronics Automation, International Conference on*, 3:658–661, 2009.

[8] Stephan Ten Hagen and Ben Kröse. Q-learning for systems with continuous state and action spaces. In *In BENELEARN 2000, 10th Belgian-Dutch Conference on Machine Learning*, 2000.

[9] Verena Heidrich-Meisner and Christian Igel. Variable metric reinforcement learning methods applied to the noisy mountain car problem. pages 136–150, 2008.

[10] R. A. Howard. Dynamic programming and markov processes. 1960.

[11] KNMI Koninklijk Nederlands Meteorologisch Instituut. http://www.knmi.nl. http://www.knmi.nl, June 2010.

[12] Javaprop. http://www.mh-aerotools.de/airfoils/javaprop.htm. http://www.mh-aerotools.de/airfoils/javaprop.htm, June 2010.

[13] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[14] Jose Antonio Martin. Matlab sarsa implementation of the mountain car problem.

[15] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. pages 15–27, 1988.

[16] Bruce McLeod. The evolition of moth main hydrofoils. http://www.teknologika.com/mothblog/the-evolution-of-moth-main-hydrofoils/, July 2010.

[17] Jamshid Nazari and Okan K. Ersoy. Implementation of back-propagation neural networks with matlab. *Electrical and Computer Engineering*, ECE Technical Reports, september 1992.

[18] Andrew Y. Ng and Michael Jordan. Pegasus: A policy search method for large mdps and pomdps. In *In Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 406–415, 2000.

[19] Andrew Y. Ng, H. Jin Kim, Michael I. Jordan, and Shankar Sastry. Autonomous helicopter flight via reinfocement learning. In *In International Symposium on Experimental Robotics*. MIT Press, 2004.

[20] Christos Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Math. Oper. Res.*, 12(3):441–450, August 1987.

[21] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In *16th European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[22] D.E. Rumelhart, G.E. Hintont, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[23] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. 1994.

[24] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[25] P.A. Scherphof. Varen op hydrofoils. Bachelor Thesis, 2009.

[26] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, 1998.

[27] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, V59(4):257–263, 1988.

[28] C. Watkins. *Learning from delayed rewards*. PhD thesis, CJCH Cambridge Univ. (United Kingdom), 1989.

[29] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[30] B. Widrow. The original adaptive neural net broom-balancer. In *1987 IEEE International Symposium on Circuits and Systems*, volume 2, pages 351–357.

A Appendix

A.1 KNMI Weather Data

Table 6: KNMI Weather Data from 1987 up til now

DDVEC	Vector mean wind direction in degrees (360=north, 90=east, 180=south, 270=west, 0=calm/variable)
FHVEC	Vector mean windspeed (in 0.1 m/s)
FG	Daily mean windspeed (in 0.1 m/s)
FHX	Maximum hourly mean windspeed (in 0.1 m/s)
FHXH	Hourly division in which FHX was measured
FHN	Minimum hourly mean windspeed (in 0.1 m/s)
FHNH	Hourly division in which FHN was measured
FXX	Maximum wind gust (in 0.1 m/s)
FXXH	Hourly division in which FXX was measured
TG	Daily mean temperature in (0.1 degrees Celsius)
TN	Minimum temperature (in 0.1 degrees Celsius)
TNH	Hourly division in which TN was measured
TX	Maximum temperature (in 0.1 degrees Celsius)
TXH	Hourly division in which TX was measured
T10N	Minimum temperature at 10 cm above surface (in 0.1 degrees Celsius)
T10NH	6-hourly division in which T10N was measured
SQ	Sunshine duration (in 0.1 hour) calculated from global radiation (-1 for ¡0.05 hour)
SP	Percentage of maximum potential sunshine duration
Q	Global radiation (in J/cm2)
DR	Precipitation duration (in 0.1 hour)
RH	Daily precipitation amount (in 0.1 mm) (-1 voor/for ¡0.05 mm)
RHX	Maximum hourly precipitation amount (in 0.1 mm) (-1 for ¡0.05 mm)
RHXH	Hourly division in which RHX was measured
PG	Daily mean sea level pressure (in 0.1 hPa) calculated from 24 hourly values
PX	Maximum hourly sea level pressure (in 0.1 hPa)
PXH	Hourly division in which PX was measured
PN	Minimum hourly sea level pressure (in 0.1 hPa)
PNH	Hourly division in which PN was measured
VVN	Minimum visibility
VVNH	Hourly division in which VVN was measured
VVX	Maximum visibility
VVXH	Hourly division in which VVX was measured
NG	Mean daily cloud cover (in octants, 9=sky invisible)
UG	Daily mean relative atmospheric humidity (in percents)
UX	Maximum relative atmospheric humidity (in percents)
UXH	Hourly division in which UX was measured
UN	Minimum relative atmospheric humidity (in percents)
UNH	Hourly division in which UN was measured
EV24	Potential evapotranspiration (Makkink) (in 0.1 mm)

