Technical Documentation

AGV Machine Learning

December 2021

Version 1.0

Status

Reviewed	Björk, Rasmus	2021-12-13
Approved		

Project Identity

Group E-mail: carhe007@student.liu.se

Homepage: https://tsrt10.gitlab-pages.liu.se/2021/toyota/

Orderer: Anton Kullberg
Phone: Not applicable
E-mail: anton.kullberg@liu.se

Customer: Oskar Bergkvist (Toyota Material Handling)
Phone: Not applicable
E-mail: oskar.bergkvist@toyota-industries.eu

Supervisor: Hamed Haghshenas
Phone: Not applicable
E-mail: hamed.haghshenas@liu.se

Course Responsible: Daniel Axehill
Phone: Not applicable
E-mail: daniel.axehill@liu.se

Project members

Name	**Post**	**E-mail**
Carl-Hampus Hedén	Project manager	carhe007@student.liu.se
Mahdi Najafi	-	mahna987@student.liu.se
Alfred Boman	Head of design	alfbo741@student.liu.se
Adam Kagebeck	-	adaka206@student.liu.se
Kalle Blomkvist	Head of software	karla625@student.liu.se
Rasmus Björk	Head of documentation	rasbj268@student.liu.se
Viktor Ekström	Head of testing	vikek514@student.liu.se

CONTENTS

carhe007@student.liu.se

DOCUMENT HISTORY

Version	Date	Changes	Made By	Reviewed
0.1	2021-12-01	Draft	All	Rasmus Björk
0.2	2021-12-03	Feedback from project advisor	All	Rasmus Björk
0.3	2021-12-13	Feedback from project orderer	All	Rasmus Björk
1.0	2021-12-14	Final version	All	Mahdi Najafi

1 INTRODUCTION

The project group have created a machine-learning-based approach for regulating the PID parameters of a controller used to control a modelled Toyota automated guided vehicle (AGV). This technical documentation describes how the project is constructed. The AGV is built on three main modules: a simulator, a controller, and a machine learning based parameter tuner. The modules arae explained in such a way as to be understandable by basic previous knowledge of the area. This technical documentation is part of the course *TSRT10 - Reglerteknik projektkurs* at Linköping University which has been created in collaboration with Toyota Material Handling. The basis of the project is an enquiry by Toyota Material Handling to investigate whether machine learning algorithms can be used to tune a PID controller. This document contains a technical description of the system description designed in [1].

2 OVERVIEW

The product is a Toyota based AGV whose mission is to learn control parameters from various environments [2]. The technical documentation is structured in the following way. Firstly, various theoretical approaches used in the construction of the system is explained in Section 3. Note that the theory section is fairly advanced and that it might not be necessary for the user to understand the theory in order to grasp the main scope of the project and to use the system. Secondly, the implementation of the system is described starting with an overview in Section 4 followed by in depth descriptions of each major system component in Sections 5-9. Section 10 describes how the results from the system is visualized in the GUI. For an in depth user manual and description of the GUI, see [3]. Section 12 present the results from using the system followed by a discussion in Section 13 and conclusions in Section 14.

3 THEORY

In this section, the theories behind the construction of the product will be shown.

3.1 Introduction to Reinforcement Learning

Reinforcement learning can be described as a method to make an agent learn a specific task simply by trial and error without any direct human interaction. This can be done in many different ways, but the core fundamentals are always the same. An Agent interacts with an environment, in our case the environment will be a Simulink model, the agent sends out an action to the environment and takes in the states and a reward that depends on how well the task was solved. The Agent then updates its behavior to maximize its total cumulative reward. In this project the agent is built from reinforcement learning methods that utilises an actor-critic structure that will be discussed further in coming sections.

Policy
The policy is a part of the agent and works as a rule book that decides what action is to be taken in a specific state. The policy is denoted by μ if it's deterministic and π if it is stochastic.

Reward and Value
The reward function r is what that tells the agent how the current policy is performing and it depends on the current

states, the actions taken and the next states. To design a good policy it's not enough to only know the reward for a certain action, instead we want to know the total expected reward when starting in a specific state s and following policy π under a time interval. The total expected reward is captured by the value function $V^{\pi}(s)$.

States and Observations
A state s contains all the relevant information that fully describes the environment in each time step. An observation o is the observable/measurable part of the state. If the state is fully observable, state and observation are the same, otherwise the observation is just a partial description of the state and may omit some information. The agent updates the policy based on the observation and the reward.

Action
The set of all the allowed actions in a specific environment is called action space. It can be either discrete or continuous depending on the environment. The action is a decision made by the agent in order to maximize the future reward.

Learning Rate
Learning rate determines how fast the agent learns the optimal policy. It controls the step size taken at each policy update in the direction towards the optimal policy. If the learning rate is too high, the updates would become too large and the agent would jump over minima or maxima and if it is too low, the training would take much longer time.

A short description of all the key concepts in RL is given in Table 1.

Table 1: Key concepts in RL

Concept	Symbol	Description
Agent	-	The controller subject to the learning
Environment	-	The world that the agent lives in and interacts with
Action	a	Control signal(s) decided by the agent
State	s	Complete description of the state of the environment
Reward	r	A numerical value that represents how good the action taken by the agent is
Policy	μ or π	The strategy that the agent follows in order to make decisions
Observation	o	A partial description of a state

3.2 Types of Reinforcement Learning

There are two main types of reinforcement learning algorithms, model-based and model-free. What type of algorithm that can be used depends on if the agent has access to a full model of the environment. If the agent has access to a full model of the environment, meaning that the value function for starting in all possible states are known, no exploration is needed and a model-based algorithm can be used. In our case a model free method will be used and therefore we need to learn or in some way estimate the reward for a certain action. The two main ways of doing this are policy optimization and Q-learning.

A policy optimization method is usually an on-policy method, which means that each update only uses data collected while acting according to the most recent version of the policy. It is usually denoted as $\pi_{\theta}(s|a)$ and tries to optimize the parameters θ. It also uses an approximator $V_{\phi}(s)$ to estimate the value function $V^{\pi}(s)$ which it uses to update the policy π. Methods based on Q-learning are usually an off-policy method, meaning that each update can use

data collected at any point during training, the opposite of the policy optimization. It uses the approximator $Q_\theta(s,a)$ to learn the optimal action-value function $Q^*(s,a)$ and is based on the Bellman equation (see Section 3.4).

3.3 Proximal Policy Optimization (PPO)

The first machine learning method is called Proximal Policy Optimization (PPO). It is a policy gradient method that can be used for environments with either discrete or continuous action space [4]. The method is based on optimizing a parametrized policy with respect to the long-term cumulative reward. This project will use a variant of PPO called PPO-Clip. The main idea in PPO-Clip is to avoid large changes between the old policy and the new policy when performing a policy update. This is achieved by introducing a specialized clipping in the objective function which prevents the new policy to get too far from the old one [5]. The algorithm is described in detail below.

Let $R_t(\theta)$ denote the ratio between the new policy's and the old policy's PDFs at time t:

$$R_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \tag{1}$$

PPO-Clip solves an optimization problem where the loss function is formulated as follows:

$$L_t^{CLIP}(\theta) = min\Big(R_t(\theta)A_t^{\pi_{\theta_k}},\ clip(R_t(\theta),\ 1-\epsilon, 1+\epsilon)A_t^{\pi_{\theta_k}}\Big) \tag{2}$$

where ϵ is a tuning parameter and *clip* is a function that keeps the probability ratio $R_t(\theta)$ within the interval $[1-\epsilon, 1+\epsilon]$ by clipping it at the interval's upper and lower bounds. Further $A_t^{\pi_{\theta_k}}$ denotes the advantage function corresponding to policy π_{θ_k} and is defined as:

$$A_t^{\pi_{\theta_k}}(s_t,a_t) = Q^{\pi_{\theta_k}}(s_t,a_t) - V_\Phi^{\pi_{\theta_k}}(s_t) \tag{3}$$

where $V_\Phi^{\pi_{\theta_k}}(s_t)$ is the on-policy value function, parametrized by Φ, which returns the expected reward if one starts in state s_t and always acts according to policy π_{θ_k}. The on-policy action-value function $Q^{\pi_{\theta_k}}(s_t,a_t)$ returns the expected reward if one start in state s_t, takes a random action a_t and acts according to policy π_{θ_k} forever after. The problem with this approach is that two value functions are needed which increases the complexity of the algorithm. A common way to overcome this problem is to estimate the advantage function as follows:

$$\hat{A}_t^{\pi_{\theta_k}}(s_t,a_t) = r_t + \gamma V_\Phi^{\pi_{\theta_k}}(s_{t+1}) - V_\Phi^{\pi_{\theta_k}}(s_t) \tag{4}$$

where r_t is the reward in time t and γ is the discount factor which determines how much weight is put on the immediate and future rewards.

Finally, PPO-clip updates the policy by maximizing the objective function, usually by taking minibatches of Stochastic Gradient Ascent (SGA) according to:

$$\theta_{k+1} = arg\max_\theta \mathbf{E}\left(L_k^{CLIP}(\theta)\right) \tag{5}$$

where $\mathbf{E}$ denotes the empirical average over a finite batch of samples. The output of the PPO is a stochastic policy π_θ represented by its mean and standard deviation.

The implementation of the PPO-Clip is summarized in Algorithm 1

Algorithm 1 PPO-Clip [4]

Input : Initial policy parameters θ_0 and value function parameters Φ_0.
For k = 0, 1, 2,... ***do***
Run the policy $\pi_k = \pi(\theta_k)$ in the environment T times.
Compute reward-to-go $R_1, ..., R_T$.
Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$.
Update the policy by maximizing the objective function $L_k^{CLIP}(\theta)$

$$\theta_{k+1} \leftarrow arg\max_\theta \frac{1}{T} \sum_{t=0}^{t=T} L_t^{CLIP}(\theta)$$

typically via stochastic gradient ascent with Adam.
Fit the value function $V_\Phi(s)$ by regression on mean squared error:

$$\Phi_{k+1} \leftarrow arg\min_\Phi \frac{1}{T} \sum_{t=0}^{T} (V_\Phi(s_t) - R_t)^2$$

typically via some gradient descent algorithm.
Endfor

3.4 Deep Deterministic Policy Gradient (DDPG)

The second method is deep deterministic policy gradient (DDPG). This algorithm is a hybrid of Q-learning and policy gradient and is used for learning in continuous action spaces [6]. The DDPG-algorithm consists of learning two parts which are a Q-function and a policy. The DDPG agent is of the actor-critic type meaning that the algorithm simultaneously learns the policy and value function. The actor, $\mu(s|\theta^\mu)$, is a policy network that takes in observations and directly returns the action that gives the highest long-term reward, making the policy deterministic. The critic, $Q(s, a|\theta^Q)$, is a Q-value network that utilizes observations and actions as inputs and returns the expectation of long-term reward (Q-value). To learn the Q-function, it uses off-policy data and the Bellman equation shown in Equation 6. Using off-policy data means that computations can be made without considering how the data was generated. The Bellman equation describes an optimal action-value function $Q^*(s, a)$. It is also possible to describe a mean-square error equation of the Bellman equation that computes how well a neural network, $Q_\phi(s, a)$, with network parameters, ϕ, satisfies the Bellman equation as shown in Equation 7.

$$Q^*(s,a) = \underset{s' \sim P}{E}\left[r(s,a) + \gamma \underset{a'}{max} Q^*(s',a') \right] \tag{6}$$

$$L(\phi, D) = \underset{(s,a,r,s') \sim D}{E}\left[\left(Q_\phi(s,a) - \left(r + \gamma \underset{a'}{max} Q_\phi(s',a') \right) \right) \right] \tag{7}$$

where D is a set of tuples of the state, action, reward and next state (s, a, r, s') and γ is the discount factor that decides the prioritization between immediate and future rewards.

DDPG-algorithm also makes use of target networks that are time-delayed copies of their original networks. If target networks are not used the agent is susceptible to divergence since the network's equations becomes independent on the values calculated by itself, hence the target networks increases stability. Both the actor and the critic each has a target network that works by periodically setting the actor/critic parameters to the latest values which are used to calculate the next state's Q-value. The actor and critic are denoted with Q' and μ' respectively. The target networks has "soft" updates, meaning that only a fraction $(1-\tau)$ of the weights, θ, are transferred, based on main networks as shown in Equation 8 and 9.

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)^{Q'} \tag{8}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)^{\mu'} \tag{9}$$

where (1-τ) $<<$ 1.

Moreover the training process for the deep neural network can be quite sensitive. This is solved by adding a replay buffer consisting of a set of tuples D. The replay buffer samples experiences in order to update neural network parameters. The size of the buffer affects the algorithms stability. If it is too small the algorithm will overfit to the latest data and if it is too large it will slow down the learning process. The last parts needed to be defined for training the model are the network's updates for the actor and the critic. The Q-values for the next state are computed by a mean square loss function between the original and updated Q-values as shown in Equation 10. The purpose of the policy function is to maximize the expected return, J. The expected return is simply calculated by the estimate of $Q(s,a)$. Maximizing the return is done by taking the derivative of the estimate w.r.t. the policy parameter, θ^{μ}. The fact that a replay buffer is used also needs to be considered which all will result in Equation 11.

$$L = \frac{1}{N}\sum_{i}\Big(y_i - Q(s_i,a_i|\theta^{Q})\Big)^2 \tag{10}$$

where y_i is calculated from the Bellman equation.

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_{i}\Big(\nabla_a Q(s,a|\theta^{Q})|_{s=s_i,a=\mu(s_i)} \nabla_{\theta\mu}\mu(s|\theta^{\mu})|s_i\Big) \tag{11}$$

In Algorithm 2 the process for the DDPG agent is described.

carhe007@student.liu.se

Algorithm 2 DDPG Algorithm [7]

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $\mathcal{R}$
for *For* episode=1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $t = 1$, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action at and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{R}$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\Sigma_i(y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
    $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$
    Update the target networks:
    $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for

3.5 Black-Box Model

A black-box model lets you estimate models from measured data. In this way, a model of a system can be modeled without knowledge of the physics of the system [8]. All that you have to know is the inputs and outputs of the system. It is those features that the data, which is used to estimate the model, contains.

Black-box models are normally described in discrete time since the data used to estimate them are collected in sampled form [8]. Equation 12 shows a general linear discrete time model.

$$y(t) = G(q,\theta)u(t) + H(q,\theta)e(t) \tag{12}$$

Here, $G(q,\theta)$ is the transfer function from the input, u(t), to the output, y(t). The matrix $H(q,\theta)$ is the transfer function from the disturbance, e(t) (white noise), to the output, y(t). The vector θ is the unknown model parameters that will be estimated and q are the shift operator.

There are several special case-models that can be modeled based on this general model. Three of these are the output error model (OE), the ARMAX model and the ARX model.

