Student Research Project

Development of a movement system for a six-legged robot based on Machine Learning

for the

Bachelor of Science

from the Course of Studies Computational Data Science at the

Cooperative State University Baden-Württemberg Stuttgart

by

Makss Golubs

on

10 June 2022

Student ID	1871161
Course	INF19A
Company	IBM Deutschland GmbH
Location	IBM-Allee 1, 71139 Ehningen
Supervisor	Prof. Dr.-Ing. Dipl. Inf. Falko Kötter

Table of Contents

Declaration by the author

The author solemnly declares that this report is entirely the product of their scholarly work unless otherwise indicated.
The author has indicated thoughts adopted directly or indirectly from other sources in the document's appropriate places. No part of this project report has been submitted before at this or any other university or institution.
The author has not published this project report in the past. The printed version is equivalent to the submitted electronic one. The author is aware that a false declaration will entail legal consequences.

Münster, 10 June 2022

Makss Golubs

Abstract

This student research project combines open-source *machine learning* tools with affordable *robotics* hardware to create a usable movement system for a six-legged robot. Linking these two technologies includes designing a *simulated 3D model* and a *communication interface* that can handle the simulation data in the real world.

Machine learning helps to optimize and streamline processes. It is a central tool in modern analytics, driving statistical research. One form, *reinforcement learning*, harnesses evolutionary learning to create algorithms based on *repetition and constant improvement*, making it more cost-efficient than supervised learning and suitable for many real-world problems.

On the other hand, robotics has made astonishing advancements over the last decades. For example, specialized robots do the jobs of former assembly line workers, or logistics personnel, while *legged robots* can do on-premise operational work in any environment designed around humans.

Joining the possibilities of two modern technologies and combining them into one workflow is the goal this project aims to achieve.

Table of abbreviations

3D	Three-Dimensional
API	Application Programming Interface
ASTERISK	Adeept Six-legged Terrestrial Robot Kit
DC	Direct Current
DHBW	Duale Hochschule Baden-Württemberg
GPU	Graphics Processing Unit
ICRAE	International Conference on Robotics and Automation Engineering
IDE	Integrated Development Environment
LED	Light-Emitting Diode
LPS	Laboratory Power Supply
ML	Machine Learning
NASA	National Aeronautics and Space Administration
PWM	Pulse-Width Modulation
RGB	Red Green Blue
RL	Reinforcement Learning
ROS	Robot Operating System
YAML	YAML Ain’t Markup Language

1 Introduction

Machine learning and robotics are the two main topics this research project aims to combine. With the help of publically available tools, *I* (referring to the author) will explore the whole development process – from the *conceptualization* to the *implementation* and *training* of the machine learning model.
This chapter includes an introduction to the relevant technology to ease the reader into the topic and a brief explanation of reinforcement learning. Additionally, I want to differentiate this project from the most closely related existing work.

1.1 Robotics & bionics

Humans should always remain the center of attention in an automated world. Researchers need to find out how to make the environment safer and more comfortable for humans, including handing dangerous tasks to *robots*. The Encyclopedia Britannica describes robots as follows.

> *„robot, any automatically operated machine that replaces human effort, though it may not resemble human beings in appearance or perform functions in a humanlike manner. By extension, robotics is the engineering discipline dealing with robot design, construction, and operation.“ [1]*

In daily business, specialized robots are used more and more in logistics and production, e.g., robotic arms constructing cars. Robots can take over work that is too risky, costly, or boring for humans. They can also collect video data from hostile and life-threatening places, e.g., ***NASA*** (National Aeronautics and Space Administration) uses their exploration rovers to discover the planetary surface of Mars [2]. Additionally, these autonomous robots can make decisions based on observations when telecommunication is unreliable.

Figure 1 – Boston Dynamics’ robot Spot with a 360° camera [3].

Boston Dynamics builds the commercially available robots depicted in the figure above to automate the progress documentation of construction sites [3]. In addition, these robots can be equipped with various sensors to create ***3D*** (Three-Dimensional) maps in real-time [4], making their surroundings more accessible to humans.

Robotics is a field where *hardware meets embedded software*. This *hardware-software gap* leads to an important question: how to program a robot to move a certain way? The answer is a *robotic simulation*, which this research project will explore. However, what movements should a humanoid robot do? What about an insectoid one? The following paragraphs examine a topic combining nature and technology to answer these questions.

Bionics is the scientific field of mimicking nature in artificial systems, primarily animals and plants [5]. Learning from nature is essential for technology. *Survival of the fittest*, for example, leads to very effective biological algorithms for many tasks. Since most living beings on earth are the product of over two billion years of evolution, if someone were to design a behavioral algorithm for a robot, the first look should go toward living animals [6].

Figure 2 — Droplets on a lotus leaf (left) and a hydrophobic material (right) [7].

One can find bionic systems elsewhere too. For example, jackets and umbrellas use the *Lotus Effect* seen in the figure above, making water droplets roll off their surface, as seen in the figure above. *Echolocation*, used by bats or dolphins, inspired radar, sonar, and medical ultrasound. Last but not least, the (human) brain and its neurons lead to modern machine learning using *neural networks* – and once scientists lay the basis for *simulating a brain*, they can imitate the learning of complex behaviors found in living creatures.

This project will take advantage of two bionic approaches. First, this project will discuss how the six-legged robot section *2.2 Robot* introduces should move based on the *movement patterns of six-legged insects* [6]. Second, the developed model will use an *artificial brain* to test whether the result is identical or even better. While “artificial brain” sounds like science-fiction, this research project will rely on current reinforcement learning technology.

1.2 Machine learning

Applications of ***ML*** (Machine Learning) include predicting a particular behavior, e.g., the movement of stock markets [8] or shopping patterns in online shops [9]. In addition, ML makes it possible to teach computers an intuition for real-world problems. As a result, ML proves its usefulness in science, economics, health services, and logistics.

One recent example is the victory of *DeepMind's AlphaGo* over a three-time European Go Champion in 2015 [10]. What made AlphaGo unique is that instead of using heuristic principles and probability calculations, DeepMind shaped reinforcement learning – an approach allowing computers to learn independently from any human influence by *trial-and-error*. ***RL*** (Reinforcement Learning) leads to unusual solutions that some perceive as *creative*. For example, AlphaGo managed to find strategies that Go players did not develop in all 3000 years of the game's existence [11].

RL includes a simple feedback loop. Chapter *4 Reinforcement learning* will describe an implementation of RL in the ML toolkit – but how does RL work in general?

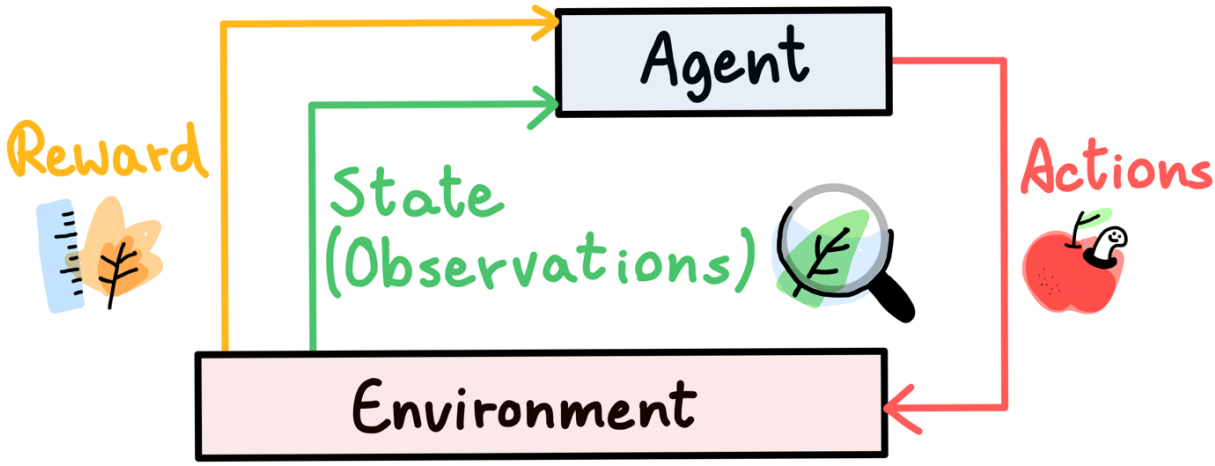

Figure 3 – Reinforcement learning feedback loop.

During the loop seen in the figure above, the *agent* observes the *state*, makes a *decision*, and executes an *action*. Actions change the agent's position in the *environment*. The new state is then fed back to the agent, additionally to the reward, which bases on the *reward function*. The reward teaches the agent to distinguish good from bad behavior. RL is *simulated survival of the fittest*; the evolutionary mechanism creates generations of agents that compete against each other – the system rewards good behavior while bad performers drop out. The best algorithms then get to mutate and reproduce. Ultimately, the final result depends heavily on the environmental model, in which the agent learns to optimize its actions.

This research project wants to create a movement algorithm based on RL that reaches the efficiency of living beings.

1.3 Related work

In summary, this project will attempt to solve four partial problems in the following chapters:

- Designing a 3D model of the physical, six-legged robot,
- using an ML framework to train a walking model via reinforcement learning,
- programming a command ***API*** (Application Programming Interface) to send the ML model result to the real robot,
- and test the movement quality between the generated model and the handmade one.

Scientists at the ***ICRAE*** (International Conference on Robotics and Automation Engineering) used RL to train a six-legged robot to navigate challenging terrain and walk around obstacles. Their work comes closest to what this research project aims to achieve, focusing on transferring the movement from the simulation to reality. Also interesting is their approach to boosting training speeds by using curriculum learning – first training the simulation in an easy terrain that gets procedurally more challenging to navigate [12].

The *focus on cross-topic transfer* differentiates this research project from the mentioned article. This research will explain ML, the used robotics hardware, and the development of an interface – instead of focusing on a single topic. Comparing this work to the heads behind the ICRAE article, this project will only go briefly over the basic ML methodology without digging too deep through the specifics of the neural network, relying on existing technologies for RL. Since robotics, especially legged robots, have made considerable advancements in recent years, developing new movement systems is the topic of many papers [13, 14]. This research project will not dive too deep into developing an all-purpose movement system for robots. However, this project will explore the process of *creating a simple, high-level movement API* to test the results of ML-generated movements in the real world.

Researchers at Google Brain and the University of California even transferred some parts of the simulated learning into the real world, including a feedback loop that measures the behavior of a robot using motion capture and compares the measurements with the simulation [15]. This loop is quite impressive; however, this project will *strictly separate the simulation from the real world* during this project. Therefore, this work will treat the beforementioned partial problems from the beginning of this section separately in each corresponding chapter – with their requirements, design process, and results.

So, without further ado, the next chapter will describe the necessary prerequisites.

2 Groundwork

This chapter shall discuss the project's requirements before the development can begin. First, the chapter defines the development's subtasks in the *timeline*, then describes the *robot* and its hardware components. Next, the *simulation environment* explains the advantages and nuances of the *Unity game engine* and the *Unity ML toolkit*. Furthermore, this chapter explores the *Arduino framework* for operating the micro-controller and summarizes the project by proposing an *architecture* including the data flow.

2.1 Timeline

What follows is the scope of the project. The previous chapter has established the differences between this research and the related work. With the partial problems in mind, the timeline for the separate steps looks as follows.

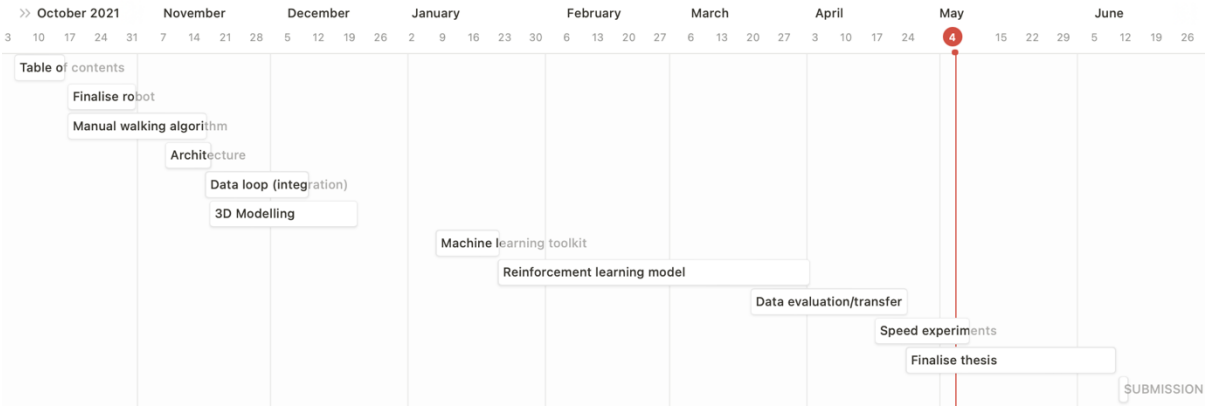

Figure 4 – Gantt chart of the whole project's timeline.

Initially, the project aims to set up the components and ensure they all work as intended. So, the tasks are to prepare the robot, send it a handcrafted algorithm, create the 3D model, and set up the ML toolkit. Following this spadework, the RL will take up the most significant chunk of the project because of the long training time between the iterations. Lastly, this project will evaluate the resulting algorithm, transfer it to the robot, and compare it to the handcrafted algorithm. The timeframe between the project's kickoff (03.10.21) and this document's submission (10.06.22) spans eight months.

