HAW HAMBURG

Bachelorarbeit

Henri Burau

Evaluierung einer Simulationsumgebung zur Umsetzung und Entwicklung eines selbstlernenden Reglers für autonome Wasserfahrzeuge

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Henri Burau

Evaluierung einer Simulationsumgebung zur Umsetzung und Entwicklung eines selbstlernenden Reglers für autonome Wasserfahrzeuge

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Tim Tiedemann
Zweitgutachter: Prof. Dr. Stephan Pareigis

Eingereicht am: 25.07.2020

Henri Burau

Thema der Arbeit

Evaluierung einer Simulationsumgebung zur Umsetzung und Entwicklung eines selbstlernenden Reglers für autonome Wasserfahrzeuge

Stichworte

Simulation, Reinforcement Learning

Kurzzusammenfassung

Das Ziel dieser Arbeit ist es zu zeigen, dass sich die vorgestellte Simulationsumgebung zur Umsetzung und Entwicklung eines selbstlernenden Reglers für autonome Wasserfahrzeuge eignet. Als Anwendungsgebiet wird ein bereits bestehendes Miniaturschiff gewählt. Die nötigen Berechnungen für die Simulation von Wasserdynamiken und Verfahren für die Entwicklung von selbstlernenden Algorithmen werden nachfolgend erklärt. Danach wird das Miniaturschiff in einer Simulation abgebildet und ein selbstlernender Regler zur Manövrierung implementiert. Die Leistung des selbstlernenden Reglers und die Genauigkeit der Simulation werden anschließend evaluiert. Die Ergebnisse dieser Evaluation zeigen, dass sich die Simulation aufgrund von verkürzten Entwicklungszyklen und flexiblen Testszenarien für die Entwicklung und Umsetzung eines selbstlernenden Reglers eignet. Außerdem reicht auch die Rechenleistung des Miniaturschiffs um die vorgestellten Softwareprozesse auszuführen. Probleme stellt noch die Genauigkeit der Simulation dar, die nicht abschließend bestimmt werden konnte.

Henri Burau

Title of Thesis

Evaluation of a simulation environment to conceptize and develop a self learning controller for unmanned surface vehicles

Keywords

Simulation, Reinforcement Learning

Abstract

The aim of this thesis is to show that an simulation environment for the implementation and development of a self-learning controller is suitable for autonomous vessels. The area of application is an already existing miniature ship. The necessary calculations for the simulation of water dynamics and methods for the development of self-learning Algorithms are explained subsequently. Afterwards the miniature ship is simulated and a self-learning controller for manoeuvring is implemented. The performance of the self-learning controller and the accuracy of simulation are afterwards evaluated. The results of this evaluation shows that the simulation is suitable due to shortened development cycles and flexible test scenarios for the development and implementation of a self-learning controller. In addition, the computing power of the miniature ship is also sufficient to execute the presented software processes. Problems are still posed by the accuracy of the simulation, which could not be conclusively determined.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

DART Dynamic Animation and Robotics Toolkit.

DDPG Deep Deterministic Policy Gradient.

DOF Degrees of Freedom.

DQN Deep Q-Networks.

HAW Hochschule für Angewandte Wissenschaften.

HIL Hardware-in-the-Loop.

IMU Internal Measurement Unit.

ML Machine Learning.

ODE Open Dynamic Engine.

OSRF Open Source Robotics Foundation.

RL Reinforcement Learning.

ROS Robot Operating System.

RTF Real Time Factor.

SNAME The Society of Naval Architects and Marine.

URDF Universal Robot Describtion File.

1 Einleitung

An der Hochschule für Angewandte Wissenschaften (HAW) Hamburg widmet sich die Forschungsgruppe autonomous systems (autosys) [Pareigis u. a., 2020] unter Anderem der Erforschung von Miniaturautomatisierung. In Zusammenarbeit mit dem "Miniaturwunderland Hamburg"[Miniatur Wunderland, 2020] werden dort autonome Fahrzeuge im Maßstab 1:87 erforscht. Das Miniaturwunderland bietet in seinen Räumlichkeiten ein geeignetes Testumfeld für Land-, Luft- und Wasserfahrzeuge. Neben einer Landschaft mit vielen Straßen und Eisenbahnstrecken gibt es dort auch ein 25,000 l Wasserbecken mit 5 cm Tiedenhub. Das Wunderland verfügt über einige Modellschiffe, die dort von Hand gesteuert das Wasser bevölkern [Miniatur Wunderland, 2020].

1.1 Miniaturschiff MS nHAWigatora

Im Rahmen eines Bachelorprojekts wurde das Miniaturschiff MS nHAWigatora entwickelt. Das Schiff ist im Maßstab 1:100. Es verfügt über einen Raspberry Pi 3 B+ und einen Raspberry Pi 4 als zentrale Recheneinheiten. Die Umgebung wird über eine Reihe von Sensoren wahrgenommen, die Bewegung im Wasser erfolgt durch die Aktorik. Im aktuellen Zustand fährt das Schiff nicht autonom, sondern wird über eine Fernsteuerung gelenkt. Für die Verarbeitung der Sensordaten und die Ansteuerung der Aktorik wird das Robot Operating System (ROS) verwendet [Stark u. a., 2020].

Das Schiff ist momentan mit folgender Aktorik ausgestattet:

Propeller Ein Propeller am Heck des Schiffs sorgt für den Antrieb recht voraus und recht achteraus. Angetrieben wird der Propeller über eine Antriebswelle von einem *IG320005-3AC21R*, der den Propeller mit bis zu 995 RPM antreibt.

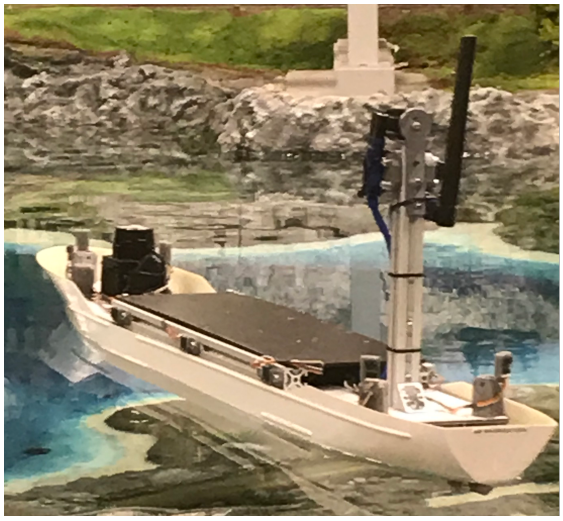


Abbildung 1.1: MS nHAWigatora im Miniaturwunderland Hamburg

Querstrahlruder Das Querstrahlruder am Bug sorgt für die Bewegung nach backbord querab und steuerbord querab. Angetrieben wird der Propeller des Querstrahlruders von einem *SPEED 500 E 12 V Graupner 1788*, der bis zu 12000 RPM erreichen kann.

Ruder Das Ruder am Heck des Schiffes bietet eine weitere Möglichkeit, um nach Backbord oder Steuerbord zu navigieren. Den aktuellen Einschlagswinkel des Ruders stellt ein Servomotor vom Typ *DES 281 BB MG* ein.

Die Umgebung nimmt das Schiff aktuell über folgende Sensorik wahr:

URG-04LX-UG01 Das Lidar von Hokuyo ist am Bug des Schiffs verbaut und nimmt mit einer Frequenz von 10 Hz 1000 Entfernungspunkte mit einer Auflösung von 0,36° auf. Das Sichtfeld des URG-04LX-UG01 beträgt 240° [Hokuyo].

GP2Y0E03 An der Reling sind zusätzlich 11 Infrarotsensoren verbaut, die Entfernungen messen. Die Sensoren haben ein Sichtfeld von $6 \pm 3°$ und nehmen Objekte in einer Entfernung von 4 cm bis 50 cm wahr [Sharp].

daA1920-30uc Die Farbkamera von Basler ist an einem Kameraturm am Heck des Schiffs befestigt. Die Aufnahmen der Kamera haben eine Auflösung von 1920×1080 Pixeln [Basler].

MPU9255 Die Internal Measurement Unit (IMU) von InvenSense hat ein Magnetometer, Accelerometer und Gyroskop [InvenSense].

1.2 Problemstellung

Für die Navigation benötigt ein autonomes Fahrzeug Informationen über Ausrichtung, Geschwindigkeit und Position. Mit diesen Informationen ist eine Regelung der Position möglich. Diese Regelung wird in einem autonomen System von einem sogenannten Regler übernommen. Die Reglersynthese für ein komplexes dynamisches System ist sehr aufwendig. Es muss sich ein umfassendes Wissen über das zu regelnde System angeeignet werden. Das trifft insbesondere auf das dynamische System Wasser zu. Viele Parameter wie Wind, Strömung und Temperatur sorgen dafür, dass dort die Berechnungen sowohl zeitaufwendig als auch komplex sind. Selbst mit einem ausführlichen Wissen über das dynamische System ist es schwer, eine optimale Regelung zu erreichen. In der erarbeiteten Lösung kann leicht eine Wechselwirkung übersehen werden [Hafner, 2009].

Außerdem ist eine einmalige Reglersynthese der MS nHAWigatora nicht ausreichend. Die Nutzung des Schiffs als Forschungsplattform umfasst auch ständige Umbauten und damit Veränderungen am Verhalten des Schiffs. Insbesondere die Aktorik, welche ein potentieller Regler nutzen würde, wurde über die Zeit mehrmals modifiziert. Das erhöht den Zeitaufwand der Bestimmung eines Reglers zusätzlich.

In der Entwicklung und Konzeption des Reglers müssen die oben erwähnten Informationen über Ausrichtung, Geschwindigkeit und Position während der Tests zuverlässig sein. Ansonsten kann keine Aussage über die Wirksamkeit gemacht werden. Außerdem wird das Identifizieren von Fehlerquellen erschwert. Für die Extraktion dieser Daten wurden bereits Arbeiten vorgestellt. Stark [2020] stellt ein monokulares Odometrieverfahren vor, welches die Kamera benutzt um die benötigten Informationen zu berechnen. Obwohl das Verfahren vielversprechend ist, verfügt es nicht über die notwendige Genauigkeit, um als alleinige Quelle für die Odometrie zu dienen. Das Verfolgen der Position des Schiffs von außerhalb ist ebenfalls nicht möglich, da es für das Becken des Miniaturwunderlands keine funktionierende Trackinganlage gibt.

Weiterhin ist die Entwicklungszeit bei Tests in der realen Welt zu groß. Die nötigen iterativen Verbesserungen lassen sich nicht schnell umsetzen, steht nicht ein geeignetes Testumfeld zur Verfügung. Obwohl die nHAWigatora im Miniaturwunderland Hamburg

regelmäßig zu Wasser gelassen werden kann, so ist es zeitaufwendig, das Schiff zwischen Hochschule und Wunderland zu transportieren. Ein tägliches Testen, wie es in der Entwicklungszeit möglicherweise nötig wäre, ist organisatorisch zu aufwendig. Zusätzlich ist die nHAWigatora das Ergebnis einer aufwendigen Arbeit über mehrere Jahre mit mehreren tausend Euro finanziellem Aufwand. Es besteht das Risiko, das Schiff durch einen fehlerhaften Regler zu versenken und dadurch einen hohen Arbeits- und Kostenaufwand zu erzeugen.

1.3 Ziel und Aufbau dieser Arbeit

1.3.1 Zielsetzung

Zur Reduzierung des Entwurfsaufwands für den Regler wird eine intelligente Reglerkomponente vorgeschlagen. Diese soll aus vorhergegangenen Regelzyklen selbstständig eine optimale Regelung erlernen, wodurch ein Expertenwissen über das dynamische System für die Konzeption des Reglers entfällt.

Das Problem der Regelung lässt sich in einen Markov-Decision-Process umwandeln. Dadurch lässt sich das Problem als ein Reinforcement Learning (RL)-Problem definieren [Sutton und Barto, 2018].

Als RL-Verfahren bietet sich zum Beispiel Deep Q-Networks (DQN) an. DQN eignet sich insbesondere für kontinuierliche Zustandsräume, wie im Fall der MS nHAWigatora [Mnih u. a., 2015].

Um eine zeitsparende und risikoarme Entwicklung des Reglers zu ermöglichen, wird außerdem eine Simulationsumgebung vorgeschlagen. Diese ermöglicht schnelle Tests und erfordert keine Hardware. Außerdem lassen sich in der Simulation Lernzyklen schneller absolvieren, indem die geeigneten Lernumgebungen schneller als Realzeit simuliert werden.

Ziel dieser Arbeit ist es, eine Simulationsumgebung für die MS nHAWigatora zu entwickeln. Die Simulationsumgebung soll anschließend dahingehend evaluiert werden, ob sie sich als Lernumgebung für einen selbstlernenden Regler eignet. Dies geschieht am Beispiel eines selbstlernenden Reglers, der nach dem DQN-Verfahren auf der MS nHAWigatora implementiert wird. Als Maß dafür wird die Leistungsfähigkeit des selbstlernenden Reglers sowie die Genauigkeit der Simulation genommen.

1.3.2 Aufbau

Diese Arbeit widmet sich zuerst den Grundlagen, die für das weitere Verständnis der Arbeit relevant sind. Danach wird erläutert, wie die Simulation umgesetzt wurde. Anschließend wird der Softwareentwurf des intelligenten Reglers vorgestellt. Anschließend werden Simulation und Regler evaluiert und die Ergebnisse vorgestellt und diskutiert. Abschließend wird ein Fazit unter die Arbeit gezogen.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen erläutert, die für das weitere Verständnis dieser Arbeit essentiell sind. Es wird zu Beginn das Konzept des Verstärkenden Lernens präsentiert, auf welchem der selbstlernende Regler der MS nHAWigatora basiert. Danach werden die physikalischen Grundlagen erläutert, die für das Verständnis der Simulation notwendig sind.

2.1 Verstärkendes Lernen

Dieser Abschnitt erklärt das Konzept des Verstärkenden Lernens (engl. Reinforcement Learning (RL)). RL ist ein Machine Learning (ML)-Verfahren in dem ein lernendes System durch Interaktion mit seiner Umgebung lernen soll, ein bestimmtes Ziel zu erreichen. Das System bekommt von der Umgebung eine Bewertung seiner Aktionen und versucht durch die Wahl seiner Aktionen diese Bewertung zu maximieren. Die Aktionen werden dem System nicht vorgegeben. Stattdessen probiert es verschiedene Aktionen aus, um herauszufinden, welche Aktion zu einer maximalen Bewertung führt. Durch die Aktionen wird auch die Umgebung verändert, das lernende System muss dann in dieser neuen Situation lernen, die bestmögliche Aktion durchzuführen. Damit ähnelt RL sehr dem Verfahren, welches Menschen oder viele Tiere benutzen um ihre Umgebung kennenzulernen und mit ihr zu interagieren [Sutton und Barto, 2018].

Verwendet ein System einen RL-Algorithmus, so wird die lernende Komponente als *Agent* bezeichnet. Die Umgebung wird *Environment* genannt. Zusätzlich zu diesen beiden Begriffen gibt es vier weitere, die RL charakterisieren: Eine *Policy*, ein *Reward Signal*, eine *Value Function* und, optional, ein *Model* der Umgebung [Sutton und Barto, 2018].

Policy Beschreibt das Verhalten des *Agent*. Mit der *Policy* wird entschieden, welche Aktion in einem Zustand ausgeführt werden soll. Die *Policy* kann eine einfache Funktion oder Lookup-Table, aber auch eine komplizierte Berechnung sein.

Reward Signal Beschreibt das Ziel eines RL-Problems. Nach jeder Aktion enthält der *Agent* ein *Reward Signal*, eine einfache Zahl, die seine Aktion bewertet. Das einzige Ziel des *Agenten* ist es, das *Reward Signal* zu maximieren. Für den *Agenten* sind Aktionen der einzige Weg, das *Reward Signal* zu beeinflussen.

Mit dem *Reward Signal* wird die *Policy* indirekt verändert. Wenn eine gewählte Aktion ein geringes *Reward Signal* bekommt, wird die *Policy* dahingehend abgewandelt, eine andere Aktion in der Situation zu wählen.

Value Function Bewertet einen Zustand im Hinblick auf mögliche zukünftige Belohnungen. Durch die *Value Function* kann der Wert (Value) eines Zustands festgestellt werden. Die *Value Function* gibt nicht nur Informationen über das direkte *Reward Signal*, das dem aktuellen Zustand folgt, sondern auch, wie hoch die Belohnung möglicher Folgezustände ist. So kann das *Reward Signal* eines Zustands niedrig sein, sein Wert der *Value Function* jedoch hoch, weil auf den Zustand Zustände mit einem hohen *Reward Signal* folgen.

Model Beschreibt das Verhalten der Umgebung in manchen RL-Verfahren. Das *Model* trifft Vorhersagen darüber, wie die Umgebung in einer bestimmten Situation auf eine Aktion reagiert, bzw. welche Folge-Situation daraus entsteht. RL-Verfahren, die ein *Model* benutzen, werden *model-based* RL-Verfahren genannt.

[Sutton und Barto, 2018]

2.1.1 Interaktion mit der Umgebung

Der *Agent* und seine Umgebung interagieren in diskreten Zeitschritten $t = 1, 2, 3, 4...$ miteinander. Zu jedem Zeitschritt bekommt der *Agent* eine Repräsentation des Zustands, $S_t \in \mathcal{S}$ wobei $\mathcal{S}$ die Menge der möglichen Zustände ist. Basierend auf S_t wählt der *Agent* eine Aktion, $A_t \in \mathcal{A}(S_t)$, wobei $\mathcal{A}(S_t)$ die Menge der möglichen Aktionen in S_t ist. Einen Zeitschritt später, als Konsequenz seiner Aktion, bekommt der Agent einen skalaren *Reward*, $R_{t+1} \in \mathcal{R} \subset R$. Die Abbildung 2.1 veranschaulicht diese Interaktion [Sutton und Barto, 2018].

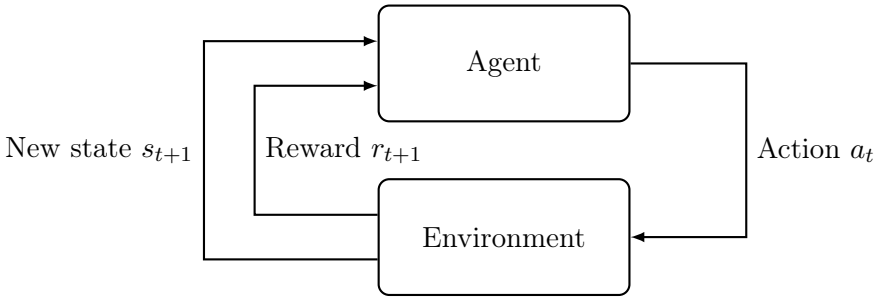


Abbildung 2.1: Interaktion zwischen dem Agenten und der Umgebung. Nach Sutton und Barto [2018]

2.1.2 Q-Learning

Q-Learning ist ein RL-Verfahren, in dem inkrementell die erwarteten *Values* für alle Aktionen in einem Zustand aktualisiert werden. Für jeden möglichen Zustand werden alle möglichen Aktionen in dem Zustand mit einer *Value* belegt. Diese *Value* enthält sowohl den direkten *Reward*, als auch den erwarteten zukünftigen *Reward*, basierend auf dem neuen Zustand, zu dem die gewählte Aktion führt. Dieses Verhalten wird durch die Q-Upate Formel ausgedrückt,

$$Q(s,a) := Q(s,a)(1-\alpha) + \alpha(R + \gamma \max Q(s_{t+1}, a)), \tag{2.1}$$

wobei Q der erwartete Wert beim Ausführen der Aktion a im Zustand s ist, R der erwartete *Reward*, α die Lernrate und γ der *discount factor*. Die Lernrate bestimmt den Einfluss neu gesammelter Erfahrungen. Bei einer Lernrate $\alpha = 0.6$ wird der neue Q-Wert zu 60% aus der neuen und zu 40% aus der alten Erfahrung berechnet. Der *discount factor* bestimmt, wie weit mögliche Folgezustände in den neuen Q-Wert einfließen sollen. Bei einem *discount factor* von 0.5 fließt die Belohnung des nächsten Folgezustands noch zu 25% mit ein. Beim Q-Learning nähert sich die Q-Funktion direkt q_*, der optimalen *policy* an [Watkins und Dayan, 1992].

Q-Learning ist sehr lernintensiv, da alle möglichen Aktionen in jedem Zustand besucht werden müssen um die optimale *policy* zu erreichen [Watkins und Dayan, 1992].

Beim klassischen Q-Learning werden die Zustands- und Aktionspaare in einer Tabelle gespeichert. Dies ist nicht anwendbar auf hochdimensionale kontinuierliche Zustandsräume. Um Q-Learning anwendbar auf diese Zustandsräume zu machen, kann beispielsweise

eine künstliches neuronales Netz als Funktionsapproximator genutzt werden [Mnih u. a., 2015].

2.1.3 Künstliche neuronale Netze

Für die Nutzung von Q-Learning in kontinuierlichen Zustandsräumen ist es nötig, die Zustände der Umgebung zu generalisieren. Zustände, die sich ähneln, erfordern häufig auch ähnliche Aktion. Wird der Zustandsraum nicht generalisiert, muss der Agent jeden einzelnen Zustand erforschen, ohne von der Erfahrung ähnlicher Zustände profitieren zu können. Bei einem kontinuierlichen Zustandsraum steigt der Lernaufwand dabei ins Unendliche. Für die nötige Zustandsapproximierung eignen sich beispielsweise künstliche neuronale Netze, die häufig in verstärkenden Lernalgorithmen eingesetzt werden [Mnih u. a., 2015]. Neuronale Netze bestehen aus einzelnen Neuronen, die miteinander verknüpft werden und so einem Gehirn nachempfunden sein sollen [Haykin, 2009].

Künstliche Neuronen

Das künstliche Neuron ist die kleinste Einheit der neuronalen Netze. Abbildung 2.2 zeigt die schematische Darstellung eines künstlichen Neurons.

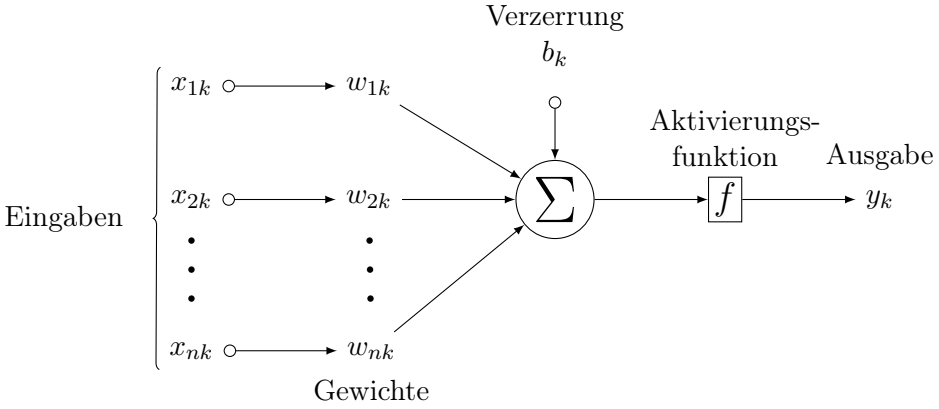


Abbildung 2.2: Ein künstliches Neuron

Ein biologisches Neuron besitzt Dentriten, die elektrische Signale von anderen Neuronen annehmen. Wenn die Summe der elektrischen Signale einen bestimmten Schwellwert überschreitet, feuert das Neuron und propagiert ein elektrisches Signal durch sein Axon und seine Synapsen. Die Synapsen sind wiederum mit den Dentriten anderer Neuronen

verbunden. Die Verbindung der Neuronen formt ein massives Netzwerk, das viele Eingaben parallel verarbeiten kann. Künstliche Neuronen sind biologisch motiviert und ähneln deswegen in der Funktionsweise den Neuronen unseres Gehirns [Haykin, 2009].

