Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Oscar Brunell Mørk

SimSerpent: A Physics-based Simulator for a Next Generation Snake Robot

SimSerpent: En Fysikkbasert Simulator for en Neste Generasjons Slangerobot

Master's thesis in Cybernetics and Robotics
Supervisor: Øyvind Stavdahl
Co-supervisor: Jostein Løwer
June 2023

NTNU
Norwegian University of
Science and Technology

Oscar Brunell Mørk

SimSerpent: A Physics-based Simulator for a Next Generation Snake Robot

SimSerpent: En Fysikkbasert Simulator for en Neste Generasjons Slangerobot

Master's thesis in Cybernetics and Robotics
Supervisor: Øyvind Stavdahl
Co-supervisor: Jostein Løwer
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

Boa, a next-generation sensor-driven snake robot, is currently under development at the Department of Engineering Cybernetics (ITK) at the Norwegian University of Science and Technology (NTNU). As a result, an assignment has been issued to research and develop a contemporary snake robot simulator that can aid in the rapid development of state estimation and control strategies for Boa. This master thesis goes into the development part of the assignment and aims to present a fully-fledged simulator along with results demonstrating its efficacy.

To begin with, the thesis presents relevant background information about snake robot research at ITK as well as the foundational theory on physics-based simulators. Subsequently, the thesis proceeds to showcase the development process of the simulator, including software specification, choice of framework, and a detailed explanation of the code itself.

Moreover, the thesis showcases four experiments completed on the snake robot simulator, where each experiment assesses a different aspect of the simulator. The first experiment demonstrates the simulator's versatility in reconfiguring the snake robot in various ways. The second experiment showcases the simulator's ability to accurately simulate collisions between objects, examining whether the results adhere to energy conservation and momentum principles. The third experiment showcases the snake robot's ability to generate sinusoidal movements both in the presence and absence of friction and discusses possible causes for the experiment's deviation from the expected theory. Lastly, the fourth experiment demonstrates the robot's utilization of collision forces to generate movement, a vital concept within snake robot research. Overall the simulator performed well and within expectations, with only minor issues present at the end.

Keywords: Robotics technology, Snake Robots, Robotic simulator, Physics Engines

Sammendrag

Boa, en neste generasjons sensordrevet slangerobot, er for tiden under utvikling ved Instituttet for teknisk kybernetikk (ITK) ved Norges teknisk-naturvitenskapelige universitet (NTNU). Som et resultat har det blitt gitt en masteroppgave for å forske på og utvikle en moderne slangerobotsimulator som kan hjelpe til med rask utviklingen av estimerings- og kontrollstrategier for Boa. Denne masteroppgaven går inn i utviklingsdelen av oppgaven og har som mål å presentere en fullverdig simulator sammen med resultater som beviser dens effektivitet.

Først presenterer oppgaven relevant bakgrunnsinformasjon om slangerobotforskning ved ITK samt grunnleggende teori om fysikkbaserte simulatorer. Deretter fortsetter oppgaven med å vise frem utviklingsprosessen til simulatoren, inkludert programvarespesifikasjon, valg av rammeverk og en detaljert forklaring av selve koden.

Videre viser oppgaven frem fire eksperimenter utført på slangerobotsimulatoren, der hvert eksperiment vurderer forskjellige aspekter ved simulatoren. Det første eksperimentet demonstrerer simulatorens allsidighet i å rekonfigurere slangeroboten på forskjellige måter. Det andre eksperimentet viser simulatorens evne til å nøyaktig simulere kollisjoner mellom objekter, og undersøker om resultatene overholder prinsippene om energisparing og fart. Det tredje eksperimentet viser frem slangerobotens evne til å generere sinusformede bevegelser både i nærvær og fravær av friksjon og diskuterer mulige årsaker til eksperimentets avvik fra den forventede teorien. Til slutt demonstrerer det fjerde eksperimentet robotens utnyttelse av kollisjonskrefter for å produsere fremdrift, som er et viktig konsept innen slangerobotforskning. Totalt sett presterte simulatoren godt og som forventet, med bare mindre problemer til stede på slutten.

Nøkkelord: Robotikk teknologi, Slange robotikk, Robotikk simulator og Fysikkmotor

Preface

This master's thesis is written in the spring of 2023 for the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. This thesis signifies the end of a 2-year study in cybernetics and robotics and covers a total of 30 ECTS. I want to thank Jostein Løwer for his continued guidance throughout this report, Øyvind Stavdahl for administrative assistance, and ITK for assistance with computers and office space.

- Oscar Brunell Mørk June 6, 2023

Contents

1 Nomenclature/Glossary

ITK	Department of Engineering Cybernetics
NTNU	Norwegian University of Science and Technology
HOAL	Hybrid Obstacle Aided Locomotion
OAL	Obstacle Aided Locomotion
POAL	Perception Obstacle Aided Locomotion
API	Application Programming Interface
GUI	Graphical User Interface
XML	Extensible Markup Language. Used for storing and structuring data
CSV	Comma-Separated Values. Used for storing data
Simulator	Collective term for physics and rendering engines working together
Physics engine	Software designed for simulating the physical behavior of objects
Rendering engine	Software designed for generating visual representation of physics
PID	PID (Proportional-Integral-Derivative) is a control algorithm used to move a system towards a setpoint

2 Introduction

Following Pål Lilljebekk's doctoral paper [1], which presented a new force/torque measurement technique related to HOAL, NTNU initiated research on snake robotics to implement this concept. This research, along with other benefits, has led to the creation of two snake robots at NTNU, named *Mamba* and *Boa*. One of the main goals of these snake robots was through experimentation to develop new state estimation and control strategies as well as prove HOAL as a viable method for locomotion. However, performing physical experiments can be both time-consuming and costly. This led to the necessity of a simulator to enhance cost-effectiveness and turnaround time when developing control strategies and is the basis for this master thesis.

After it was decided that a simulator would be developed, a project assignment and a master thesis were produced. The goal of the project assignment was to lay the groundwork for the master thesis, and the assignment therefore focused on researching existing simulators and physics engines. Based on this research, a few simulators or physics engines were selected and experimented with to determine which platform was optimal for developing a snake robot simulator. The results indicated that all tested physics engines were suitable. Still, Isaac Sim was preferred due to its continuous updates and limited prior research, making it an intriguing platform for development.

This master thesis is the continuation of the previously mentioned project assignment completed during fall 2022 at NTNU and aims to produce a full-fledged simulator by its end. The master thesis presents previous research on snake robotics at NTNU, the development process, and an explanation of the code. Lastly, it presents experiments on the simulator that showcase its efficacy and discusses future development options.

2.1 Structure of thesis

To begin with, the thesis presents relevant background information about snake robot research at ITK as well as the foundational theory on physics-based simulators. It is important to mention here that Chapter 3 to 3.6 of this theory is adapted from 12.3, which was written as a preliminary project during the fall of 2022. Subsequently, the thesis proceeds to showcase the development process of the simulator, including software specification, choice of framework, and a detailed explanation of the code itself.

Moreover, the thesis showcases four experiments completed on the snake robot simulator, where each experiment assesses a different aspect of the simulator. The first experiment demonstrates the simulator's versatility in reconfiguring the snake robot in various ways. The second experiment showcases the simulator's ability to accurately simulate collisions between objects, examining whether the results adhere to energy conservation and momentum principles. The third experiment showcases the snake robot's ability to generate sinusoidal movements both in the presence and absence of friction and discusses possible

causes for the experiment's deviation from the expected theory. Lastly, the fourth experiment demonstrates the robot's utilization of collision forces to generate movement, a vital concept within snake robot research.

2.2 Caveats

2.2.1 Comment on references in the thesis

The thesis has multiple references to other research articles, software solutions, and companies to help explain various concepts. To make the thesis read more fluently, it was decided to reference each new concept, article, software, and company the first time they are mentioned and not after. This means that if a reference is missing, it can often be found earlier in the thesis. However, in cases where it remains appropriate, references will be reiterated.

2.2.2 Time limitations

After experiments had been conducted and results had been recorded, it was noticed that experiment three, lateral undulation, deviated from theory. It was later discovered that MuJoCo does not natively support anisotropic friction, which was thought to be the leading cause. However, to properly conduct the experiment, anisotropic friction would have to be added manually which would be quite time-consuming. Due to this issue being discovered during the last week before the delivery date, the project did not have time to implement this. This is also mentioned during the report, but this subchapter acts as a more detailed description of the problem.

2.3 Interpretation of thesis assignment

The thesis assignment is shown in 2.4 and clearly states the goals of the master thesis in an ordered list. Goal number one and two have been perceived as stated and are presented in Chapter 3 and 4. Goal number three has been perceived as making an educated decision based on the results of the project assignment combined with the software specification to select an optimal physics engine and develop a simulator on top of it. Goal four has been perceived as answered by a combination of this master thesis, well-documented code, and a readme file for the software present on GitHub. Lastly, goal five has been answered as part of this project's experiments showcased in Chapter 8.

2.4 The thesis assignment

NTNU
Norwegian University of
Science and Technology

Faculty of Information technology
and Electrical Engineering
Department of Engineering Cybernetics

Project Assignment

Student's name: Oscar Mørk

Field: Engineering Cybernetics

Title (Norwegian): Fysikkbasert simulator for neste generasjons slangerobot

Title (English): Physics-based simulator for a next generation snake robot

Description:
Boa, a next generation sensor-driven snake robot, is currently under development at ITK. This assignment seeks to develop a physics-based snake robot simulator, enabling rapid development of state estimation and control strategies for the snake robot. Snake robot simulation poses strict requirements on the performance of the physics engine both in terms of accuracy and execution time. In this assignment you will further develop the results from the prior project thesis into a complete snake robot simulator.

1. Give a brief overview of the history and development of snake robots at ITK.
2. In collaboration with the HOAL-team at ITK, create a prioritized list of requirements and features for the snake robot simulator.
3. Develop and implement the snake robot simulator, based on the aforementioned requirements and features, and the results from the project thesis.
4. Create complete and accessible documentation for the software.
5. (Optional) If time permits: Showcase the performance of your simulator with a simulated snake robot.

Co-supervisor(s): Jostein Løwer, NTNU

Trondheim, 24.08.2022

Øyvind Stavdahl
Supervisor

3 Background and Theory

Chapter 3 to 3.6 is adapted from 12.3, which was written as a preliminary project during the fall of 2022

This chapter covers relevant theory and background for the report. It explains the theory behind Obstacle Aided Locomotion (OAL), the current state of snake robots at ITK as well as what a simulator is and why it is beneficial for further development of the snake robot.

3.1 Obstacle-Aided Locomotion (OAL)

In 2008, *Transeth et al.* published a paper named *Snake Robot Obstacle-Aided Locomotion: Modeling, Simulations, and Experiments* [2]. This paper discusses two methods serpents use to advance through the terrain, called *lateral undulation* and *obstacle-aided locomotion*. The first term, *lateral undulation*, explains how snakes exploit the asymmetrical scales and scutes on their belly to create friction in a single direction, thus leading to propulsion. This method of locomotion is the most researched and implemented in snake robotics [3], but it makes the robot overly dependent on the surface it moves upon. The paper, therefore, instead looks at the other method of propulsion, *obstacle-aided locomotion* called *OAL* for short. This method employs the concept of *obstacle exploitation*, which implies that instead of avoiding obstacles, the snake uses them to push against and create momentum. In this way, the snake and snake robots become more adaptable to their environment, allowing them to move where their entire body cannot make contact with the ground. OAL has also evolved into a more specialized version called *Hybrid Obstacle Aided Locomotion* or *HOAL* for short. HOAL is the combination of obstacle-aided locomotion and a technique called *HPFC, Hybrid Position/Force Control*. HPFC as explained by *T. Yoshikawa et al.* [4] is a control strategy that combines position control and force control to achieve a specific task. It is relevant to know of HOAL for later explanations, but OAL will remain the focus of this report.

The mathematical theory behind OAL is extensive. One essential part of this theory, as explained by *Løwer et al.* [5] is contact force estimation which can be explained by looking at an example snake robot shown in Figure 1. A snake robot consists of multiple linkages, which are represented by red, blue, and green cylinders. In each of these linkages, there exists a servo motor to generate torque as well as a force/torque measurement system to measure said torque. In an ideal world, where it is assumed that the friction acting on a linkage is either known or equal to zero, the only forces acting on a linkage are the forces/torques of the previous and next linkage as well as that of external objects the linkage is in contact with. Friction is denoted F_R, force measurement from contact with previous linkage as h_{n-1}, force measurement from contact with the next linkage as h_n, external forces as f_{ext} and the summation of all external force vectors as f_{tot}. Based on Newton's second law shown in Equation (2), the sum of forces equals mass times

acceleration. The equation for external forces acting on a linkage can then be shown mathematically as (3). By adding up all the external force vectors as shown in Equation (4), it is possible to calculate a total force vector that gives the direction of motion the snake will achieve by pushing against the external objects.

$$F_R \approx 0 \tag{1}$$

$$\Sigma F = ma \tag{2}$$

$$f_{ext} = ma - h_n - h_{n-1} \tag{3}$$

$$f_{tot} = f_{ext_1} + f_{ext_2} + f_{ext_3} \ldots + f_{ext_n} \tag{4}$$

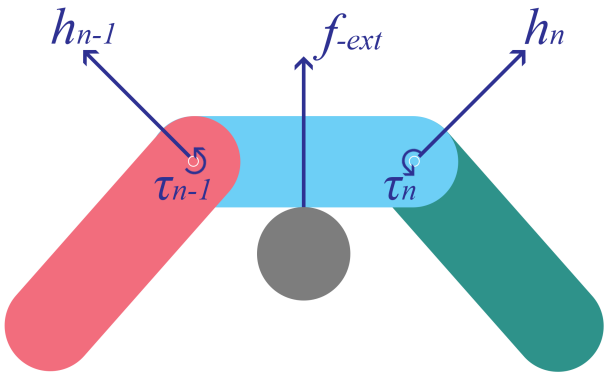


Figure 1: Forces and torques applied on a 2-jointed 2D-snake pushing up against an object. Courtesy of [6].

3.2 Contemporary snake robots developed at ITK

In 2011 a researcher named *Pål Liljebäck* published his doctorate thesis titled *Modelling, development, and control of snake robots* [1]. The thesis presents experiments with two different snake robots named Wheeko and Kulko, where Kulko was used to experiment on the premise of OAL. Kulko succeeded in measuring contact forces with external objects and was, as stated by Pål Liljebäck, fit for further OAL research. He did however in the last chapter of Kulko propose another method for environment sensing based on force measurements which could simplify the snake robot design. This led to The Department of Engineering Cybernetics called ITK at NTNU starting work on creating a new functional snake robot based on *OAL* with this new environment sensing method in mind. The team that works on this is called the HOAL team and will be referred to this for the remainder of the report. As per writing, the HOAL team has produced two contemporary snake robots, Mamba and Boa.

3.2.1 The Mamba snake robot

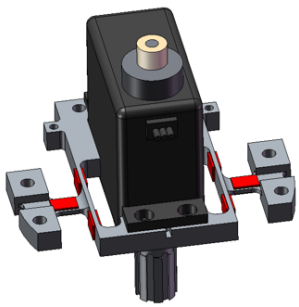
The following subsection is adapted from [6]

Mamba is the second snake robot based on the OAL premise created at the Norwegian University of Science and Technology, shown in Figure 2. NTNU used the snake robot for experiments on both the ground and in water. One of the main differences between *Mamba* and the previous version Kulko presented by Pål Lilljebekk was the new intrinsic force/torque sensor system.

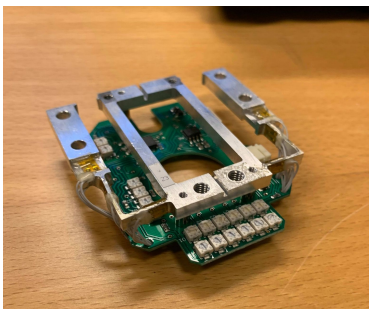
Figure 2: The Mamba snake robot, courtesy of [7]

The sensor system of the Mamba snake robot

The force/torque measurement system in the *Mamba* robot is based on strain gauges. As shown in Figure 3, the strain gauges are mounted on an aluminum frame perpendicular to each other, enabling it to measure force and torque on three axes, making it a 6-axis/multi-axis force torque transducer. Figure 3a and Figure 3b present two different versions of the sensor system, as several attempts have been made to create and improve the system. The situation, however, as indicated by *Fredrik Veslum* [7], showed that both systems had problems with noise, hysteresis, and temperature deviation. This, including that Mamba's electronics had become outdated at the time of Fredrik Velum's experiments, made the robot unfit for further development.



(a) 3D-model of an older version(v0) of the force-torque measurement system



(b) The strain gauge sensor system (v3) mounted on the amplifier circuit board

Figure 3: The strain gauge sensor system in the *Mamba* snake robot. Courtesy of [7]
.

3.2.2 The Boa snake robot

After the problems with *Mamba*, the HOAL team started developing a new snake robot called *Boa*. It is the third snake robot based on OAL created at NTNU and is currently close to being in a runnable state and being used for experimentation. Compared to the previous snakes, the Boa snake robot employs a new version of Pål Liljebäck's measurement system based on industrially created sensors instead of *Mambas* in-house design.

Figure 4: The Boa snake robot, courtesy of [5]

The sensor system of the Boa snake robot
Boas measurement system, similar to Mambas, still uses strain gauge measurement technology to measure forces and torques but has instead opted into using commercially available sensors combined together. This sensor combination is developed by the HOAL team in cooperation with [6] and uses two different sensors locked together by a 3D-printed linkage to achieve necessary measurements. Boa is, however, still under development, so the sensor system solution may have changed since the time of writing.

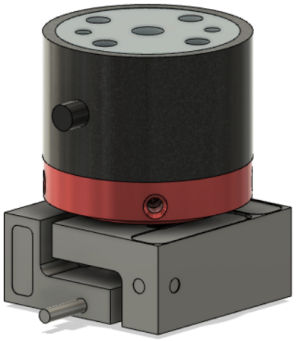


Figure 5: Boa sensor system, courtesy of [6]

3.3 Previous simulators created for snake robot research

The topic of creating a viable simulation environment at ITK for a snake robot has been attempted twice before. The first attempt, created in 2018, was called *Snakesim* shown in [8], and the second attempt was a master thesis produced in 2019 called *Simulator for Obstacle Aided Locomotion in Snake Robots* shown in [9].

3.3.1 SnakeSim

The first simulator, as mentioned, was called Snakesim [8] and was a ROS[10] + Gazebo[11] based simulator created to provide a virtual rapid-prototyping framework for *Perception Driven Obstacle-Aided Locomotion (POAL)*. It featured the ability to simulate a snake robot in environments cluttered with obstacles, as well as the ability to add or remove sensors as necessary. A depiction of how the software system communicated can be seen in Figure 6. According to the research article's conclusion, the simulator worked very well. It made it possible to swap out the sensor reading portion in the simulator with the sensors from the physical snake robot. In this way, it was a plug-and-play system that could first asses how a snake robot would react to an environment and then experiment with the robot in the same environment.

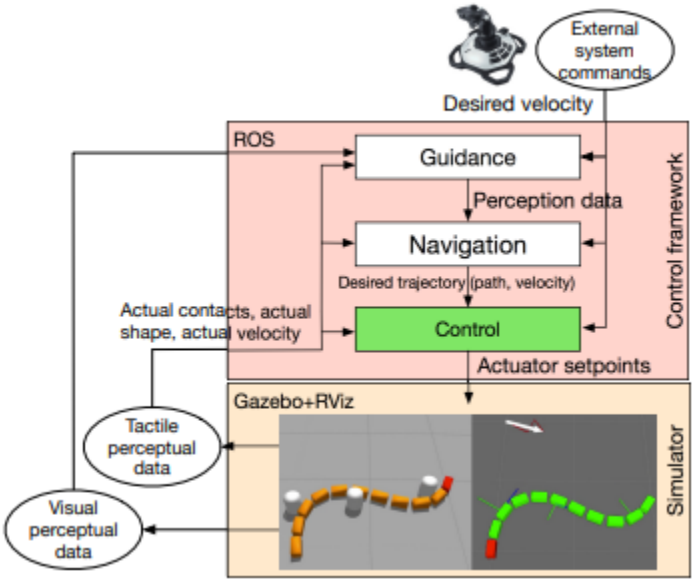


Figure 6: Snakesim software architecture, courtesy of [8]

3.3.2 Simulator for OAL

The second simulator created at NTNU for snake robot research was *Simulator for Obstacle Aided Locomotion in Snake Robots*[9]. This simulator was developed as part of a master's thesis and was created with guidance from one of the creators of Snakesim. Instead of using an existing simulator framework, this simulator was developed solely in Matlab, implementing the physics functions directly. This allowed the simulator to be tailor-made towards testing specific concepts, but as the paper discusses, there were cases where the laws of energy conservation and momentum were violated. This means it ended up with simulations that do not match the real world, making the results of the simulations less valuable. The paper does however conclude that the premise of *HOAL* should be further researched.