The following section describes the technology this project will use with the concept out of the way.

2.2 Robot

The robot in this research project is an *Adeept Hexapod* Spider Robot kit [16]. The detailed instruction manual is under appendix *documents/instructions*. The robot consists of the following parts and subsystems:

- 13 *servo motors*, two for each leg and one for the robot's head,
- an ***LED*** (Light-Emitting Diodes) module with 6 ***RGB*** (Red Green Blue) LEDs,
- an ultrasonic *distance sensor*,
- a *WiFi module* for remote control,
- a six-axis motion tracking *gyroscope and accelerometer*,
- an Arduino-based *AdeeptPixie driver board*, connecting all the components, and finally,
- polymer, laser-cut body parts, and frame.

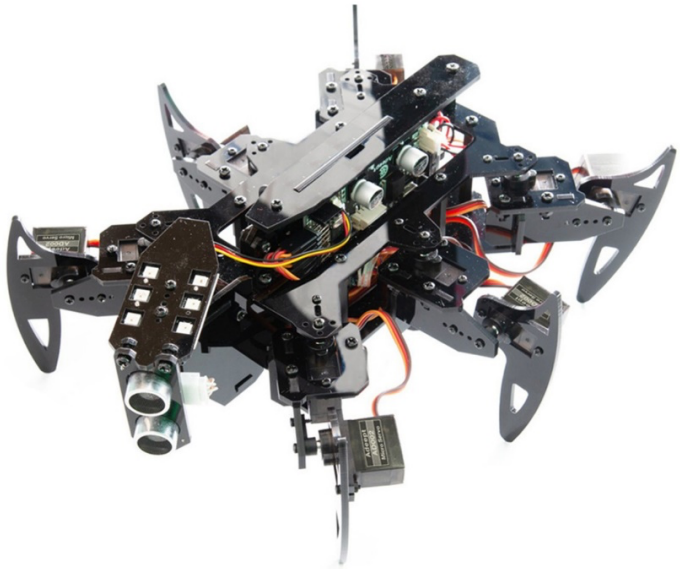

Figure 5 – Image of the Adeept hexapod robot from the manufacturers' website [16].

This document will refer to the project's robot as ***ASTERISK*** *(*Adeept Six-legged Terrestrial Robot Kit). Most of these components will not be relevant for this project. Only the motors, the polymer frame, and the driver board play a role in making the robots walk. Nevertheless, it is interesting to think of more possible applications of the different parts. For example, instead of compiling the ML results into discrete commands stored on the micro-controller, one could imagine controlling the robot directly over WiFi, allowing fast testing of multiple solutions without reprogramming the controller. In addition, the gyroscope could help analyze if the simulation corresponds to the robot's movements, like the work from the University of California [15]. With time in mind, these ideas are not feasible – the machine learning will take approximately five months to complete, which means there is little time left to optimize the machinery. Therefore, the project shall remain focused on the main topic of generating a movement algorithm using machine learning.

ASTERISK holds 12 servo motors that control its six legs. A servo motor rotates parts of a machine with high precision [17]. In addition, Adeept provides the robot kit with its own ***PWM*** (Pulse-Width Modulation) 9g hobby servo motors.

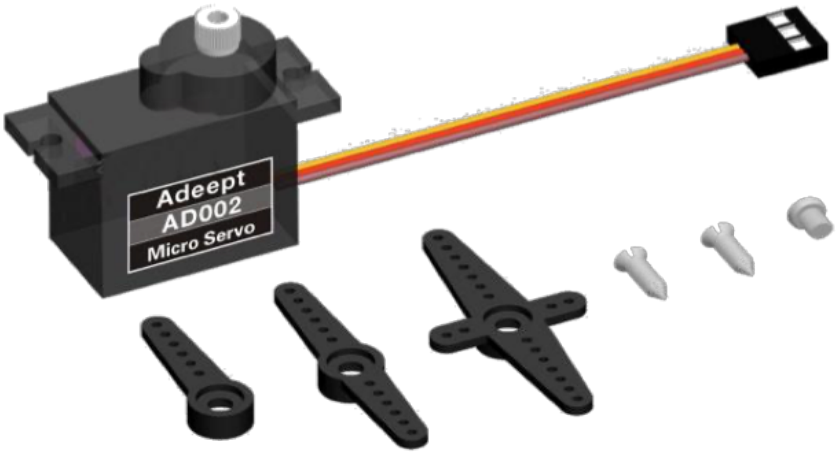

Figure 6 – A rendered image of the Adeept AD002 Micro Servo motor [18].

Adeept's servos, depicted above, work using PWM, which means they receive a 50 Hertz pulse and depending on the duration, the servo rotates to a given angle. Programatically and from within the Arduino framework, the angle is sent to the servo using the `write()` function, and the controller then converts it to the required pulse duration, e.g., 45° equals a 1 ms pulse, or 90° equals a 1.5 ms pulse [18].

Looking at the servos' technical limitations, as discussed in section *5.1 Robot legs API*, the `read()` function that returns the servo's current position is unreliable. It is unclear why, but a quick test proves that the servos in ASTERISK do not return the correct values when the system reads the angles – writing an angle of 90° and subsequently reading the same servo's angle returns the value with a margin or error of ±20°. The command interface should therefore save the written value into a variable.

Another issue arose when testing the servos: one of the set's provisioned servo motors overheated and broke. Luckily, Adeept added a spare servo to the kit, making ordering another one from their website unnecessary. Nevertheless, this incident shows that these *moving parts are prone to break*, so the user must treat them with care. So, the commands sent to the robot should not be too demanding or too frequent. Section *4.3 Implementation* will take up this topic again when discussing the movement value sampling.

It is worth noting that servo motors do not simulate a continuous, easing movement – once they receive the command angle, they immediately jump to the requested angle. Therefore, section *3.2 Physics* will respect this when designing the 3D model.

With the hardware to simulate discussed, the following section highlights feasible simulation frameworks.

2.3 Simulation

There are many freely available, open-source frameworks for ML and RL [19], and some of them provide implementations for 3D simulation. One of them is the *Unity Machine Learning Agents Toolkit* (referred to as ML toolkit from now on), based on the Unity game engine. *Game engines* are tools for constructing video games but are also suitable for *simulating real-world physics* [20]. Taken from the official documentation:

> *"The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents." [21]*

Unity is an excellent tool for beginners and provides many resources for getting started – and as a result, developers use it in various industries to create interactive experiences. For example, Unity uses *Nvidia PhysX* for its 3D physics simulation [22], a cutting-edge physics engine. In addition, the ML toolkit uses *PyTorch*, an open-source ML framework.
Many well-developed technologies are at play here, making Unity a good choice for this task. Most importantly, Unity combined with the ML toolkit allows the power of Unity's 3D simulation to train a hexapod model to walk directly inside the game environment – it *combines the simulation and the RL* into a single platform.
The publicly available GitHub repository of the ML toolkit has 15 examples to try out and use as a reference for this work's ML agents. One of those examples, similar to what the project aims to achieve, is the Crawler. It is a 3D model of a quadruped which learns to walk by collecting as many green cubes as possible in the given timeframe [23].

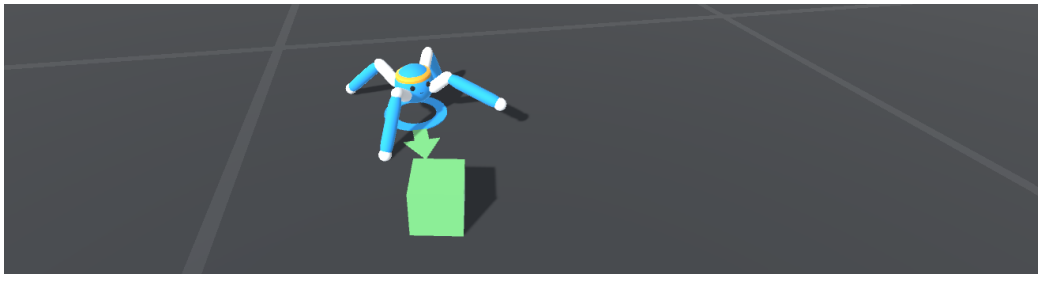

Figure 7 – Machine Learning Agents example "Crawler".

The ML toolkit requires a ***GPU*** (Graphics Processing Unit) because the 3D physics simulation affects the machine's processing power. Also, ML requires many matrix multiplications, especially suited for GPUs. Therefore, the ***DHBW*** (Duale Hochschule Baden-Württemerg) provides this project a *ThinkPad P52* with an *NVIDIA Quadro P1000* integrated GPU [24].

Since the research project will not produce a monetized product, Unity's personal license is sufficient. A developer needs to download *UnityHub* [25] and the latest editor version (*2021.2.11f* during writing) and run it on the provided laptop.

2.4 Programming

While developers can script in Unity using *C#*, most embedded systems use lower-level programming languages to spare themselves from performance-intensive runtime environments [26]. For example, the hexapod's microcontroller, the AdeeptPixie driver-board, stems from the *Arduino* community's open-source software, relying on *C++*.

> *"Arduino is an open-source electronics platform based on easy-to-use hardware and software. It is intended for anyone doing interactive projects." [27]*

This project requires the Arduino framework to *communicate with the driver board*. The framework takes the code, compiles it to machine code according to the microcontroller's architecture, and reprograms the driver-board over its serial port. Additionally, *open-source* allows developers to help themselves with various existing, high-quality C++ libraries to manipulate, control, and debug their hardware. Arduino offers an ***IDE*** (Integrated Development Environment) for users to download these libraries, compile the code, and upload it to the micro-controller.

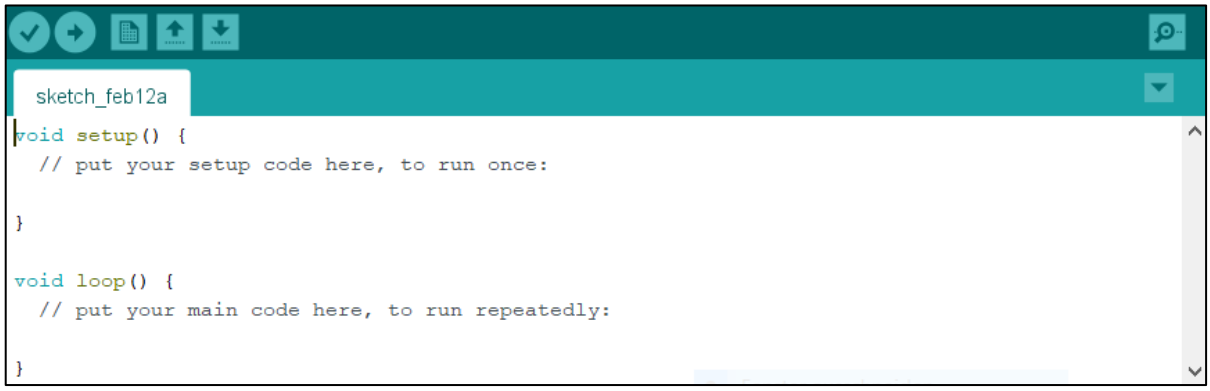

Figure 8 – Screenshot from the Arduino IDE.

On the other hand, the Arduino IDE is a bare-bones text editor. For an inexperienced C++ programmer, it is difficult to exploit the full potential of the programming language without intelligent coding assistance. Arduino's IDE is lightweight and easy to set up but does not supply this project with the tools to efficiently use *object-oriented programming*. It lacks a view of the file hierarchy and makes it challenging to include objects from other folders. It is decent enough for concepts but not for big projects.
However, there is a powerful alternative for Arduino development; *JetBrains' CLion* IDE.

The setup for CLion is more complex than Arduino's IDE, but the development speed increases using *code generation*, *intelligent refactoring*, and the *visual debugger*. To start programming the hexapod's Arduino-based micro-controller, CLion requires an installation of *PlatformIO*.

> *"PlatformIO is a cross-platform, cross-architecture, multiple frameworks, professional tool for embedded systems engineers and for software developers who write applications for embedded products." [28]*

PlatformIO standardizes the interface between the coding environment and the micro-controller, making it easier to compile complex projects and upload them to the hardware. Of course, the project does not require CLion to interact with the Arduino-based devices, but it makes the development easier.

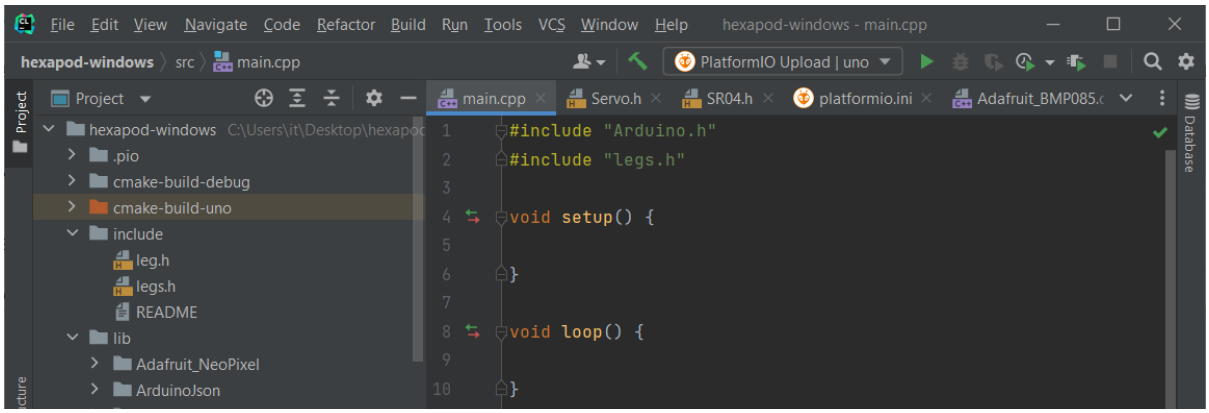

Figure 9 – Screenshot from JetBrains' CLion IDE.

