Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

IMU-based robot pole balancing

vorgelegt von

Sven Kristof Schmidt

MIN-Fakultät

Fachbereich Informatik

Studiengang: Informatik B. Sc.

Matrikelnummer: 6217064

Abgabedatum: 17.03.2023

Erstgutachter: Dr. Norman Hendrich

Zweitgutachter: Dr. Florens Wasserfall

Abstract

In robotics a careful consideration which sensors should be used for gathering necessary information to perform a certain task can be crucial to its success. Prior experiments [1] show, that a robotic arm is able to learn the balancing of an inverted pendulum or pole based on visual data. In this thesis data provided by an inertial measurement unit (IMU) is used in an attempt to teach a robotic arm to balance a pole in two dimensions. The results suggest, that balancing a pole using an IMU is possible for a basic *cartpole* setup, where the cart moves freely in two dimensions. Balancing the pole with a robotic arm however, proved itself to be too difficult, as it becomes impossible to calculate the pole's orientation, if the IMU is subjected to high linear accelerations and thus requires a very smooth and stable movement of the arm in a two dimensional plane.

Contents

1 Introduction **2**
- 1.1 Theoretic Foundation . . . 2
  - 1.1.1 Reinforcement Learning . . . 2
  - 1.1.2 Q-learning . . . 4
- 1.2 ROS . . . 5
  - 1.2.1 MoveIt . . . 6

2 Setup **7**
- 2.1 Basic Setup . . . 7
  - 2.1.1 IMU . . . 9
- 2.2 Development setup . . . 9
  - 2.2.1 WSL-2 . . . 11
  - 2.2.2 Simulation Environment . . . 11
  - 2.2.3 Third Party Modules . . . 13
- 2.3 Implementation . . . 13
  - 2.3.1 Learning Agent . . . 13
  - 2.3.2 ROS Publisher and Subscriber . . . 14
  - 2.3.3 Robot Controller . . . 16

3 Results **17**
- 3.1 2-Dimensional Cartpole . . . 17
- 3.2 Simulation of the Diana-7 robotic arm . . . 28

4 Conclusion **35**

1. Introduction

In recent years, the use of robotic arms in industrial and research applications has grown significantly, with these machines being used to perform increasingly complex tasks with high precision and accuracy. In this thesis we want to let a robotic arm try to balance a pole based on the data provided by an inertial measurement unit (IMU).

The problem of balancing a pole using a robotic arm can be seen as a variation of the well known $cartpole$ [6] problem, with increased complexity in the control of the 'cart's' movement. This task requires a high degree of precision and control, as even small deviations from the optimal position can cause the pole to fall over. We want to explore, if the data provided by an IMU is sufficient and that it would allow a learning agent to solve the problem in comparison to other methods used to observe the environment [1].

We will use a reinforcement learning approach to first teach an agent to balance the pole in a classic 2-dimensional cartpole setup, in order to identify the difficulties specifically related to using an IMU as a source of real-time data. We will then attempt to use the knowledge gained from the first setup and evaluate if a robotic arm can be controlled with sufficient precision to directly emulate the previous setup.

Overall, this thesis aims to evaluate the potential and difficulties of using IMUs as a source of real-time data in highly dynamic situations and if it can be used in solving complex tasks such as balancing a pole.

1.1 Theoretic Foundation

In this experiment a reinforcement learning approach, or more specifically a Q-learning algorithm, was used in order to train an agent, that is balancing the pole. The following chapter is giving a brief explanation of the theoretic foundations and was heavily reliant on [2]. Therefore additional citation was used to specify more accurately where exactly a specific piece of information was found.

1.1.1 Reinforcement Learning

Reinforcement learning is based on real world observations, in which humans and animals learn through interaction with their environment and observation of the effect their actions have

on it [2] [21]. In machine learning this concept is realized by letting the learning individual, also called $agent$, learn how they can maximize a numerical reward when interacting with the $environment$. It is important that the agent learns on their own through trial and error, as this is one of the core concepts of reinforcement learning.

The main elements of reinforcement learning, aside from the aforementioned agent and environment, are the $policy$, immediate $reward$ and long term $value$. The policy largely determines which action is taken by the agent on encountering a given state. The immediate reward is a form of evaluation how beneficial the state of the environment is, after the agent has taken their action. The value of an action is more of a long term reward, as it specifies the expected cumulative reward the agent could receive further into the future [2] [23].

The value of an action can be higher than that of another one, even if the immediate reward from taking it is lower, because it might allow the agent to enter a state in which new actions promise an even greater reward than what could have been received before. The goal of any reinforcement algorithm is to maximize the long term reward and to prefer the value of an action over its immediate reward [2] [23].

One of the challenges an agent might find themselves confronted with, that is unique to reinforcement learning, is when to exploit their acquired knowledge and when to explore new possibilities. Just using exploitation will cause an agent to favor less than optimal actions over ones that would be better, because the expected reward of the former is known to the agent, while that of the latter isn't. However if the agent just explores using random actions all the time, they can not maximize the reward function in the long term by using the information they learned. A mix of both exploration and exploitation is necessary, as every state has to be visited multiple times before the expected reward can be reliably determined [2] [22].

One possible way to deal with the problem of exploration and exploitation is to use the ϵ-greedy method, in which a parameter $0 < \epsilon < 1$ determines when to explore and when to exploit. An agent using this method will choose to explore with a probability of ϵ and to exploit with a probability of $1 - \epsilon$ [2] [3] [24].

Formally reinforcement learning is modeled after a Markov decision process. An agent interacting with its environment is at time step t presented with the state s_t, causing them to take action a_t. On the next time step $t + 1$ the environment is now in state s_{t+1} and the agent receives a reward r_{t+1} giving rise to the sequence $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}, ...$ Any state at any given time is solely dependent on its directly preceding state and the action taken. As such the state s_{t+2} is neither dependent on s_t nor on a_t. This allows it to define a function $p_a(s, s')$, for the state transition probability from s to s' with action a [2] [25].

The Markov decision process can then be defined by the tuple (S, A, R_a, P_a) in which

- S is a set of states s a system can take on, also called state space,

- A is a set of actions a that can be taken, also called action space,
- $R_a(s, s')$ is the reward received for transitioning from state s into state s' after taking action a.
- $P_a(s, s')$ is the probability, that taking an action a causes the system to transition from state s into state s',

The policy based on which an agent selects the action in a given state can be defined as a function mapping the state space to the action space. The goal of the Markov decision process is to find an optimal policy function $\pi*$. [2]26 [27]

1.1.2 Q-learning

The reinforcement learning algorithm used in this experiment is called Q-learning [5]. It requires no model, and is an off-policy algorithm. This means no knowledge about the state transition probability is needed and the optimal action value function is approximated independently from the policy [2] [23] [28] [29].

As a tabular algorithm that stores all values of the action value function $Q(s, a)$ in a table [2] [21], Q-learning requires a discrete state and action space [6]. This means in a case were the state space of the environment is continuous, it first needs to be approximated by a discrete state space. This can be done by creating buckets in which all states are sorted into.

Once the requirements for the state and action space are met, the Q-learning algorithm can be described by the following steps [2] [6] [28]:

- Before the loop initialize $Q(s, a)$ for all $s \in S$ and $a \in A$.
- Select $a_t(s_t) = max_a Q(s_t, a)$ or $a_t(s_t) = rnd_a Q(s_t, a)$ using ϵ-greedy policy.
- Observe reward r_{t+1} and next state s_{t+1}
- Update $Q(s_t, a_t)$ following the Bellman equation,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right) \tag{1.1}$$

  where α is the learning rate and γ is the discount factor for future rewards.
- Repeat the loop from step two by selecting $a_{t+1}(s_{t+1})$

As many other reinforcement learning algorithms do, Q-learning also suffers from the *curse of dimensionality*. This means that for large state and action spaces the algorithm will take exponentially longer to converge [6]. As an example, the state space used to describe the environment in this experiment has four variables $S = (p, v, \theta, \omega)$. If each of those variables can take on ten different values and if the action space also consists of ten different values, then the size of Q becomes $size_Q = size_S * size_a = 10^5$. It is therefore beneficial to keep the state and action spaces as small as possible (but as large as necessary).

1.2 ROS

The framework used to operate the robot is called ROS (Robot Operating System) [7], it is an open-source software framework for building robotic systems. It provides a set of libraries and tools to help developers create complex robotic applications by abstracting many of the low-level details of robotics, including hardware interfaces, messaging protocols, and driver development. ROS is language independent and has been implemented in C++, Python, and Lisp. Experimental libraries for Java and Lua also exist. The language used in this thesis when working with ROS is **Python**. At its core, ROS is a messaging system, that enables communication between different software modules running on a robot. The implementation of this experiment is highly reliant on three of the core concepts of ROS, which are $publisher$, $subscriber$ and $services$. Since these concepts are important to understanding the certain elements of this thesis, they will be given a short explanation.

A publisher node is an executable, which will constantly broadcast messages to a set topic [8]. A publisher can be created through the command

```
pub = rospy.Publisher('\topic_name',
                      MessageType,
                      queue_size)
```

With this the node declares that it will publish a message of certain type to the topic with the specified name. Whenever a publisher wants to publish something they can do so using the command

```
pub.publish(message)
```

It is important, that the message being published is of the specified type. A message of the type $String$ can not be published on a topic that is advertised with a message of the type $Float$, as every message type represents a class with a fixed data structure.

A subscriber node is a node that subscribes itself to an existing topic [8]. It can do so by using the command

```
rospy.Subscriber('\topic_name', MessageType,
                 callback_function, queue_size)
```

Whenever a new message is published by a publisher node the $callback\_function$ of the subscriber is executed allowing them to react to the message. Additionally to subscribing to the topic the subscriber node also needs to implement the $callback\_function$.

```
def callback_function:
    #do something
```

A node can be a publisher on multiple topics and be subscribed to multiple other topics at the same time.

Services are used for $request$ and $reply$ interaction in ROS. They are somewhat similar

to public functions of classes in many object oriented programming languages. Every service is offered under a unique name and consists of a pair of the same message structures that are available for topics in the publisher and subscriber model. One message is for the request and is send to the node providing the service. The other is send as a reply to the node requesting the service. The requesting node is inactive while it waits for the reply to a requested service.

1.2.1 MoveIt

MoveIt is an open-source software package for motion planning and manipulation in robotics [14]. One of the key features of MoveIt is its integration with ROS, which allows it to work seamlessly with other ROS packages and tools.

In this experiment MoveIt is used for operating the robotic arm. The motion planning function of the package is actually only used at the very beginning of the simulation, before starting the training, to move the robotic arm into a suitable start position. This is done using its integration with the graphical user interface Rviz.

The motion planning feature was not used to move the arm during the simulation because it is simply not quick enough, needing up to 100ms when less than 10ms are available. However the services computing the **forward kinematics** and **inverse kinematics** provided by MoveIt were used.

The forward kinematics are used to determine the position of the last link within a move group, the end-effector, based on a given set of joint states. There is always exactly one solution for the forward kinematics.