Die Ausgabe eines künstlichen Neurons k wird durch seine Eingänge x_{nk} bestimmt. Der Summierer summiert die gewichteten Eingänge auf und addiert dann die Verzerrung. Dieser Wert wird als die Aktivierung eines Neurons bezeichnet, welcher dann in die Aktivierungsfunktion eingesetzt wird. Dies ist dann die Ausgabe y_k des künstlichen Neurons. Mathematisch lässt sich ein Neuron mit n Eingängen x und Gewichten w, der Verzerrung b und Aktivierungsfunktion $f(x)$ definieren als:

$$\begin{aligned} \text{Aktivierung} &= \sum_{i=0}^{n} x_i w_i + b \\ \text{Ausgabe} &= f(\text{Aktivierung})\,. \end{aligned}$$

Die Gewichte w eines Inputs x können sowohl positive als auch negative skalare Werte sein und bestimmen den Einfluss des Inputs auf den Output des Neurons [Haykin, 2009].

Die Aktivierungsfunktion definiert die Ausgabe eines Neurons. Es gibt grundsätzlich zwei verschiedene Arten von Aktivierungsfunktionen:

Schwellwertfunktionen die ab Erreichen eines Wertes eins ausgeben und ansonsten null. Mathematisch definiert als:

$$\varphi(v) = \begin{cases} 1 & if\ v \geq 0 \\ 0 & if\ v < 0 \end{cases}$$

Sigmoide Funktionen die eine S-Form bilden und am häufigsten in Neuronalen Netzwerken verwendet werden. Ein Beispiel für eine sigmoide Funktion ist die *logistische Funktion*, wie Abbildung 2.3 dargestellt.

Sigmoide Funktionen werden häufig genutzt, da sie (anders als Schwellwertfunktionen) differenzierbar sind. Diese Eigenschaft ist notwendig für den Lernprozess neuronaler Netze, wie beschrieben in Abschnitt 2.1.3 zu *Backpropagation* Haykin [2009].

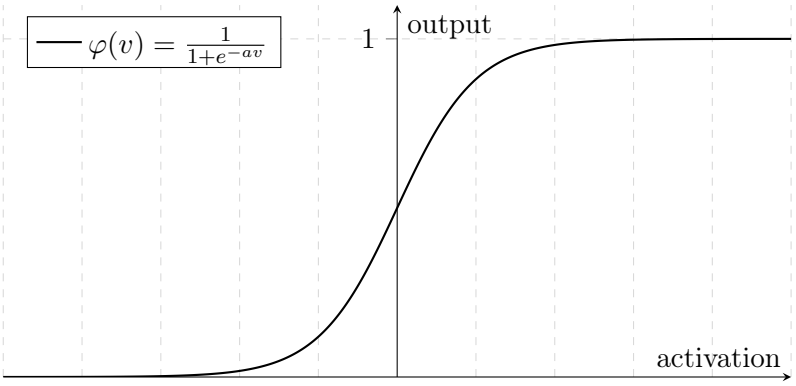


Abbildung 2.3: Logistische Funktion

Multilayer Perceptrons

Das *Multilayer Perceptron* bezeichnet eine Struktur von neuronalen Netzen, in der die Neuronen in *Layern* organisiert sind. Die Neuronen müssen dabei eine Aktivierungsfunktion benutzen, die differenzierbar ist. Der erste *Layer* des Netzwerks wird als *Input Layer* bezeichnet, der letzte als *Output Layer*. Die Ausgabe des *Output Layers* bestimmt die Ausgabe des gesamten Netzwerks. Zusätzlich muss es mindestens einen *Hidden Layer* geben, dessen Neuronen nicht zum Eingabe- oder Ausgabe-*Layer* gehören. Die Ausgabe der Neuronen ist mit jedem Neuron des folgenden *Layers* verknüpft. Abbildung 2.4 zeigt die schematische Darstellung eines *Multilayer Perceptrons*.

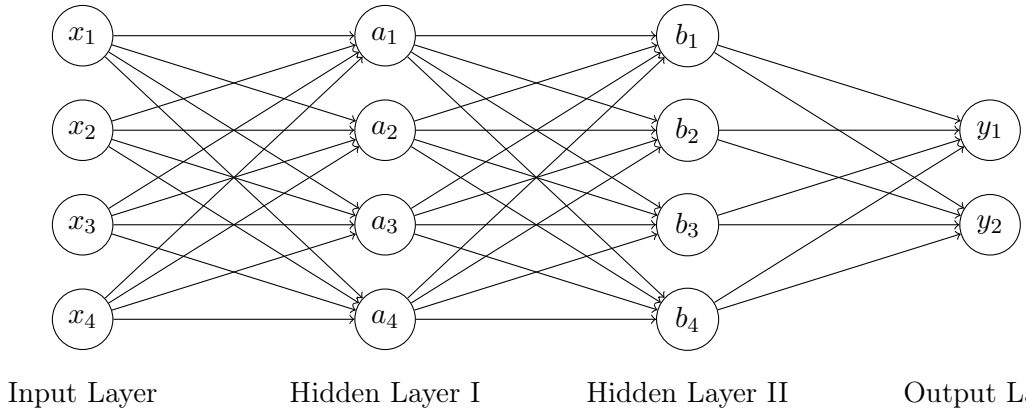


Abbildung 2.4: Ein multilayer Perceptron mit zwei Hidden Layers

Die *Hidden Layers* sollen in dieser Netzwerkstruktur als *Feature Detectors* dienen. Nach mehreren Durchläufen des Lernprozesses entdecken die versteckten Neuronen nach und nach die Features, die die Trainingsdaten charakterisieren Haykin [2009].

Backpropagation

Backpropagation ist ein populärer Algorithmus, um ein *Multilayer Perceptron* zu trainieren. Das Training besteht aus zwei Schritten:

- In der *Forward Phase* sind die Gewichte der Neuronen konstant und das Eingangssignal des *Input Layers* wird durch das gesamte Netzwerk, *Layer* für *Layer*, propagiert bis es den *Output Layer* erreicht. Die Ausgabe wird in dieser Phase nur durch die Aktivierung der einzelnen Neuronen und deren Ausgabe bestimmt.
- In der *Backward Phase* wird ein Errorsignal erzeugt, indem die Ausgabe des Netzwerks mit der optimalen Ausgabe verglichen wird. Das Errorsignal wird dann wieder durch das gesamte Netzwerk propagiert, *Layer* für *Layer*, diesmal aber rückwärts, vom *Output Layer* zum *Input Layer*.

Haykin [2009]

Abbildung 2.5 zeigt die beiden Phasen schematisch.

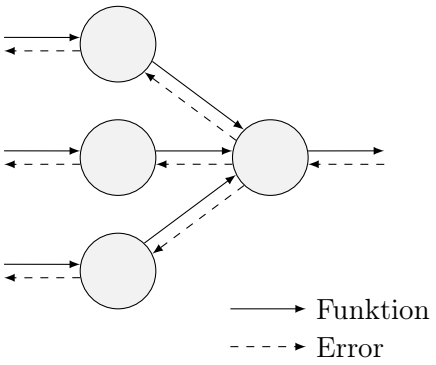


Abbildung 2.5: Darstellung des Error- und Function-Signals. Nach Haykin [2009]

Um ein neuronales Netz mit dem Backpropagation-Algorithmus zu trainieren, ist eine Menge an Trainingsdaten notwendig. Die Trainingsdaten bestehen aus einzelnen Stichproben, *Samples* genannt, die aus einer Eingabe mit seiner gewünschten Ausgabe bestehen. Diese *Samples* werden dann in kleinere Mengen, sogenannte *Batches*, aufgeteilt. Dann wird das neuronale Netz mit den *Batches* trainiert. Für jedes *Sample* in einem *Batch* wird dann der Backpropagation-Algorithmus angewandt Haykin [2009].

2.1.4 Deep Reinforcement Learning

Der naive Ansatz, lediglich die Q-Tabelle durch ein künstliches neuronales Netz zu ersetzen, führt nicht zu dem gewünschten Ergebnis. Überwachte Lernverfahren, wie neuronale Netze, erfordern einen Datensatz mit einer zufälligen Verteilung von Stichproben, sodass jeder *Batch* eine ähnliche Verteilung hat. Außerdem müssen die *Samples* voneinander unabhängig sein. Dies ist nicht der Fall bei RL, dort baut jede Erfahrung (bestehend aus Zustand, Aktion, Belohnung und Folgezustand (s, a, r', s')) aufeinander auf. Das führt zu Problemen: Bekommt der Agent eine hohe Belohnung und trainiert dann das Netz mit der eben gemachten Erfahrung, dann bekommt dadurch der nächste Zustand ebenfalls eine höhere Bewertung. Wird eine ganze Trajektorie so abgespeichert, führt das zu einer Bildung von lokalen Minima.

Zur Lösung dieses Problem wird von Mnih u. a. [2015] ein Deep Reinforcement Learning-Ansatz mit dem Namen Deep Q-Networks (DQN) vorgestellt, der zwei entscheidende Verbesserungen enthält:

Experience Replay Die Historie der vergangenen Erfahrungen (s, a, r', s') wird bis zu einem gewissen Grad gespeichert (zum Beispiel die Erfahrungen der letzten tausend Episoden). Von dieser Historie wird ein *Mini-Batch* mit 32 *Samples* gebildet um das neuronale Netzwerk zu trainieren. Diese Verteilung ähnelt dann einer zufälligen und unabhängigen Verteilung, die für überwachtes Lernen notwendig ist.

Target Network Statt einem neuronalen Netz werden zwei neuronale Netzwerke Θ und $\Theta-$ genutzt. Θ wird genutzt, um die Erfahrungen zu sammeln und auf $\Theta-$ werden diese Erfahrungen gelernt. Nach einer gewissen Anzahl von Episoden wird Θ mit $\Theta-$ synchronisiert. Dadurch wird die Q-Funktion temporär fixiert und verläuft sich nicht in lokalen Minima. Dies wird auch dadurch sichergestellt, dass sich Parameteränderungen nicht sofort auf beide Netzwerke auswirken.

2.2 SNAME Notation

Die The Society of Naval Architects and Marine (SNAME) hat eine Notation für die Bewegung eines Körpers in einer Flüssigkeit festgelegt, diese Notation wird auch in dieser Arbeit verwendet. Die Benennung der Achsen kann der Abbildung 2.6 entnommen werden, die Zeichen der Tabelle 2.1. Hammad u. a. [2017]

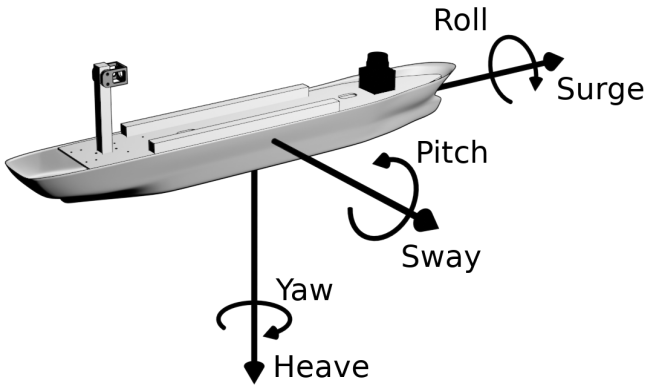


Abbildung 2.6: Freiheitsgrade eines Wasserfahrzeugs

DOF	Kräfte und Drehmomente	Geschwindigkeiten	Positionen und Eulersche Winkel
Surge	X	u	x
Sway	Y	v	y
Heave	Z	w	z
Roll	K	p	Φ
Pitch	M	q	Θ
Yaw	N	r	Ψ

Tabelle 2.1: Die Notation von SNAME für Wasserfahrzeuge

2.3 Trägheitsmoment

Das Trägheitsmoment gibt die Trägheit eines starren Körpers gegenüber einer Änderung seiner Winkelgeschwindigkeit bei der Drehung um eine gegebene Achse an. Damit spielt es die gleiche Rolle, wie die Masse im Verhältnis von Kraft und Beschleunigung. Das Trägheitsverhalten eines starren Körpers kann mithilfe eines Trägheitstensors angegeben werden, für den gebräuchliche das Formelzeichen I genutzt wird. Der Drehimpuls eines Körpers kann wie folgt aus dem Trägheitstensor und der Winkelgeschwindigkeit berechnet werden,

$$\underbrace{\vec{L}}_{\text{Drehimpuls}} = \underbrace{I}_{\text{Trägheitstensor}} \cdot \underbrace{\vec{\omega}}_{\text{Winkelgeschwindigkeit}} \tag{2.2}$$

Dargestellt wird der Trägheitstensor als 3x3 Matrix. Dabei bilden die drei Elemente der Hauptdiagonale die Trägheitsmomente des Körpers um die jeweilige Achse im Koordinatensystem [Goldstein u. a., 2006].

2.4 Maneuvering Theory

In diesem Abschnitt werden die mathematischen Grundlagen für die Simulation von Hydrodynamischen Kräften erklärt. Auf ein Schiff auf der Wasseroberfläche wirken Wind, Wellen und Strömungen ein. Das Schiff bewegt sich dabei mit 6 Freiheitsgraden. Für die mathematische Darstellung von hydrodynamischen Kräften gibt es unterschiedliche Modelle für verschiedene Anwendungsfälle.

Fossen [2011] stellt unter anderem die *Maneuvering Theory* vor. Dieses Modell ist ausgelegt auf ein Schiff, das mit einer konstanten Geschwindigkeit in ruhigem Wasser manövriert. Dieses Modell mit 6 Freiheitsgraden für Wasserfahrzeuge lässt sich ausdrücken als

$$M_{RB}\dot{\nu} + C_{RB}(\nu)\nu + M_A\dot{\nu}_r + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r + g(\eta) = \tau_{antrieb} + \tau_{wind} + \tau_{wellen} \quad (2.3)$$

wobei

$$\eta = [x, y, z, \Phi, \Theta, \Psi]^T \quad (2.4)$$

$$\nu = [u, v, w, p, q, r]^T \quad (2.5)$$

$\nu_r = \nu - \nu_c$
ν_c - Wasserströmung
M_{RB} - Inertialmatrix
M_A - Addierte Masse auf die Inertialmatrix
$C_{RB}(\nu_r)$ - Coriolis- und Zentipetalmatrix
$C_A(\nu_r)\nu_r$ - Addierte Masse auf die Zentrifugal- und Zentipetalmatrix
$D(\nu_r)\nu_r$ - Dämpfungsmatrix

Die Bestandteile der *Maneuvering Theory* setzten sich zusammen aus Festkörper-Kräften, hydrodynamischen Kräften und hydrostatischen Kräften. M_{RB} und C_{RB} bilden zusammen den Teil der Festkörpereigenschaften. M_A, $C_A(\nu_r)\nu_r$ und $D(\nu_r)\nu_r$ bilden den hydrodynamischen und $g(\eta)$ den hydrostatischen Teil [Fossen, 2011].

Die hydrodynamischen Kräfte enthalten die hinzugefügte Masse. Diese wird hinzugefügt, um die Trägheit der umgebenden Flüssigkeit zu repräsentieren, denn durch die hinzugefügte Masse muss mehr Kraft aufgewandt werden, um das Wasserfahrzeug zu bewegen. Das Hinzufügen der Masse geschieht durch M_A und $C_A(\nu_r)\nu_r$. Ein weiterer Bestandteil der hydrodynamischen Kräfte ist die hydrodynamische Dämpfung, welche die Kräfte der

Oberflächenspannung und Hubkräfte enthält. Diese Effekte werden in der Dämpfungsmatrix $D(\nu_r)\nu_r$ ausgedrückt [Fossen, 2011].

Die hydrostatischen Kräfte bilden den Auftrieb des Wasserfahrzeugs ab und werden mit $g(\eta)$ abgebildet [Fossen, 2011].

3 Simulation

In diesem Kapitel wird die Simulation der MS nHAWigatora erläutert. Dafür werden zunächst die Anforderungen der Simulation aufgezählt und dann der gewählte Simulator Gazebo vorgestellt. Dann wird die Darstellung der MS nHAWigatora in dem Simulationsprogramm ausgeführt.

3.1 Anforderungen

Für eine Integration in das bestehende Softwaresystem und zur Reduzierung des Entwicklungsaufwands ergeben sich folgende generelle Anforderungen an die Simulation:

Simulation von Wasser Damit das gelernte Wissen des selbstlernenden Reglers eventuell auch in die Realität übertragen werden kann, muss die Simulation eine ausreichend genaue Repräsentation von Wasser ermöglichen.

3D-Physics Engine Die Simulation zur Umsetzung eines selbstlernenden Reglers könnte auch in einem zweidimensionalen Simulator durchgeführt werden, da die Simulation vor allem in einer Ebene abläuft. Um die Simulation jedoch auch in Verbindung mit den Sensoren wie der Kamera und der IMU nutzen zu können, ist eine realistische dreidimensionale Simulation notwendig.

Integration in ROS Das Softwaresystem auf der MS nHAWigatora basiert auf dem ROS. Um den Entwicklungsaufwand zu reduzieren, muss sich die Simulation gut in dieses System integrieren lassen. Bestenfalls soll die Entwicklung von Navigationsalgorithmen in der Simulation von der Realität so wenig wie möglich abweichen.

Abstraktion der Lern-Schnittstelle Die Simulation soll hauptsächlich zur Entwicklung eines selbstlernenden Reglers genutzt werden. Dies geschieht durch das Erproben

verschiedener Algorithmen aus dem Bereich des verstärkenden Lernens. Die Simulation sollte eine abstrahierte Schnittstelle für Lernalgorithmen zur Verfügung stellen, damit unterschiedliche Algorithmen schnell ausprobiert werden können.

3.2 Gazebo

Gazebo [Open Source Robotics Foundation, 2014] wurde als Simulator gewählt, da es unter den untersuchten Simulatoren die beste Integration für unbemannte Wasserfahrzeuge bietet. Die Simulatoren Webots [Cyberbotics Ltd., 2020], Morse [Echeverria u. a., 2011] und CoppeliaSim [Coppelia Robotics GmbH, 2020] unterstützen ebenfalls Wasseroberflächen oder einfache Fluidsimulationen, aber bei der Recherche konnten keine Beispiele für konkrete Wasserfahrzeuge gefunden werden. Gazebo bietet ohne Erweiterungen keine Möglichkeiten für die Simulation von hydrodynamischen Eigenschaften. Es gibt jedoch Plugins, die Gazebo um diese Eigenschaften erweitern, vorgestellt zum Beispiel von Paravisi u. a. [2019] und Bingham u. a. [2019]

Die Wahl der Physicsengine wird von Gazebo freigestellt; unterstützt werden die 3D-Physicsengines Open Dynamic Engine (ODE) [Russ Smith, 2020], Bullet [Coumans und Bai, 2016–2019], Dynamic Animation and Robotics Toolkit (DART) [Lee u. a., 2018] und Simbody [SimTK, 2020]. Diese Physicsengines sind alle für dreidimensionale Berechnungen ausgelegt und werden auch in der Industrie genutzt, DART wird unter anderem auch für andere RL-Anwendungen genutzt.

Die Entwicklung von Gazebo erfolgt durch die Open Source Robotics Foundation (OSRF), welche auch ROS entwickelt. Deswegen gibt es eine umfassende Integration von Gazebo in ROS und eine große Community, die dafür Anleitungen und Packages veröffentlicht. Dies gilt insbesondere für die Einbindung von unter ROS unterstützen Sensortypen. Gazebo bietet für häufig genutzte Sensoren eine Schnittstelle an, die zum Beispiel die Nutzung von LIDAR, Kamera und IMU-Sensoren erleichtert.

Zum Vergleich und zur Entwicklung von Lernalgorithmen aus dem Bereich des verstärkenden Lernens wurde das Open AI Gym [Brockman u. a., 2016] entwickelt. Durch eine standardisierte Schnittstelle können Algorithmen und Umgebung getrennt voneinander entwickelt werden. Eine Integration von OpenAI Gym in ROS und Gazebo, mit dem Namen *gym-gazebo* wurde bereits entwickelt [Zamora u. a., 2016].

3.3 Architektur

In diesem Kapitel wird eine Architektur vorgestellt, die eine vollständige Simulation der nHAWigatora in Gazebo ermöglicht.

Das bestehende Softwaresystem der nHAWigatora basiert, wie oben erwähnt, auf dem ROS. ROS setzt eine Publisher-Subscriber-Architektur um. Das System besteht aus einzelnen Nodes, die miteinander über Topics kommunizieren. Jedes Topic hat einen Namen, über den die Nodes subscriben oder publishen können. Außerdem haben Topics einen festen Nachrichtentyp. Nachrichten haben immer einen Zeitstempel und einen Link. So kann jede Nachricht einem Punkt im Koordinatensystem und in der Zeit zugeordnet werden.

Für jeden Sensor gibt es auf der nHAWigatora einen eigenen Node. Die Aktoren werden gesammelt über einen einzelnen Node angesteuert. Um die nHAWigatora in der Simulation darzustellen, werden alle Nodes, die Sensoren auslesen oder Aktoren ansteuern, in der Simulation ersetzt, indem die Topics der realen Hardware von Nodes in der Simulation gepublished werden. Eine Übersicht der Nodes ist in Abbildung 3.1 zu erkennen. Alle Nodes mit einem blauen Hintergrund werden von Simulation abgebildet. Auf die Integration der Sensorik wird in Kapitel 7.6.3 weiter eingegangen, auf die der Aktorik in Kapitel 7.6.1.

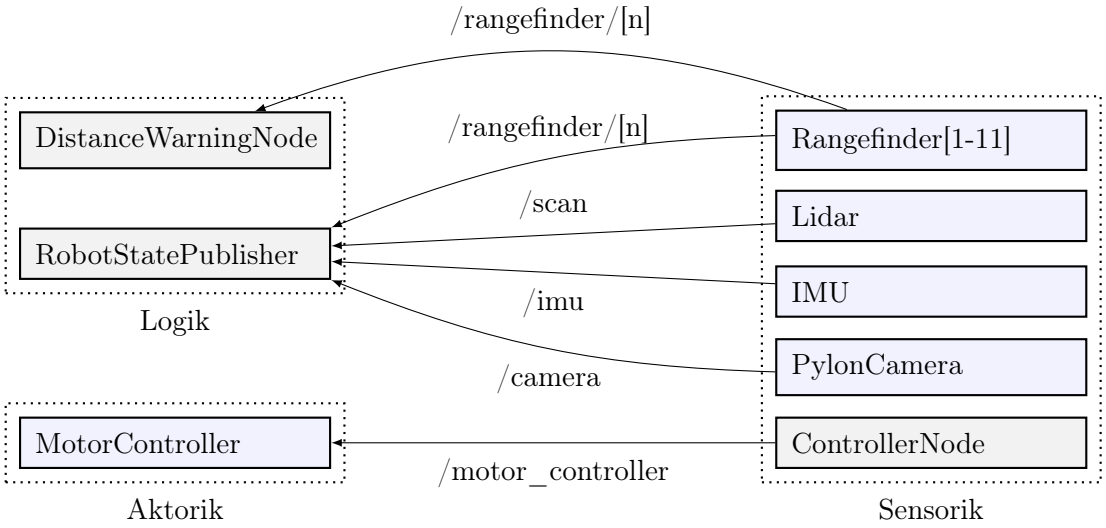


Abbildung 3.1: Struktur der ROS-Nodes auf der MS nHAWigatora

Zusätzlich wird die Starrkörperdynamik des Schiffs simuliert. Dafür werden die vorhandenen Physicsengines von Gazebo verwendet, die eine Beschreibung der physikalischen

Eigenschaften benötigen. Die Parametrisierung der physikalischen Eigenschaften wird in Kapitel 3.4.2 besprochen.