3.3.3 Motivation for creating another novel snake simulator

As shown in Chapter 3.3.1 and 3.3.2, both present a viable option to provide simulations for snake robot research. However, due to different circumstances, such as lack of version control and lack of git commits, both are either lost or nonfunctional as of writing. They are by no means useless as they provide an abundance of information on how to develop a simulator, but it does require a new one to be created. This also underlines the importance of thorough version control for the coming master thesis.

3.4 The structure and theory of physics simulators

Another important subject is the concept of what a simulator is, what it consists of, and its use cases. This is because a simulator is a widely used term to explain anything which digitally mimics a real-world phenomenon, but a simulator usually consists of multiple programs interacting. Here namely, the physics engine and the rendering engine work in unison, as shown in Figure 7.

3.4.1 The rendering engine

The rendering engine is the software responsible for controlling and rendering graphics in a simulation. It does this by continuously receiving information from the physics engine and using the positional data to appropriately move the objects in the simulation. It is a valuable tool to provide a more human-interactable environment and varies from simulator to simulator. Some of the most common rendering engines are *OpenGL*[12] used by MuJoCo[13] and PyBullet[14] and *OGRE*[15] used by Gazebo[11] and Isaac Sim[16].

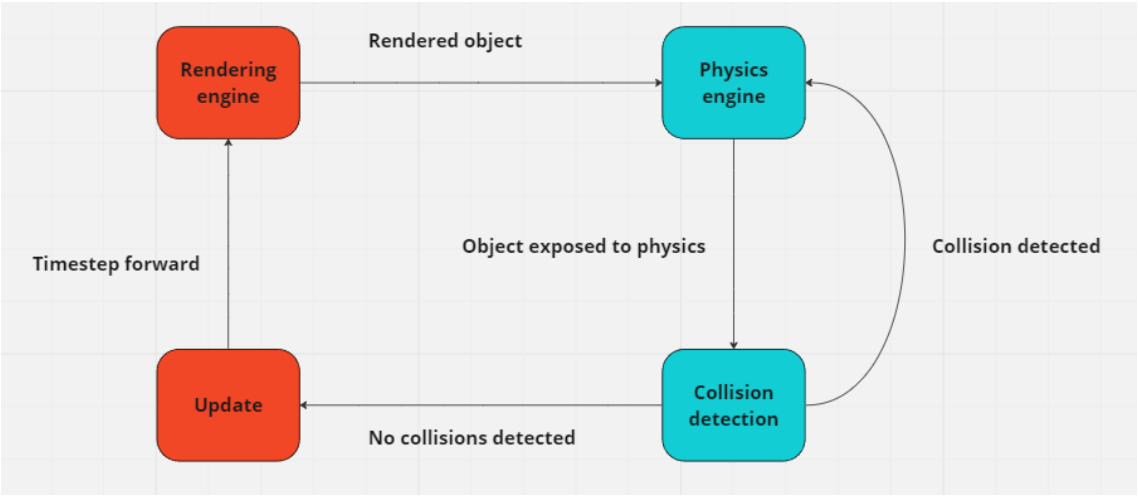


Figure 7: The simulation loop where the rendering engine is marked blue and the physics engine is marked orange

3.4.2 The physics engine

The physics engine is the piece of the simulator that calculates how the simulation should proceed, from one timestep to the next. It is essential in any real-time simulation system, and many different versions exist. The physics engine can be further broken down into two parts, namely *collision detection* and the *physical engine calculation* part. These two work together to calculate and check for collisions continuously in the simulation, making the movement and behavior of objects in the virtual world appear more realistic.

3.5 Contemporary physics engines

In today's market, there exist many viable physics engines. However, some stand out and appear more often than others in both research and games. Four of these physics engines, which are relevant to robotic research, are presented here.

3.5.1 Bullet

Bullet [14] is, at the time of writing, one of the more implemented physics engines within robotic research. It is an open-source project built in 2013 and is offered by many front-runners within simulation, such as Gazebo[11], CoppeliaSim[17], and many video game platforms. It is written in C/C++ and is continuously updated by the community to help it evolve and stay updated with current needs. Its latest release at the time of writing is Bullet 3.2.4, released on the 25. April 2022.

3.5.2 ODE

ODE [18] is another of the more used physics engines similar to Bullet. It is also open source and is present in almost all widely used robotics simulators such as Gazebo, Webots[19], and CoppeliaSim. It is written in C/C++ and was first released on 8. may 2001, making it relatively old from a technological perspective. Its last stable release was July 30, 2020, meaning it is still being updated to this day.

3.5.3 MuJoCo

MuJoCo [13] is one of the newer physics engines starting to make more and more appearances in simulator usage. It is known to be very good at specifically collision physics [20], making it interesting for OAL's premise. It is likewise to the others written in C/C++ and is open source. MuJoCo was first released in 2015 but re-released as MuJoCo 2.0 in 2018. Its latest stable release is MuJoCo 2.3.0, released on the 18. Oct 2022 and is continuously being updated through community contributions.

3.5.4 PhysX

PhysX [21] is an open-source real-time physics engine developed by Nvidia [22] as part of their Nvidia GameWorks software suite. It was initially released as just PhysX in 2003 but has since then been updated all the way to the newest PhysX 5, which has its latest stable release on October 12, 2021. It is mainly known as a video game engine, but in NVIDIA's newest project Isaac Sim, it has been included as the sole physics engine meaning Nvidia is trying to make it more applicable to research as well. It has interfaces with both C and Python.

3.6 Common weaknesses in contemporary physics engines

In general, one of the common issues with physics engines is the concept of collision detection and its problem with high-velocity objects. It is something that stems from the very nature of physics engines and how they use time steps to move physics forward. This means that if two objects manage to collide between time steps, they are not caught by the physics engine leading to objects intertwining. This further leads to physics breaking down and, as a result, can ruin simulations. This kind of collision detection that evaluates objects at every time step is called DCD (*Discrete collision detection*). However, a solution to this problem exists, which is called CCD (*Continuous collision detection*). CCD, instead of just checking for collisions at each time step, also tries to predict the movement of the object until the next time step occurs. In this way, it is able to prevent collision problems but does, in turn, require a lot more computational power meaning that it is not always a viable solution when real-time is of the essence. It is worth mentioning that ODE, Bullet, MuJoCo, and PhysX all offer CCD.

3.7 The MJCF robot modeling language

Before a robot is able to be simulated in any way, shape, or form, it first has to be declared in a way a computer understands. This is where the robot modeling language MJCF becomes important and is the primary way the simulated robots have been created during this master thesis. More information on how MJCF files work can be found in Reference [23], but the most important parts are covered in this subchapter.

3.7.1 The structure of MJCF

The MJCF modeling language is an XML-based language consisting of many data object types combined in varying parent/child relationships to create fully functioning robots. It creates these parent/child relationships by using indentation quite similar to Python, where the less indented object becomes the parent of the more indented object. An example of how a simple MJCF formatted file looks can be seen in code snippet 8. In this

example, one can see how the primary parent < *mujoco* > contains the child < *worldbody* > (the simulation world), which again has the child < *body* > (the box_and_sphere). If one executes the XML example, it will produce the object shown in Figure 9.

```
<mujoco>
  <worldbody>
    <light name="top" pos="0 0 1"/>
    <body name="box_and_sphere" euler="0 0 -30">
      <joint name="swing" type="hinge" axis="1 -1 0" pos="-.2 -.2 -.2"/>
      <geom name="red_box" type="box" size=".2 .2 .2" rgba="1 0 0 1"/>
      <geom name="green_sphere" pos=".2 .2 .2" size=".1" rgba="0 1 0 1"/>
    </body>
  </worldbody>
</mujoco>
```

Figure 8: MJCF example code from deepminds MuJoCo python tutorial [24]

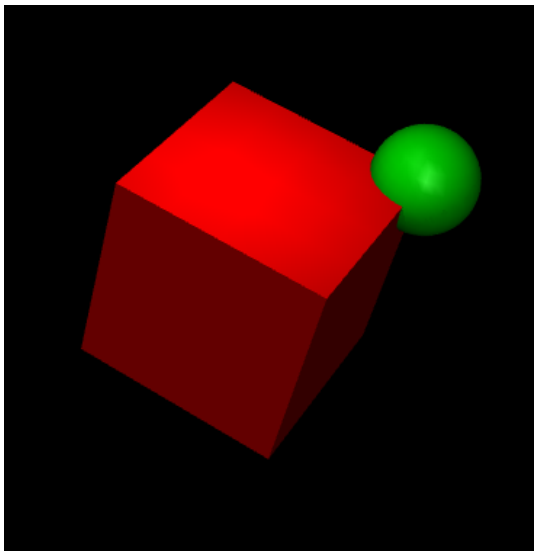


Figure 9: Executed version of code snippet 8 from deepminds MuJoCo tutorial [24]

3.7.2 The main classes of MJCF

As seen in code snippet 8, there are many XML tags that are used to create Figure 9, such as worldbody, body, joint, and geom. In the MJCF file format, there exists probably upwards of 100 different tags one can use to customize one's robot but to help simplify for the reader, this chapter will cover the most important ones used in the thesis. The entire XML file for one configuration of the snake robot can be seen in Appendix 3 for a more elaborate MJCF structure.

XML-tag	About
`mujoco`	Tells the compiler that this is a MuJoCo model. Other examples could be the URDF tag or HTML tag.
`worldbody`	The main simulation world, where all simulated objects are added.
`option`	Specific simulation options for the physics engine such as integrator type, timestep size, and gravity.
`asset`	Used to define texture and material options to give color and patterns to objects in the simulation.
`default`	Used to define properties for all instances in a simulation to help reduce code duplication. (For example, if one declares a joint with range = -90 90, every joint in the simulation will have this property if not told explicitly otherwise).
`body`	An abstract container for physical elements such as geom and joints.
`geom`	A physical geometrical figure such as cylinder, sphere, or box.
`joint`	A physical joint such as a hinge, ball, or slide.
`actuator`	Collective tag for actuators such as servomotors and muscles.
`sensor`	Collective tag for sensors such as accelerometer, velocimeter, and gyroscope.

Table 1: The main XML tags used in the snake robot

3.8 Friction and snake robots

One important subject to discuss regarding snakes and how they propel their bodies is the concept of friction. This is because snakes behave differently based on the underlying friction type of the ground, meaning for simulation purposes, it is important to realize what friction types are being used.

3.8.1 Friction

Friction is a force that occurs when two surfaces come into contact and attempt to move or slide relative to each other. This will create a force that moves in the opposite direction of motion between the two objects, thus either preventing or reducing momentum. Friction can be divided into different categories, where two of the main ones are *static friction* and *kinetic friction*. Static friction, also known as resting friction, is friction that prevents objects from moving while standing still. Static friction provides an increasingly counteractive force toward momentum until it is overcome, allowing an object to start sliding or moving. Kinetic friction is friction that acts on two surfaces moving relative to each other. Both these types of friction refer to the concept of *dry friction* or *Coloumb friction*, which is widespread in simulator usage [25] including MuJoCo.

3.8.2 Isotropic and anisotropic friction

Friction can be divided into several categories based on the nature of the friction and the objects it interacts with. Two other categories that refer to different properties of friction in relation to the directionality of forces and motion are *isotropic* and *anisotropic* friction. Isotropic friction occurs when the frictional properties are the same in all directions. In other words, regardless of the direction the object is moving, the amount of dry friction it receives will be the same. On the other hand, anisotropic friction is when the frictional properties are not the same in all directions. This means an object can receive more counteractive friction when moving upwards than left or right. Figure 10 shows an example of this concept, where the arrows have varying sizes to indicate varying amounts of friction.

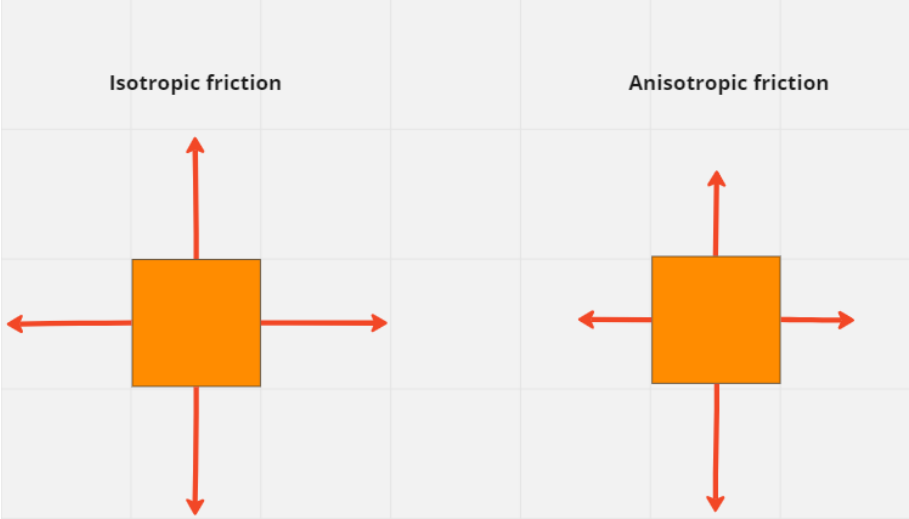


Figure 10: Example of isotropic (left box) and anisotropic friction (right box). The boxes indicate the same object, and the arrows indicate the amount of friction forces the box receives moving in each direction. One can clearly see here that when isotropic friction is present, the friction amount is the same in every direction, while this is not the case for anisotropic friction

3.8.3 Effects of isotropic and anisotropic friction on snake robots

Snakes behave differently based on the friction type of the material it moves upon, meaning whether the material allows for anisotropic friction or not. Suppose the material allows for anisotropic friction by allowing the snake to use the scutes on its belly to achieve more friction in one direction than the other. In that case, it is able to create propulsion in that direction, as stated by Pål Liljebäck [1]. If the material does not allow for anisotropic friction due to the scales' inability to achieve contact with irregularities in the ground, it will simply stay in place during movement. This concept is the basis for lateral undulation mentioned in Chapter 3.1.

4 Software specification

Before starting the implementation of the snake robot simulator, it was decided to create a software specification outlining the main goals of the software. This chapter presents these main goals as bullet points from Subchapter 4.1 to Subchapter 4.6. Lastly, this chapter presents a task list that was used to prioritize implementation assignments to help prevent *mission creep* during development. Mission creep is the concept of a project continuously adding more tasks and goals to its original development plan. This can often lead to the project slipping out of hand, causing missed deadlines and lacking functionality. A more compact version of the software specification bullet points presented in this chapter can be seen in Figure 11.

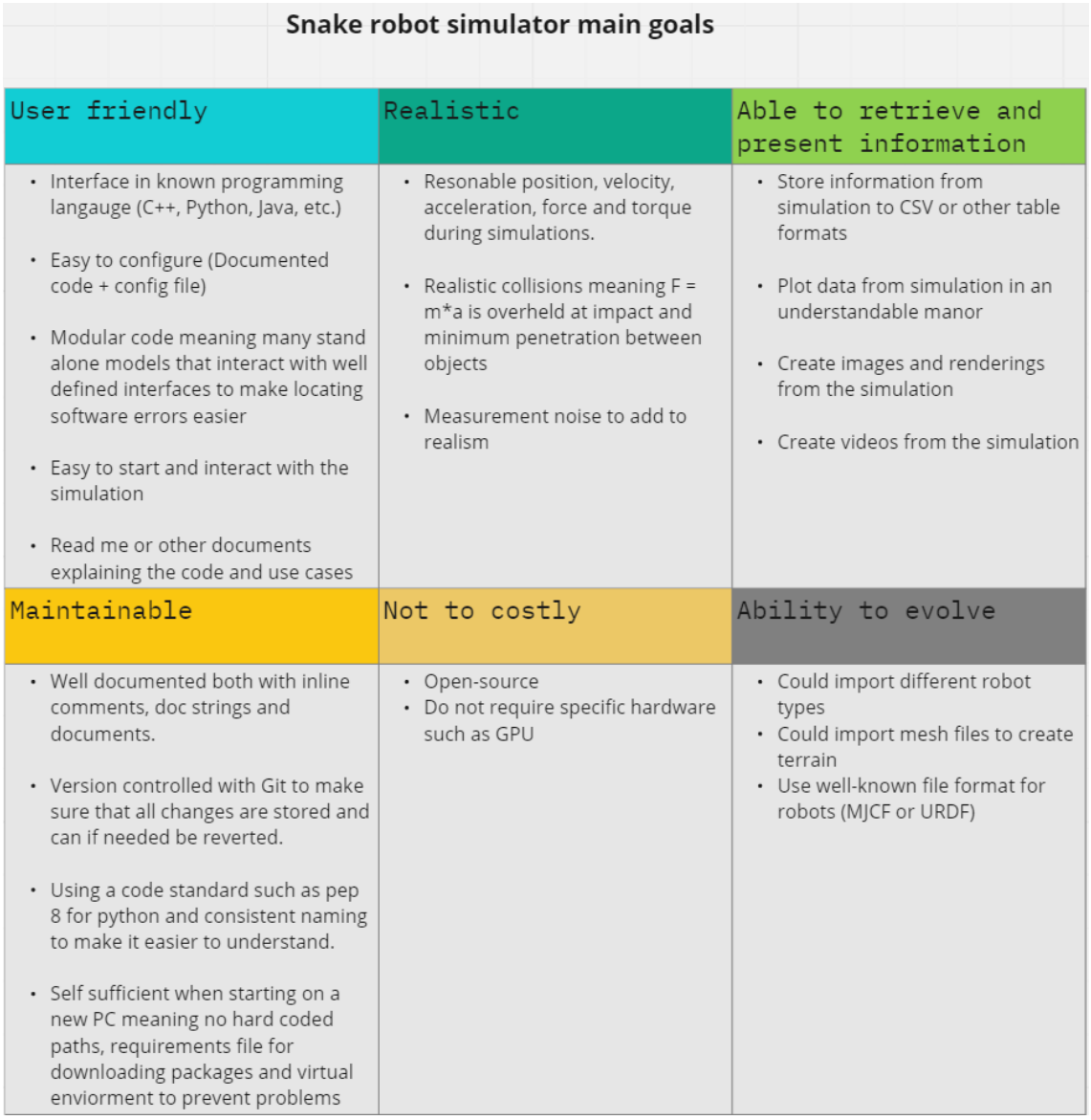


Figure 11: Compact version of the software specification goals presented in this chapter

4.1 Design goal: User-friendliness

The first point on the list was making sure the software was user-friendly. This was important as the software, upon completion, was to be handed over to other researchers that may or may not have strong programming knowledge but still should be able to use the software. This meant that the software had to be produced and structured in such a way that anyone with general programming and math knowledge should be able to understand what was happening and should also be able to edit the simulation to their needs. To achieve this, the software should therefore contain:

1. Interface in a known programming language (Python, C++, etc.). Doing this increases the likelihood that researchers could use and test the software even if they are not very good at programming.
2. Easy to configure (Well structured code + config file). This will help reduce the turnover time between different experiments and simplify configuring the software for specific purposes.
3. Modular code meaning many stand-alone models that interact with well-defined interfaces to make locating software errors easier.
4. Easy to start and interact with the simulation meaning clear entry points for where to start the simulator, where to control the robot, where to add more objects to the simulator, and where to store data.
5. README file or other documents explaining the code and use cases to help reduce the necessary learning time to use the software.