3.5.1 *Output Error*

An OE model does not model the properties of the disturbance signals and sets $H(q,\theta) \equiv 1$. This gives the model described in Equation 13.

$$y(t) = G(q,\theta)u(t) + e(t) \tag{13}$$

3.5.2 *ARMAX*

ARMAX model the two transfer functions with the same denominator but different nominators. If A(q) describes the denominator, B(q) the nominator in $G(q,\theta)$ and C(q) the nominator in $H(q,\theta)$, the ARMAX model can be written according to Equation 14

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{C(q)}{A(q)}e(t) \tag{14}$$

3.5.3 *ARX*

The ARX model is a special case of the ARMAX model where the $C(q) \equiv 1$. This gives the model in Equation 15.

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t) \tag{15}$$

3.6 Three Parameter Model

The idea with the three parameter model is to be able to estimate the dynamic of an unknown system by study the response of a step made in the input signal of the system. The model contains three parameters that need to be determined, these are the static gain of the system (K), the time constant (T) and the time delay (L). The model can be written according to Equation 16.

$$G(s) = \frac{K}{Ts+1}e^{-sL} \tag{16}$$

From a step response, the three parameters can be determined. The parameter K is determined by the static gain of the step response. The parameter T is determined by the time it takes for the system to reach 63 % of its final value in the step response and L is determined by the time between when the step is preformed and when the system reacts to the step.

3.7 RGA Analysis

RGA analysis is used to measure the degree of cross correlation in a system [9]. The RGA value is defined as below in Equation 17

$$RGA(G) = G. * (G^{-1})^T \tag{17}$$

where G is the system matrix and ".*" indicates element wise multiplication. The value of a row i and a column j indicates how strong the connection between input signal u_i and the output signal y_j are.

3.8 Decoupled control

Decoupled control are used when you want to do decentralized controlling of a system, but there are no natural couples of input- and output signals. With the decoupled control, a change of variable will be performed where $\tilde{y} = W_2 y$ and $\tilde{u} = W_1^1 u$. This results in a transfer function

$$\tilde{G} = W_2(s)G(s)W_1(s) \tag{18}$$

that is as near a diagonal matrix as possible. Here, G is the original transfer function of the system. This will result in a regulator described below.

$$u = -W_1 F_y^{diag} W_2 y \tag{19}$$

where F_y^{diag} is the PID controller where each input will control each output separately. The matrices $W_1(s)$ and $W_2(s)$ are thus two matrices that will manipulate the system so that one input signal controls one output signal and affects the other output signals as little as possible, i.e turn the system matrix, $\tilde{G}$, into a diagonal matrix. These matrices can be hard to find and are often designed to fulfill this in the stationary case ($s = 0$) [9].

3.9 Path Generator

In this chapter the theory used to construct the path generation software will be explained.

3.9.1 *Motion planning*

In order to create a path that could realistically be followed by a moving vehicle the path is created by simulating a car with random steering wheel inputs. The movement of the car can be planned using motion planning. Motion planning is a strategy for taking the vehicle from one state to another. States can for example be position or orientation. To make the strategy more realistic constrains are introduced on for example steer angle and acceleration as well as physical characteristics such as the length of the wheel axis.

3.9.2 *RRT*

RRT is a path-planning algorithm. The algorithm can be viewed in Algorithm 3.

Algorithm 3 RRT [10]

$\mathcal{T}.init(x_{init})$
for k=1 **to** K **do**
  $x_{rand} \leftarrow RANDOM\_STATE();$
  $x_{near} \leftarrow NEAREST\_NEIGHTBOR(x_{rand}, \mathcal{T});$
  $x_{new} \leftarrow STEER(x_{near}, x_{rand}, \Delta x);$
  $\mathcal{T}.add\_vertex(x_{new});$
  $\mathcal{T}.add\_edge(x_{near}, x_{new});$
  if $x_{new} = x_{goal};$
    end
end for
Return $\mathcal{T}$

The algorithm works as follows. A random node is selected in the generated environment, which in our case is a 100x100 grid. The algorithm will iterate K times. Afterwards, the nearest neighbor to the random node is selected and the graph is steered towards it; wherever the graph lands is the new node. The two nodes are then connected and, unless the new node is the goal node, the process is repeated. The algorithm either ends when it has iterated K times or when the goal has been reached. The path is constructed of arrays which contain the x- and y-coordinates, the orientation angle, and the velocity.

4 SYSTEM OVERVIEW

In the present section the complete system is presented followed by descriptions of the systems components. A schematic view of the complete system is presented in Figure 1.

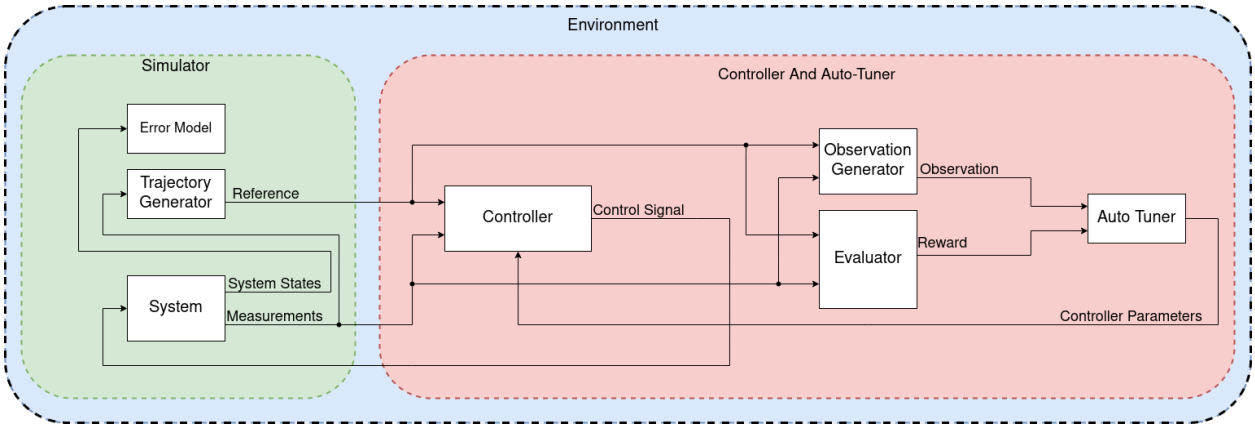


Figure 1: A schematic view of the complete systems components together with their respective inputs and outputs. The red area mark the area that would be implemented in the physical AGV. In that case the green area would be exchanged with the physical AGV. The blue area marks the environment with which the auto-tuner interacts.

The main components of the complete system is the *Simulator*, the *Controller*, the *Observation Generator*, the *Evaluator* and the *Agent*.

- The *Simulator* is responsible for simulation of the physics of the AGV, for calculating errors that can be used for visualisation and for providing a reference signal for the controller. The *Simulator* is further described in Section 5.
- The *Controller* is responsible for calculating control signals for the AGV and is described further in Section 6.
- The *Observation Generator* is responsible for providing the *Agent* with an observation signal and is described further in Section 7.
- The *Evaluator* is responsible for providing the *Agent* with a reward signal and is described further in Section 8.
- Finally, the *Agent* is responsible for providing the *Controller* with control parameters and is further described in Section 9.

The complete system in simulink can be viewed in Figure 2.

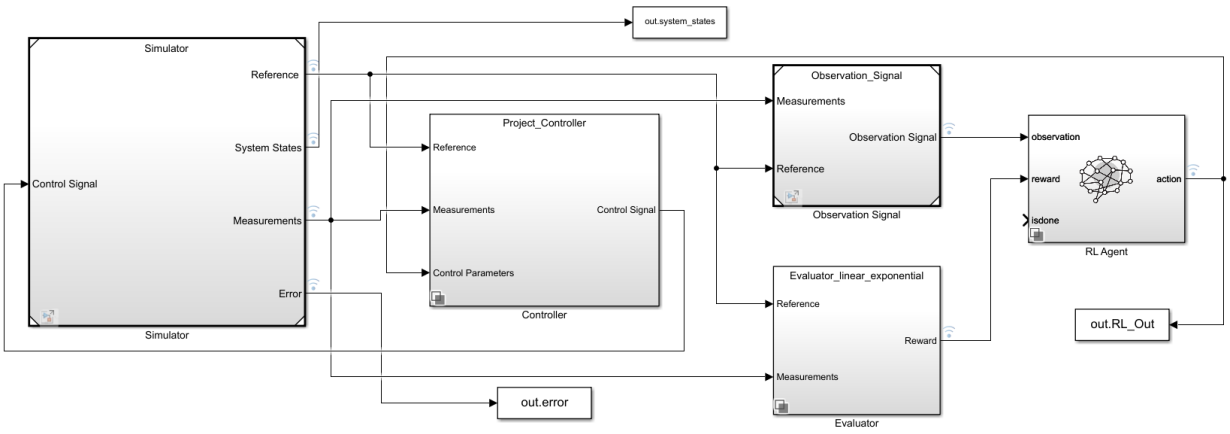


Figure 2: The complete Simulink model.

5 SIMULATOR

The simulator is structured in simulink as shown in Figure 3.

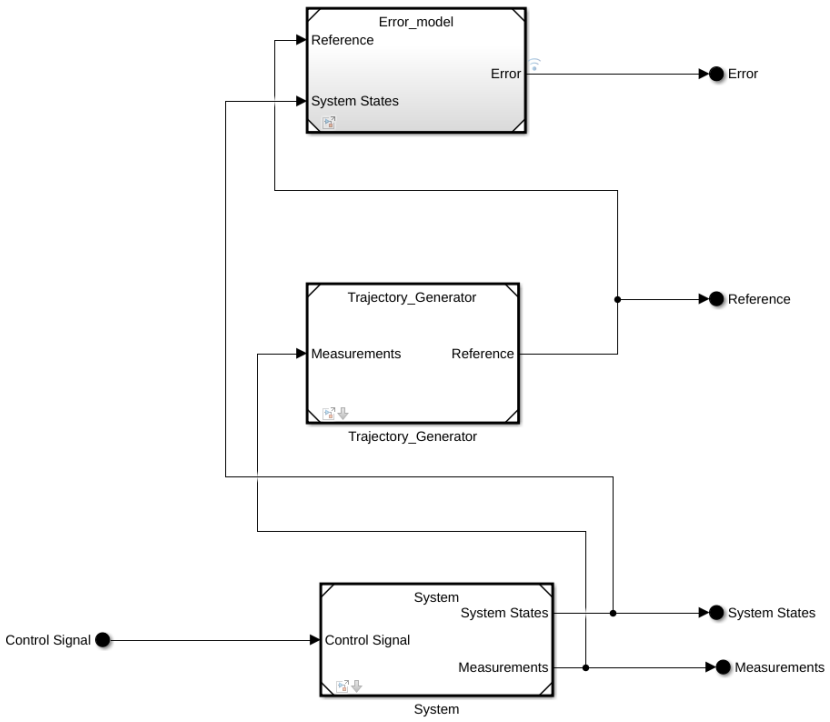


Figure 3: Simulator model in simulink.

5.1 System model

In this section, the physical models which describes the AGV are presented and explained. The system mainly consists of two subsystems, the electrical motor model and the kinematic model of the AGV. The two models are described below. The system model template in simulink can be viewed in Figure 4

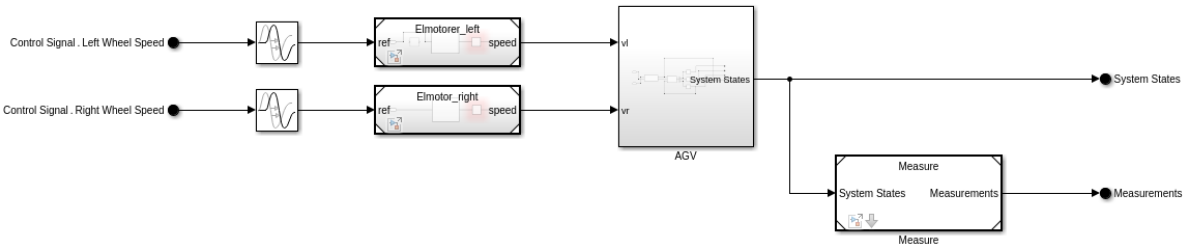


Figure 4: Simulator model in Simulink

5.1.1 *Electrical motor*

The AGV has two electrical motors, one to each wheel. Since it is the same type of motors powering the two wheels, the dynamical models of the two motors are the same. The models were estimated from measurement data provided by Toyota. The data had been collected from the AGV when it was driving. The data contained a lot of features. The features used to estimate the electrical motor were the reference velocity of each wheel and the actual velocity of each wheel. The provided data was divided into three data sets of equal size. Two of the data sets were used to estimate the model of the electrical motor.

To estimate a model of the electrical motor, the System Identification Toolbox in Matlab was used. The data for the right wheel (reference velocity and the actual velocity) of the first data set was divided into training data (Working data) and validation data. The split was 50/50. The mean and trend of all the data sets was removed before the estimation and validation were preformed. First, several polynomial models was estimated using ARX, ARXMAX and OE models (see Section 3.5). The order of the models was kept as low as possible to get simple models. Some state space models were also estimated.

The evaluation of the models was mainly done by looking at the performance of fit to the validation data, the location of the zeros and poles of the models and the cross correlation of the input and output to the model. After some estimations, an ARX model order [3 1 3] was chosen. The fit to the validation data of the model can be seen in Figure 5 together with an other estimated model. Both models fit the validation data very well, about 98 %.

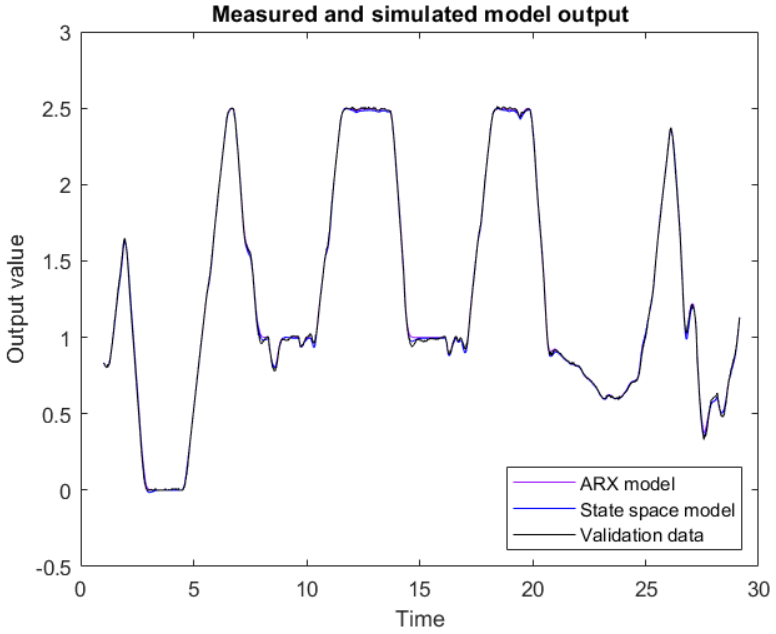


Figure 5: The output values of the estimated models and the validation data.