Das dynamische System Wasser wird durch das *USV-Gazebo-Dynamics-Plugin* realisiert, welches ebenfalls in der URDF der nHAWigatora parametrisiert wird.

Abschließend wird in Kapitel 3.4.5 eine Konfiguration der simulierten Welt gezeigt, die sich als Lernumgebung eignet.

3.4 Umsetzung

In diesem Kapitel wird erklärt, wie die nHAWigatora und ihre Komponenten kommunizieren und wie die Komponenten, die mit der Außenwelt interagieren, simuliert werden. Außerdem wird erläutert, wie die Umgebung konfiguriert ist, um sich als Lernumgebung zu eignen.

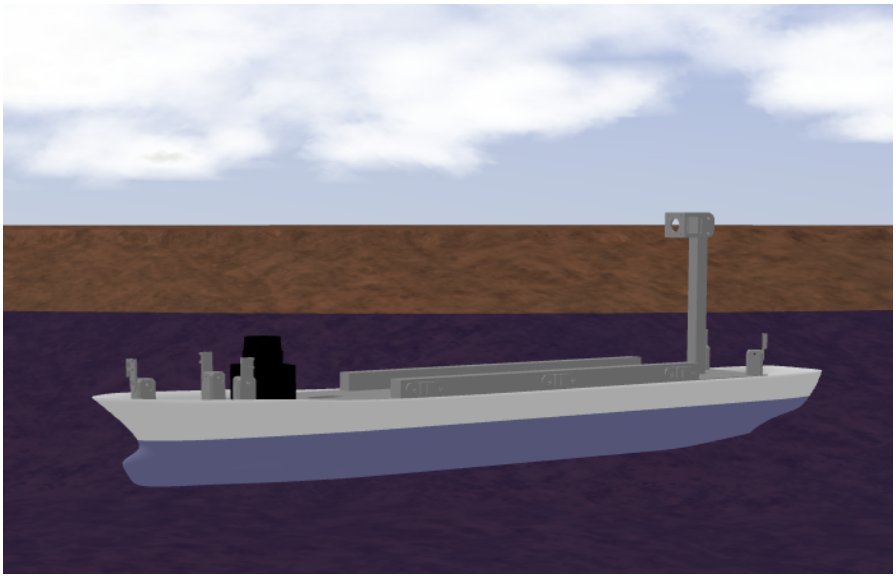


Abbildung 3.2: Modell der MS nHAWigatora in Gazebo

3.4.1 URDF

Das Schiff wird durch eine Universal Robot Describtion File (URDF) in Gazebo repräsentiert. Aus dieser URDF generieren die Programme von ROS und Gazebo eine 3D-

dimensionale Darstellung des Schiffs, die zum Einen der Visualisierung und zum Anderen der physikalischen Berechnungen dient. Die URDF besteht aus Links, die miteinander verbunden sind und jeweils eine visuelle Darstellung und physikalische Eigenschaften definieren. Der unterste Bezugspunkt für alle Links der nHAWigatora ist der *base_link*. Der *base_link* ist wiederum mit dem *base_footprint* verbunden.

Die einzelnen Links in der URDF stellen reale Bauteile auf dem Schiff dar. Um das Verhalten des Bootes so akkurat wie möglich in der Simulation wiederzugeben, ist die genaue Positionierung der einzelnen Bauteile auf der virtuellen nHAWigatora wichtig. Da die Deckplatte, auf der die Sensoren und der Kameraturm montiert sind, mit einer CNC-Fräse produziert wurde, gab es von ihr ein genaues 3D-Modell. Die Bezugspunkte der Bauteile auf der Deckplatte sind, durch die erkennbare Position der gefrästen Montierungslöcher, bis auf wenige Millimeter genau.

Eine Übersicht über den Aufbau der URDF ist in Abbildung 3.3 zu finden. Für die Übersichtlichkeit wurden die *sharp_links* nicht weiter ausgeführt, da insgesamt 11 Sharp-Sensoren auf dem Schiff verbaut sind. Davon befinden sich an den *tray_links* rechts und links jeweils drei Sensoren, drei befinden sich im Bug und zwei im Heck des Schiffs.

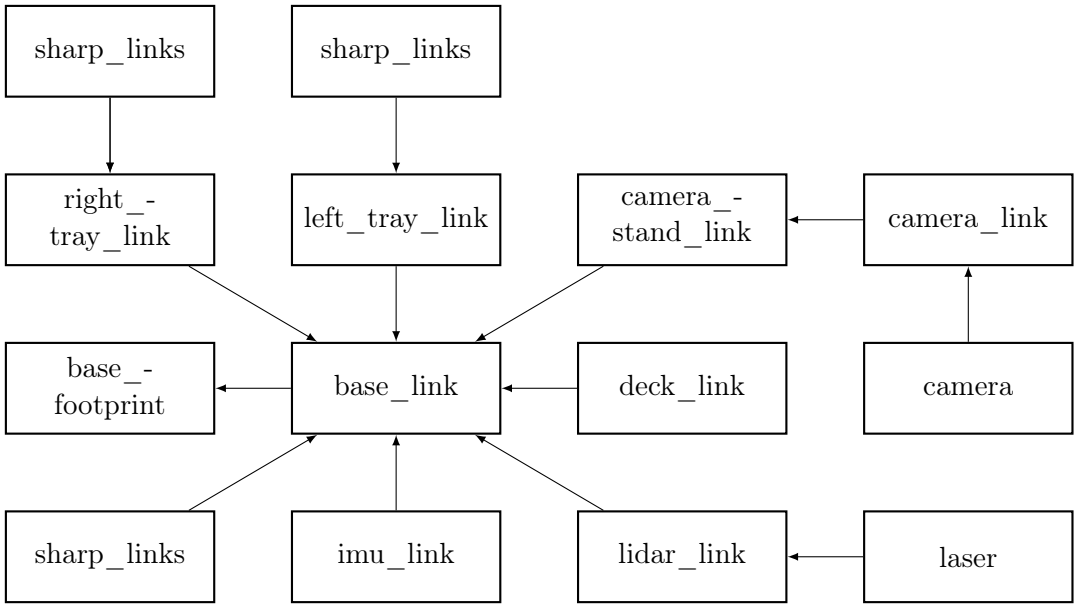


Abbildung 3.3: Aufbau der URDF für die nHAWigatora.

3.4.2 Physikalische Eigenschaften

Als physikalische Eigenschaften lassen sich das Trägheitsmoment, der Massemittelpunkt, die Masse und der Kollisionskörper in Gazebo definieren. Daraus berechnet die gewählte Physicsengine dann die Reaktion des Körpers auf Krafteinwirkungen. An vielen Stellen wird auf Vereinfachungen zurückgegriffen, die zu einer weniger akkuraten Simulation mit einfacheren Berechnungen führt. Dies fördert wiederum die geringe Berechnungsdauer pro Zeiteinheit, welche für eine gute Lernumgebung wichtig ist.

Das Gewicht der nHAWigatora beträgt insgesamt 5,2 kg. In der URDF kann für jeden Link ein Gewicht angegeben werden. Damit die Starrkörperdynamik nicht zu aufwendig zu berechnen ist, wurde nur der *Base Link* mit einem Gewicht versehen.

Das Trägheitsmoment wird mithilfe eines Trägheitstensors angegeben. Da die nHAWigatora aus vielen beweglichen Komponenten besteht, deren Position sich leicht ändern lässt, ist eine genaue Berechnung des Trägheitstensors nicht zweckmäßig. Alleine das Verschieben der Gewichte zum Austarieren, was bei jedem neuen Test im Wunderland gemacht wird, würde ein erneutes Nachmessen des Tensors nach sich ziehen. Stattdessen wird der Trägheitstensor eines Quaders mit derselben Masse, homogener Masserverteilung und denselben Maßen als ausreichend genaue Darstellung gewählt. Der Trägheitstensor I eines solchen Quaders mit der Höhe h in z-Richtung, der Breite b in y-Richtung, der Länge l in x-Richtung und der Masse m lässt sich mathematisch beschreiben als:

$$I = \begin{pmatrix} \frac{m}{12}(b^2 + h^2) & & \\ & \frac{m}{12}(l^2 + h^2) & \\ & & \frac{m}{12}(l^2 + b^2) \end{pmatrix} \tag{3.1}$$

Die für die nHAWigatora bestimmten Werte sind in Tabelle 3.1 zu finden.

Dimension	Wert
Höhe	9,7 cm
Breite	13,9 cm
Länge	96,9 cm
Masse	5,2 kg

Tabelle 3.1: Physikalische Grundwerte der nHAWigatora

Eingesetzt in den formalen Trägheitstensor aus 3.1 ergeben die Werte aus der Tabelle 3.1 den Tensor, mit dem der Trägheitsmoment der nHAWigatora in der Simulation berechnet wird:

$$I = \begin{pmatrix} 0.0125 & 0 & 0 \\ 0 & 0.411 & 0 \\ 0 & 0 & 0.4153 \end{pmatrix} \tag{3.2}$$

Auch für den Kollisionskörper wurde ein Quader mit den Werten aus Tabelle 3.1 als Näherungswert genommen. Hier ist der Hintergrund ebenfalls eine vereinfachte Berechnung. Wird das vorhandene 3D-Modell der nHAWigatora als Kollisionskörper verwendet, wird die Simulationsgeschwindigkeit deutlich verringert. Der Kollisionskörper ist für die Simulation von geringer Bedeutung, da eine akkurate Berechnung der Kollision nicht notwendig ist, sondern nur eine Registrierung von Kollisionen. Da die Berechnungen für Wasserverdrängung nicht mit dem Kollisionskörper zusammenhängen, wird durch einen verbesserten Kollisionskörper die Simulation nicht realistischer.

Die physikalischen Eigenschaften von Flüssikeiten werden nicht nativ von Gazebo unterstützt. Dafür wird das *usv-gazebo-dynamics*-Plugin [Bingham u. a., 2019] verwendet. Dieses Plugin setzt die hydrodynamischen und hydrostatischen Teile der *Meuvering Theory* [Fossen, 2011] um. Für dieses Plugin müssen folgende Parameter für die nHAWigatora bestimmt werden: $X_{\dot{u}}, Y_{\dot{v}}, N_{\dot{r}}, Y_{\dot{r}}, X_u, X_{u|u|}, Y_v, Y_{v|v|}, N_r, N_{r|r|}, Z_w, K_p, M_q$. Diese Parameter bilden die Koeffizienten der Matrizen M_A, $C_A(\nu_r)$ und $D(\nu_r)$, durch die das 6-DOF Maneuvering Modell charakterisiert wird. Die Anordnung der Koeffizienten in der Matrix M_A für hinzugefügt Masse ist

$$M_A = \begin{bmatrix} -X_{\dot{u}} & 0 & 0 & 0 & 0 & 0 \\ 0 & -Y_{\dot{v}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -N_{\dot{r}} \end{bmatrix}. \tag{3.3}$$

Die Coriolis-Zentripetal Matrix $C_A(\nu_r)$ für hinzugefügte Masse ist

$$C_A(\nu_r) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & Y_{\dot{v}} v_r + Y_{\dot{r}} r \\ 0 & 0 & 0 & 0 & 0 & -X_{\dot{u}} u_r \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -Y_{\dot{v}} v_r - Y_{\dot{r}} r & X_{\dot{u}} u_r & 0 \end{bmatrix}. \tag{3.4}$$

Der lineare und quadratische Druck wird in der Matrix $D(\nu_r)$ abgebildet, die sich darstellen lässt als

$$D(\nu_r) = \begin{bmatrix} X_u + X_{u|u|}|u| & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_v + Y_{v|v|}|v| & 0 & 0 & 0 & 0 \\ 0 & 0 & Z_w & 0 & 0 & 0 \\ 0 & 0 & 0 & K_p & 0 & 0 \\ 0 & 0 & 0 & 0 & M_q & 0 \\ 0 & 0 & 0 & 0 & 0 & N_r + N_{r|r|}|r| \end{bmatrix}. \tag{3.5}$$

Die Wahl der Koeffizienten für das Modell der nHAWigatora in Gazebo können der Tabelle 6.7 entnommen werden.

Parameter	Wert
$X_{\dot{u}}$	$1\,\mathrm{kg}$
$Y_{\dot{v}}$	$1\,\mathrm{kg}$
$N_{\dot{r}}$	$0{,}1\,\mathrm{kg\,m^2}$
X_u	$10\,\mathrm{kg\,s^{-1}}$
$X_{u\|u\|}$	$0\,\mathrm{kg\,m^{-1}}$
Y_v	$0{,}1\,\mathrm{kg\,s^{-1}}$
$Y_{v\|v\|}$	$0\,\mathrm{kg\,m^{-1}}$
N_r	$1\,\mathrm{kg\,m^2\,s^{-1}\,rad^{-1}}$
$N_{r\|r\|}$	$0\,\mathrm{kg\,m^2\,rad^{-2}}$
Z_w	$10\,\mathrm{kg\,s^{-1}}$
K_p	$1\,\mathrm{kg\,m^2\,s^{-1}\,rad^{-1}}$
M_q	$1\,\mathrm{kg\,m^2\,s^{-1}\,rad^{-1}}$

Tabelle 3.2: Parameter für die Berechnung der hydrodynamischen Kräfte

Außerdem müssen die Parameter für die Berechnung der hydrostatischen Kräfte angegeben werden. Dazu gehört die Dichte des Wassers, die Wasserhöhe und die horizontale

Fläche des Wasserfahrzeugs. Für die Simulation der nHAWigatora wurde die Wasserdichte auf 997,8 $\mathrm{kg\,m^3}$, die obere Wassergrenze auf 3 m und die horizontale Fläche der nHAWigatora auf 0,12 $\mathrm{m^2}$ festgelegt.

3.4.3 Sensorik

In diesem Abschnitt wird erklärt, wie die Sensorik der nHAWigatora in der Simulation abgebildet wird.

Kamera

Die Kamera wird mit dem Gazebo-Sensortyp *camera* in Kombination mit dem *libgazebo_ros_camera*-Plugin simuliert. Um den vorhandenen *PylonCamera*-Node auf der nHAWigatora zu ersetzen, werden Bilder über das */camera*-Topic verbreitet. Die reale Kamera auf der nHAWigatora ist eine daA1920-30uc der Firma Basler, die Bilder mit einer Auflösung von 1920×1080 Pixeln aufnimmt. Die Kamera in der Simulation nimmt mit der gleichen Auflösung auf und veröffentlicht 20 Bilder pro Sekunde. Die simulierte Kamera wird manuell verzerrt, um die aufgenommenen Bilder mit denen der echten Kamera vergleichbar zu machen. Dafür werden die Verzerrungs-Koeffizienten der daA1920-30uc gewählt. Der Vektor der Verzerrungs-Koeffizienten lässt sich mathematisch beschreiben als:

$$\begin{bmatrix} k_1, & k_2, & p_1, & p_2 \end{bmatrix} \tag{3.6}$$

Eingesetzt mit den Werten der nHAWigatora ergibt sich der verwendete Vektor der Verzerrungs-Koeffizienten (hier gerundet auf drei Stellen):

$$\begin{bmatrix} -0.267, & 0.055, & 0.005, & -0.001 \end{bmatrix} \tag{3.7}$$

Zusätzlich wurde ein Gaußsches Rauschen zur Kamera hinzugefügt, mit einer Standartabweichung von 0.014 um den Mittelwert 0.

Abbildung 3.4 zeigt eine Aufnahme der Kamera einer simulierten Umgebung. In Abbildung 3.5 ist eine Aufnahme der echten Kamera zu erkennen. Zwischen den beiden

Kameras ist ein deutlicher Unterschied zu erkennen, da die simulierte Kamera kein Binning unterstützt.

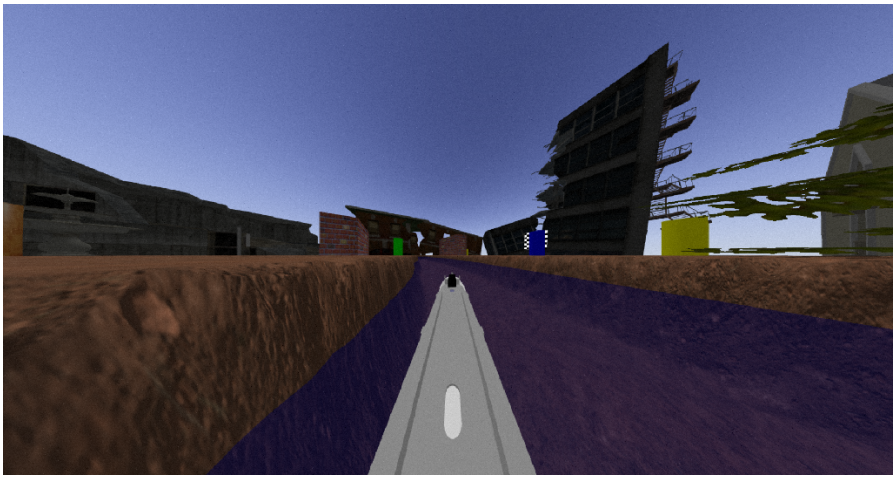


Abbildung 3.4: Aufnahme der simulierten Kamera

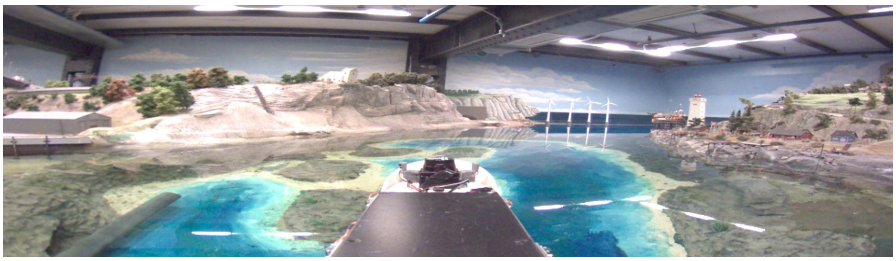
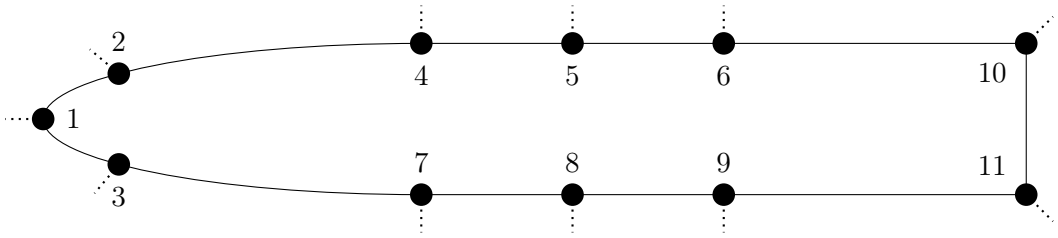


Abbildung 3.5: Aufnahme der Kamera auf der nHAWigatora

Distanzsensoren

Die Distanzsensoren auf der nHAWigatora sind GP2Y0E02A Infrarotsensoren von Sharp. Die Sensoren haben ein Sichtfeld von $6 \pm 3°$ und nehmen Objekte in einer Entfernung von 4 cm bis 50 cm wahr. In dem bestehenden Softwaresystem werden die Nachrichten der Distanzsensoren über die *Range-Message* verbreitet. Um die Rangefinder-Nodes auf der nHAWigatora zu ersetzten, müssen Nachrichten über das */rangefinder/[name]*-Topic zur Verfügung gestellt werden. Auf der nHAWigatora sind insgesamt 11 Distanzsensoren verbaut, in dem bestehenden Softwaresystem werden sie nach ihrer Position benannt. Der Distanzsensor am Bug des Schiffs verbreitet seine Nachrichten zum Beispiel über das Topic */rangefinder/gallion*. Eine Übersicht für alle Distanzsensoren lässt sich in Abbildung 3.6 finden.



1 /rangefinder/gallion
2 /rangefinder/bug_steuerbord
3 /rangefinder/bug_backbord
4 /rangefinder/vorschiff_steuerbord
5 /rangefinder/vorschiff_backbord
6 /rangefinder/mittschiffs_steuerbord
7 /rangefinder/mittschiffs_backbord
8 /rangefinder/pflicht_steuerbord
9 /rangefinder/pflicht_backbord
10 /rangefinder/heck_steuerbord
11 /rangefinder/heck_backbord

Abbildung 3.6: Schematische Darstellung der Position der Distanzsensoren

Für die Darstellung in der Simulation wird der Gazebo-Sensortyp *ray* in Kombination mit dem *libgazebo_ros_range*-Plugin verwendet. Der Sensortyp *ray* wird genutzt, um Distanzsensoren zu simulieren, die Entfernungen mithilfe eines Lasers messen. Dafür wird ein Netz aus horizontalen und vertikalen Strahlen konfiguriert, wie in Abbildung 3.7 zu sehen. Wird ein Strahl unterbrochen, wird das von dem *libgazebo_ros_range*- Plugin registriert und als *Range*-Nachricht über das korrespondierende */rangefinder/[name]*-Topic verbreitet.

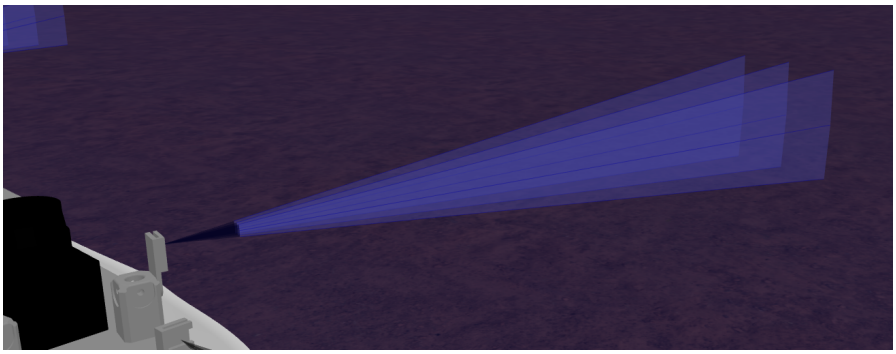


Abbildung 3.7: Visualisierung des Sensortyps *ray* für die Distanzsensoren

Die Parameter der Plugins wurden so gewählt, dass die Sensoren vergleichbare Distanzen und Blickwinkel mit den realen Sensoren haben. Jeder simulierte Sensor verbreitet seine Nachrichten mit einer Frequenz von 10 Hz. Für eine realistischere Abbildung der Sensoren werden die Messdaten durch ein gaußsches Rauschen verzerrt. Dafür wurde eine Normalverteilung mit einer Streuung von 0,132 mm um den Erwartungswert festgelegt.