4.2 Design goal: Physical realism

The second point on the list was ensuring that the simulations presented from the simulator were realistic. This meant that the simulator should be able to simulate an object, in this case, a robotic snake, and the values returned from the simulator, such as position, velocity, acceleration, interaction force, and torque, should all be reasonable and correct values. This is important so that experiments run in the simulator realistically mimic experiments run on the real-world counterpart, allowing control algorithms to be developed on the simulator and used for the real-world snake robot. To achieve this, the software therefore had to be able to:

1. Deliver reasonable physical values for position, velocity, acceleration, force, and torque. By reasonable, the project means there should be a connection between the size of the snake and how much force is present. For example, if a snake has a linkage that is 15 cm long and 8 cm across, there should not be more actuator torque present than about $\pm 3nm$.

2. Complete realistic collisions (meaning that newtons law of Force = mass times acceleration holds and that objects are not able to penetrate one another)
3. Use an established physics engine to prevent breaking the laws of energy conservation and momentum as in a previous simulator attempt. See Chapter 3.3.2 for more information.
4. Emulate measurement errors on all measurements, such as position, velocity, acceleration, force, and torque, to help mimic real-world conditions.

4.3 Design goal: Data acquisition and presentation

The third point on the list was creating software that was able to both fetch, store and present different types of simulation data in an understandable manner. This was important so that during the simulation run time as well as after, it was possible to scrutinize the data and create plots, renderings, or videos for research and research articles. This meant that the software had to be able to:

1. Store information from simulation to CSV or other table formats
2. Plot data from simulation in an understandable manner
3. Create images and renderings from the simulation
4. Create videos from the simulation

4.4 Design goal: Maintainability

The fourth point was making sure that the software should be easily maintainable, meaning that if changes had to be made, it should be described somewhere how the code works, what each part does, and how to apply changes. This meant that the code should be:

1. Well documented with inline comments, doc strings, and documents.
2. Version controlled with Git [26] to ensure all changes are stored and can be reverted if needed.
3. Using a coding standard such as pep 8 [27] for Python and consistent naming to make it easier to understand.
4. Runnable on any Windows operating system without significant changes to the code so that when the program is run on a new computer, it can be started quickly. This means no hard-coded paths, requirements file for downloading packages, and a virtual environment to prevent version collisions between packages and Python versions.

4.5 Design goal: Affordability

The fifth point was ensuring the simulator did not require too much monetary investment. This was important as this is supposed to be used for research purposes, and therefore the lower the cost, the better. The best case scenario would be if it were completely open source so that it could easily be shared with other researchers. This meant that the simulator, if possible, should be:

1. Based on open-source works to reduce cost and promote accessibility for other researchers
2. Do not require specific hardware such as Graphics Processing Unit (GPU)

4.6 Design goal: Expandability

The final point in the software specification was making sure the software had the ability to evolve. This means it should be attempted as far as possible to write code that does not lock the software to one single XML-type robot so that it can be adapted for other uses later on if necessary. This means that the code should be able to:

1. Simulate different robot types, such as snake or humanoid, without too many changes to the main simulation file
2. Import mesh files to create terrain so that new terrain can easily be added
3. Use a well-known file format for robots such as MJCF discussed in Chapter 3.7 or URDF discussed in this Article [28]

4.7 Prioritized list of desired features

The priority list is tabular based on the software specification that tries to create more concrete tasks and priority levels based on the goals of the specification. It is divided into four main priority levels, critical, high, medium, and low, and the project aimed to at least implement everything in the critical and high sections of the list. This is because these points were necessary to make the simulator useful enough for research purposes. The medium priority parts are nice to have features that simplify the simulator's usage, as well as add other useful tools to increase the simulator's value. Lastly, the low-priority tasks are tasks that should be looked into if time allows or can be used for future work after the thesis is complete.

Features for the snake robot simulator

	Task	Priority
☐	Possibility to import robot from MJCF or URDF format	Critical ⚠
☐	Ability to return all relevant physical values from simulation either as a pandas data frame or CSV	Critical ⚠
☐	README file on how the software works	High
☐	Rewrite XML file to be dynamic so it is possible to change snake from config file	High
☐	UML diagram for proposed communication between software interfaces	High
☐	Config file that allows you to easily customize simulation options (works for headless too)	High
☐	Auto world generation (meaning creating maps filled with randomly placed cylinders)	High
☐	Ability to plot interesting information such as positions or others (probably controlled from GUI)	Medium
☐	Possibility to simulate measurement noise to make the simulations more realistic	Medium
☐	Auto world generation with varying surfaces such as rocks, cement, and so on	Medium
☐	Ability to import mesh-files from pictures to create terrain	Medium
☐	Research machine learning gyms for use towards the simulator	Low
☐	Research cloud computing possibilites for simulator usage	Low

Figure 12: Prioritized list for feature implementation

5 Development of the simulator

After the software specification and priority list were completed in cooperation with the supervisors of the master thesis, development started on the simulator. Based on the results from the project assignment shown in the result and discussion chapters in Appendix 12.3, it was decided to continue with Isaac Sim as the main simulation platform. This did however change to MuJoCo during the development process as explained in Chapter 5.2.

5.1 Development with Isaac-sim

Isaac Sim [16] is as presented in Chapter 4.1 of Appendix 12.3 a semi-free to use physics development tool that can be used for simulating different kinds of robotics. By semi-free, it means that if it is used for research or personal projects, it is free to use, but for commercial products, one has to pay to access the software. It is built on top of the PhysX physics engine and was chosen for further development during the preliminary project due to Isaac Sim containing all necessary tools to accurately simulate physics along with future potential in cloud computing and machine learning (see the conclusion of Appendix 12.3).

Isaac-sim, once installed through Nvidia Omniverse [16], gives the user access to an extensive library of prebuilt Python functions as well as documentation. This meant that after installation, development started with small steps, where the first step was to create

a simulation from a designated Python script. This Python script had to be able to spawn a simple object (cube) into a simulated world and return data from the object, such as position and velocity. Due to existing demonstrations, this was relatively simple, and some example code from Nvidia can be seen in Listing 1, which achieves this.

Listing 1: Example code for spawning a cube into a simulated world in Isaac Sim. Courtesy of Nvidia [16]

```
from omni.isaac.examples.base_sample import BaseSample
import numpy as np
# Can be used to create a new cube or to point to an already existing
    cube in stage.
from omni.isaac.core.objects import DynamicCuboid

class HelloWorld(BaseSample):
    def __init__(self) -> None:
        super().__init__()
        return

    def setup_scene(self):
        world = self.get_world()
        world.scene.add_default_ground_plane()
        fancy_cube = world.scene.add(
            DynamicCuboid(
                prim_path="/World/random_cube",
                name="fancy_cube",
                position=np.array([0, 0, 1.0]),
                scale=np.array([0.5015, 0.5015, 0.5015]),
                color=np.array([0, 0, 1.0]),
            ))
        return
```

The next logical step was then to start adding specific robotic structures through either the recognized URDF format [29] or MuJoCo's MJCF XML format 3.7.

5.2 Problems with Isaac-sim

Once the project started investing time in adding its own URDF or MJCF files to the simulation, errors began to appear. An already prebuilt URDF example was not able to load into the simulator due to a variety of issues which made the project look more into using the MJCF file format. A simple snake-like structure was then created in the MJCF format, and after using Isaac Sims' built-in UI MJCF loader, a snake was able to appear in the simulated world. However, once the same results were attempted to be replicated through the use of a Python script, the program only seemed to return errors. After about a week of trial and error, a patch update came to Isaac Sim, which seemingly added the missing MJCF functionality, making the snake able to load from a Python script. This however meant that functionality that already was said to be in the simulator was, in fact, being added over time, which meant that other errors that started to appear could also be from missing functionality. This would lead to the master thesis being held up

by waiting for continuous patch updates, posing a significant problem. The choice was therefore made to switch over to the MuJoCo physics engine as the project already had developed an MJCF XML file that could be ported into MuJoCo instead.

5.3 Switching to MuJoCo

Once the project had figured out that MuJoCo seemed to be a better solution, testing began on spawning the snake XML file into a simulated world. This was achieved relatively quickly due to MuJoCo's simple setup and many tutorials. Furthermore, it was possible to choose between either using the C++ MuJoCo build or using its quite developed Python API. Both frameworks were tested, but since Python was already used for development towards Isaac Sim, it was decided to continue with Python for MuJoCo as well.

5.4 MuJoCos simulator platform

MuJoCo [13], including being a physics engine, also features a fully-fledged simulation framework combining its physics engine discussed in Chapter 3.5.3 with the OpenGL rendering engine. It was initially developed by Robotics LLC [30] but was acquired and made freely available by DeepMind [31] in October 2021. The simulator platform has some key features that are useful to understand to be able to use MuJoCo appropriately. These key features are all well explained in [32], but a short summary taken from MuJoCo's documentation can be seen in Figure 13.

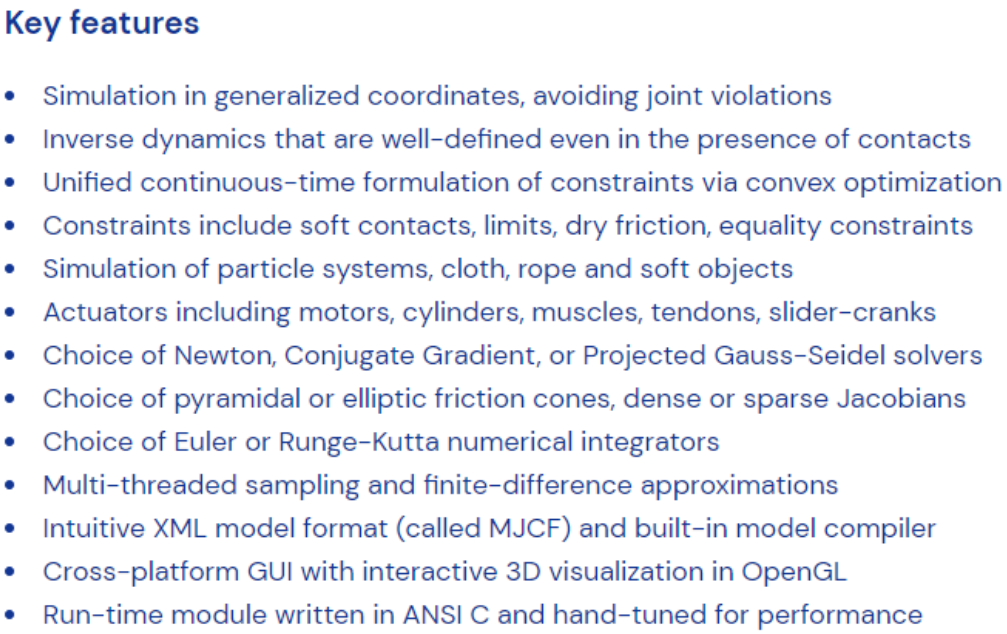


Figure 13: MuJoCo key features from MuJoCo's documentation. Courtesy of [32]

6 The design and structure of the simulator

This chapter details the software's structure by explaining the Python simulator's main modules. It is helpful for getting a better understanding of how each module interacts and what their purpose is within the simulator. This chapter also explains different choices that were made during development and why some solutions were chosen over others. The complete code is as per writing on GitHub [26] under the private repository SimSerpent [33]. For access, refer to the owner of the GitHub organization the repository is posted under. This repository also contains a ReadMe file that details how to use the software, even though this chapter also explains it in detail.

6.1 Software module setup

Before writing any software, a diagram was created to help showcase how the software should be structured, what modules should be developed, and how they should interact with each other. This helped prevent unnecessary bloating of software and repetitive code being created. The final diagram can be seen in Figure 14 after it has been revised several times. It consists of many files but can generally be broken down into four main modules: the files that create the model, the main simulation files that run the simulations, the storing/plotting files for data, and the configuration file that controls many aspects of the simulations.

6.2 Simulation module

The first module in the snake robot simulator is the simulation module which handles all aspects of the simulation from beginning to end. This module consists of four main Python files, `simulate_snake.py`, `get_snake_info.py`, `ctrl_snake.py`, and `view_model.py`. This module is responsible for the following:

- Starting the simulation
- Stepping through the simulation
- Fetching data from the simulation during run time
- Controlling the robot present in the simulation
- Viewing models outside of the simulation

This module uses a few simple data structures that contain almost all the information necessary for the simulation. These are `MjModel`, `MjData`, and `MjvOption`, which are

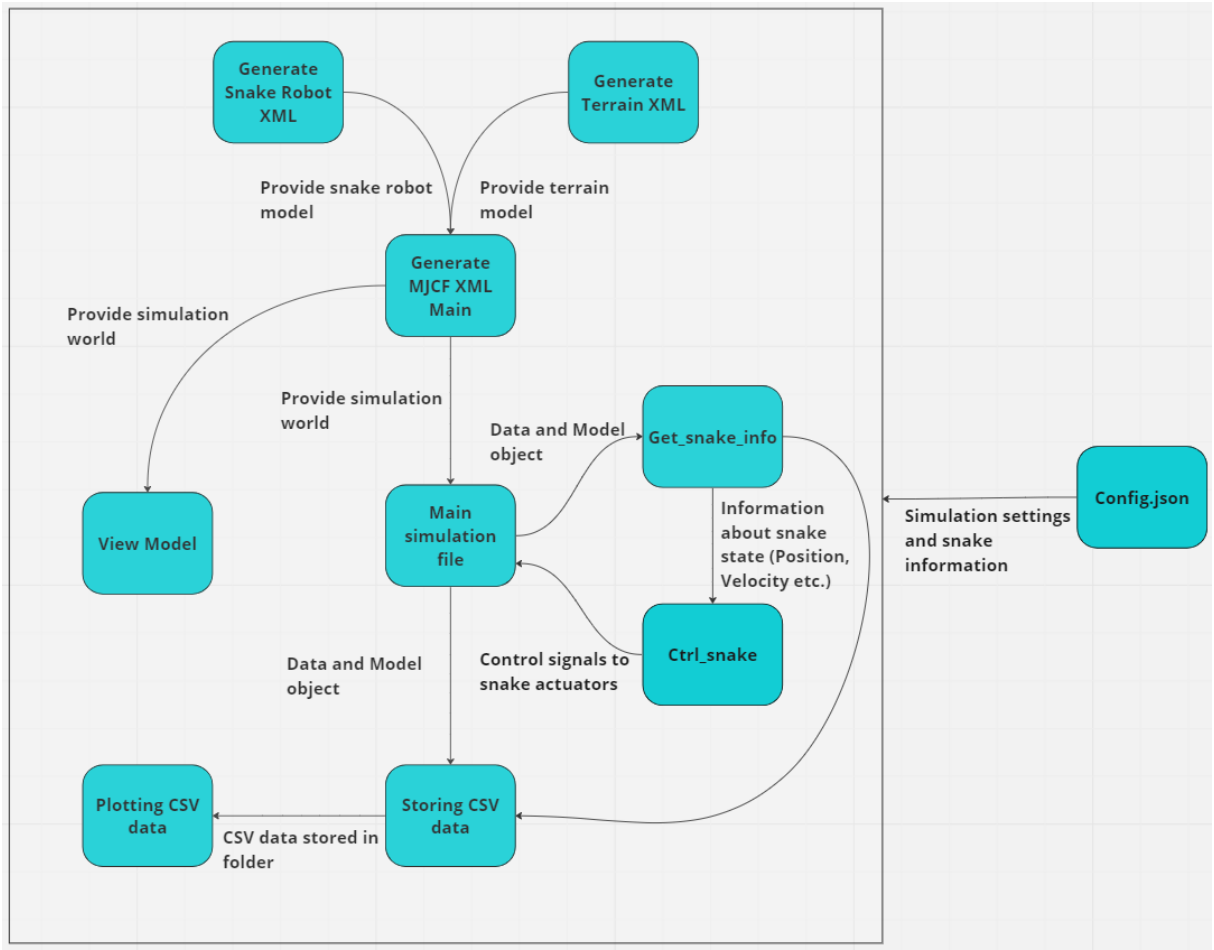


Figure 14: Software diagram explaining relationships between the different modules in the program. The four main modules are the simulation module, the configuration module, the XML generation module, and the data acquisition module

shortly explained in tabular 2. For more information, see the MuJoCo documentation found in [13]:

Data structure	About
`Mjmodel`	The main data structure holding the MuJoCo model. It is treated as constant by the simulator and contains information such as the number of bodies, joints, generalized coordinates, and so on.
`Mjdata`	The data structure holding the simulation state. It is the workspace where all functions read their modifiable inputs and write their outputs.
`MjvOption`	The data structure with simulation options. It corresponds to the MJCF element option. One instance of it is embedded in `Mjmodel`.

Table 2: The different main data structures found in a MuJoCo simulation

6.2.1 simulate_snake.py

The `simulate_snake.py` file is responsible for initializing, starting, running, and ending the simulation. It consists of three main parts, a set of callback functions, the simulation initialization, and the main simulation loop. The callback functions allow interaction with the simulation during runtime and must exist in this file, otherwise, the simulation cannot call them. The simulation initialization is, as the name implies, an initialization, and this creates all elements and data structures necessary for the simulation. Lastly, the main simulation loop runs until a set amount of simulation time has passed, which can be edited in the configuration file. An example explaining the execution of the file can be seen in Figure 15.

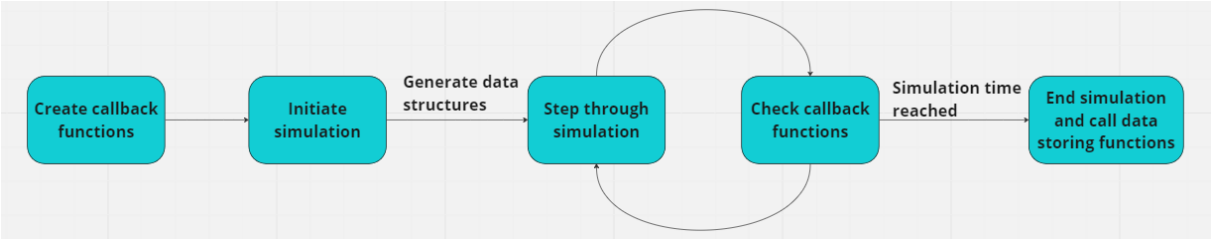


Figure 15: Main simulation loop for one run of the `simulate_snake.py` python file

6.2.2 get_snake_info.py

The `get_snake_info.py` file is responsible for returning all necessary information for every linkage and joint that exists within the snake robot. This is achieved through many simulated sensors and existing API functions built into MuJoCo. It uses the `Mjdata` data structure discussed in Table 2 as this structure contains the updated simulation state at every time step. It contains four main functions being:

Function	About
`get_link_info`	Returns a dictionary with all information necessary to determine a link's whereabouts, meaning position, velocity, acceleration, current angle, and angular velocity
`get_joint_info`	Returns a dictionary with all necessary information to determine a joint's state such as joint angles and actuator torques in the joint.
`get_contact_info`	Returns a dictionary containing a list of all contacts present in the simulation as well as the distance between those objects to check for penetration.
`get_force_info`	Returns a dictionary containing the size of all external and internal force interactions for each linkage.

Table 3: The main information functions found in the simulation

6.2.3 ctrl_snake.py

The `ctrl_snake.py` file controls the snake robot in the simulation environment during runtime. It achieves this by having access to the actuators declared in the `snake.xml` file and using the `data.ctrl` function to give commands to the actuators. Three types of actuators are present in the `snake.xml` file: a torque actuator, a position actuator, and a velocity actuator. Their names imply their purpose meaning that the torque actuator is a torque-controlled servo motor that allows you to provide a set amount of torque directly to the actuator. The position actuator is used for position control and enables you to provide the wanted angle in radians as input. The position actuator itself then determines the amount of torque necessary to achieve that position. Lastly, the velocity actuator is used for velocity control and allows you to control how fast the joint it is connected to is allowed to move. An example figure of how this works can be seen in Figure 16.

