Department of Mechanical Engineering

FACULTY OF ENGINEERING AND DESIGN

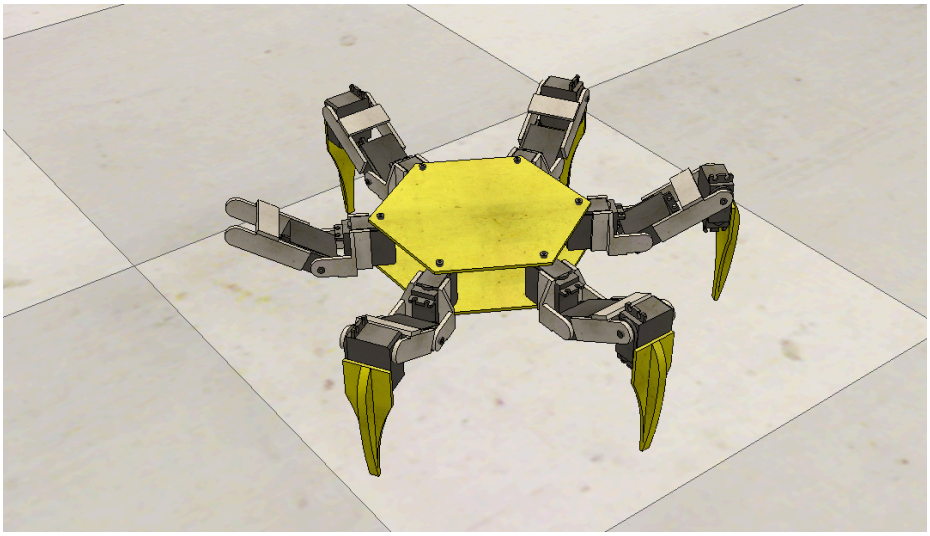
FINAL YEAR BEng PROJECT REPORT (ME30227)

Developing a machine learning algorithm to handle robotic defects

Hugo de Rohan Willner

4th May 2021

Word count: 11,231



"I certify that I have read and understood the entry in the Student Handbook for the Department of Mechanical Engineering on Cheating and Plagiarism and that all material in this assignment is my own work, except where I have indicated with appropriate references."

HDRW

Supervisor: *Ioannis Georgilas*

Assessor: *Sabina Gheduzzi*

Developing a machine learning algorithm to handle robotic defects

FINAL YEAR BEng PROJECT REPORT (ME30227)

Hugo de Rohan Willner

4th May 2021

Abstract

Most robotic controllers, in charge of the way in which robots take actions such as walking, are designed with the assumption that the robot's components will never break throughout its lifecycle. On the other hand, when humans or other animals break limbs, they learn how to adapt their behaviour to compensate for such changes. This project attempts to replicate this learning ability in legged robots. In this project, *Artificial Intelligence* methods were used to teach legged robots how to limp after suffering from a range of mechanical defects. Two machine learning algorithms were explored and tested: *Q-Learning* and *Genetic Algorithms*.

The first method, *Q-Learning*, was tested on a simplified problem. It was found that it would not be feasible to scale this method in order to tackle the original project brief.

The second method, based on *Genetic Algorithms* proved successful. This algorithm was implemented in a novel manor, where the motion of the robot's joints was controlled by compound periodic equations. The algorithm that was developed in this project can be used in two ways depending on the severity of the damage suffered by the robot. For lighter defects, the algorithm can adapt existing walking patterns, known as gaits, to deal with the change in dynamics. For more complex cases where simultaneous defects are applied on the robot, the algorithm can generate original gaits from scratch. In both of these cases, the gaits generated by the algorithm allow the damaged robot to walk as fast as before the damage was applied.

Acknowledgements

I would like to thank my supervisor, Dr. Ioannis Georgilas, for his great assistance and feedback along this assignment. His advice and supervision helped keep my focus in the right direction in this very explorative project.

I would also like to thank my tutor, Dr. Sabina Gheduzzi, who offered continuous help along my years of study at the University of Bath, especially during this unconventional final year of studies throughout the pandemic.

I would also like to mention the help of my fellow student Jahan Daya for his creative input in the final developments of this project.

I also want to thank Dr Alan Hunter and Dr Nigel Johnston for their great courses in Modelling Techniques, which inspired me to work on this project and continue my academic development in the field of computer science.

Finally, I would like to thank my friends and family for their support, without which this project would not have been possible.

Table of Contents

1. Introduction

The research summarised in this paper is part of a larger effort to develop what is known as *robotic resilience*. This area of research deals with the issue of unexpected mechanical or electronic failure of robotic systems, and the ways of dealing with it. Since the adoption of the word "robot" in the early 1920s in the English language [1], the importance of robotics in modern society has been continually increasing. By consequence, society's dependence on the reliability of such systems has become increasingly significant. Today, robots play a key role in extremely diverse industries, from flexible manufacturing processes that have been created thanks to adaptive robotics [2], to defence applications such as unmanned ground vehicles [3]. However, as pointed out by Turing award winning J. F. Corbatò:

> *"[We should not] wonder if some mishap may happen, but rather ask what one will do about it when it occurs"*

In the context of autonomous robotics, this quote implies that research is needed to develop control systems that can deal with scenarios that were not anticipated in the original design brief.

The area of research of this project was narrowed down to legged robots. This family of robotics was chosen as it presents interesting characteristics, many of which have famously been demonstrated by the Boston Dynamics research group [4]. Thanks to their isolated footholds, legged robots are notable for their superior capabilities in unpredictable terrain compared to wheeled systems [5]. Given that these robots are expected to achieve high levels of autonomy in variable conditions, which could potentially produce failures in sub-systems, it follows that robotic resilience research efforts should be focused on this family. Though the control algorithms developed in this project were designed for robots with any L number of legs, the machine learning process was developed using a hexapod robot, where L=6. This particular type of legged robot is ideal for this project as they have been the subject of many different research papers on the topic of robotic resilience in previous years. One of the key advantages of hexapods is their high static stability compared to lower L robots such as bipeds (L=2), as shown in the figure below.

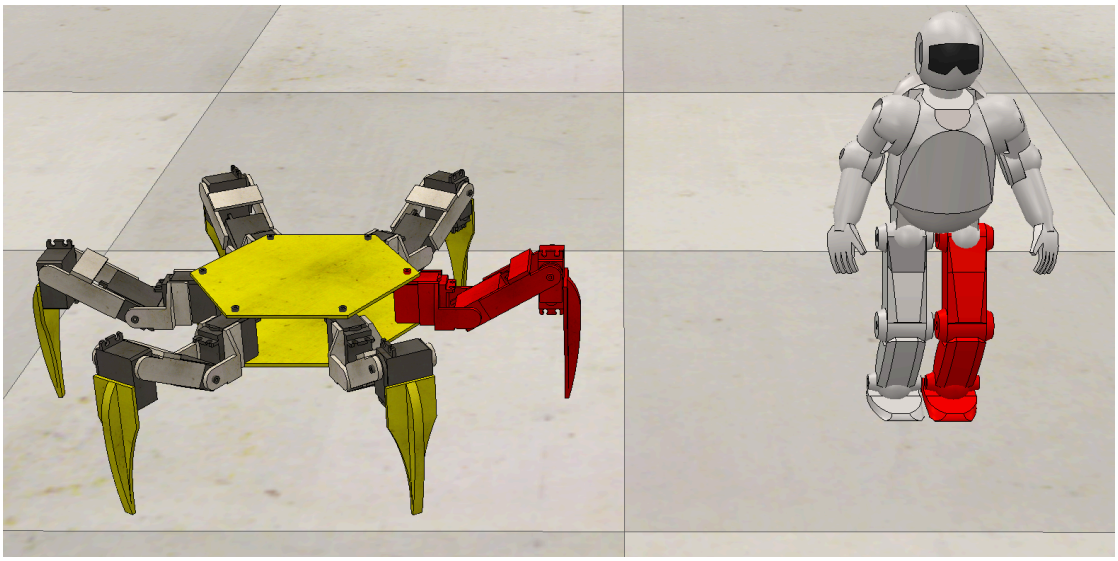


Figure 1 – Comparison of a hexapod and bipedal robot, and example failure modes. The hexapod will maintain static stability after removal of the parts shown in red.

Furthermore, due to their high number of actuators, hexapods allow for incremental failure modes, which lead to gradual dynamic responses that can be handled by the control algorithm. Bipedal robots, on the other hand, run the risk of complete failure when one of the actuators fail, which could not be handled by the control algorithm.

When discussing the dynamics of legged robots, one of the key concepts is what is known as the robot's gait, which can be defined as follows: "The gait of an articulated living creature, or a walking machine, is the corporate motion of the legs, which can be defined as the time and location of the placing and lifting of each foot, coordinated with the motion of the body, in order to move the body from one place to another." [6]. Using a combination of traditional control theory and Newtonian models, controllers can be designed to produce such gaits in legged robots. However, in the context of resilience, these controllers are severely limited, due to their static understanding of the system. The main approaches used to tackle the issue of resilience using traditional controllers have combined intensive testing and high levels of mechanical redundancy in robots [7]. Although these methods have proven their effectiveness in several sectors such as aeronautics, they also present several drawbacks, mainly associated with high costs of operation and design. Indeed, such methods require all failure modes to have been explored in the design phase of the system, and a solution found for each one of these modes. Furthermore, these methods require additional diagnostic systems to be added to the robot, in the form of sensors, in order to identify the failure mode once the robot is in operation.

Another, more recent approach to resilience has been to let the robot determine the best course of action on its own, learning a new behaviour based on its current condition. This type of computational method is known as *reinforcement learning*, which is part of the larger machine learning family. This method is commonly used in problems where an overall objective is defined, but the exact method of achieving the objective is not. It was decided to us this method as it appeared most suited to the aims of this project. Indeed, the objective is for the hexapod to walk a certain distance in an acceptable amount of time, but the ways in which it achieves this objective are not constrained. The methods used in this approach are highly influenced by *biomimetics*: the emulation of natural models for the purpose of solving complex human problems. Indeed, one trait of animal behaviour is the capacity to adapt to unforeseen situations such as broken limbs, learning to use their functional limbs in a new way. Exploring different types of computational methods, this project attempts to reproduce this behaviour with a hexapod robot.

Given this background research, the aims of this project were set as follows:

- Determine a suitable simulation environment
- Investigate different types of machine learning algorithms
- Implement and tune the selected algorithm in an innovative manner on the specific case of the hexapod robot
- Test the resilience of the algorithm by simulating different defects

2. Literature review

2.1. Simulation environment

Given the breadth of this area of research, a number of different simulation environments were considered for this project, many of which are discussed by Noori et al [8]. One of the key requirements for the simulation environment was the ability of interaction via Python scripts. Indeed, the overall project was written in Python in order to open the possibility of using the many available data science and machine learning libraries. A summary of the options of interest is outlined below:

- MATLAB SimMechanics. This Simulink based environment is designed for modelling and simulation of 3D dynamic systems. Its block-based design makes it the simplest software to interact with. Another advantage of this environment is that the model can be interacted with from any part of the MATLAB suite. This environment was successfully used for a similar research project [9].
- Gazebo simulator. This is an open-source 3D robotics simulator, maintained by Open Robotics. This simulator can use different physics solvers, such as ODE and Bullet, and can be interfaced with via Python. This environment was also successfully used for a similar research project [10].
- CoppeliaSim (formerly known as V-REP). Being an open-source 3D robotics simulator, this environment is quite similar to Gazebo. It can also use a range of different solvers and can be interfaced with via a Python-supported remote API (Application Programming Interface). Scripts can also be written within the software in a C-based language called Lua.

Having reviewed the relevant literature, CoppeliaSim was chosen for a number of reasons, including *a priori* knowledge of the software by the author and the existence of a hexapod within the software's library of default robots.

2.2. Gait generation and analysis

A great research effort exists in the area of gait generation for legged robots. Before looking into computationally generated gaits, traditionally generated gaits must first be investigated. In the earliest cases of gait generation, these gaits were bioinspired. Indeed, since legged robots usually have biological equivalents, such as insects being equivalent to hexapods, researchers have used these animals' gaits as inspiration for their robots.