The inverse kinematics are used to determine a set of joint states based on a given position of the end-effector of a move group. Different combinations of joint states can lead to the same position of the end-effector, which is why the inverse kinematics can have multiple solutions. The given point could also be out of reach for the robot, which means it is possible for the inverse kinematics to have no solution.

It should be noted, that moving the end-effector from a starting point to a target point might cause a great shift in the joint states of the arm even though the points are close to each other, simply because no other configuration of joint states would allow the arm to reach the target. A requirement for the starting position was therefore, that no large shifts in the joint states would be necessary for the arm to reach any given position within the vicinity of the starting point.

2. Setup

Before any training of the learning agent can be done, some issues concerning the physical setup of the experiment need to be sorted out. The main issue is how rotation around the pole's z-axis can be prevented, since the IMU used in this experiment doesn't have a magnetometer. This means that the pole's rotation around the z-axis can not be measured. If the pole were to rotate around the z-axis with $\theta_z = \pi$ and fall over to the left then the IMU would send the same data as if it had no rotation around z ($\theta_z = 0$) and fell over to the right. The learning agent could therefore not differentiate between those cases but would need to take two different actions.

Smaller issues concern the length of the pole, its diameter and mass as well as where the IMU is mounted to the pole. These are important to know, when training the agent in a simulator first, because the physical setup and the simulation setup should be as close as possible.

2.1 Basic Setup

The model of the robotic arm, that is supposed to balance the pole, is a Diana7, which is a force-controlled robotic arm with 7 degrees of freedom.

The pole used in this experiment has a length of $l = 1m$ and a diameter of $d = 0.008m$. It has a mass of m $0.170kg$ and the center of mass should be close to the middle. The IMU is mounted on the bottom of the pole on a connector block with a spherical joint. At the front side of the connector a pin is placed that connects with a hole in the socket to prevent rotation around the z-axis. Fig 2.1 and 2.2 show the pin-in-hole concept from both the bottom and the top.

To prevent the pole from completely falling over, the socket has wings added to each of its corners, that allow for wires or rubber bands to be attached. Alternatively a cup could be placed around the pole to limit the maximum rotation angle around its x- and y-axis. The full physical setup can be seen in the figures 2.3 to 2.6. They show the winged socket in detail, as well as some pictures from the pole held by the robotic arm.

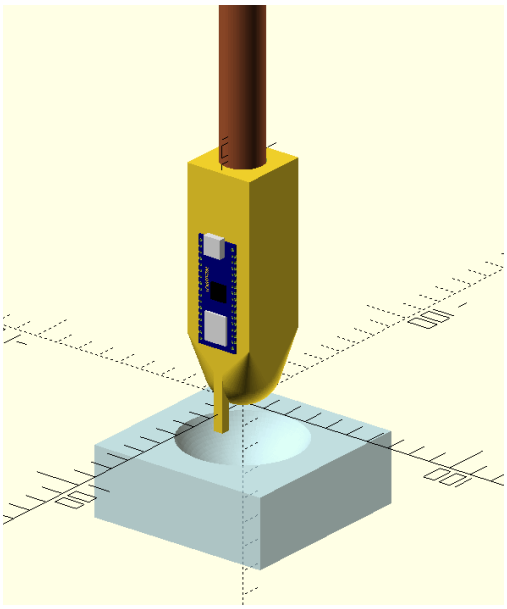

Figure 2.1: Pin-in-hole concept of the spherical joint connector. Top view angle

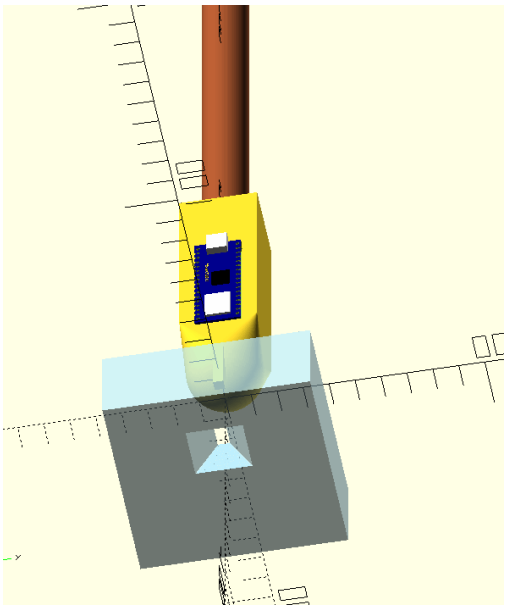

Figure 2.2: Pin-in-hole concept of the spherical joint connector. Bottom view angle

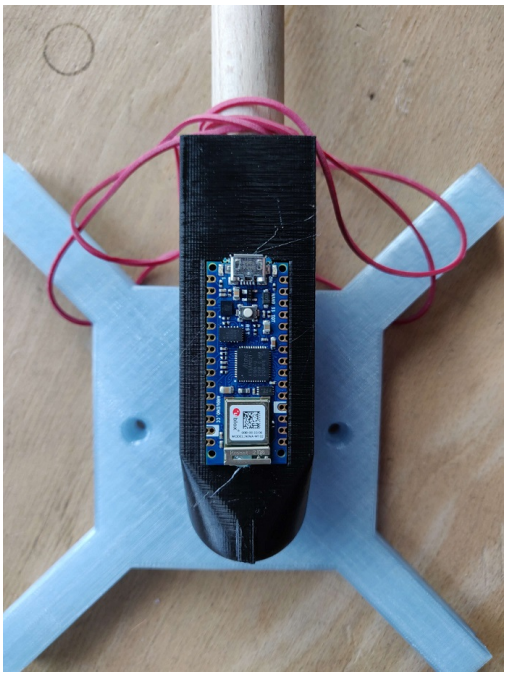

Figure 2.3: Laboratory shot of the IMU and the winged socket.

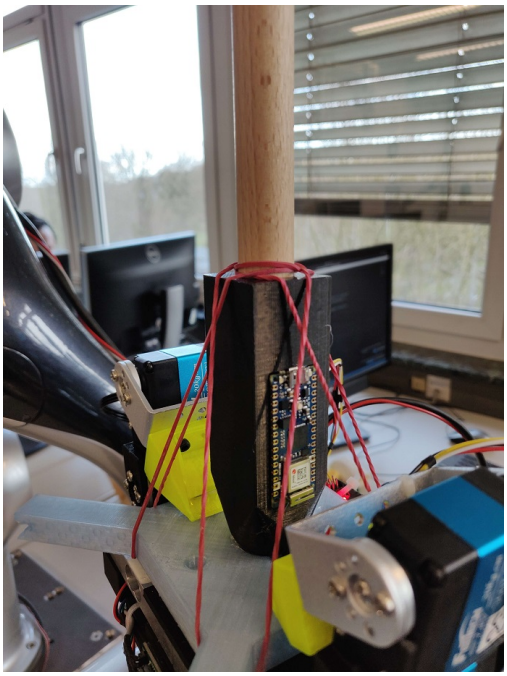

Figure 2.4: Close up laboratory shot of pole held in place by rubber bands

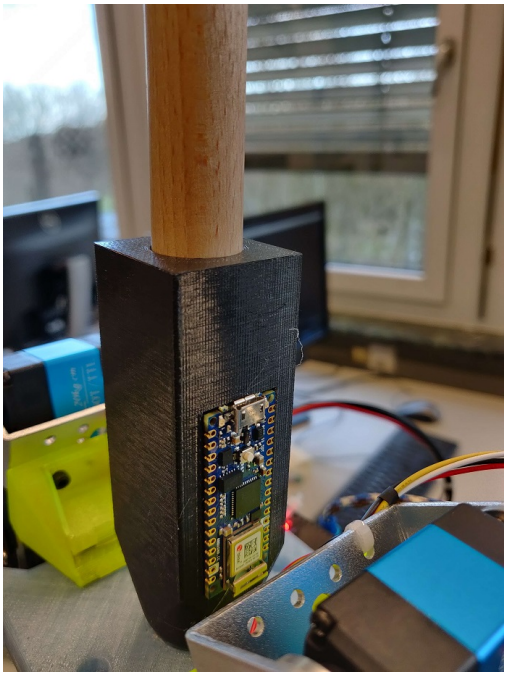

Figure 2.5: Close up laboratory shot of the robotic arm holding the pole

Figure 2.6: Panorama laboratory shot of the robotic arm holding the pole

2.1.1 IMU

The microchip, which is used in this setup, is the 'Arduino Nano 33 IoT'. It has a programmable micro controller, comes with a micro USB port, as well as a WiFi and Bluetooth module for connectivity, and it contains the inertial measurement unit (LSM6DS3), which combines an accelerometer and a gyroscope. The manufacturer of the board provides an IDE (Arduino IDE), which can be used to program it. The manufacturer also provides a library for easy access of the IMU (Arduino_LSM6DS3), which takes care of the IMU's initialization and allows to easily read the data of both the accelerometer and gyroscope through a simple command. It is important to note that the axes of the IMU are fixed according to 2.7 and might need to be transformed depending on how the board is mounted to the pole. Fig 2.5 shows that the IMU is mounted with its x-axis facing upwards, which means the z-axis and x-axis have to be switched before the data is transmitted.

2.2 Development setup

The ROS distribution used in this experiment is 'ROS Noetic Ninjemys' or just noetic for short. It was released on May 23rd 2020 [9] and is primarily targeted at the Ubuntu 20.04 release [10]. The Development was exclusively done on a machine with Windows 10 as its operating system, with which ROS noetic is incompatible. Because of this the development had to be done within a virtual environment with 'WSL2' being one of the simplest ones.

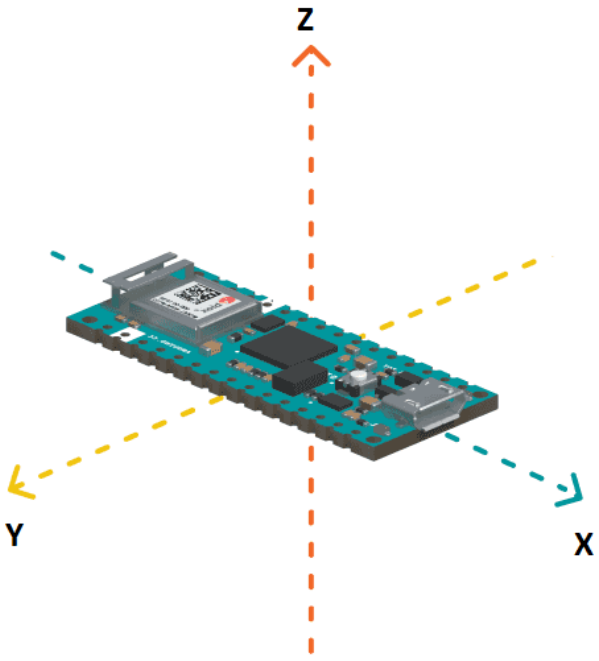

Figure 2.7: Schematic of the Arduino Nano 33 IoT and the axes of its IMU [17]

2.2.1 WSL-2

WSL-2 stands for 'Windows Subsystem for Linux 2' and is a feature in Windows 10 that enables users to run a full Linux environment directly on their Windows desktop. It runs a full Linux kernel in a lightweight virtual machine, allowing users to install any Linux distribution and run it natively on their Windows computer. It is possible to access the Windows file system from within the Linux environment, making the transfer of files between virtual environment and native machine very easy [19]. The development of my ROS package required some features that were not supported by WSL-2 alone, which is why some additional configuration of the environment was needed.