LIDAR

Auf der nHAWigatora ist das LIDAR URG-04LX-UG01 am Bug des Schiffs verbaut. Die *LaserScan*-Nachrichten des *Hokuyo*-Nodes werden über das */scan*-Topic verbreitet. Für die Abbildung in der Simulation wurde der Gazebo-Sensortyp *gpu_ray* in Verbindung mit dem *libgazebo_ros_gpu_laser*-Plugin verwendet. Der Blickwinkel von 240° des simulierten Sensors entspricht dem realen Gegenstück. Ebenso die Reichweite von 0,2 m bis 4 m. Für eine realistischere Simulation besitzen die Daten außerdem ein normalverteiltes Rauschen mit einer Streuung von 2,538 mm um den Erwartungswert hinzugefügt.

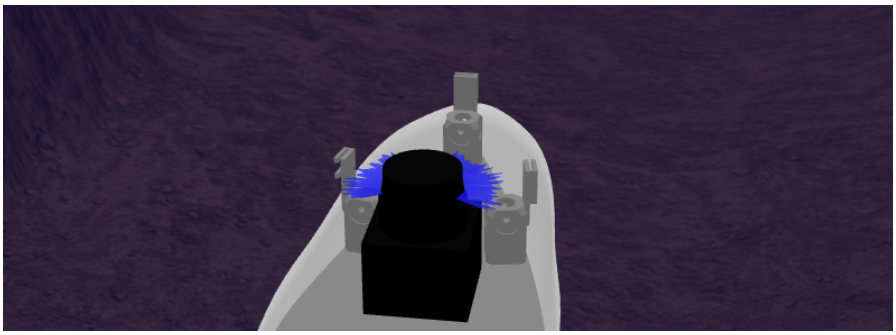


Abbildung 3.8: LIDAR auf der simulierten nHAWigatora

IMU

Die IMU auf der nHAWigatora ist vom Typ MPU9255 und hat 9 Degrees of Freedom (DOF). Die Daten werden mit einem Gyroskop, Accelerometer und Magnetometer erhoben. In ROS sendet der *IMU*-Node *IMU*-Nachrichten an das */imu/raw*-Topic. Um diesen Sensor zu simulierten, wurde das *libhector_gazebo_ros_imu*-Plugin verwendet. Für realistischere Daten wurden die verschiedenen Sensoren mit einem normalverteiltem Rauschen belegt. Der Erwartungswert μ und die Varianz σ der einzelnen Sensoren und Achsen können der Tabelle 3.3 entnommen werden.

	x		y		z	
	μ	σ	μ	σ	μ	σ
Beschleuningssensor (10^{-3} m/s)	0	11.335	0	11.311	0	23.084
Gyroskop (10^{-3} °/s)	1.776	0.849	-2.770	1.018	1.886	0.871

Tabelle 3.3: Dimensionierung des normalverteilten Rauschens

Das gewählte Plugin bietet keine Unterstützung für eine konstante Drift in Yaw-Richtung. Deswegen wird der *imu_drift*-Node eingesetzt, der einen aufsummierten Fehler an in Yaw-Richtung hinzufügt. Dafür wird das gaussche Rauschen in dem *libhector_gazebo_ros_imu*-Plugin für die Drift deaktiviert. Der *imu_drift*-Node subscribed sich dann auf das von dem Plugin veröffentlichte Topic */imu/raw* und rotiert den Orientierungsvektor um ein normalverteiltes Rauschen mit $\mu = 0.01$ und $\sigma = 0.00001$.

3.4.4 Aktorik

Das bestehende ROS-System steuert die Aktorik über den *MotorController*-Node mit Nachrichten vom Typ *Float64* über die beiden Topics */motor_controller_node/set_propeller*, für den Propeller, und */motor_controller_node/set_thruster* für das Bugstrahlruder. Die Aktorik der nHAWigatora wird durch das *usv-gazebo-thrust*-Plugin simuliert.

Die Motoren des Bugstrahlruders und des Propellers haben jeweils unterschiedliche Antriebskräfte. In der Simulation hat der Propeller eine Antriebskraft von 0,1 N vorwärts und 0,1 N rückwärts. Das Bugstrahlruder ist mit 0,6 N nach steuerbord und 0,5 N nach backbord parametrisiert.

Der Wertebereich für die Ansteuerung der Aktorik ist $[-1..1]$. Damit wird die prozentuale Drehzahl der Antriebe bestimmt. Eine Nachricht von 0.2, unter dem Topic */motor_controller_node/set_propeller*, würde also bedeuten 20% der maximalen Umdrehung in Richtung vorwärts, eine Nachricht mit dem Wert -0.5 50% in die umgekehrte Richtung.

3.4.5 Aufbau der Umgebung

Die Umgebung des Schiffs, die der Entwicklung des selbstlernenden Reglers dient, ist zunächst nur eine leere Wasseroberfläche ohne Hindernisse. Der Wasserspiegel wurde auf eine Höhe von 3 m festgelegt. Die physikalischen Berechnungen erfolgen durch die Physicsengine ODE. Damit der selbstlernende Regler schnell verschiedene Szenarien erfahren kann, wurde die *max_step_size* der physikalischen Berechnungen auf 0.01 gesetzt, statt sie auf dem Standartwert 0.001 zu belassen. Das resultiert in einer höheren Geschwindigkeit der simulierten Zeit in Vergleich mit der Realzeit, im Folgenden wird die Metrik Real Time Factor (RTF) als $RTF = simtime/realtime$ genutzt. Auf einem Rechner mit

dem Intel Prozessor i7-7700k, der AMD Grafikkarte R9 380 und 16 GB Arbeitsspeicher hatte die Simulation einen RTF von 128.78.

4 Selbstlernender Regler

Ziel dieser Arbeit ist es zu zeigen, dass die Simulation dazu geeignet ist, als Testumgebung zur Entwicklung und Umsetzung eines selbstlernenden Reglers zu fungieren. Dafür wird in diesem Kapitel ein selbstlernender Regler vorgestellt, der in der Simulation entwickelt wurde. Es wird zunächst auf die Anforderungen eingegangen, um dann eine Architektur vorzustellen, die den Anforderungen genügt. Danach wird erklärt, wie der Regler in der oben erklärten Simulation integriert und trainiert wurde.

Die Aufgabe des Reglers soll es sein, die nHAWigatora zu einem angegebenen Punkt zu manövrieren. Dies wird es dem Schiff erlauben, Trajektorien abzufahren, da diese sich in Wegpunkte aufteilen lassen. Als Informationen werden dem Regler die aktuelle Position, Geschwindigkeit und Ausrichtung, sowie die Position des Ziels zur Verfügung gestellt. Die Herausforderung für den Regler ist es, die Aktorik der nHAWigatora so anzusteuern, dass das Ziel erreicht wird.

4.1 Anforderungen

Aus dem Einsatzgebiet auf dem Miniaturschiff nHAWigatora und der Domäne der Wasserfahrzeuge ergeben sich folgende Anforderungen an den selbstlernenden Regler:

Bei der Position des Schiffs und des Ziels handelt es sich um kontinuierliche Werte. Das gewählte Verfahren für den selbstlernenden Regler muss in der Lage sein, in einem kontinuierlichen Zustandsraum operieren zu können.

Der Regler wird zunächst zwar nur in der Simulation genutzt, soll aber auch in der Realität anwendbar sein. Für die Berechnungen auf dem Schiff stehen jedoch nur zwei Raspberry Pi Computer zur Verfügung. Deswegen ist es nötig, ein Verfahren zu benutzen, das auch mit wenigen Ressourcen operabel ist. Das bezieht sich sowohl auf den Lern-, als auch den Aktionswahlprozess.

4.2 Architektur

Um den oben genannten Anforderungen zu genügen, wird der Deep Q-Networks (DQN) Ansatz verwendet. Durch die Nutzung von neuronalen Netzen als Funktionsapproximator eignet sich dieser Ansatz für kontinuierliche Zustandsräume.

Der Zustandsraum hat 6 Freiheitsgrade. Zuerst die Position (x, y) des Schiffs, abzüglich der Position des Ziels. Dadurch lernt der Regler die Navigation zum Ziel relativ zu der eigenen Position und erlangt dadurch ein allgemeines Wissen über die Lösung der Aufgabe. Zusätzlich zu der Position wird dem Regler außerdem die Ausrichtung Ψ übergeben. Die letzten drei Freiheitsgrade sind die Geschwindigkeiten u, v, r in diese Richtungen. Dies ist notwendig, um die Markov-Eigenschaft zu erfüllen.

Der Aktionsraum wird nicht kontinuierlich, sondern diskret dargestellt. Es können vier verschiedene Aktionen gewählt werden: Gradaus, Rückwärts, Links und Rechts.

4.3 Umsetzung

Für die Umsetzung wurde das *gym-gazebo*-Plugin genutzt, welches eine Integration des Open AI Gyms in Gazebo erlaubt. Die Umgebung und der Agent des lernenden Systems werden getrennt voneinander, gegen eine Schnittstelle, definiert. Dadurch lassen sich verschiedene Agenten- und Umgebungskombinationen ausprobieren.

4.3.1 Umgebung

Die Umgebung stellt den Zustandsvektor, Aktionsvektor und das Belohnungssignal als Schnittstelle zur Verfügung. Intern wird die Reaktion auf die Aktionen des Agenten bestimmt, die dem Agenten über das Belohnungssignal und den Zustandsvektor mitgeteilt wird. Das Belohungssignal bestimmt außerdem die Aufgabe des Agenten.

Der Zustandsvektor ist 6-stellig und hat einen Wertebereich von $[-\infty, \infty]$, der Akionsvektor besteht aus 4 diskreten Werte und das Belohungssignal hat einen Wertebereich von $[-\infty, \infty]$.

Die Berechnung des Zustandsvektors erfolgt, wie oben erwähnt, aus der Odometrie und der Position des Ziels. Diese Werte werden mithilfe der ROS-API ermittelt. Die zwei folgenden Abschnitte erläutern, wie das Ziel und die Odometrie in der Simulation bestimmt werden.

Ground Truth

Das Softwaresystem der nHAWigatora ist noch nicht in der Lage, die Position, Geschwindigkeit und Ausrichtung über die verbaute Sensorik zu erkennen. Diese Informationen bilden die Odometrie, die zusammen mit der Position des Ziels den Zustand der Umgebung definieren. Dies ist der Grund, warum eine Simulation entwickelt wurde, der diese Informationen entnommen werden können. Dafür wird das *libgazebo_ros_p3d*-Plugin verwendet. Das Plugin stellt eine optimale Odometrie über das */odom*-Topic mit einer *Odometry*-Nachricht zur Verfügung.

Ziel

Der Zielpunkt, zu dem der selbstlernende Regler manövrieren soll, wird über den ROS-Service *goal_node* zur Verfügung gestellt. Der Service nimmt eine *Pose*-Nachricht entgegen und berechnet daraus den Abstand zum Ziel, den Zustandsvektor und ob das Ziel erreicht wurde. Das Ziel gilt als erreicht, wenn die Entfernung weniger als 50 cm beträgt.

Der Zielpunkt selbst wird von dem *goal_node* zufällig erzeugt. Der Radius des Kreises, in dem ein neues Ziel zufällig generiert wird, der *seed* des benutzten Zufallsgenerators und die Höhe des Wasser können bei der Erzeugung des Services gesetzt werden.

Zur Visualisierung des aktuellen Ziels wird außerdem eine *Marker*-Nachricht über das */goal_marker*-Topic veröffentlicht. Damit lässt sich der Fortschritt des lernenden Reglers in dem Visualisierungsprogramm RViz als gelbe Kugel beobachten, wie in Abbildung 4.1 zu sehen.

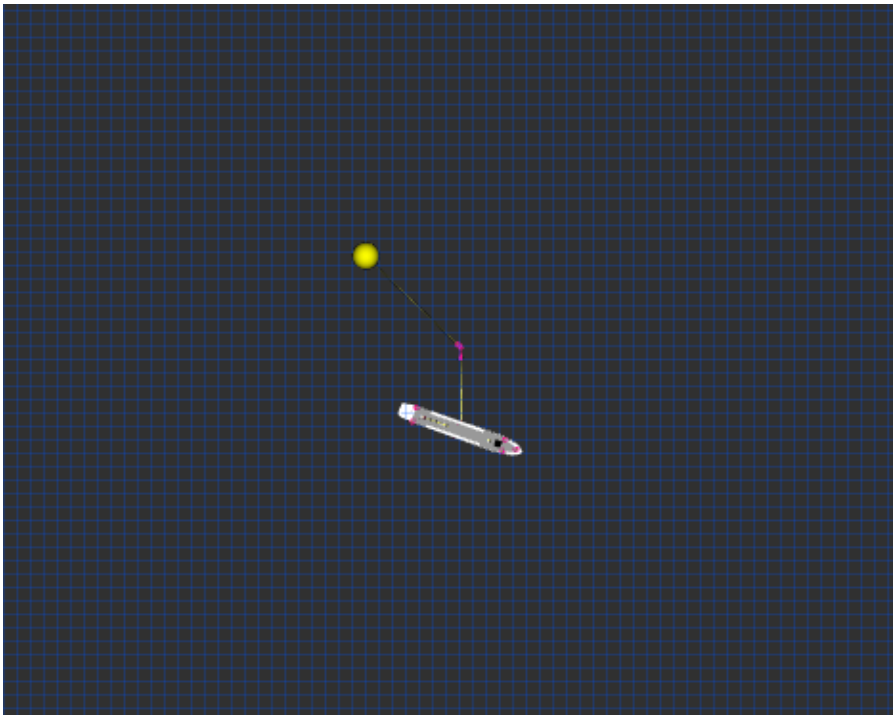


Abbildung 4.1: Visualisierung des Schiffs und des Ziels in RViz

Zustand

Als Zustand wird dem Agenten die relative Position zum Ziel, der Winkel Ψ und die Geschwindigkeiten u, v und r übergeben. Dies lässt sich ausdrücken als

$$Zustand = [x_{ship} - x_{goal}, u, y_{ship} - y_{goal}, v, \Psi, r]. \tag{4.1}$$

Die Berechnung des Zustands erfolgt durch Informationen, die über das Topic */odom* von der Simulation und über den Service */get_state* zur Verfügung gestellt werden.

Aktion

Wie oben erwähnt, werden die Aktionen als diskrete Richtungen vorwärts, rückwärts, links und rechts zur Verfügung gestellt. Dafür werden Steuersignale als Tupel in der Form $(propeller, thruster)$ an das Topic */motor_controller* geschickt. Für die vier Richtungen

werden folgende Toupel verwendet

$$Forward = (1, 0), Backward = (-1, 0), Left = (0, 1), Right = (0, -1) \tag{4.2}$$

Belohnungssignal

Das Belohnungssignal bestimmt die Aufgabe, die der Agent zu bewältigen hat. Die interne Berechnung bleibt ihm verborgen, aber er versucht die Belohnung pro Episode zu maximieren. Der *goal_ node*, der im vorherigen Abschnitt vorgestellt wurde, liefert unter anderem auch den Abstand zum Ziel und ob das Ziel erreicht wurde. Die Belohnung kann dargestellt werden als

$$Belohnung = -1 * \sqrt{x_{rel}^2 + y_{rel}^2} \tag{4.3}$$

wobei

$x_{rel} = x_{ship} - x_{goal}$

$y_{rel} = y_{ship} - y_{goal}$

Erreicht der Agent das Ziel, wird er mit einem konstanten Wert von 50 belohnt. Einen weiteren Sonderfall stellt die Abbruchbedingung dar. Ist der selbstlernende Regler nach 60 s realer Berechnungsdauer nicht in der Lage, das Ziel zu erreichen oder entfernt sich mehr als 5 m vom Ziel, wird die aktuelle Episode abgebrochen und ein neues Ziel gesetzt. Der Agent wird bei einem Abbruch der Episode mit einer Belohnung von -200 bestraft.

4.3.2 Agent

Der Agent ist die lernende Komponente des Systems, in dem der Deep Q-Networks (DQN)-Ansatz umgesetzt wird. Für das Approximieren der Q-Werte wird ein neuronales Netz mit 6 Neuronen im Inputlayer und 4 Neuronen im Outputlayer gewählt. Es werden zwei Hidden Layer mit je 300 Neuronen verwendet, die untereinander komplett vernetzt sind. Der Erfahrungspeicher kann 1.000.000 Werte halten, aus denen jeweils 64 zu einem Mini-Batch zusammengefasst werden. Die Lernrate des Q-Learnings ist 0.000125 und der Discount-Faktor 0.99. Das Lernen beginnt, sobald es 64 Erfahrungen im Erfahrungsspeicher gibt, um ein Mini-Batch zu bilden.

4.4 Training

Der selbstlernende Regler wird mit 1500 Episoden und 1000 Lernschritten pro Episode trainiert. Das Training erfolgt zunächst auf einem Computer mit deutlich höheren Ressourcen als dem auf dem Schiff verbauten Raspberry Pi 3B+. Stattdessen wird ein Computer mit einem Intel Core i7-7700k, einer AMD R9 380 und 16GB Arbeitsspeicher zum Trainieren genutzt.

5 Evaluationskonzept

In diesem Kapitel wird erklärt, wie die Simulation hinsichtlich der Nutzbarkeit zur Umsetzung und Entwicklung eines selbstlernenden Reglers bewertet wird. Das Ziel ist es zu zeigen, dass die Simulation die Entwicklung und Umsetzung des selbstlernenden Reglers beschleunigt. Der direkte Vergleich des Reglers zwischen Realität und Simulation ist mangels Odometrie in dem Testbecken des Wunderland nicht möglich, da das Schiff nicht in der Lage ist, diese zu berechnen und die Trackinganlage im Miniaturwunderland defekt ist. Deswegen werden zunächst die in der Simulation repräsentierten Objekte mit denen in der Realität verglichen. Dann wird geguckt, ob das Training auch auf den geringen Ressourcen des Raspberry Pis möglich ist. Danach wird die Leistung des selbstlernenden Reglers in verschiedenen Testumgebungen und unterschiedlichen Hyperparametern bewertet. Ist die Simulation bis zu einem gewissen Grad realitätsgetreu und lässt sich das Training auf der Schiffshardware durchführen, dann kann angenommen werden, dass eine ähnliche Leistung auch in der Realität von dem selbstlernenden Regler erreicht werden kann.

5.1 Simulation

In diesem Abschnitt wird darauf eingegangen, wie die simulierten Komponenten der nHAWigatora mit der Realität verglichen wurden. Dabei liegt der Fokus auf den physikalischen Eigenschaften, die für eine korrekte Bewegung im Wasser sorgen, also konkret die Festkörpereigenschaften wie Gewicht und Trägheitstensor, die Konfiguration des Plugins zu Berechnung des Wassers und die Dimensionierung der Kraft, die durch die Aktorik ausgeübt wird. Die Sensorik wird hingegen nicht ausführlich betrachtet, da sie in der aktuellen Konfiguration noch keine Auswirkungen auf den selbstlernenden Regler haben. Eine Ausnahme bildet dabei die IMU, die auch für die Evaluation genutzt wird und deswegen möglichst genau abgebildet werden muss.

5.1.1 Aktorik

Um die simulierte Aktorik mit der Realität zu vergleichen, wird die ausgegebene Kraft des Bugstrahlruders und des Propellers gemessen. Dabei werden für jeden Motor die Werte des Wertebereichs $[-1..1]$ in Schritten von 0.1 abgefragt. In der Simulation kann die Ausgabe der Motoren direkt abgefragt werden, in der Realität wird die ausgegebene Kraft mithilfe eines Kraftmessers gemessen.

Die Tests wurden im Miniaturwunderland Hamburg durchgeführt. Um die Kraft zu messen, wird eine Seite eines Kraftmessers am Rand des Beckens montiert. Die andere Seite wird zunächst am Bug und danach am Heck montiert. Die Tests werden anschließend wie oben beschrieben durchgeführt.

5.1.2 Festkörpereigenschaften

Die Genauigkeit der Festkörpereigenschaften geschieht durch den Vergleich der simulierten und der realen IMU. Dafür werden verschiedene Testszenarien im Testbecken des Miniaturwunderlands durchgeführt:

- Drehimpulse in und entgegen der Yaw-Richtung durch das Bugstrahlruder
- Schubimpulse durch den Propeller in und entgegen der Surge-Richtung

Vergleichbare Tests werden in der Simulation durchgeführt. Dann werden die Werte der beiden IMUs verglichen. Dabei wird die Messung betrachtet auf die der Kraftimpuls die größte Einwirkung hat. Bei den Drehimpulsen ist dies die Winkelgeschwindigkeit r um die z-Achse des Gyroskops und bei den Schubimpulsen ist es die Beschleunigung u in Richtung der x-Achse des Beschleuningssensors.

5.1.3 Sensorik

Wie oben schon erwähnt, wurde die Sensorik nur bedingt genau evaluiert, da sie auf den Entwicklungsprozess des selbstlernenden Regler keinen Einfluss hat. Die IMU bildet eine Ausnahme, da sie benötigt wird, um das Verhalten des Schiffs im Wasser zu evaluieren.

IMU

Die IMU wird innerhalb der Simulation durch die Drift und dem gaußschen Rauschen auf allen Achsen charakterisiert. Deshalb müssen diese Werte, der verbauten IMU bestimmt werden. Dazu wird das Schiff in eine Ruhelage gebracht und eine Serie von Daten aufgezeichnet. Dann werden Beschleunigungssensor und Gyroskop hinsichtlich des gaußschen Rauschens evaluiert. Die Unterschiede der Standardabweichung und Varianz zwischen Realität und Simulation werden für alle Achsen verglichen. Dann wird die Drift der Realität mit der in der Simulation verglichen.

Lidar

Für das Lidar wurden lediglich die Werte für das gaußsche Rauschen evaluiert. Dafür wird eine Serie von Daten des Lidars in Ruhelage aufgezeichnet. Dann wird das gaußsche Rauschen der aufgezeichneten Daten mit denen in der Realität verglichen.

Distanzsensoren

Die Distanzsensoren werden genau wie das Lidar mit der Realität verglichen. Es werden Daten aufgenommen und dann das gaußsche Rauschen der Simulation mit dem der Realität verglichen. Das Schiff befindet sich bei der Aufnahme der Daten in Ruhelage und ein Objekt befindet sich in konstanter Entfernung vor dem zu evaluierenden Distanzsensor.

5.2 Selbstlernender Regler

In diesem Abschnitt wird das Evaluationskonzept des selbstlernenden Reglers vorgestellt. Ziel der Evaluation ist es festzustellen, ob der selbstlernende Regler durch die Simulation zu einem Lernergebnis kommt und wie erfolgreich das gelernte Verhalten das Schiff manövrieren kann.

Dafür wird zunächst die Belohnung der Episoden in einem Lernprozess betrachtet. Anhand der Belohnungen kann dann festgestellt werden, ob das lernende System in der Lage ist, die optimale Belohnung zu erreichen.

5.2.1 Rastertest

Danach wird die Leistung des Reglers beim Anfahren verschiedener Ziele getestet. Dem Schiff werden dafür eine Reihe von Zielen gegeben und anhand der benötigten Zeit wird bewertet, wie erfolgreich das Manöver zum Erreichen des Ziels war. Außerdem wird betrachtet, ob das selbstlernende System in der Lage ist, die Ziele zu erreichen. Zwischen dem Anfahren der Ziele wird das Boot wieder auf seine Ursprungsposition zurückgesetzt. Erreicht der selbstlernende Regler mehr als die Hälfte der Ziele, wird er als erfolgreich eingestuft. Dieser Test wird im weiteren Verlauf dieser Arbeit als Rastertest bezeichnet. Die Anordnung der Ziele für diesen Testfall können der Abbildung 5.1 entnommen werden.