Figure 6 show the zeros and poles localization of the two models. One can clearly see that the poles of the other model are located outside the unit circle, making the model unstable. This is the reason why the ARX model was chosen.

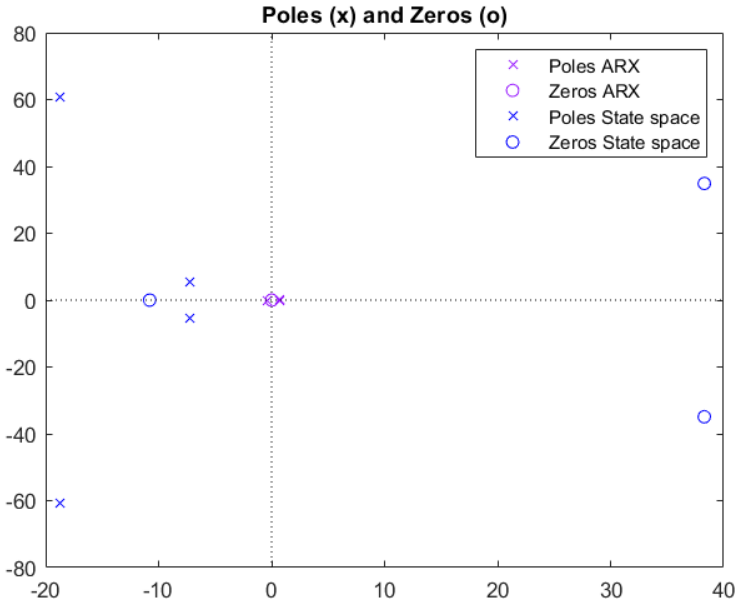


Figure 6: Localization of the zeros and poles of the estimated models.

The chosen model was evaluated on the other data sets mentioned above. The fit to the data for the model was between 95-98 %. The model are given in discrete form according to

$$A(z)y(t) = B(z)u(t) + e(t) \tag{20}$$

here,

$$A(z) = 1 - 0.9968z^{-1} - 0.06231z^{-2} + 0.1968z^{-3}$$

and

$$B(z) = 0.1381z^{-3}.$$

This gives the discrete transfer function

$$G(z) = \frac{0.1381z^{-3}}{1 - 0.9968z^{-1} - 0.06231z^{-2} + 0.1968z^{-3}} \tag{21}$$

Since the other Simulink models are in continuous time the discrete transfer function is made continuous in Matlab using the command "d2c". The resulting transfer function is presented below.

$$G(s) = \frac{-8.005s^3 + 1559s^2 - 2.072 \cdot 10^5 s + 9.325 \cdot 10^6}{s^4 + 127.3s^3 + 3.038 \cdot 10^4 s^2 + 9.771 \cdot 10^5 s + 9.298 \cdot 10^6} \tag{22}$$

As one can see, G(s) is not so simple. To further simplify the model, the model was implemented in Simulink with a step as input signal. The output of the model was studied and a three parameters model was made to fit the dynamics (see Section 3.6). The performed step can be viewed in Figure 7.

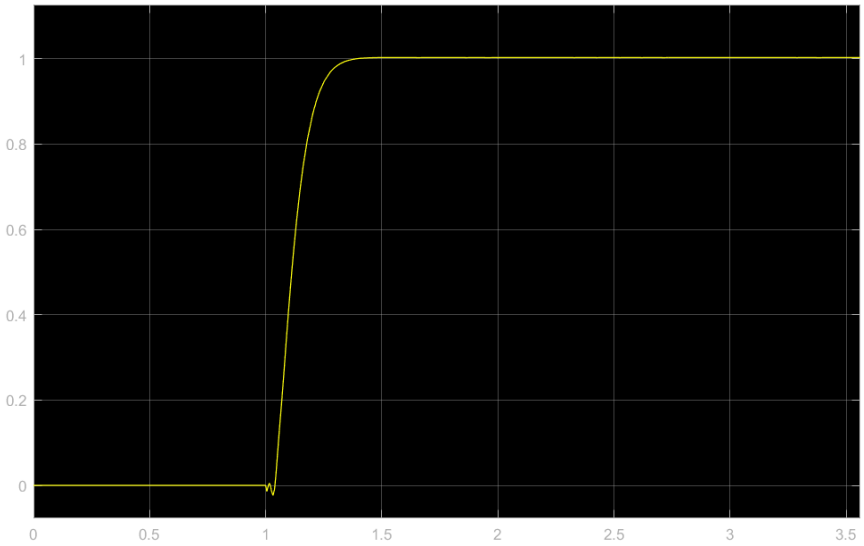


Figure 7: The output of the estimated model when the input signal is a step with 1 as final value. The time is plotted on the horizontal axis and the output values are plotted on the vertical axis.

In Figure 7, one can see that the static gain is equal to 1, the time delay is approximately 0.04 seconds and the time constant is about 0.1 seconds. This results in the final model of the electrical motor which is given in Equation 23 as a first order system with a time delay.

$$G(s) = \frac{1}{0.1s+1}e^{-0.04s} \tag{23}$$

Figure 9 shows the final model of the electrical motor implemented in Simulink. The input to the model is the reference velocity of the wheel that the motor is powering and the output is the velocity of the wheel. Figure 8 shows the final model together with the transfer function in Equation 22. One can see that the two models are performing very similarly.

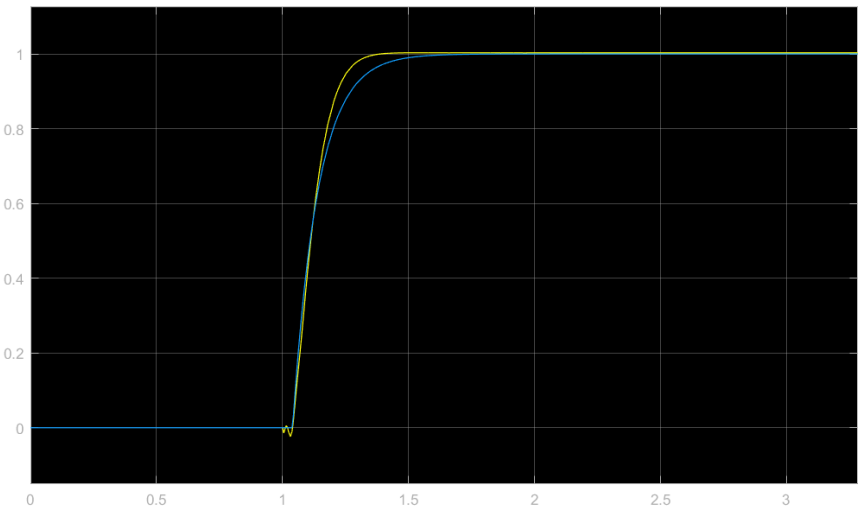


Figure 8: The output of the estimated ARX model and the final model of the electrical motor when the input signal is a step with 1 as final value. The time is plotted on the horizontal axis and the output value are plotted on the vertical axis.

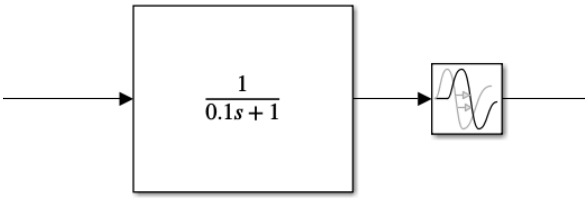


Figure 9: The figure shows how the final model of the electrical motor is implemented in Simulink.

5.1.2 *Disturbance implementation in the electrical motor*

There are several disturbances that can affect the motors in the AGV. For instance, the required torque in the motor can increase suddenly because the mass that the vehicle is carrying increases or are unevenly distributed on the vehicle, which would result in an increasing time delay. The required torque can also decrease for instance if the friction between the wheel and the floor is decreasing, which would result in a decreasing time delay. Therefore, the time delay in the motor can be adjusted from the GUI as a way to simulate the mentioned disturbances. For each episode during training of the auto-tuner, time delay in the left and right motor is uniformly sampled between zero and a max value that is configured in the GUI.

5.1.3 *Kinematic model*

The AGV will drive around in a global coordinate system where the coordinate axis are fixed to the environment. The AGV itself will constitute a local coordinate system with origin centered in between the two driving wheels and the x axis always pointing forward in the heading of the AGV. The two coordinate systems are illustrated in Figure 10. The angle between the x axis in the global coordinate system (x) and the x axis in the local coordinate system (x') is theta (θ) and will be referred to as the heading angle.

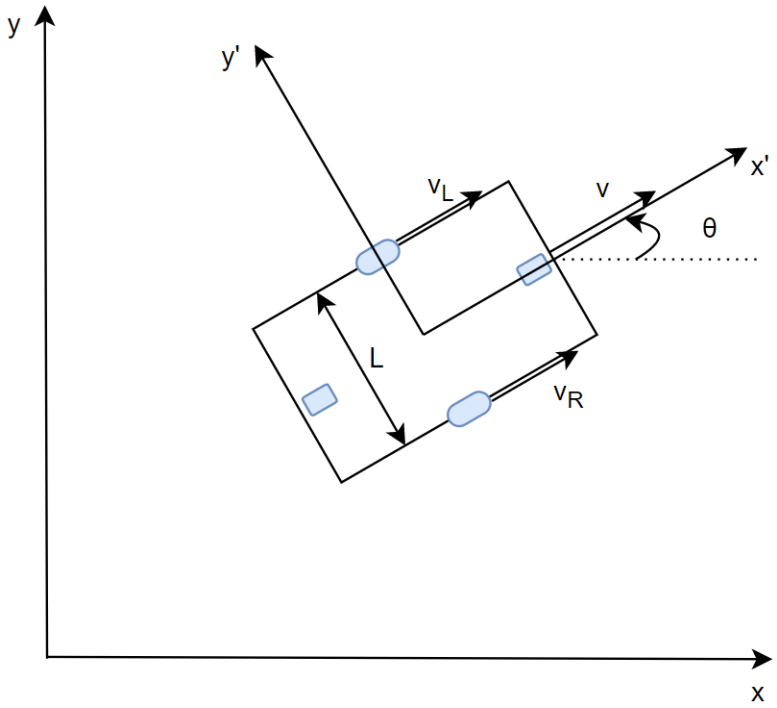


Figure 10: An illustration of the local and global coordinate system.

A point in the global coordinate system can, with the local coordinates, be described as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} \tag{24}$$

The position and orientation of the AGV will be determined by the wheel velocities of the side wheels powered by the two electrical motors. The velocity and the angular rate can be calculated as

$$v = \frac{v_r + v_l}{2} \tag{25}$$

$$\omega = \frac{v_r - v_l}{L} \tag{26}$$

where L is the length of the wheel axle, v is the AGV aligned velocity and ω is the angular rate of the vehicle. This corresponds to the state-space model

$$\begin{bmatrix} \dot{x}_l \\ \dot{y}_l \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0 & 0 \\ 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} v_r \\ v_l \end{bmatrix} \tag{27}$$

Equation 24 and 27 give the final kinematic model which gives the global coordinates of the AGV in state space form. The model in global coordinates is thus

$$\begin{bmatrix} \dot{x}_g \\ \dot{y}_g \\ \theta \end{bmatrix} = \begin{bmatrix} 0.5\cos\theta & 0.5\cos\theta \\ 0.5\sin\theta & 0.5\sin\theta \\ 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} v_r \\ v_l \end{bmatrix} \tag{28}$$

The implementation of the global kinematic model is presented in Figure 11, 12 and 13.

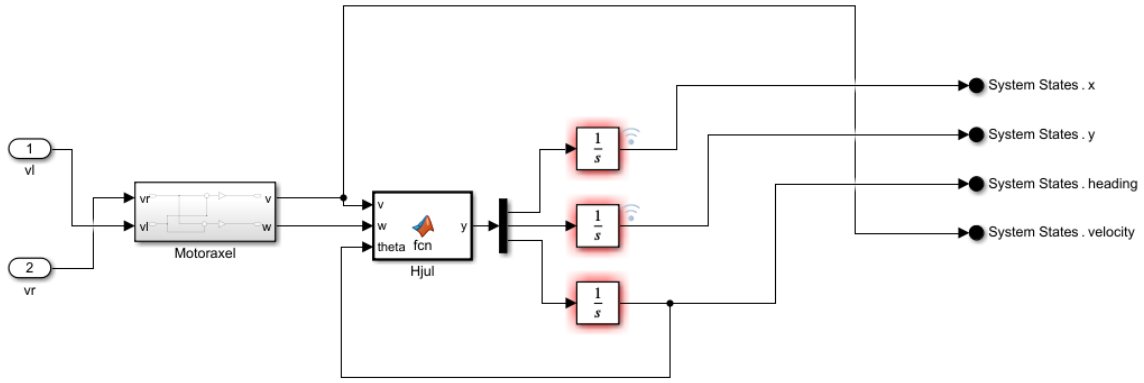


Figure 11: The Kinematic model implemented in Simulink.

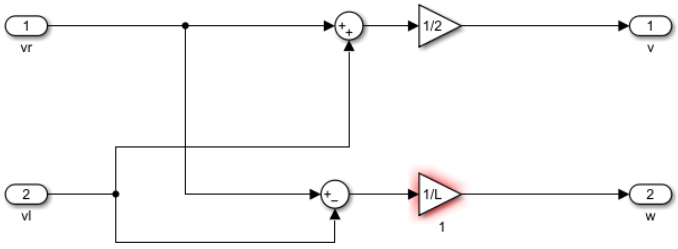


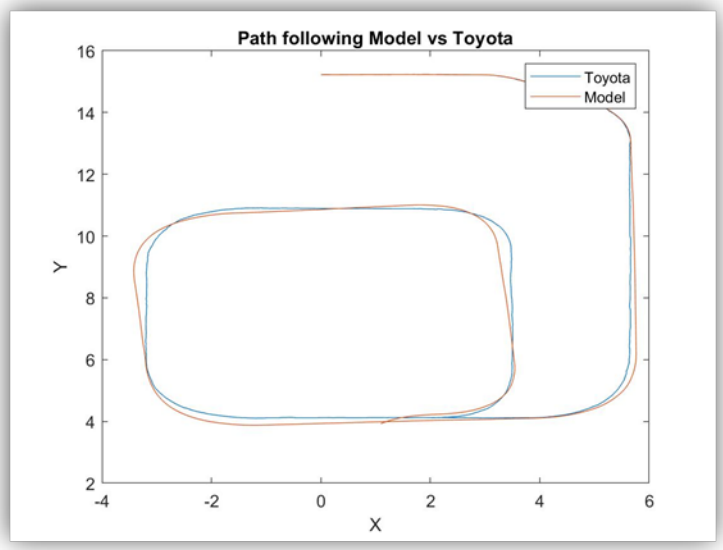
Figure 12: The block "Motoraxel" implementation in Simulink.

```
function y = fcn(v,w,theta)

x_prick = cos(theta)*v;
y_prick = sin(theta)*v;
theta_prick = w;

y = [x_prick; y_prick; theta_prick];
```

Figure 13: The Matlab function "Hjul" which is a part of the implemented Kinematic model in Simulink.