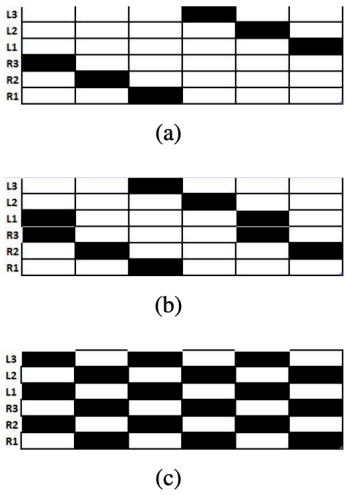


Figure 2 – Bioinspired gait diagram. White colour indicates that the foot is in contact with the ground. a) Metachronal Gait b) Ripple Gait c) Tripod Gait [11]

Shown in the figure above are a few examples of bioinspired gaits for a hexapod robot as described by Campos et al. [11]. One of the most popular gaits used in hexapod robotics is the Tripod Gait shown above. This is due to the gait's combination of stability and speed.

Given its simplicity, the method of bioinspiration is still in use today. Q. Chang and F. Mei [12], for example, have used image processing techniques to identify ant walking gaits and attempt to reproduce them on a hexapod robot.

However, though many types of gaits have been generated from human intuition and bioinspiration, the case for using computational generation methods such as machine learning is made stronger given the amount of theoretically possible gaits. Indeed, this number can be modelled by equation 1 below [13]:

$$N = (2L - 1)! \tag{1}$$

Where:

- N is the number possible of gaits
- L is the number of legs

In the case of a hexapod robot where L=6, there are more than 39 million possible combinations to choose from. For simple scenarios where the robot is functioning normally, human intuition can lead to very effective gaits, though there is always a possibility that even more effective gaits are yet to be discovered. However, when compensating for defects, the search becomes more complex. This is where machine-learning algorithms, which have proven to be great tools for analysing massive search spaces [14], have the most potential.

Gaits can be generally split into two subsets:

- Periodic gaits, where the position of joints is modelled by a cyclic function of time.
- Aperiodic gaits, where a function determines the incremental changes in joint positions at each timestep.

One of the advantages of using computational methods is that this allows for gaits to be generated dynamically, i.e., in an aperiodic manner. It should be noted that periodic -or cyclic- gaits are best suited for flat, predictable terrain. However, so called "free" or aperiodic gaits, first generated in the early 90s [15], are capable of handling unpredictable terrain.

2.3. Algorithms and associated models

Many different types of machine-learning algorithms have been used in an attempt to solve the gait generation problem. Though these algorithms tend to be customised to their specific task, a number of similarities can be found in their approaches. A summary of the methods that were considered is presented below:

2.3.1. Central patterns generators

Inspired by their natural counterpart, these simple artificial neural networks are based on a feed-forward central controller that generates rhythmic motion without the need of sensory feedback. These models tend to create periodic gaits, however, in the latest developments using this method, multiple CPGs are combined, so that learning behaviours can be produced [16].

2.3.2. Reflex models

"Reflex models" are the dynamic equivalent of CPGs, where local controllers based in each leg are represented by interdependent agents in the machine learning algorithm. These models have successfully been used for gait generation [17]. In more recent research projects, "Reflex models" and CPGs have been used in combination to produce more resilient gaits, able to deal with deficiencies in legs [18].

2.3.3. Q-Learning

As one of the most common methods used to tackle reinforcement learning problems in general, Q-Learning has been used in many different research projects for gait generation. Essentially, this method uses a feedback loop where an Agent takes an Action based on its current State, and receives a Reward based on the action taken. This loop is iterated through at each step of the simulation. An in-depth explanation of this method is provided later in section 3.1. In the context of gait generation, the essential challenge in using this method is to reduce the size of the State and Action space. Indeed, in an ideal world, the complete State space should be a list of every possible combination of actuator positions. Additionally, the Action space should be a list of every possible permutation from one State to another. However, this naïve approach is far too computationally expensive, and so alternative methods have been used.

One approach, devised by M. Shahriari [9], reduces the State space to three locations of the leg-tip, as shown in the figure below. Using this approach, the Action space is simply the movement from one State to another. These movements are achieved using inverse kinematics, which can be simply implemented in most simulation environments. This approach was used and further developed by Ing. R. Woering [19], who was able to achieve more complex behaviours such as jumping.

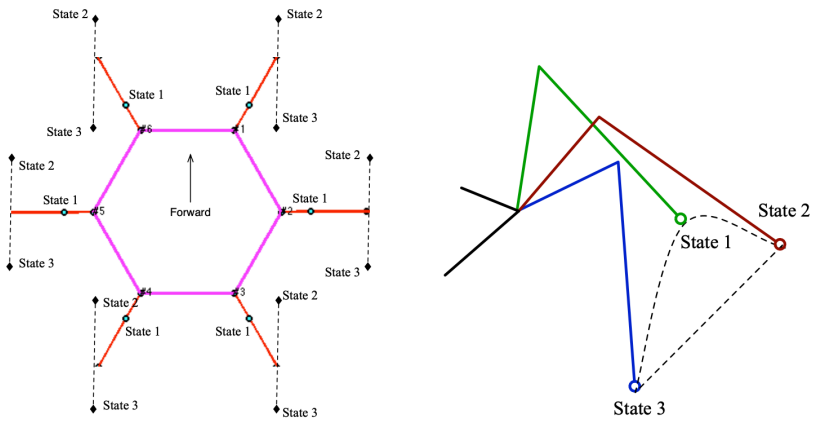


Figure 3 – Simplifying the State-Action Space in Q-Learning by using three possible states per leg

Another, more popular approach is to bypass the problem of large State-Action spaces by using neural networks to approximate the Policy function. This method, known as Deep Q-Learning, has yielded extremely impressive results in many gait generation projects, from bipeds [20], to hexapods [21]. However, it was deemed too complex to be used within the scope of this project.

2.3.4. Genetic Algorithms

A genetic algorithm is a search tool, inspired by Darwin's theory of natural evolution. The algorithm loops through a process of fitness testing, selection and generation of populations, until a certain objective is met. An in-depth explanation of this method is provided later in section 4.1. Each member of the population is defined by a set of adjustable parameters, analogous to an individual's genes. These algorithms are used in many different machine learning problems. A number of different research projects have used this tool in the context of gait generation. The overall structure of the algorithms does not vary excessively from project to project. Indeed, the main differentiation between different implementations of this method is found when selecting what the population parameters represent.

One approach, used by K. Gupta et al. [22], is to define the robotic gait as a finite repeated sequence of N translations from one state to another, where states are defined as leg tip positions. This method also uses inverse kinematics to move the legs from one state to another. Each member of the population is given its own sequence of positions to loop through, and each generation optimises the sequences more and more. Using this approach, the genetic algorithm simply searches for the optimal sequence of translations for each leg to achieve an acceptable gait.

Another approach, used by J. Kon [10], is to define the robotic gait as a combination of time dependent equations to define the angle of each actuator. The exact equation is shown below, where ϑ_i is the angle of the i^{th} actuator.

$$\theta_i(t) = A_i sin(B_i t + C_i) \tag{2}$$

For each actuator, the A, B and C coefficients are what differs from one member to another. In a similar fashion of population generation and testing, the algorithm searches for the optimal combination of coefficients to produce the desired gait.

2.4. Potential areas of innovation

Having concluded the initial literature review, the first two objectives set out in the introduction had been completed, i.e., selecting an appropriate simulation environment and investigating different types of machine learning algorithms in the context of gait generation. The two algorithms that were selected for further investigation were the Q-Learning and Genetic algorithms. The further investigation was to be undertaken in the form of technical testing of the algorithms. However, given that the third aim of this project was to implement these algorithms in an innovative manner, initial ideas of innovation were explored at this stage.

In the case of Q-Learning, the plan for innovation was to define a new type of State-Action space, different to that proposed by M. Shahriari [9]. Indeed, it was hypothesised that defining the State-Action space as a range of selectable angles for each actuator would enable more advanced and complex behaviours for hexapod, enabling it to deal with a larger number of defects. One of the risks associated with such a method however was the high computational power needed to deal with this larger State-Action space.

In the case of the Genetic algorithm, the plan for innovation was to advance on the work done by J. Kon [10], by inserting cosine and constant terms to equation 2 shown above. The insertion of these terms would allow for more complex motion profiles for each joint, which could potentially allow the hexapod to deal with more complex failure modes.

3. First investigation

The first method selected for further investigation was Q-Learning. In this section, the method, results, and decision for moving onto the Genetic algorithm are explained. Given the complexity of the Q-Learning algorithm, the plan for this part of the project was to test the algorithm on simple problems such as a basic computer game in order to determine the feasibility of implementing it for the hexapod problem.

3.1. Q-Learning: Method Overview

The Q-Learning algorithm is based on a few key concepts, who's relationships are outlined in the diagram below. At each timestep, the Agent takes an Action which affects its Environment. The Agent then receives feedback in the form of a State and Reward function. In reinforcement learning, the function used to determine what Action should be taken given the State and Reward is called a Policy. The loop shown below continues until a user-defined limit is met, such as the Agent reaching an objective. The algorithm then restarts a new Episode.

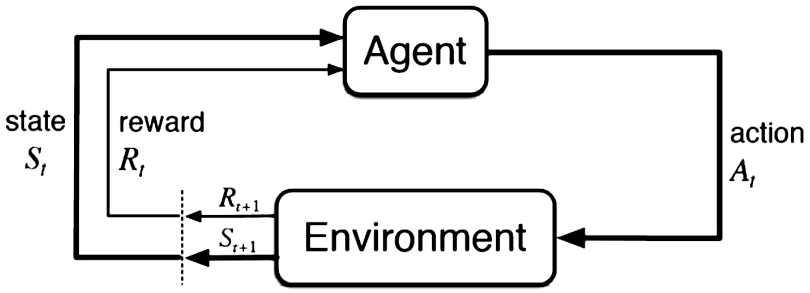


Figure 4 – A Reinforcement Learning Loop

The main tool used in the Policy for this method is a Q-Table, an example is shown below. This table is essentially used as the Agent's "memory", by keeping a record of the Rewards that the Agent receives for taking a particular Action in a particular State.

Table 1 – Q-Table Structure

	A_1	A_2	$A_{...}$
S_1	$Q(S_1,A_1)$	$Q(S_1,A_2)$	$Q(S_1,A_{...})$
S_2	$Q(S_2,A_1)$	$Q(S_2,A_2)$	$Q(S_2,A_{...})$
$S_{...}$	$Q(S_{...},A_1)$	$Q(S_{...},A_2)$	$Q(S_{...},A_{...})$

The equation used to fill this table is known as the Bellman equation. A simplified version of this equation is shown below (eq. 3). It is a sum of the Reward r_t received for a given State and Action pair, as well as the maximum possible Reward that the Agent could receive by taking a subsequent action from the new State.

$$Q_{new}(s_t, a_t) = [r_t + \gamma * \max_a Q(s_{t+1}, a)] \quad (3)$$

The version of the Bellman equation used for this project, shown in eq. 4 below, takes the previous value, Q_{old} into account, when iterating through. The α coefficient is used to produce a weighted average of the old and new values.

$$Q_{new}(s_t, a_t) = \alpha * \left[r_t + \gamma * \max_a Q(s_{t+1}, a)\right] + (1 - \alpha) * Q_{old}(s_t, a_t) \quad (4)$$

Finally, the method used for this project also incorporated a way of avoiding falling into local maxima in the action space. Indeed, one of the tradeoffs one must make in machine learning is between exploitation and exploration. If the Agent were only using eq. 4 in terms of Policy, then it would only take actions that have proven to give good results, this is known as exploitative behavior. However, explorative behavior is also very valuable as it allows the Agent to discover new Actions that might produce even higher results. The method used to deal with this challenge is called an Epsilon-greedy strategy, a common method used in Q-Learning algorithms. Using this method, the Agent will have a tendency to explore random actions instead of following eq. 4. A pseudocode for this method is shown below:

```
p = random()

if p < Ɛ :
        take random action
else:
        take current-best action
```

Figure 5 – Epsilon-greedy pseudocode

$\mathcal{E}$ is set to start at an initial value and decays as the algorithm loops through, so that the actions start as very explorative and tend to a more exploitative behaviour.

3.2. Investigating the potential of the algorithm

3.2.1. Applying the method to Frozen Lake

In order to assess the feasibility of applying this algorithm in the context of gait generation, the method was first applied to a simple computer game called Frozen Lake. An image from the game is shown in the figure below. The game is part of the OpenAI Gym library [23], a toolkit composed of different environments for developing reinforcement learning algorithms. If the investigation proved successful, the algorithm could then be integrated with CoppeliaSim via the Gym API (Application Programming Interface).

Figure 6 – Image taken from Frozen Lake. The player, shown in red at the start S, can move up, down, left, right, and needs to get the goal G, without falling through the holes H.