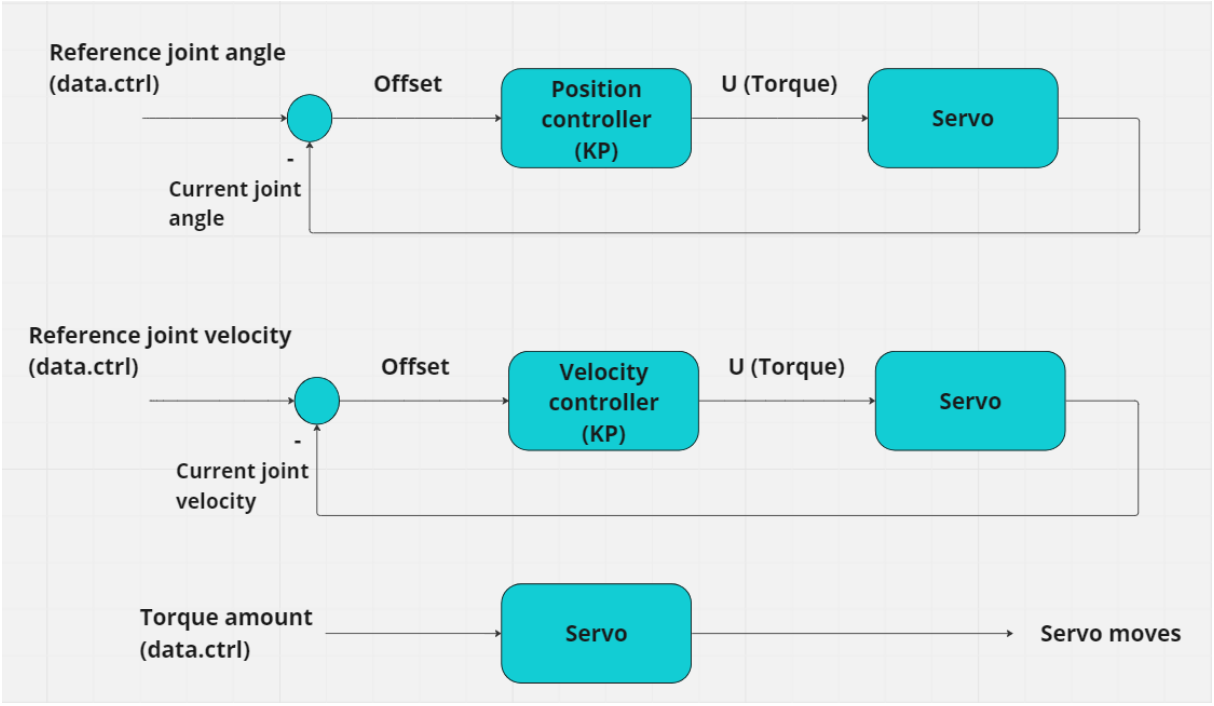


Figure 16: Diagram showcasing torque, position, and velocity control

As mentioned previously and seen in Figure 16, one has to use `data.ctrl` to send control signals to each actuator. `Data.ctrl` is simply a list containing a reference to each actuator, meaning that the first `#nr_of_links - 1` is the torque controllers as they are declared first in the XML file. The next `#nr_of_links - 1` are the position controllers and the last `#nr_of_links - 1` are the velocity controllers. This should be kept in mind when working with the controllers. Another important note is that all the actuators are SISO meaning single input, single output. One, therefore, has to combine position and velocity servos to, for example, create a PD controller.

6.2.4 view_model.py

The final piece of the simulation module is the `view_model.py` file. This file calls MuJoCos built-in viewer class, allowing the user to bring up a simulator window and portray the selected MJCF file in a simulated world. This viewer also contains many valuable tools to help study the model before doing simulations, such as the ability to control motors manually, produce graphs for position and velocity, and much more. A screenshot of this viewer with a novelty snake robot can be seen in Figure 17.

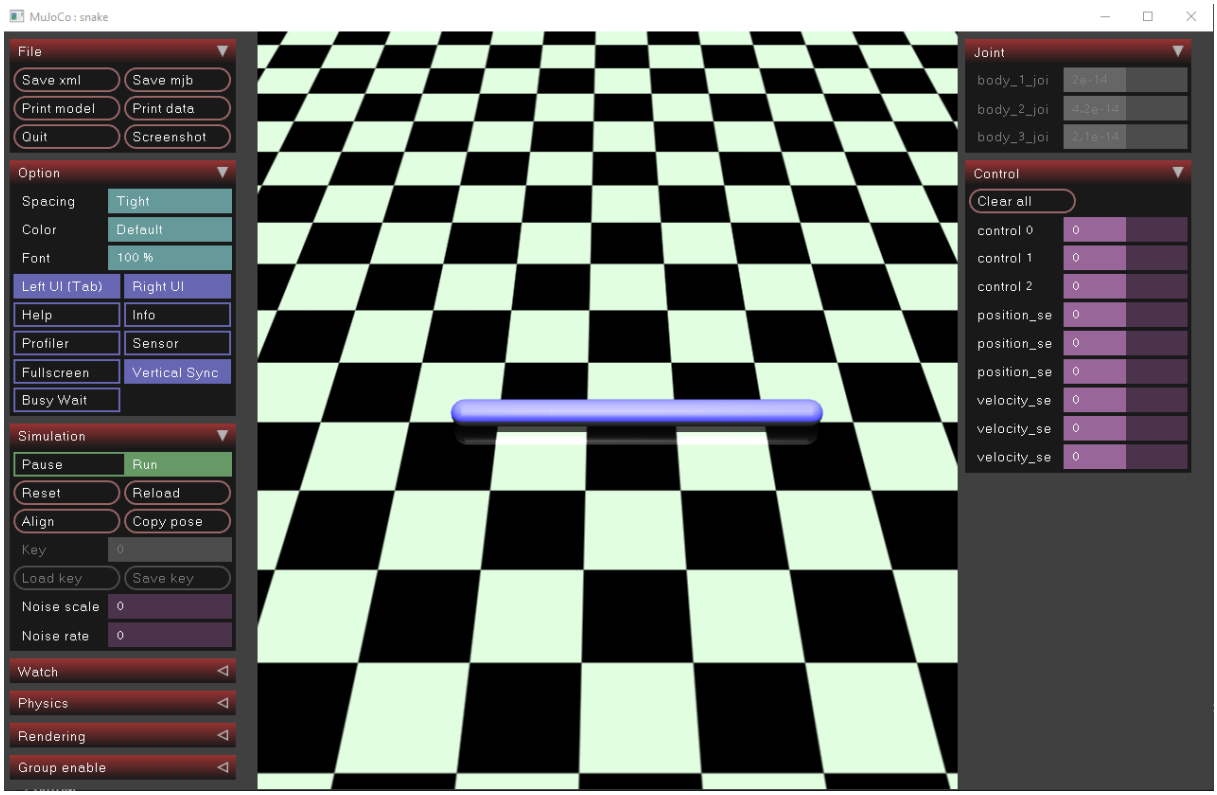


Figure 17: MuJoCo viewer in use

6.3 Configuration module

The second module in the snake robot simulator is the configuration module. This module works as a configuration hub for all parts of the simulator and allows users easy access to many different settings to help customize their simulation. This module consists of only one file, the `config.json` file, and can be seen in Appendix 2. The configuration file is divided into seven categories: general, visual, snake specifications, control parameters, terrain, prebuilt experiments, and storing data. This categorization aims to facilitate ease of use for users by separating configurations based on the specific aspects of the simulation they modify. A short summary can be seen in Table 4.

Config class	About
`general`	Controls general simulation options such as simulation time, simulation timestep, and integrator type.
`visual_options`	Controls visualization options for the simulation to help showcase important features such as contact points and contact forces.
`snake_specifications`	Controls how the snake should look, the amount of torque it should have, placement, and other useful options.
`control_parameters`	Controls parameters used for controlling the actuators in the snake such as Kp variables and PID variables.
`terrain`	Gives the option to include prebuilt terrain into the simulation world.
`prebuilt_experiments`	Gives the option to conduct prebuilt experiments to showcase the simulator use cases.
`store_data_to_csv`	Gives the option to select what data to store as CSV if the user wants to store anything at all.

Table 4: The main configuration categories found in the simulator

6.4 Generate simulation world module

The third module in the snake robot simulator is the generate simulation world module. It consists of three files, the `gen_snake_model_mjcf.py`, `gen_terrain_model_mjcf.py`, and `generate_mjcf_main.py`. This module is responsible for the following:

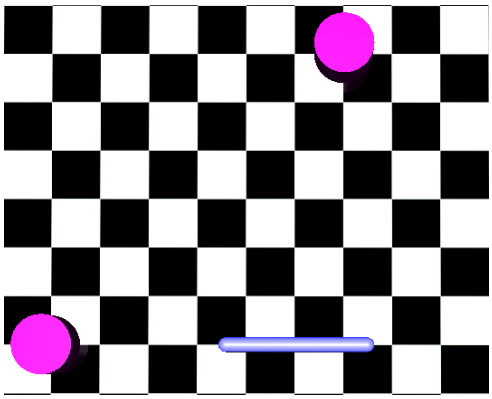
- Generating a snake XML model based on the config file
- Generating a terrain XML model based on the config file
- Combine them both together and provide a simulation world named `snake.xml`

A lot of this module is built on top of a GitHub repository named MJCF by iandanforth [34]. This repository is important as it provides Python bindings to automatically generate XML tags directly from Python.

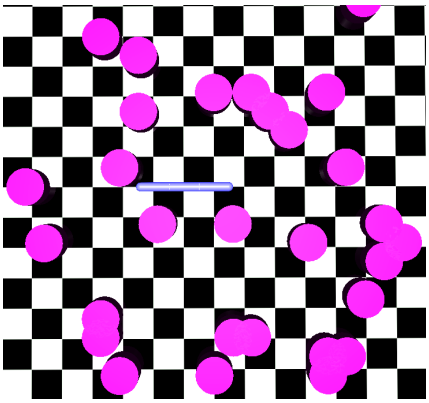
6.4.1 gen_snake_model_mjcf.py and gen_terrain_model_mjcf.py

The `gen_snake_model_mjcf.py` file is responsible for generating and combining all necessary pieces to create a snake XML file. It is divided into three major parts, the function that generates all the body parts, the function that generates all the actuators, and the function that generates all the sensors. These three body pieces are then returned to the main `generate_mjcf_file` to be combined into one large XML file.

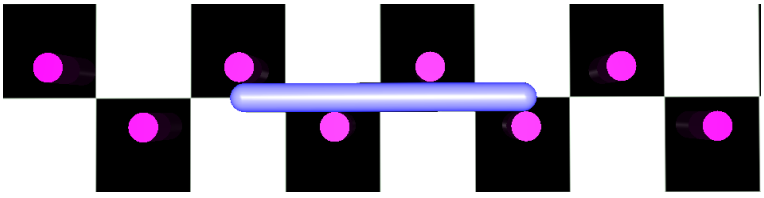
`gen_terrain_model_mjcf.py` have similar responsibilities as `gen_snake_model_mjcf.py`, but instead creates all necessary terrain objects, which it returns to the main `generate_mjcf_file`. It contains three prebuilt terrains that can be called from the configuration file, generate two cylinders that create two cylinders that can be used for collision testing, generate cylinder path that produces a path of cylinders for the snake to traverse, and generate random cylinders which spawns a bunch of cylinders around the snake in random configurations. These three prebuilt terrains can be seen in Figure 18:



(a) Prebuilt terrain one which can be enabled by setting generate two cylinders to true in the config file



(b) Prebuilt terrain two, which can be enabled by setting generate random cylinders to true in the config file



(c) Prebuilt terrain three, which can be enabled by setting generate cylinder path to true in the config file

Figure 18: Prebuilt terrain models

If one wants to build their own terrain models, there are prebuilt functions that allow this present in the `gen_terrain_model_mjcf.py` file. Call those functions at the bottom of `generate_mjcf_main.py` to create new terrain models.

6.4.2 gen_mjcf_main.py

The final piece of the generate simulation world module is the `gen_mjcf_main.py` python file. It is responsible for creating all pieces of the `snake.xml` file, and an example output from the Python file can be seen in Appendix 2. For a more in-depth explanation of how the file structure works, see Chapter 3.7, but in short, the file creates:

- A MuJoCo XML model with the < *mujoco* > tag

- A set of main classes such as < *option* >, < *asset* >, < *default* >, < *worldbody* >, < *actuator* > and < *sensor* >. These are all explained in Chapter 3.7.2.
- A set of children to all the main classes, such as different sensors for the < *sensor* > class, different actuators for the < *actuator* > class, and different body pieces for the < *worldbody* > class
- Lastly it adds terrain objects to the XML file which can be seen at the bottom of the python file.

To change anything in the `snake.xml` file, one should make changes in this module as the `snake.xml` file is regenerated every time one of the main simulation scripts is run.

6.5 Storing and plotting information module

The last module in the snake robot simulator is the storing and plotting module. This module consists of two files, `store_csv_data.py`, and `plot_csv_data.py`. The term CSV stands for (*comma separated values*) and is a widely used file format for storing information such as positional data for a robot. This module is responsible for the following:

- Storing different simulation data, such as during run-time to CSV format with apt names and folder with date and time.
- Plot interesting data with prebuilt functions based on existing CSV data.

The stored data can be customized from the configuration file by setting data the user does not want to be stored to false. One can also select to store no data as CSV, which can be helpful so the CSV's does not pile up under the CSV folder.

6.6 Trade offs during development

During the development of the snake robot simulator, there were many possible solutions for structuring both the software and the snake robot. This subchapter presents some of the trade-offs made during development and why these trade-offs were made.

6.6.1 Using MJCF instead of URDF

In general, when designing robots to use in a simulation environment, they are usually designed using either the URDF format or the MJCF format. URDF stands for universal robot descriptive format and can be read more about in Article [29] while MJCF stands

for MuJoCo Modeling XML File and is presented in Chapter 3.7. Both are viable formats as both uniformly describe a robot in the end, but there are some key differences. One of the main differences is that:

> "MJCF provides more elements and attributes than URDF format [23], however only a few of them need to be defined explicitly by the user while others can take default values. According to the developers, this makes MJCF files on average shorter and more readable than URDF files defining the same robots" (*Mikhail Ivanou et al.*[35]).

URDF also does not allow motor or controller descriptions in the XML file [35]. Ultimately, it was selected to go with the MJCF format mainly because of the MJCF's ability to provide more detailed and concise descriptions of a snake robot.

6.6.2 Using simulated sensors instead of taking data directly

There were two possible options when deciding how to store information from the snake robot during simulations. It was possible to either store data directly from the simulation state found in `Mj.data` or implement sensors to record data. Both options provide the same result, but implementing it through sensors would require more work. However, the benefit of using sensors is that it allows for a more realistic simulation since the real robot also will read its data from sensors found in the snake robot. These real-world sensors will naturally sometimes experience noise which can be simulated when using simulated sensors instead of fetching data from the `Mj.data` object. The sensors also convert data from the quaternion format MuJoCo uses for most values to cartesian form, which is easier to understand when looking at data. These two reasons made the project decide to go with sensors where it was possible to gather data to help with realism and readability.

6.7 How to use the software

A lot of information is found in Chapter 6 that explains how the entire software works. However, this subchapter aims to present a shorter how-to-use manual to help potential users quickly grasp the most essential aspects and start simulating.

6.7.1 How to run a simulation

The simplest way to run a simulation and store data is to run the `simulate_snake.py` file. This will read the configuration file `config.json`, call `generate_mjcf_main.py` to create a robotic snake and spawn it into a simulation world. The robotic snake will not do anything specific by running this file, but this is how to start a simulation.

6.7.2 How to control the snake robot

To control the snake robot during simulation, one has to do mainly two things. First, one has to go to the config file and set the control method to either torque, position, velocity, or intVelocity. It is only possible to select one type as this config is used to tell the simulator which control function it will use for its callbacks (meaning the function it calls once every timestep). Once a method is selected, go to the `ctrl_snake.py` file. Here there are a couple of functions that can be used:

- Use the init controller function to set initial values to actuators such as proportional gain. This function is called once during simulation init. This can and should be controlled from the config file.
- Use init snake pos to set the initial placement for the snake to move it around or rotate it before the simulation starts. This can and should be controlled from the config file.
- Use either torque control, position control, velocity control, or intVelocity control based on the method selected in the config file. In these functions, one can write control code to move the actuators. Access the actuators using the list `data.ctrl`.
- When accessing the controllers, one has to access the `data.ctrl` list at the correct indexes to control the correct actuators.
  `data.ctrl[0:(nr_of_links - 1)]` accesses the torque controllers,
  `data.ctrl[(nr_of_links-1):(nr_of_links-1)*2]` accesses the position controllers,
  `data.ctrl[(nr_of_links-1)*2:(nr_of_links-1)*3]` accesses the velocity controllers and
  `data.ctrl[(nr_of_links-1)*3:(nr_of_links-1)*4]` accesses the intVelocity controllers.

7 Experiment method

In order to assess the simulator, a wide range of experiments could be conducted. However, after thorough discussions, it was determined that four distinct experiments would be conducted to rigorously evaluate the simulator from various perspectives. Each subchapter explains how to set up and complete each experiment, with each subchapter containing one tabular, which are settings to be used in the configuration file.

7.1 Experiment I, Configuring the snake

The first experiment would be to run the main simulation file with different sets of configurations in `config.json`. The purpose of this test is to evaluate the simulator's capability in generating various snake robots, ensuring that everything functions as intended. The main result from this test should be that no matter the length and width of each linkage, number of linkages, and other configurations, everything within the snake robot should still be placed correctly every time, and it should not break the simulator during start-up. The three different configuration sets that will be attempted are:

Configuration set	A	B	C
Number of links	2	5	15
Mass of link	0.3 kg	0.3 kg	0.3 kg
Link length	0.8 m	2 m	1 m
Link radius	0.25 m	0.5 m	1 m
Head position [x, y, z]	[0, 0, 0]	[10, 5, 0]	[0, 0, 5]
Head rotation around [x, y, z]	[0, 0, 0]	[0, 0, 180]	[180, 0, 0]
Snake color RGBA	[0, 0, 0.8, 1] (blue)	[1, 0, 0, 1] (red)	[0, 1, 0, 1] (green)

Table 5: Configuration settings for snake robot in experiment one

7.2 Experiment II, Collision experiments

The second experiment would be to run a couple of collision tests between a snake consisting of only one linkage and a cylindrical hindrance. This experiment should be able to show that during a head-on collision, the forces recorded from the collision should coincide with the mass of the snake linkage times its acceleration at the point of impact (F = m*a). This will be tested on both the x and y axis as these are the most relevant measurements for a 2-D snake robot. The experiment will also be attempted with a snake consisting of multiple linkages to ensure newtons second law (F = m*a) still holds despite changes in mass. To apply forces to the snake robot so a collision may take place, the `data.qvel`

function will be used. `Data.qvel` references the `Mj.data` [2] structure in MuJoCo and qvel stands for quaternion velocity and is a list containing the quaternion velocities for each linkage in the snake robot. This means, for example, that `Data.qvel[0]` = 5 applies a speed of 5 m/s to the snake robot's head in the x direction and `data.qvel[1]` = 5 applies a speed of 5 m/s to the snake robots head in the y direction. The configurations used for the two snake configurations are shown in Table 6.

Configuration set	A	B
Number of links	1	10
Mass of link	0.3 kg	0.3 kg
Link length	0.10 m	0.10 m
Link radius	0.08 m	0.08 m
Head position [x, y, z]	[0, 0, 0]	[0, 0, 0]
Head rotation around [x, y, z]	[0, 0, 0]	[0, 0, 0]
Friction	0	0
Integrator type	RK4 (Runge Kutta 4)	RK4 (Runge Kutta 4)
Simulation timestep	0.01 s (100 FPS)	0.01 s (100 FPS)
Generate two cylinders	True (Tells the system to generate two cylinders placed 10 m away to use for collision experiment)	True (Tells the system to generate two cylinders placed 10 m away to use for collision experiment)

Table 6: Configuration settings for snake robot in experiment two

7.3 Experiment III, Lateral undulation

The third experiment would be to attempt a known method of propulsion called lateral undulation. Lateral undulation is when a snake does a sinusoidal movement that propagates horizontal waves through the body of the snake from head to tail. These horizontal waves will push against any irregularities in the terrain, causing propulsion as explained by *Liljebäck et al.*[36]. This concept can be seen in practice in Figure 19 and be read more about in theory Chapter 3.1. Due to time limitations as mentioned in Chapter 2.2.2, the project is not able to add irregularities to the simulation, and only the sinusoidal movement part of the experiment will be able to be completed. The simulation will instead test lateral undulation with and without friction present to test the simulator's ability to accurately model the snake robot's dynamics, as well as assess the simulator's ability to emulate friction. Isotropic friction (3.8.2) is used instead of anisotropic friction (3.8.2) since MuJoCo does not allow anisotropic friction between objects unless specifically designed to do so. Therefore, in theory, this will mean the snake should not be able to move forward and should stay in place when sinusoidal movements occur, as explained in Chapter 3.8. The snake robot will have the configuration shown in Table 7.