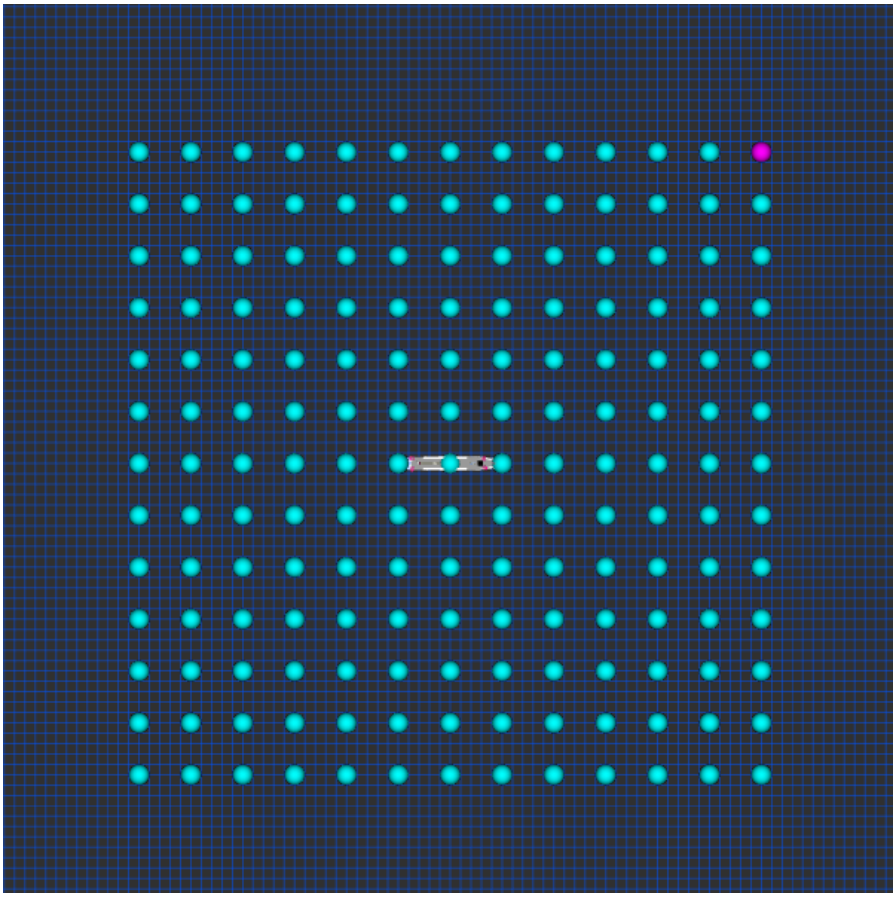


Abbildung 5.1: Visualisierung des Rastertests

5.2.2 Pfadtest

Dann wird das eigentliche Anwendungsgebiet des selbstlernenden Reglers betrachtet. Die Aufgabe des Softwaresystems im Bordcomputers wird die Anfahrt von Zielen sein, die Bestandteil einer Trajektorie sind. Dafür wird ein festgelegter Rundkurs abgefahren. Die Leistung des selbstlernenden Reglers wird an der benötigten Zeit für eine Runde gemessen. Der zu fahrende Rundkurs wird in Abbildung 5.2 gezeigt. Dieser Test wird im weiteren Verlauf dieser Arbeit als Pfadtest bezeichnet.

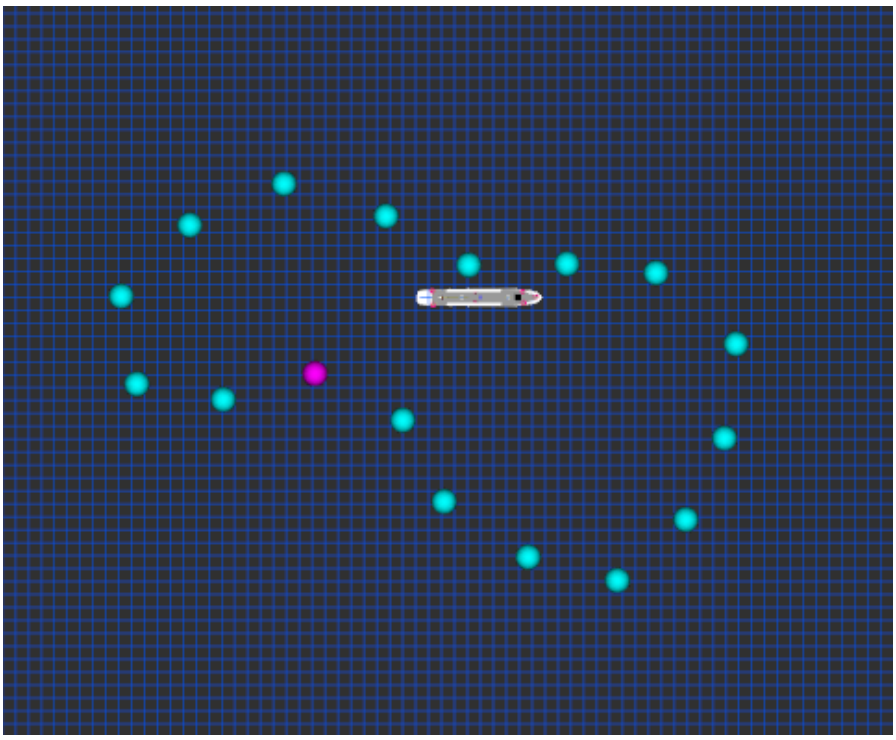


Abbildung 5.2: Visualisierung des Pfadtests

5.2.3 Testvariationen

Außerdem wird untersucht ob der selbstlernende Regler auch verschiedene Varianten der nHAWigatora manövrieren kann. Dazu werden zwei verschiedene Testszenarien durchgeführt:

Aktormisskonfiguration Die ausgebende Kraft der Aktorik wird stark verändert. Anstatt in alle Richtungen die gleiche Kraft auszugeben sind Propeller und Bugstrahlru-

der nun in in der einen Richtung nur halb so stark wie in der Anderen. D.h. die Richtung "vorwärts"bringt doppelt so viel Schub wie "rückwärtsünd "links"doppelt so viel wie "rechts". Damit wird evaluiert ob, der Regler auch dann noch ein erfolgreiches Lernverhalten erreicht, wenn die Aktorik sich verändert, wie es bei der nHAWigatora durch Umbauten häufig der Fall ist.

Verändertes Gleitverhalten Die verwendeten Parameter des *usv-gazebo-plugins* werden beliebig verändert. Dadurch ändert sich die Reaktion des Schiffs auf die Kräfte der Aktorik.

Der Erfolg der Testvariationen wird dann, genau wie zuvor, durch den Raster- und Pfadtest bestimmt.

5.3 Ressourcenaufwand

Der in diesem Abschnitt vorgestellte Test hat das Ziel, festzustellen, ob das Training des vorgestellten Algorithmus' auch auf der vergleichsweisen leistungsschwächeren Hardware des Schiffs funktioniert.

Die Durchführung eines realen Tests in dem Wasserbecken des Miniaturwunderlands ist mangels Odometrie nicht möglich. Stattdessen wird die Simulation genutzt, um einen Hardware-in-the-Loop (HIL) Test durchzuführen. Die ROS-Nodes, die für das Lernen verantwortlich sind, werden dabei auf den Computern des Schiffs ausgeführt. Die Berechnungen der Simulation geschehen jedoch auf einem Desktop-Computer.

Dadurch wird der Aufwand des Lernens auf dem Bordcomputer messbar. Betrachtet wird vor allem die Auslastung der CPU.

6 Evaluationsergebnisse

In diesem Kapitel werden die Evaluationsergebnisse des oben erläuterten Evaluationskonzepts vorgestellt. Eine Einordnung der Ergebnisse erfolgt in Kapitel 7. Die Evaluationsergebnisse werden in der gleichen Reihenfolge repräsentiert wie die Evaluationskonzepte, zuerst wird auf die Simulation eingegangen, dann auf den selbstlernenden Regler und zuletzt auf den Resourcenaufwand.

6.1 Simulation

6.1.1 Aktorik

Die Kraft der Aktorik konnte nur für die Extremwerte bestimmt werden. Der verwendete Kraftmesser verfügt mit einer Auflösung von 10 g nicht über die notwendige Genauigkeit, um die Werte zu erfassen, wenn die Aktorik schrittweise angesprochen wird. Stattdessen wurde die maximale Kraft der beiden Aktoren in die beiden möglichen Richtungen bestimmt. Die dabei gemessenen Werte können der Tabelle 6.1 entnommen werden.

Aktor	Kraft	Realität	Simulation
Propeller	100%	10 g	10,3934 g
Propeller	−100%	10 g	10,3254 g
Bugstrahruder	100%	60 g	62,6431 g
Bugstrahruder	−100%	50 g	51,9456 g

Tabelle 6.1: Extremwerte der Aktorik

6.1.2 Festkörpereigenschaften

In diesem Abschnitt werden die Evaluationsergebnisse der Festkörpereigenschaften vorgestellt. Der Messaufbau aus dem Evaluationskonzept konnte nicht durchgeführt werden

und wurde deshalb angepasst. Zu dem Zeitpunkt der Durchführung verfügt die nHAWigatora nämlich nicht über eine funktionale Aktorik. Stattdessen wurde der Antrieb von Propeller und Bugstrahlruder mit einer manuellen Bewegung des Schiffs simuliert. Dann wurde ein ähnlicher Impuls in der Simulation durchgeführt.

Das erste Testszenario ist ein Kraftimpuls auf das Schiff in Surge-Richtung. Abbildung 6.1 zeigt die x-Achse des Beschleunigungssensors der IMU. Links sind die Werte der realen IMU zu erkennen und rechts die Werte der simulierten IMU. Die Y-Achse zeigt die gemessene Beschleunigung in m/s^2 und die X-Achse die Zeit in Samples. Die Daten wurden mit einer Frequenz von 100 Hz aufgenommen.

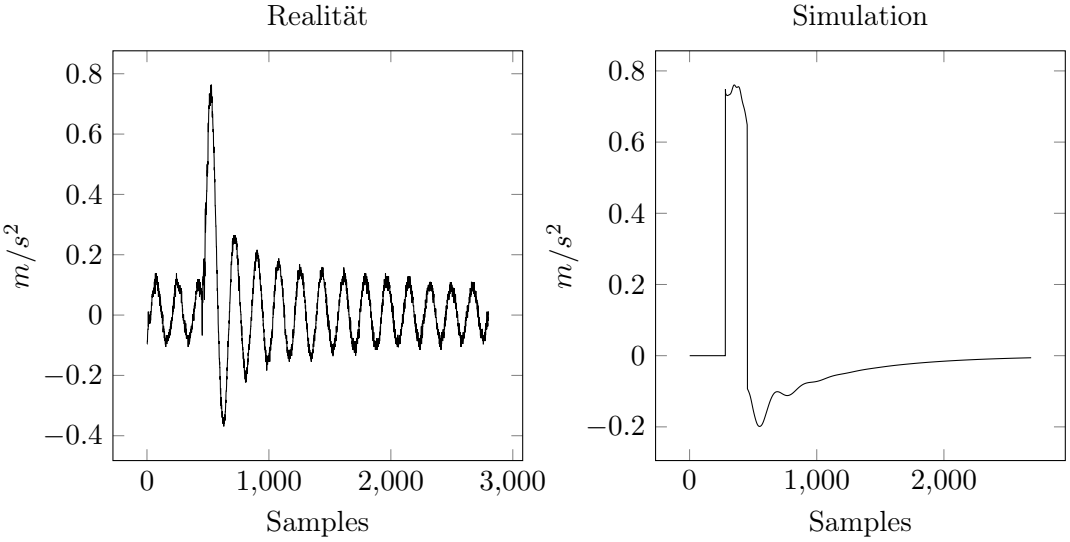


Abbildung 6.1: Impuls in die Surge Richtung

Der Kraftimpuls in die Surge-Richtung ist deutlich an der Abbildung zu erkennen. Die Werte der Realität zeigen einen deutlichen Anstieg der Beschleunigung bei ca. Sample 450, die der Simulation bei ca. Sample 280. Der Anstieg der Beschleunigung zeigt eine Bewegung nach vorne. Die Spannweite der betrachteten Daten beträgt für die Realität ungefähr 1,15 m/s^2 und für die Simulation ungefähr 0,95 m/s^2.

Ein bedeutender Unterschied ist in dem Verlauf der Werte zu erkennen. Der Verlauf der realen Werte zeigt ein oszillierendes Verhalten. Die Werte schwingen schon, bevor ein Impuls auf das Schiff ausgeübt wird. Ab dem Impuls nimmt die Schwingung stark zu und danach wieder ab, bis sie wieder ihren vorherigen Wert erreicht. Die Frequenz der Schwingung ist ungefährt 5 Hz. In den Werten der Simulation ist vor und nach dem

Impuls keine Schwingung zu erkennen. Der Impuls sorgt für einen deutlichen Anstieg der gemessenen Beschleunigung und nach dem Impuls fällt die Beschleuningung in den negativen Bereich, das Schiff bremst also ab. Die Kurve nähert sich dann der 0 an, bis diese nach ca. 3000 Samples erreicht wird.

In Abbildung 6.2 ist die Messung der IMU aus dem zweiten Testszenario zu erkennen, dem Kraftimpuls entgegen der Surge-Richtung.

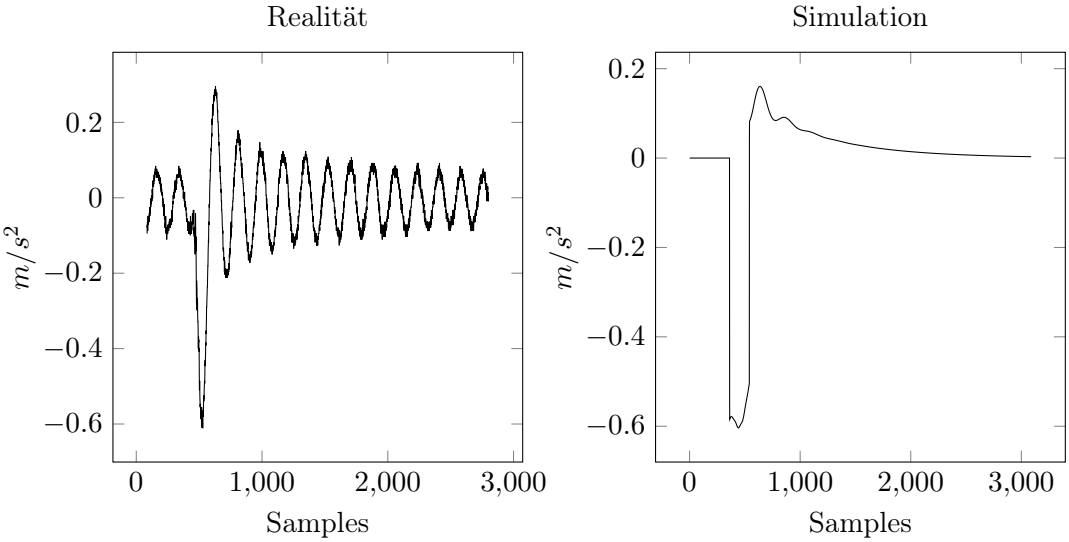


Abbildung 6.2: Impuls entgegen der Surge Richtung

In der Abbildung ist der Kraftimpuls zu erkennen. In der Simulation wirkt eine stark negative Beschleunigung ab ca Sample 360, in der Realität kann die negative Beschleunigung ab ca Sample 400 erkannt werden. Die Spannweite der Daten beträgt in der Realität ungefähr $0{,}9\,\mathrm{m/s^2}$ und in der Simulation $0{,}75\,\mathrm{m/s^2}$.

In dieser Abbildung ist wieder ein oszillierendes Verhalten der Realität erkennbar. In der Simulation kann, wie schon bei dem Kraftimpuls in die Surge-Richtung, dieses Verhalten nicht festgestellt werden.

Abbildung 6.3 zeigt das dritte Testszenario: Ein Drehimpuls in Yaw-Richtung. Abgebildet ist die z-Achse des Gyroskops. Wie in den vorherigen Abbildungen sind links die Werte der realen und rechts die Werte der simulierten IMU zu erkennen. Auf der Y-Achse ist die gemessene Winkelgeschwindigkeit r in °/s und auf der X-Achse die Zeit in Samples zu erkennen. Auch hier wurden die Daten mit 100 Hz aufgezeichnet.

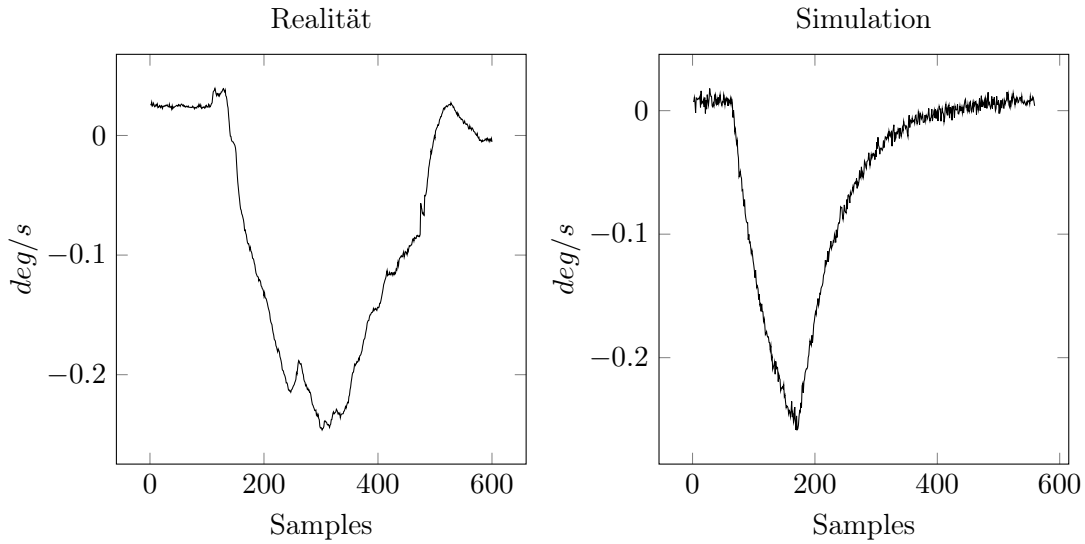


Abbildung 6.3: Impuls in Yaw Richtung

In der Abbildung ist die Drehung in die Yaw-Richtung deutlich zu erkennen. Die Werte der Realität zeigen den Beginn des Drehimpulses ab ca. Sample 150, in den Werten der Simulation ist der Beginn ab ca. Sample 100 zu sehen. Die Spannweite beträgt in der Simulation ungefähr 0,26 °/s und in der Realität 0,28 °/s.

Der Verlauf der Werte unterscheidet sich zwischen den beiden Datensätzen deutlich. Die in der Realität aufgenommenen Werte zeigen, dass die Phasen, in denen die Drehgeschwindigkeit zunimmt und danach abnimmt, ungefähr gleich lang sind. In der Simulation macht die Phase der zunehmenden Geschwindigkeit dagegen nur 1/3 aus und die Phase der abnehmenden Geschwindigkeit 2/3.

Abschließend zeigt Abbildung 6.4 das letzte Testszenario, die Drehung entgegen der Yaw-Richtung. Die Abbildung der Ergebnisse erfolgt genau wie in der vorherigen Abbildung.

Die Drehung entgegen der Yaw-Richtung lässt sich in der Abbildung erkennen. Ab ca. 320 Samples beginnt die Drehung in der Realität, in der Simulation beginnt sie ab ca. 350 Samples. Die Spannweite der Daten beträgt in der Simulation 0,15 °/s und in der Realität 0,17 °/s.

Der Verlauf der beiden Kurven ähnelt sich sehr. Die Phasen der zunehmenden Geschwindigkeiten sind ungefähr gleich lang. Die Phase der abnehmenden Geschwindigkeit ist bei

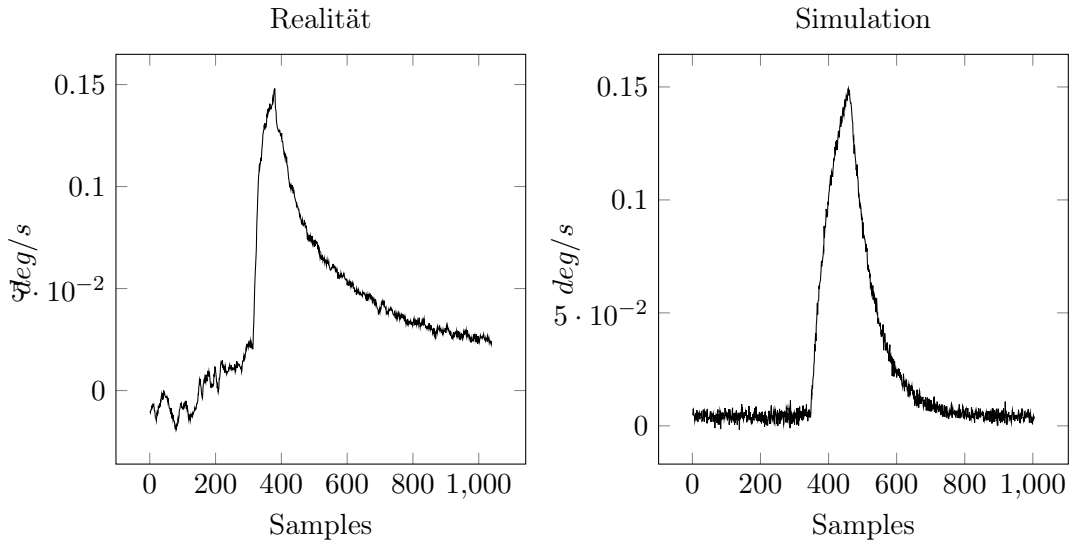


Abbildung 6.4: Impuls entgegen entgegen der Yaw Richtung

den Daten der Realität deutlich länger zu beobachten. Bei Sample 1000 ist die Abbremsung der Rotation in der Realität noch nicht abgeschlossen, während dies bei den Daten der Simulation schon bei Sample 800 zu beobachten ist.

6.1.3 Sensorik

IMU

Für die Evaluation des Rauschens für die IMU wurde eine Serie von Messdaten über mehrere Minuten sowohl in der Simulation als auch in der Realität aufgenommen. Zuerst wurde der Beschleunigungssensor untersucht. Das Schiff befand sich dabei in einer ruhigen Position. Mithilfe der *numpy*-Python-Bibliothek wurde dann das normalverteilte Rauschen auf allen Achsen ermittelt. Exemplarisch für den Beschleuningssensor ist in Abbildung 6.5 das Ergebnis für die Z-Achse abgebildet. Die gemessenen Werte werden für die Visualisierung in ein Histogramm mit 10 Balken übertragen. Die Höhe der Balken zeigt, wie viele Werte innerhalb ihres Bereichs auf der x-Achse liegen. Die über dem Balkendiagramm liegende Kurve zeigt die Kurve der auf die Daten approximierten Normalverteilung. Um zu veranschaulichen, dass die im Balkendiagramm gezeigten Werte genutzt wurden, um die Koeffizienten der Normalverteilung zu bestimmen, wurde die Kurve um den Faktor 18 skaliert.