A.2 GPS Track information

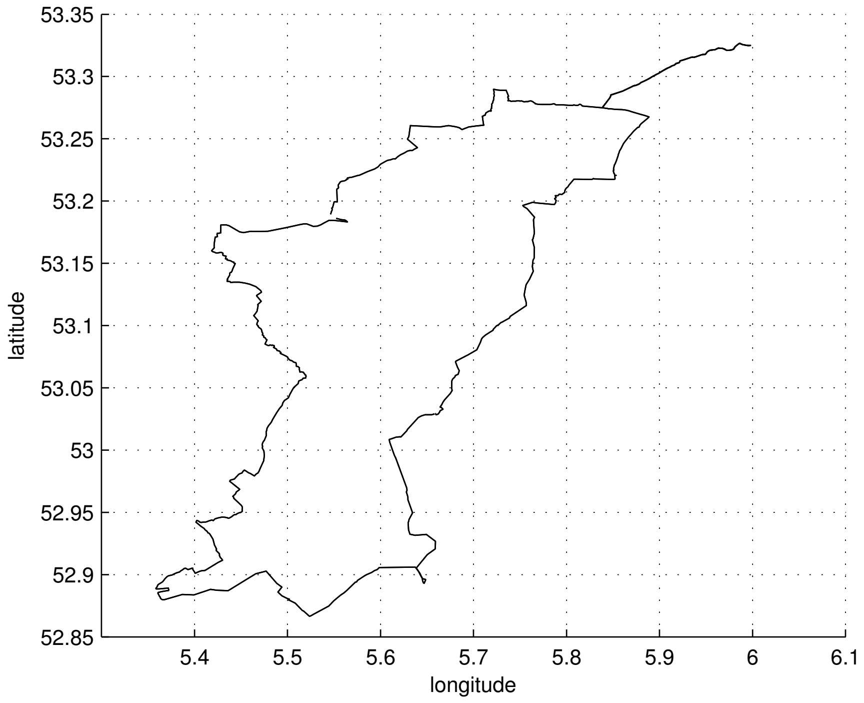


Figure 27: Entire track of the Frisian solar challenge 2010

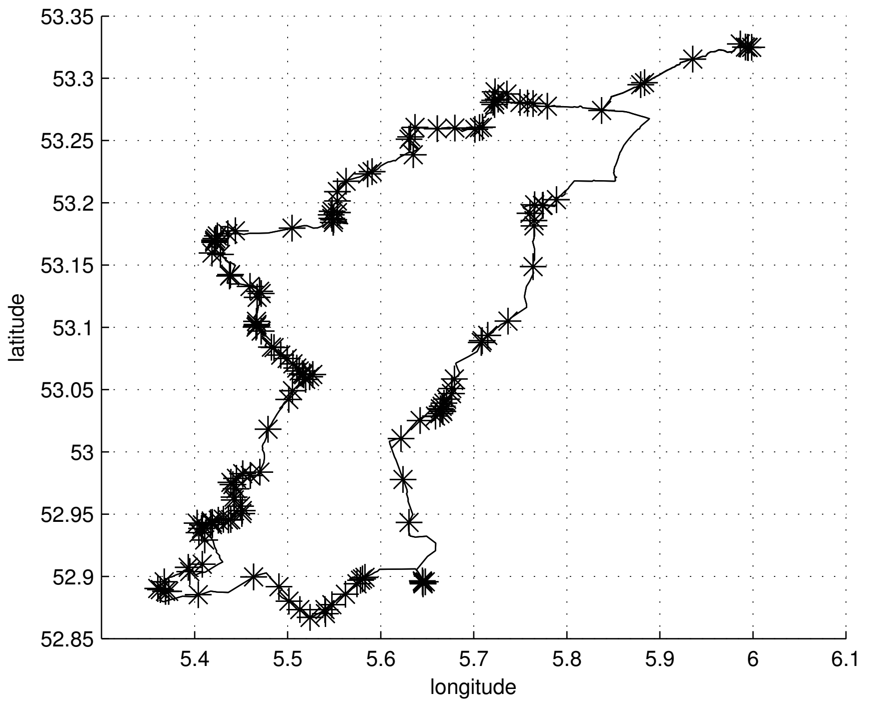