Number of links	10
Mass of link	0.3 kg
Link length	0.10 m
Link radius	0.08 m
Head position [x, y, z]	[0, 0, 0]
Head rotation around [x, y, z]	[0, 0, 0]
Integrator type	RK4 (Runge Kutta 4)
Control method	Position
Simulation timestep	0.01 s (100 FPS)
Do_sinus	True (Tells the position actuators to move as a sinusoidal)
Maximum torque	$\pm 3Nm$
Friction	0,5 (Makes the floor act like a concrete floor)
Joint damping	0.5
Joint armature	0.2
Position actuator KP	1
Velocity actuator KP	0.4

Table 7: Configuration settings for snake robot in experiment three

Figure 19: Figure showcasing lateral undulation. Courtesy of [37]

7.4 Experiment IV, Form closure

The fourth experiment would attempt the concept of form closure in the simulator. Shortly explained, form closure is a concept where the goal is to take an object and completely immobilize it. When the object has been entirely immobilized with respect to rigid body transformations, meaning it can no longer move or rotate in the plane, the object is said to be under form closure. This concept can be read more about in an article by Jostein Løwer [38] and a visual representation of this concept can be seen in his figure in Figure 21. If we attempt this concept on the simulated snake robot, it will look like Figure 20 as it can not possibly move in any direction if we assume all the joints are stiff. If we then move joint angles towards the left by shifting the values in a list, we should observe according to the theory that the snake robot will move in a predetermined pattern shown in Figure 20. This experiment tests the simulator's ability to control and move the snake robot in specific manners as well as using obstacle-aided collisions to generate movement.

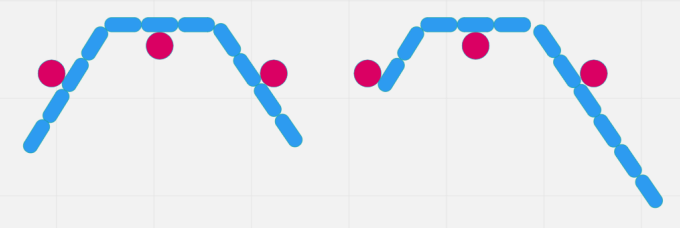


Figure 20: Figure showcasing form closure concept for a snake robot. The left configuration showcases the starting position, and the right configuration showcases the expected position of the snake after joint angles are shifted to the left

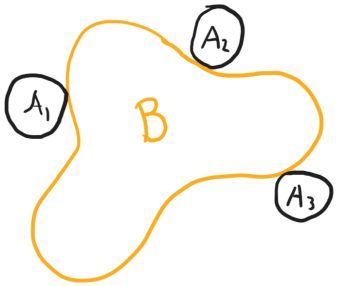


Figure 21: Figure showcasing form closure concept where object B has been completely immobilized in rotation and translation. Courtesy of [38]

Number of links	16
Mass of link	0.3 kg
Link length	0.10 m
Link radius	0.08 m
Head position [x, y, z]	[0, 0, 0]
Head rotation around [x, y, z]	[0, 0, 0]
Integrator type	RK4 (Runge Kutta 4)
Control method	Torque
Simulation timestep	0.01 s (100 FPS)
Form closure	True (Tells the simulator to place the snake in a specific position as well as create the terrain around it)
Maximum torque	$\pm 3Nm$
Friction	0.05
Joint damping	0.5
Joint armature	0.2
Torque actuator kp	1
PID [Kp, Ki, Kd]	[0.4, 0, 0]

Table 8: Configuration settings for snake robot in experiment four

8 Experiment results

8.1 Experiment I, Configuring the snake

The results of this experiment showcase three different snake configurations with varying amounts of links, lengths and sizes of each linkage, different starting placements, and different colors. There have also been placed cylinders in the simulation world for reference scale. These have the coordinates [-10, 0, 0] and [0, 5, 0] with a radius of 1 meter and height of 2 meters.

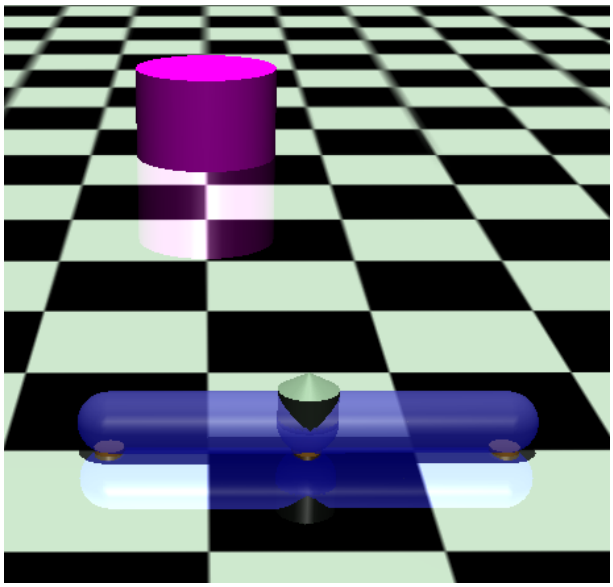


Figure 22: Configuration one showcasing a 2-linked snake. The object in the middle of the snake is an actuator that has been set to be visible to showcase the joint

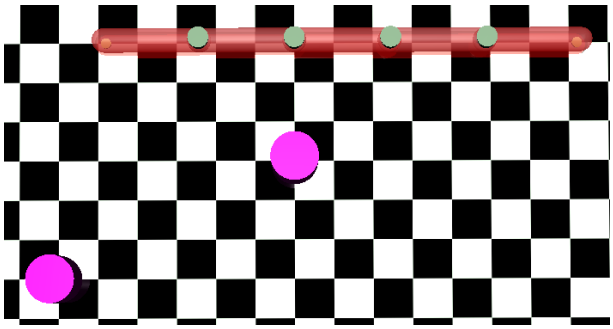


Figure 23: Configuration two showcasing a 5-linked snake. The gray objects along the snake are actuators that have been set to be visible to showcase the joints

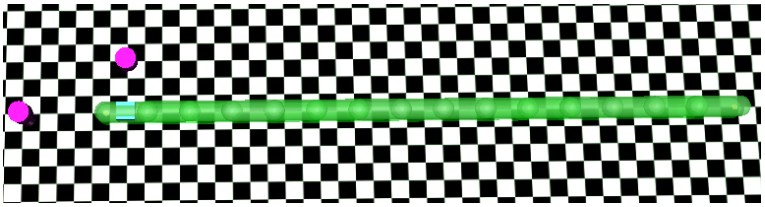
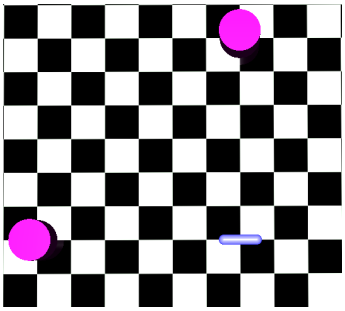


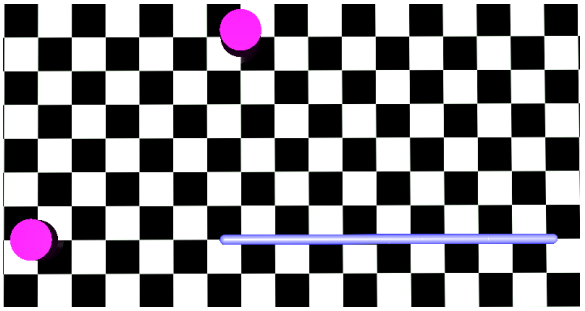
Figure 24: Configuration three showcasing a 15-linked snake. Actuators are not visible here as the snake is upside down due to head rotation

8.2 Experiment II, Collision experiments

This experiment's results showcase four collision tests completed to test the simulator's ability to handle contact forces. Two of the collisions are completed with one linkage on both the x and y-axis, while two of the collisions are done with ten linkages on both the x and y-axis. The simulation setup can be seen in Figure 25 and the results in Figure 26, 27, 28 and 29. It is important to highlight that when storing the information from these collision tests, a simple filter was implemented to filter out unreasonably large values. This was because the simulator, at some points, gave out unreasonably large collision forces of up to 100.000 newtons for a split second which ruined the plots. The contact forces with the ground have also been set to zero as these also became unreasonably high at some points, but since the tests measure x and y collisions, they are irrelevant.



(a) Simulation world for collision test one. Two objects to crash into where the snake has one linkage



(b) Simulation world for collision test two. Two objects to crash into where the snake has ten linkages

Figure 25: Simulation world used for both collision tests

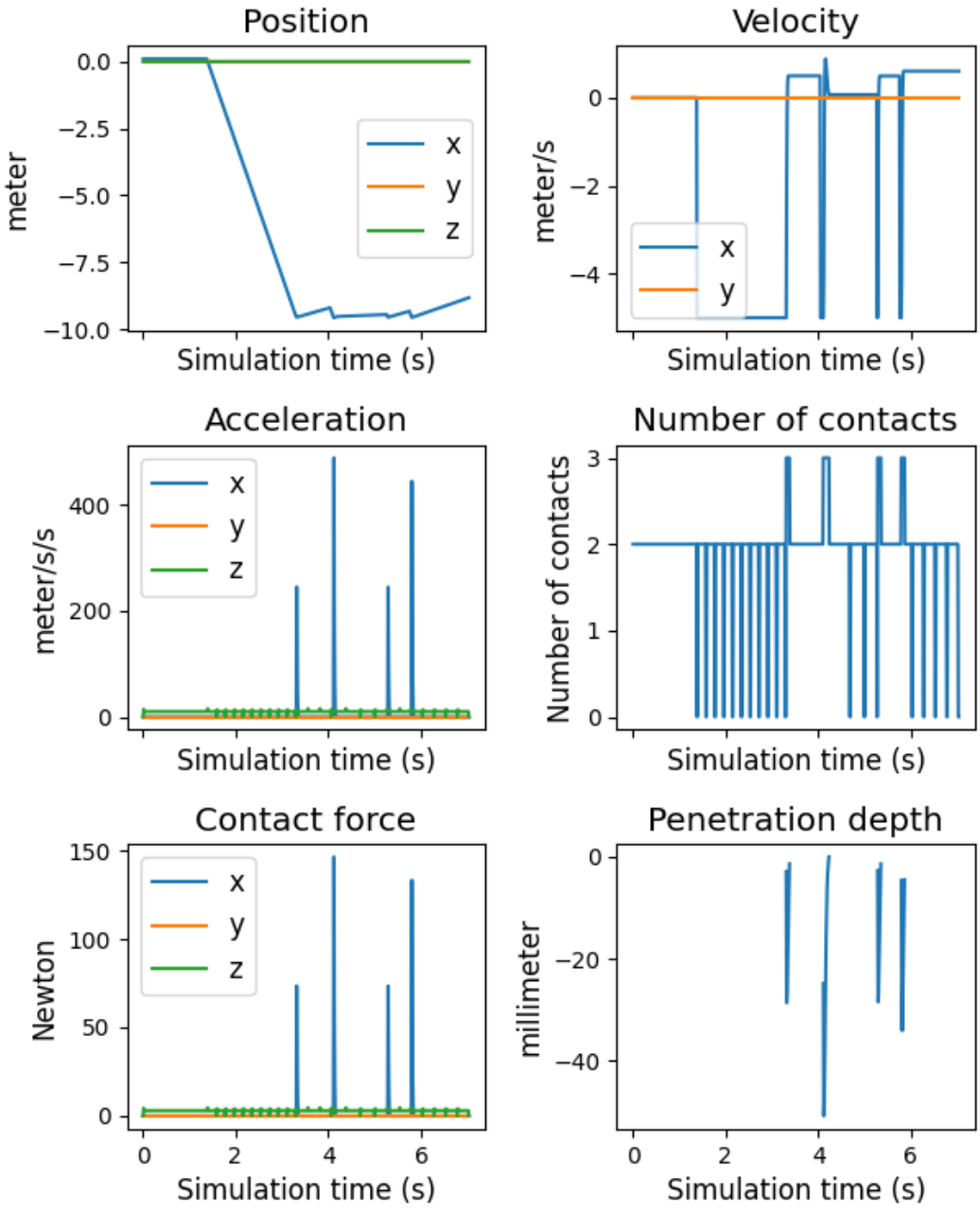


Figure 26: Collision test for one linkage with a collision along the x-axis. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment and how far the head penetrated the cylinder during the collision.

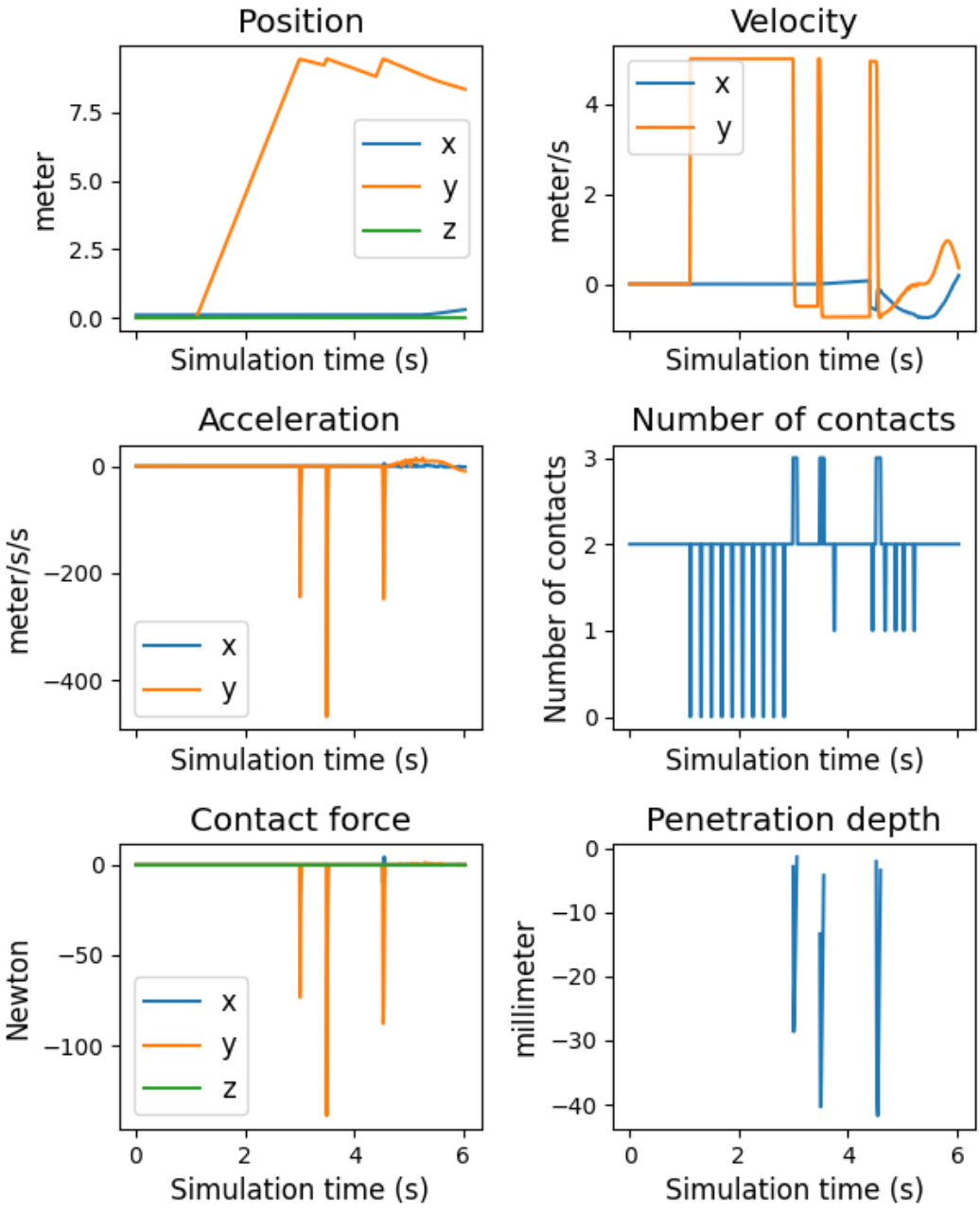


Figure 27: Collision test for one linkage with a collision along the y-axis. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment and how far the head penetrated the cylinder during the collision.

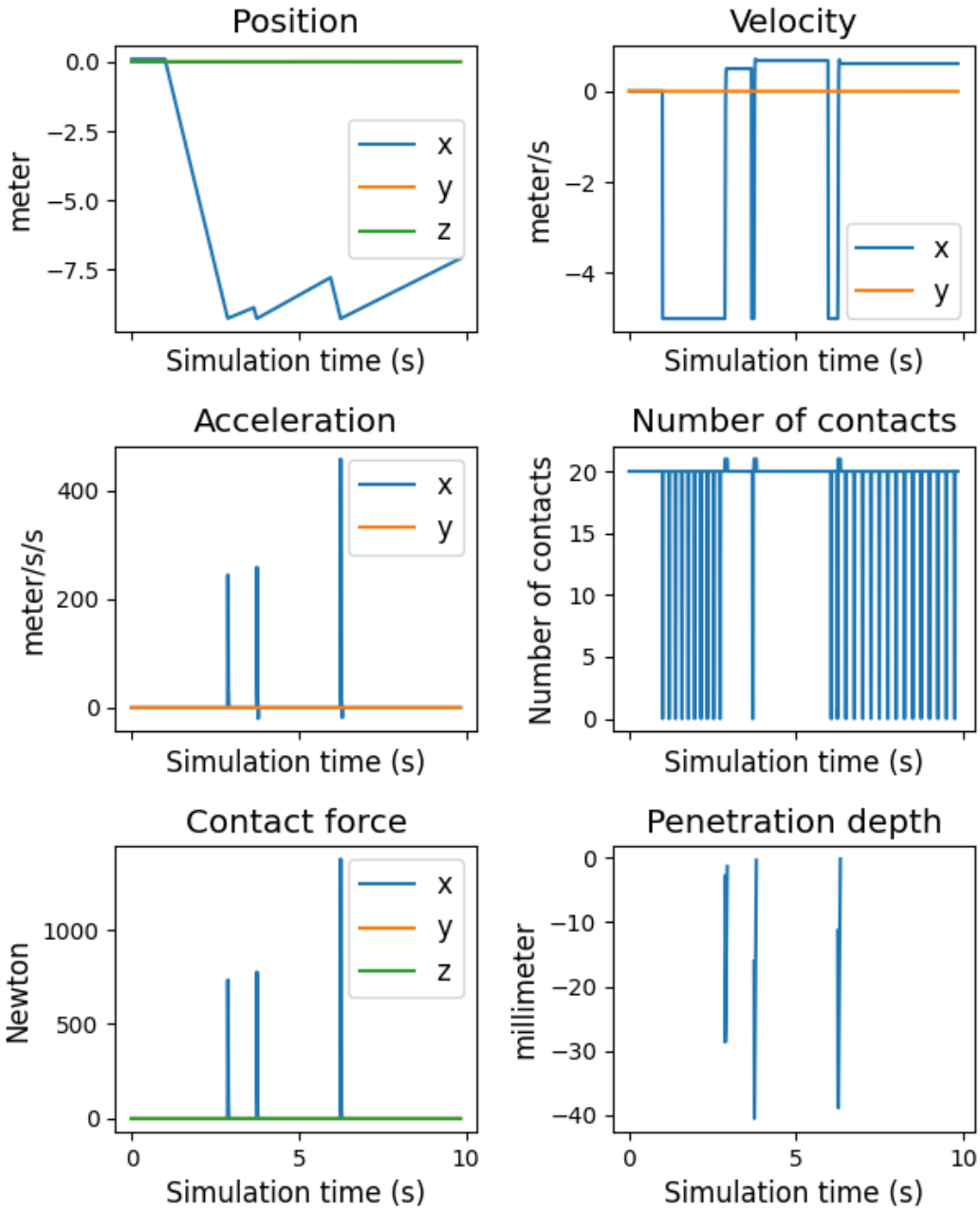


Figure 28: Collision test for ten linkages with a collision along the x-axis. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment and how far the head penetrated the cylinder during the collision.