Unfortunately, as chapter *5 Command interface* will discuss, the code provided by Adeept is not of adequate quality, which means the project includes the development of a movement system and API to control the robot.
To summarize all the tasks discussed previously, the following section describes the conceptual side of the project.

2.5 Architecture

Bringing data from the simulation to the robot is a challenge best described as a *gap between the simulated model and the physical implementation*. This gap originates from the subtle differences between a simulated environment and the real world, e.g., the legs' friction on the ground, the limitations/speed of a servo, or a faulty physics simulation.
Recreating the natural world perfectly with all its nuances is not trivial. Therefore, this project must address the gap carefully to produce the best test conditions. With that in mind, the following figure depicts a rough idea of how the data flow shall work.

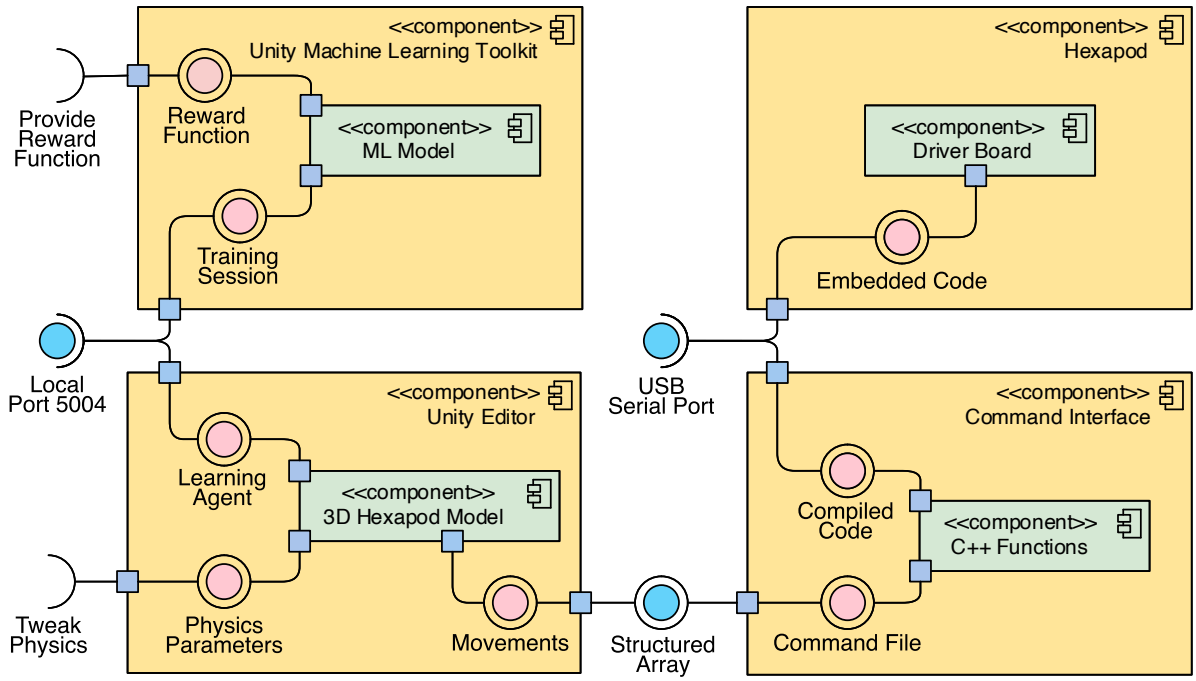

Figure 10 – The proposed data flow architecture for the research project.

The four `<<component>>` boxes represent the four partial problems of this project. First, using the ML toolkit, the user needs to provide a *reward function* – an essential topic in chapter *4 Reinforcement learning*. Then, the connected *learning agent* will generate movements based on the rewards. Finally, a developer needs to tweak the *physics parameters* to make the beforementioned implementation gap smaller to resemble real-world conditions.

Once the training finishes, the system extracts the model-generated movements from the simulation, transforms them into a *structured array* within a *command file*, and loads them into the C++ API. The API then compiles the movements within the IDE and loads them into the *driver board* using its *serial port*, making the robot execute them in reality.

The idea is to fully detach the machine learning (taking place inside the simulation) from the physical robot – only once the simulated model moves in a satisfactory way is the data transferred.

Chapter *3 Designing a 3D model* describes making a physics-based model in Unity using simple game objects. After that, the bulk of the ML concept, training, and results will be the topic of chapter *4 Reinforcement learning*. Then, chapter *5 Command interface*, aims to create the necessary C++ functions to control the robot with a hard-coded sample command file. Finally, the algorithm will be uploaded and tested in chapter *6 Real-world applications* form a verdict. This structure treats each partial problem separately, and if complications should occur during any step, the other parts of the project remain intact.

3 Designing a 3D model

The central question of this chapter will be how to construct a 3D model of a physical robot in Unity. Robotic simulation is not a trivial exercise, so the following sections will discuss how to create a simple solution to achieve the architectural requirements.

3.1 GameObjects

ASTERISK is a six-legged robot with two servo motors for each leg – *Figure 11* shows how the 3D model representing the real robot looks. The model consists only of `GameObjects`, Unity's primary form of 3D shapes.

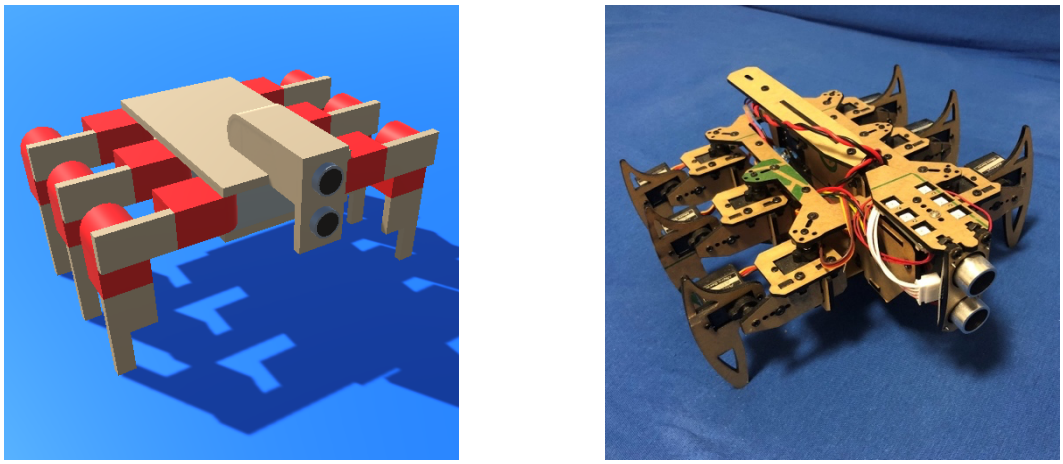

Figure 11 – Comparison between the 3D model of ASTERISK and the real-world robot.

The 3D model on the left side of *Figure 11* does not need to resemble the robot 1:1 in favor of *abstraction*. The most crucial aspect is the 3D model's similarity with the real robot. These similarities include the leg distance, the number of joints, and their degrees of freedom. The 3D model has all its *joints colored red* to distinguish them from the non-moving parts – they *resemble the robot's servo motors*. The head serves no significant purpose for now. However, it will become helpful when section *4.2 Constraints* define the ML constraints since it provides information on the relative rotation of the model.

The depicted model is still a hollow husk of `GameObjects` with no physical properties. Furthermore, several additional attributes, like the joints' mobility and speed, are missing. The following section shall discuss which of Unity's built-in `Components` can recreate real-world ASTERISK physical properties as closely as possible.

3.2 Physics

Unity provides a *robotics simulation toolkit* [29]. However, the architecture of this project does not rely on the main advertised features of the robotics toolkit, including ***ROS*** (Robot

Operating System) integration and live monitoring. One feature from the robotics toolkit, the so-called `ArticulationBody`, could be helpful to simulate the servo motors – but at this stage, Unity's build-in `Joints` should suffice to describe the motor's characteristics. In addition, `Joints` can link `GameObjects` together in different ways. `HingeJoints` can simulate the rotations of servo motors.

It is impossible to attach the legs as the body's children because this makes the legs inherit all the body's transform properties – including the rotation, position, and (problematically) the scale. Therefore, each time a `HingeJoint` rotates, the scaling of the leg changes because its relative rotation to its parent `GameObject` changes. Thankfully, FixedJoints can hold the legs onto the body without the inheritance problem.

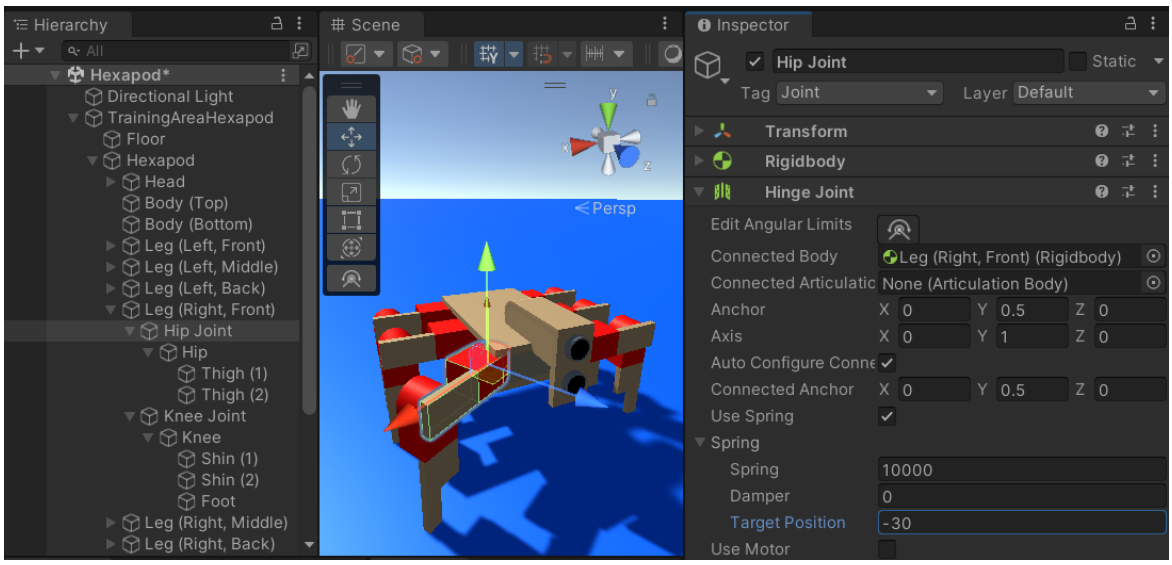

Figure 12 – Hierarchy of the Hexapod's GameObjects and Inspector of a sample Hip Joint.

The above figure shows the `Hierarchy` of the Hexapod's `GameObjects` on the left side, starting with an empty `Hexapod GameObject`, including the `Body`, `Head`, and `Legs` on the same level. Each `Leg` consists of two `HingeJoints`, one for the hip and one for the knee. These `HingeJoints` are essential for the physics simulation, specifically their `Spring` property (*Figure 12*, right side under `Inspector`) which holds fundamental physics properties. A `Spring`'s `TargetPosition` defines an angle that the `GameObject` wants to reach using a physics-based rotation. Contrary to just setting the `Transform Rotation` of an object, this does not teleport the `GameObject` into a particular `Rotation`, which means that physical obstacles act as they would in the real world – blocking the `Leg` from going any further. On the other hand, the `Spring` value indicates the force of the `GameObject` trying to reach its `TargetPosition`. The rest of the 3D model is equipped with `Rigidbodies`, further enforcing the physical properties of the model.

The following chapter shall implement an ML training model with the 3D model looking and behaving similarly to its real-world counterpart.

4 Reinforcement learning

Using RL, this chapter will follow the training steps to make the 3D model created in the last chapter walk. The first section carefully designs the *reward function* and the training model's *constraints*. Then comes the implementation of the *ML toolkit* into the 3D model's scripts. Finally, this chapter will discuss the *training* and its *results*.

4.1 Reward function

This section will consider a crucial aspect of successful RL – the reward function. Rewards are the only *numeric feedback* the training agent gets from the system. They are the linchpin of the training since *every action is measured relative to its impact on the total reward*.
Rewards do not need to be complex, as proven by a research paper on locomotion movement by DeepMind [30]. Instead, they used a simple reward function that revolves around the agent's *velocity* along the environment's forward axis. As a result, both the humanoid and the quadruped could move quickly in rich environments and avoid obstacles.
This section considers the design of two reward functions and tests both during training. The first one is the *simple reward function*, based on the premise of DeepMind's paper [30], only rewarding the forward velocity along the Z-axis. The idea of this approach is that the simulation learns to walk in the best possible way on its own – no briefing concerning what makes "good" movement is required. Another benefit is that there are no additional sources of error from interactions between rewards if there is only one metric.

$$reward_{simple} = velocity_Z \qquad \textit{Equation 1}$$

The second reward concept, the *complex reward function*, includes metrics that judge the movement's quality. The central question is what qualifies "good" movement. Even if "good" seems subjective, there are clear guidelines grounded on the assumption that "good" is equal to "natural", shaped by evolution. For this purpose, the following reward function shall consider the movement of the stick insect, as analyzed in The New York Times on how a stick insect walks [6].

$$reward_{complex} = \begin{cases} velocity_Z \\ -acceleration_Z \\ -instability = -velocity_{X,Y} - rotation_{X,Y,Z} \end{cases} \qquad \textit{Equation 2}$$

What *Equation 2* takes from [6], the forward movement should be *translational*, meaning it follows a straight line without rotation [31]. Therefore, any movement that is not translational should be penalized, including movement along the X- and Y-axis and rotations along all three axes. *Equation 2* summarizes those non-translational movements in an *instability* variable.