5.2 System Validation

In order to validate the model described in Section 5.1 the system outputs are compared to measurement data from a real world AGV when using the the same inputs. The inputs and validation data have been obtained from tests done by Toyota. The X and Y target position from the tests are fed into the system and used as reference data and then compared to the results generated by the AGV during the test. Example of the output can be seen in Figure 14



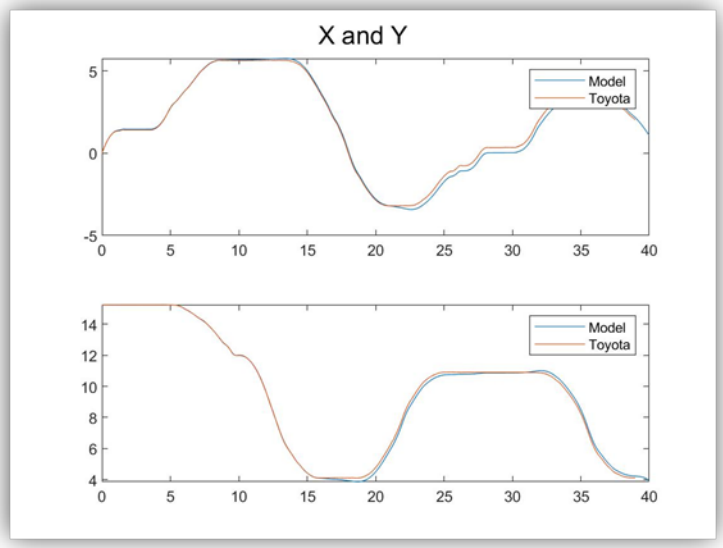


Figure 14: Plot comparing the outputs of the model to the measurements obtained by Toyota

In Figure 14, one can see that the model reflects the real AGV very well, although it is not perfect. However, the results are considered to be good enough to use the model in the simulator.

5.2.1 *Measurements*

The measurement component is responsible for creating realistic measurement signals from the system states that is outputted from the system component. This is accomplished by adding white noise to each system state. The covariance of this noise can be indirectly controlled by setting the peak power distribution or *noise power*. For each episode during training of the auto-tuner, the noise power for each measured signal is uniformly sampled between zero and a max value that is configured in the GUI. Figure 15 shows how the measurements are implemented in Simulink.

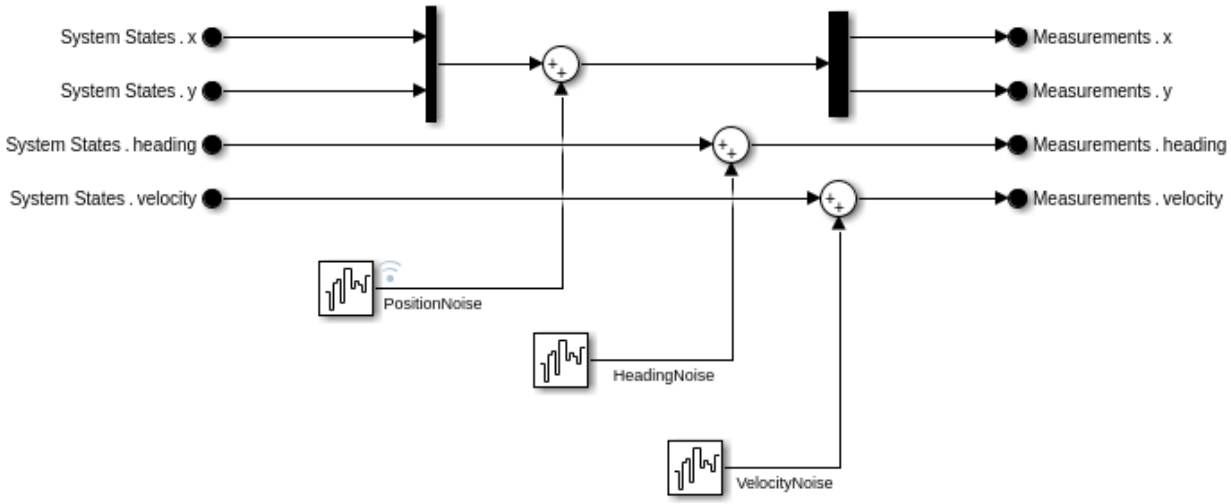


Figure 15: Implementation of the measurements in Simulink.

5.3 Path- and Reference Generation

In this subsection the path generation and reference generation is described.

5.3.1 *Path-generation*

Below is a description of how the two path generation methods are created.

5.3.1.1 RRT

The creation of the RRT path is visualized in Figure 16.

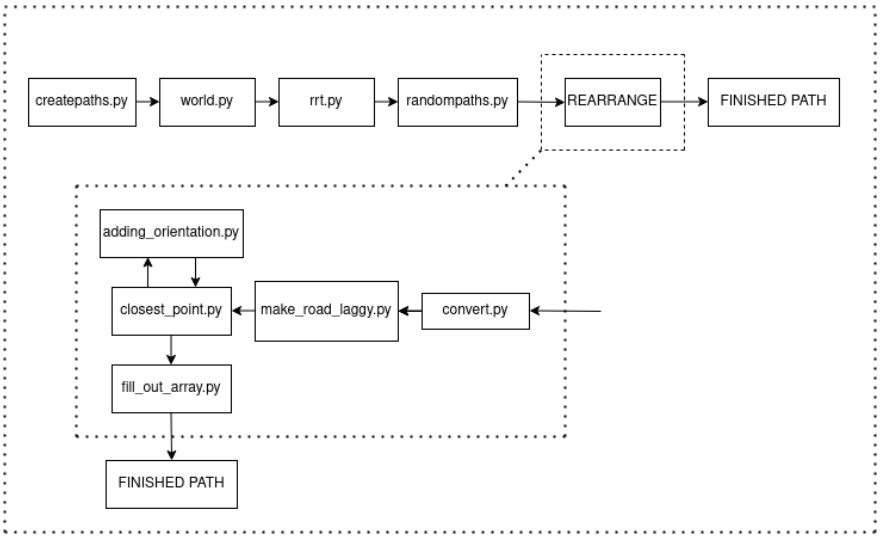


Figure 16: RRT construction

The user input is analyzed in createpaths.py (the input is how many paths that shall be created). The amount of paths is later combined in randompaths.py with the world creation made by world.py and rrt.py. The paths are then rearranged in multiple steps. The rearrange is made to make the path not have two similar nodes in the paths and also to make the path as difficult as possible.

5.3.1.2 SPLINE PATH

The model is based on the theory from Section 3.9.1 and has four main states, X, Y, heading (θ) and velocity (v). New state values are calculated with an even time interval T_s using acceleration (a) and steer angle (δ). To generate a realistic behavior δ has been limited to $\pi/4$. For the purpose of generating paths acceleration and velocity are kept constant while the steer angle is randomized. The odds are set to a two in five chance to give a max steer angle in each step and a three in five chance to give an input angle of a random number between 0.698 rad and -0.698 rad. The path is made up of all the X and Y coordinates the path model visits. Since the distance error for the controller is calculated as the distance to the closest (x,y) position it is important to be able to adjust the density of the states. This is done by using equidistant states. This approach has been turned into a function which can be looped to create any specified integer number of randomized paths for training the agent. Settings for path density and number of paths can be found in the GUI settings in the Path settings section. It is also possible to change the variables in the *run_simulator.m* file. Examples of paths generated by the spline function can be seen in the Figures 41 and 42 in Section 12.

5.3.2 *Reference Generation*

The reference generator block in simulink can be viewed in Figure 17. The simulink model takes measured inputs of x- and y-coordinates of the AGV and calculates what point/node to advance towards in the path-following algorithm based on which is closest. The reference orientation uses a look-ahead, meaning that the reference is set to the orientation a number of points forwards. The reference velocity is however set to a fixed value. The velocity and orientation is sent to the controller, see Figure 2 for clarification.

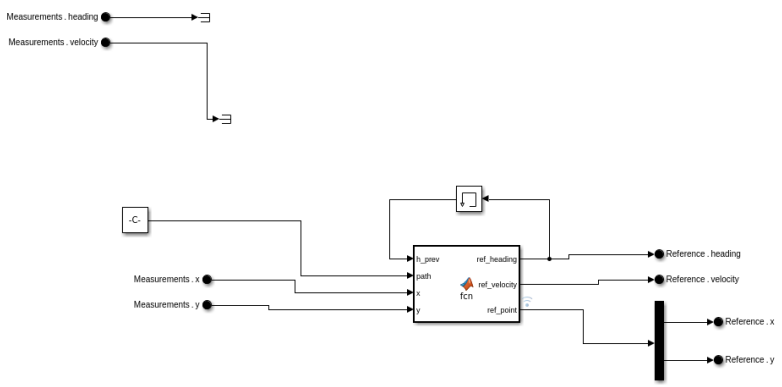


Figure 17: Reference generation

5.4 Error Model

The error model is used to calculate the errors and path deviations and later is used to analyse the performance of the agents and the static tuning. The model uses the system states and the reference path to calculate the heading, velocity and distance errors. The Simulink implementation can be viewed in Figure 18.

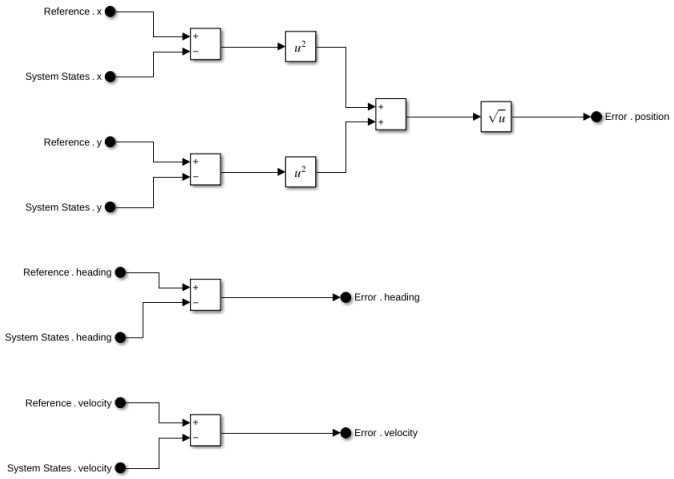


Figure 18: Error model

carhe007@student.liu.se

6 CONTROLLERS

Here the systems two controllers are described. The first one called the Project Controller is the main controller used to regulate the AGV. The second controller called the Toyota Reference Controller is used to perform the system validation described in Section 5.2. The top level controller block in the Simulink model containing both controllers can be viewed in Figure 19. The reference signal is received from the reference generator, see Section 5.3.2 for further details, and the controller yields a control signal which is used in the simulator, see Figure 2.

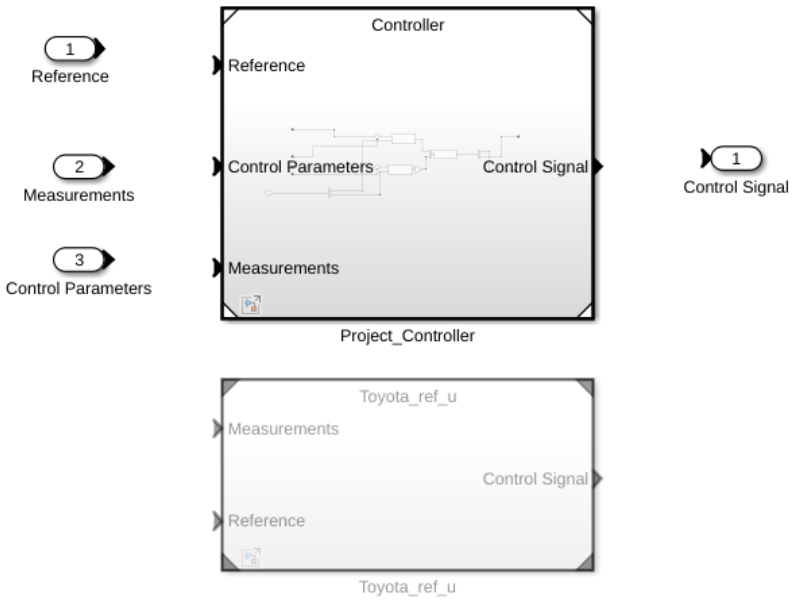


Figure 19: Top level controller block containing both the Project Controller and the Toyota Reference Controller. The block is implemented as a variant block in Simulink enabling the user to easily select a controller from the GUI.

6.1 Project Controller

The Project controller controls the wheel speed of the two driving wheels in the AGV to reach the desired velocity and heading angle of the vehicle. The inputs to the controller are thus the reference velocity and the reference heading, described in Section 5.3.2, and the outputs are the reference speed of the right and left motor in the AGV. The controller is a PID controller.

Since the number of input signals and output signals are equal, Decentralized control will be used. To determine if there are any naturally couples of input- and output signals, an RGA-analysis is made. Since the controller controls the velocity and heading of the AGV, Equation 27 is the system that the controller controls. If the velocity in the y-direction (since it is always zero and not controlled) the system can be written as

$$G = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & -0.5 \end{bmatrix} \tag{29}$$

The RGA are calculated according to Equation 17 in Section 3.7 and are presented below.

$$RGA = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \tag{30}$$

As one can see, there are no input signal that affect one of the outputs signals more than the other. Because of this, decoupled control is suitable to use (see Section 3.8). With

$$W_1 = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

and

$$W_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

the controller becomes decoupled with the first input signal (v_r) controlling the first output signal (v) and the second input signal (v_l) controlling the second output signal (θ). The project controller can be viewed in Figure 20.

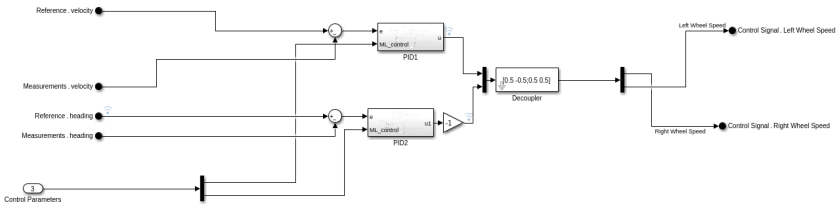


Figure 20: Project controller

Both PID controllers are identical in Simulink and can be viewed in Figure 21.