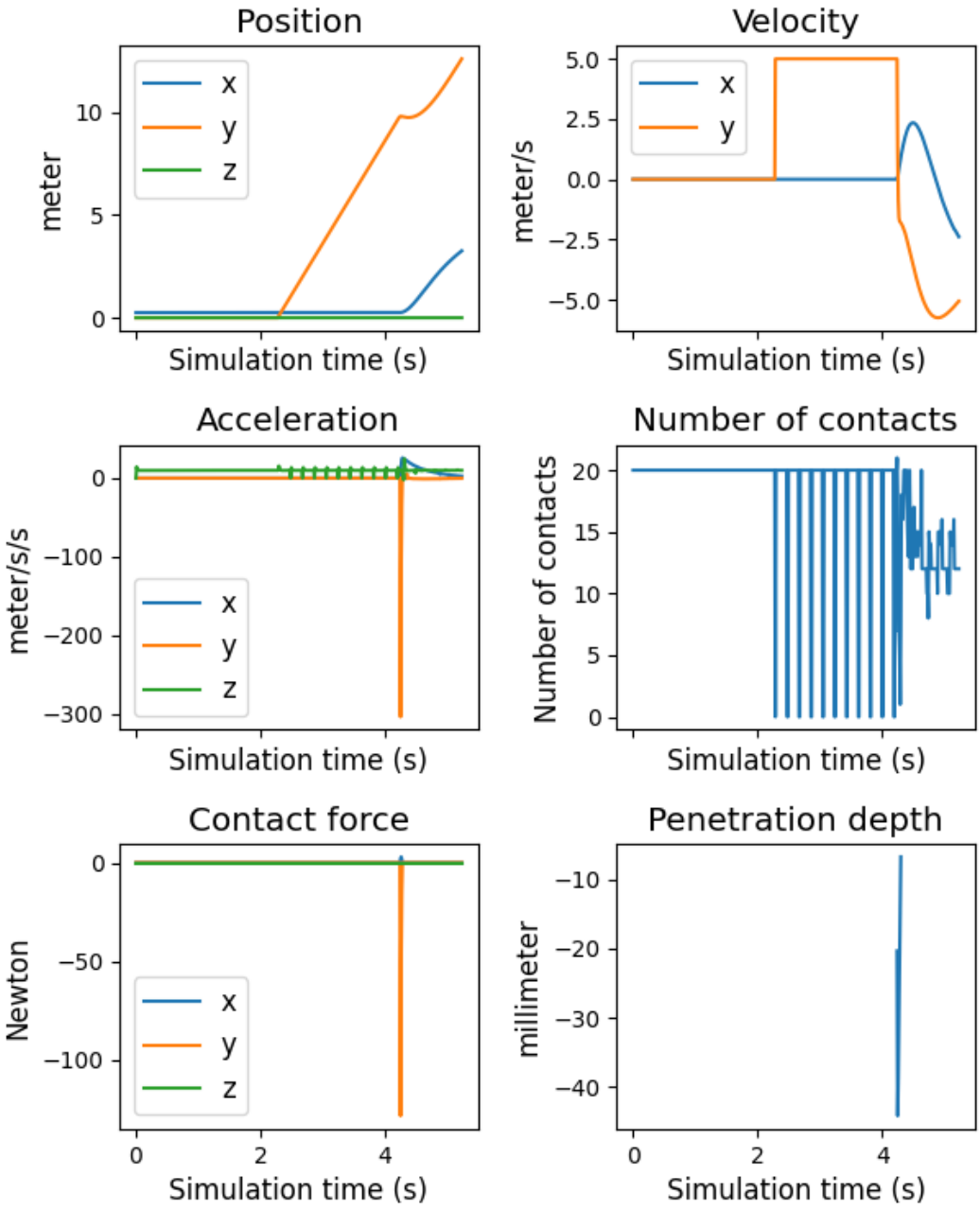
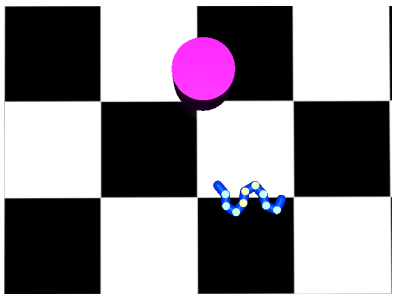


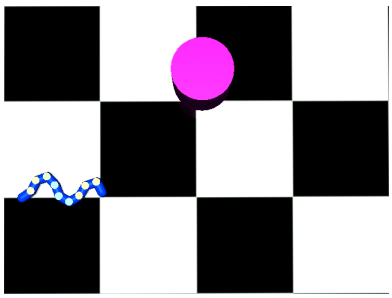
Figure 29: Collision test for ten linkages with a collision along the y-axis. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment and how far the head penetrated the cylinder during the collision.

8.3 Experiment III, Lateral undulation

The results of experiment three, shown in Figures 31, 32, and 33, showcase how a snake is able to complete specific controlled movements such as a sinusoidal when the actuators are used in a specific order. The experiment also shows that the snake robot creates propulsion even though only isotropic friction is introduced or when no friction is present. This is further discussed in Chapter 9.3. To achieve the sinusoidal movement, a sine wave from 0 - 180 degrees was given to the position actuators. The position actuators then move each joint to the correct angle. Finally, the sine wave is propagated through the robot by shifting the values backward through each position actuator.



(a) Snake placement during the beginning of lateral undulation experiment. The white dots on the snake are the actuators changing color based on how much torque is being used from 0 - $\pm 3Nm$



(b) Snake placement at the ending of lateral undulation experiment. The white dots on the snake are the actuators changing color based on how much torque is being used from 0 - $\pm 3Nm$

Figure 30: Snake position at beginning and end of lateral undulation experiment

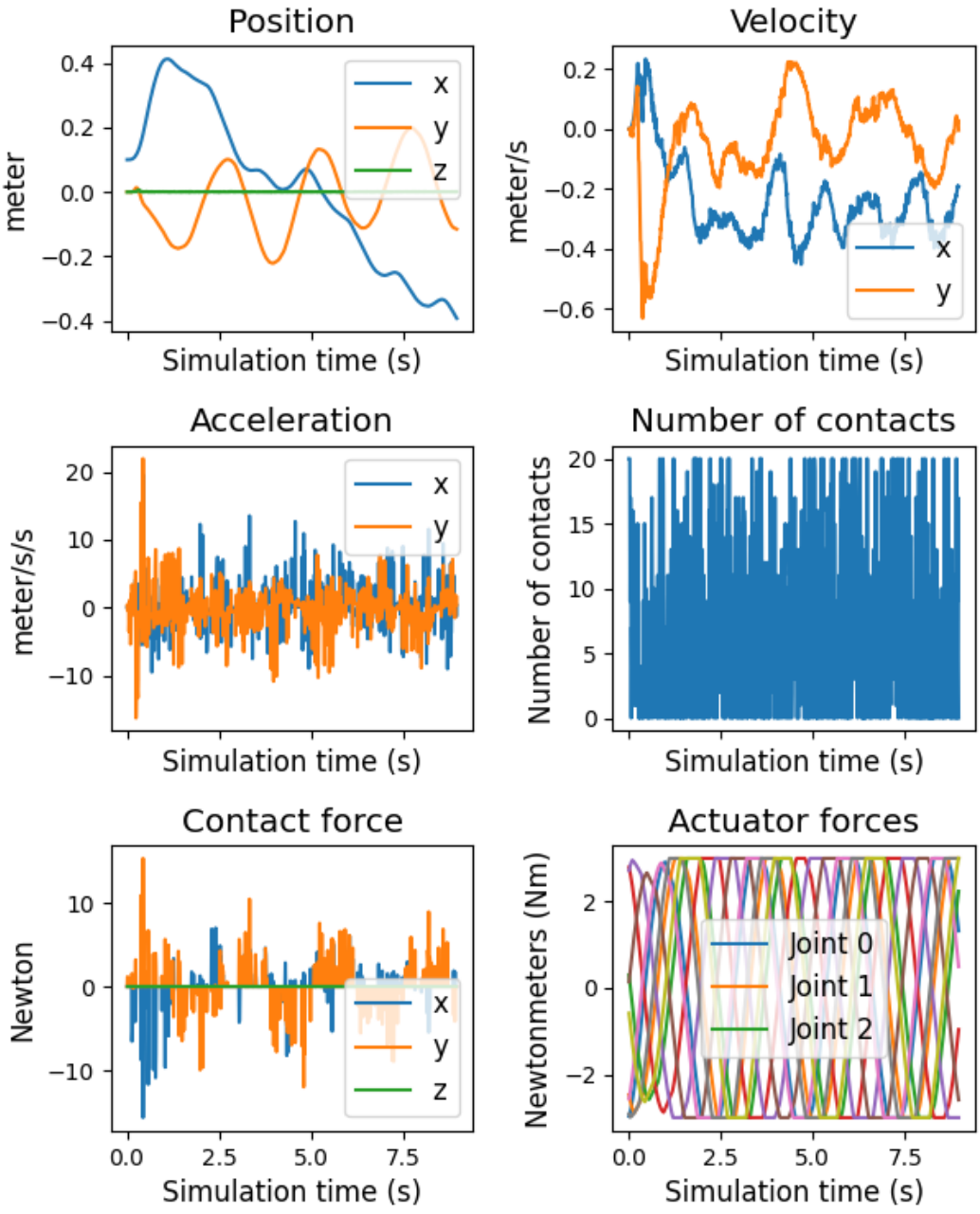


Figure 31: Lateral undulation experiment for snake robot over 10 seconds. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment as well as the actuator forces for each actuator.

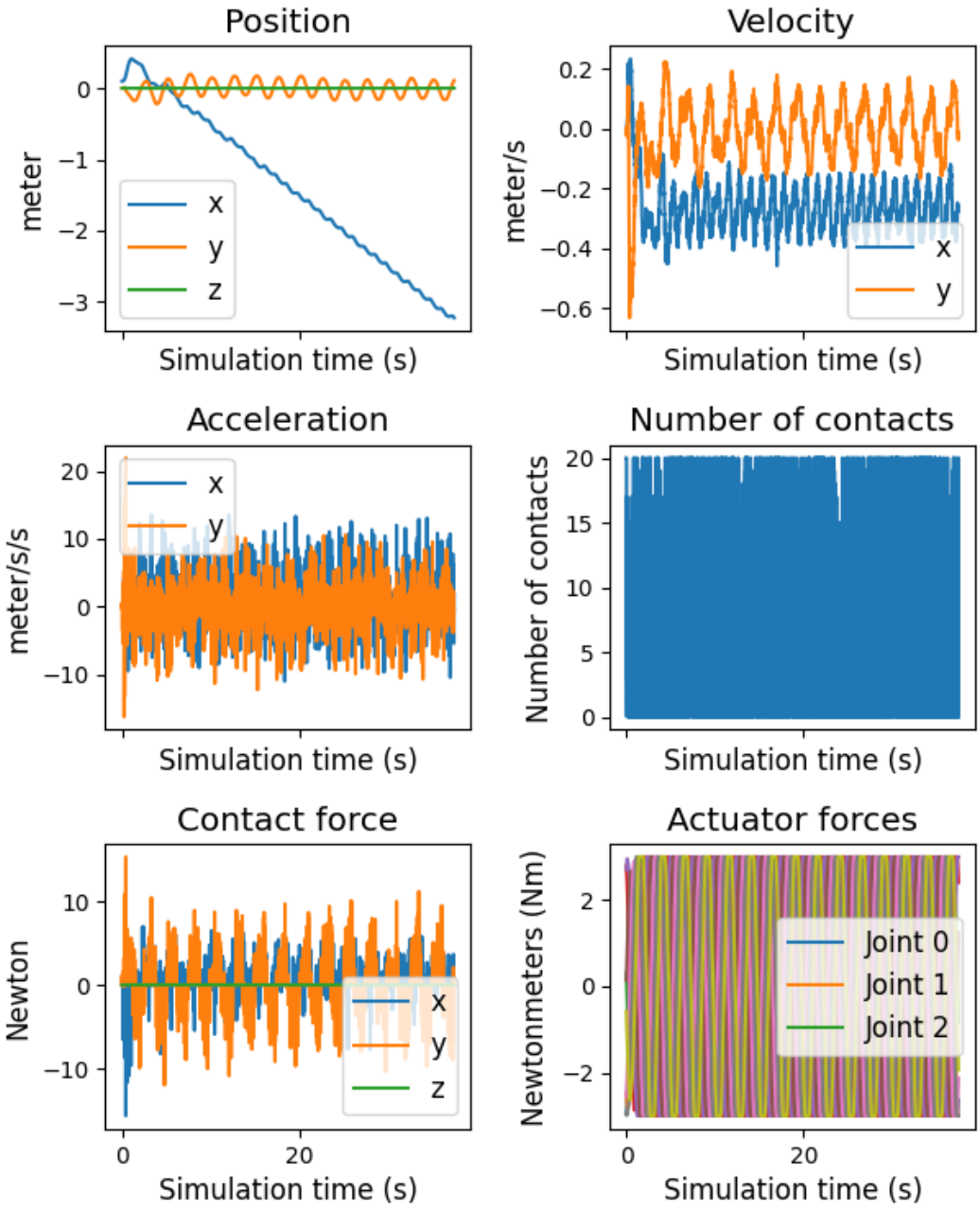


Figure 32: Lateral undulation experiment for snake robot over 40 seconds. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment as well as the actuator forces for each actuator.

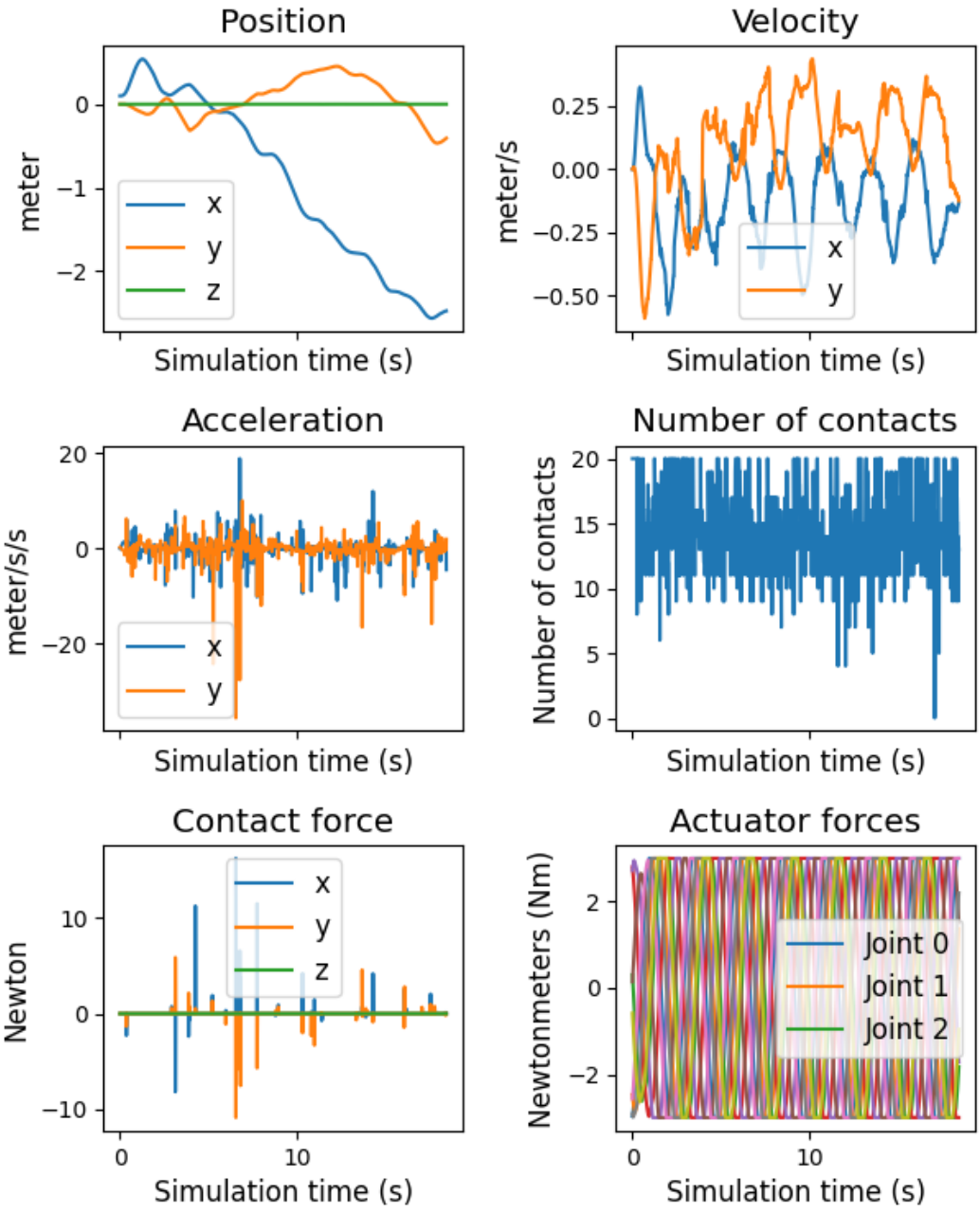
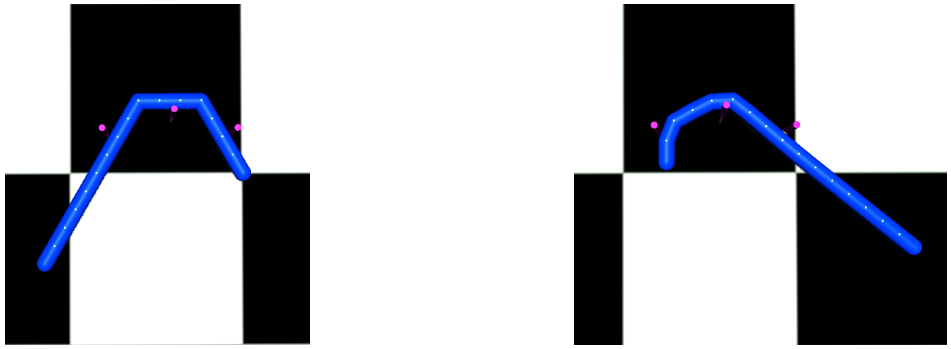


Figure 33: Lateral undulation experiment for snake robot over 20 seconds without friction. The first row of the plot shows the head's position and velocity during the experiment. The second row shows the head's acceleration and the total number of contacts present in the simulator at any given time. The third row shows the measured contact forces for the head during the experiment as well as the actuator forces for each actuator.

8.4 Experiment IV, Form closure

Experiment four showcases the snake robot simulator attempt at the concept of form closure. This concept can be read more about in Chapter 7.4. To control the snake, a simple PID controller was created for each torque actuator present in the snake robot. The PID controllers were then used for position control, and the output from the PID controllers was given directly to torque actuators, thus achieving the desired joint angles.