Also, the forward velocity of good locomotion should be constant, which equals an *acceleration* of zero – so the complex reward function can penalize the model according to the acceleration.

```
1  private void CalculateComplexReward()
2  {
3      // Calculate the metrics to measure locomotion quality.
4      _velocity = _rigidbody.velocity.z;
5      _acceleration = math.abs(
6          (_velocity - _velocityPrevious) / Time.fixedDeltaTime);
7      _instabilityRot = CalculateInstabilityRotation();
8      _instabilityPos = CalculateInstabilityPosition();
9      var instability = _instabilityRot + _instabilityPos;
10
11     // Set previous variable for the next action.
12     _velocityPrevious = _velocity;
13
14     // Return the reward function's result.
15     return _velocity - instability - _acceleration;
16 }
```

Code Snippet 1 – Complex reward function implemented in a Unity script. (C#)

Unity measures the velocity of each `Rigidbody`, which developers can access as seen in line *4* – the object `_rigidbody` is a referenced `Component` in the `Hexapod GameObject`. The acceleration is the derivation of the velocity, which lines *5-6* calculated as the difference between the current and the previous velocity, relative to the `Time.fixedDeltaTime`, which is not dependent on the framerate. Since the instability calculation is extensive, moving it to other functions (lines *7-9*) cleans the code. For the rest of the code, please refer to the linked GitHub repository [32] – the instability is calculated by measuring position and rotation differences between measurements. Finally, the simple reward works analogously, with `return _velocity;` in line *16*.

4.2 Constraints

A model's constraints disallow specific behavior. However, it is not always obvious what constraints the model requires because it is natural for ML to create unpredictable results. Therefore, developers regularly add constraints during the training when the model behaves in displeasing ways, like bug-fixing.

One example from this project is that during the first few iterations, the hexapod began to *spin around* the Y-axis (upward) when the training started. While there is nothing wrong with going through unfavorable movements during the initial training stage, it made the hexapod optimize forward movement by spinning. Although the complex reward function penalizes this behavior of excess rotation, it is still beneficial to disallow it entirely because a *direct*

forward path without detour is desirable, and *any form of 180° rotation is undesirable* in this model's case. Constraining the spin also prevents the hexapod from trying to walk backward or sideways.

The hexapod can *flip on its head* relatedly, rendering itself incapable of movement along any axis. Furthermore, it cannot stand back up when lying on its head because of the servo's restrictions. Therefore, this situation should lead to the termination of the episode, as *Code Snippet 2* describes.

Lastly, the model needs to train *moving forward in a straight line*. So, the vertical plane on which the hexapod walks should be narrow. The model's goal becomes to balance on the vertical plane, with no other direction to go but forward.

```
1  // End episode with penalty when hexapod falls off
2     the plane, turns around, or turns on its head.
3  if (   (transform.localPosition.y < -3)
4      || (head.transform.position.z < transform.position.z)
5      || (lowerBody.transform.position.y
6              > upperBody.transform.position.y))
7      return true;
8  return false;
```

Code Snippet 2 – Boolean function summing up the model's constraints. (C#)

Code Snippet 2 line *3* prevents the model from falling off the vertical plane. As discussed in section *3.1 GameObjects*, the head proves practical in line *4* to check if the hexapod has turned to face backward. Lines *5-6* check whether the upper body plate is below the lower body plate (`Body(Top)` and `Body(Bottom)` in *Figure 12*) for the hexapod to not flip. Section *4.3 Implementation* will show in what context *Code Snippet 2* goes. The training area setup looks like *Figure 13* shows.

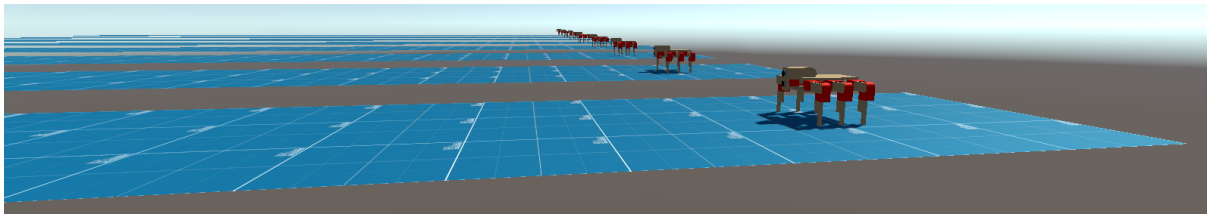

Figure 13 – Initial training setup for multiple hexapod agents.

Each plane is a separated training area, including one hexapod agent each. All agents in the scene train simultaneously and independently – the results accumulate in the end. Keep in mind that the *Z-axis* goes to the figure's left (*Hexapod's forward*), the *X-axis* goes from the back to the front (*Hexapod's left*), and the *Y-axis* goes *up*.

4.3 Implementation

Unity uses the standard `MonoBehaviour` base class for all the `GameObjects`. However, the ML toolkit requires an `Agent` object instead, which swaps the `Update()` function (executed once per frame) for three different methods:

- `OnEpisodeBegin()` runs when the agent fails, succeeds, or when the time runs out.
- `CollectObservations(VectorSensor sensor)` gets the values according to which the agent makes decisions – these observations can, for example, concern an object's movement speed, rotation, or position.
- `OnActionReceived(ActionBuffers actions)` processes the agent's actions and assigns rewards. It receives the generated decisions in an array for each training step.

The following paragraphs will inspect the agent's functions in depth.

```
1  public override void OnEpisodeBegin()
2  {
3      // Reset all joint angles.
4      foreach (var joint in joints)
5          SetSpringPosition(joint, 0);
6
7      // Bring spider back to the start position.
8      var spiderTransform = transform;
9      spiderTransform.localPosition = new Vector3(0, 0, 0);
10     spiderTransform.rotation = Quaternion.identity;
11
12     // Reset velocity
13     _rigidbody.angularVelocity = Vector3.zero;
14     _rigidbody.velocity = Vector3.zero;
15 }
```

Code Snippet 3 – OnEpisodeBegin() running when the agent resets. (C#)

When the hexapod agent does something that ends the episode, it resets its state. Lines *4-5* reset the joints to neutral positions. Keep in mind that the neutral angle in the agent's `HingeJoints` is 0°, but it resembles the 90° angle of the robot. Lines *8-10* set the position and rotation of the agent back to 0. Lastly, lines *13-14* reset the physics model's velocity – this is important for when the agent falls off the platform and builds momentum. If the function did not reset the `Rigidbody`'s velocity, it could catapult the agents forward once the episode resets.

```
1  public override void CollectObservations(VectorSensor sensor)
2  {
3      // Joints' rotations.
4      foreach (var joint in joints)
5          sensor.AddObservation(joint.spring.targetPosition);
6
7      // Distance from the center of the plane.
8      sensor.AddObservation(transform.localPosition.x);
9
10     // Rotation on the Y-axis.
11     var bodyPositionX = upperBody.transform.position.x;
12     var headPositionX = head.transform.position.x;
13     sensor.AddObservation(bodyPositionX - headPositionX);
14 }
```

Code Snippet 4 – CollectObservations() receiving all metrics and joint positions. (C#)

Observations are the information on which the model bases its decisions. Those could be, for example, a ray cast, simulating a distance sensor, to find obstacles.

In this project's case, the simulation needs to find an efficient *locomotion movement algorithm*. Locomotion means repeating the movements and making decisions based on the previous leg positions. So, the observations include all 12 joints with angles (lines *4-5*). In line *8*, the agent gets its X-axis position, which should help the hexapod walk straight along the Z-axis without falling off the narrow plane.

The hexapod's *rotation towards its left and right* is not as trivial to calculate. While Unity offers a Y-axis rotation in its `transform` API, it has a fundamental problem: within the user's understanding, Unity uses Euler angles to calculate rotation, but actually, it uses quaternions in the background [33]. Furthermore, *Euler angles are ambiguous*; reading the same Y-axis rotation could return any two distinctive values between 0 and 180, which would only confuse the model. For this purpose, lines *11-13* calculate a proxy that resembles the Y-axis rotation based on the head's and body's X-axis positions relative to each other. By subtracting `headPositionX` from `bodyPositionX`, the model receives a value > 0 when it turns to the right or a value < 0 when it turns to the left.

```
1  public override void OnActionReceived(ActionBuffers actions)
2  {
3      SetAllJoints(actions);
4
5      SetReward(CalculateSimpleReward());
6
7      if (!ReachedConstraint(transform)) return;
8      EndEpisode();
9  }
```

Code Snippet 5 – OnActionReceived() executing movements, rewards, and penalties. (C#)

The function from *Code Snippet 5* has three tasks:

- line *3* uses the model's decision and sets all the joint positions,
- line *5* calculates and sets the reward based on the chosen reward function,
- line *7* checks if the model reached any of the constraints and, if so, ends the episode.

The ML toolkit's `DecisionRequester Component` generates the required *decision array* according to the requirements. The decision array must consist of *12 discrete values* (between 0 and 90), representing the `HingeJoint`'s target angles.

`OnActionReceived()` takes the model's decision after a fixed amount of *academic steps*. One academic step equals one `FixedUpdate()` cycle, with a sum of 50 times per second. The `DecisionRequester` includes a *decision period* parameter (between 1 and 20) to specify how many academic steps pass between each decision. The following equation shall set this parameter to 15, which means Unity calls `OnActionReceived()` every 300ms:

$$\Delta T_{decision} = \frac{interval}{academic\ steps} * decision\ period = \frac{1s}{50} * 15 = 0.3s = \underline{\underline{300ms}} \qquad \textit{Equation 3}$$

This frequency corresponds with the *servo motor's technical limitations* since too many commands lead to an overcharge of the microcontroller, as section *6.1 Power supply* will elaborate. So, when generating the *command file*, the system can sample one command line per `OnActionReceived()` call from the model.

This section concludes the scripting setup, but before the training can begin, the *hyperparameters* still need some consideration – the predefined constants that are not trained [34]. These values specify the training conditions for the ML algorithm. In the ML toolkit, the hyperparameters are listed in a ***YAML*** (YAML Ain't Markup Language) config file [35]. Optimizing hyperparameters is an extensive topic, but thankfully, the developers behind the ML toolkit provide a default configuration that shall be sufficient for this project because it includes parameters optimized for RL problems.

The following section will use all the preparations from this chapter to run the simulation.

4.4 Training

The simulation needs to try out different configurations, including the reward function and the physics parameters, to see whether the overall result can improve. Therefore, this section will begin with the standard Unity physics and the complex reward function – corresponding to the order in which the training units were conducted.

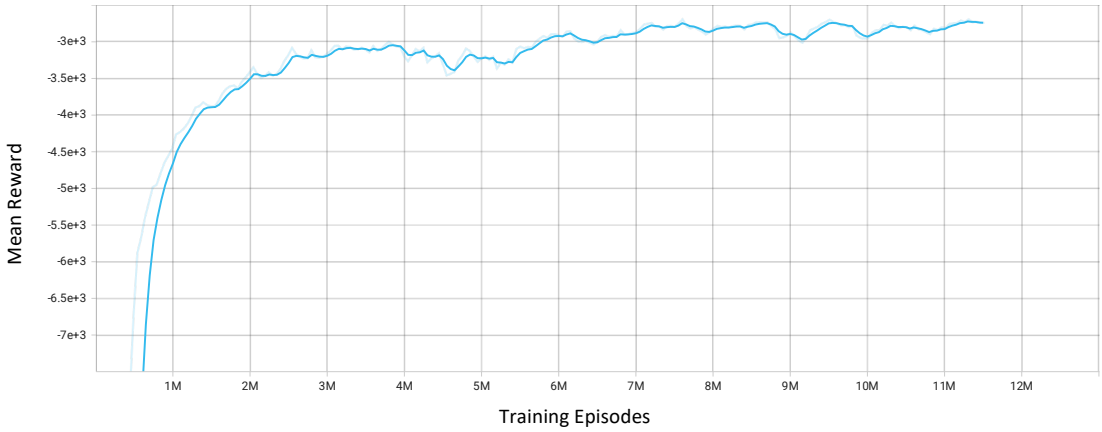

Figure 14 – Mean complex reward with standard Unity physics.

The ML toolkit comes with an installation of *TensorBoard*, which automatically generates graphs like the line graph above. *Figure 14* shows a *typical learning curve*: it starts with a low mean reward where the agent tries random actions to see what they do – and progresses quickly into a higher reward through the episodes. Finally, the training levels off at the turning point of around 2 million episodes. Keep in mind that the Y-axis scaling is insignificant since it depends on the size and frequency of the rewards, which are equivalent between episodes.