Connecting an USB device

It is necessary to connect the Arduino board to the Linux environment for the purpose of transmitting the IMU's data through serial connection. WSL-2 does not support a USB connection natively but it can be done using the module 'usbipd-win'. Following the instructions from [20] will install the $usbipd$ service and command line tool to be usable in PowerShell on the Windows desktop and the necessary usbipd client package on the Linux environment inside WSL-2. Once this is done USB devices can be attached and detached using the commands

```
usbipd wsl attach --busid <busid>
```

and

```
usbipd wsl detach --busid <busid>
```

while the command

```
usbipd wsl list
```

will list all available devices.

2.2.2 Simulation Environment

The simulator Gazebo is used to pre-train the agent. The simulation setup can be split into two cases.

The first case is a basic 'cartpole' setup [6] but in two dimensions. The setup is created by placing two sliders in the simulation space. The first slider is connected to the world frame through a prismatic joint and can only move along the x-axis. The second slider is connected to the first one also through a prismatic joint and can only move along the y-axis. The pole carrying an IMU is connected to the second slider through two revolute joints, with one allowing rotation around the x-axis and one allowing rotation around the y-axis. A picture of the setup is depicted in fig 2.8.

In the second simulation setup a model of the Diana7 is used in place of the slider. The model consist basically of 8 links $base\_link, link\_1, ..., link\_7$ all connected to their neighbor through a revolute joint. The pole and IMU are the same as in the previous setup and

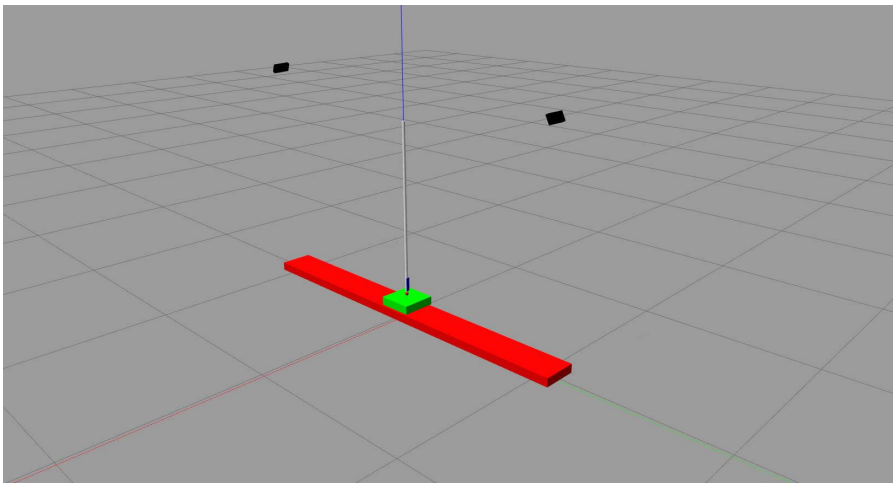

Figure 2.8: Gazebo screenshot of the x-y-slider setup

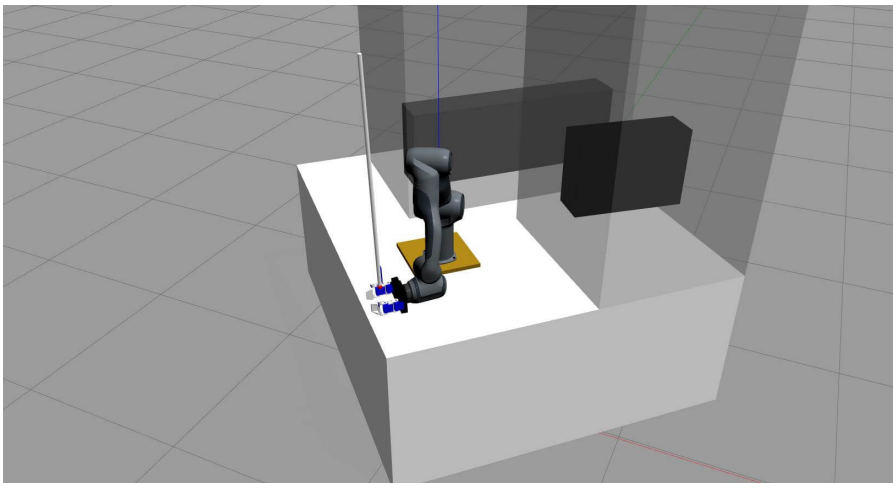

Figure 2.9: Gazebo screenshot of the simulation setup using the Diana7 model.

are connected to the last link $link\_7$ through two revolute joints. A picture of the setup is depicted in fig 2.9.

Both packages creating these simulation setups were provided to me, and are not my own creation.

2.2.3 Third Party Modules

Two third party modules were used to handle the data provided by the IMU:

- **rosserial** [11] is a module that allows the wrapping of ROS messages and sends them over a serial port or network socket. A specified client library exists for Arduino boards called $rosserial\_arduino$. Tutorials are provided alongside the client library that explain in detail how different types of data can be accessed and transmitted using rosserial. A node of the type rosserial_python needs to run on the host machine otherwise no data can be received from the Arduino device.
- **imu_filter_madgwick** [12] is used to process the raw data from the IMU. It fuses the measured angular velocity and linear acceleration into an orientation quaternion. The module allows for dynamic reconfiguration of some key parameters most importantly the parameter $gain$. With low values of the parameter leading to a smoother signal at the cost of slower computation time and higher values leading to faster convergence but more noise.

2.3 Implementation

The implementation of the learning agent is done in three steps.

- The primary agent, which implements the Q-learning algorithm and is responsible for choosing an action based on a given state.
- The ROS publisher and subscriber listening to topics containing necessary information about the environmental state, and publishing the actions chosen by the agent to an appropriate command topic.
- The robot controller, that calculates the trajectory the robotic arm needs to move along in order to execute a chosen action.

2.3.1 Learning Agent

The primary learning agent is implemented by the class *CartHandler* within the file *imu_listener.py* (and *robot_controller.py*), which should be referenced for detailed information of the complete implementation, but a brief explanation for the basic structure of the class will be given in the following section.

On initialization of the class, a four-dimensional array, called *stateList*, representing the state

space and an one-dimensional array, called *actionList*, representing the action space are created. The *stateList* contains four lists with the state values of position, velocity, angle and angular velocity the system can be sorted into. The *actionList* is basically a list of values corresponding to the velocity with which the robotic arm should be moved.

From the *stateList* and *actionList* two five-dimensional Q-tables are created, one for training movement along the x-axis and one for training movement along the y-axis. Both Q-tables are initialized with random values.

The agent determines an action, when its function *getActionFromStateX* (or *getActionFromStateY*) is called. It first takes a given state vector $S(p, v, \theta, \omega)$ and turns it into an index vector $I_s(i_p, i_v, i_\theta, i_\omega)$ where $i_p, i_v, i_\theta, i_\omega$ are the indices of the values $p_i, v_i, \theta_i, \omega_i$ from the *stateList* to which the given state values are closest to. This way the given state vector, which can consist of real number values, is discretized. The index i_a is the index, at which the Q-table $Q(i_p, i_v, i_\theta, i_\omega, i_) = Q_max(i_p, i_v, i_\theta, i_\omega)$ takes on the maximum value for the given state indices. The index a_i is then used to get the action from *actionList*.

During training, the current state index vector I_s and action index a_i are stored until the next step, at which an immediate reward is calculated based on the new state $S'(p', v', \theta', \omega')$ and the equation

$$R_S = \begin{cases} -10000 - 50 \cdot |p'| - 100 \cdot |\theta'|, & \text{if } -0.7 < p' < 0.7 \\ & \text{or } -0.25 < \theta' < 0.25 \\ 10 - 10 \cdot |10\theta'|^2 - 5 \cdot |p'| - 10 \cdot |\omega'|, & \text{otherwise.} \end{cases} \tag{2.1}$$

The immediate reward given is positive if the system is in a state with small angle θ', angular velocity ω' and if it is close to origin position. It can quickly take on high negative values if any of the state values is deemed not beneficial. The reward takes on a negative value of higher magnitude if the state is deemed to be a fail state. The Q-table for state S is then updated following eq1.1.

The training mode itself is activated through the command *setTrainingMode*, which takes values for the learning rate α, the discount γ and the initial exploration rate ϵ as arguments. The exploration rate ϵ decays with a factor of 0.999 on every step, but it has a minimal value of $\epsilon_min = 0.003$.

Alongside the primary agent a debugging tool has been implemented in form of the class *Analytics*. The purpose of this class is to count how often the system is in a certain state, and how often a certain action is taken.

2.3.2 ROS Publisher and Subscriber

The ROS publisher and subscriber are implemented by the class *DataListener*, for which two different versions exist depending on the source file.

The class *DataListener* implemented in *imu_listener.py* is designed for a scenario, where the pole balancing has been simplified to a two-dimensional 'cartpole' problem, in which the pole is balanced on a simple cart that can move along the x- and y-axis.

The main objective of the *DataListener* is to build the state vector $S = (p, v, \theta, \omega)$. To do this the class creates subscribers for the topics */imu/data_raw*, */imu/data* and */joint_states*. The data topics contain linear acceleration and angular velocity measured by the IMU as well as its orientation, which is either the exact orientation provided by the simulator or the computed one calculated by the filter module. The orientation provided by the simulator is more accurate and much more convenient to use, but it would not be available using the actual IMU on the Arduino board.

Before the state vector S is build, the orientation, which is provided in quaternion representation needs to be transformed into Euler angles. After this is done S can be created by adding the position and velocity directly from the joint states. It should be noted, that the rotation around the x-axis corresponds to movement along the y-axis and vice versa.

The action, once determined by the agent, is published to the topics **_joint_co- ntroller/-command* by the *DataListener*. This happens for every callback to the topic */imu/data* (or */imu/data_raw*).

In the scenario, where the pole is balanced by a robotic arm instead of a cart, the current position and velocity of said arm are not directly available by subscribing to the topic */joint_states*. The class *DataListener* implemented in the file *robot_controller.py* is therefore slightly modified. Instead of subscribing to the topic */joint_states*, a listener for the topic */tf* is created. On each callback for the topic */imu/data* (or */imu/data_raw*) a lookup for the coordinate transformation from the links */world* to *link_7* is done, to get the latest position of the latter, which is the last link of the robotic arm excluding the gripper and is therefore its end-effector. This position is stored each step, which allows it to approximate the current velocity using the distance traveled between the last and the current step as well as the time that passed between them.

The action is not published by the *DataListener* class in this scenario but by the class *MotionController* instead, because of the increased complexity of moving the robotic arm compared to moving the cart.