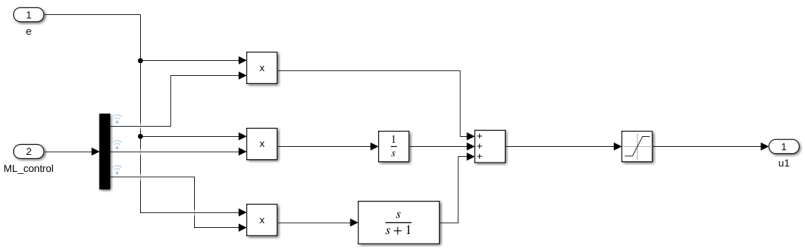


Figure 21: PID

6.2 Toyota Reference Controller

Figure 22 show the Simulink implementation of the Toyota Reference Controller that is used for the system validation described in Section 5.2.

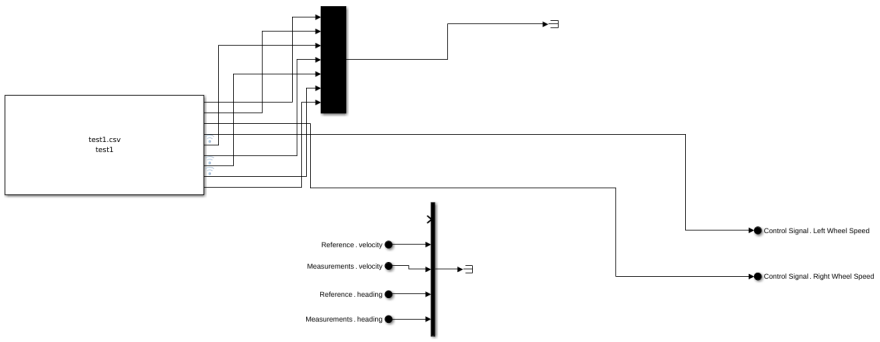


Figure 22: The Simulink implementation of the Toyota Reference Controller.

7 OBSERVATION GENERATOR

The observation signal is defined as the absolute values of the position errors in x and y directions, heading error and velocity error according to the following equations:

$$\begin{aligned} o_x &= |x_{meas} - x_{ref}| = |e_x| \\ o_y &= |y_{meas} - y_{ref}| = |e_y| \\ o_\theta &= |\theta_{meas} - \theta_{ref}| = |e_\theta| \\ o_v &= |v_{meas} - v_{ref}| = |e_v| \end{aligned} \tag{31}$$

where z_{ref} and z_{meas} are the reference and measured values of the parameter z. Figure 23 shows the implementation of the observation generator in Simulink.

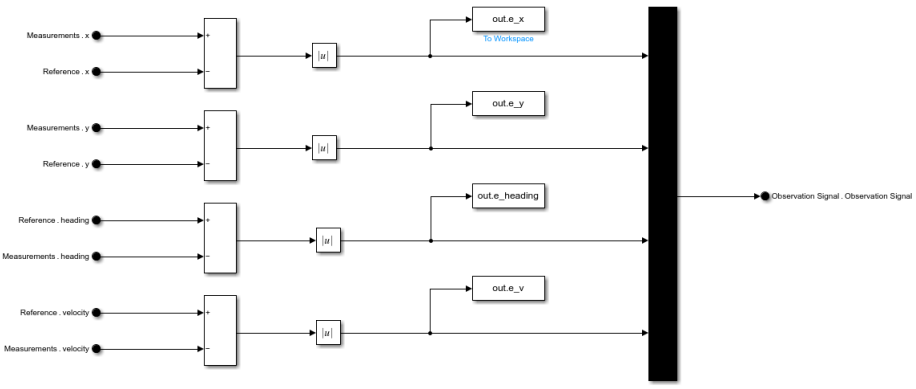


Figure 23: Implementation of the observation generator in Simulink.

8 EVALUATOR

In this section the reward signals are described.

8.1 Linear Reward

The linear reward function is the sum of the absolute values of the heading and velocity errors multiplied by their corresponding constants K_θ and K_v. The linear reward function is defined as:

$$r_{lin} = |e_\theta| \cdot K_\theta + |e_v| \cdot K_v \tag{32}$$

where e_θ is the heading error, e_v is the velocity error and K_θ and K_v are heading and velocity constants. These constants are tuning parameters and should be adjusted based on the type of the agent and/or the purpose of the training. For instance, if the velocity constant is larger than the heading constant, a change in the velocity error affects the reward more than what an equal change in heading error would do. As a consequence, the agent would lay more weight on keeping the reference velocity as compared to the heading since it would maximize the reward. Figure 24 shows the implementation of the linear reward function in Simulink.

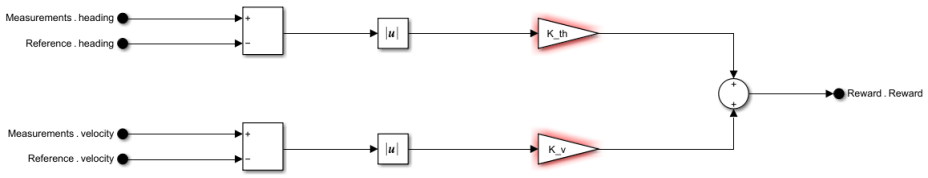


Figure 24: Implementation of linear reward in Simulink.

8.2 Linear Exponential Reward

The linear-exponential reward function consists of two components, a linear component and an exponential component. The idea is that the absolute value of the reward should increase linearly when the heading and velocity errors lay between zero and a threshold value and exponentially otherwise. In order to make this function work as intended, the reward must be negative so that a large value for the reward could be considered as a bad reward. In this way, the agent is encouraged to keep the errors in the linear part in order to maximize the reward. The linear-exponential reward function is defined according to:

$$r_{lin-exp} = r_\theta + r_v \tag{33}$$

where

$$r_\theta = \begin{cases} |e_\theta| \cdot K_\theta & \text{if} \quad |e_\theta| < C_\theta \\ \exp(|e_\theta| - C_\theta) \cdot K_\theta & \text{if} \quad |e_\theta| >= C_\theta \end{cases} \tag{34}$$

$$r_v = \begin{cases} |e_v| \cdot K_v & \text{if} \quad |e_v| < C_v \\ \exp(|e_v| - C_v) \cdot K_v & \text{if} \quad |e_v| >= C_v \end{cases} \tag{35}$$

and where e_θ, e_v, K_θ and K_v are as before and C_θ and C_v are heading and velocity thresholds. Figure 25 illustrates the implementation of the linear-exponential reward function in Simulink.

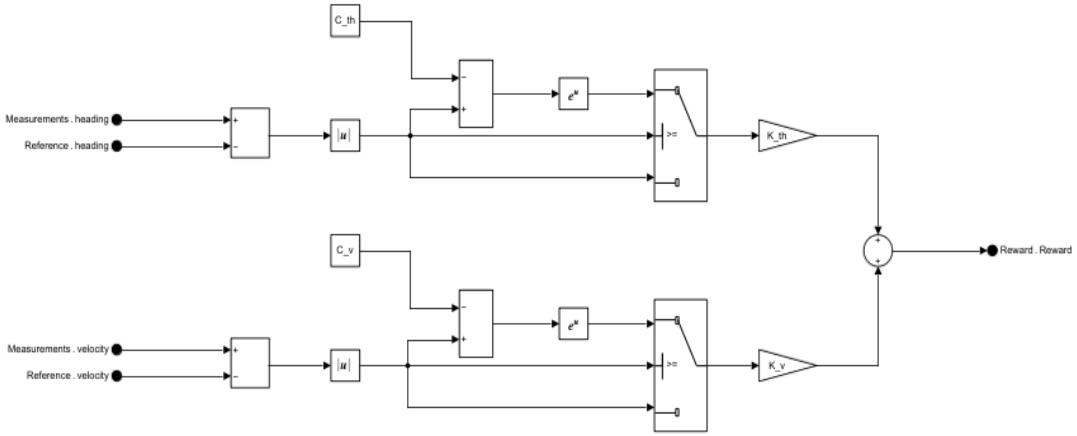


Figure 25: Implementation of linear-exponential reward in Simulink.

9 AUTO TUNER

The agents used for the auto tuner are built by neural networks that utilizes a number of different layers, each with their own functionalities. The used layers are described below.

FEATURE INPUT This layer is the start of every network and it inputs feature data to the network and applies data normalization.

FULLY CONNECTED (FC) The fully connected layer multiplies its layer's input with a weight matrix and adds a bias vector. This layer has the most affect on the size and the needed computational time of the network. The number of hidden units, which is a hyperparameter, needs to be defined. In this project this value can be changed both when using the GUI and the code.

RELU Rectified linear unit (RelU) is an activation layer that performs a threshold operation on the input. The input values that are less than zero are set to zero. This layer speeds up the training process since it has fewer flat gradients.

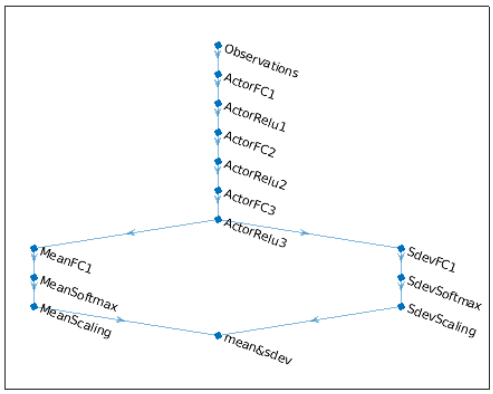
TANH This is an activation layer that applies the *tanh* function on the layer's inputs. The pro of this layer is that it gives zero-centered output and thereby supports the backpropagation process. The output has a value in the range of [-1,1].

SOFTMAX The softmax layer normalizes the input into a probability distribution. The output from this layer is in the range of [0,1].

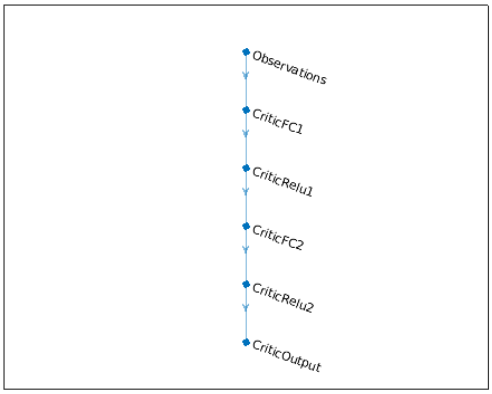
SCALING The scaling layer scales and biases the input the input and is used, in this project, to create lower and upper boundaries for the actions.

CONCATENATION The concatenation layer takes inputs and concatenates them. It is used for the actor network in the PPO-method to combine the mean and standard deviation paths.

Figure 26 and Figure 27 show an overview of the neural networks of the PPO and DDPG agents implemented in MATLAB. It is common practice to have fully connected and activation layers following one another. The activation layers introduce non-linearities while the fully connected helps by learning non-linear combinations. The actor network of the PPO-algorithm uses observations as inputs, introduces non-linearities and hidden units to learn them. The path is divided into two which corresponds to a Gaussian probability distribution, using the softmax layer, with mean value and standard deviation. Before an output is received the two paths are scaled to create the boundaries for the actions. The critic network is simpler, with less layers, and returns the corresponding expectation of the discounted long-term reward as output. When using the DDPG-algorithm the actor network is similar to the PPO critic. It takes in observations and then gives an output in the shape of control parameters used for controlling the AGV. It differs since the actions are scaled to limit the action. The critic network starts with two paths which are observations and actions. These two paths are combined with a concatenation layer. After a few additional layers it returns the expectation of long-term reward as output.

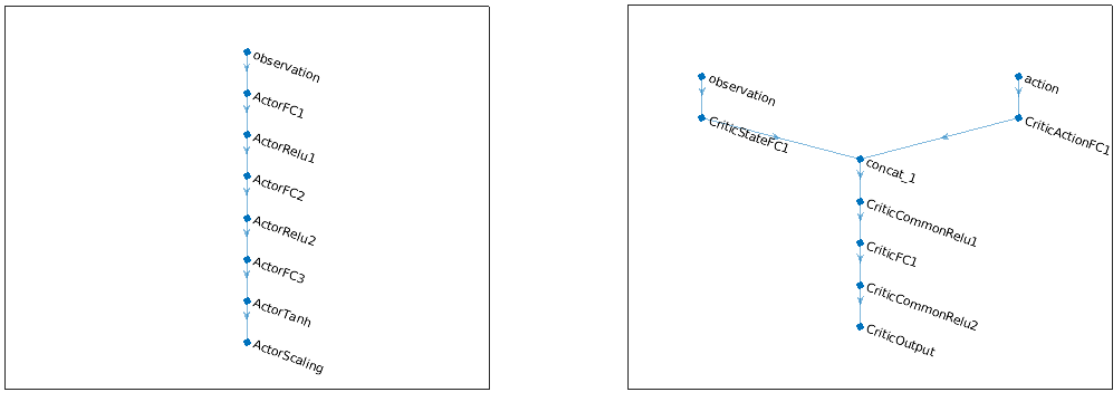


(a) Actor network of the PPO agent.



(b) Critic network of the PPO agent.

Figure 26: Neural network structure of the PPO agent.



(a) Actor network of the DDPG agent.

(b) Critic network of the DDPG agent.

Figure 27: Neural network structure of the DDPG agent.

Furthermore the representations for the actor and critic for each method was created from the networks using in-built functions in MATLAB. The complete agent ready for training could then be defined from said representations. How this was done can be seen in the code in the appendix (see **??**).

9.0.1 *Static Agent*

In addition to the machine learning based auto-tuner, a *Static Agent* has been implemented that outputs constant control parameters that can be set from the GUI.

9.1 Validation Methodology

To validate the result of the auto-tuner a number of simulations were performed with varying measurement noise powers and control delays. For each simulation, the maximum path deviation, heading error and velocity error was calculated and compared with the performance requirements [2]. The result from these validation tests are presented in Section 12.2.

10 VISUALIZATION

In this section the visualization of the system output is explained.

10.1 Visualisations

In the GUI there are three tabs that visualize the performance and training of the agent. The first tab is presented in Figure 28 and shows the results of the most recent simulation. Six different plots are shown. On the left the path heading error and velocity error are visualized. The path plot shows the reference path compared to the position of

the AGV for every time step. The heading and velocity error shows the difference between the reference and the actual value and how it changes over time. On the left side the positional errors are shown. The path deviation total is the euclidean distance between the current position of the AGV and the closest point in the reference path. The path deviation in x and y are calculated in a similar way but only displays the difference in a certain direction.