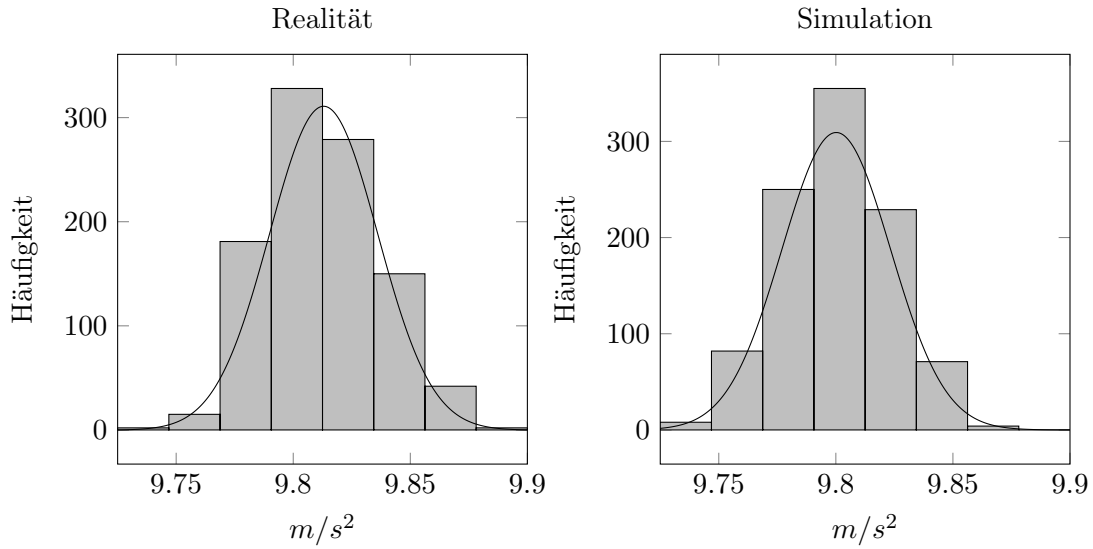


Abbildung 6.5: Rauschen in der Z-Achse des Beschleuningssensors

Die Werte der berechneten Koeffizienten für die Normalverteilung sind in der Tabelle 6.2 zu finden.

Achse	μ	σ
	m/s^2	m/s^2
X	0.029867	0.011325
Y	-0.430443	0.011311
Z	9.812985	0.023084

Achse	μ	σ
	m/s^2	m/s^2
X	0.017224	0.011416
Y	0.000111	0.011263
Z	9.800431	0.023125

Tabelle 6.2: Koeffizienten des realen (links) und simulierten (rechts) Beschleuningssensors

Aus der Tabelle lässt sich entnehmen, dass die Werte für die Verteilung σ nahezu identisch sind. Die größte Abweichung hat dabei die Verteilung der X-Achse mit einer Abweichung von $91 \cdot 10^6\,m/s^2$, was vernachlässigbar gering ist. Die Abweichung des Mittelwerts μ ist jedoch wesentlich größer, mit einer maximalen Abweichung von $0{,}430\,334\,m/s^2$. Der Unterschied des Mittelwerts lässt sich auch in Abbildung 6.5 erkennen, die Normalverteilung der Simulation liegt deutlich weiter links als die Kurve in der rechten Abbildung.

Anschließend wurde das Gyroskop der IMU untersucht. Die analysierten Daten stammen aus der gleichen Aufzeichnung wie die zuvor betrachteten Daten des Beschleuningssensors. Auch hier wurden die Koeffizienten der Normalverteilung berechnet. Als Veranschaulichung dient in Abbildung 6.6 die Y-Achse des Gyroskops, hier wurde die Kurve der Normalverteilung nicht skaliert.

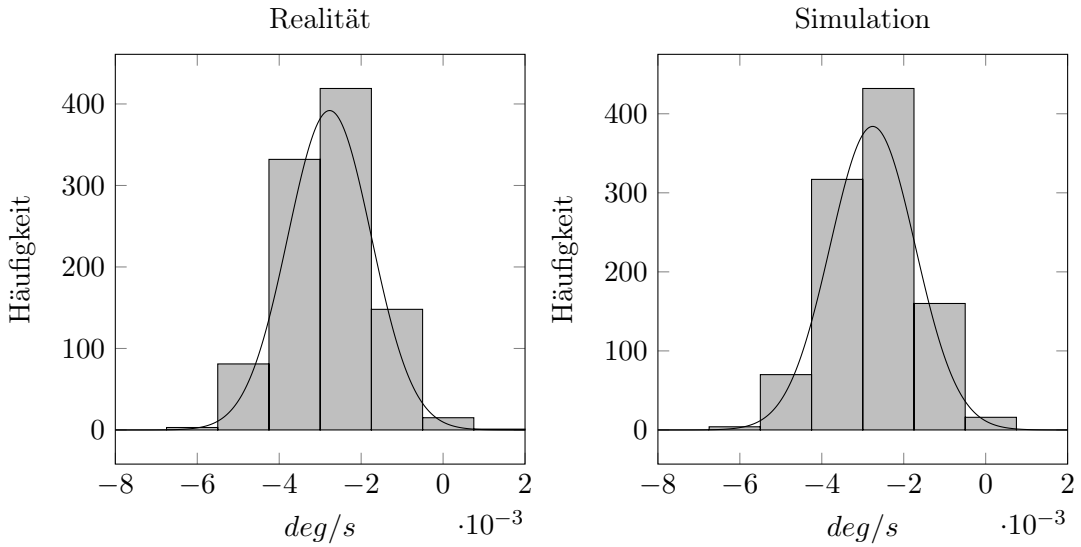


Abbildung 6.6: Rauschen auf der Y-Achse des Gyroskops

Die Werte der Koeffizienten für die Normalverteilung lassen sich der Tabelle 6.3 entnehmen.

Achse	μ	σ	Achse	μ	σ
	°/s	°/s		°/s	°/s
X	0.001776	0.000849	X	0.001772	0.000850
Y	-0.002770	0.001018	Y	-0.002758	0.001039
Z	0.001886	0.000871	Z	0.001892	0.000874

Tabelle 6.3: Koeffizienten des realen (links) und simulierten (rechts) Gyroskops

In Tabelle 6.3 lässt sich erkennen, dass sowohl der Mittelwert μ als auch die Verteilung σ nur sehr kleine Abweichungen zwischen Simulation und Realität haben. Die größte Abweichung des Mittelwerts und der Verteilung hat die Y-Achse mit $12 \cdot 10^6$ °/s und $21 \cdot 10^6$ °/s. Das Gyroskop ist zwischen Realität und Simulation kaum zu unterscheiden.

Zuletzt wird die konstante Drift in die Yaw-Richtung betrachtet. Die Daten stammen ebenfalls aus dem zuvor analysierten Datensatz. Um Realität und Simulation miteinander zu vergleichen, zeigt Abbildung 6.7 den Unterschied zwischen der simulierten Drift und der realen Drift nach 1000 Samples. Da die IMU mit einer Frequenz von 20 Hz Daten verbreitet, wird hier also die Drift nach 50 s abgebildet. Auf der X-Achse sind die Samples aufgetragen und auf der Y-Achse die Orientierung der IMU auf der Z-Achse. Zu Beginn, beim 1. Sample, weichen die beiden Graphen kaum voneinander ab, die Differenz zwischen

den beiden ist $17 \cdot 10^{6\circ}$. Zum Ende hin wächst die Differenz, bis sie beim letzten Sample $949 \cdot 10^{6\circ}$, was auch noch ziemlich genau ist.

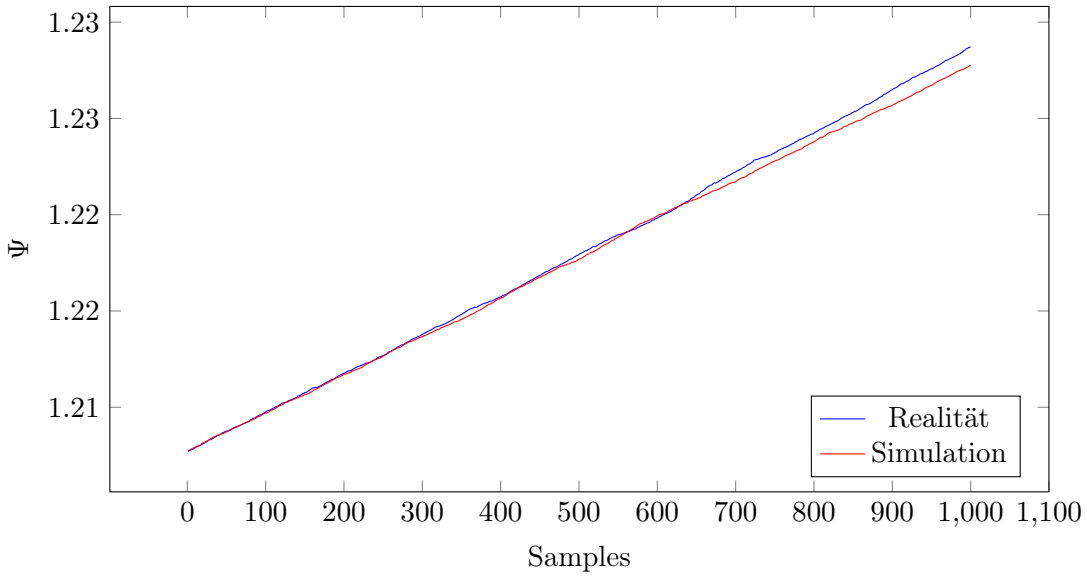


Abbildung 6.7: Drift um die Z-Achse der Orientierung

Um die Drift noch detailierter zu evaluieren, wird die Fehlerverteilung zwischen den einzelnen Samples betrachtet. Der Mittelwert und die Verteilung kann, genau wie oben bei Beschleuningssensor und Gyroskop, approximiert werden. In Abbildung 6.8 wurden die beiden Fehlerverteilungen visualisiert. Für eine bessere Visualisierung wurde die Kurve um den Faktor 0.015 auf der Y-Achse skaliert.

Die berechneten Koeffizienten der Normalverteilung unterscheiden sich kaum: In der Simulation hat der Driftfehler einen Mittelwert μ von $20 \cdot 10^{6\circ}$ und eine Verteilung σ von $9 \cdot 10^{6\circ}$, während der Driftfehler in der Simulation die Koeffizienten $\mu = 20 \cdot 10^{6\circ}$ und $\sigma = 9 \cdot 10^{6\circ}$ hat. Dies lässt sich auch in der Abbildung erkennen. Dort sind die Daten nahezu identisch, die Realität weist nur eine leicht nach rechts verschobene Datenmenge auf.

Lidar

Die Daten für die Evaluation des Lidars wurden aufgenommen, indem das Schiff in Ruhelage in einem Raum ohne sich bewegende Objekte stand. Dann wurde ein Datenpunkt ausgewählt und die gemessene Entfernung identisch in der Simulation nachgestellt. Mit

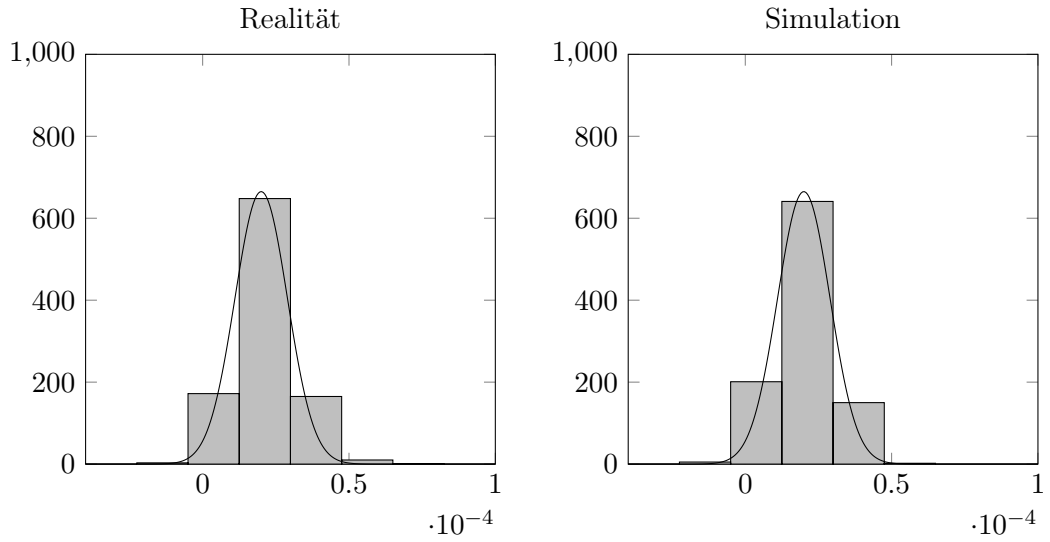


Abbildung 6.8: Rauschen des Driftfehlers zwischen Samples

beiden Testaufbauten wurden jeweils 3000 Samples Daten aufgenommen. Da das Lidar mit einer Frequenz von 10 Hz Daten aufnimmt, wurden also über 5 Minuten lang die Daten aufgezeichnet. Dann wurden auf diesen Daten, wie zuvor bei der Evaluation der IMU, die Koeffizienten einer Normalverteilung berechnet. Die Werte der Simulation und der Realität sind in Abbildung 6.9 zu sehen.

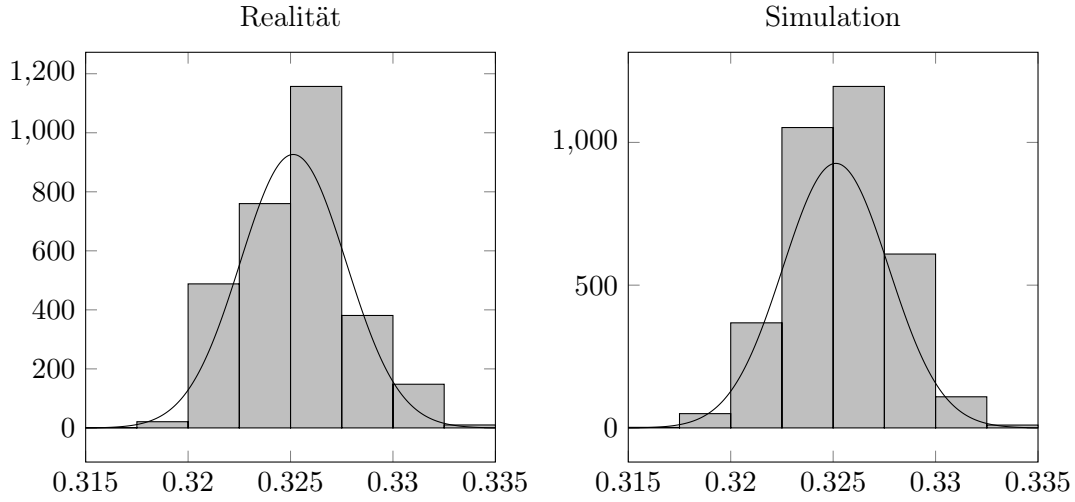


Abbildung 6.9: Rauschen der gemessenen Lidarwerte in Simulation und Realität

Für die Simulation wurden die Koeffizienten als $\mu = 0{,}325\,458\,\mathrm{m}, \sigma = 0{,}002\,538\,\mathrm{m}$ und in der Realität als $\mu = 0{,}325\,132\,\mathrm{m}, \sigma = 0{,}002\,583\,\mathrm{m}$ bestimmt. Der Unterschied der

Mittelwerte beträgt 0,326 mm. Die Abweichung ist zwar relativ groß, ist wahrscheinlich jedoch in dem Testaufbau begründet. Die Entfernung wurde nur mit Messgeräten festgelegt, die eine Genauigkeit von 1 mm haben. Der relevantere Wert ist dabei die Differenz zwischen der Verteilung, die 0,045 mm beträgt. Dieser Wert wird nur bedingt durch den Testaufbau beeinträchtigt und ist zudem sehr gering.

Distanzsensoren

Genau wie zuvor bei dem Lidar wurde für die Evaluation der Distanzsensoren eine Serie von Daten von dem Schiff in Ruhelage in einem Raum ohne bewegende Objekte aufgenommen. Als zu messendes Objekt wurde vor dem Sensor in Position *vorschiff_ backbord* (siehe Abbildung 3.6) eine Box mit einer Entfernung von 10 cm aufgestellt. Dann wurden die Daten, die über das Topic *rangesesor/vorschiff_ backbord* gesendet wurden, für einen Zeitraum von ungefähr 4 Minuten aufgezeichnet, in dem 2500 Samples der Werte erfasst wurden. Dann wurden wieder die Koeffizienten der Normalverteilung berechnet. Abbildung 6.10 zeigt eine Visualisierung dieser Ergebnisse. Zur besseren Veranschaulichung wurde die Y-Achse der Gaußkurve um den Faktor 0.3 skaliert.

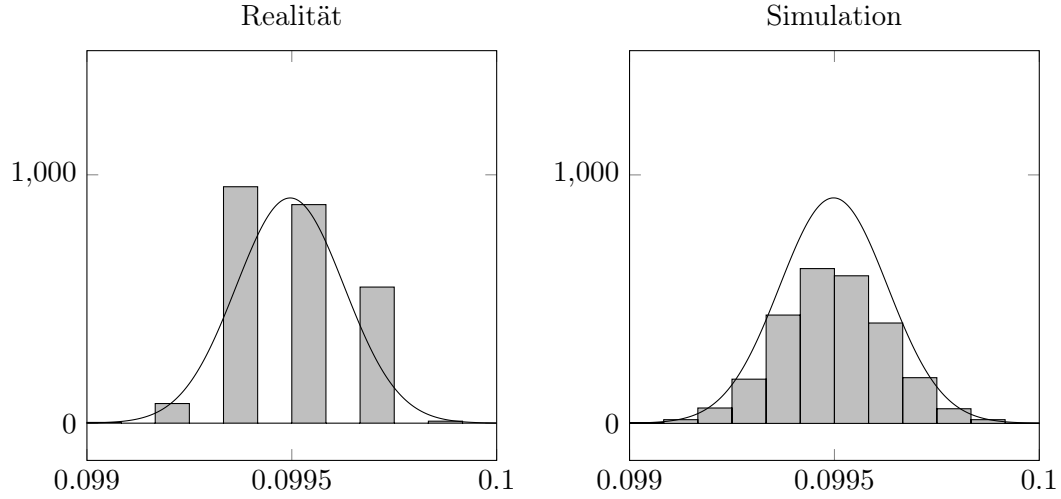


Abbildung 6.10: Rauschen der gemessenen Distanzwerten in Simulation und Realität

Die Koeffizienten wurden als $\mu = 99{,}498\,\text{mm}; \sigma = 0{,}132\,\text{mm}$ in der Simulation und $\mu = 99{,}496\,\text{mm}; \sigma = 0{,}132\,\text{mm}$ berechnet. Zwischen der Verteilung σ der Werte gibt es also keinen Unterschied, zwischen dem Mittelwert μ gibt es nur eine vernachlässigbar

kleine Differenz von 0,002 mm. In der Abbildung 6.10 ist jedoch ein deutlicher Unterschied zwischen den beiden Histogrammen erkennbar. In der rechten Abbildung, der Abbildung der Simulation, sind die gemessenen Werte gleichmäßig über alle Balken des Histograms verteilt. In der linken Abbildung, die des echten Sensors, werden jedoch nur 5 Balken abgebildet. Dies ist in der Genauigkeit des Sensors begründet. Die Messungen des echten Sensors enthalten nur 5 Werte, beschränkt durch die 16 Bit, mit denen sie an die verarbeitende Software weitergeleitet werden. Die Simulation führt die Berechnungen mit Fließkommazahlen hingegen mit 64 Bit durch. Das erzeugt die erwartete Verteilung der Balken in der rechten Abbildung.

6.2 Selbstlernender Regler

In diesem Abschnitt werden die Ergebnisse der Evaluation des selbstlernenden Reglers gezeigt. Zuerst wird der auf ein vorhandenes Lernverhalten geprüft, danach wird die gelernte Erfahrung bewertet.

In Abbildung 6.11 ist der Verlauf des Lernens nach 1500 Episoden zu erkennen. Auf der X-Achse sind die Episoden aufgetragen, auf der Y-Achse die erreichte Belohnung. Ein Lernverhalten ist deutlich zu erkennen. In den ersten 50 Episoden erreicht der Agent fast gar nicht das Ziel und damit eine Belohnung von 50 sondern die Belohnung schwankt zwischen -400 und -1100. Nach den ersten 50 Episoden ist ein Lernverhalten erkennbar; der Agent erreicht regelmäßig seine Ziele mit einer Bewertung von 50, bis nach 200 Episoden ein kurzer Abschnitt ohne Belohnungen unter -100 zu sehen ist. In dem weiteren Verlauf des Lernprozesses sind einige Stellen erkennbar, in denen wieder deutlich schlechtere Ergebnisse erzielt werden. Dies ist beispielsweise um die Episode 350 zu erkennen. Die dort erreichten Belohnungen sind schlechter als die Belohnungen zum Beginn des Lernens. Ein ähnliches Phänomen tritt 100 Episoden später ab Episode 450 auf. Auch hier treten mehrfach neue Minima für die Belohnungen auf. Danach wird das Verhalten des Agenten deutlich besser bewertet, bis ab Episode 800 erneut ein Abschnitt schlechter Bewertung zu sehen ist. Danach werden die gesammelten Belohnungen deutlich besser, bis am Ende Belohnungen um das optimale Ergebnis 50 zu erkennen sind.

Über den gesamten Verlauf sind deutliche Schwankungen in den Belohnungen zu erkennen. Zwar lässt sich ein Trend der Verbesserung sehen, dieser Trend wird aber oft durch große Ausreißer unterbrochen.

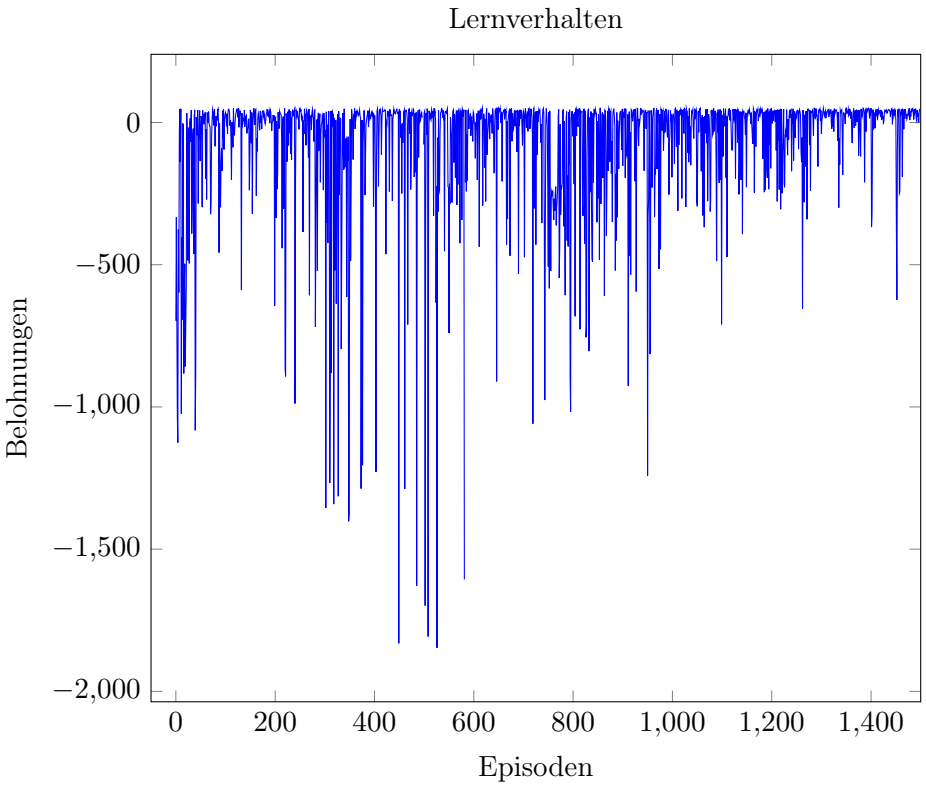


Abbildung 6.11: Belohnungen pro Episoden

Abbildung 6.12 zeigt das Testraster, nachdem der Agent Rastertest absolviert hat. Zur Durchführung dieses Tests wurde das trainierte neuronale Netz nach 1500 Episoden genommen, dessen Trainingserfolge auch in der oben gezeigten Abbildung 6.11 dargestellt werden. Die in grün markierten Ziele hat der Agent erreicht, die als rot markierten Ziele wurden nicht erreicht. Wie in der Abbildung zu erkennen, erreicht der Agent in diesem Test ein sehr gutes Ergebnis, indem er 166 von 169 möglichen Zielen erreicht.