Both versions implement a function to reset the pole and cart/arm to their origin positions, if the system is determined to be in a fail state, at which either the position is too far away from its origin or the angle θ is so large that the pole is seen as having fallen down.

2.3.3 Robot Controller

The class *MotionController* is implemented in the file *robot_controller.py* and its primary objective is to calculate the trajectory, which the arm has to move along. This is necessary because the action chosen by the agent is in the form of a velocity but the arm can only be moved to a new position and not into a direction with a specified velocity.

On initialization the *MotionController* first sets its current position as its origin, making it necessary to first move the robotic arm into a desired starting position by using the graphical user interface of the motion planning tool MoveIt. This origin is not only used on reset as a point to return to, as the orientation and z-axis coordinate are also always used as part of the target position, whenever a new trajectory is calculated.

A new goal for the robotic arm is created on every call of the function *motion*, which is in turn called on every callback of the subscriber to the */imu/data* topic in the class *DataListener*. To determine a new trajectory the current position is first calculated using the current joint positions of the robotic arm and a service called */compute_fk* provided by MoveIt, which calculates the position using forward kinematics. The distance the arm should be moved in x- and y-direction is added to this position in order to create the trajectory goal. The exact distance is dependent on the action determined by the agent and a constant time interval of roughly $10ms$ because the IMU publishes its data with 108Hz.

Once the target position as been determined a service called */compute_ik* is used, which calculates the joint states for a given position using inverse kinematics and is also provided by MoveIt. Since the distances that needs to be traveled in the given time frame can be several centimeters long, multiple points between the start and target positions are calculated. This is done because the trajectory is not necessarily following a straight line and additional way points along the path might help to smooth out the movement. One of the drawbacks doing this, is that calculating the inverse kinematics takes some time, which means creating too many way points is also not feasible, since the time allocated to the completion of the entire callback is $10ms$. Taking longer would cause the *DataListener* to miss some of the messages send by the IMU and the movement of the robotic arm to slow down.

The trajectory was at first published to the topic */diana7_trajectory_controller-/follow_joint_trajectory/goal* and a queue size of 1. This caused the simulation to arbitrarily run into a segmentation fault after a relatively short time. The reason for this went undetected for a long time until it was finally solved by switching to the topic */diana7_trajectory_controller/command* and a queue size of 5.

3. Results

The experiment was split into three steps. During the first step the agent would learn how to balance the pole on a cart moving in two dimensions inside a simulator. The second step would use the pre-trained agent and further train it to balance the pole using a robotic arm inside a simulator. The last step would have been to use this agent to then balance the pole on a real Diana-7 robotic arm. This last step had to be forfeited however, as the agent never learned to balance the pole on the arm inside the simulator, which means the experiment failed on step two.

3.1 2-Dimensional Cartpole

The main objective of this step is to find a setup, that allows an agent to learn, how a pole is balanced by only using corrective movement in two dimensions in an acceptable amount of time. Since it is to be expected, that each step will require further training, the time needed for the learning agent to stably balance the pole is of utmost importance.

The time it takes for the agent to learn is highly dependent on the size of the Q-table, which is in turn dependent on the size of the state and action spaces. It is therefore beneficial to choose the state and action space as small as possible, while still maintaining high enough resolution of both spaces for the agent to still effectively learn.

The state space used is $S = (P, V, \Theta, \Omega)$ with

$$
\begin{aligned}
P &= (-0.4, -0.1, 0, 0.1, 0.4), \\
V &= (-0.75, -0.25, 0, 0.25, 0.75), \\
\Theta &= (-0.25, -0.15, -0.05, -0.025, 0.025, 0.05, 0.15, 0.25), \\
\Omega &= (-0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3).
\end{aligned}
$$

The action space used is

$$
A = (-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1).
$$

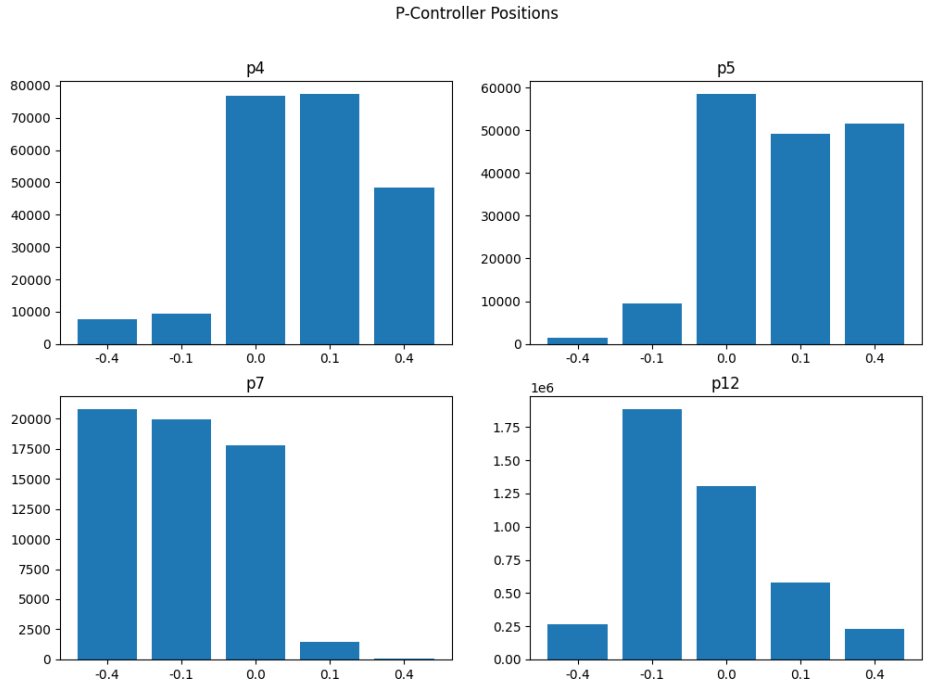

Figure 3.1: Frequency of the position y for different values of the parameter p

The learning rate α and discount factor γ in eq.1.1 were kept constant throughout all simulations, as modifying these parameters for each setup was not part of the optimization process.

The parameter values for each setup were therefore: $\alpha = 0.25$, $\gamma = 0.99$.

The controller, that determines how quick the cart reaches an assigned velocity is a p-controller (pid-controller with $i = 0, d = 0$). The first step for training the agent is therefore to find the correct range for the parameter p in which the agent can operate.

The figures 3.1 to 3.4 show the frequency with which the cart and pole take on certain state values while moving along the y-axis.

It is visible, that cart and pole are both heavily inclined to move in a single direction for values of $p \leq 7$. Fig 3.1 shows the cart to be either only on the positive or only on the negative side along the y-axis. Fig 3.2 shows that the velocity, with which the cart moves along the y-axis is a little more evenly spread, but still with a clear peak in one direction.

The pole is also more inclined to either always fall over to the left or always fall over to the right as depicted in fig 3.3. For values of $p < 7$ the agent also tends to choose the