At first sight, this is a decent curve that indicates that the training has reached a local optimum; however, it is not entirely what one would expect. All values on the Y-axis are negative, which means that the agents receive high penalties and optimize for the lowest penalty. In the complex reward function (*Equation 2*), there are two variables causing penalties; instability (undesired movements) and acceleration. So, it makes sense that the agent, trying random actions initially, starts with high penalties and low rewards.

As discussed in section *4.2 Constraints*, multiple, easy-to-reach constraints end an episode immediately. The combination of *high penalties and constraints* ending the episode leads to an unpredicted behavior – *the simulation did not learn to walk*; *it learned the most efficient way to end the episode* because this was the quickest way to prevent penalties. *Figure 15* sums up the model's behavior in this training session.

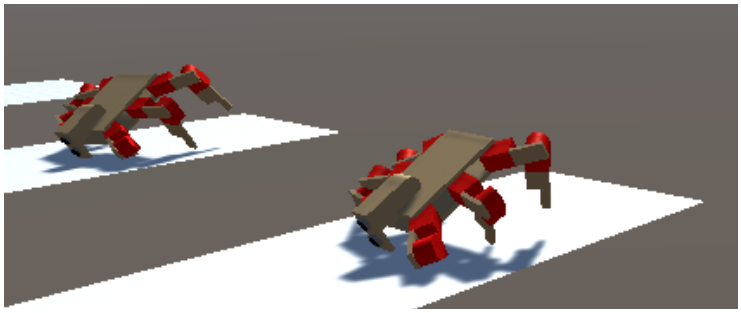

Figure 15 – The ML model learns to somersault to end the simulation & minimize penalties.

The figure shows a lousy result. In retrospect, combining penalties with easily reachable constraints is not advisable. So, there are multiple solutions to try, either changing the penalty or the constraints. The simulation should keep the constraints because the model always reaches them when committing to unwanted behavior. The complex reward function can discourage the "somersault model" by penalizing the model every time it hits a constraint and terminates the simulation.

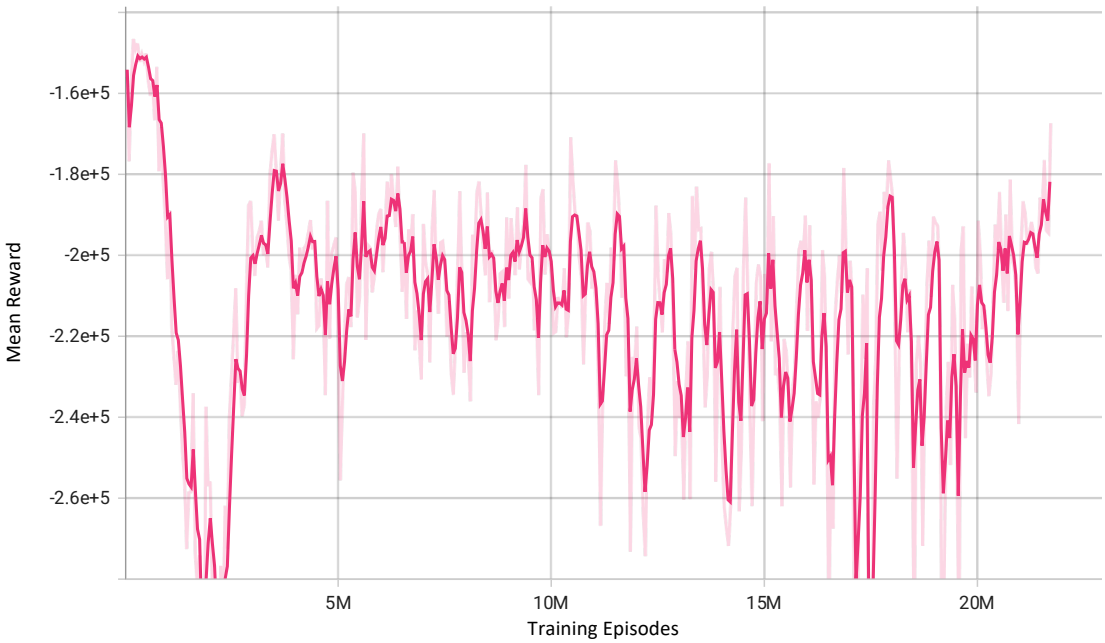

Figure 16 – Mean complex reward with a penalty for hitting a constraint.

This figure depicts a sobering training result – the training starts with a local optimum by random chance, which the training cannot reach again within over 20 million episodes. Section *4.1 Reward function* mentioned that the complex reward function allows for more sources of errors – and *training the model consumes a large portion of project time*. For example, training the model for 20 million episodes takes roughly 17 hours on the available hardware. With a fluctuating curve like in *Figure 16*, the model does not look too promising, so the effort shall concentrate on the simple reward function – as this approach is less likely to produce disappointing results due to the reduced sources of errors.

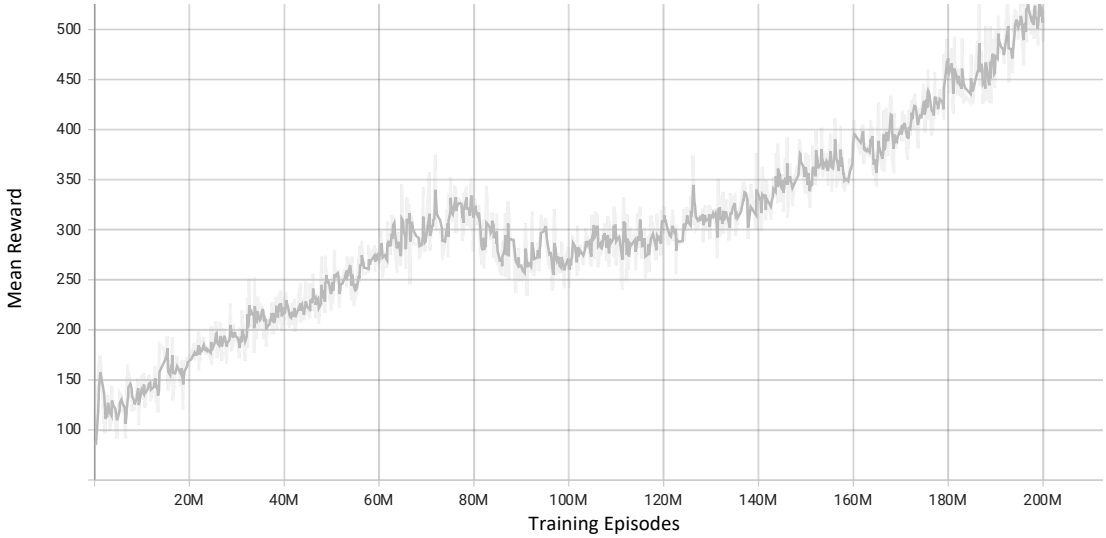

Figure 17 – Mean simple reward throughout 200M training episodes.

After circa seven days of training, the model passed 200 million episodes. The curve depicted above looks exceedingly promising – it has a steady, linear rise from the beginning until around 80 million episodes, where it reaches a local optimum. At that point, it lowers the reward, probably in favor of a different approach. Therefore, it can reach an even higher reward in total, with a steady growth between episode 100 million and the training's preliminary end. As *Figure 17* depicts, the mean reward is only positive because the simple reward function employs no negative reward penalties.

This curve is a strong indicator that further training is advisable. However, *Unity's physics engine poses an issue* beyond the model's performance and improvement ability. The simulation's movements could be described as *bouncy*, not portraying the natural world's gravity. Appendix *simulation/bouncy_hexapod* shows this bouncy behavior. So, the solution is to *adjust the gravity scale*.

Scaling the gravity is necessary for the simulation to become more realistic. Up to this point, this work assumed the physics engine mimics a realistic behavior. The Unity community discussions forum found a feasible solution to this issue before [36]. It is necessary to fine-tune the simulation's gravity scale by trail-and-error until the model's movements look similar to how the robot in the real world could act – primarily, not jumping around.

With this approach, this scenario's most realistic gravity scale is about 10x, equivalent to an acceleration of $9.81 * 10 = 98.1 \frac{m}{s^2}$ (relative to Unity's implementation of gravity and mass). With this configuration, the robot mostly sticks to the ground while it walks, which means the simulation can proceed with the following attempt to train the model.

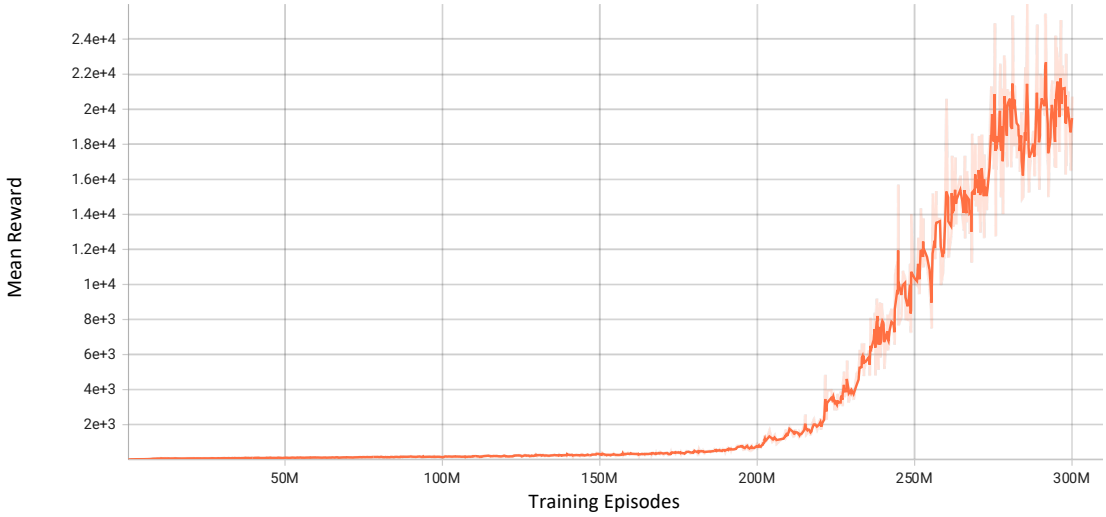

Figure 18 – Mean simple reward using a high gravity during 300M episodes.

After circa 12 days of training, the simulation reaches the 300 million episodes mark. The mean reward axes of *Figure 17* and *Figure* 18 are comparable because they have the same reward function. However, this training records a mean reward of circa 1000 at episode 200M, whereas the previous training only shows a reward of half as much at the same point. This higher value must be a coincidence in training, making this model double as efficient at the 200M mark as the previous one since ML is not deterministic.

The most astounding discovery is the exponential rise of the mean reward, increasing it by more than 2000% between episodes 200M and 280M. Another promising sign is that the reward reaches a local optimum around the 280M point, which means the model does not significantly improve until the end. This plateau indicates a perfect moment to stop the training for now, with the training times and the approaching deadline being the two most significant contributing factors. The efficiency in which the simulation moves forward is comprehensible in appendix *simulation/300M_commands_vid*, and the difference in movement before the exponential reward rise in appendix *simulation/200M_commands_vid*.

These videos conclude the topic of RL, and the project progresses to the next chapter about how to transfer the training result to the real world.

5 Command interface

There is the possibility to use the code Adeept provides with the hexapod. However, it is of poor quality – it is tough to understand, and the source code for all the components lies within one large file [37]. Furthermore, it appears the developers did not care to repurpose the code. Therefore, this chapter will detail an object-oriented approach to the robot's code and create a uniform *API for the command file*.

5.1 Robot legs API

Two components are necessary to manipulate the robot's legs; a single `Leg` object consisting of two servos and an array of `Legs` that can move sequentially.

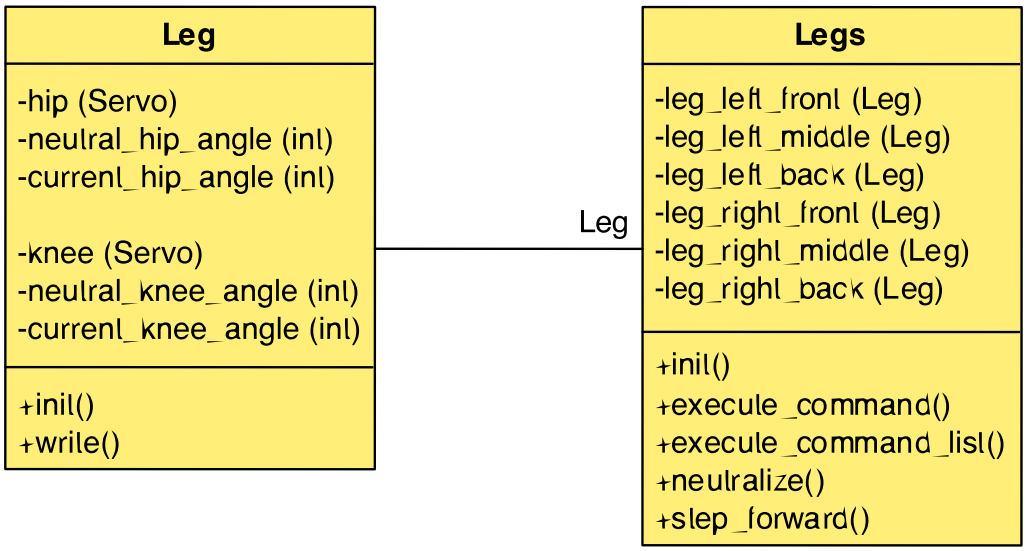

Figure 19 – UML diagram of the robot's command interface.