(a) Snake placement during the beginning of form closure experiment

(b) Snake placement at the ending of form closure experiment

Figure 34: Snake position at beginning and end of form closure experiment

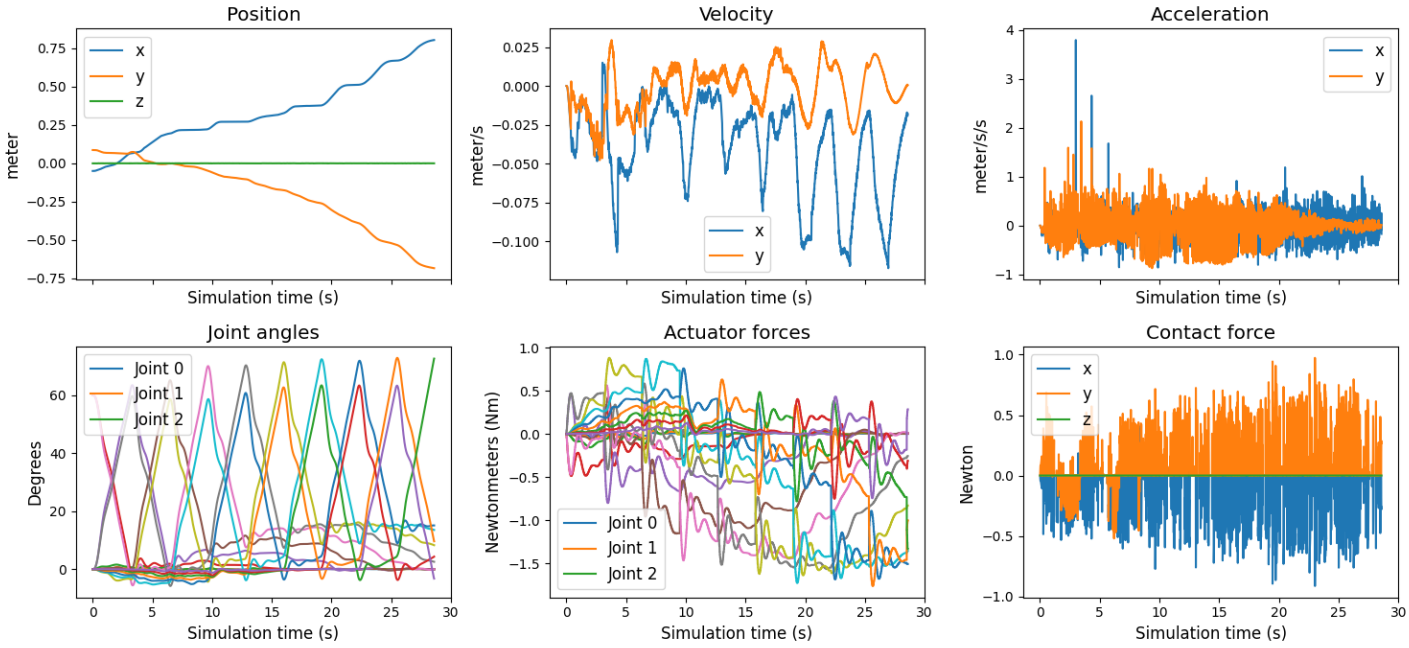


Figure 35: Form closure experiment for snake robot consisting of 16 linkages. The first row showcases the head's position, velocity, and acceleration throughout the simulation. The second row showcases the joint angles for each joint, the actuator forces for each actuator, and the contact force for the head during the simulation

9 Discussion and results

This section discusses the results presented in Chapter 8 and the implication of these. It is divided into four subchapters, and each subchapter discusses one of the results.

9.1 Experiment I, configuring the snake

The first experiment showcases the simulator's ability to change the snake robot's configuration from the configuration file. It showcases three specific configurations presented in Chapter 7.1, and all these configurations were selected as they are easily visible in figures and pictures. There is however possible to change a lot more than just the settings chosen for this experiment, and the whole configuration file with all options can be seen in Appendix 2. The results prove that no matter the configuration used for the snake robot, all actuators, joints, and body parts are placed correctly when auto-generated. This makes the simulator much more user-friendly, as one does not need to understand the MJCF file format to create a snake robot specific to one's use case. It also makes changing the snake robot much less time-consuming, as providing manual changes to the MJCF file is often a time-consuming endeavor.

9.2 Experiment II, Collisions tests

The second experiment showcases the simulator's ability to store and plot data from simulations as well as the viability of the physics engine MuJoCo. As can be seen from the figures in Chapter 8.2 there was completed a total of four collision tests with the simulated snake robot. Figure 26 and 27 showcase collision tests featuring only one linkage colliding into a cylinder on both the x and y-axis. As can be seen from the results, both the position and velocity of the linkage change as expected during a collision, with rapid changes present in both. Acceleration and contact forces also seem to be in accordance to uphold newtons second law of Force = Mass * Acceleration, as can be seen in the results in Chapter 8.2. Lastly, the number of contacts aligns well with when the collisions occur, and penetration depth is shown to be present but is only a few centimeters at most. Since the cylinder has a radius of 0.5 meters, this amount of penetration is not much, considering it is a head-on collision at relatively high speeds.

The same logic goes for the collision tests consisting of multiple linkages, where position, velocity, number of contacts, and penetration depth behave the same way. One can also see that acceleration and force measurements are higher for the collision on the x-axis when multiple linkages are involved. This is because there is now more mass involved in the collision. These four collision tests display that MuJoCo is able to provide realistic collisions with varying objects despite changes in mass. It is however important to note here that the plots made in the results do include a filter for high values, as mentioned

in the results Chapter 8.2. This was due to the simulator sometimes producing values up to 100.000 newtons or above, meaning there might still be some problems with the force measurements.

9.3 Experiment III, Lateral undulation

The third experiment showcases the simulator's ability to control the snake robot during runtime as well as testing friction within the simulation environment. The experiment works by making the snake robot complete a specific sinusoidal movement attempting the concept of lateral undulation that can be read more about in theory Chapter 3.1.

Benefits

As can be seen from the actuator forces in the result plots 31, and 32, as well as the snapshots from the simulation 30, the snake is able to complete the sinusoidal movement. This result is useful as it shows the possibility of moving the snake robot in specific patterns allowing users to also implement control algorithms in the same manner. Another interesting result is comparing the contact forces between the lateral undulation experiment with friction 32 and the lateral undulation experiment without friction 33. In the experiment with friction, one can clearly see from the contact forces plot that the head of the snake robot continuously meets friction forces. This is because there are no other points of contact for the head during the simulation, so the constantly shifting forces must be from friction. These friction forces are also noticeably absent in the contact forces plot from the lateral undulation experiment without friction, reinforcing this idea. This result is useful as it shows the simulator can easily turn on and off friction from the configuration file and that friction forces are applied correctly during simulation.

Issues

On the other hand, there are also two apparent flaws that need to be addressed. As discussed in the experimental method for this experiment 7.3, the snake robot should not be able to generate propulsion due to the friction being isotropic and not anisotropic. However, as can be seen from the position plots in the results of the lateral undulation experiments with friction 32, the snake clearly moves toward the left (x-becomes more negative over time). We reason that there are two main causes for this behavior.

The first theory lies in how MuJoCo defines contact between each snake robot linkage, which is a capsule, and the ground plane in a simulation. MuJoCo defines that each capsule has two points of contact with the ground when the snake linkage lies still. This means that in total, there should be twenty points of contact at any given time as long as the entire snake robot touches the ground plane when it consists of ten linkages. However, as can be seen in the number of contacts plot in Figure 31, the number of contacts shifts during sinusoidal movement. This means that the friction that should be isotropic can suddenly work in an anisotropic manner for short amounts of time, causing the observed propulsion.

The second theory is that the specific combination of capsule-to-plane collision with the specific contact dimension used for this experiment `condim = 3` does actually cause anisotropic friction. This is based on a forum discussion [39] from the developers of MuJoCo, where it is discussed how to implement anisotropic friction even though MuJoCo does not natively support it. Both reasons could be equally likely, but due to the time limitation of the assignment (see Chapter 2.2.2), the project does not have the capacity to research the issue further. A proposed solution is however presented in Chapter 11.4 as part of future work.

The other problem can be seen if one looks at the resulting plot from the lateral undulation experiment without friction 33. Here the snake is also able to move around even though no friction is present. This is after testing, the results of actuators seemingly causing collisions with forward joints causing contact forces, as seen from the result plots. This does however not seem to be a problem when friction is present, as the friction forces seem to be enough to counteract this movement. However, this problem can be solved by implementing *contact pairs*. This is a feature in MuJoCo that allows a developer to define all contacts that the simulator should register and give specific instructions to MuJoCo on how to handle each contact. In this way, removing all contact forces that do not come from collisions with the ground should be possible, making the snake stay in place.

9.4 Experiment IV, Form closure

The fourth experiment showcases the simulator's ability to perform complex experiments that simultaneously require many interactions and moving parts. As can be seen in the results from Chapter 8.4, the snake is able to create the expected movement when joints are shifted during form closure. This can be seen from the snapshots of the simulation as well as from the position plot showcasing how it steadily moves its head downwards to the right. One can also, from the plot, see how the joint angles shift over time as well as actuator forces, acceleration for the head of the snake, and velocity for the head of the snake. One thing of note that can be seen from the snapshots of the simulation 34 is that the snake robot has slipped a little towards the left, causing a slight rotation in its placement. This is a somewhat expected phenomenon called *lateral slippage*, but the snake robot is still under form closure even if this is the case. If one adds two more cylinders to change the form closure hold from second order to first order form closure, the issue should disappear according to Jostein Løwers article [38]. This was tested in a separate experiment which can be seen in Figure 36. Here it is possible to see that the snake robot receives less slippage towards the left, as expected in theory.

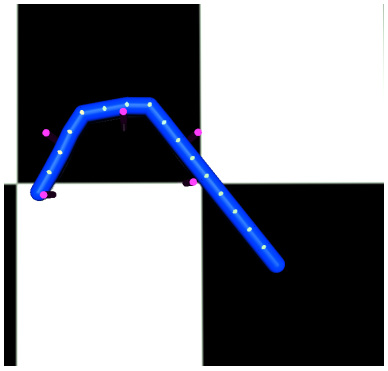


Figure 36: Snake robot performing the form closure experiment, with first-order form closure instead of second-order. The white dots are the actuators set to visible to showcase the position of the joints

10 Conclusion

After the software had been created and experiments were conducted, the simulator had many benefits and a few issues. Firstly the simulator is able to auto-generate a snake robot with many different configurations and simulate it for a desired amount of time. This makes the simulator versatile, which is a significant benefit when the current snake robot Boa still might see changes during construction and implementation. This also makes the simulator future-proof, as one will be able to simulate other kinds of snake robots when new designs emerge. The simulator is also able to accurately portray collision forces, position, velocity, and acceleration for the entirety of the snake robot, making it easy to conduct experiments and use the data for research purposes. Lastly, the simulator is easy to use due to everything being controlled by a configuration file making it user-friendly whether you are an expert programmer or not. On the other hand, there are also a few drawbacks. The simulator sometimes struggles with contact forces appearing where one would expect to find none. This can be seen in the results from Chapter 8.3 showcasing the lateral undulation experiment without friction. Here the system portrays contact forces on both the x and y axis when there should, in theory, be none. The simulator also lacks the implementation of anisotropic friction to complete lateral undulation properly. This however is something that can be added later, as seen in future work Chapter 11.4.

In conclusion, the simulator works well and accomplishes many of the goals set in the software specification Chapter 4, such as user-friendliness, physical realism, data acquisition and presentation, maintainability, affordability, and expandability. The project was also able to implement almost all the features presented in the priority list in Figure 12. Therefore, the simulator is a huge success in the eyes of the project and should be able to be used for future research in snake robotics.

11 Future work

This chapter details improvements to the simulator that both could and should be made if time would have allowed. Future software users can use this chapter to help understand what should be developed further to make the simulator both better and more optimal.

11.1 Better control algorithms

The first step to help improve the simulator would be to improve the control algorithms used for the actuators in the simulator. As per writing, the snake robot features four types of actuators in each joint, torque, position, velocity, and intvelocity. The first three, torque, position, and velocity, all take the same input argument as their name implies, while intvelocity has a built-in integrator for its velocity, which allows one to create a PI controller. Currently, the control strategies used in the simulator are simple PD control for the lateral undulation experiment and PID control for the form closure experiment. If a better control system is introduced, making the snake move as intended will be easier. It should, however, still be noted that both PID and PD worked quite well for the experiments completed in this thesis.

11.2 Force/torque measurement changes

Currently, both contact forces and actuator torques are measured directly from the `Mj.data` object present in the MuJoCo simulator. This works completely fine, but to make the simulator even more realistic, these measurements could be measured with simulated sensors instead. This would accomplish two things. Firstly it would allow the program to introduce sensor noise to the force and torque measurements in the snake, making it possible to test the snake with realistic noise. Secondly, it makes the simulated snake robot more portable as sensors are written directly into the MJCF code of the robot, meaning that if the robot is ported into another simulation, one could measure these sensors the same way as in MuJoCo.

11.3 Divide up snake.xml into three separate files

While creating the automatic snake generation Python script, the project did not know much about how the MJCF format worked. This meant that the Python script that creates the simulation environment is divided into three Python files, but the output, however, all ends up in the same MJCF file named `snake.xml`. It was discovered later that it is possible to use the < *include* > tag in MuJoCo to combine different MJCF files into one upon simulation, meaning that the terrain and snake robot could be two separate files entirely. This would both increase readability and allow either of the MJCF files to

be used for other projects independently of the other. This means, for example, that it would be possible to import terrain from other projects simply as the MJCF file into a folder, and the terrain would be simulated.

11.4 Implement contact pair for anisotropic friction

As mentioned in Chapter 9.3 discussing the results of the lateral undulation experiment, MuJoCo does not natively support anisotropic friction. However, a snake robot simulator would want to have this, as one of the main ways a serpent creates propulsion is through this anisotropic friction. There does, however, exist a method to implement anisotropic friction in MuJoCo as discussed by MuJoCo's creators, deepmind, in a discussion forum [39]. The discussion talks about using the `contact pair` XML tag to define custom contact interactions for specific contact pairs. This means it is possible to define a contact pair between each linkage in the snake robot and the ground, where custom friction components can be added. According to the discussion, it should be possible to define friction in this contact pair to mimic anisotropic friction and even comes with an example of how to implement it. To implement this into the autogeneration of the snake robot in the simulator, one should create the new tag $< contact >$ on the same level as $< default >$ and $< asset >$. One should then proceed to use a for-loop to generate one $< pair >$ between each linkage and the ground plane while specifying specific friction values. Lastly, this should be added to the `snake.xml` file to add it to the simulation.

References

[1] Pål Liljebäck. *Modelling, development, and control of snake robots*. Vol. 2011:70. Doktoravhandlinger ved NTNU (trykt utg.) Norwegian University of Science, Technology, Faculty of Information Technology, Mathematics, and Electrical Engineering, Department of Engineering Cybernetics, 2011. ISBN: 9788247126677.

[2] A. A. Transeth et al. "Snake Robot Obstacle-Aided Locomotion: Modeling, Simulations, and Experiments". In: *IEEE Transactions on Robotics* 24.1 (2008), pp. 88–104. DOI: 10.1109/TRO.2007.914849.

[3] Aksel Andreas Transeth and Kristin Ytterstad Pettersen. "Developments in Snake Robot Modeling and Locomotion". In: *2006 9th International Conference on Control, Automation, Robotics and Vision*. 2006, pp. 1–8. DOI: 10.1109/ICARCV.2006.345142.

[4] T. Yoshikawa and A. Sudou. "Dynamic hybrid position/force control of robot manipulators-on-line estimation of unknown constraint". In: *IEEE Transactions on Robotics and Automation* 9.2 (1993), pp. 220–226. DOI: 10.1109/70.238286.

[5] Jostein Løwer, Irja Gravdahl, Damiano Varagnolo, and Øyvind Stavdahl. "Proprioceptive contact force and contact point estimation in a stationary snake robot". In: *IFAC-PapersOnLine* 55.38 (2022). 13th IFAC Symposium on Robot Control SYROCO 2022, pp. 160–165. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2023.01.149. URL: https://www.sciencedirect.com/science/article/pii/S240589632300157X.

[6] Victor Melhuus Joel Mörlin and Oscar Mørk. *Intrinsic Force-Torque Sensor System for a Next Generation Snake Robot*. 2021. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2781042 (visited on October 12, 2022).

[7] Fredrik Veslum. "Assessment of the Mamba snake robot sensor system". In: (2020).

[8] Filippo Sanfilippo, Øyvind Stavdahl, and Pål Liljebäck. "SnakeSIM: a ROS-based control and simulation framework for perception-driven obstacle-aided locomotion of snake robots". In: *Artificial Life and Robotics* 23 (August 2018). DOI: 10.1007/s10015-018-0458-6.

[9] Atussa Koushan. *Simulator for Obstacle Aided Locomotion in Snake Robots*. 2019. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2780900/no.ntnu%3Ainspera%3A56990118%3A20964868.zip?sequence=2 (visited on October 12, 2022).

[10] Open Robotics. *ROS homepage*. URL: https://www.ros.org/ (visited on May 22, 2023).

[11] Open Robotics. *Gazebo homepage*. URL: https://gazebosim.org/home (visited on May 22, 2023).

[12] Khronos group. *OpenGL homepage*. URL: https://www.opengl.org/ (visited on May 23, 2023).

[13] MuJoCo. *MuJoCo Website*. URL: https://mujoco.org/ (visited on May 2, 2023).

[14] Erwin Coumans. *Bullet homepage*. URL: https://pybullet.org/wordpress/ (visited on May 26, 2023).

[15] The OGRE team. *OGRE homepage*. URL: https://www.ogre3d.org/ (visited on May 28, 2023).

[16] Nvidia. *Isaac-Sim documentation*. URL: https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/overview.html (visited on April 25, 2023).

[17] Coppelia Robotics AG. *CoppeliaSim homepage*. URL: https://www.coppeliarobotics.com/ (visited on May 22, 2023).

[18] Russ Smith. *ODE homepage*. URL: https://www.ode.org/ (visited on May 26, 2023).

[19] Cyberbotics Ltd. *Webots homepage*. URL: https://cyberbotics.com/ (visited on May 22, 2023).

[20] Tom Erez, Yuval Tassa, and Emanuel Todorov. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4397–4404. DOI: 10.1109/ICRA.2015.7139807.

[21] Nvidia. *PhysX homepage*. URL: https://developer.nvidia.com/physx-sdk (visited on May 26, 2023).

[22] Nvidia. *Nvidia homepage*. URL: https://www.nvidia.com/en-us/ (visited on May 30, 2023).

[23] MuJoCo. *MJCF explanation*. URL: https://mujoco.readthedocs.io/en/latest/XMLreference.html (visited on May 2, 2023).

[24] DeepMind. *MuJoCo Python API tutorial*. URL: https://colab.research.google.com/github/deepmind/mujoco/blob/main/python/tutorial.ipynb (visited on May 21, 2023).

[25] Simon Pabst, Bernhard Thomaszewski, and Wolfgang Straßer. "Anisotropic Friction for Deformable Surfaces and Solids". In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 149–154. ISBN: 9781605586106. DOI: 10.1145/1599470.1599490. URL: https://doi.org/10.1145/1599470.1599490.

[26] Github. *Github homepage*. URL: https://github.com (visited on May 21, 2023).

[27] Python. *Python PEP8 standard*. URL: https://peps.python.org/pep-0008/ (visited on May 21, 2023).

[28] ROS industrial. *URDF format by ROS*. URL: https://industrial-training-master.readthedocs.io/en/melodic/_source/session3/Intro-to-URDF.html (visited on May 21, 2023).

[29] Formant. *URDF explanation*. URL: https://formant.io/urdf/ (visited on May 10, 2023).

[30] Robotics Technologies LLC. *Robotics LLC homepage*. URL: https://roboticstechno.com/ (visited on May 28, 2023).

[31] DeepMind. *DeepMind homepage*. URL: https://www.deepmind.com/ (visited on May 31, 2023).

[32] MuJoCo. *MuJoCo key features*. URL: https://mujoco.readthedocs.io/en/latest/overview.html (visited on May 2, 2023).

[33] Oscar Brunell Mørk. *SimSerpent repository*. URL: https://github.com/Boa-Snake-Robot/SimSerpent (visited on May 22, 2023).

[34] iandanforth. *MJCF github repository*. URL: https://github.com/iandanforth/mjcf (visited on May 5, 2023).

[35] Mikhail Ivanou, Stanislav Mikhel, and Sergei Savin. "Robot description formats and approaches: Review". In: *2021 International Conference "Nonlinearity, Information and Robotics" (NIR)*. 2021, pp. 1–5. DOI: 10.1109/NIR52917.2021.9666120.

[36] Pål Liljebäck, Kristin Y. Pettersen, Øyvind Stavdahl, and Jan Tommy Gravdahl. "Lateral undulation of snake robots: a simplified model and fundamental properties". In: *Robotica* 31.7 (2013), pp. 1005–1036. DOI: 10.1017/S0263574713000295.