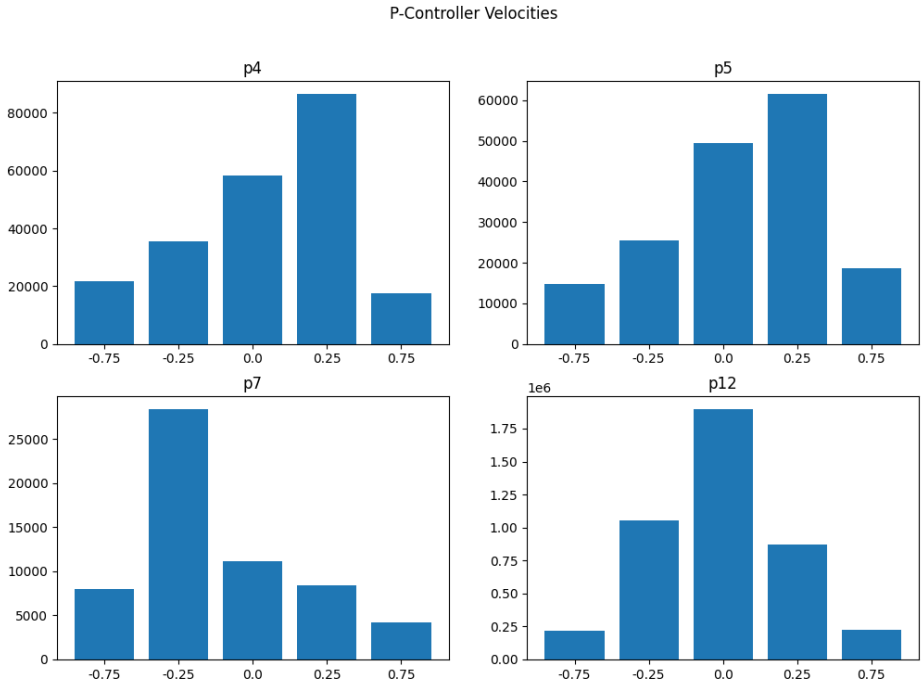

Figure 3.2: Frequency of the velocity v for different values of the parameter p

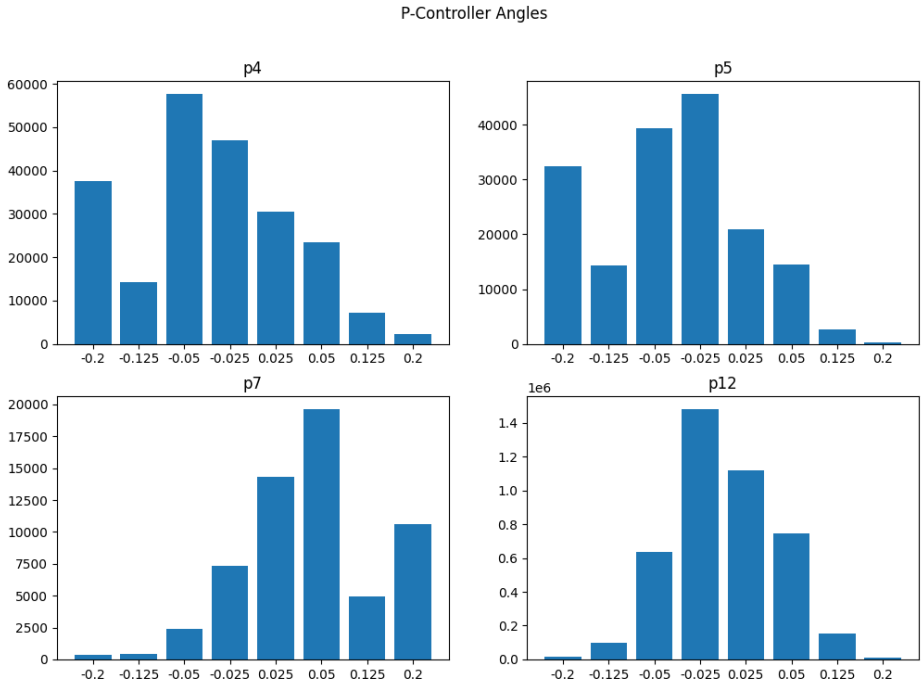

Figure 3.3: Frequency of the angle θ for different values of the parameter p

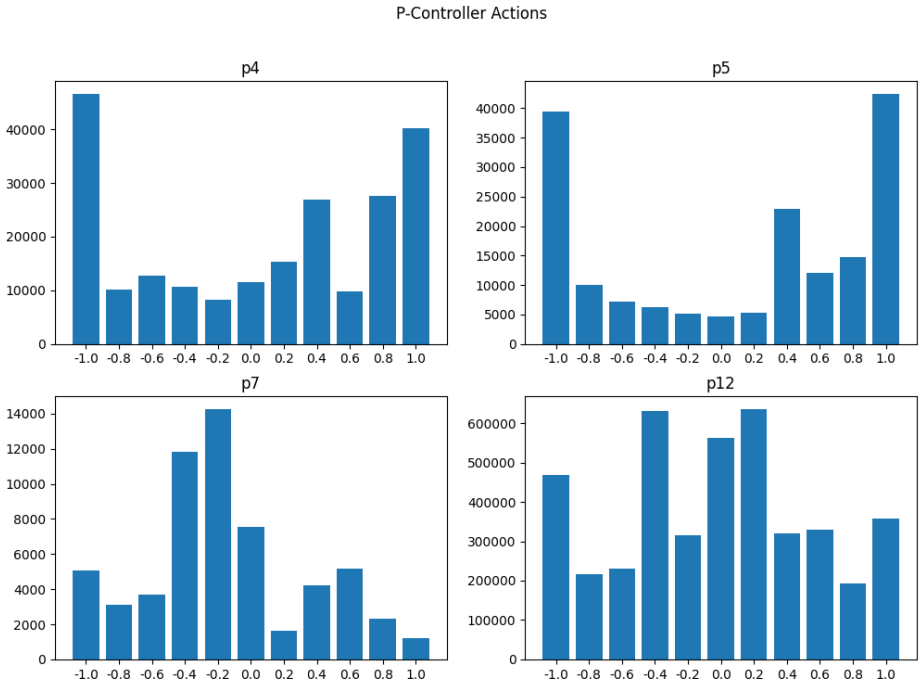

Figure 3.4: Frequency of the action a for different values of the parameter p

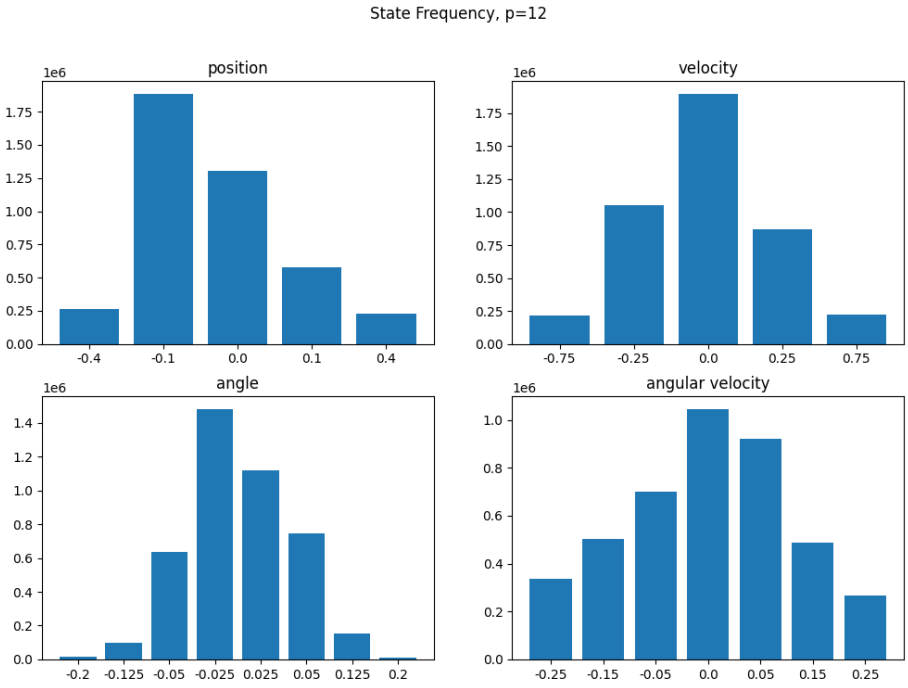

Figure 3.5: Y-axis frequency of state values for parameter $p = 12$

strongest actions significantly more often then the rather moderate ones. For $p \geq 7$ the actions taken become more centered and for $p = 12$ all state values are more centered as shown by fig 3.5

The agent was trained on the given state and action space and a value $p = 12$ for 12 hours until it was able to balance the pole stably. Movement along the x-axis was restricted for both cart and pole during that time.

The parameter p needed to be tuned for movement along the x- and y-axis separately. A first attempt with $p = 60$ was made for x-axis movement because the part of the cart that moves along the x-axis has 10 times the mass of the part moving along the y-axis.

Fig 3.6 and 3.7 show that the state and action values are even better centered than they are for $p = 12$ on the y-axis. The agent is therefore trained for 12 hours again, this time with the y-axis movement restricted and a value of $p = 60$. Once the agent learned to balance the pole separately for both x- and y-axis, the agent was trained a 3rd time for 12 hours with no restriction to the movement.

All in all this means the agent needed 36 hours with the relatively small state and action

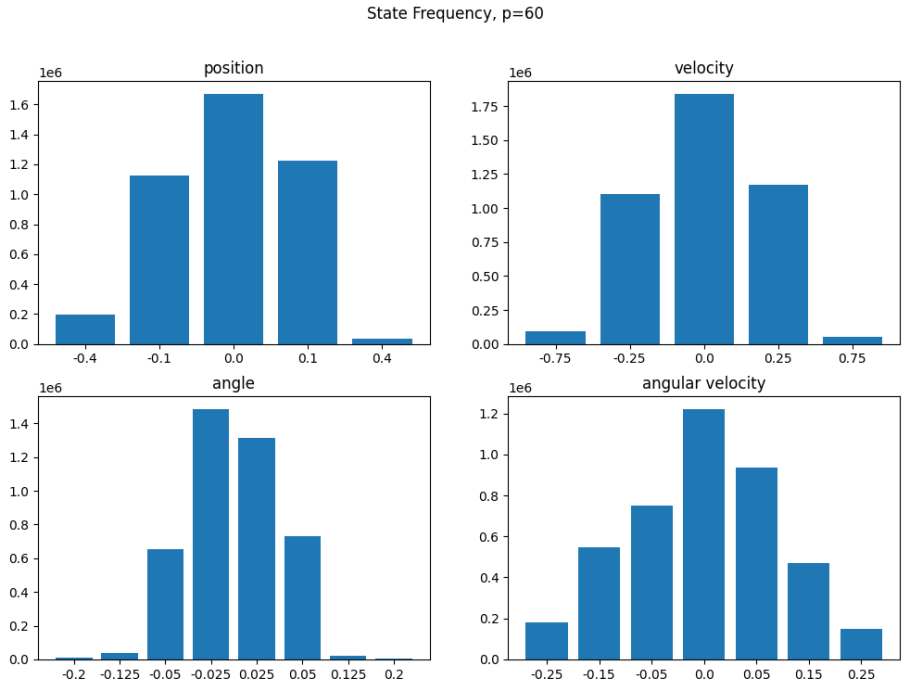

Figure 3.6: X-axis frequency of state values for parameter $p = 60$

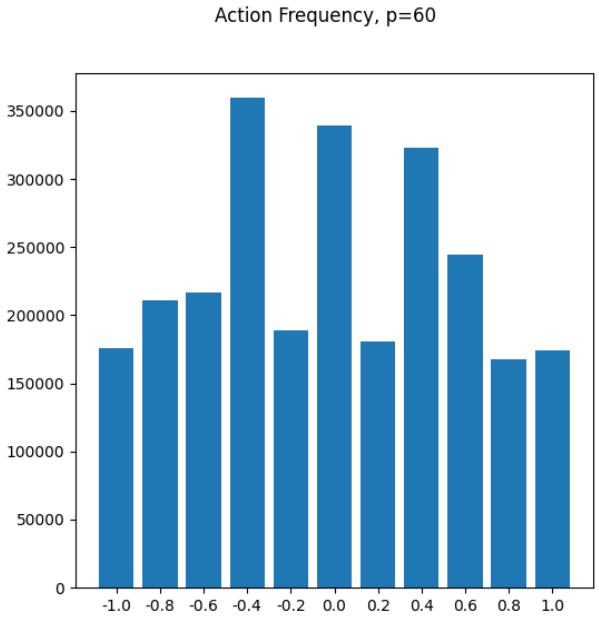

Figure 3.7: X-axis frequency of action values for parameter $p = 60$

space to learn, how to balance the pole stably. An attempt was made to increase the size of the state and action space for higher accuracy and even more resistant balancing using the new state space $S' = (P, V, \Theta, \Omega)$ with

$$
\begin{aligned}
P &= (-0.5, -0.3, -0.1, 0, 0.1, 0.3, 0.5), \\
V &= (-0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75), \\
\Theta &= (-0.25, -0.15, -0.1, -0.05, -0.025, 0, 0.025, 0.05, 0.1, 0.15, 0.25), \\
\Omega &= (-0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3),
\end{aligned}
$$

and the new action space

$$
\begin{aligned}
A = (&-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, \\
&0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1).
\end{aligned}
$$

However even after 72 hours of combined training the agent did not learn to balance the pole on that state and action space and no further attempts were made to improve the former results.

The training up to this point was done using the true orientation of the IMU, which is directly provided along the linear acceleration and angular velocity by the simulator. In reality the orientation is not directly available an needs to be calculated from the other two values using a filter from the package *imu_filter_madgwick*.

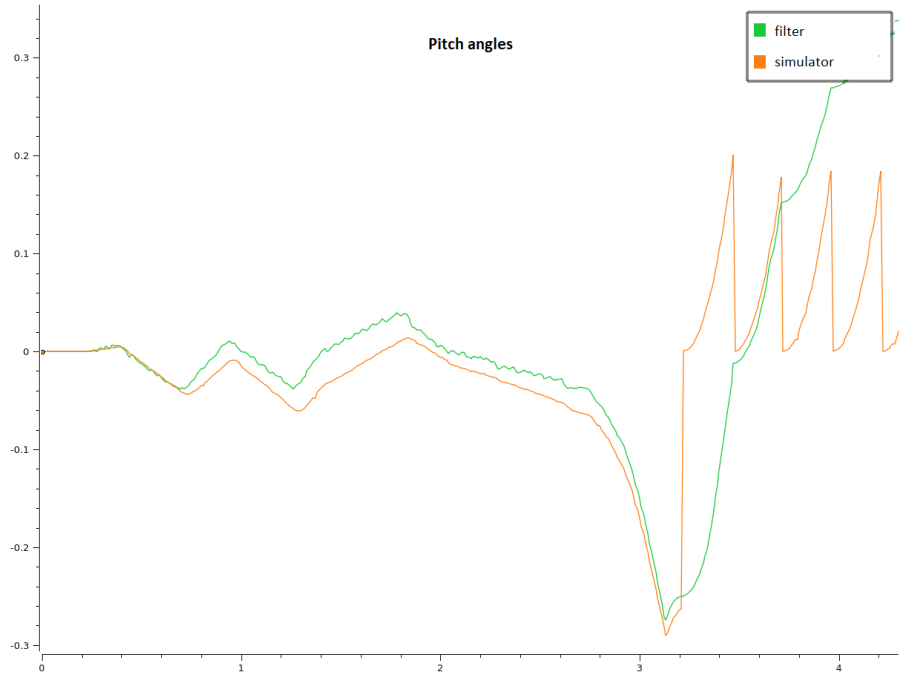

Figure 3.8: Pitch angle calculated by the filter based on IMU data and true angle provided by the simulator

Unfortunately the pole fell within five seconds of starting the simulation when using the orientation calculated by the filter instead of the true one provided by the simulator. Fig 3.8 and 3.9 show the pitch and roll angles from both the filter and simulator. It shows that both pitch and roll angle from the filter start to diverge slowly from their true value, though the general forms of the graphs are quite similar up until the first fail state is reached, at which point the immediate reset causes the filter to lose orientation completely.