Figure 28: Entire track of the Frisian solar challenge 2010, stars denote all the bridges

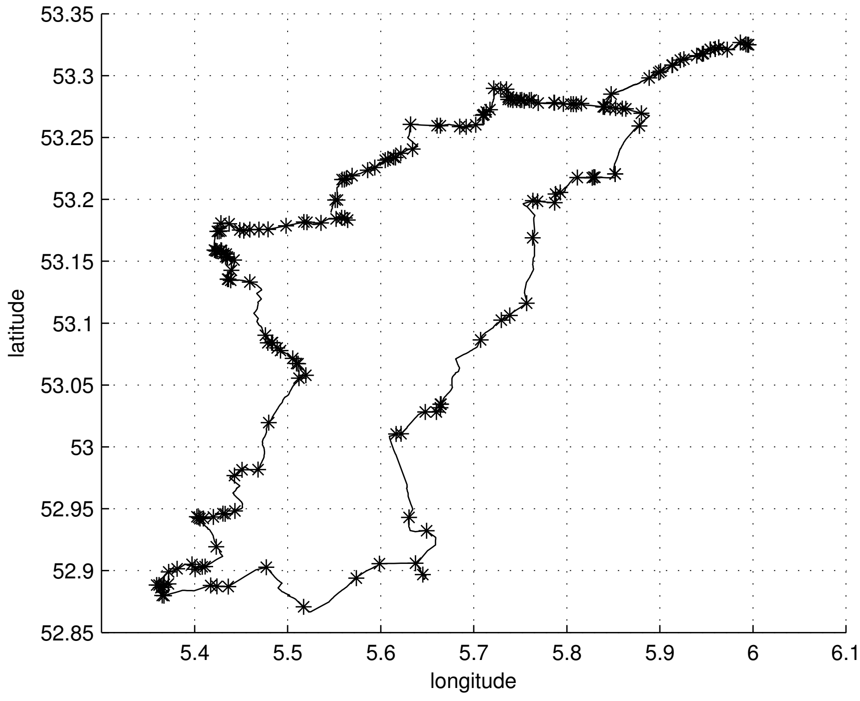
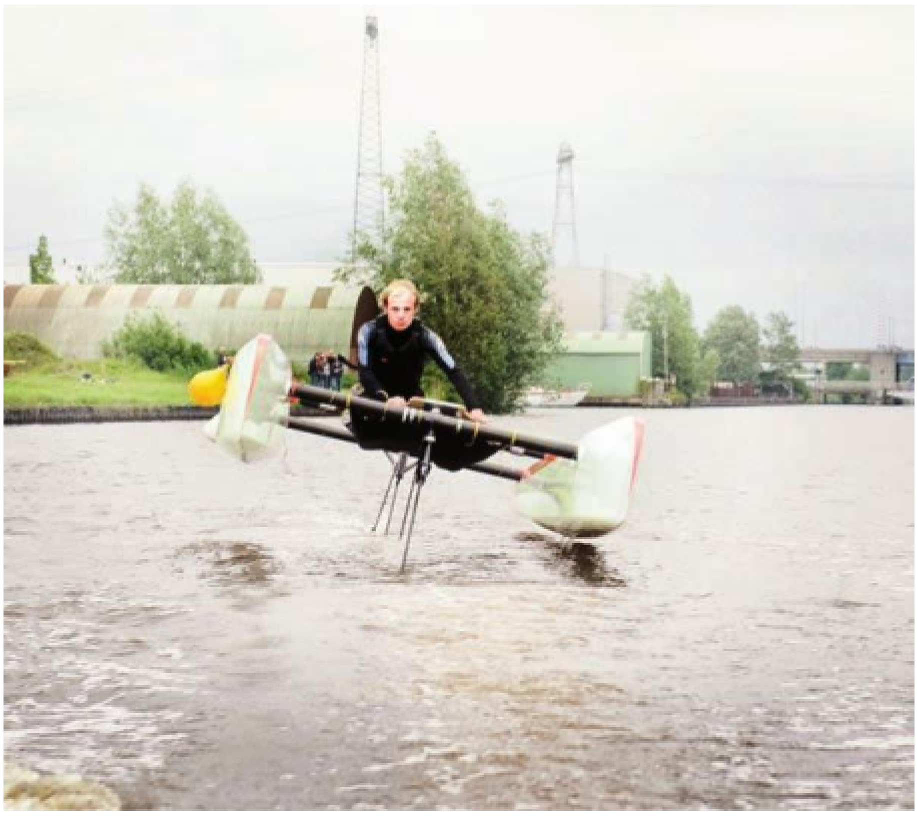