In Tabelle 6.4 wird die Leistung von drei trainierten Netzen im Rastertest verglichen. Alle Netze wurden mit den gleichen Hyperparametern und Umgebungen trainiert. Alle 200 Episoden wurde das Netz zwischengespeichert, damit im Nachhinein die Leistung des selbstlernenden Regler zu dieser Episode bewertet werden konnte. In der Tabelle wird die Zahl der erreichten Ziele und eine Durchschnittszeit angegeben. Die Durchschnittszeit ist die Zeit, die durchschnittlich benötigt wird, um ein Ziel zu erreichen. Die Zeiten der Ziele, die nicht erreicht wurden, wird nicht mit einbezogen.

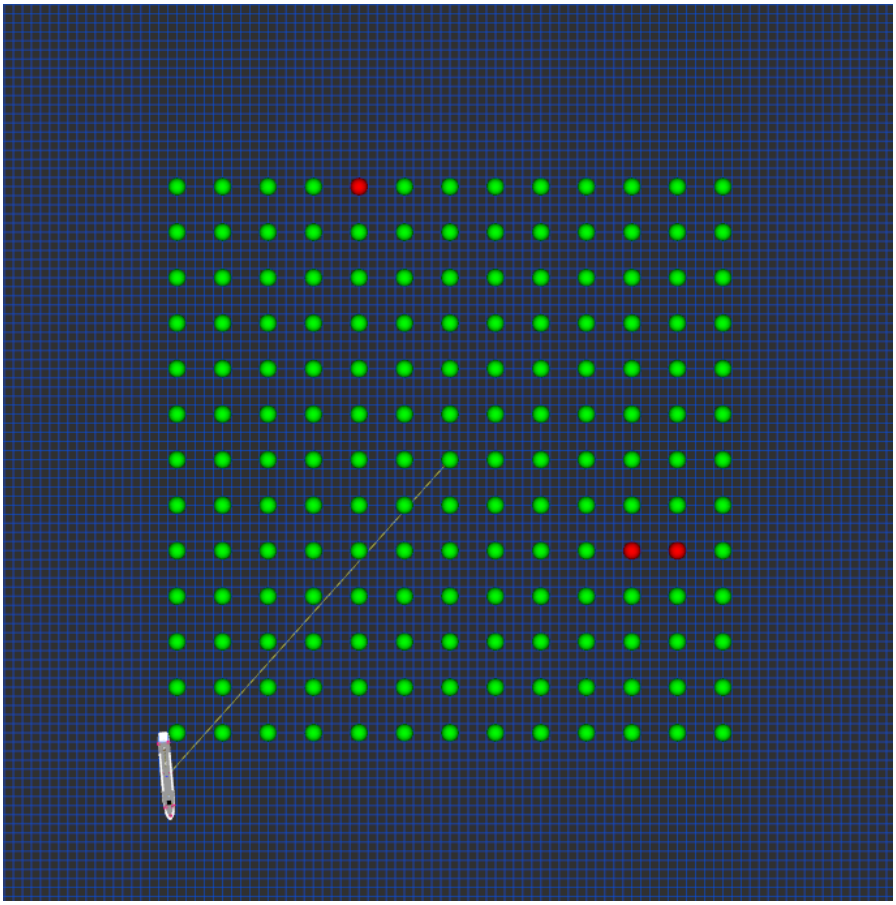


Abbildung 6.12: Ergebnisse des Rastertests

Für das Training benötigte der erste Durchlauf 7134 s, der zweite Durchlauf 14 103 s und der dritte Durchlauf 6573 s.

Trotz der gleichen Parameter erreichen die drei Netze nicht ähnliche Ergebnisse, stattdessen unterscheiden sich die Ergebnisse der Durchläufe stark. Wie in Kapitel 5 bestimmt, gilt der Rastertest als erfolgreich, wenn mehr als die Hälfte der Ziele erreicht wird. Das vorgestellte Testszenario enthält 169 Zielpunkte, erfolgreich ist ein Regler also bei mehr als 84 erreichten Zielen. Das schlechteste Ergebnis erzielt der erste Durchlauf nach 1400 Episoden mit nur 92 erreichten Zielen, alle Episoden aller Durchläufe erzielen also ein erfolgreiches Ergebnis. Aus dem zweiten Durchlauf erreicht das Netz in zwei Stadien, nach 1400 und 1500 Episoden, das perfekte Ergebnis, indem alle 169 Punkte erreicht werden. Die schlechteste Durchschnittszeit hat der zweite Durchlauf nach 600 Episoden mit 2,5931 s.

	Durchlauf 1		Durchlauf 2		Durchlauf 3	
Episode	Ziele	∅ s	Ziele	∅ s	Ziele	∅ s
200	141	0,7897 s	143	1,1072 s	159	1,0623 s
400	159	0,8732 s	114	1,2248 s	146	0,7396 s
600	160	0,9002 s	110	2,5931 s	161	0,7707 s
800	159	0,7489 s	146	1,3635 s	133	1,7164 s
1000	167	0,9675 s	138	0,9148 s	145	0,8260 s
1200	159	1,0017 s	166	1,1255 s	113	1,9830 s
1400	92	0,7424 s	169	1,0653 s	119	1,0363 s
1500	149	1,4744 s	169	0,7388 s	153	1,5942 s

Tabelle 6.4: Ergebnisse des Rastertests drei verschiedener Durchläufe

Abbildung 6.13 zeigt das Schiff, nachdem es den Pfadtest vollständig abgeschlossen hat. In der Abbildung wurden dabei alle Ziele vom Schiff erreicht. Für die Evaluation des Pfadtests werden nur diejenigen Ergebnisse betrachtet, in denen das Schiff ebenfalls alle Ziele erreichen konnte.

In Tabelle 6.5 sind die Ergebnisse des Pfadtests der drei Durchläufe zu erkennen, die schon oben in Tabelle 6.4 betrachtet wurden. Wie schon erwähnt, werden nur diejenigen Ergebnisse mit aufgeführt, in denen alle Ziele erreicht werden konnten.

	Durchlauf 1	Durchlauf 2	Durchlauf 3
Episode	Zeit	Zeit	Zeit
200	-	-	-
400	-	-	-
600	14,2150 s	-	4,8909 s
800	-	-	-
1000	10,0422 s	-	-
1200	13,2109 s	4,4258 s	-
1400	-	3,0880 s	-
1500	6,7158 s	3,4102 s	13,9994 s

Tabelle 6.5: Ergebnisse des Pfadtests für drei verschiedene Durchläufe

Die benötigte Zeit der Ergebnisse beträgt zwischen 3,088 s und 14,2150 s. Besonders gut schneidet in diesem Testszenario der zweite Durchlauf ab, da ihm die drei besten Ergebnisse zugeordnet werden können. Vergleicht man die erfolgreichen Stadien der drei Durchläufe in 6.5 mit den Ergebnissen aus der Tabelle 6.4, dann lässt sich feststellen, dass die erfolgreichen Durchläufe mindestens 153 Ziele in dem vorherigen Test erreicht haben.

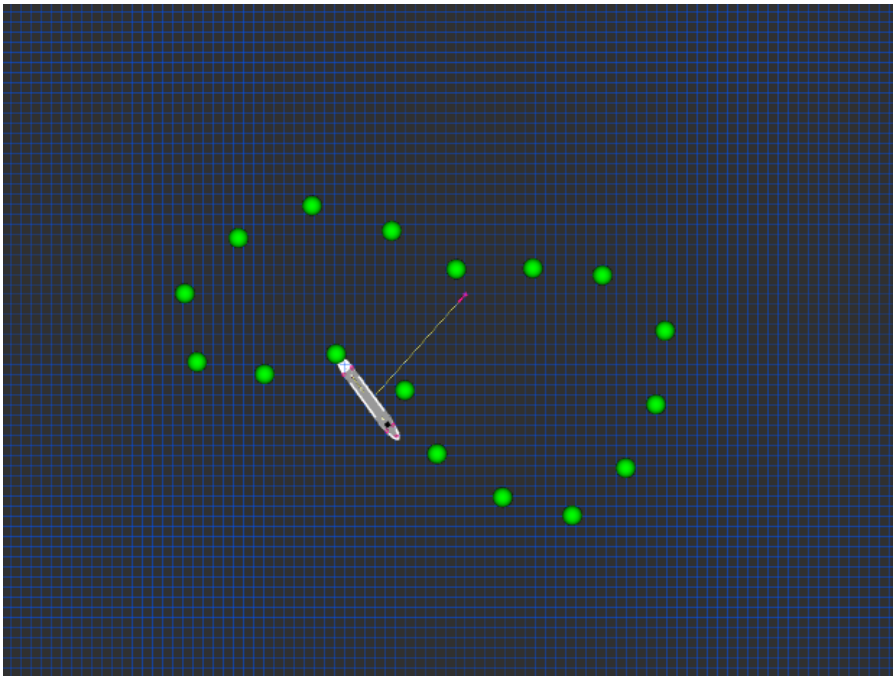


Abbildung 6.13: Vollständig abgeschlossener Pfadtest

6.2.1 Abgeänderte Testszenarien

Aktormisskonfiguration

In Tabelle 6.6 sind die Ergebnisse des abgewandelten Testszenarios mit der veränderten Aktorik zu sehen. Dabei wurden die Ergebnisse des Pfadstests mit denen des Zielanfahrttests zusammengelegt. Für das Lernen benötigte dieser Durchlauf 7268,45 s.

Episode	Ziele	∅ s	Pfad
200	150	1,0709 s	-
400	150	0,8510 s	-
600	150	0,6742 s	-
800	167	0,7441 s	-
1000	129	0,5395 s	-
1200	91	1,0017 s	-
1400	148	0,8635 s	-
1500	156	0,7105 s	-

Tabelle 6.6: Ergebnisse der veränderten Aktorik

Auch mit veränderter Aktorik sind die Ergebnisse des Zielanfahrttests für alle Stadien des selbstlernenden Regler erfolgreich. Das beste Ergebnis liegt bei 167, während das schlechteste Ergebnis bei 91 Zielen liegt. Dieser selbstlernende Regler ist jedoch in keinem Stadium in der Lage, den Pfadtest erfolgreich abzuschließen.

Verändertes Gleitverhalten

In der Tabelle 6.7 sind die Parameter zu sehen, die für das *usv_gazebo*-Plugin genutzt wurden, um ein anderes Gleitverhalten der nHAWigatora zu simulieren. Durch die veränderten Werte gleitet das Schiff länger und Drehungen in und entgegen der Yaw-Richtungen benötigen einen höheren Kraftaufwand.

Parameter	Wert
$X_{\dot{u}}$	$2\,\mathrm{kg}$
$Y_{\dot{v}}$	$0{,}5\,\mathrm{kg}$
$N_{\dot{r}}$	$0{,}1\,\mathrm{kg\,m^2}$
X_u	$4\,\mathrm{kg/s}$
$X_{u\|u\|}$	$0\,\mathrm{kg/m}$
Y_v	$6\,\mathrm{kg/s}$
$Y_{v\|v\|}$	$0\,\mathrm{kg/m}$
N_r	$2{,}4\,\mathrm{kg\,m^2/(s\,rad)}$
$N_{r\|r\|}$	$0\,\mathrm{kg\,m^2/rad^2}$
Z_w	$4\,\mathrm{kg/s}$
K_p	$2\,\mathrm{kg\,m^2/(s\,rad)}$
M_q	$0{,}5\,\mathrm{kg\,m^2/(s\,rad)}$

Tabelle 6.7: Veränderte Parameter für die Berechnung der hydrodynamischen Kräfte

In Tabelle 6.8 sind die Ergebnisse des Szenarios mit dem veränderten Gleitverhalten zu erkennen. Wie in Tabelle 6.6 wurden die Ergebnisse des Zielfahrttests und Pfadtests zusammengefasst.

Auch in diesem Testszenario konnte der selbstlernende Regler in jedem Stadium den Zielfahrttest bestehen. Das beste Ergebnis erreicht er nach 1400 Episoden mit 165 Zielen und das schlechteste Ergebnis nach 1000 Episoden mit 97 Zielen. Dieser Regler erreicht dreimal ein erfolgreiches Abschließen des Pfadtests.

Episode	Ziele	∅ s	Pfad
200	165	0,4894 s	7,2211 s
400	147	2,3153 s	-
600	167	1,2141 s	12,1481 s
800	160	0,9094 s	-
1000	97	2,4091 s	-
1200	142	0,7112 s	-
1400	165	1,6541 s	-
1500	146	1,2197 s	3,3756 s

Tabelle 6.8: Ergebnisse der veränderten Wasserdynamik

6.3 Ressourcenaufwand

In diesem Kapitel wird der gemessene Ressourcenaufwand des selbstlernenden Reglers vorgestellt. Die Messungen wurden während des Lernens auf dem Bordcomputer der nHAWigatora durchgeführt. Der Bordcomputer ist ein Raspberry Pi 3B+ und verfügt über einen Cortex-A53, dessen 4 Kerne mit 1,4 GHz getaktet wird, sowie über 1 GB Arbeitsspeicher. Abbildung 6.14 zeigt die Auslastung der Prozessorkerne nach ca. 200 Episoden. Die durchschnittliche Auslastung der Kerne liegt bei ungefähr 33%. Während des Lernens weicht die durchschnittliche Auslastung nur wenig von dem in der Abbildung vorgestellten Wert ab. Die Auslastung der Kerne liegt stets zwischen 25% und 45%. Der RAM ist nur mit der Hälfte seiner maximalen Kapazität ausgelastet.

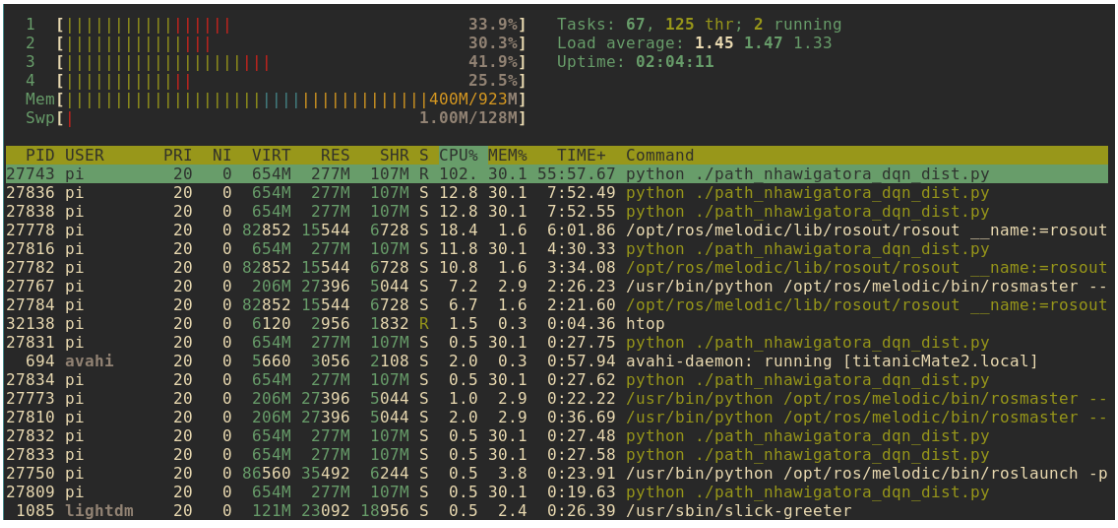


Abbildung 6.14: Prozessorauslastung des Raspberry PIs

In Abbildung 6.15 ist die Auslastung der CPU im Zeitverlauf zu erkennen. Die gezeigten Werte sind die durchschnittliche Last über die letzten 15 Minuten. Da der Raspberry Pi 4 Kerne besitzt, wäre die maximal Auslastung 4. In der Abbildung ist der Start des Lernens zu erkennen. Nach ungefähr einer Minute ist zu erkennen, dass sich der Leistungsverbrauch des Pis bei einer Last von ca. 1.15 festsetzt.

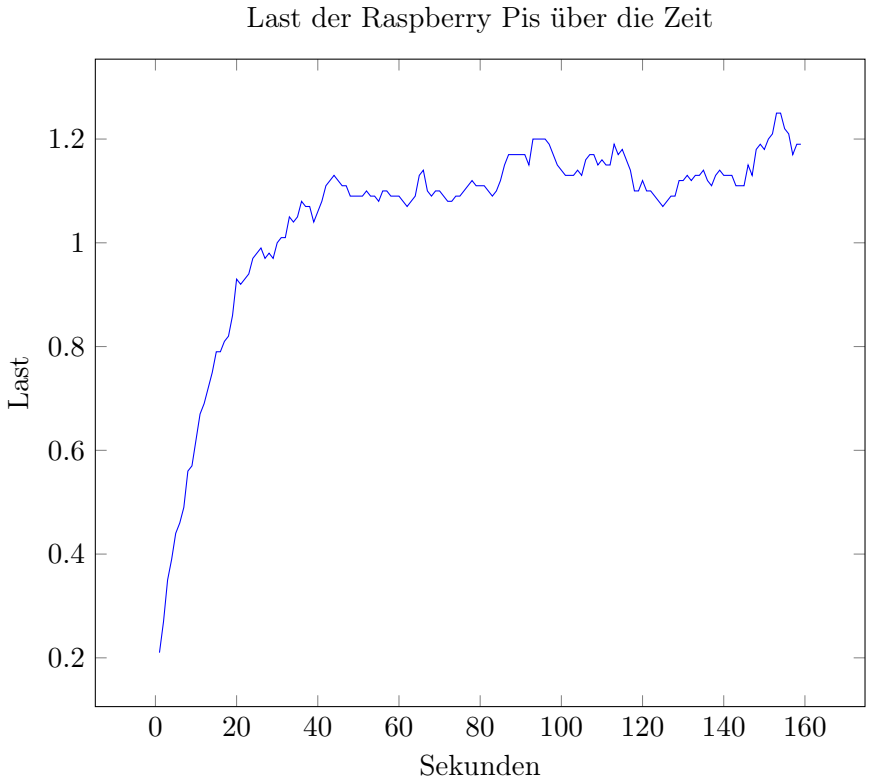


Abbildung 6.15: Last auf dem Raspberry Pi im Zeitverlauf

7 Diskussion

In diesem Kapitel werden die Ergebnisse der Evaluation diskutiert und eingeordnet. Die Ergebnisse der Evaluation haben gezeigt, dass sich die Simulation für die Entwicklung und Umsetzung eines selbstlernenden Reglers eignet. Das gelernte Verhalten des in der Simulation trainierten Reglers konnte in den durchgeführten Tests sehr gute Ergebnisse erzielen. Über die Nutzbarkeit des selbstlernenden Reglers in der Realität konnten die Ergebnisse jedoch keine abschließende Antwort liefern.

7.1 Lerngeschwindigkeit

Ein wichtiger Bestandteil der Eignung der Simulation ist die Zeit, die benötigt wurde, um den selbstlernenden Regler zu trainieren. Bereits nach wenigen Episoden konnten sehr gute Ergebnisse erzielt werden. Die Evaluationsergebnisse zeigen in Tabelle 6.4, dass der dritte Durchlauf bereits nach 200 Episoden 159 von 169 Zielen erreichen kann. Für dieses Ergebnis benötigte die Simulation eine Berechnungszeit von 1400 s.

Ein vergleichbares Training in der Realität, würde für ein ähnliches Ergebnis deutlich mehr Zeit benötigen. Da die Simulation mit einem RTF von 125 durchgeführt wurde, wäre die Dauer des realen Trainings also ungefähr 48 Stunden, statt der 25 Minuten in der Simulation. Dieses Ergebnis zeigt deutlich, dass die größte Stärke der Simulation in der Beschleunigung des Lernvorgangs liegt. Dadurch konnten verschiedene Hyperparameter und Umgebungen schnell ausprobiert und evaluiert werden.

Die erhöhte Geschwindigkeit der Simulation hat jedoch auch den Nachteil, dass dafür die Genauigkeit verloren geht. Wird das Verhalten des Schiffs in Realzeit simuliert, können wesentlich mehr Berechnungen durchgeführt werden, was die Simulation präziser macht. Deshalb ist es möglich, dass das gelernte Verhalten des Reglers nicht in der Realität anwendbar ist, dies zu evaluieren ist jedoch nicht Teil dieser Arbeit, da die dafür notwendigen Tests mangels Odometrie nicht möglich sind.

7.2 Lernprozess

Die während der Evaluation erreichten Ergebnisse zeigen, das der Regler in der Lage ist, ein Verhalten zu lernen, das sowohl den Rastertest als auch den Pfadtest mit einem guten Ergebnis abschließen kann. Entgegen der Erwartung erreichen sogar alle Stadien des selbstlernenden Reglers mehr als die Hälfte der Ziele im Rastertest und bestehen ihn damit. Vor der Durchführung der Tests wurde angenommen, dass die Mehrheit der getesteten Netze den Rastertest nicht bestehen, vor allem die Netze aus niedrigen Episoden. Es überrascht vor allem, dass auch der Pfadtest bestanden werden kann. Der verwendete DQN-Algorithmus sollte nur als Beispiel für einen selbstlernenden Regler dienen. Die Ergebnisse zeigen jedoch, dass der Algorithmus möglicherweise sogar in der Realität verwendet werden kann.

Ein gutes Ergebnis ist weiterhin, dass der selbstlernende Regler auch Variationen der Umgebung mit einem guten Ergebnis abschließen kann. Die Regler, die in einem veränderten Testszenario trainiert wurden, konnten ein ähnlich gutes Ergebnis erzielen wie die zuvor trainierten Regler. Der selbstlernende Regler ist also auch in der Lage, eine andere Aktorikkonfiguration oder ein anderes Gleitverhalten zu erlernen. Beides kommt durch Veränderungen an der nHAWigatora häufig vor. Die Ergebnisse bedeuten auch, dass der selbstlernende Regler eine mögliche perfekte Abbildung der Realität in der Simulation erlernen kann.