There are two steps that can be taken to improve the performance of the filter. The first one is to simply reduce the gain parameter of the filter, which results in it being more accurate but slower with the calculation. The second step is to reduce the parameter p of the p-controller responsible for moving the cart in y-direction.

As shown by fig 3.10 the linear acceleration in y-direction measured by the IMU can get close to $1g$, which might lead to the filter not being able to accurately approximate orientation.

The agent is retrained for the parameters $p_x = 60$ and $p_y = 8$, were the value of p_y was determined to be close to the minimum, that still allows the agent to work in the previous section. The training was successful and finished quickly after only 4 hours by using the

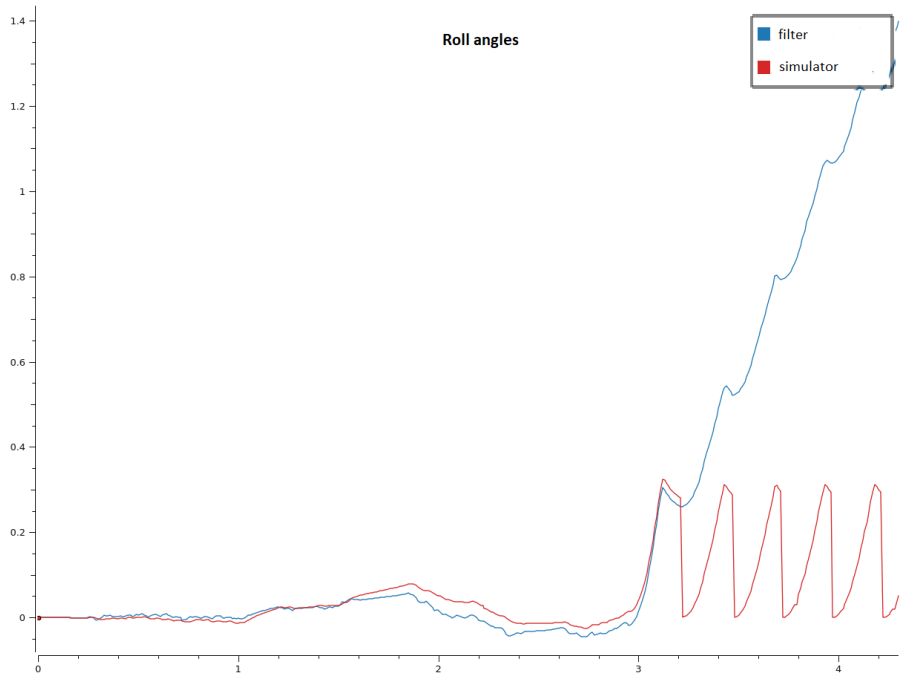

Figure 3.9: Roll angle calculated by the filter based on IMU data and true angle provided by the simulator

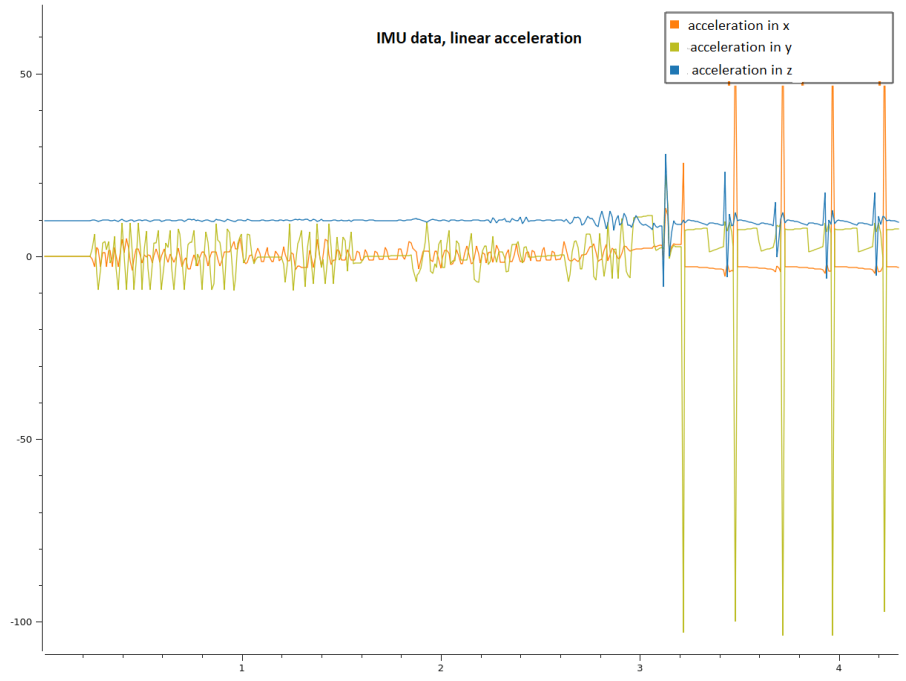

Figure 3.10: Linear accelerations measured by the IMU

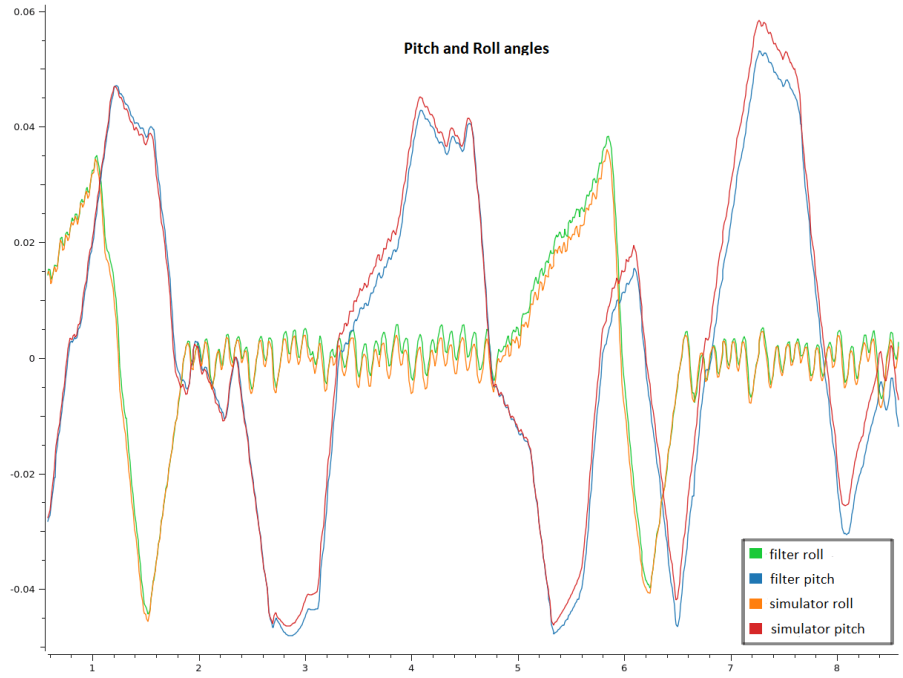

Figure 3.11: Roll and pitch angle calculated by the filter based on IMU data and true angle provided by the simulator

existing Q-table as a starting point. Additionally to reducing the gain of the filter and the p parameter of the controller, a buffer was added to the reset function, which holds the pole in place for a few seconds after it was brought back to the origin due to the system reaching a fail state. This allows the the filter to re-orientate itself after the reset and makes it possible to add further training for the agent using the data provided by the filter should this be needed.

Fig 3.11 shows both the pitch and roll angles provided by the filter and simulator. It can be seen, that the accuracy, with which the filter approximates the orientation, has significantly improved. From fig 3.12 we can see, that the reduction of p_y also had the desired effect and reduced the maximum value the linear acceleration in y-direction can reach. This highly likely had a positive effect on the accuracy of the filter.

After the improvements done to the filters accuracy, the agent is now able to balance the pole significantly longer than before. It is still not as stable as when using the actual orientation provided by the simulator, but the result is good enough as a proof of concept that the 'cartpole' problem can be solved in two dimension based on the data provided by an IMU. This is why no further training was done on the filtered data, even though it could possibly increase the performance of the agent even further.

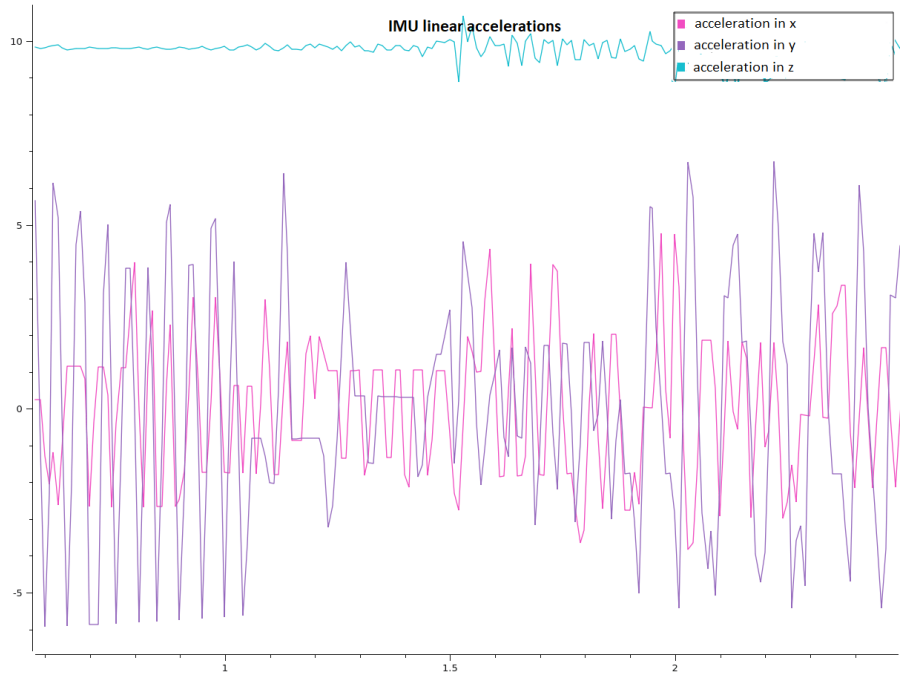

Figure 3.12: Linear accelerations measured by the IMU with reduced y-direction acceleration due to smaller parameter p_y

3.2 Simulation of the Diana-7 robotic arm

The agent pre-trained on the 2-dimensional 'cartpole' is next used to try balancing the pole using a robotic arm of the model Diana-7 inside the simulator. Movement of the pole was restricted to 1-dimension again, allowing only rotation around the x-axis, which means it could only fall over along the y-axis. Similar to the parameter p in the previous section the parameters t, a, s were implemented for this step, with t being a time parameter, a an action parameter and s the number of points calculated for the trajectory.