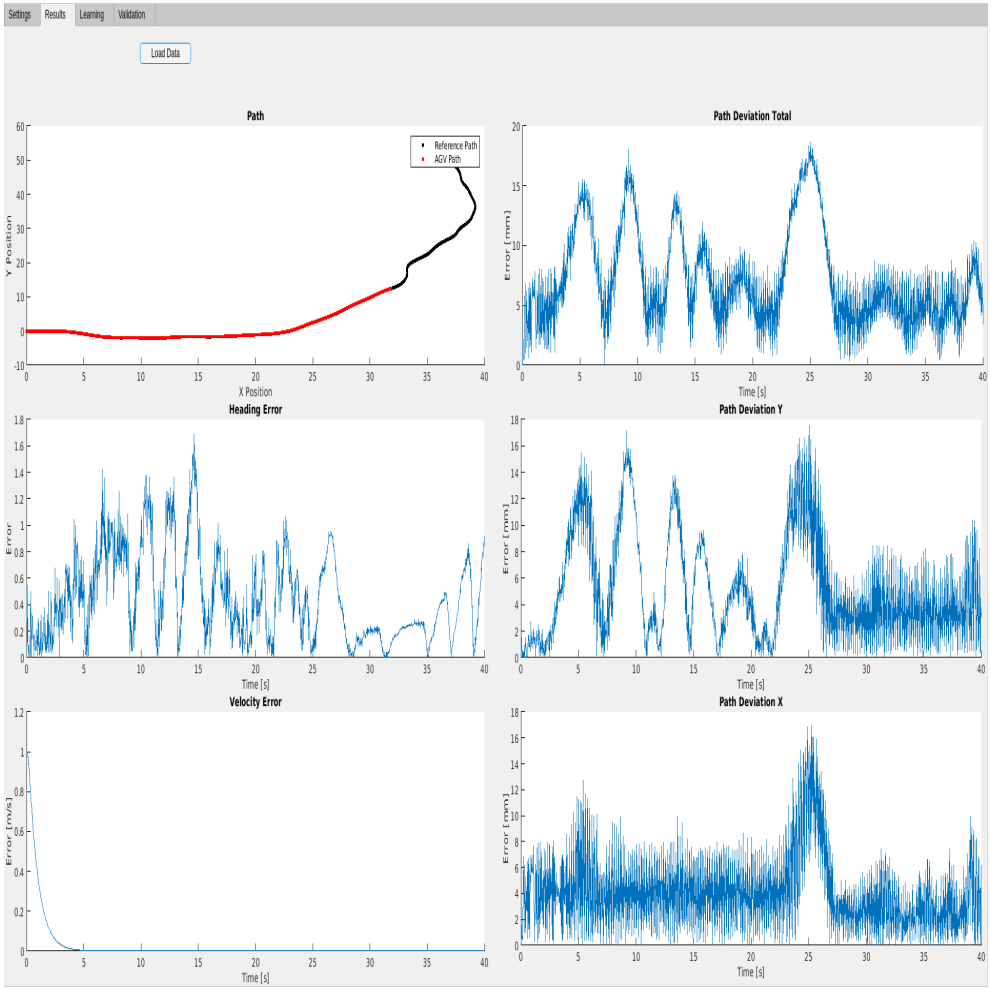


Figure 28: Results tab of the GUI

The learning tab is displayed in Figure 29 and shows three plots. The learning process graph shows the accumulated reword for each episode during training and the 100 episode moving average. Total distance error gives the accumulated distance error for each episode. The third graph shows the average PID parameters for each episode.

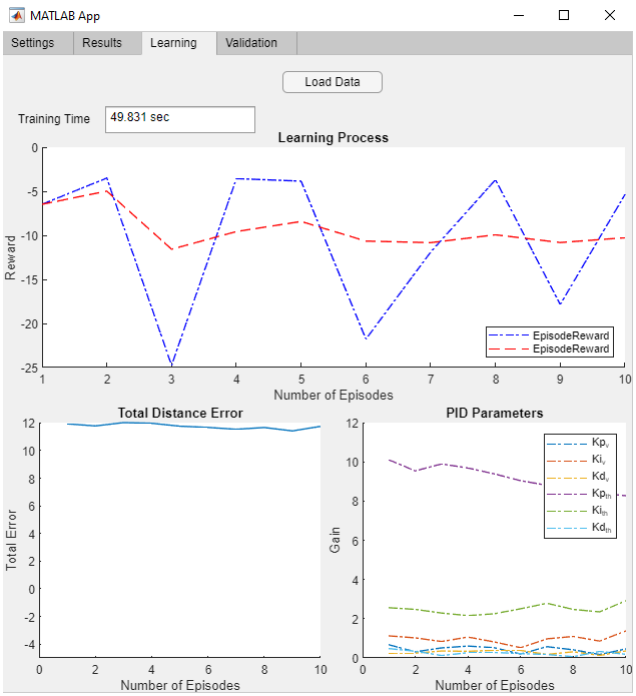


Figure 29: Learning tab of the GUI

The last tab is the validation tab and shows the results from the agent validation, see Figure 30. It displays the highest path deviation, heading error and velocity error from each validation episode. It is also possible to display the performance requirements from the requirements specification [2].

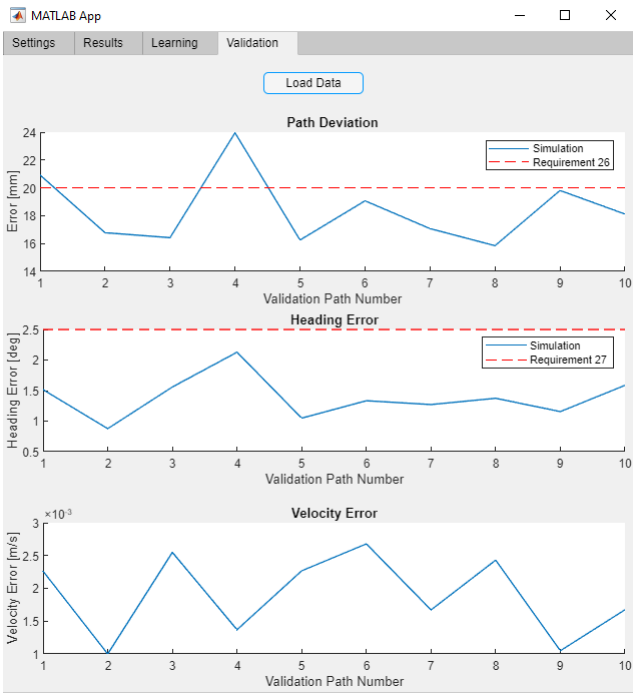


Figure 30: Validation tab of the GUI

11 INITIALIZATION

In this section the execution flow for running the system is described. To start the simulation you first start the GUI file. This will present you with a lot of options which are described in detail in the user manual. To start the simulation you press the start button. This exports all of the settings as variables to the work space along with one extra variable which tells the program that we are running the program via the GUI. The first file to start from the GUI is the *run_simulator.m*. This file contains all the functionality of the GUI but is less user friendly. The run file in turn fetches the the auto-tuner settings from the selected reinforcement learning architecture, generates paths via the *generate_paths.m* file and exports settings to all of the Simulink modules. The program then starts the training of the agent which is saved in a folder called *saved agents*. When the agent is fully trained the textitrun_simulator.m then runs a simulation of the system with the trained agent at the path selected by the *path index plot* setting and saves relevant data to the work space. After this step the user has the option to visualise the data in the GUI. If they wish to do so the GUI will fetch the requested variables from the work space and plot them. A simplified explanation of this process can be see in Figure 31.

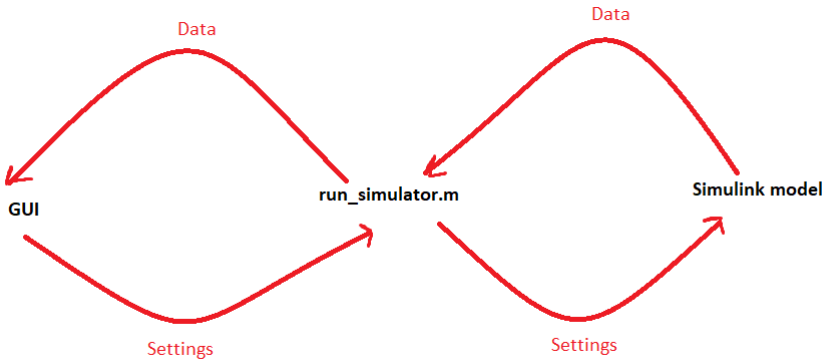


Figure 31: Simplification of the flow when running the software

12 RESULTS

In this section the results are presented.

12.1 Training

In this section, the results of the training with the two reward functions discussed in Section 8 are presented. To be able to do a fair comparison between these two reward functions, the settings in Table 2, of which some are the same for both functions are used.

Table 2: Used parameters when training with two different reward functions.

Variable	Linear Reward	Linear-Exponential Reward
K_θ	-8	-8
K_v	-10	-10
T_s [s]	2	2
Simulation time [s]	70	70
# of nodes in HL	35	35
# of episodes	3000	3000
C_θ	-	0.043
C_v	-	0.5

12.1.1 *Linear Reward*

A training process of a PPO agent using the linear reward function is displayed in Figure 32 and the results obtained by the last trained agent is shown in Figure 33. As seen in Figure 32, both the average and episode rewards has a smooth behavior and increases as the number of episodes increases. The bottom right plot shows the different controller gains that the agent chooses in order to learn the optimal policy.

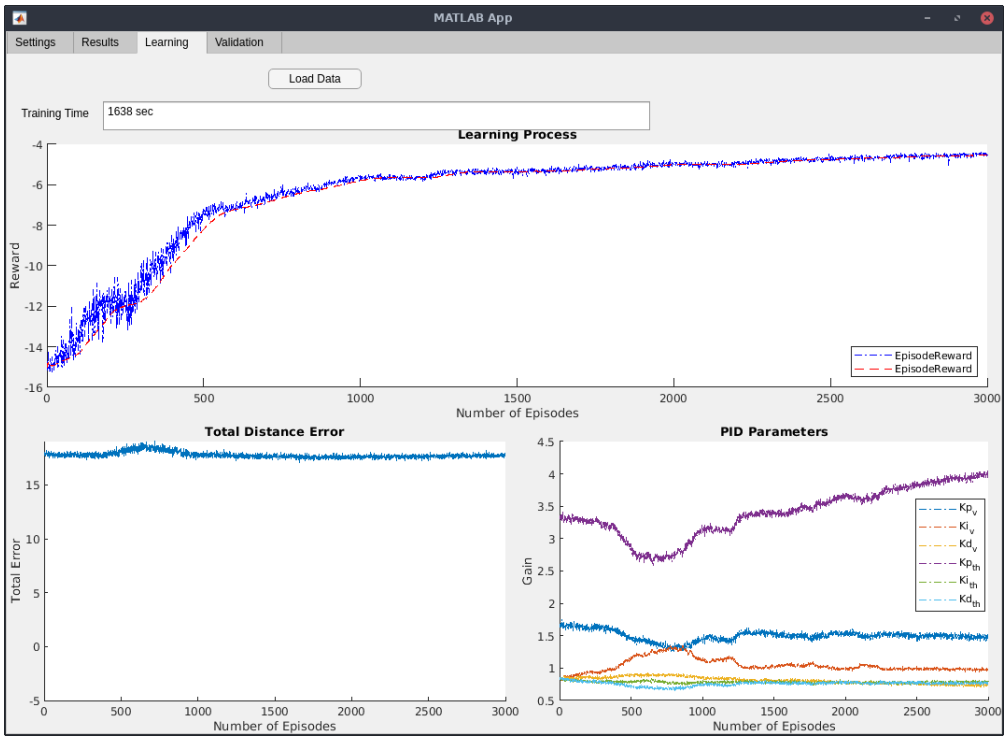


Figure 32: Learning process of a PPO agent using linear reward.

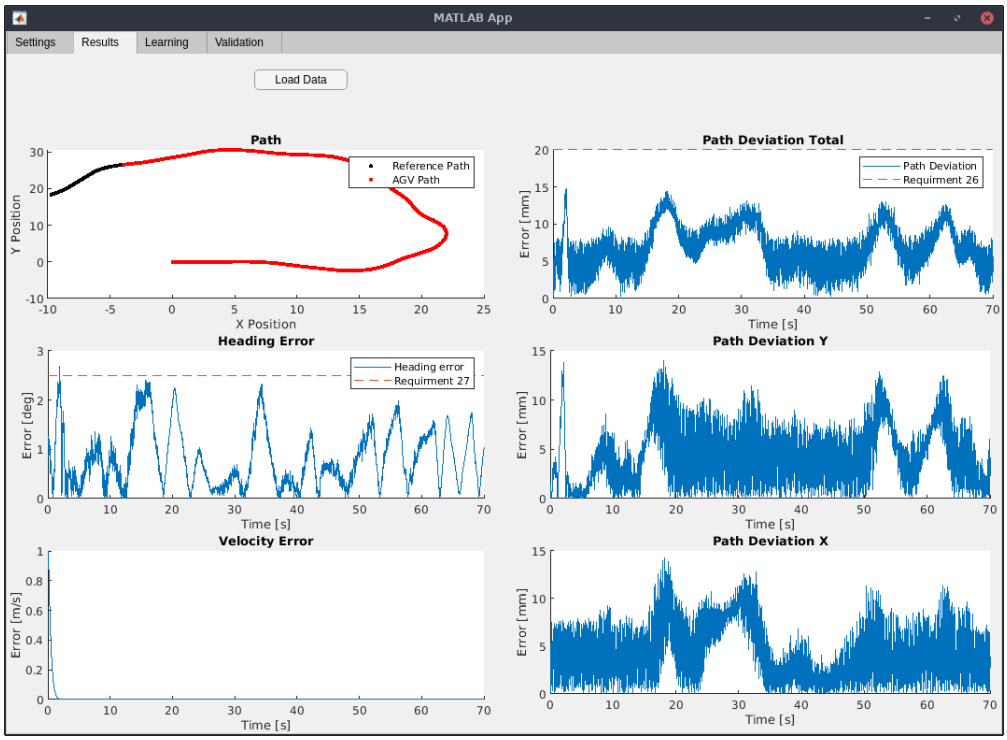


Figure 33: Training results for a PPO agent using linear reward.

12.1.2 *Linear Exponential Reward*

Figure 34 shows training process of a PPO agent using linear-exponential reward function. The results corresponding to this training are displayed in Figure 35. Similar to the linear case, the average reward increases with increasing number of episodes in this case as well, but the behavior of the episode reward is not as smooth as the linear case. There are some differences in the performance of the two agents as well. As seen in Figure 35, the agent using the linear-exponential reward takes much longer time to achieve the desired velocity as compared to the agent that uses linear reward (see Figure 33). These differences however are a direct consequence of the choice of the parameters. In order to get a reward that works well for a specific problem, one should tune these parameters which in most cases is a very time-consuming task.