The following steps were taken in order to apply the Q-Learning method to this problem:

- The State is the player's current position in the game.
- The available Actions are moving up, down, left, right
- At each step, the Reward is set to 1 if the goal G is reached, and 0 in any other case. The game ends when the agent falls through a hole or reaches the goal, and the algorithm starts a new episode.

3.2.2. Frozen Lake Results

The algorithm was tested by tuning the different hyperparameters: the learning rate α, the discount rate γ and the greediness ε. In order to achieve consistent results, the algorithm was run for 10,000 episodes for every parameter test. By process of trial and error, the optimal parameters were found for this particular problem, their values were:

- α : 0.1
- γ : 0.99
- ε : initial value of 1 and decaying by 0.001 per episode.

The average reward per episode using these parameters is shown in the figure below:

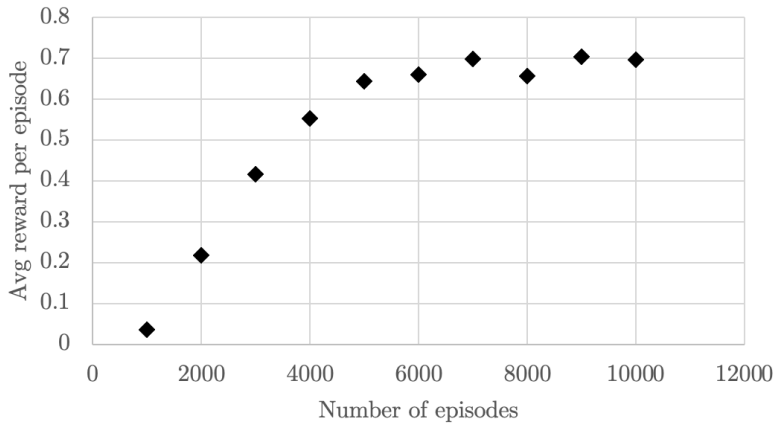


Figure 7 – Success rate of the Q-Learning algorithm from 0 to 10,000 episodes using the optimal hyperparameters found by process and trial and error

Figure 5 shows that, though the average reward first start increasing linearly, a plateau is reached at 0.71 rewards per episode. This means that the Agent successfully beats the Frozen Lake game with a 71% percent success rate at the end of its training by Q-Learning.

Though these results were promising, a number of areas of concern were found during the testing of this method, most of which were related to computation time. To explore this issue, the computation time for 10,000 episodes was recorded for 10 runs using the optimal parameters discussed above. The results are shown in the figure below:

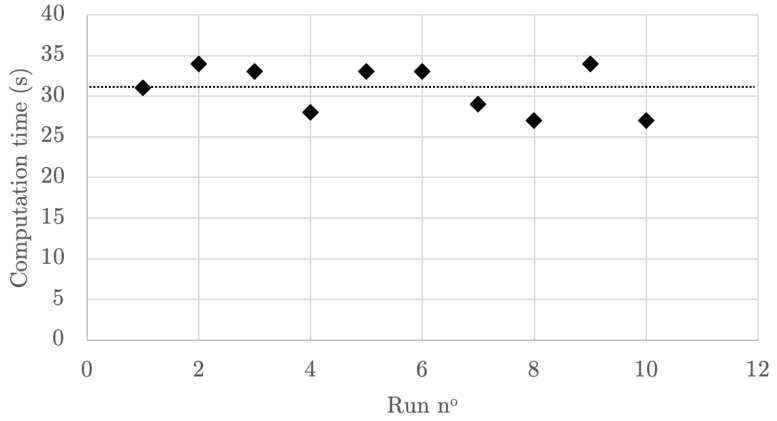


Figure 8 – Computation time of the Q-Learning algorithm on Frozen Lake for 10,000 episode runs

The average computation time for these 10 runs was 31.2s. In order to extrapolate an estimate of the time needed to solve the Hexapod problem using this approach, the average steps per successful episode needed to be analysed. This data is shown in the figure below:

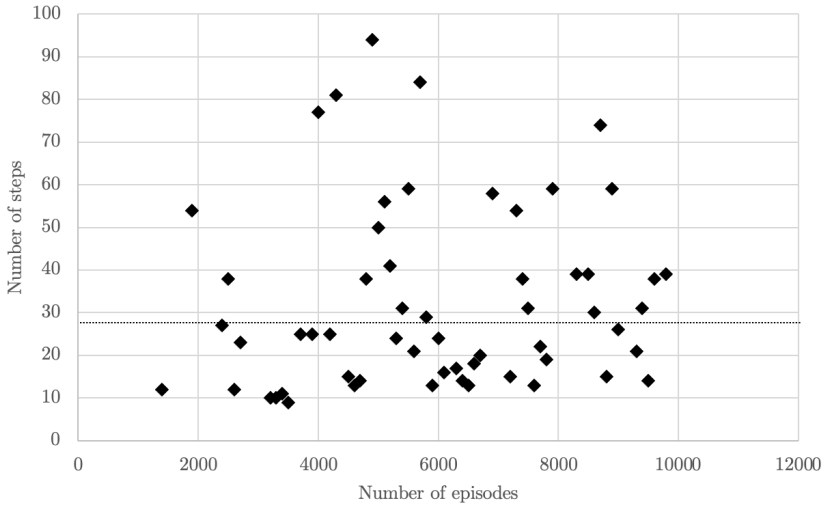


Figure 9 – Number of steps per successful episode from 0 to 10,000 episodes using the optimal Q-Learning parameters

It should be noted that, by comparing figures 5 and 7, even though the success rate of the Agent using the Q-Learning algorithm increases linearly between 2000 and 6000 episodes, the number of steps taken in these episodes does not seem to follow a clear pattern. Although this result can be initially surprising, as it would be expected that the number of steps needed to win the game would reduce as the Q-Table was iteratively updated, it can be explained by the Reward function that was used. Indeed, the Reward function did not take the number of steps into account: an Agent taking 10 steps would get the same Reward as an Agent taking 100 steps to complete the game. This shows the importance of designing the correct reward function, which was focused on later in the project.

With these results in hand and a better understanding of the core functions used in Q-Learning, the feasibility of scaling this method up to the Hexapod problem could be assessed.

The average number of steps per successful episode was found to be 32,3. Given that the hexapod was expected to walk for 10 seconds per episode, an estimate of around 2000 steps per episode was be calculated, using a simulation step of 50ms (standard step in CoppeliaSim). Hence, by extrapolation, the average runtime to solve the Hexapod problem was estimated to take around 30 mins. However, this estimation did not take into account the computation time needed to simulate the hexapod motion at each step, as well as the data transfer time used by the CoppeliaSim API. Furthermore, given the number of degrees of freedom of the hexapod, the State-Action space would be much larger than that of the Frozen Lake game. This first estimate was therefore very optimistic, and a much longer processing time of at least 24h would realistically be achieved given the available computational power and storage access speed.

Given these results, it was found that although this method showed promising results, it would be unfeasible to implement in more complex problems such as the Hexapod problem given the short timeframe of this project.

4. Second investigation

Having analysed the results of the first investigation, the second reinforcement learning algorithm was investigated. This latter algorithm only processes data once at the start of each simulation, instead of running at every time step like the Q-Learning counterpart. Hence it was expected to be less computationally expensive and could therefore be directly implemented on the Hexapod problem.

4.1. Method: Genetic Algorithm

4.1.1. Genetic Theory

A genetic algorithm is a biomimetic search tool that has long been used in solving reinforcement learning problems, a basic structure is shown in the figure below. The algorithm reflects the processes of natural selection, where the fittest individuals are selected to produce the subsequent generation. The algorithm is composed of a population of members that are characterised by their "genes", a collection of pre-determined parameters. Each member is tested against the same characteristics and is given a score based on its performance. This is known as fitness test and is analogous to the reward function in the Q-Learning method. A new generation of members is then produced, based on the previous generation's scores and parameters. A number of different methods were explored to complete this step of the algorithm, these are investigated in more detail in section 4.1.4. Finally, the new generation is then tested and scored. This cycle continues until a certain objective is met, such as a generation count, or when a member of the population obtains a desired score.

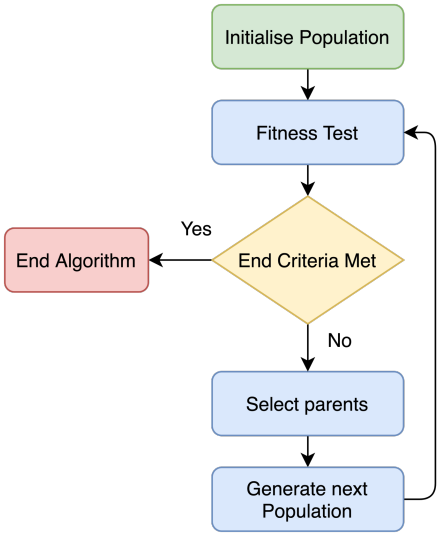


Figure 10 – Genetic Algorithm Flowchart

4.1.2. Overview: Applying GA to the Hexapod problem

The parameters used to define a member in the Hexapod population are associated with the motion of the Hexapod's 18 servomotors. Each of the six legs of the Hexapod have three degrees of motion, as shown in the figure below.

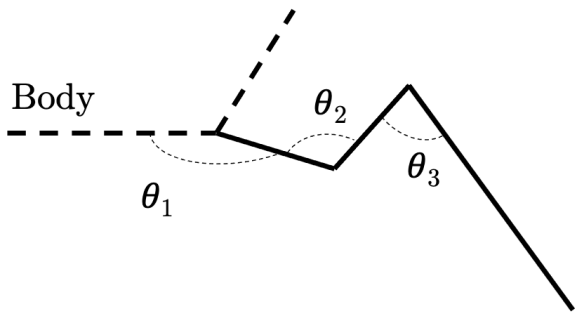


Figure 11 – Degrees of motion of the hexapod leg

A dynamic controller was set up in CoppeliaSim for each servomotor. Each of these angles is controlled by the following equation:

$$\theta_i(t) = A_i sin(B_i t + C_i) + D_i cos(E_i t + F_i) + G_i \quad (2)$$

Coefficients A and D give the amplitude of the motion profile, B and E relate to the frequency of the motion, C and F are phase offsets, and finally G is a general static offset for each joint.

The A, B, ..., G coefficients in the joint motion equation are what varies from member to member. A member is therefore defined by 126 different parameters, who's structure is defined in the following section. During the initialisation phase, each of these parameters is randomly generated for each member of the population. The appropriate ranges for each parameter were determined by a process of trial and error during the development stage of this project and are later discussed in the Results section. As explained in the Literature Review, the reasoning behind choosing this equation was that promising results were found in previous research using a sin only function. It was hypothesised that including further cos and constant parameters would allow for more complex motion profiles, which may lead to more capable gaits for high numbers of failures in the hexapod.

4.1.3. Data Management

One of the key challenges in creating the algorithm was designing an efficient data stream in the Genetic Algorithm loop. Given that one of the priorities in machine learning is to reduce the processing time within each learning loop, the main concern when developing the overall structure of the algorithm was to limit the number of data transfers from the GA Script to CoppeliaSim and vice versa. One of the ways that this was achieved was by running all of the motion scripts as well as the position capture from within the CoppeliaSim architecture, instead of sending new target positions over the API for each joint at each simulation timestep. The final structure of this data stream is shown in the figure below

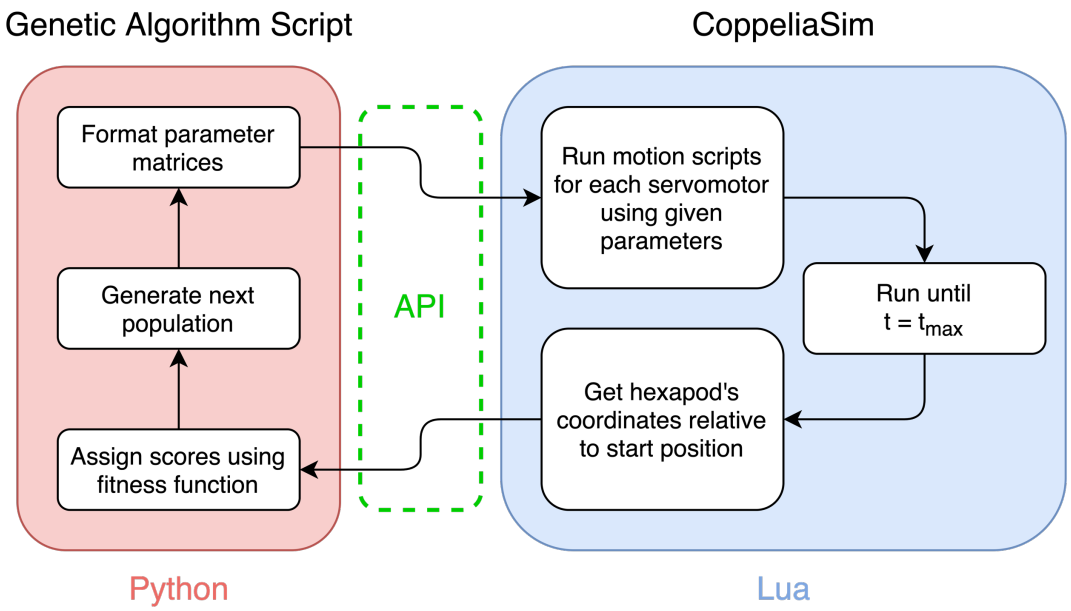


Figure 12 – Data Flow Overview

For a population of size N, the GA script prepares N matrices to be formatted and sent over the CoppeliaSim API. Each matrix is three-dimensional. The first dimension represents the 3 different joints forming a leg shown in figure 9. The second dimension represents each of the hexapod's six legs. Finally, the third dimension represents the A, B, C, ... parameters in equation 2. The GA script was written in Python as this language is commonly used for ML and general data science related projects.