Since movement of the pole is restricted to 1-dimension, the robotic arm also only needs to moves along one axis, in this case the y-axis. The distance from starting to end point of a trajectory is determined by

$$d = y_a * \frac{t}{108} * a,$$

where y_a is the action value chosen by the agent. Fig 3.13 reveals why the introduction is necessary, as it shows the frequency, with which states are visited, for all parameters set to 1. It can be seen from the figure, that the action chosen by the agent results in barely any movement by the robotic arm and the pole always falling over in the same direction.

However fig 3.14, which compares the distance of the trajectory goal to the actual distance

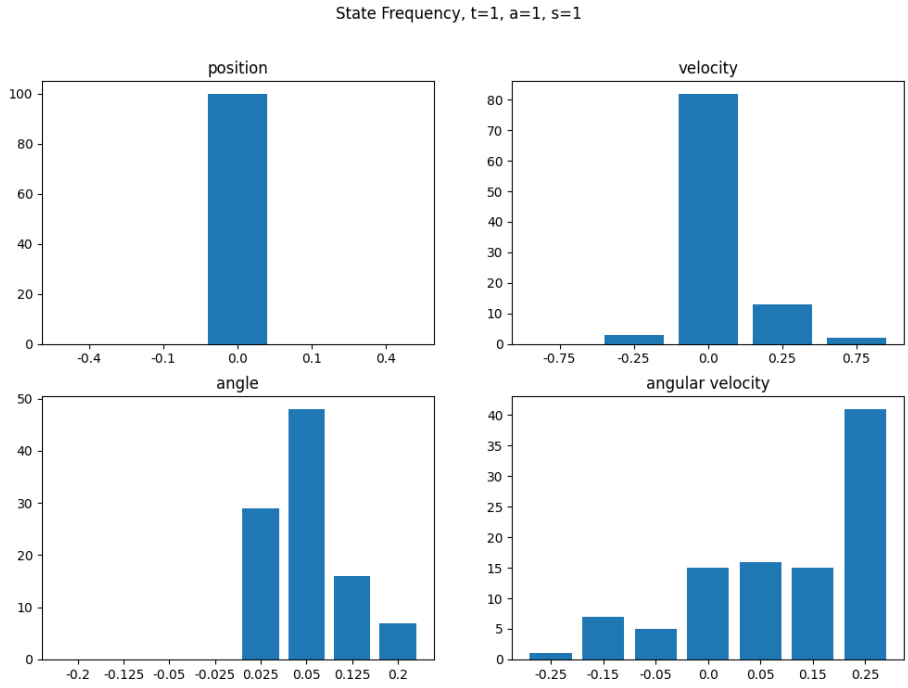

Figure 3.13: State frequency for y-axis movement with no parameter modification

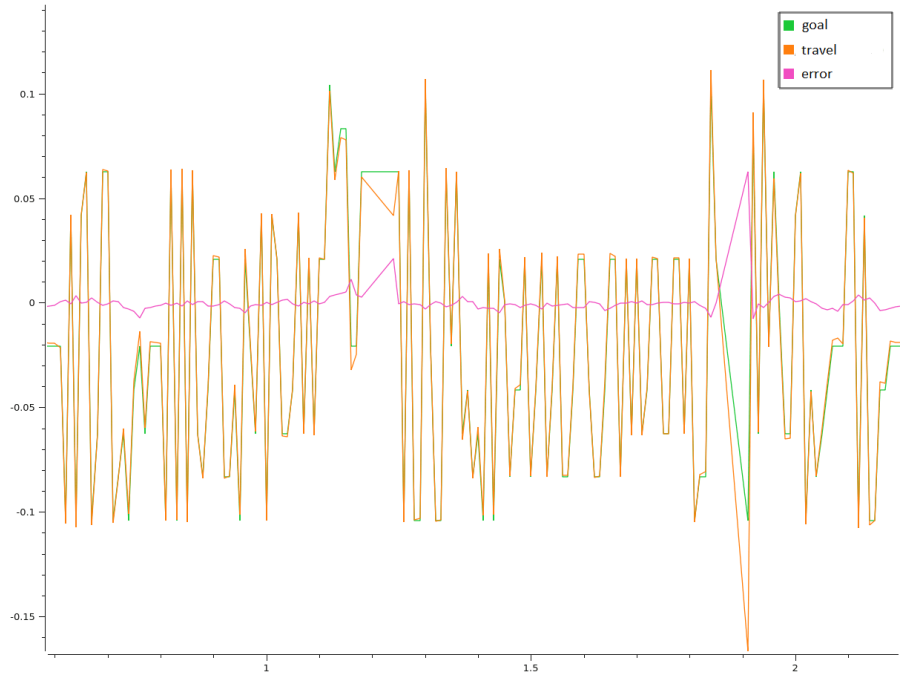

Figure 3.14: Comparison of the trajectory goal and the distance traveled

traveled by the robotic arm indicates that the robot is actually pretty accurate when moving to the ordered position. This means the action chosen by the agent result in a movement that is simply too slow and too small to prevent the pole from falling over.

The action parameter a was therefore introduced to increase the distance traveled by the arm for all actions, also resulting in a higher velocity. The time parameter $t$was introduced in order to make sure the robotic arm is still in movement when it receives the next trajectory hoping it would smooth out the movement. It proved to be largely ineffective however. The steps parameter was introduced because for higher action parameters and farther distance traveled the arm would move in a visible curve instead of a straight line. Additional way points on the trajectory would force the arm closer along a straight line, but not completely and at the cost of a significantly longer calculation time.

Fig 3.15 and 3.16 show the frequency with which states are visited for action parameters $a = 3$ and $a = 5$. The simulation using an action parameter of $a = 5$ was the only one that showed a significant improvement of the agents ability to balance the pole, however even then the pole would usually fall over after 3-5 seconds.

It can be seen from the figures 3.15 and 3.16 that the system is often in a state of high angular velocity, which is a problem because the agent learned to minimize the angular velocity in the previous steps and might still be unfamiliar with it, when it comes to handling a

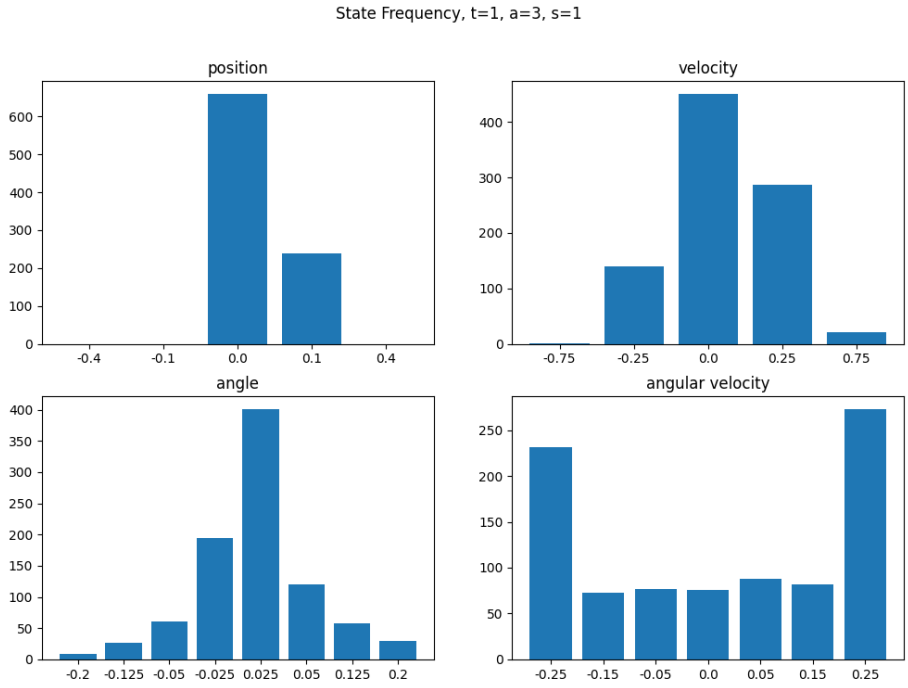

Figure 3.15: State frequency for y-axis movement with action parameter increased to $a = 3$

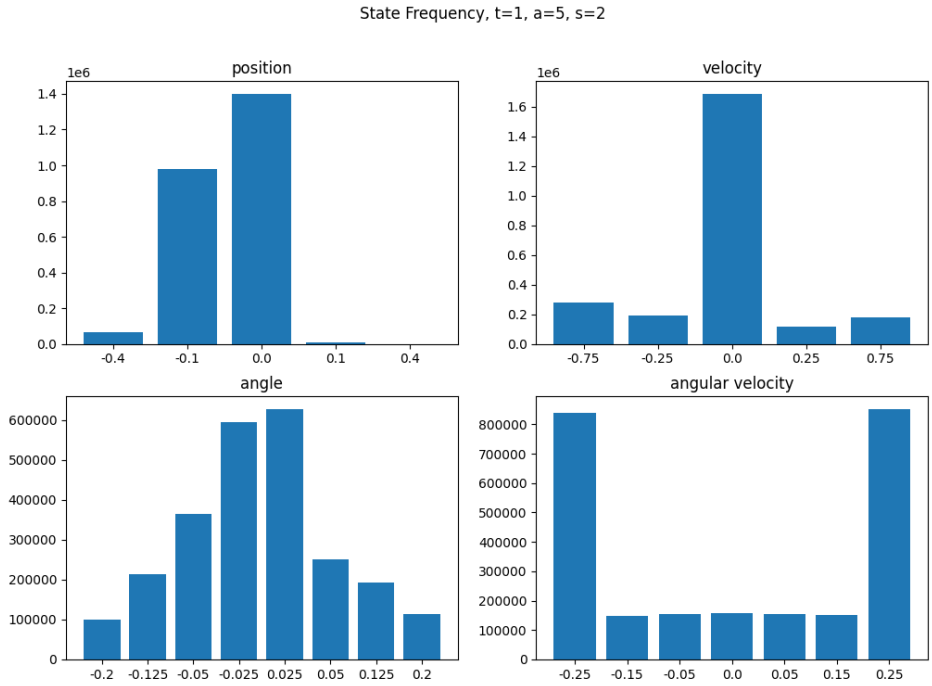

Figure 3.16: State frequency for y-axis movement with action parameter increased to $a = 3$ and double the amount of steps in the trajectory