7.3 Vergessen von leistungsstarkem Verhalten

Die Leistung des selbstlernenden Reglers selbst ist anders, als zuerst angenommen wurde. Die erwarteten Ergebnisse waren eine kontinuierliche Verbesserung des gelernten Verhaltens, da mit der Zeit der Zustandsraum mehr erforscht wird. Mit der gesammelten Erfahrung sollte dann die Leistung des Reglers immer besser werden. In traditionellen Q-Learning-Aufgaben, beispielsweise in einer Grid World, an der Sutton und Barto [2018] das Q-Learning erklärt, ist ein solches Verhalten zu beobachten. Stattdessen zeigen die Ergebnisse, dass der selbstlernende Regler erfolgreich gelerntes Verhalten wieder "vergisst". Diese Ergebnisse werden auch in der Tabelle 6.4 sichtbar. Der erste Durchlauf erreicht seinen Höhepunkt in Episode 1000 mit 159 erreichten Zielen und nimmt dann wieder ab. In Episode 1400 können wiederum nur 92 Ziele erreicht werden.

Dieses Verhalten ist für den verwendeten DQN-Algorithmus ein bekanntes Problem. Roderick u. a. [2017] stellt für diesen Effekt den Begriff "Catastrophic Forgetting"[Roderick u. a., 2017] vor. Für das "Catastrophic Forgetting"gibt es verschiedene Ursachen. Zum einen ist das Approximieren eines großen Zustandsraum mithilfe eines neuronalen Netzes und der regelmäßigen Batch-Updates von sich aus schon instabil. Die von Mnih u. a. [2015] vorgeschlagenen Verbesserungen, das *Experience Replay* und *Target Network*, helfen, die Instabilität deutlich zu minimieren, jedoch nicht, sie zu eliminieren. Ein weiterer Grund ist das Benutzen der Q-Werte, um eine Policy zu erhalten, anstatt direkt die Policy zu verbessern. Ein Nebeneffekt dieses Verfahrens ist, dass die Verbesserung des Q-Funktions-Approximator eine schlechtere Policy liefern kann [Roderick u. a., 2017].

Damit das in dieser Arbeit vorgestellte Verhalten in der Realität genutzt werden kann, darf nicht das gelernte Verhalten am Ende des Lernprozesses in der Realität eingesetzt werden. Stattdessen müssen die Stadien des selbstlernenden Reglers zwischengespeichert und auf ihre Leistungen untersucht werden. Das Stadium mit den besten Leistungen sollte dann in der Realität eingesetzt werden, um ein bestmögliches Ergebnis zu erreichen.

7.4 Verhaltensmuster

Während der Evaluation konnte bei den selbstlernenden Reglern ein gemeinsames Verhaltensmuster erkannt werden. Viele Regler hatten eine ähnliche Strategie, um ein Ziel zu erreichen: Zunächst wird das Boot mithilfe des Bugstrahlruders und des Propellers in die Nähe des Ziels manövriert. Der Regler verfehlt das Ziel knapp und fährt dann direkt wieder in die Position zurück, in der er zuvor war. Dies wird dann schnell hintereinander ausgeführt, was in einer wischenden Bewegung resultiert. Nachdem die "Wischenbewegungëinige Zeit ausgeführt wurde, gelingt es dem Regler, das Ziel zu erreichen. Vor allem die Stadien der Regler mit einem erfolgreichen Ergebnis in dem Rastertest wiesen dieses Verhalten auf. Erwartet wurde aber, dass der Regler direkt das Ziel anfährt, ohne, dass zuvor eine Wischbewegung durchgeführt wird. Durch diese Bewegung wird die summierte Belohnung des Reglers nämlich immer weniger.

Eine Erklärung für dieses Verhalten ist die Diskretisierung der Aktionen. Der Regler hat dabei nur den maximalen Schub der Aktorik zur Auswahl. Deswegen ist es nicht möglich, präzise Änderungen an seiner Position und Ausrichtung vorzunehmen. Dadurch schießt das Schiff direkt am Ziel vorbei, was der Regler zu kompensieren versucht, indem er die entgegengesetzte Aktion auswählt.

Der Regler bräuchte also die Möglichkeit, das Schiff deutlich genauer zu manövrieren. Dies könnte zum Beispiel geschehen, indem die vier möglichen Aktionen durch vier weitere ergänzt werden, die nur 25% des möglichen Schubs auf die Aktorik geben. Eine weitere Möglichkeit wäre das Verwenden eines Verfahrens, welches kontinuierliche Aktionen unterstützt. Dafür würde sich zum Beispiel der Deep Deterministic Policy Gradient (DDPG)-Algorithmus eignen, der von Lillicrap u. a. [2016] vorgestellt wurde. Diese Lösungsansätze umzusetzen würde jedoch den Umfang dieser Arbeit überschreiten.

7.5 Berechnungsaufwand

Das Lernen auf dem Raspberry Pi 3b+ benötigt den Ergebnissen der Evaluation nach nur 25%- 45% der möglichen CPU-Ressourcen. Der Verlauf der CPU-Auslastung zeigt keinen deutlichen Anstieg der Auslastung über die Zeit, abgesehen von der Startphase des Programms. Die Auslastung ist sehr konstant und enthält keine großen Sprünge. Die Erwartung war es, solche Sprünge in der Auslastung zu erkennen, da die beiden neuronalen Netzwerke θ und $\theta-$ nach einer gewissen Anzahl von Episoden synchronisieren. Wahrscheinlich ist der benötigte Aufwand jedoch nicht so aufwendig wie angenommen, da nur die gelernten Parameter kopiert werden mussten. Der Prozess der Backpropagation benötigt vermutlich deutlich mehr CPU-Zeit, weshalb der Aufwand des Kopierens vernachlässigt werden kann.

Der gemessene CPU-Verbrauch ist nur der Verbrauch des selbstlernenden Systems. Um herauszufinden, ob die Ressourcen des Raspberry Pi 3b+ auch ausreichen, um den selbstlernenden Regler zusammen mit den anderen Softwareprozessen auf der nHAWigatora laufen zu lassen, muss die Gesamtauslastung des Systems gemessen werden. Erfahrungswerte zeigen jedoch, dass die CPU-Auslastung der nHAWigatora ohne den selbstlernenden Regler etwa 20% beträgt. Die Ressourcen reichen also aus, selbst wenn das komplette Bordprogramm auf dem Schiff ausgeführt wird.

Sollten die Ressourcen nicht ausreichen, gibt es verschiedene Möglichkeiten, den selbstlernenden Regler effizienter zu machen. Aktuell ist das selbstlernende System in der interpretierten Programmiersprache Python geschrieben. Die Geschwindigkeit der Berechnung könnte deutlich steigen, wenn man das Programm in eine kompilierte Programmiersprache wie C++ übersetzt. Eine weitere Möglichkeit wäre es, auf dem Schiff

zusätzliche Hardware mit Tensor Processing Units (TPUs) zu verbauen. TPUs sind dedizierte Rechenkerne für die Berechnung von ML-Anwendungen, die vor allem Berechnungen beschleunigen können, welche die Programmbibliothek Tensorflow benutzt, wie der in dieser Arbeit vorgestellte selbstlernende Regler.

7.6 Simulationsgenauigkeit

Die Eignung der Simulationsumgebung für das Entwickeln und Umsetzen eines selbstlernenden Reglers ist auch abhängig von der Genauigkeit der Simulation und der Realität. In diesem Abschnitt werden die Ergebnisse der Evaluation von Aktorik, Festkörpereigenschaften und Sensorik diskutiert.

7.6.1 Aktorik

Die Evaluation der Aktorik konnte keine zufriedenstellende Ergebnisse liefern. Das genutzte Messwerkzeug hat eine zu geringe Genauigkeit, um die Kräfte der Motoren zu erfassen. Deswegen kann über den Kraftverlauf des Propellers und Bugstrahlruders nicht viel gesagt werden. Das verwendete Plugin für die Darstellung der Antriebe, *usv-gazebo*, ist in der Lage, die ausgeübte Kraft der Aktoren als logistische oder lineare Funktion darzustellen, sowie die Maximalkraft der Aktoren in beide Richtungen festzulegen. Die gemessenen Werte genügen nur, um Letzteres zu parametrisieren. Um die Genauigkeit der Simulation zu erhöhen, müssten die Aktoren nochmal mit einem exakteren Messinstrument untersucht werden. Eine solche Messung konnte nicht mehr in diese Arbeit aufgenommen werden, da dies ein zu großer zeitlicher Aufwand gewesen wäre. Außerdem ist zum Zeitpunkt dieser Arbeit die Aktorik der nHAWigatora durch einen Kurzschluss defekt. Da diese Tests nicht durchgeführt werden konnten, kann keine zufriedenstellende Aussage über die durch die Aktorik ausgeübte Kraft in der Realität gemacht werden, was dazu führt, dass auch nicht die Genauigkeit der Simulation bewertet werden kann.

7.6.2 Festkörpereigenschaften

In den Evaluationsergebnissen zeigt sich, dass die Simulation das Gleitverhalten der nHA-Wigatora auf dem Wasser zu großen Teilen genau abbildet. Die Reaktion auf einen Impuls in Yaw- und Surge-Richtung sind sich sehr ähnlich. Bei den Bewegungen in und entgegen

der Surge-Richtung konnte in der Realität aber eine Schwingung erkannt werden, die in der Simulation nicht zu sehen war.

Die Erklärung dafür könnte das Stampfen der nHAWigatora durch die Wellen im Testbecken des Miniaturwunderlandes sein. Durch das Aufschaukeln der Wellen wird die IMU der nHAWigatora leicht rotiert, was die Beschleunigung in der X-Achse beeinflusst. Dadurch ist eine Schwingung von ungefähr 5 Hz zu sehen. In der Simulation gibt es keine Repräsentation von Wellen, deswegen ist auch dort keine Schwingung in der Beschleunigung der X-Achse zu erkennen. Um die Genauigkeit der Simulation zu steigern, müsste ein Plugin für die Simulationsumgebung *Gazebo* entwickelt werden, welches Wellen simulieren kann. Dies wurde in der vorliegenden Arbeit nicht gemacht, da es über den Rahmen dieser Arbeit hinausgehen würde.

Weiterhin wäre eine Evaluation der Kraftimpulse interessant, die durch die Aktorik auf das Schiff ausgeübt wird. Sind die Werte der IMU dann in der Simulation und Realität ähnlich, kann davon ausgegangen werden, dass die Aktorik und Festkörpereigenschaften richtig parametrisiert wurden. Wie oben schon erwähnt, ist die Aktorik der nHAWigatora zu diesem Zeitpunkt defekt und solche Tests konnten nicht in die Arbeit mit aufgenommen werden.

7.6.3 Sensorik

Die Evaluierung der Sensorik zeigt, dass die Sensorik der nHAWigatora in der Simulation sehr akkurat abgebildet wurde. Die Werte des normalverteilen Rauschens sind für die IMU, die Distanzsensoren und das Lidar nahezu identisch.

Ein Unterschied zwischen Simulation und Realität ist momentan noch die Ungenauigkeit der Sensoren in der Realität durch die Anzahl der Bits, die für die Abfrage der Sensorwerte zur Verfügung stehen. Eine mögliche Erweiterung der Simulation könnte es sein, auch solche Ungenauigkeiten abzubilden.

Die Messungen der Sensoren beeinflussen in der aktuellen Architektur der Umgebung noch nicht den selbstlernenden Regler. In Zukunft könnten die Sensoren jedoch genutzt werden um weitere Testszenarien zu simulieren. Dazu zählt zum Beispiel das Ausweichen von statischen und dynamischen Hindernissen.

8 Fazit

In dieser Arbeit wurde eine Simulationumgebung für das Miniaturschiff nHAWigatora geschaffen und in dieser Simulationsumgebung ein selbstlernender Regler implementiert.

Um die Simulation umzusetzen, wurden die Bestandteile des Softwaresystems der nHA-Wigatora identifiziert, die mit der Hardware auf dem Schiff kommunizieren. Diese wurden durch entsprechende simulierte Komponenten ersetzt. In der Simulationsumgebung wurde ein realistisches Testszenario konstruiert, in dem der selbstlerndende Regler trainiert werden kann. Für die Kommunikation mit einer selbstlernenden Komponente wurde eine standardisierte Schnittstelle für RL-Verfahren zur Verfügung gestellt.

Für den selbstlernenden Regler wurde eine Architektur entwickelt und implementiert, die dem gestellten Testszenario genügt und über die Schnittstelle für RL-Verfahren mit der Simulation kommuniziert.

Die durchgeführten Tests konnten zeigen, dass sich die Simulationsumgebung für die Entwicklung und Umsetzung eines selbstlernenden Reglers für autonome Wasserfahrzeuge eignet. Dies liegt vor allem an dem reduzierten Zeitaufwand im Vergleich zu der Entwicklung und Umsetzung in der realen Welt. Außerdem konnten verschiedene Testszenarien unkompliziert ausprobiert werden. Zusätzlich dazu erzielte das vorgestellte lernende System auch auf den vergleichsweise geringen Ressourcen des Bordcomputers auf dem Miniaturschiff Lernerfolge. Der resultierende selbstlernende Regler ist im besten gemessenen Fall in der Lage, Ziele in einem Radius von 5 m zu erreichen. Auch Szenarien, die dem eigentlichen Anwedungsgebiet entsprechen, konnten auch erfolgreich absolviert werden. Probleme hat der selbstlernende Regler noch durch das "Vergessen"von bereits gelernten Erfahrungen, wodurch keine kontinuierliche Verbesserung mit fortschreitender Lerndauer erreicht werden konnte. Außerdem erlernt der Regler ein unnatürlich wirkendes Verhaltensmuster, indem er versucht, Ziele mit eine wischenden Bewegung zu erreichen.

Die durchgeführten Tests konnten außerdem nicht abschließend den Genauigkeitsgrad der Simulation bestimmen. Die Testergebnisse zeigen, dass die Sensorik gut abgebildet ist.

Auch Festkörpereigenschaften wurden erfolgreich abgebildet, durch das Fehlen von Wellen ist ein wichtiger Faktor der Wasserdynamik jedoch nicht vorhanden. Das Verhalten der Aktorik konnte in der Simulation nur ungenau abgebildet werden, da die zugehörigen Tests nicht mit der notwendigen Messgenauigkeit durchgeführt werden konnten.

8.1 Ausblick

Damit das gelernte Verhalten aus der Simulation direkt in die reale Welt übertragen werden kann, muss sich die Genauigkeit der Simulation verbessern. Besonders wichtig dafür ist die Aktorik der nHAWigatora, die mit ausführlichen Messungen genauer bestimmt und dann in der Simulation abgebildet werden muss.

Die Simulation verfügt momentan auch noch nicht über den Einfluss von Wellen, Strömung und Wind. Diese Kräfte spielen in der Realität eine große Rolle in der Navigation von Fahrzeugen auf der Wasseroberfläche. Um diese Kräfte in der Simulation abzubilden, müsste das verwendete Plugin in der Simulationumgebung erweitert werden. Eine andere Lösung wäre es, ein anderes Plugin zu verwenden, welches die Effekte von Wind, Wellen und Strömung bereits in die Simulation der Wasserdynamik miteinbezieht.

Weiterhin könnten die simulierten Szenarien um statische und dynamische Hindernisse ergänzt werden, die von der Sensorik erfasst werden können. Der selbstlernende Regler kann darauf untersucht werden, ob er in der Lage ist, Hindernissen auszuweichen. Dies würde eine Aussage über die Nutzbarkeit des Reglers in einem realen Umfeld ermöglichen.

Auch ein anderer Lernalgorithmus für das Lösen der vorgestellten Testszenarien wäre interessant. Das verwendete DQN-Verfahren hat seine Grenzen, wie auch in den Tests zu erkennen war. Zum Beispiel könnte das DDPG-Verfahren genutzt werden. Dadurch könnten zum einen die instabilen Lernerfahrungen stabilisiert werden um eine konstant steigende Leistung des selbstlernenden Reglers zu erreichen. Außerdem wird in dem Verfahren auch ein kontinuierlicher Aktionsraum nutzbar gemacht. Dies bedeutet eine präzisere Steuerungsmöglichkeit der Schiffs, was möglicherweise die unnatürlichen Wischbewegungen eliminieren könnte.

Außerdem wäre interessant, ob die in der Simulation gelernten Verhaltensweisen auch direkt auf die Realität übertragbar sind. Dafür müssten in dem Testbecken des Miniaturwunderlands oder einem vergleichbaren Testbecken eine externe Trackinganlage zur

Verfügung stehen, damit das Testszenario aus der Simulation reproduziert werden kann. Stattdessen könnte auch das Schiff selbst eine Odometrie entwickeln, zum Beispiel durch die Fusion der auf dem Schiff verfügbaren Sensoren.

Literaturverzeichnis

[Basler] Basler: *daA1920-30uc - Basler dart.* Basler AG (Veranst.). – URL https://www.baslerweb.com/en/products/cameras/area-scan-cameras/dart/daa1920-30uc-s-mount/. – [Letzter Zugriff: 17.07.2020]

[Bingham u. a. 2019] Bingham, Brian ; Aguero, Carlos ; Mccarrin, Michael ; Klamo, Joseph ; Malia, Joshua ; Allen, Kevin ; Lum, Tyler ; Rawson, Marshall ; Waqar, Rumman: Toward Maritime Robotic Simulation in Gazebo, 10 2019, S. 1–10

[Brockman u. a. 2016] Brockman, Greg ; Cheung, Vicki ; Pettersson, Ludwig ; Schneider, Jonas ; Schulman, John ; Tang, Jie ; Zaremba, Wojciech: *OpenAI Gym.* 2016

[Coppelia Robotics GmbH 2020] Coppelia Robotics GmbH: *Coppelia Sim.* 2020. – URL https://www.coppeliarobotics.com/. – Zugriffsdatum: 2020-05-15

[Coumans und Bai 2016–2019] Coumans, Erwin ; Bai, Yunfei: *PyBullet, a Python module for physics simulation for games, robotics and machine learning.* http://pybullet.org. 2016–2019

[Cyberbotics Ltd. 2020] Cyberbotics Ltd.: *Webots: robot simulator.* 2020. – URL https://www.cyberbotics.com/. – Zugriffsdatum: 2020-05-15

[Echeverria u. a. 2011] Echeverria, G. ; Lassabe, N. ; Degroote, A. ; Lemaignan, S.: Modular OpenRobots Simulation Engine: MORSE. In: *Proceedings of the IEEE ICRA*, 2011

[Fossen 2011] Fossen, Thor I.: *Handbook of Marine Craft Hydrodynamics and Motion Control.* Wiley, 2011

[Goldstein u. a. 2006] Goldstein, H. ; Poole, C.P. ; Safko, J.L.: *Klassische Mechanik.* Wiley, 2006. – URL https://books.google.de/books?id=MyalPQAACAAJ. – ISBN 9783527405893

[Hafner 2009] Hafner, Roland: Dateneffiziente selbstlernende neuronale Regler. In: *University Osnabrück* (2009)

[Hammad u. a. 2017] Hammad, Mohanad M. ; Elshenawy, Ahmed K. ; Singaby, M.I. E.: *The notation of SNAME (1950) for marine vessels.* Jul 2017. – URL https://plos.figshare.com/articles/dataset/The_notation_of_SNAME_1950_for_marine_vessels_/5179675/1

[Haykin 2009] Haykin, Simon S.: *Neural networks and learning machines.* Third. Pearson Education, 2009

[Hokuyo] Hokuyo: *Scanning Laser Range Finer Hokuyo URG-04LX-UG0.* Hokuyo Automatic Co., Ltd. (Veranst.). – URL http://docs.roboparts.de/URG-04LX_UG01/Hokuyo-URG-04LX_UG01_spec.pdf. – [Letzter Zugriff: 17.07.2020]

[InvenSense] InvenSense: *MPU-9250.* InvenSense Inc. (Veranst.). – URL https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf. – [Letzter Zugriff: 17.07.2020]

[Lee u. a. 2018] Lee, Jeongseok ; Grey, Michael X. ; Ha, Sehoon ; Kunz, Tobias ; Jain, Sumit ; Ye, Yuting ; Srinivasa, Siddhartha S. ; Stilman, Mike ; Liu, C. K.: DART: Dynamic Animation and Robotics Toolkit. In: *Journal of Open Source Software* 3 (2018), Nr. 22, S. 500. – URL https://doi.org/10.21105/joss.00500

[Lillicrap u. a. 2016] Lillicrap, Timothy P. ; Hunt, Jonathan J. ; Pritzel, Alexander ; Heess, Nicolas Manfred O. ; Erez, Tom ; Tassa, Yuval ; Silver, David ; Wierstra, Daan: Continuous control with deep reinforcement learning. In: *CoRR* abs/1509.02971 (2016)

[Miniatur Wunderland 2020] Miniatur Wunderland: *Miniatur Wunderland Hamburg - Modellbahn, Modelleisenbahn Hamburg.* 2020. – URL https://www.miniatur-wunderland.de/. – [Letzter Zugriff: 17.07.2020]

[Mnih u. a. 2015] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg u. a.: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Nr. 7540, S. 529–533

[Open Source Robotics Foundation 2014] Open Source Robotics Foundation: *Gazebo.* 2014. – URL http://gazebosim.org/. – Zugriffsdatum: 2020-05-15

[Paravisi u. a. 2019] Paravisi, Marcelo ; H. Santos, Davi ; Jorge, Vitor ; Heck, Guilherme ; Gonçalves, Luiz ; Amory, Alexandre: Unmanned Surface Vehicle Simulator with Realistic Environmental Disturbances. In: *Sensors* 19 (2019), Mar, Nr. 5, S. 1068. – URL http://dx.doi.org/10.3390/s19051068. – ISSN 1424-8220

[Pareigis u. a. 2020] Pareigis, Stephan ; Tiedemann, Tim ; Becke, Martin ; Meisel, Andreas: *autosys*. Jul 2020. – URL https://autosys.informatik.haw-hamburg.de/. – [Letzter Zugriff: 17.07.2020]

[Roderick u. a. 2017] Roderick, Melrose ; MacGlashan, James ; Tellex, Stefanie: Implementing the Deep Q-Network. In: *ArXiv* abs/1711.07478 (2017)

[Russ Smith 2020] Russ Smith: *Open Dynamics Engine*. 2020. – URL https://www.ode.org/. – Zugriffsdatum: 2020-07-26

[Sharp] Sharp: *Sharp GP2Y0E03*. Sharp (Veranst.). – URL https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0e02_03_appl_e.pdf. – [Letzter Zugriff: 17.07.2020]

[SimTK 2020] SimTK: *Simbody*. 2020. – URL https://simtk.org/projects/simbody/. – Zugriffsdatum: 2020-07-26

[Stark 2020] Stark, Franek: Monokulare visuelle Odometrie auf einem autonomen Miniaturschiff. (2020). – URL https://franekstark.de/files/thesis.pdf

[Stark u. a. 2020] Stark, Franek ; Schnirpel, Thorben ; Burau, Henri: *nHAWigatora miniature ship*. Jul 2020. – URL https://autosys.informatik.haw-hamburg.de/platforms/2019nhawigatora/. – [Letzter Zugriff: 17.07.2020]

[Sutton und Barto 2018] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. – URL http://incompleteideas.net/book/the-book-2nd.html

[Watkins und Dayan 1992] Watkins, Christopher J. C. H. ; Dayan, Peter: Q-learning. In: *Machine Learning* 8 (1992), Nr. 3, S. 279–292. – URL https://doi.org/10.1007/BF00992698. – ISSN 1573-0565

[Zamora u. a. 2016] Zamora, Iker ; Lopez, Nestor G. ; Vilches, Victor M. ; Cordero, Alejandro H.: Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. In: *arXiv preprint arXiv:1608.05742* (2016)

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Evaluierung einer Simulationsumgebung zur Umsetzung und Entwicklung eines selbstlernenden Reglers für autonome Wasserfahrzeuge

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original