Once the parameters are received, a CoppeliaSim script written in Lua reformats these parameters and uses them as inputs, setting the motion of each joint. At the end of the test duration, the hexapod's coordinates are sent back to the GA script. These are then used to calculate the fitness score and generate the next population to be tested.

4.1.4. Initialisation

Two modes of initialisation were used in this project:

1. For heavy defects, such as 2 or more entire legs being impaired, highly original gaits would need to be produced. Hence, in these cases, the hexapod population was initialised with randomised parameters before looping through the genetic algorithm.
2. For lighter defects, such as a single joint being impaired, it was expected that the solution could be a variation of a pre-existing gait. In this case the population could be initialised with predefined parameters, before looping through the genetic algorithm. Given the advantages discussed in the Literature Review, the Tripod gait was selected. The parameters inputted in equation 2 to produce a tripod gait are shown in the table below.

Table 2 – Coefficients used to initialise a tripod gait

Legs	*Joints*	*A*	*B*	*C*
1, 3, 5	1	π/4	8	π
	2	0.2	8	5π/2
	3	1.3	0	3π/2
2, 4, 6	1	π/4	8	0
	2	0.2	8	3π/2
	3	1.3	0	3π/2

4.1.5. Genetic Recombination

One of the key steps in a genetic algorithm loop is the genetic recombination function, where the genetic information of previous generations is passed onto the next. Two methods were explored in this project, but all were based on the principle of genetic crossover, which is inspired by the crossover that happens at the chromosomic level during biological reproduction. A single point crossover is shown below:

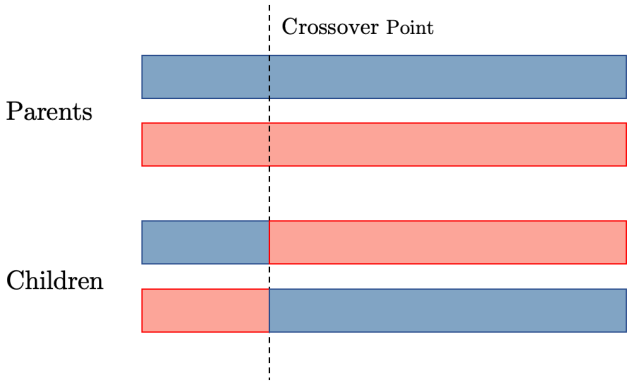


Figure 13 – Example of a single point crossover, a method of genetic recombination inspired by biological reproduction.

In a single point crossover, a point on the parent's chromosomes is randomly selected. This is known as the "crossover point". Parametric information contained either side of this point is swapped when being passed onto the children. This type of crossover produces two offspring, carrying genetic information from both of their parents.

One of the challenges of implementing this method in the Hexapod GA was deciding what the chromosome represented in terms of parameters. Indeed, the way in which this method is implemented in the algorithm is dependent on the data structure used to represent these parameters. A chromosome could represent the entire 126 parameters per member, in which case the crossover would happen once per member. It was decided however that a member would have a chromosome per joint, i.e., 18 in total. A diagram of the implemented crossover is shown below.

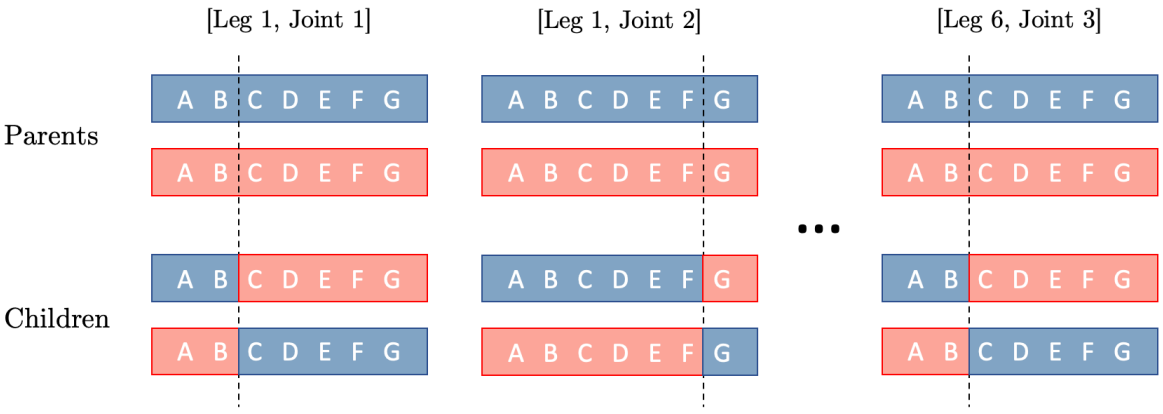


Figure 14 – The single point crossover method implemented in this algorithm. Each chromosome is composed of the 7 different motion coefficients per joint.

The second crossover method explored in this project is known as uniform crossover. In this simple type of crossover, every bit of information in the chromosome is chosen from each parent with equal probability. To implement this method, each three-dimensional parameter matrix is flattened into a single dimension, forming a single long chromosome. This new vector of size 126 can then be iterated through for each parent, using the method described above. A pseudocode for this method is shown below.

```
p1 = first parent's parameters
p2 = second parent's parameters
c = child's parameters

loop :
        x = random()
        if x > 0.5:
                c[i] = p1[i]
        else:
                c[i] = p2[i]
```

Figure 15 – Pseudocode for the implementation of uniform crossover in the Hexapod GA

NB: using this method, the selection of each parameter is random, however the available parameters are not. For example, the B coefficient for second joint of the 4th leg of a *child* hexapod will be randomly chosen between the B coefficient for second joint of the 4th leg of either the *first* or *second* parent.

4.1.6. Mutation

After genetic recombination comes the mutation function in the genetic algorithm. This function has the same objective as the Epsilon-greedy strategy used in Q-Learning, which is to avoid trapping the evolution of the population into a local maximum. This method is inspired by biological mutation, which alters one or more gene values in a chromosome from its initial state. To implement this method in the Hexapod GA, a simple function was written, and its pseudocode is shown below.

```
r  = randomly initialised parameters
m = parameters to be mutated

loop :
        x = random()
        if x < mutation rate :
                m[i] = r[i]
```

Figure 16 – Pseudocode for the implementation of the mutation method in the Hexapod GA

This function iterates through the child's parameters and assigns random parameters at a frequency that depends on the mutation rate. Setting this mutation rate is a trade-off between exploration and exploitation, that is found in many different types of reinforcement learning algorithms. By trial and error, it was found that a mutation rate of 5% gave satisfactory results.

4.1.7. Fitness function

In a genetic algorithm, the fitness function takes an arbitrary form that is set by the user. For this project, the main objective was to produce a gait that enabled the hexapod to walk in the desired direction. The function for this algorithm was initially defined as the x coordinate of each member at the end of the testing period t_{max}, as shown below. More advanced fitness functions were later defined, taking other coordinates into account to avoid sideways movement.

$$score_1 = x(t_{max}) \tag{3}$$

The fitness function then ranks the members of the population in terms of their score in order for them to be processed by the genetic recombination function.

4.2. Preliminary results: parameter tuning

During the development of this algorithm, initial testing was done with fully functional hexapods and randomised initial parameters. This was done in order to tune all the parameters and select the appropriate methods of genetic recombination, mutation, and fitness testing described above. Each decision was made with the aim of producing a stable and efficient gait in limited computation time. The algorithm was then tested on hexapods with a range of different defects to assess its resilient capabilities.

4.2.1. Initial results using a simple recombination function

The first functioning version of the GA used a basic genetic recombination method where the highest scoring half of the population was kept, and the lowest scoring half was replaced by randomly generated hexapods. The first results are shown in the figure below.

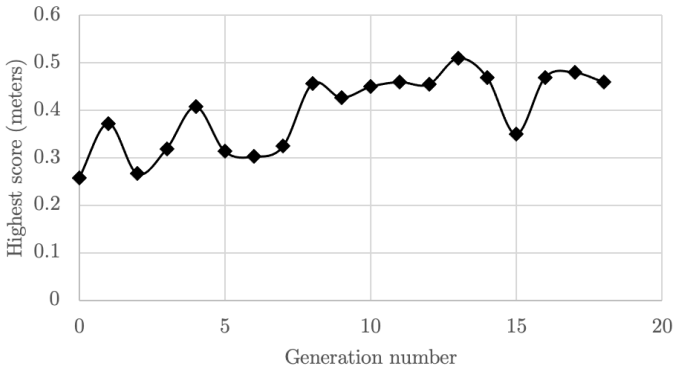


Figure 17 – Initial results of the GA presented as the highest score of the population for 18 generations

The figure above shows the highest score of the tested population for an initial run of 18 generations. The average score instantly rises to a positive value, signifying forward movement was achieved. This can be explained by the algorithm removing any low scoring hexapods that produced a "reverse gait". These initial results showed the potential of the algorithm. However, they also showed the limitations of using such a basic recombination method, as the scores quickly plateau to a value of 0.47 m.

During this preliminary testing period, it was found that an optimal population size of 8 members would need to be used. This size was determined as a trade-off between

higher opportunities of gait exploration for a larger population and a faster computation time for a smaller population. The size 8 population is shown in the figure below.

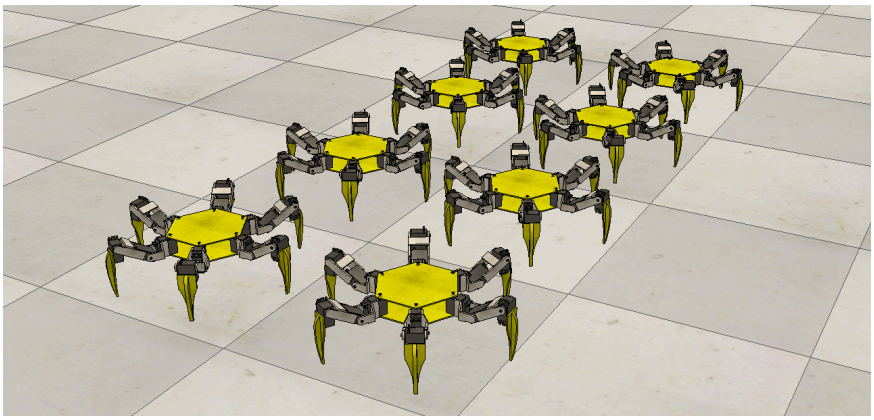


Figure 18 – Hexapod population of size 8 in CoppeliaSim. This population size was found to be an ideal trade-off between computation time and gait exploration.

4.2.2. Adjustments to the fitness function

The results of this initial testing phase also provided key insights that would be needed to adjust the fitness function.

The most noticeable issue was that unstable behaviours such as the hexapods "flipping" forwards were being rewarded by the fitness function. Indeed, as explained in the Method section, the fitness function initially assigned scores based on the distance travelled at the end of the testing episode. However, the hexapods that flipped over at the end of the episode would travel slightly further than the rest of the population and would therefore be assigned the highest score. This behaviour would be passed onto the next generation via the parameter matrix.