Figure 29: Entire track of the Frisian solar challenge 2010, stars denote corners with an angle between 80 and 120 degrees

A.3 The Solar Boat



Solar Team Groningen

Figure 30: Test run of the solar boat without the solar panels.

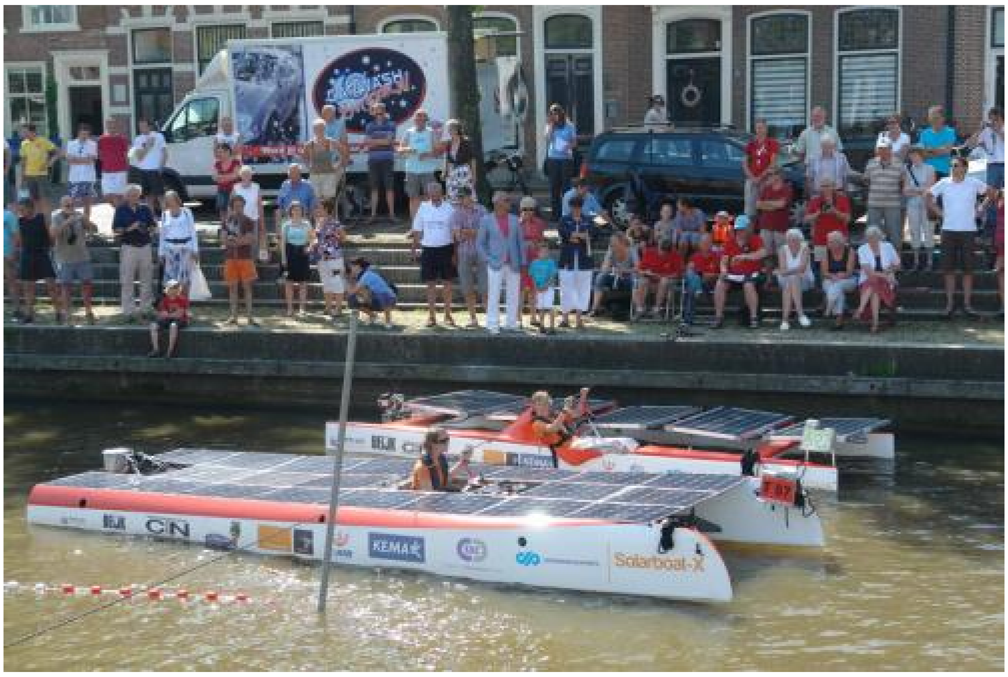


Figure 31: The finished solar boat at the starting line in Franeker of the Frisian Solar Challenge 2010.