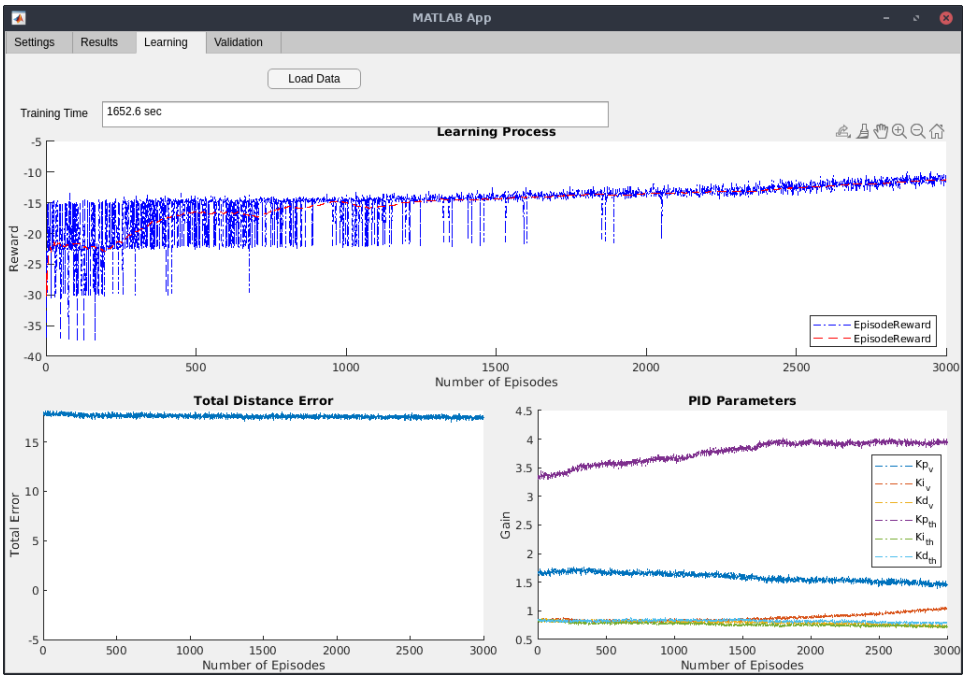


Figure 34: Learning process of a PPO agent using linear-exponential reward.

Figure 35: Training results for a PPO agent using linear-exponential reward.

12.2 Performance

In this section, the performance of the complete system is displayed.

12.2.1 *Static Agent*

The performance of the static agent described in Section 9.0.1 is used as a benchmark to which the performance of the RL agents can be compared. The static agent's controller gains were tuned manually by testing different values until an acceptable performance level was reached. The resulting parameters are displayed in Table 3. In the table, l and r refer to the left and right electric motors and P, I and D denote the proportional, integral and derivative gains of a PID controller (see Figure 21).

Table 3: Controller gains of the static agent.

Controller Gains	Value
K_{P_l}	0.3
K_{I_l}	1.5
K_{D_l}	0
K_{P_r}	5
K_{I_r}	0.1
K_{D_r}	0

The static agent was then used with these parameters to run the simulation 20 times with 20 randomly generated paths using the validation function in the GUI. Figure 36 shows the resulting validation plots in which the requirement associated to each plot is drawn as a dashed red line (see [2]). As seen in this figure, the static agents performs well keeping the velocity error and path deviation under the desired levels but the heading error crosses its threshold in at least five simulations.

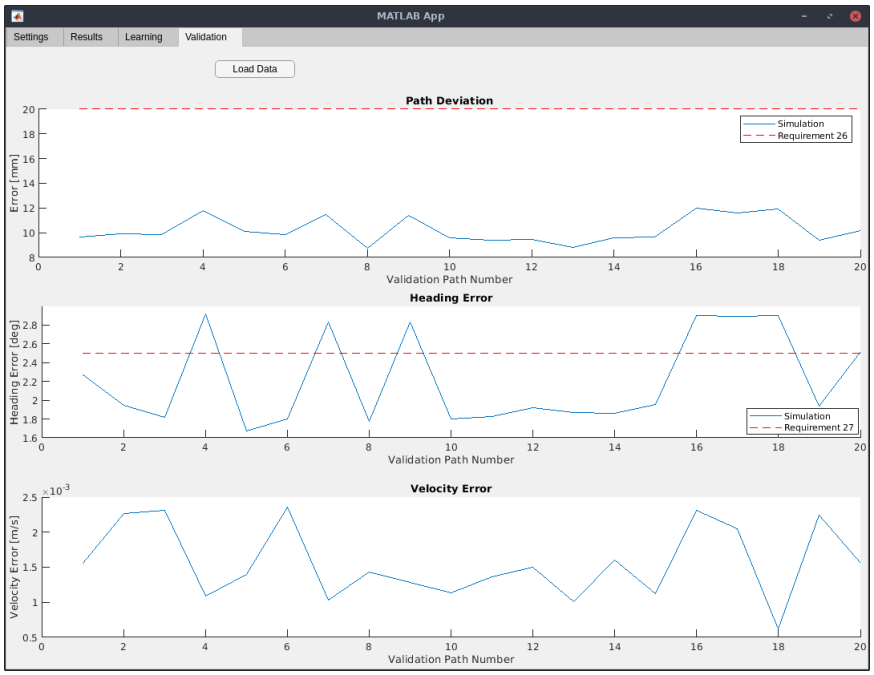


Figure 36: Validation results for the static agent for v = 1 m/s.

12.2.2 *PPO Agent*

The validation results for the PPO agent are shown in Figure 37. The agent was trained for 5000 episodes using the linear reward function. The learning rate was set to 0.001 and the number of nodes in the hidden layers was 35. Table 4 contains all the variables used for this training. The noise is sampled with a sample time of 0.025 s in Simulink. Combining this sample time and the given noise power values results in a standard deviation of 6.32e-4. The validation was performed in the same way as the static agent with 20 randomly generated paths. In contrast to the static agent, the PPO agent meets all the requirements (see [2]).

Table 4: Parameters used to train the PPO agent.

Variable	Value
K_θ	-3
K_v	-10
T_s [s]	2
Simulation time [s]	80
# of nodes in HL	35
# of episodes	5000
T_D left motor	0.025
T_D right motor	0.025
Noise power x pos	1e-8
Noise power x pos	1e-8
Noise power velocity	1e-8
Noise power heading	1e-8

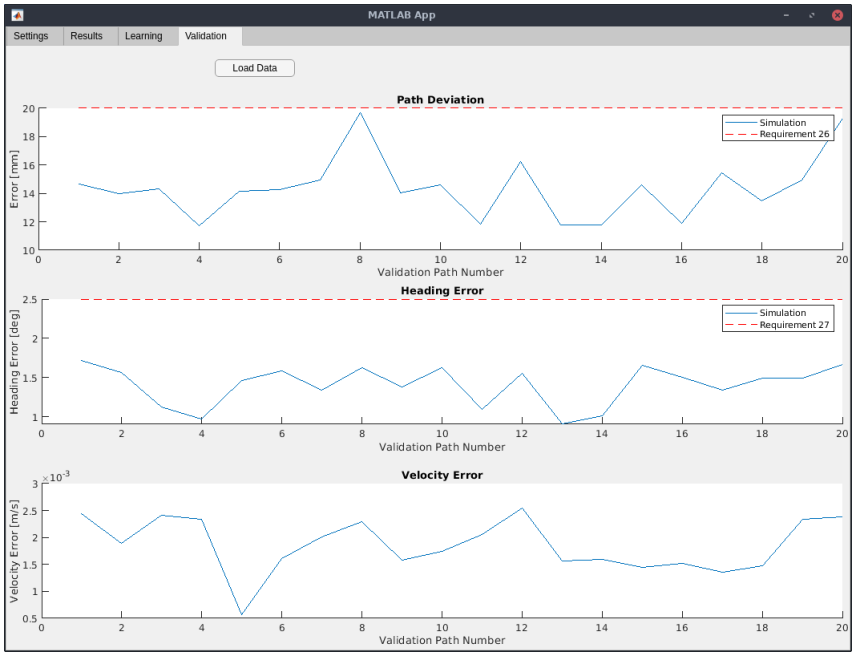


Figure 37: Validation results for the PPO agent for v = 1 m/s.

The generated controller gains by the PPO agent for one simulation are displayed in Figure 38. The oscillations in the gains reveals the stochastic nature of the PPO agent which generates the actions from a normal distribution with a specific mean and standard deviation.

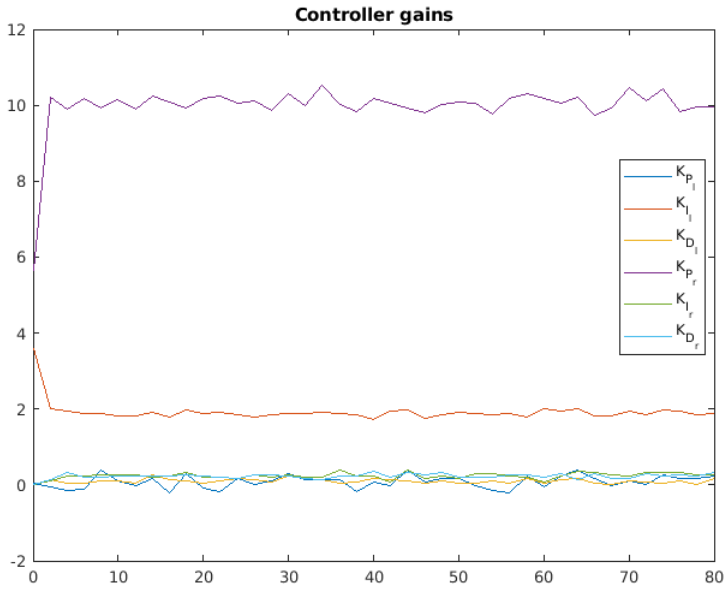


Figure 38: Controller gains generated by the PPO agent during one simulation.

The complete results of the same simulation is shown in Figure 39.

Figure 39: Performance of the PPO agent during one simulation.

12.2.3 *DDPG Agent*

The same validation, as in Section 12.2 was made for the DDPG agent. The result is shown in Figure 40. The agent was trained for 1000 episodes with 20 hidden units for each fully connected layer and a learning rate of $1e-4$ for both the critic and the actor. Figure 40 shows that the maximum error for each validation path is lower than the required value.

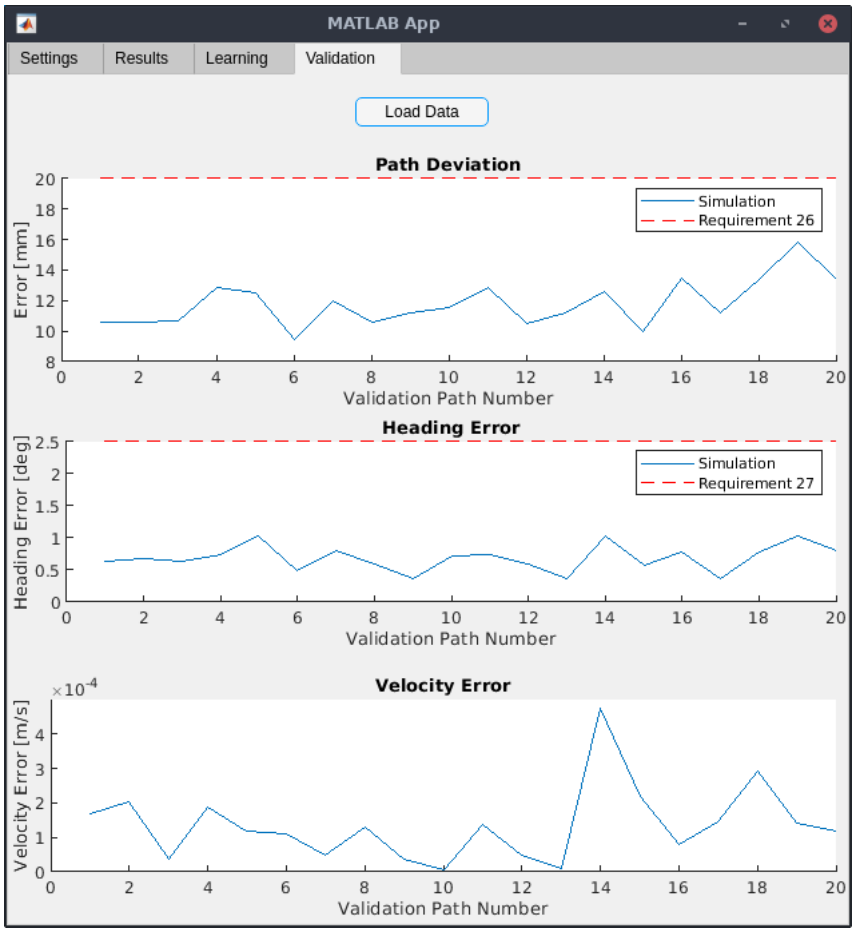


Figure 40: Validation results for the DDPG agent for v = 0.8 m/s.

12.3 Different Environments

In this section examples of randomly generated paths and disturbances are shown. In Figure 41 and 42, four examples of randomly generated *spline paths* (see Section 5.3.1.2) are shown.

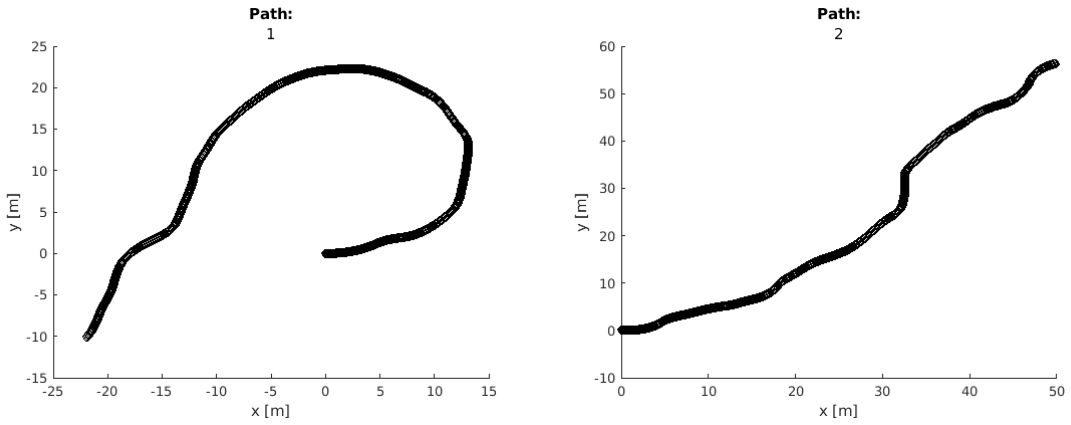


Figure 41: Path 1 and 2 used for training.