Starting with the `Leg`, it should programmatically initiate with a `neutral_angle` which has one primary reason; to *calibrate the physical legs*. Although the system commands 90° to the servo, the robot's leg does not stand at a perfect 90° angle – it is often necessary to adjust the angles manually, degree-by-degree. The calibration only needs to be done once, and the values can then lie in memory for the specific robot used in this research project. Initializing the `neutral_angle` also sets the `current_angle` to the same value.

Furthermore, Arduino's built-in `Servo` already has some valuable methods, one of them being `write()`, which sets the angle for a given `Servo`. This project's framework's code uses the native `write()` method and calculates the new angle relative to the `current_angle`. Finally, the function saves the resulting value into the `current_angle`. While there is a `read()` function inside the `Servo`, it is not very reliable, as discussed in *2.2 Robot*, making

`current_angle` the *single version of truth* [38] or the only reliable source of the angles' positions.

```
/* @param absolute_angle: New angle of the Servo.
 * @param knee_or_hip: Select the Servo of a leg,
 *         e.g., knee = 'k' or hip = 'h'.
 */
void write(int absolute_angle, char knee_or_hip)
{
    switch (knee_or_hip)
    {
        case 'h':
            this->current_hip_angle
             = absolute_angle - 90 + this->neutral_hip_angle;
            this->hip.write(this->current_hip_angle);
            break;
        case 'k':
            this->current_knee_angle
             = absolute_angle - 90 + this->neutral_knee_angle;
            this->knee.write(this->current_knee_angle);
            break;
        default:
            return;
    }
}
```

Code Snippet 6 – Leg.write() method that changes the angle of a Leg's Servo. (C++)

The `Legs` object consists of all six of the ASTERISK's legs. The critical method is `execute_command_list()`, implementing a loop of `execute_command()` calls to make the robot move according to a *structured array*.

```
/* @param command_list: 2D array consisting of commands,
 *         including info on delay time and 12 servo angles.
 * @param amount_of_commands: Integer that describes how many
 *         commands are in the command_list.
 */
void execute_command_list(int command_list[][13],
                          int amount_of_commands)
{
    for (int command_index = 0;
         command_index < amount_of_commands;
         command_index++)
    {
        execute_command(command_list[command_index]);
    }
}
```

Code Snippet 7 – Legs.execute_command_list() method that updates all Servos. (C++)

The following section concerns the shape for the input to `execute_command()`.

5.2 Command file

Section *2.5 Architecture* has mentioned the structured input arrays, so this section shall define their precise structure. First, consider *G-Code* – a widespread command control for automated machine tools [39], e.g., robotic manufacturing arms, to tell their motors what to do. It is a text file including one command and an array of parameters for each line.
Commands for the robot are always the same in this case, making the explicit declaration of one unnecessary. It is interesting how a developer could manipulate the robot's hardware (including its RGB LEDs or the Sensors, for instance), but this is out of scope for this research project, as section *2.2 Robot* has established. So, the lines of this project's file will only need an assortment of parameters.

```
1  300, 120, 90,  60, 90, 120, 90, 120,  90,  60,  90, 120,  90
2  300, 120, 90,  60, 60, 120, 90, 120, 120,  60,  90, 120, 120
3  300,  60, 90, 120, 60,  60, 90,  60, 120, 120,  90,  60, 120
4  300,  60, 90, 120, 90,  60, 90,  60,  90, 120,  90,  60,  90
5  300,  60, 60, 120, 90,  60, 60,  60,  90, 120, 120,  60,  90
6  300, 120, 60,  60, 90, 120, 60, 120,  90,  60, 120, 120,  90
```

Code Snippet 8 – Example command file for taking one step forward. (TXT)

This table shows an array of six consecutive commands, forming a single step of the *regular tripod gait of an insect*. Each line includes *13 parameters*, the first being a delay in milliseconds, followed by one angle for each `Servo` of ASTERISK's 12 servos. Chapter *6 Real-world applications* shall use this algorithm inspired by nature [40] against the ML-generated algorithm.

```
1  /* @param commands: Array of 13 integer values. Delay at index
2   *                  0, absolute servo angles at index 1 to 12.
3   */
4  void execute_command(int commands[13]) {
5      delay(commands[0]);
6
7      leg_left_front.write(commands[1], 'h');
8      leg_left_front.write(commands[2], 'k');
9      leg_left_middle.write(commands[3], 'h');
10     leg_left_middle.write(commands[4], 'k');
11     leg_left_back.write(commands[5], 'h');
12     leg_left_back.write(commands[6], 'k');
13
14     leg_right_front.write(commands[7], 'h');
15     leg_right_front.write(commands[8], 'k');
16     leg_right_middle.write(commands[9], 'h');
17     leg_right_middle.write(commands[10], 'k');
18     leg_right_back.write(commands[11], 'h');
19     leg_right_back.write(commands[12], 'k');
20 }
```

Code Snippet 9 – Legs.execute_command() method to set all legs' Servos once. (C++)

The `Legs.execute_command()` function moves the legs on the left side, then the legs on the right side, from front to back, always the hip `Servo` first, and then the knee `Servo`.
Look at `Legs.execute_command_list()` with *Code Snippet 8* as an input to further analyze what happens. First, when executing a command, the `delay()` blocks any other command to the driver board for the duration of the first entry of the structured array in milliseconds. Then, each `Servo` of each `Leg` is set in order, going through the parameters. Finally, the execution repeats for each line separately.
The first line `300,120,90,60,90,120,90,120,90,60,90,120,90` translates to:

> *Wait for 0.3 seconds, then move the left-front, left-back, and right-middle legs forward by 30°, and the left-middle, right-front, and right-back legs backward by 30° while leaving all knee joints in a 90° neutral position.*

This command interface proves to be very powerful for developers once they get used to it because it allows them to design movements based on precise angles quickly.
Using `Leg.execute_command()`, the `neutralize()` method is just a single command, with a delay of 0 ms, setting all the Servos to an angle of 90° simultaneously.

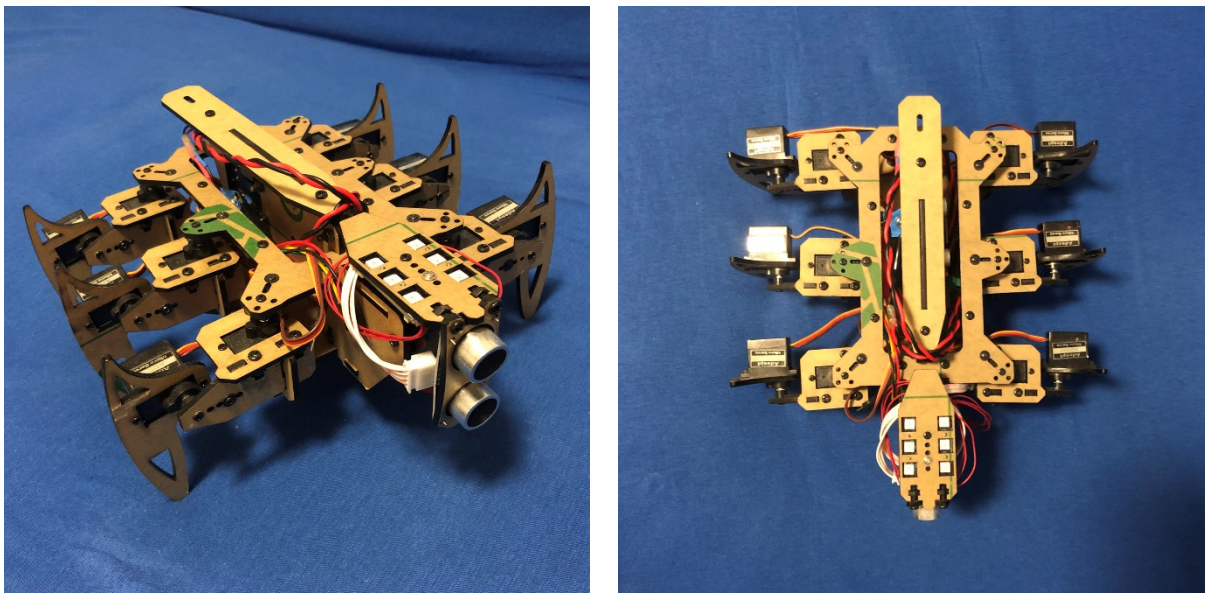

Figure 20 – ASTERISK in its neutral position (all servos in 90°).

Successfully executing the `neutralize()` command (and making ASTERISK walk and perform a cute dance for testing purposes) proves G-Code as a command control. The following section will take the ML model from the previous chapter and extract the movement by sampling the ML model's decisions.

5.3 Generating a command file

The system must first extract the command file from the movements Unity generates. The initial plan was to measure the rotations of the `HingeJoints` by sampling them, but there is a more practical way. The system can write each *decision array* the ML model generates into a file while the simulation walks. Consider the following code snippet.

```
1  private void WriteToCommandFile(ActionBuffers actions)
2  {
3      var delay = 1000 / 50 * _decisionRequester.DecisionPeriod;
4      var command = delay.ToString();
5
6      foreach (int action in actions.DiscreteActions)
7          command += ", " + (action + 45);
8
9      using var writer
10         = File.AppendText("Assets/CommandFiles/CommandFile"
11                           + _commandFileIndex + ".txt");
12
13     writer.WriteLine("{" + command + "},");
14 }
```

Code Snippet 10 – Command file API sampling the simulations movement. (C#)

The above code snippet executes each `OnActionReceived()` loop every 300 ms, as *Equation 3* has established. This frequency is automatically calculated depending on the `DecisionPeriod` in line *3*. Line *4* saves the resulting delay in the command line variable. Then the actions are iterated and written into the string in lines *6-7*; here, the order in which the system writes the values to the file matters, as the robot expects a specifically structured array. Therefore, the 3D model's `HingeJoints` need to attach to the actions array in the same order as the servos connect to the microcontroller. Finally, lines *9-11* create a new command text file, and line *13* appends the string.

The next chapter will load the sample file from appendix *simulation/300M_commands* into the robot using the robot's serial port.

6 Real-world applications

Creating the command line interface was the first step in transferring an algorithm to the physical robot. After training an algorithm, this chapter will showcase how the robot performs compared to the 3D simulation. This chapter will discuss the *hardware limitations*, the *experiment conditions*, and the *differences* between the robot's and the simulation's movements.

6.1 Power supply

ASTERISK runs on a pair of rechargeable batteries, each providing a maximum voltage of 3.7V. Using a multimeter, the measured voltage of one battery after charging is around 3.4V, resulting in a total voltage of circa 7V. However, since the driver board runs on 6V, both batteries barely provide the necessary charge and fail to do so for an extended period. This lack of power is an issue; neither moving the servos nor uploading code is possible with a voltage below 6V. Furthermore, since the batteries drop in voltage very quickly, using them in development would involve countless recharging. An ***LPS*** (Laboratory Power Supply), which lets a developer adjust voltage and electrical current, would solve this problem. Nevertheless, this project runs on a budget of nearly zero, and a decent LPS can be expensive, so the following paragraph describes building a makeshift one.

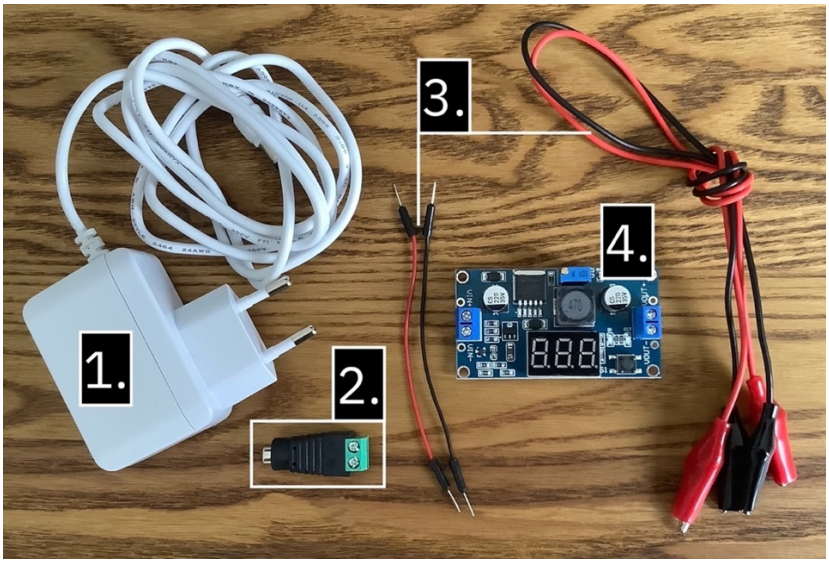

Figure 21 – Components for building a makeshift laboratory power supply.

The parts can be bought online and include:

1. a 24W (12V, 2A) ***DC*** (Direct Current) power adapter,
2. a standard DC plug to single wire adapter,
3. four jumper cables, with and without crocodile clips,
4. a DC-DC voltage converter

After assembling the LPS, it is hooked up to a power socket and, using the crocodile clips, the battery slot's connectors.