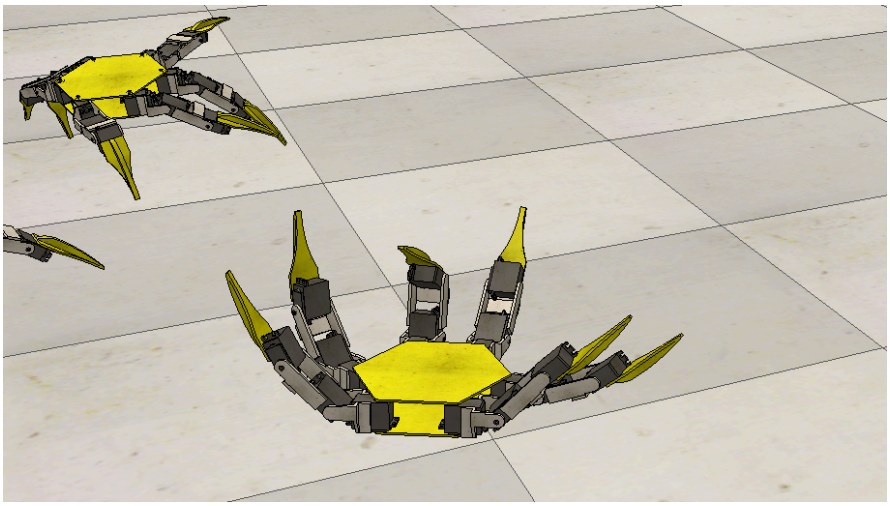


Figure 19 – Hexapod "flipping" behaviour. The hexapod in the foreground has travelled further to the right than the other member in the background.

This type of behaviour is an example of a local maximum in the algorithm, where the population would not be able to achieve any higher scores after this manoeuvre since all members would be stuck upside down.

The fitness function was therefore changed to accommodate for this: an inverted hexapod would now be assigned a score of -10 to ensure that it's behaviour would not be passed on.

```
x = x distance at t_max

if (hexapod is flipped):
        score = -10
else:
        score = x
```

Figure 20 – Updated fitness function pseudocode, ensuring that "flipping" behaviour is not rewarded

Finally, another method was implemented to avoid local maximums. During this testing phase it was observed that some hexapods were exploiting the fact that the score was based on the x location at the same t_{max} for every generation. Unstable behaviours were being developed by the hexapods, where the gait would function correctly until t_{max}, but would not be sustainable after that time. It was found that such behaviours could be avoided by alternating the values of t_{max} by ± 1s for every other generation.

An optimal value of 10 seconds was found for t_{max}, which was found to be a reasonable compromise. It was found that small values of t_{max} would generate irregular gaits that would not be sustainable for longer operation times, however larger values of t_{max} incurred longer computation times, which were to be avoided as stated in this section's introduction.

Given this definition of the fitness function, the tripod gait, as described in table 2 in the Method section, is acquires a score of 1.43, which means that a hexapod using this gait will travel 1.43m in during the 10 second test.

4.2.3. Comparing crossover methods

During this initial testing period, the two genetic recombination techniques described in the Method section were also compared. This was done by running the algorithm for 225 generations using both of these techniques and recording the score of the highest performing member per generation.

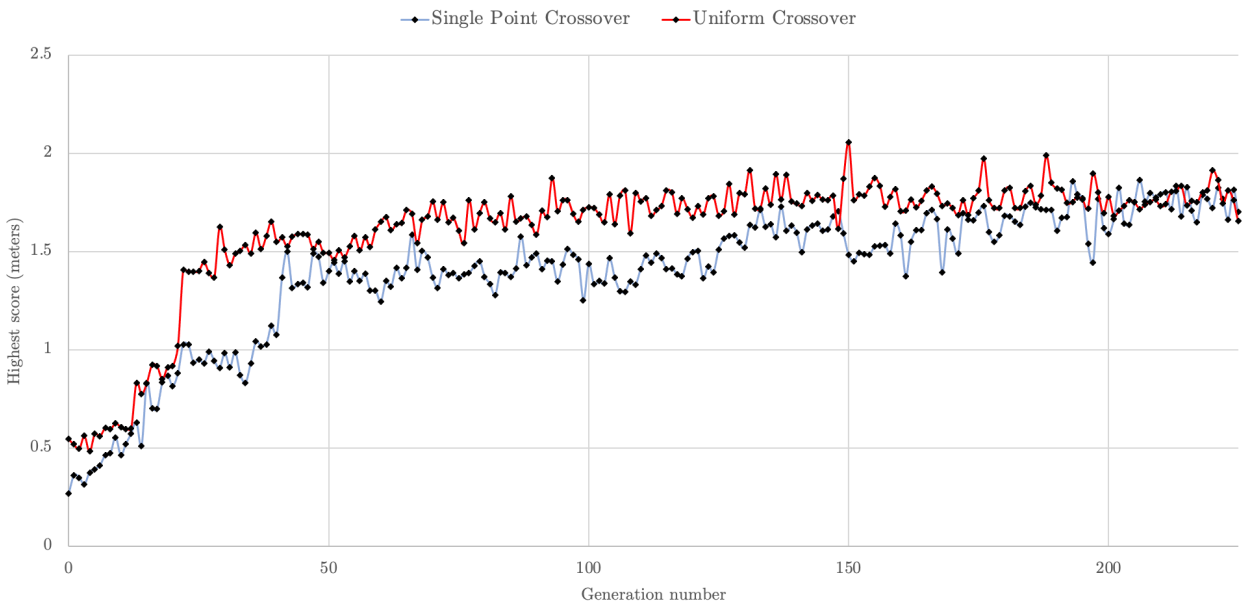


Figure 21 – Comparison of the highest scoring member per generation using single point and uniform crossover methods

The figure above shows that both techniques seem to lead to a common asymptote of around 1.75m. However, the hexapod population using the uniform crossover method consistently outperforms the single point population. This is especially noticeable in the early generations, from gen 25 to 45. Given that computation time is a key requirement for this algorithm, the uniform crossover method was chosen for use in the final algorithm.

4.2.4. Associated uncertainties

It should be noted that when tuning the algorithm by selecting the hyperparameters such as the mutation rate, or by selecting an appropriate crossover method, a level of uncertainty is associated with each decision. Indeed, as explained in the Method section, there is an associated randomness with the score results of the Genetic Algorithm. Indeed, this randomness is introduced during the initialisation of the population, and during the mutation stage in the algorithm loop. Hence when comparing two parameters, such as the crossover methods shown above, comparing the results of a single run per parameter may lead to uncertainties. An ideal process would be to run the algorithm multiple times for each parameter, and compare the average outcomes, in a Monte-Carlo style aggregation. However, such a process would have been too time consuming given the short time frame of this project.

4.3. Final results: resilience testing

Having finished tuning the parameters of the Genetic algorithm, its performance could then be assessed by applying it to hexapods presenting a range of different defects. As noted in section 4.2.2., the standard tripod gait, to which the resulting gaits were compared, achieves a score of 1.43m.

4.3.1. Light defects

In the case of light defects – as discussed in the Method section – the hexapods population was to be initialised with parameters enabling a standard tripod gait.

An example of a defect that was tested was the partial removal of a front leg, as shown in the figure below.

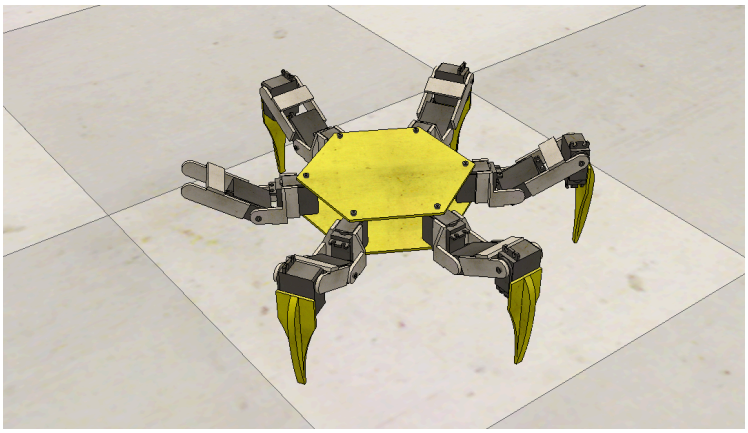


Figure 22 – Example of a light defect applied to a hexapod, the partial removal of a front leg.

First of all, the motion of the hexapod was run with no change in the original parameters, to observe how badly the gait was affected by this defect. When running the standard tripod gait on a hexapod with the defects described above, the score achieved is 0.75m, or 52% percent of the score achieved with a fully functional hexapod. Furthermore, the tripod gait is rendered highly unstable by this defect, as shown in the figure below.

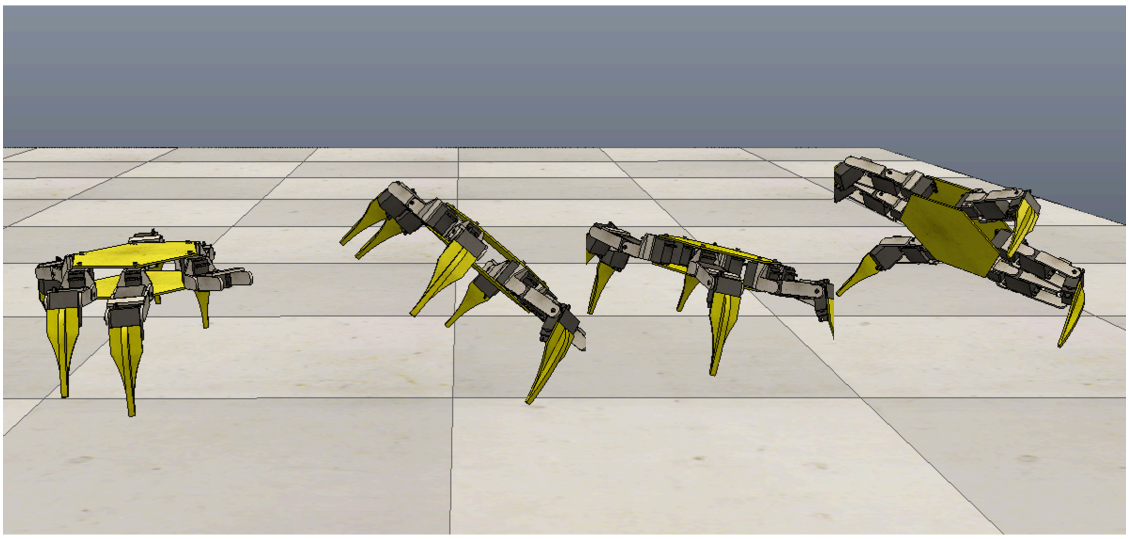


Figure 23 – "Rocking" motion of a damaged hexapod following the original tripod gait designed for a fully functional hexapod (movement is left to right)

The figure above shows four frames of the simulation running the standard tripod gait on the damaged hexapod. A "rocking" motion is observed. This is due to the hexapod shifting its weight onto a non-existing leg during the walking sequence.

The genetic algorithm was then run using the parameters selected in the previous stage, and the score results are shown below.

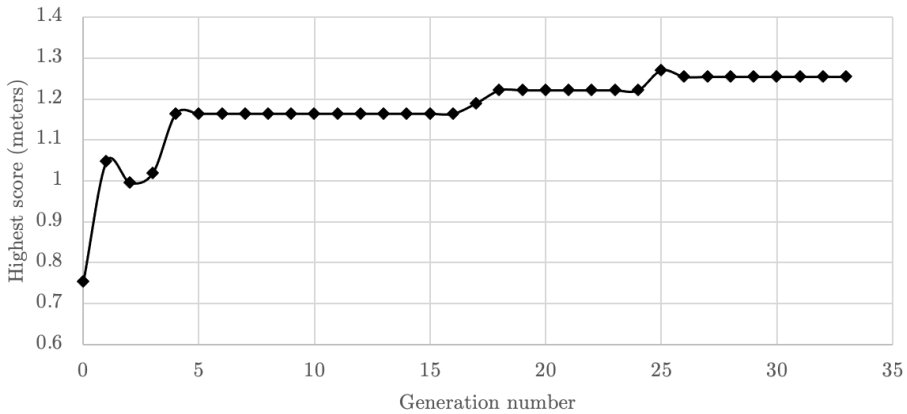


Figure 24 – Results of the GA applied to a hexapod with a partially removed leg, starting from the standard tripod gait.

The figure above shows the evolution of the hexapod score for each generation of the genetic algorithm. As explained above, the initial score is 0.75m. However, in only 4 generations the gait is adapted and stabilised. The motion of this computationally generated gait is shown in the figure below. By the 4th generation, the score achieved is 1.16m, or 81% of the score achieved by a fully functional hexapod. The score then continues to rise at a slower pace, to a value of 1.26m by the 30th generation.