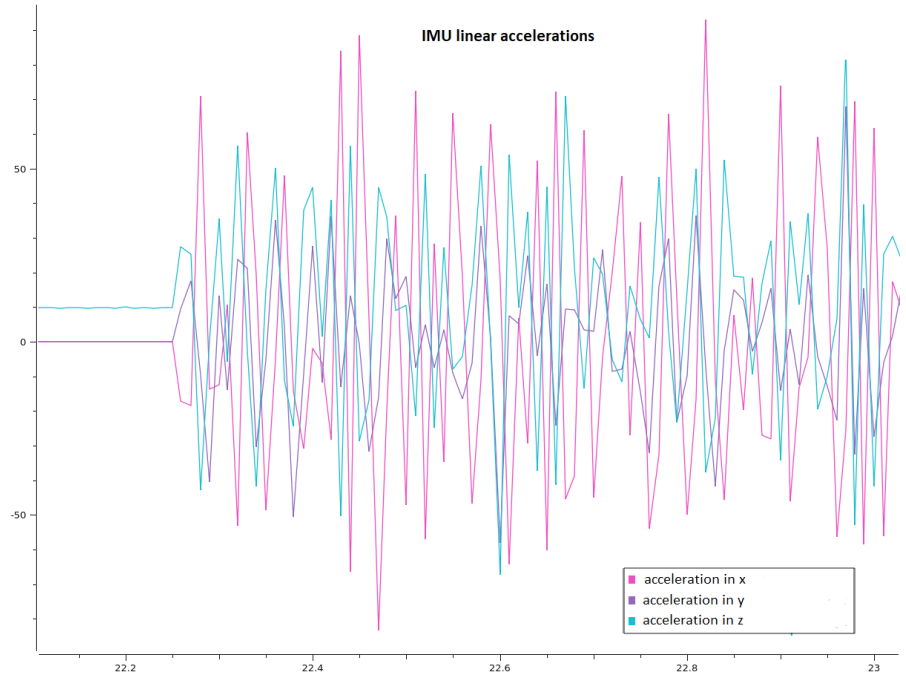

Figure 3.17: Linear accelerations measured by the IMU during simulation with a robotic arm

situation where they are constantly high.

The reason for those high angular velocities is likely that the robotic arm is subject to severe vibrations causing it to constantly inject additional energy into the system. Fig 3.17 reveals how severe those vibrations are. It shows the linear accelerations measured by the IMU and that they reach up to multiple g in all directions making the data extremely noisy.

It remains unclear what exactly causes these vibrations. They might be a side effect of the robotic arm not moving in a completely straight line, which makes it necessary to always explicitly set the x- and z- coordinates of the trajectory goal to be those of the origin position in order to restrict its movement along the x- and z-axis. Movement along the other axes is however just as accurate as it is along the y-axis shown in fig 3.14 but much smaller in amplitude. It is possible that the vibrations occur at the beginning of each step when the command to execute a new trajectory replaces the old one and high acceleration on change in direction causes the arm to vibrate.

In the previous section it could be seen, that filter used to calculate the orientation of the IMU based on its measurement of the linear acceleration and angular velocity, becomes increasingly inaccurate if the measurement is not dominated by the acceleration on the z-axis due to gravity. Considering how noisy the IMU's data as shown in fig 3.17 is, it wouldn't be surprising if the filter is unable to provide any feasible data itself.

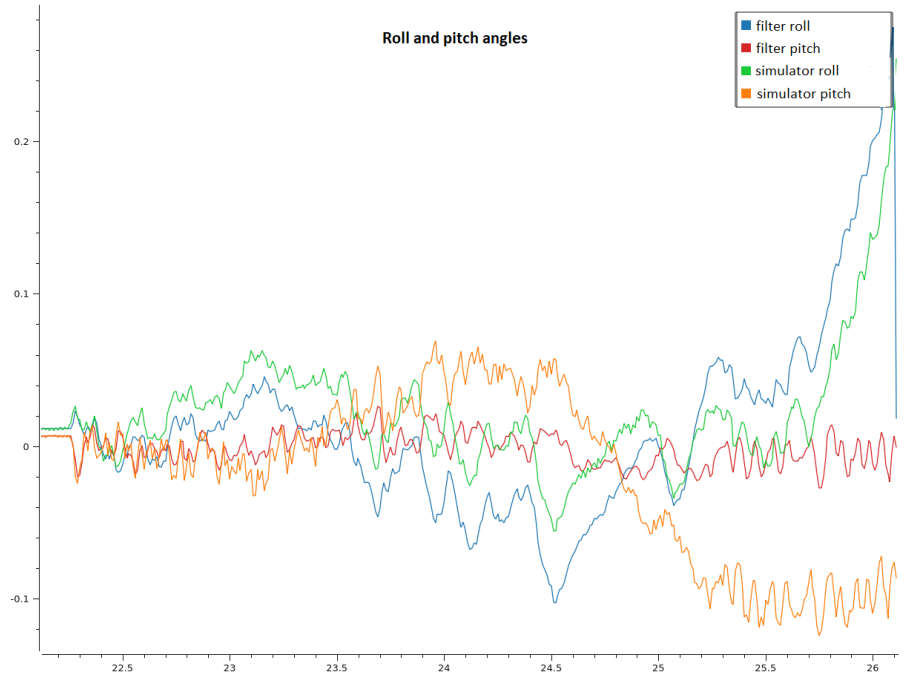

Figure 3.18: Comparison of pitch and roll angles provided by the filter and the simulator during simulation with a robotic arm

Fig 3.18 shows that the filter still does surprisingly well to approximate the angles, seeing how the form of both graphs are still pretty similar. However it also shows a significant divergence of the actual orientation and the calculated one. Reducing the gain parameter further could improve the results, however it caused the filter to publish its result with a time stamp later than the one on the next message of the IMU.

Code efficiency was one of the major problems during the experiment. The IMU publishing with a rate of 108Hz gave only 10ms to handle each callback. The service '/compute_ik' needed at least 2ms to reliably generate a solution for the inverse kinematics and only 2 points could be calculated in order to allow the simulation to run close to full speed.

Unfortunately the agent was never able to learn how to balance the pole for more than 5 seconds, even with movement restricted to 1-dimension, due to the strong vibrations of the arm. Continuing to train the agent using the arm might still bear some results, however with the large divergence of the orientation provided by the filter from the actual one, it is unlikely that the arm would ever be able to balance the pole outside the simulator. The last step, to let an actual robotic arm try to balance the pole, is therefore forfeited

4. Conclusion

Unfortunately the experiment failed, likely because I could not prevent the vibrations during the movement of the robotic arm. It is however possible to conclude three major issues that could be addressed in order to allow a future experiment to succeed. These issues are first, the IMU's reliability to provide the accurate orientation while it is constantly moving around and therefore subject to potentially high linear accelerations. Second is the code efficiency. Third are the vibrations, probably the most import issue to fix as they affect the other two issues as well.

An agent could possibly learn how to balance a pole using an IMU as it was proven when using a cart moving in two dimensions. However for this to work properly the IMU cannot be subject to high linear accelerations, either mandated by the chosen action or as a side effect due to a different problem. This means a very smooth movement is absolutely necessary if an IMU is used to provide information about the pole's orientation. It can also mean that an IMU might not be feasible for balancing the pole if it is too short or too light, even though it might be possible doing so using a different approach. The pole in this experiment wasn't tempered with however, so no final judgment can be passed in this regard.

Code efficiency is very important, since there is relatively little time for the agent to act. Increasing the time frame would result in a loss of reactivity and would require the distance traveled in each step to increase accordingly. This would in turn require the computation of more way points in order to keep the arm from moving in a curve. Since more way points result in a longer computation time, this is not a good solution to increase efficiency anyways, even without regarding the loss in reactivity that comes along with it. A possible solution would be to decouple the determination of the action and the implementation of it by publishing the action to a topic and have another node implement it. This way the node making the callback on a message from the IMU would be free to react to the next one even if the chosen action has not resulted in any movement yet. However this might also cause issues with reactivity as the agent would be rewarded for an action which had not been executed yet and with its effect still unknown.

Smoothing out the movement would probably have the greatest effect. It is possible that the vibrations not only affect the IMU but also dominate the behavior of the pole. This would mean that slower and smaller movement is sufficient to balance the pole if the vibrations don't have to be compensated for. A possible solution would be to switch from 'MoveIt' to

a framework that allows it to just set a velocity and direction for the robot to follow along, or implement it themselves. However this would be a huge project on itself, and might not even help if the vibrations are a result of the rapid changing of directions after each message from the IMU.

Bibliography

[1] Christopher G. Atkeson, Stefan Schaal. Learning Tasks From A Single Demonstration. 1997 IEEE, International Conference on Robotics and Automation. April 1997.

[2] Richard S. Sutton, Andrew G. Barto. Reinforcement Learning: An Introduction (2nd Edition). Cambridge, MA: MIT Press; 1998.

[3] Michel Tokic, Günther Palm, (2011), "Value-Difference Based Exploration: Adaptive Control Between Epsilon-Greedy and Softmax"

[4] Dimitri P. Bertsekas, Dynamic Programming and Optimal Control 3rd Edition, Volume II

[5] Watkins, C.J.C.H. (1989). Learning from delayed rewards. PhD Thesis, University of Cambridge, England.

[6] Swagat Kumar, Balancing a CartPole System with Reinforcement Learning, arXiv:2006.04938

[7] http://wiki.ros.org/ROS/Introduction

[8] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber

[9] http://wiki.ros.org/Distributions

[10] http://wiki.ros.org/noetic

[11] http://wiki.ros.org/rosserial

[12] http://wiki.ros.org/imu_filter_madgwick

[13] https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/

[14] Ioan A. Sucan and Sachin Chitta, "MoveIt", [Online] Available at moveit.ros.org.

[15] https://docs.arduino.cc/hardware/nano-33-iot

[16] https://www.arduino.cc/reference/en/libraries/arduino_lsm6ds3/

[17] https://docs.arduino.cc/tutorials/nano-33-iot/imu-accelerometer, Image modified based on data read.

[18] https://content.arduino.cc/assets/mkr-microchip_samd21_family_full_datasheet-ds40001882d.pdf

[19] https://learn.microsoft.com/en-us/windows/wsl/about

[20] https://learn.microsoft.com/en-us/windows/wsl/connect-usb

[21] R. Sutton, A. Barto. Reinforcement Learning, Introduction

[22] R. Sutton, A. Barto. Reinforcement Learning, 1.1

[23] R. Sutton, A. Barto. Reinforcement Learning, 1.3

[24] R. Sutton, A. Barto. Reinforcement Learning, 2.1, 2.2

[25] R. Sutton, A. Barto. Reinforcement Learning, 3.1

[26] R. Sutton, A. Barto. Reinforcement Learning, 3.6

[27] R. Sutton, A. Barto. Reinforcement Learning, 3.5

[28] R. Sutton, A. Barto. Reinforcement Learning, 6.5

[29] R. Sutton, A. Barto. Reinforcement Learning, 3.5