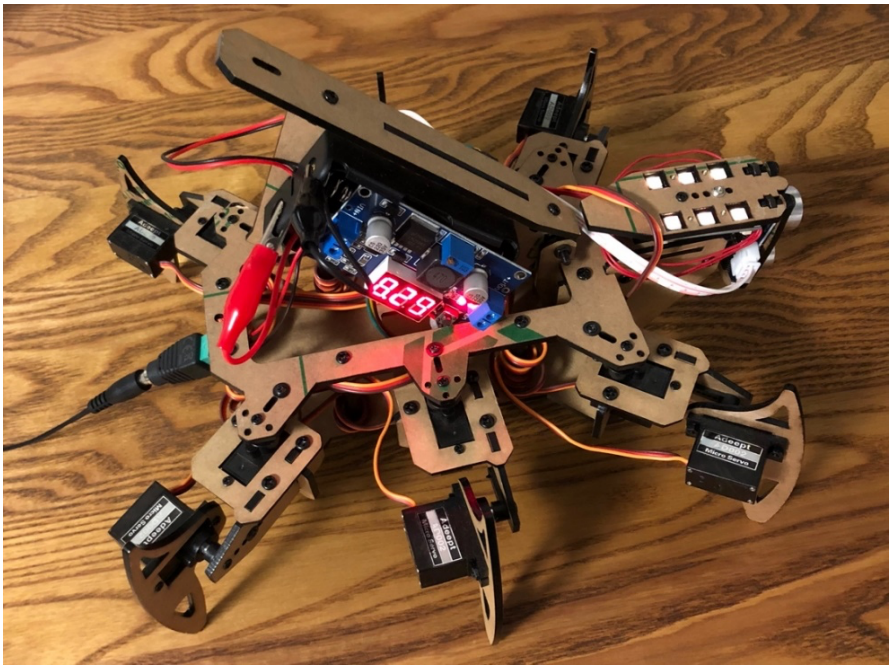

Figure 22 – An assembled robot with a built-in laboratory power supply.

The figure above shows the fully assembled hexapod. The makeshift LPS works but is somewhat unreliable since the crocodile clips are prone to slip off their connectors. However, a closer look at the instructions proves that the voltage converter is unnecessary.

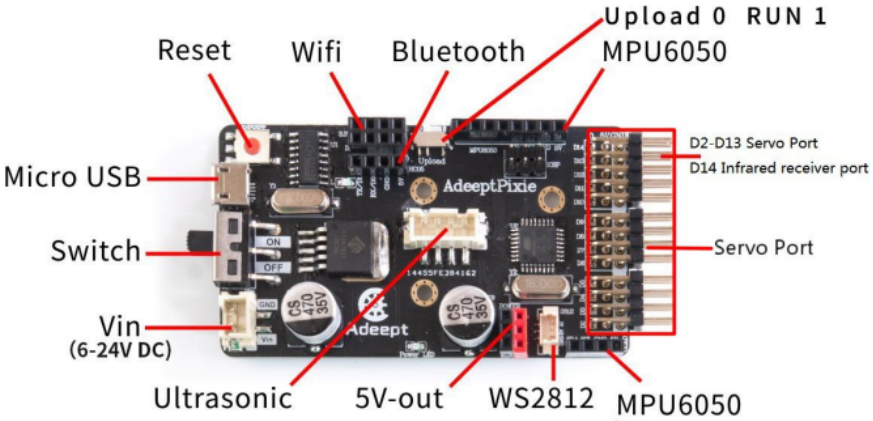

Figure 23 — AdeeptPixie driver board overview [16]

On the left side, *Figure 23* shows Vin (6-24V DC), which means the board can withstand an input voltage of up to 24V and requires at least 6V to function. The power adapter has an output voltage of 12V and is ideally suited for a direct connection.

The system no longer requires the voltage converter, but it is also impossible to connect the crocodile clips directly to the driver board. So, the idea is to take a battery connector where the crocodile clips can connect instead and solder it directly to the driver board's Vin pins.

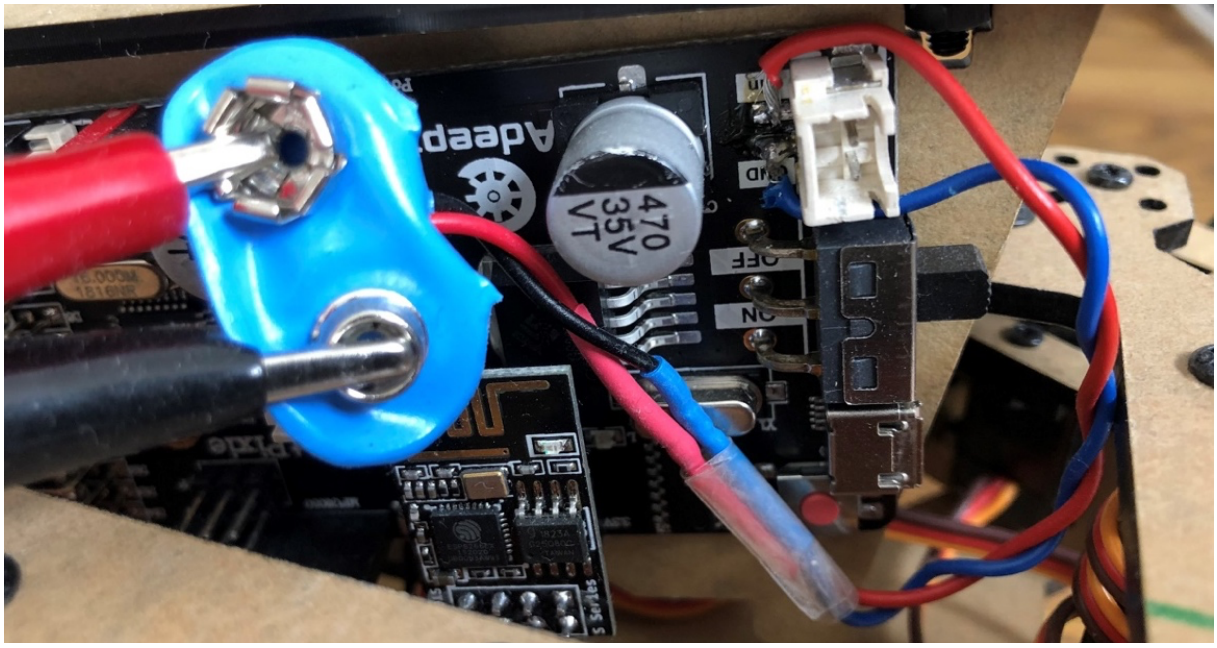

Figure 24 — Battery connector soldered directly on the AdeeptPixie.

With this layout, ASTERISK is noticeably less bulky. Furthermore, using the DC plug to single wire adapter on the connection between the power adapter and the jumper cables, one can quickly (un-)plug the robot from the power supply. This setup achieved the goal, and ASTERISK will have a constant power supply without any battery fluctuations. However, it is now also restricted by cable, which should not be an issue within the topic of this project.
With the power supply in place, the upload should work as intended. By adding the command file generated by the ML model to the code, PlattformIO attempts to upload the new command through the serial port.

```
1  Configuring upload protocol...
2  AVAILABLE: Arduino
3  CURRENT: upload_protocol = arduino
4  Looking for upload port...
5  =============== [FAILED] Took 1.09 seconds ===============
6
7  Process finished with exit code 1
```

Code Snippet 11 – Terminal output of a failed upload using PlattformIO.

Unfortunately, the upload to the robot fails when it is using its servos, producing the above error message. The reason for the error is an overcharge caused by the driver board being unable to handle sending pulse signals to the servos and receiving an upload at the same time. This upload blockage means the system cannot upload new commands to the robot, but it also means that faulty commands will damage the servos for as long as one turns the power off. The main issue is that the board cannot receive signals when powered off.

Still, there is a way to reset and reupload data to the driver board. One needs to unplug all the servos, power on the board – which now is not overcharged by the servo motors – and upload the new code.

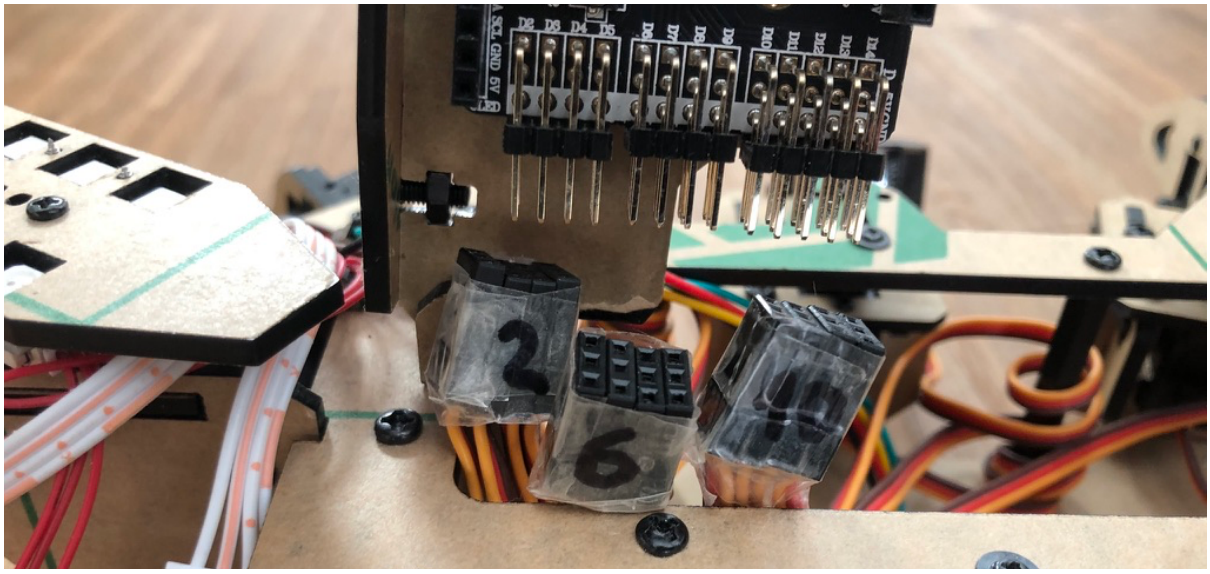

Figure 25 – AdeeptPixie diver board servo pins and labeled servo connectors.

It is beneficial to tape the servo connectors in four-packs and label them to limit the work necessary to replug them into the driver board after each reset. In the case of the figure above, the labels indicate the index of the first pin. The packs plug into the pins *D2-D5*, *D6-D9*, and *D10-D13*. Additionally, the pins plug in the same order as iterated through in the simulation's 3D model in *Code Snippet 9*, front to back, left to right.

Please note that the driver board's connectors should never change while the board is running. So before connecting the servos to their pins, one needs to turn the power supply off and disconnect the serial port connection.

Now that PlattformIO loaded the command file into the driver board using the code from the second GitHub repository [41], the following section examines the robot's movements in the real world.

6.2 Experiment setup & results

This section shall consider three different floor surfaces with varying attributes to give the algorithms a chance to show their potential in different environments. All of the grounds depicted below have various properties. For example, the *cushion* has high friction but gives in to the robot's weight, making it challenging to walk on. On the other hand, the *hardwood* floor has a very smooth, low friction surface, but it is unflexible, firmly letting the robot stand on it. Lastly, *cardboard* combines the attributes of both grounds; high friction where the legs find a decent grip and a stiff, non-yielding surface.

Figure 26 – Three floor textures used for testing the resulting movements on different surfaces – cushion (left), hardwood (middle), and cardboard (right).

Two algorithms shall be tested on all three surfaces to determine their efficiency. The first will be the *natural* walking algorithm, the tripod gait – secondly, the *generated* algorithm resembles the beforementioned command file.

The experiment for each setup is the same. Inbetween the setups, one needs to reset the robot by unplugging all servo motors and loading the neutral position on the driver board. Then, one replugs the servos and places the robot on a start position of the corresponding surface. Lastly, the video recording takes place – PlattformIO uploads the code to the robot, which executes the predefined commands.

6.3 Results

The reader can examine the experiments' results in the appendix *robot*. The folder has six different videos – three videos per algorithm and two videos per ground surface.

The natural algorithm performs well on the cushion and the cardboard, as in appendix *robot/natural_on_cushion* and *robot/natural_on_cardboard*. It can use the quick, precise movements of three legs at a time to push itself forward and make leaps, adjusting the legs in between. Appendix *robot/natural_on_hardwood* shows a different story; the robot's legs fail to grasp the slippery hardwood. The movement seems too fast, and the polymer legs slide front to back on the wood surface.

Now, for the generated algorithm. Appendix *simulation/300M_commands_vid* shows that the simulation does not move symmetrically. Instead, the 3D model turns slightly to its right and uses its right middle and rear leg to gain momentum while the legs on the left side stabilize the whole body. As a result, the movements do not transition well to the real world. Appendix *robot/generated_on_hardwood* shows a similar behavior: the robot does not move from the spot because the movements are too hectic and the ground is too plain.

On the other hand, appendix *robot/generated_on_cushion* shows movement to the side, but the legs struggle with the cushion – both middle and back right legs form a scrape in the

ground since they are so close to each other the whole time, digging the robot a hole. Lastly, in the appendix *robot/generated_on_cardboard*, the robot moves far but backward.