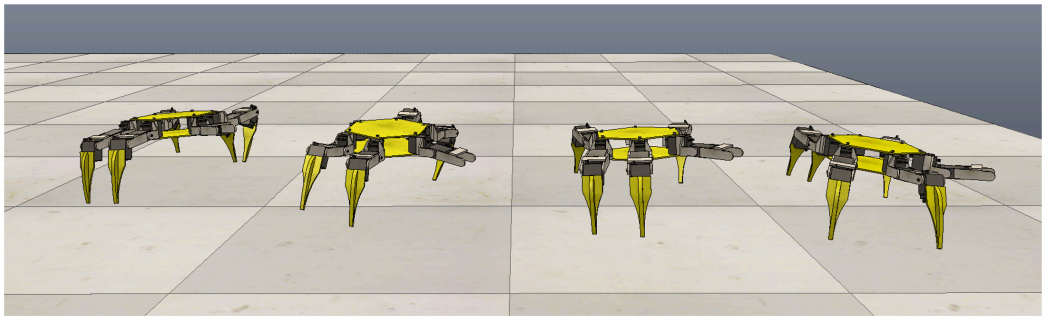


Figure 25 – Stabilised gait generated by the genetic algorithm for a hexapod with a partially removed front leg (movement is left to right)

4.3.2. Heavy defects

Hexapods with heavy defects were initialised with randomised parameters as the solution was expected to be too far from the original tripod gait. An example of the heavy defects applied is shown in the figure below.

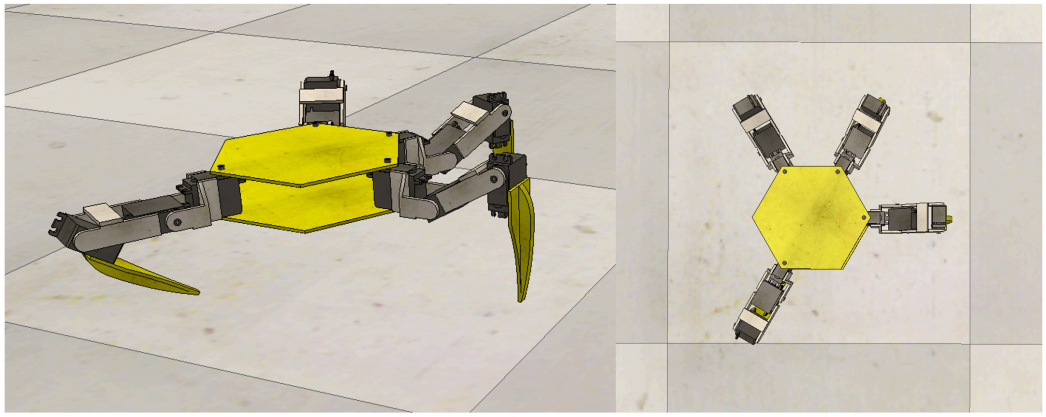


Figure 26 – Example of a heavy defect applied to a hexapod robot – a combination of leg removal and servomotor locking.

In this test, two of the six legs of the hexapods were removed, making this robot a quadruped. The legs were removed such that the final leg distribution was asymmetrical, as seen on the right in the figure above. This was done intentionally for two reasons. Firstly, to add to the challenge of this defect combination, as it was expected that an asymmetrical distribution would be harder to solve for. Secondly, since standard quadrupedal gaits already exist for symmetrical robots, the outcome would be more novel for an asymmetrical distribution. Furthermore, the third joint of the first leg was locked in position, as shown in the left side of the figure above. This was done in order to replicate a mechanical or electronic failure of the servomotor. The GA was then applied to a hexapod population with this combination of defects, and the results are shown below.

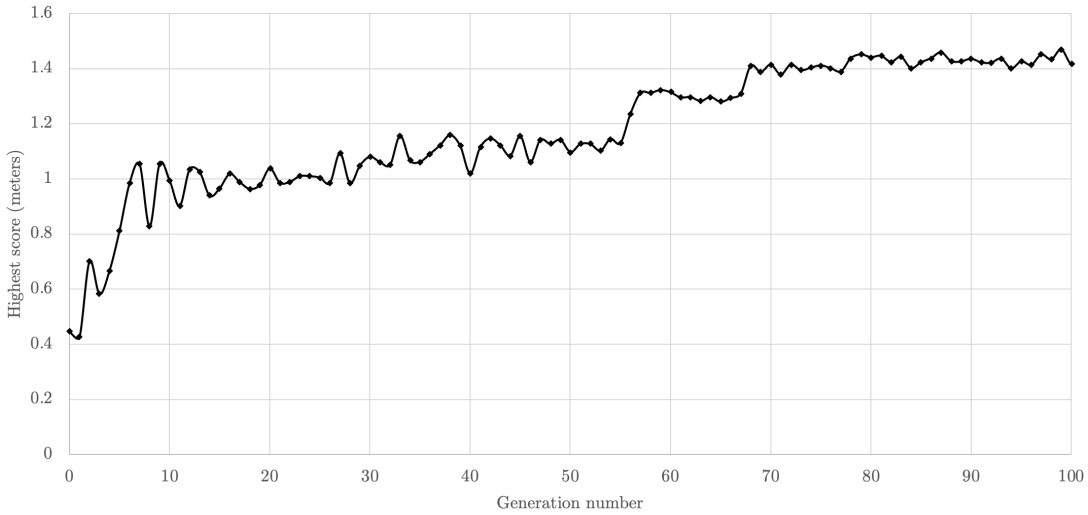


Figure 27 – Results of the GA applied to a hexapod with a combination of defects, starting from randomised parameters

The figure shows the evolution of the hexapods given the combination of defects described above. Due to the randomised initialisation, the starting score is relatively low at 0.42m. However, even though the hexapod in this example was exposed to an impressive combination of defects, the GA generates a gait producing the same score as the original tripod gait (1.43m) in less than 80 generations.

5. Discussion

Draw the threads of your project work together and relate outcomes to the literature review. Discuss errors and uncertainty. Refrain from introducing too much new material in this section. Discuss any ethical implications of your findings

5.1. On the investigative outcomes

One of the preliminary objectives of this project was to investigate different types of machine learning algorithms. The two algorithms selected for investigation: Q-Learning and Genetic Algorithms were intentionally chosen for their differences in the way they function. The respective investigations showed that these functional differences lead to fundamental dissimilarities between the potential gaits generated by these algorithms, which are summarised in the table below. Indeed, it was found that the Q-Learning algorithm, which processes data at every timestep of the simulation, could only be used to produce an aperiodic, or "free" gait. On the other hand, it was found that the genetic algorithm, which processes data once per simulation, could only be used to produce periodic gaits.

Table 3 – Differences in the investigated algorithms and consequences on the gaits produced

Algorithm	*Genetic*	*Q-Learning*
Frequency of data processing	Once per simulation	Once per frame
Types of gaits generated	Periodic	Aperiodic

Though these investigations put forward the different types of techniques used in machine learning, they also cast a light on the concepts shared by these algorithms, which are summarised in the table below. For example, similarities were found between the fitness function used by the GA and the reward function used by the Q-Learning algorithm. Both of these functions are used in the same goal of giving the Agent feedback on the effect of its actions on its environment. Another concept shared by these two algorithms is the encouragement of explorative behaviour. In the case of the GA, this was done by the mutation function, whereas in the Q-Learning algorithm, this was done by employing an epsilon-greedy strategy.

Table 4 – Concepts shared by the investigated algorithms and their respective functions

Algorithm	*Genetic*	*Q-Learning*
Environment feedback	Fitness function	Reward function
Explorative behaviour	Mutation	Epsilon-greedy

The main outcome of the technical testing of the Q-Learning algorithm on the Frozen Lake game was the decision to focus on the Genetic Algorithm. This decision was driven by the high computation time of around 24hrs per solve-run that was expected if the former were to be upscaled to a more complex problem such as the hexapod. This anticipated computation time is in agreement with the literature discussed earlier in the report. Indeed, in a project of similar scale, M. Shahriari found that updating the Q-Table values to 98% completeness took 10 days of computation [9]. Though the specific machine used in this project is not known, this confirms the estimate and consequent decision that was taken.

5.2. Conceivable areas of implementation

With further development, the final algorithm developed in this project could potentially be implemented in the areas listed below:

A. A realistic implementation would be to use such an algorithm in the development phase of the control system of a legged robot. This algorithm could help designers with conceiving different gaits depending on the type of damage that the robot has been affected to. These gaits could be stored onboard the robot, as parameter matrices or other formats. The appropriate gait could then be implemented when damage is sensed by the robot. Such an approach would not require adding much computational power to existing robotic platforms, since the parameter matrices are relatively light. Such robots would not be considered as fully resilient, as they would be limited by the number of damage scenarios explored by the designers. However, this method would nonetheless drastically reduce the design time of the adaptive gaits.

B. A more ambitious implementation would be to run the algorithm onboard the robot. The fitness testing of the generated gaits could be run in two ways.

First, each gait could be simulated in an environment such as CoppeliaSim, as was done during this project. For this however, precise diagnostics systems would be needed to replicate the damage in the simulation. Furthermore, this would require a highly capable computer to be available onboard the robot.

A second method would be to physically run the fitness test on the existing robot in the real world. This would remove the need for diagnostics. Indeed, one of the advantages of the genetic algorithm is that it is never actually "aware" of damage incurred by the robot. However, this option would risk subjecting the robot to more damage during the testing of the gaits. Furthermore, as shown in section 4.3.1., light damage takes around 4 generations of 8 hexapods to be adjusted to, or about 6 minutes of real test time. This time is drastically reduced in simulation environments as the hexapods are run in parallel (around 30 seconds for this example).

A combination of both options could address most the issues stated above. The initial generations could be simulated until an acceptable score is reached. A final few generations could then be physically run on the robot to account for errors in the damage diagnosis. Such an approach, albeit based on a different method of machine learning, was shown to be successful by Cully et al. [24].

5.3. Uncertainties, limitations and areas of future work

Though the final algorithm shows promising results, a number of limitations were exposed. The main areas of concern are related to the overall quality of the solutions, as well as the efficiency of the algorithm.

First of all, although the solutions generated were awarded satisfactory scores, the stability and realism of the gaits would need further investigation. Indeed, some of the gaits produced a "rocking" motion, similar to that shown in figure 22. Other gaits were found to curve instead of following a perfectly straight line. These issues could be dealt with by refining the fitness function to take a larger set of inputs into account, such as the yaw and tilt of the hexapod during the test and awarding a higher score to gaits with lower ground-to-foot contact time.

Furthermore, in terms of realism, it was found that some of the gaits would take advantage of the low coefficient of friction between the hexapod feet and the ground in the simulation environment, by sliding the legs along the ground for some of the gait period. The realism of the generated gaits could be tested by implementing the motion profiles on a physical hexapod and comparing its motion to that of the simulated equivalent.

Secondly, future work could look into increasing the overall efficiency of the algorithm. As explained in section 4.2.5., the time spent investigating the optimal crossover method was limited by the timeframe of the project, and the final method used was therefore relatively basic. Since this part of the genetic algorithm is a key factor in achieving faster conversions towards the set goal in fewer generations, dedicating more time to optimising this function would greatly increase the overall efficiency of the algorithm.

Finally, the simulation runtime could also be optimised. This could be done by running CoppeliaSim in headless mode, i.e., without the graphical user interface (this did not work on the author's machine). This would allow for increased population sizes, increasing the search space of the algorithm and ultimately reducing the total time to solution.

6. Conclusion

The overall project objectives were successfully met. An iterative search for a machine learning method suitable for teaching hexapods how to limp was undertaken. After an extensive literature review, it was found that two methods would be of interest: Q-Learning and Genetic Algorithms. CoppeliaSim, an open-source physics engine was found to be an appropriate environment to test and develop these algorithms. The main advantage of using this environment was the ability to interact with the simulation via an external application program interface.

The first investigation into Q-Learning provided many useful insights. Testing this algorithm on a simple computer game, Frozen Lake, provided results that were used to estimate the computation time needed if the algorithm were to be upscaled to the hexapod gait problem. Though this first algorithm showed some potential, it was found that, with the available computational power and storage access speed, the runtime of the algorithm could be estimated to at least 24hrs. This estimate was in agreement with previous literature in this research area. Given this estimate, the second method was then investigated.

The second investigation into Genetic Algorithms proved successful in completing the final project objectives. The algorithm that was developed iteratively selected hexapod parameters related to the motion of the legs, generating increasingly superior ways of walking, known as gaits. A novel motion profile equation was implemented, enabling more complex gaits to be generated.