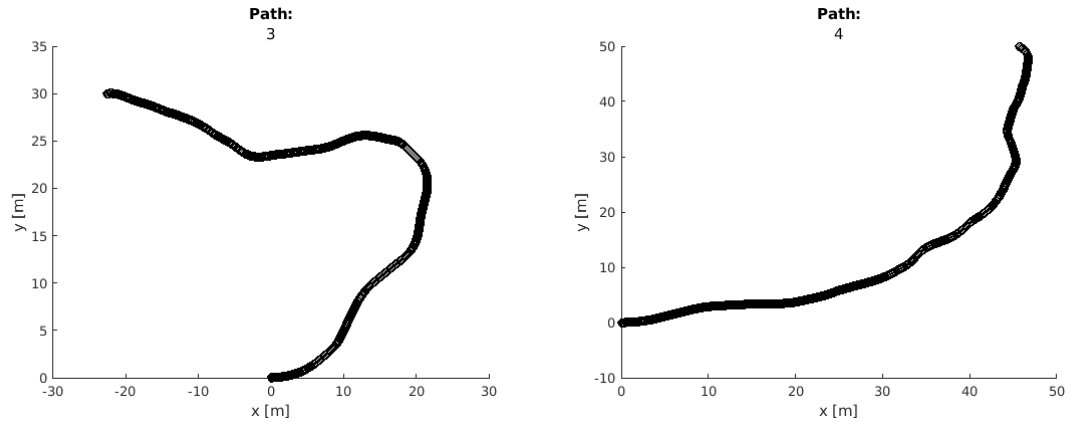


Figure 42: Path 3 and 4 used for training.

In Figure 43 and 44, four examples of randomly generated *rrt* (see Section 5.3.1.1) are shown.

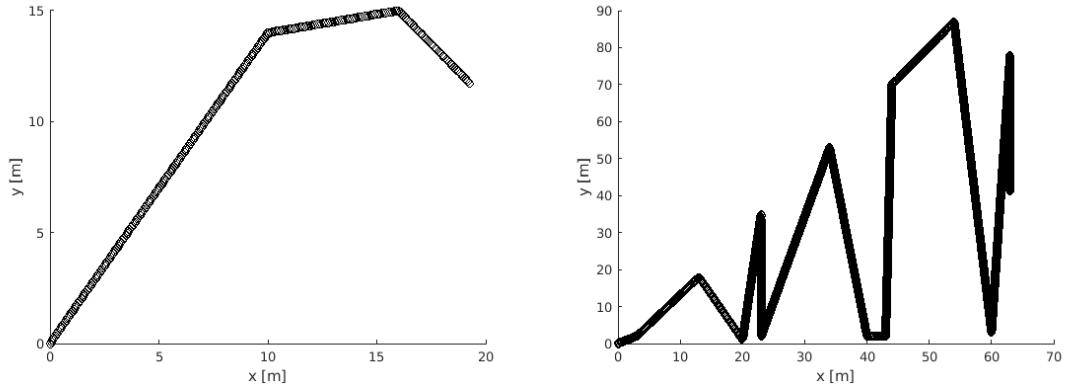


Figure 43: Path 1 and 2 used for training.

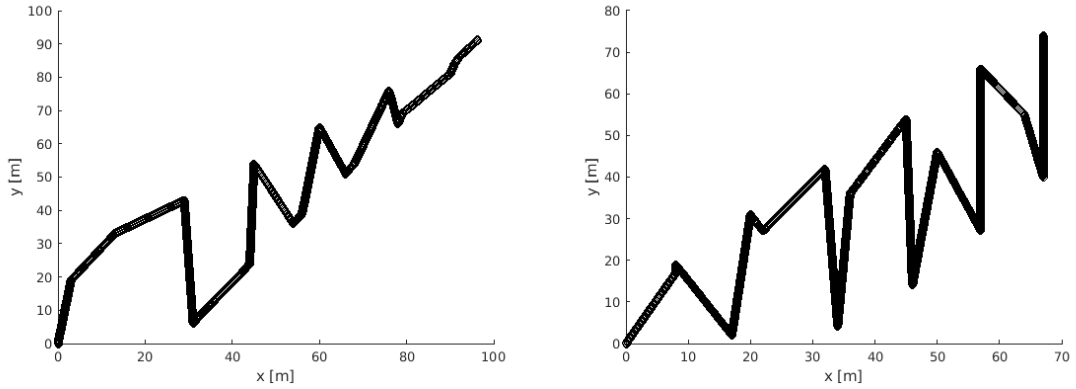


Figure 44: Path 3 and 4 used for training.

In Figure 45 the actual heading over time and the measured heading over time is plotted for one simulation.

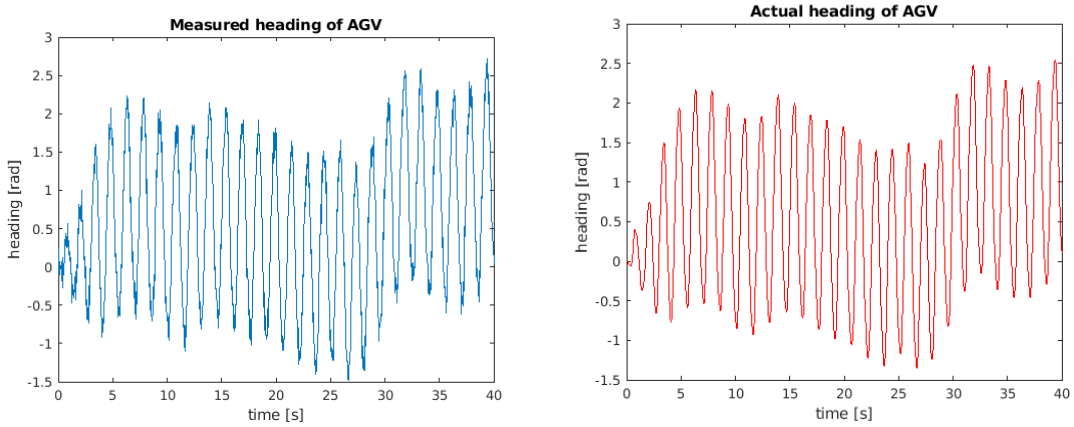


Figure 45: The same simulation with and without added heading measurement noise.

In Figure 46 the actual velocity over time and the measured velocity over time is plotted for one simulation.

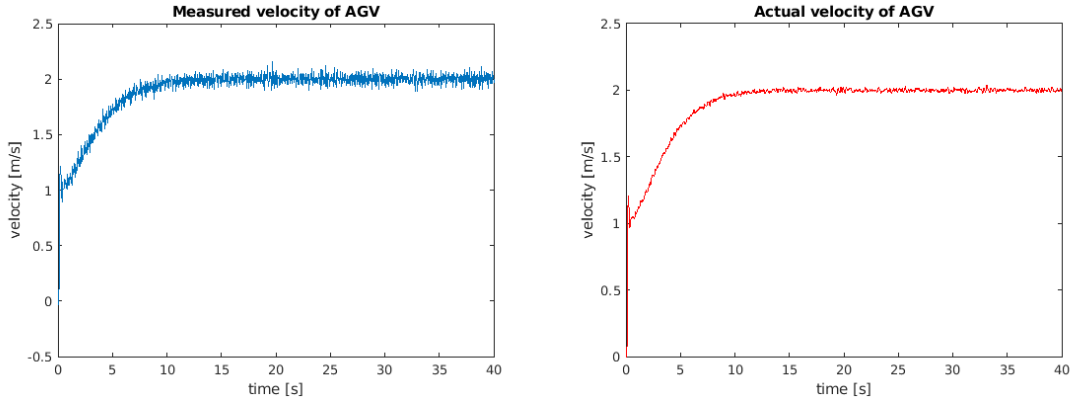


Figure 46: The same simulation with and without added velocity measurement noise.

In Figure 47 the actual position over time and the measured position over time is plotted for one simulation.

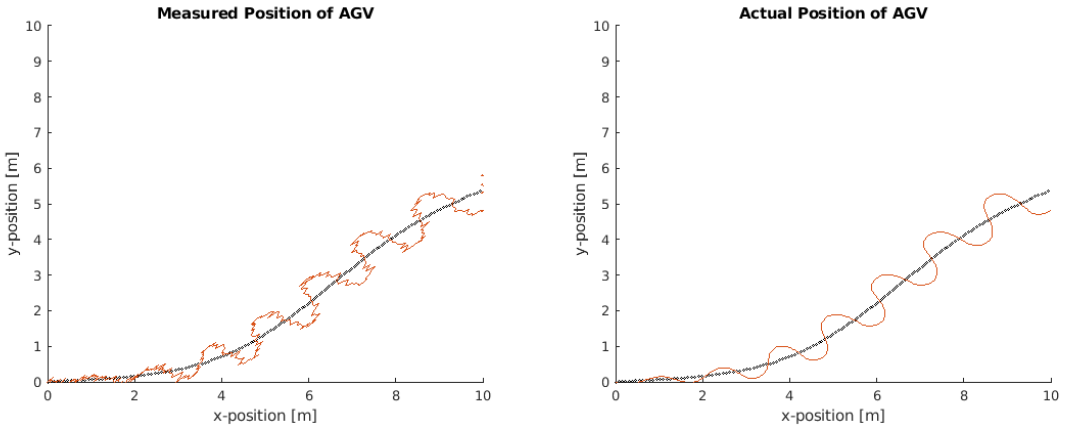


Figure 47: The same simulation with and without added position measurement noise. The black line shows the reference path of the AGV. Note that a large noise power was used in order to clearly show the noise.

13 DISCUSSION

From the results presented in Figures 36, 37 and 40 one can see that both the static agent and the two machine learning based agents perform well in controlling the AGV. However, the static agent is not always able to keep the heading error within the requirement. Both of the machine learning based agents are able to keep the path-, heading- and velocity errors within requirements. Because of this, one could say that both the machine learning based agents are better or more robust than the static agent.

Since the auto-tuner only choose tuning parameters, the performance of controlling the AGV will to some degree be constrained of the design of the controller and the path following. For instance, if the design of the controller is bad, tuning the parameters may not help to increase the performance to an acceptable level. This does not mean that the ability to evaluate the machine learning method is lost since the performance of the auto-tuner is compared with the static agent in this work. However, if the static agent would not be able to control the AGV because of poor choice of control design or path following method, the evaluation of the auto-tuner could have been problematic. In light of this, it should be noted that the path deviation metric displayed in Section 12 should be viewed with some caution in regard to the performance of the auto-tuner. The reason for this is that the auto tuner affects the path deviation only indirectly, since both the implemented controller and the auto-tuner is set up to minimize the heading and velocity error. If the generated reference heading and velocity is not sufficiently accurate, the controller could perfectly minimize the heading and velocity error while still maintaining a large path deviation.

The machine learning methods used to tune the PID-parameters themselves require a great deal of tuning and choice of parameters to be able to perform well. This raises the question whether it is worth using machine learning to tune the PID-parameters in the controller, especially when the number of received parameters are few. Another approach could be that instead of using machine learning to tune the PID parameters in an already existing controller, one could simply just use machine learning as the controller. This will of course also require tuning, but the process of designing a whole controller is avoided.

13.1 Post-development

There are certain features that the project group would have liked to develop if the project were to continue.

13.1.1 *Path-following*

A significant improvement in the path-following algorithm is ensuring that it can handle crossings. As of now, the path-following algorithm will always choose the closest node; in essence, if a node - a shortcut - exists that is closer to the AGV's position yet which will steer the AGV towards an unintended node, the current path-following algorithm will steer towards this erroneous node. This was also the major flaw of the RRT as its sharp corners often caused the path-follower to take shortcuts in the corners. An improved path-following algorithm would consider the closest node as well as its relative position in the path so as to avoid shortcuts.

13.1.2 *Physical Electrical Motor Development*

An interesting further work area is the modeling of the electrical motor. Many disturbances can be modeled through the motor, for instance position of the load that the AGV is carrying and chance of frictions between the floor and the wheels. If a physical model of the motor, for instance a state space model, was developed, measurements of the wheel speed and estimations of the required torque could perhaps be used as observations to the agent. Then the agent could be trained to recognize these disturbances and improve its ability to handle these scenarios. This would make the control more robust which could be beneficial in environments with a lot of unpredictable occurrences.

13.1.3 *Machine Learning Methods*

When it comes to the machine learning methods there are quite a bit of things that could be experimented with further. There are a great amount of possibilities when dealing with networks and all the different parameters used for the agent and its training. The architecture of the actor and critic networks can be implemented in many different ways, with different amount and types of layers. To explore several network structures was not a part of this project and could be interesting to develop further. This combined with changing the amount of hidden units could give an agent that performs better.

13.1.4 *Machine Learning Implementation*

In this project the agent gives actions in the form of control parameters that directly controls the AGV. This could easily be changed, with the existing system, to instead tune already existing PID-parameters meaning that the agent only finely tunes the system. Another interesting implementation is to remove the controller completely and let the agent directly give actions as the control signals for the AGV.

13.1.5 *Disturbances*

An observation during the project was a tendency for the agent having trouble in finding good parameter values if there was a large time delay in the motor (see section 5.1.2. Another interesting area of future research would be to do a more formal study if any of the disturbances is more difficult for the agents to handle.

13.1.6 *Real world implementation*

The results of this study is completely based on the simulator output. For a more robust conclusion the agent and training would need to be integrated in a real world system.

14 CONCLUSION

Based on the results obtained, one can say that it is possible to use machine learning methods to tune parameters in a controller, with better results than manual tuning. The method has potential to save a great deal of time and work when designing a controller, but if the number of tuned parameters in the controller is few and the variations of the AGV's environment is not too large, the process can be more time consuming than to tune the parameters manually. Additionally, the project has been possible due to the simulation. If instead the project was for a real AGV to learn its control parameters, in a real/non-virtual environment, the tuning would not be realistic. The amount of iterations needed to tune the parameters would not be attainable in a real scenario. However, if the simulations is sufficiently accurate, then this process for tuning the controller is satisfactory; although it changes the focus of the assignment from machine learning tuning to creating a sufficiently accurate simulation. For a real-world application, it is advisory to combine a simulation environment with a real environment: start by completing a base-tuning in a simulation environment; as the base-tuning is completed, one can then present the AGV in a real environment where it can continue learning over a longer period of time.

REFERENCES

[1] C. H. Heden et al, "Design specification," Oct 2021.

[2] ——, "Requirement specification," Sep 2021.

[3] ——, "User manual," Nov 2021.

[4] OpenAI, "Proximal policy optimization," 2018, accessed: 2021-10-04. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, accessed: 2021-11-29. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[7] OpenAI, "Deep deterministic policy gradient," 2018, accessed: 2021-10-04. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[8] T. Glad, L. Ljung, and A. Hansson, *Modeling and Identification of Dynamic Systemsr*. Studentlitteratur, 2021.

[9] T. Glad and L. Ljung, *Reglerteori: flervariabla och olinjara metoder*. Studentlitteratur, 2003.

[10] M. LaValle S., "Rapidly-exploring random trees: A new tool for path planning," 1998.

[11] OpenAI, "Key concepts in rl," 2018, accessed: 2021-10-06. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#

[12] A. Holgersson and J. Gustafsson, "Trajectory tracking for automated guided vehicle," 2021.

[13] C. H. Heden et al, "Test plan," Oct 2021.

[14] E. Frisk, "Tsfs12 hand-in exercise 3." [Online]. Available: https://gitlab.liu.se/vehsys/tsfs12/-/blob/master/Handin_Exercises/HI3_VehicleControl/exercise3.pdf