These results conclude that the transition of an efficient, generated movement algorithm to the real world has failed during this research project – however, it could optimize by implementing methods described in *7 Conclusion*. The ML-generated algorithm works splendidly inside the simulation; in the real world, it performed poorly because of the deviating conditions. Nevertheless, the following paragraphs shall briefly analyze the generated values and compare them to the movement values inspired by an actual insect [40].

left_front		left_middle		left_back		right_front		right_middle		right_back	
hip	knee	hip	knee	hip	knee	hip	knee	hip	knee	hip	knee
120	90	60	90	120	90	120	90	60	90	120	90
120	90	60	60	120	90	120	120	60	90	120	120
60	90	120	60	60	90	60	120	120	90	60	120
60	90	120	90	60	90	60	90	120	90	60	90
60	60	120	90	60	60	60	90	120	120	60	90
120	60	60	90	120	60	120	90	60	120	120	90

Table 1 – Colorized G-Code of movement algorithm of the tripod gait.

As the table above shows, each leg has a reoccurring pattern. Knee joints move upwards and back to the center, while hip joints go back and forth. Keep in mind that the servo motors on the left side have an inverted axis compared to the right side. This repeating pattern leads to the precise movement in the appendix *robot/natural_on_cardboard*.

left_front		left_middle		left_back		right_front		right_middle		right_back	
hip	knee	hip	knee	hip	knee	hip	knee	hip	knee	hip	knee
48	45	61	82	134	79	115	64	118	125	124	120
106	91	105	55	74	80	47	53	130	47	133	56
66	90	132	113	134	72	128	121	133	127	120	94
91	83	107	45	53	83	46	125	132	45	130	50
55	77	132	107	130	71	123	132	133	128	119	94
109	57	93	48	76	72	60	113	132	50	127	58
65	82	130	111	134	72	123	81	132	130	124	92
97	79	62	45	63	80	80	53	131	46	124	63

Table 2 – Colorized G-Code of movement algorithm generated by the ML model.

The generated algorithm turns out to be much more chaotic, as *Table 2* shows. However, there are repeating patterns, indicating locomotion behavior – but instead of repeating in blocks of three like in *Table 1*, all joints seem to have adopted an individual pattern. For example, the front left hip has a reoccurring movement of four angles; low-high-low-center. On the right side, the middle and back hips take a passive approach, pointing straight for the most part and using their corresponding knees to kick the simulation forward using a high-low repetition. This pattern moves the simulation forward quickly and efficiently, as shown in appendix *simulation/300M_command_vid*. At the same time, it performs weakly in the real world, as seen in appendix *robot/generated_cardboard*, and in no way like insects in nature.

These results close this student research project. The last chapter summarizes the learnings from the previous year of research and gives a brief outlook for the future.

7 Conclusion

Before jumping into the evaluation, this chapter wants to appreciate that the project completed the proposal in section *2.5 Architecture*. This work spawned a 3D model resembling Adeept's robot. Using the Unity ML toolkit, the simulation learned how to navigate along a straight path using six actuators efficiently. The command interface includes an output for Unity to generate the command file and an input for the robot to perform these movements in the real world. However, it was good that all four main topics in this research project were separated because delays at any stage did not significantly cause the other stages to suffer in quality. All the topics were developed in parallel, without too many cross-dependencies during this year of research. This approach saved precious time and resources.

Nevertheless, the most problematic issue in this project was the deviating conditions between simulation and reality. After weeks of training and tweaking the parameters, the ML model became highly efficient in walking. The research proved that a simple reward function of a single parameter is enough to generate complex movement. With more computing power and shorter training times, the project might have explored fine-tuning the variables to enhance the training. However, considering the tight timeframe, focusing on the simple reward function could have saved time in other areas.

Furthermore, the ML model's efficiency did not transition to the robot. The project's timeframe was too tight to investigate possibilities of tweaking the physics model to resemble the reality more closely. Another issue the project experienced lies in Unity's representation of physics. The floaty 3D simulation did not accurately represent the real world; even after tweaking the gravity parameters, it abuses Unity's physics engine to glide over the plane.

Including the robot's behavior into the training's feedback loop also turns out to be essential. It makes sense that the robot performs inadequately because not only do the physics differ, but the machine learning's most significant advantage – basing decisions on observations – is entirely lacking from the robot. ASTERISK does not have the sensors (e.g., positional feedback from servos, inertial measurements from gyroscope and accelerometer) or a powerful driver board to handle an actual machine learning model. Furthermore, the command file the system sends to the robot is static and does not react to the current environment. So, to harness the full potential of RL in the real world, one would need to create the conditions to allow for the RL feedback loop *Figure 3* shows. Therefore, if someone attempts something similar, this work's author recommends focusing on bridging the gap between simulation and reality.

Acknowledgments

Personal thanks go to *Patrick Winterhalder*, M.Sc. in aerospace engineering at the University of Stuttgart, for the great help handling the recalcitrant hardware and its power supply.

Personal thanks to *Falko Kötter* for the supervision and many of the thought-provoking ideas that went into this work.

Last but not least, thanks to the reader for their interest in this work

Bibliography

[1] Encyclopedia Britannica, "robot," 04 February 2021. [Online]. Available: https://www.britannica.com/technology/robot-technology. [Accessed 30 April 2022].

[2] NASA, "MARS Exploration Rovers," [Online]. Available: https://mars.nasa.gov/mer/. [Accessed 18 02 2022].

[3] L. Goode, "Boston Dynamics' Robots Won't Take Our Jobs … Yet," WIRED, 26 October 2020. [Online]. Available: https://www.wired.com/story/get-wired-podcast-14-boston-dynamics/. [Accessed 30 April 2022].

[4] https://velodynelidar.com/blog/velodyne-creating-the-future-of-mobile-robots-with-nvidia/, "https://velodynelidar.com/blog/velodyne-creating-the-future-of-mobile-robots-with-nvidia/," 04 March 2021. [Online]. Available: https://velodynelidar.com/blog/velodyne-creating-the-future-of-mobile-robots-with-nvidia/. [Accessed 30 April 2022].

[5] Encyclopaedia Britannica, "bionics," 04 August 2021. [Online]. Available: https://www.britannica.com/technology/bionics. [Accessed 30 April 2022].

[6] D. Frank and J. Gorman, "How a Stick Insect Walks | ScienceTake," The New York Times, 30 October 2018. [Online]. Available: https://www.youtube.com/watch?v=24P0NzJdBig. [Accessed 31 May 2022].

[7] asknature, "Superhydrophobicity – The Lotus Effect," University of Colorado Boulder, 2011. [Online]. Available: https://asknature.org/resource/superhydrophobicity-the-lotus-effect/. [Accessed 30 April 2022].

[8] K. (. Li, "Predicting Stock Prices Using Machine Learning," Neptune Labs, 25 January 2022. [Online]. Available: https://neptune.ai/blog/predicting-stock-prices-using-machine-learning. [Accessed 30 April 2022].

[9] R. Mitra, "How-to-Use Machine Learning for Buying Behavior Prediction: A Case Study on Sales Prospecting," Medium, 18 April 2019. [Online]. Available: https://medium.com/omdena/how-to-use-machine-learning-for-buying-behavior-prediction-a-case-study-on-sales-prospecting-c496edd894cd. [Accessed 30 April 2022].

[10] DeepMind, "AlphaGo," [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far. [Accessed 08 01 2022].

[11] M. Fu, "What are some novel Go strategies discovered by AlphaGo Zero?," Quora, 2018. [Online]. Available: https://www.quora.com/What-are-some-novel-Go-strategies-discovered-by-AlphaGo-Zero?share=1. [Accessed 04 June 2022].

[12] B. Qin, Y. Gao and Y. Bai, "Sim-to-real: Six-legged Robot Control with Deep Reinforcement Learning and Curriculum Learning," *4th International Conference on Robotics and Automation Engineering (ICRAE),* 2019.

[13] L. Fang and F. Gao, "Type Design and Behavior Control for Six Legged Robots," *Chinese Journal of Mechanical Engineering,* vol. 31, 2018.

[14] H. Yamamoto, S. Kim and Y. Ishii, "Generalization of movements in quadruped robot locomotion by learning specialized motion data," *ROBOMECH Journal,* vol. 7, 2020.

[15] T. Haarnoja, S. Ha, A. Zhou and J. Tan, "Learning to Walk via Deep Reinforcement Learning," 19 June 2019. [Online]. Available: https://arxiv.org/pdf/1812.11103.pdf. [Accessed 3 May 2022].

[16] Adeept, "Adeept Hexapod Spider Robot Kit," [Online]. Available: https://www.adeept.com/adeept-hexapod-spider-robot-kit-for-arduino-with-android-app-and-python-gui-spider-walking-crawling-robot-steam-robotics-kit-with-pdf-manual_p0130.html. [Accessed 5 11 2021].

[17] W. Gastreich, "What is a Servo Motor and How it Works?," 27 August 2018. [Online]. Available: https://realpars.com/servo-motor/. [Accessed 26 May 2022].

[18] Adeept, "Hexapod Spider Robot Kit for Arduino-V4.0," [Online]. Available: https://www.adeept.com/learn/detail-43.html. [Accessed 26 May 2022].

[19] M. Walia, "Free And Open-Source Reinforcement Learning Frameworks and Projects Available Online For Machine Learning Engineers," 04 January 2021. [Online]. Available: https://medium.com/analytics-vidhya/free-and-open-source-reinforcement-learning-frameworks-and-projects-available-online-for-machine-93e5f47aeb61. [Accessed 04 May 2022].

[20] G2, "Best Physics Engine Software," [Online]. Available: https://www.g2.com/categories/physics-engine. [Accessed 07 May 2022].

[21] Unity Technologies, "ml-agents," 06 01 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents. [Accessed 08 01 2022].

[22] Unity Technologies, "Physics," [Online]. Available: https://docs.unity3d.com/Manual/PhysicsSection.html. [Accessed 05 07 2022].

[23] Unity Technologies, "ml-agents/release_10," [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/release_10_docs/docs/Getting-Started.md. [Accessed 12 02 2022].

[24] Lenovo, "ThinkPad P52," [Online]. Available: https://www.lenovo.com/de/de/laptops/thinkpad/p-series/P52/p/22WS2WPWP52?orgRef=https%253A%252F%252Fwww.google.com%252F. [Accessed 07 May 2022].

[25] Unity Technologies, "Download Unity," [Online]. Available: https://unity3d.com/get-unity/download. [Accessed 07 May 2022].

[26] GeeksForGeeks, "Top 10 Best Embedded Systems Programming Languages," 07 May 2019. [Online]. Available: https://www.geeksforgeeks.org/top-10-best-embedded-systems-programming-languages/. [Accessed 13 May 2022].

[27] Arduino, "What is Arduino?," [Online]. Available: https://www.arduino.cc/. [Accessed 12 02 2022].

[28] PlatformIO, "What is PlatformIO?," [Online]. Available: https://docs.platformio.org/en/latest//what-is-platformio.html. [Accessed 10 02 2022].

[29] Unity Technologies, "Robotics Simulation," [Online]. Available: https://unity.com/solutions/automotive-transportation-manufacturing/robotics. [Accessed 17 04 2022].

[30] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," 07 July 2017. [Online]. Available: https://arxiv.org/abs/1707.02286. [Accessed 08 May 2022].

[31] A.-L. Sanders, "What is Translational Motion?," 20 April 2022. [Online]. Available: https://www.infobloom.com/what-is-translational-motion.htm. [Accessed 08 May 2022].

[32] M. Golubs, "GitHub (makssyz / hexapod-unity)," 2022 April 2022. [Online]. Available: https://github.com/makssyz/hexapod-unity. [Accessed 08 May 2022].

[33] Unity Technologies, "Rotation and Orientation in Unity," [Online]. Available: https://docs.unity3d.com/2018.4/Documentation/Manual/QuaternionAndEulerRotationsInUnity.html. [Accessed 29 May 2022].

[34] P. Radhakrishnan, "What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network?," 09 August 2017. [Online]. Available:

https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a. [Accessed 09 May 2022].

[35] Unity Technologies, "GitHub (Unity-Technologies / ml-agents)," 15 December 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md. [Accessed 09 May 2022].

[36] E. (. name), "Why does Rigidbody 3d not have a gravity scale?," 03 June 2010. [Online]. Available: https://forum.unity.com/threads/why-does-rigidbody-3d-not-have-a-gravity-scale.440415/. [Accessed 13 May 2022].

[37] Pommaq, "GitHub / Pommaq / Adeept_Hexapod," 19 01 2021. [Online]. Available: https://github.com/Pommaq/Adeept_Hexapod. [Accessed 17 04 2022].

[38] One Network Enterprises, "What is SVOT?," [Online]. Available: https://www.onenetwork.com/supply-chain-management-resources/supply-chain-glossary/what-is-svot/. [Accessed 07 May 2022].

[39] Thomas Publishing Company, "An Introduction to G-Code and CNC Programming," [Online]. Available: https://www.thomasnet.com/articles/custom-manufacturing-fabricating/introduction-gcode/. [Accessed 07 May 2022].

[40] P. Ramdya, R. Thandiackal, R. Cherney, T. Asselborn, R. Benton, A. J. Ijspeert and D. Floreano, "Climbing favours the tripod gait over alternative faster insect gaits," 17 02 2017. [Online]. Available: https://doi.org/10.1038/ncomms14494. [Accessed 02 04 2022].

[41] M. Golubs, "GitHub (makssyz / hexapod-arduino)," 07 June 2022. [Online]. Available: https://github.com/makssyz/hexapod-arduino. [Accessed 09 June 2022].

[42] S. Bhatt, "5 Things You Need to Know about Reinforcement Learning," KDnuggets, 28 03 2018. [Online]. Available: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html. [Accessed 10 04 2022].

Table of Figures

Table of Code Snippets

Table of Tables