Having developed the overall structure of the algorithm, a first testing phase was undertaken to develop and tune different hyperparameters such as the population size, the mutation rate and the genetic crossover method. A population of size 8, a mutation rate of 5%, and a bitwise crossover method, were found to be the ideal parameters for this project.

Once all the parameters were set, the algorithm's capacity to generate adaptive gaits was evaluated during a secondary testing phase. Examples of light and heavy damage recovery were shown. The algorithm successfully generated an adaptive gait for a lightly damaged hexapod, achieving 81% of the undamaged score in only 4 generations for a total of 30 seconds of processing time. Heavier defects combining the removal of multiple components and blocking of motors, were also tested. Again, the algorithm successfully produced an adapted gait, which achieved 100% of the original undamaged score in less than 80 generations. Finally, the potential for implementation of such an algorithm in real world robotics was discussed, while also taking to account the errors and limitations encountered in the project.

7. Bibliography

[1] Christopher Intagliata, "The Origin Of The Word 'Robot,'" Apr. 22, 2011. https://www.sciencefriday.com/segments/the-origin-of-the-word-robot/ (accessed Apr. 07, 2021).

[2] S. Makris, *Cooperating robots for flexible manufacturing*. Cham, Switzerland: Springer, 2021.

[3] Álvaro Rocha, Manolo Paredes-Calderón, and Teresa Guarda, *Developments and Advances in Defense and Security : Proceedings of MICRADS 2020*, 1st ed. Singapore: Imprint: Springer, 2020.

[4] Boston Dynamics, "BigDog Overview," *YouTube*. https://www.youtube.com/watch?v=cNZPRsrwumQ&ab_channel=BostonDynamics (accessed Apr. 08, 2021).

[5] Hardarson and Freyr, *Locomotion for difficult terrain*. KTH, 1998.

[6] A. Mahajan and F. Figueroa, "Four-legged intelligent mobile autonomous robot," *Robotics and Computer-Integrated Manufacturing*, vol. 13, no. 1, pp. 51–61, Mar. 1997, doi: 10.1016/S0736-5845(96)00028-2.

[7] M. L. Visinsky, J. R. Cavallaro, and I. D. Walker, "Robotic fault detection and fault tolerance: A survey," *Reliability Engineering and System Safety*, vol. 46, no. 2, pp. 139–158, Jan. 1994, doi: 10.1016/0951-8320(94)90132-5.

[8] F. M. Noori, D. Portugal, R. P. Rocha, and M. S. Couceiro, "On 3D simulators for multi-robot systems in ROS: MORSE or Gazebo?," in *SSRR 2017 - 15th IEEE International Symposium on Safety, Security and Rescue Robotics, Conference*, Oct. 2017, pp. 19–24, doi: 10.1109/SSRR.2017.8088134.

[9] M. Shahriari and A. A. Khayyat, "DESIGN, IMPLEMENTATION AND CONTROL OF A HEXAPOD ROBOT USING REINFORCEMENT LEARNING APPROACH." Accessed: Apr. 18, 2021. [Online].

[10] J. Kon, "Gait Generation for Damaged Hexapods using Genetic Algorithm." Accessed: Apr. 18, 2021. [Online]. Available: https://scholarworks.rit.edu/theses.

[11] R. Campos, V. Matos, M. Oliveira, and C. Santos, "GAIT GENERATION FOR A SIMULATED HEXAPOD ROBOT: A NONLINEAR DYNAMICAL SYSTEMS APPROACH." Accessed: Apr. 15, 2021. [Online].

[12] Q. Chang and F. Mei, "A Bioinspired Gait Transition Model for a Hexapod Robot," 2018, doi: 10.1155/2018/2913636.

[13] J. Kon and F. Sahin, "Gait Generation for Damaged Hexapods using a Genetic Algorithm," in *SOSE 2020 - IEEE 15th International Conference of System of Systems Engineering, Proceedings*, Jun. 2020, pp. 451–456, doi: 10.1109/SoSE50414.2020.9130561.

[14] R. B. Segal, "Machine learning as massive search," 1997, Accessed: Apr. 15, 2021. [Online]. Available: https://digital.lib.washington.edu:443/researchworks/handle/1773/6863.

[15] P. K. Pal and K. Jayarajan, "Generation of Free Gait—A Graph Search Approach," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 299–305, 1991, doi: 10.1109/70.88139.

[16] G. Ren, W. Chen, S. Dasgupta, C. Kolodziejski, F. Wörgötter, and P. Manoonpong, "Multiple chaotic central pattern generators with learning for legged locomotion and malfunction compensation," *Information Sciences*, vol. 294, pp. 666–682, Feb. 2015, doi: 10.1016/j.ins.2014.05.001.

[17] H. Cruse, T. Kindermann, M. Schumm, J. Dean, and J. Schmitz, "Walknet—a biologically inspired network to control six-legged walking," *Neural Networks*, vol. 11, no. 7–8, pp. 1435–1447, Oct. 1998, doi: 10.1016/S0893-6080(98)00067-7.

[18] M. S. Erden and K. Leblebicioğlu, "Free gait generation with reinforcement learning for a six-legged robot," *Robotics and Autonomous Systems*, vol. 56, no. 3, pp. 199–212, Mar. 2008, doi: 10.1016/j.robot.2007.08.001.

[19] I. R. Woering and D. Kostic, "Simulating the 'first steps' of a walking hexapod robot." Accessed: Apr. 18, 2021. [Online].

[20] J. Won and J. Lee, "Learning Body Shape Variation in Physics-based Characters," *ACM Trans. Graph*, vol. 38, p. 12, 2019, doi: 10.1145/3355089.3356499.

[21] M. Schilling, K. Konen, F. W. Ohl, and T. Korthals, "Decentralized Deep Reinforcement Learning for a Distributed and Adaptive Locomotion Controller of a Hexapod Robot." [Online]. Available: https://github.com/kkonen/.

[22] A. Manglik, K. Gupta, and S. Bhanot, "Adaptive gait generation for hexapod robot using Genetic Algorithm," Feb. 2017, doi: 10.1109/ICPEICES.2016.7853681.

[23] "OpenAI Gym." https://gym.openai.com/ (accessed Apr. 21, 2021).

[24] A. Cully, J. Clune, D. Tarapore, and J. B. Mouret, "Robots that can adapt like animals," *Nature*, vol. 521, no. 7553, pp. 503–507, May 2015, doi: 10.1038/nature14422.

Appendix:

Appendix A:
Main GA code (Python):

```
# Testing the API (works 8PopCos)
# This script works in continuous API mode
# To run in headless mode:
# /Applications/coppeliaSim.app/Contents/MacOS/coppeliaSim -h /Users/guy/Documents/3rd\
Year/FYP/Scenes/8Pop.ttt

# Import relevant modules
import sim
import numpy as np
import ctypes
import sys
import time
import random
import math

def connectToAPI():

    # Init connection
    sim.simxFinish(-1)   # just in case, close all opened connections
    clientID=sim.simxStart('127.0.0.1',19997,True,True,5000,5)   # Connect to
CoppeliaSim

    if clientID != -1:
        print('Connected to remote API server')
    else:
        print('Connection unsuccessful')
        sys.exit()

    return clientID

def initPop(popSize):
    # Init with random params
    # Returns list of members
    members = []
    for i in range(popSize):
        # Create new member:
        m = Member(i)

        # Initial 3 sin params, A B C G change this later
        p = 7

        # Init array
        params = np.zeros((3, 6, p))

        # Create Sin Param Array with random values
        # Body Joints
        for leg in range(6):
            params[0, leg, 0] = round(random.uniform(0, 0.5), 2)  # A
            params[0, leg, 1] = round(random.randrange(0, 8, 2), 2)    # B
            params[0, leg, 2] = round(random.uniform(0, 3.14), 2)    # C
            params[0, leg, 3] = round(random.uniform(0, 0.5), 2)  # D
            params[0, leg, 4] = round(random.randrange(0, 4, 1), 2)    # E
            params[0, leg, 5] = round(random.uniform(0, 3.14), 2)    # F
            params[0, leg, 6] = round(random.uniform(-1, 1), 2)   # G

        # Knee Joints
        for leg in range(6):
            params[1, leg, 0] = round(random.uniform(0, 0.5), 2)  # A
            params[1, leg, 1] = round(random.randrange(0, 8, 2), 2)    # B
            params[1, leg, 2] = round(random.uniform(0, 3.14), 2)    # C
            params[1, leg, 3] = round(random.uniform(0, 0.5), 2)  # D
```

```
            params[1, leg, 4] = round(random.randrange(0, 8, 2), 2)    # E
            params[1, leg, 5] = round(random.uniform(0, 3.14), 2)    # F
            params[1, leg, 6] = round(random.uniform(0, 1), 2)    # G

        # Foot Joints
        for leg in range(6):
            params[2, leg, 0] = round(random.uniform(0, 0.5), 2)  # A
            params[2, leg, 1] = round(random.randrange(0, 8, 2), 2)    # B
            params[2, leg, 2] = round(random.uniform(0, 3.14), 2)    # C
            params[2, leg, 3] = round(random.uniform(0, 0.5), 2)  # D
            params[2, leg, 4] = round(random.randrange(0, 8, 2), 2)    # E
            params[2, leg, 5] = round(random.uniform(0, 3.14), 2)    # F
            params[2, leg, 6] = round(random.uniform(0, 1), 2)    # G

        m.params = params

        members.append(m)

    return members

def initPopTripod(popSize):
    # To initialize with tripod gait
    # Returns list of members

    offset = math.pi/6


    members = []
    for i in range(popSize):
        # Create new member:
        m = Member(i)

        # Initial 3 sin params, A B C G change this later
        p = 7

        # Init array
        params = np.zeros((3, 6, p))

        # Create Sin Param Array with tripod values

        freqency = 8

        # Start with legs 0,2,4

        # Body Joints
        for leg in [0,2,4]:
            params[0, leg, 0] = 0.2 # A
            params[0, leg, 1] = freqency          # B
            params[0, leg, 2] = math.pi    # C
            params[0, leg, 3] = 0 # D
            params[0, leg, 4] = 0 # E
            params[0, leg, 5] = 0  # F
            params[0, leg, 6] = 0  # G

        # Knee Joints
        for leg in [0,2,4]:
            params[1, leg, 0] = 0.2  # A
            params[1, leg, 1] = freqency    # B
            params[1, leg, 2] = math.pi*5/2   # C
            params[1, leg, 3] = 0  # D
            params[1, leg, 4] = 0    # E
            params[1, leg, 5] = 0   # F
            params[1, leg, 6] = 0    # G

        # Foot Joints
        for leg in [0,2,4]:
            params[2, leg, 0] = 1.3  # A
            params[2, leg, 1] = 0   # B
            params[2, leg, 2] = math.pi*5/2    # C
            params[2, leg, 3] = 0 # D
            params[2, leg, 4] = 0  # E
            params[2, leg, 5] = 0   # F
            params[2, leg, 6] = 0   # G

        # Offset outside legs (0 and 4)
```

```
        params[0, 0, 6] = - math.pi/4  # G
        params[0, 4, 6] = math.pi/4  # G

        # Reverse middle leg (2)
        params[0, 2, 0] = - params[0, 2, 0]  # A

        # Legs 1,3,5

        # Body Joints
        for leg in [1,3,5]:
            params[0, leg, 0] =  0.2 # A
            params[0, leg, 1] = freqency          # B
            params[0, leg, 2] = 0  # C
            params[0, leg, 3] = 0 # D
            params[0, leg, 4] = 0 # E
            params[0, leg, 5] = 0  # F
            params[0, leg, 6] = 0  # G

        # Knee Joints
        for leg in [1,3,5]:
            params[1, leg, 0] = 0.2  # A
            params[1, leg, 1] = freqency    # B
            params[1, leg, 2] = math.pi*3/2   # C
            params[1, leg, 3] = 0  # D
            params[1, leg, 4] = 0    # E
            params[1, leg, 5] = 0   # F
            params[1, leg, 6] = 0    # G

        # Foot Joints
        for leg in [1,3,5]:
            params[2, leg, 0] = 1.3  # A
            params[2, leg, 1] = 0   # B
            params[2, leg, 2] = math.pi*5/2    # C
            params[2, leg, 3] = 0 # D
            params[2, leg, 4] = 0  # E
            params[2, leg, 5] = 0   # F
            params[2, leg, 6] = 0   # G

        # Offset outside legs (1 and 5)
        params[0, 1, 6] =  math.pi/4  # G
        params[0, 3, 6] = - math.pi/4  # G

        # Reverse middle leg (3)
        params[0, 3, 0] = - params[0, 3, 0]  # A

        # Reverse leg 1
        params[0, 1, 0] = - params[0, 1, 0]

        m.params = params

        members.append(m)

    return members

def testPop(members,timeOut):
    # Start simulation
    returnCode = sim.simxStartSimulation(clientID, sim.simx_opmode_oneshot_wait)
    print('Start Sim ReturnCode ' + str(returnCode))

    # Send timeOut
    returnCode = sim.simxSetIntegerSignal(clientID, 'timeOut', timeOut,
sim.simx_opmode_oneshot_wait)
    print('TimeOut ReturnCode ' + str(returnCode))

    # Send params to all members
    for member in members:
        # Format params
        params = member.params
        packedData = sim.simxPackFloats(params.flatten())
        raw_bytes = (ctypes.c_ubyte * len(packedData)).from_buffer_copy(packedData)
        signal = 'signal_' + str(member.id)

        # Send param packets
        returnCode = sim.simxSetStringSignal(clientID, signal, raw_bytes,
sim.simx_opmode_oneshot_wait)
```

```
        print('params sent to ' + str(member.id))
        print('ReturnCode ' + str(returnCode))

def uniformCross(members):
    # Number of params, change when adding cos
    p = 7

    # Remove bottom half of list
    members = members[:len(members) // 2]
    # Generate new Gen
    for i in range(0, len(members), 2):
        p1 = members[i].params.flatten()
        p2 = members[i+1].params.flatten()
        # Go through all params and attribute randomly to child
        c1 = uniformChild(p1, p2)
        c2 = uniformChild(p1, p2)
        # Assign new values to children and add to list
        m1 = Member(100+i)
        m2 = Member(101+i)
        m1.params = c1.reshape(3, 6, p)
        m2.params = c2.reshape(3, 6, p)
        members.append(m1)
        members.append(m2)

    # Reassign IDs
    i = 0
    for member in members:
        member.id = i
        i += 1
    return members


def singlePointCross(members):
    # Number of params, change when adding cos

    # Remove bottom half of list
    members = members[:len(members) // 2]
    # Generate new Gen
    for i in range(0, len(members), 2):
        p1 = members[i].params
        p2 = members[i+1].params
        c1 = singlePointChild(p1,p2)
        c2 = singlePointChild(p1,p2)
        m1 = Member(100+i)
        m2 = Member(101+i)
        m1.params = c1
        m2.params = c2
        members.append(m1)
        members.append(m2)


    # Reassign IDs
    i = 0
    for member in members:
        member.id = i
        i += 1

    return members

def singlePointChild(p1,p2):

    params = np.zeros((3, 6, 7))
    for leg in range(6):
        for joint in range (3):
            x = random.randrange(0,6,1)
            for coef in range (7):
                if coef > x:
                    params[joint,leg,coef] = p1[joint,leg,coef]
                else:
                    params[joint,leg,coef] = p2[joint,leg,coef]

    # Mutating in here
    mut = initPop(1)[0].params.flatten()
    params = params.flatten()
```

```
    for j in range(len(params)):
        x = random.random()
        if x > 0.95:
            params[j] = mut[j]

    params = params.reshape(3, 6, 7)

    return params

def uniformChild(p1,p2):

    c = np.zeros(len(p1))
    mut = initPop(1)[0].params.flatten()
    for j in range(len(p1)):
        x = random.random()
        if x < 0.5:
            c[j] = p1[j]
        elif x < 0.95:
            c[j] = p2[j]
        else:
            c[j] = mut[j]

    return c

def checkStatus():
    while True:
        # poll the useless signal (to receive a message from server)
        sim.simxGetIntegerSignal(clientID, 'Status', sim.simx_opmode_blocking)

        # check server state (within the received message)
        e = sim.simxGetInMessageInfo(clientID, sim.simx_headeroffset_server_state)

        # check bit0
        not_stopped = e[1] & 1

        if not not_stopped:
            break
        else:
            print('Waiting for sim to stop')

class Member:
    def __init__(self, id):
        self.params = []
        self.id = id
        self.position = None
        self.score = None

    def getPosition(self):
        signal = 'endSignal_' + str(self.id)
        returnCode, endSignal = sim.simxGetStringSignal(clientID, signal,
sim.simx_opmode_oneshot_wait)
        position = list(sim.simxUnpackFloats(endSignal))
        while returnCode == 8 or len(position) == 0:
            print('waiting for position from member ' + str(self.id))
            time.sleep(3)
            returnCode, endSignal = sim.simxGetStringSignal(clientID, signal,
sim.simx_opmode_oneshot_wait)
            position = list(sim.simxUnpackFloats(endSignal))

        self.position = position

    def calcScore(self):
        x = self.position[0]
        y = self.position[1]
        z = self.position[2]
        self.score = x
        if z < 0:
            print('Man down')
            print(self.id)
            self.score = -10



# MAIN SCRIPT:
```

```
# HyperParams
popSize = 8   # Needs to be a multiple of 4
scoreLimit = 20
timeOut = 10
increment = 0.3
startIncrement = 1

# Check for existing connection
if 'clientID' not in globals():
    clientID = connectToAPI()

# setup Status Signal
sim.simxSetIntegerSignal(clientID,'Status',1,sim.simx_opmode_blocking)

# --------Init Phase--------

# Initialise population with random parameters
members = initPopTripod(popSize)

# Test the population
testPop(members, timeOut)

# Get final position for each member
for member in members:
    member.getPosition()

# Stop Sim
returnCode = sim.simxStopSimulation(clientID, sim.simx_opmode_oneshot_wait)
print('End Sim ReturnCode ' + str(returnCode))

# Calc score based on position
for member in members:
    member.calcScore()

# Sort members by score (highest score will get lowest index in list)
members.sort(key=lambda x: x.score, reverse=True)

# Fetch highest score
highScore = members[0].score

# Set Gen counter and scoreList
gen = 0
scoreList = []
floatTime = timeOut
print('Finished testing Generation ' + str(gen) + ' \nHighscore:  ' + str(highScore))

# --------Loop Phase--------
# Loop until score limit is reached
while highScore < scoreLimit:
    gen += 1
    # Set timeOut
    if gen % 2 == 0:
        timeOut = round(floatTime+random.randrange(-1,3,1))
        print('----floatTime')
        print(floatTime)
        print('----timeOut')
        print(timeOut)
    else:
        timeOut = floatTime

    print('Start testing Generation ' + str(gen))
    # Generate new members
    members = uniformCross(members)

    # Wait for sim to restart:
    checkStatus()

    # Test the population and get results (same as Init Phase)
    testPop(members, timeOut)
    for member in members:
        member.getPosition()

    # Stop Sim
    returnCode = sim.simxStopSimulation(clientID, sim.simx_opmode_oneshot_wait)
```

```
    print('End Sim ReturnCode ' + str(returnCode))

    for member in members:
        member.calcScore()
    members.sort(key=lambda x: x.score, reverse=True)

    # Update console:
    print('Finished testing Generation ' + str(gen) + ' \nScoreList:')
    for member in members:
        print(member.score)

    # Add highscore to list
    highScore = members[0].score
    if timeOut == 10:
        scoreList.append(highScore)
        print('added high score to list')
    np.save('ScoreList', scoreList)


# Debugging here
for member in members:
    print(member.score)
```

Appendix B
CoppeliaSim Motion Script (LUA)

```
function sysCall_init()
    -- Set timeOut
    timeOut = 10

    -- Robot ID
    ID = 0
    signal = 'signal_'..tostring(ID)
    endSignal = 'endSignal_'..tostring(ID)

    -- Get joint handles
    bodyJoints={-1,-1,-1,-1,-1,-1}
    kneeJoints={-1,-1,-1,-1,-1,-1}
    footJoints={-1,-1,-1,-1,-1,-1}
    for i=1,6,1 do
        bodyJoints[i]=sim.getObjectHandle('hexa_joint1_'..i-1)
        kneeJoints[i]=sim.getObjectHandle('hexa_joint2_'..i-1)
        footJoints[i]=sim.getObjectHandle('hexa_joint3_'..i-1)
    end

    -- Get body handle
    bodyHandle = sim.getObjectHandle('hexa_body')

    -- Get initial position
    pos1 = sim.getObjectPosition(bodyHandle,-1)

    -- Get initial angles
    bodyAngles={-1,-1,-1,-1,-1,-1}
    kneeAngles={-1,-1,-1,-1,-1,-1}
    footAngles={-1,-1,-1,-1,-1,-1}
    for i=1,6,1 do
        bodyAngles[i]=sim.getJointTargetPosition(bodyJoints[i])
        kneeAngles[i]=sim.getJointTargetPosition(kneeJoints[i])
        footAngles[i]=sim.getJointTargetPosition(footJoints[i])
    end

    -- Init Console to Print
    --consoleHandle=sim.auxiliaryConsoleOpen('ouput stuff',25,2)

    -- Clear param signal
    sim.clearStringSignal(signal)

    -- Init params
    params = {}
    for i=1,8*3*6 do -- wrong size but works anyway
        params[i]=0
    end
    params = reshape(params)

    -- Init sendPos
    sendPos = true

    -- Debugging here
    dumpTable(params)
    dumpTable(pos1)



end
```

```
function sysCall_cleanup()

end

function sysCall_actuation()

    -- Get time
    t = sim.getSimulationTime()

    -- Check for params in String format
    packedTable = sim.getStringSignal(signal)

    -- New signal received - update params table
    if (packedTable ~= null) then

        -- Unpack params from API and reshape
        --sim.auxiliaryConsolePrint(consoleHandle,'Success! \n')
        params = sim.unpackFloatTable(packedTable)
        dumpTable(params)
        params = reshape(params)
        dumpTable(params)

        -- Clear signal
        sim.clearStringSignal(signal)
    end

    -- Check for timeOut
    check = sim.getIntegerSignal('timeOut')
    if (check ~= null) then
        timeOut = check
    end


    -- Actual Actuation here:
    -- Setting new target values using params - joints called individually
    for leg=1,6 do
        -- Calc new target value (sin is in rad)
        bodyAngles[leg]    =        params[1][leg][1]*math.sin(params[1][leg][2]*t    +
params[1][leg][3]) + params[1][leg][4]*math.cos(params[1][leg][5]*t + params[1][leg][6])
+ params[1][leg][7]
        kneeAngles[leg]    =        params[2][leg][1]*math.sin(params[2][leg][2]*t    +
params[2][leg][3]) + params[2][leg][4]*math.cos(params[2][leg][5]*t + params[2][leg][6])
+ params[2][leg][7]
        footAngles[leg]    =        params[3][leg][1]*math.sin(params[3][leg][2]*t    +
params[3][leg][3]) + params[3][leg][4]*math.cos(params[3][leg][5]*t + params[3][leg][6])
+ params[3][leg][7]

        -- Set new target value
        sim.setJointTargetPosition(bodyJoints[leg],bodyAngles[leg])
        sim.setJointTargetPosition(kneeJoints[leg],kneeAngles[leg])
        sim.setJointTargetPosition(footJoints[leg],footAngles[leg])

    end


    -- Send position when time is up
    if (t > timeOut and sendPos) then
        -- Get current position compared to start
        pos2 = sim.getObjectPosition(bodyHandle,-1)

        -- Calc delta
        endPos = {}
        for i=1,2 do
            endPos[i] = pos2[i]-pos1[i]
```

```
        end

        -- Get absolute position in
        endPos[3] = pos2[3]

        -- Send table
        dumpTable(endPos)
        packedTable = sim.packFloatTable(endPos)
        sim.setStringSignal(endSignal, packedTable)

        --  Stop sending
        sendPos = false
    end



end


function reshape(t)
    -- Reshape to joints x legs x params table
    joints = 3 -- change this when adding cos
    legs = 6
    params = 7

    n = {} -- new matrix
    index = 0
    for j=1,joints do
        n[j] = {}
        for l=1,legs do
            n[j][l]={}
            for p=1,params do
                index = index + 1
                n[j][l][p] = t[index]
            end
        end

    end

    return n

end

-- Dumping to console
function dump(o)
   if type(o) == 'table' then
      local s = '{ '
      for k,v in pairs(o) do
         if type(k) ~= 'number' then k = '"'..k..'"' end
         s = s .. '['..k..'] = ' .. dump(v) .. ','
      end
      return s .. '} '
   else
      return tostring(o)
   end
end


function dumpTable(t)
    --sim.auxiliaryConsolePrint(consoleHandle,'Table: \n'..dump(t)..'\n')
end
```