[37] Z. V. Guo and L. Mahadevan. "Limbless undulatory propulsion on land". In: *Proceedings of the National Academy of Sciences* 105.9 (2008), pp. 3179–3184. DOI: 10.1073/pnas.0705442105. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.0705442105. URL: https://www.pnas.org/doi/abs/10.1073/pnas.0705442105.

[38] Jostein Løwer. "A tutorial on form closure".

[39] Deepmind. *Github forum discussion on anisotropic friction in MuJoCo*. URL: https://github.com/deepmind/mujoco/issues/67 (visited on May 28, 2023).

12 Appendix

12.1 Configuration file

Listing 2: Configuration file

```
{
    "xml_path": "snake.xml",
    "simulation_time": 100,
    "simulation_timestep": 0.01,
    "integrator_type": "RK4",
    "plot_data": false,

    "visual_options": {
        "show_contact_force": false,
        "show_contact_points": false,
        "show_actuators": false,
        "show_joints": false,
        "show_center_of_mass": false,
        "make_transparent": false
    },

    "snake_specifications": {
            "nr_of_links": 2,
            "link_mass_in_kg": 0.3,
            "link_lenght_in_m":0.10,
            "link_radius_in_m":0.08,
            "maximum_torque_in_nm": [-3, 3],
            "friction": 0.5,
            "joint_damping": 0.5,
            "joint_armature": 0.2,
            "sensor_noise": "disable",
            "sensor_noise_amount": 0,
            "snake_color_rgb": [0, 0.2, 1, 1],
            "head_position_xyz": [0, 0, 0],
            "head_rotation_xyz": [0, 0, 0]
    },

    "control_parameters":{
        "control_method": "position",
        "torque_actuator_KP": 1,
        "position_actuator_KP": 0.7,
        "velocity_actuator_KP": 0,
        "IntVelocity_actuator_KP": 50,
        "PID_parameters_Kp_Ki_Kd": [0.4, 0, 0]
    },

```

```
42      "terrain":{
43          "generate_two_cylinders": false,
44          "generate_random_cylinders": false,
45          "generate_cylinder_path": false
46      },
47
48      "prebuilt_experiments":{
49          "form_closure": false,
50          "do_sinus" : false
51      },
52
53      "store_data_to_csv": {
54          "store_data": false,
55          "position": false,
56          "velocity": false,
57          "accel": false,
58          "angle": false,
59          "angular_vel": false,
60          "angular_accel": false,
61          "torque_actuator_forces": false,
62          "position_actuator_forces": false,
63          "velocity_actuator_forces": false,
64          "joint_angles": false,
65          "external_contact_forces": false
66      }
67  }
```

12.2 MJCF file for snake robot with two linkages

Listing 3: snake.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <mujoco model="snake">
3      <option gravity="0 0 -9.81" integrator="RK4" timestep="0.01">
4          <flag sensornoise="disable"/>
5      </option>
6      <asset>
7          <texture builtin="gradient" height="100" rgb1="1 1 1" rgb2="0 0 0"
               type="skybox" width="100"/>
8          <texture builtin="flat" height="1278" mark="cross" markrgb="1 1 1"
               name="texgeom" random="0.01" rgb1="0.8 0.6 0.4" rgb2="0.8 0.6
               0.4" type="cube" width="127"/>
9          <texture builtin="checker" height="100" name="texplane" rgb1="0 0
               0" rgb2="0.8 0.8 0.8" type="2d" width="100"/>
10         <material name="MatPlane" reflectance="0.5" shininess="1" specular
               ="1" texrepeat="60 60" texture="texplane"/>
11         <material name="geom" texture="texgeom" texuniform="true"/>
12     </asset>
13     <default>
```

```
14        <joint armature="0.2" axis="0 0 1" damping="0.5" frictionloss="0"
              limited="true" range="-90 90" type="hinge"/>
15        <geom friction="0.5" mass="0.3" pos="0.1 0 0" rgba="1 0 1 1" size=
              "0.08 0.1" type="capsule" zaxis="1 0 0"/>
16    </default>
17    <size nuser_actuator="1"/>
18    <worldbody>
19        <light cutoff="100" diffuse="1 1 1" dir="0 0 -1.3" directional="
              true" exponent="1" pos="0 0 1.3" specular="0.1 0.1 0.1"/>
20        <geom conaffinity="1" condim="3" material="MatPlane" name="floor"
              pos="0 0 -0.08" rgba="0.8 0.9 0.8 1" size="100 100 100" type="
              plane" zaxis="0 0 1"/>
21        <body name="head" pos="0.2 0 0">
22            <geom name="head_geom" rgba="0 0.2 1 1"/>
23            <freejoint/>
24            <site name="head_site" pos="0.2 0 0"/>
25            <body name="body1" pos="0.2 0 0">
26                <geom name="body_1_geom" rgba="0 0.2 1 1"/>
27                <joint name="body_1_joint"/>
28                <site name="body_1_site" pos="0.2 0 0"/>
29            </body>
30        </body>
31    </worldbody>
32    <actuator>
33        <motor ctrllimited="true" ctrlrange="-3 3" forcelimited="true"
              forcerange="-3 3" joint="body_1_joint"/>
34        <position kp="0" ctrllimited="true" ctrlrange="-3 3" forcelimited=
              "true" forcerange="-3 3" joint="body_1_joint" name="
              position_servo1"/>
35        <velocity kv="0" ctrllimited="true" ctrlrange="-3 3" forcelimited=
              "true" forcerange="-3 3" joint="body_1_joint" name="
              velocity_servo1"/>
36        <intvelocity kp="0" actrange="-1.57 1.57" ctrllimited="true"
              ctrlrange="-3 3" forcelimited="true" forcerange="-3 3" joint="
              body_1_joint" name="IntVelocity_servo1"/>
37    </actuator>
38    <sensor>
39        <velocimeter site="head_site" name="velocity_sensor0" noise="0"/>
40        <accelerometer site="head_site" name="accel_sensor0" noise="0"/>
41        <gyro site="head_site" name="gyro_sensor0" noise="0"/>
42        <velocimeter site="body_1_site" name="velocity_sensor1" noise="0"/
              >
43        <accelerometer site="body_1_site" name="accel_sensor1" noise="0"/>
44        <gyro site="body_1_site" name="gyro_sensor1" noise="0"/>
45    </sensor>
46 </mujoco>
```

12.3 Specialization project report fall 2022

Abstract

Boa, a next generation sensor-driven snake robot, is currently under development at the Department of Engineering Cybernetics (ITK) at the Norwegian University of Science and Technology (NTNU). As a result, an assignment has been issued to research and develop a contemporary snake robot simulator which can aid in rapid development of state estimation and control strategies for Boa. This project looks into the research part of the assignment and aims to both present different simulator frameworks and asses their viability for snake robot research. Firstly the paper presents relevant background information about snake robot research at ITK as well as theory on what a simulator is and what a simulator consists of. The paper then proceeds to showcase seven different relevant simulators for snake robot research and presents information in a structured tabular for comparison. Lastly the paper showcases experiments completed on three of the simulators, aswell as a discussion and a conclusion on what simulator should be used for further development and research.

Keywords: Robotics technology, Snake Robots, Robotic simulator, Physics Engines

Sammendrag

Boa, en neste generasjons sensordrevet slangerobot, er nåværende under konstruksjon ved Instituttet for Teknisk Kybernetikk (ITK) ved Norges teknisk-naturvitenskapelige universitet (NTNU). Grunnet dette, har det blitt utgitt en oppgave om å undersøke og utvikle en slangerobot simulator som kan bidra til raskere utvikling av tilstandsestimering og kontroll algoritmer for Boa. Denne rapporten fokuserer på undersøkelsesdelen av oppgaven og har som mål å både presentere forskjellige simulator rammeverk og vurdere deres brukbarhet for slangerobot forskning. Først presenterer rapporten relevant bakgrunnsinformasjon om slangerobot forskning ved ITK sammen med teori om hva en simulator er og om hva den består av. Rapporten fortsetter så med å vise frem syv forskjellige relevante simulator for slangerobot forskning og presenterer informasjon i en strukturert tabell for enkel sammenlikning. Til slutt viser rapporten eksperimenter gjennomført på tre av simulatorene, sammen med en diskusjon og en konklusjon rundt hvilken simulator som einer seg best for videre utvikling og forskning.

Nøkkelord: Robotikk teknologi, Slange robotikk, Robotikk simulator og Fysikkmotor

Abstract

Boa, a next generation sensor-driven snake robot, is currently under development at the Department of Engineering Cybernetics (ITK) at the Norwegian University of Science and Technology (NTNU). As a result, an assignment has been issued to research and develop a contemporary snake robot simulator which can aid in rapid development of state estimation and control strategies for Boa. This project looks into the research part of the assignment and aims to both present different simulator frameworks and asses their viability for snake robot research. Firstly the paper presents relevant background information about snake robot research at ITK as well as theory on what a simulator is and what a simulator consists of. The paper then proceeds to showcase seven different relevant simulators for snake robot research and presents information in a structured tabular for comparison. Lastly the paper showcases experiments completed on three of the simulators, aswell as a discussion and a conclusion on what simulator should be used for further development and research.

Keywords: Robotics technology, Snake Robots, Robotic simulator, Physics Engines

Sammendrag

Boa, en neste generasjons sensordrevet slangerobot, er nåværende under konstruksjon ved Instituttet for Teknisk Kybernetikk (ITK) ved Norges teknisk-naturvitenskapelige universitet (NTNU). Grunnet dette, har det blitt utgitt en oppgave om å undersøke og utvikle en slangerobot simulator som kan bidra til raskere utvikling av tilstandsestimering og kontroll algoritmer for Boa. Denne rapporten fokuserer på undersøkelsesdelen av oppgaven og har som mål å både presentere forskjellige simulator rammeverk og vurdere deres brukbarhet for slangerobot forskning. Først presenterer rapporten relevant bakgrunnsinformasjon om slangerobot forskning ved ITK sammen med teori om hva en simulator er og om hva den består av. Rapporten fortsetter så med å vise frem syv forskjellige relevante simulator for slangerobot forskning og presenterer informasjon i en strukturert tabell for enkel sammenlikning. Til slutt viser rapporten eksperimenter gjennomført på tre av simulatorene, sammen med en diskusjon og en konklusjon rundt hvilken simulator som einer seg best for videre utvikling og forskning.

Nøkkelord: Robotikk teknologi, Slange robotikk, Robotikk simulator og Fysikkmotor

Preface

This project report is written in the fall of 2022 for the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The report is part of a preliminary project before a written master thesis in the spring of 2023 and serves as a starting point for said master thesis. This report covers 7.5 study points due to being part of a two year curriculum instead of the usual 15 study points.

I would like to thank Jostein Løwer for continued guidance throughout this report and ITK for assistance with both computers and office space.

- Oscar Mørk December 19, 2022

0.1 Perception of thesis assignment

The thesis assignment shown in 0.2 clearly states the goals of the project in a structured list. However the first point on *Give a brief overview of available physics engines and visualizers* have been perceived as give a brief overview of available physics engines and simulator frameworks that can be relevant for robotic research. This was because the amount of available simulators and physics engines number in the 30-40s and many would not be relevant at all for snake robot research. Including this, point number five has also been perceived as optional if time allows. Everything else was understood as stated in the thesis assignment.

0.2 The thesis assignment

NTNU
Norwegian University of
Science and Technology

Faculty of Information technology
and Electrical Engineering
Department of Engineering Cybernetics

Project Assignment

Student's name: Oscar Mørk

Field: Engineering Cybernetics

Title (Norwegian): Fysikkbasert simulator for neste generasjons slangerobot

Title (English): Physics-based simulator for a next generation snake robot

Description:
Boa, a next generation sensor-driven snake robot, is currently under development at ITK. This assignment seeks to develop a physics-based snake robot simulator, enabling rapid development of state estimation and control strategies for the snake robot. Snake robot simulations pose strict requirements on the performance of the physics engine both in terms of accuracy and execution time. In this assignment you will research different physics engines and visualizers that might be suitable for developing a snake robot simulator.

1. Give a brief overview of available physics engines and visualizers ("platforms").
2. Make a short-list of platforms that are suitable for snake robot research.
3. Perform selected simple simulations to evaluate the performance of the physics engines and visualizers for the given application.
4. Give a recommendation for a physics engine and/or visualizer that should be used for further development of a snake robot simulator.
5. If time permits: Showcase simple simulations of a snake robot with the chosen platform.

Co-supervisor(s): Jostein Løwer, NTNU

Trondheim, 24.08.2022

Øyvind Stavdahl
Supervisor

Contents

1 Nomenclature/Glossary

ITK	Department of Engineering Cybernetics
NTNU	Norwegian University of Science and Technology
HOAL	Hybrid Obstacle Aided Locomotion
OAL	Obstacle Aided Locomotion
POAL	Perception Obstacle Aided Locomotion
API	Application Programming Interface
GUI	Graphical User Interface
Simulator	Collective term for physics and rendering engines working together
Physics engine	Software designed for simulating the physical behavior of objects
Rendering engine	Software designed for generating visual representation of physics

2 Introduction

After Pål Lilljebekks doctorate paper [1] on a new force/torque measurement technique surrounding *HOAL*, NTNU began research on snake robotics implementing this premise. This combined with other possible research benefits lead to the creation of two snake robots named *Mamba* and *Boa*. One of the goals for these snake robots was through experimentation to develop new control strategies and state estimation algorithms aswell as proving HOAL as method for locomotion. However performing physical experiments on real-life snake robots is a time consuming and costly task. From a research perspective, it is desirable to test software, control strategies and state estimation algorithms on a simulated robot, as simulations are cost efficient and allows for rapid turn-around time when testing new strategies. As a result, a conscious effort has been made to develop a new snake robot simulator for research into snake robots and OAL.

This assignment is the preliminary work for creating such a simulator where the end goal is researching, gathering, presenting and selecting an existing simulator framework to design a snake robot simulator with.

3 Background and theory

This chapter covers relevant theory and background for the report. It explains theory behind Obstacle Aided Locomotion (OAL), the current state of snake robots at ITK aswell as what a simulator is and why it is beneficial for further development of the snake robot.

3.1 Obstacle Aided Locomotion (OAL)

In 2008, *Transeth et al.* published a paper named *Snake Robot Obstacle-Aided Locomotion: Modeling, Simulations, and Experiments* [2]. This paper discusses how serpents mainly use two ways to advance through terrain, called *lateral undulation* and *obstacle aided locomotion*. The first term, *lateral undulation*, is used to explain how snakes exploit the asymmetrical scales and schutes found on their belly to create friction in a single direction, thus leading to propulsion. This method of locomotion is the most researched and implemented in snake robotics [3] , but makes the robot overly dependent on the surface it moves upon. The paper therefore rather looks at the other method of propulsion, *obstacle aided locomotion* called *OAL* for short. This method employs the concept of *obstacle exploitation* which implies that instead of avoiding obstacles, the snake uses them to push against and create momentum. In this way the snake and snake robots become more adaptable to the environment they are in and allows it to move where its entire body cannot make contact with the ground. OAL has also evolved into a more specialized version called *Hybrid Obstacle Aided Locomotion* or *HOAL* for short. HOAL is the combination of obstacle aided locomotion and a technique called *HPFC, Hybrid Position/Force Control*. HPFC as explained by *T. Yoshikawa et al.* [4] is a control strategy that combines position control and force control to achieve a specific task. It is relevant to know of HOAL for later explanations, but OAL will remain the focus of this report.

The theory behind OAL can be explained by looking at an example snake robot shown in figure 1. A snake robot consists of multiple linkages which are represented by the red, blue and green cylinders. In each of these linkages, there exists a servo motor to generate torque aswell as a force/torque measurement system to measure said torque. In an ideal world, where it is assumed that the friction acting on a linkage is either known or equal to zero, the only forces acting on a linkage are the forces/torques of the previous and next linkage aswell as that of external objects the linkage is in contact with. Friction is denoted F_R, force measurement from contact with previous linkage as h_{n-1}, force measurement from contact with the next linkage as h_n, external forces as f_{ext} and the summation of all external force vectors as f_{tot}. Based on Newtons second law shown in (2), the sum of forces equals mass times acceleration. The equation for external forces acting on a linkage can then be shown mathematically as (3). By adding up all the external force vectors as shown in (4), it is possible to calculate a total force vector which gives the direction of motion the snake will achieve by pushing against the external objects. This is an important premise for *OAL*.

$$F_R \approx 0 \tag{1}$$

$$\Sigma F = ma \tag{2}$$

$$f_{ext} = ma - h_n - h_{n-1} \tag{3}$$

$$f_{tot} = f_{ext_1} + f_{ext_2} + f_{ext_3} \ldots + f_{ext_n} \tag{4}$$

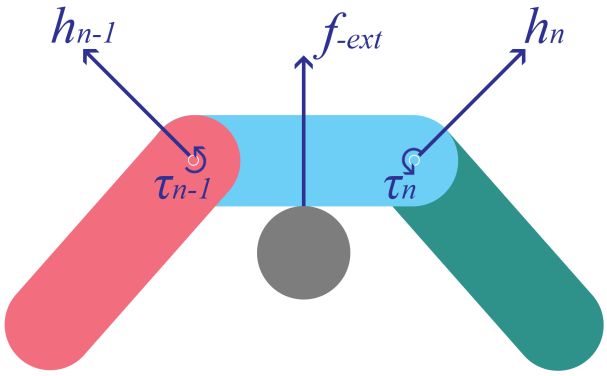


Figure 1: Forces and torques applied on a 2-jointed 2D-snake pushing up against an object. Courtesy of [5].

3.2 Contemporary snake robots developed at ITK

In 2011 a researcher named *Pål Lilljebekk* published his doctorate thesis titled *Modelling, development, and control of snake robots* [1]. The thesis presents experiments with two different snake robots named Wheeko and Kulko, where Kulko was used to experiment on the premise of OAL. Kulko ended being able to measure contact forces with external objects and was as stated by Pål lilljebekk fit for further OAL research. He did however in the last chapter of Kulko propose another method for environment sensing based on force measurements which could simplify the snake robot design. This lead to The Department of Engineering Cybernetics called ITK at NTNU starting work on creating a new functional snake robot based on *OAL* with this new enviorment sensing method in mind. The team that works on this is called the HOAL team and will be refereed to this for the remainder of the report. As per writing the HOAL team have produced two contemporary snake robots, named Mamba and Boa.

3.2.1 The Mamba snake robot

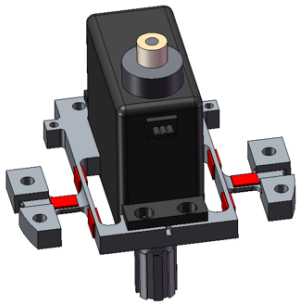
The following subsection is adapted from [5]

Mamba is the second snake robot based on the OAL premise created at the Norwegian University of Science and Technology, shown in Figure 2. NTNU used the snake robot for robotic testing on both ground and in water. One of the main differences between *Mamba* and the previous version Kulko as presented by Pål lilljebekk was the new intrinsic force/torque sensor system.

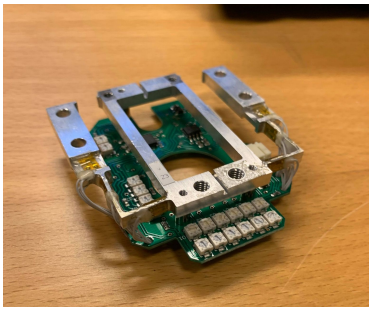
Figure 2: The Mamba snake robot, courtesy of [6]

The sensor system of the Mamba snake robot

The force/torque measurement system in the *Mamba* robot is based on strain gauges. As shown in figure 3, the strain gauges are mounted on an aluminum frame perpendicular to each other, enabling it to measure force and torque on three axes, making it a 6-axis/multi-axis force torque transducer. Figure 3a and Figure 3b presents two different versions of the sensor system, as there have been several attempts to both create and improve the system. The situation however, as indicated by *Fredrik Veslum* [6] showed that both systems had problems with noise, hysteresis and temperature deviation. This including that Mambas electronics had become outdated at the time of Fredrik Velum's experiments made the robot unfit for further development.



(a) 3D-model of an older version(v0) of the force-torque measurement system



(b) The strain gauge sensor system (v3) mounted on the amplifier circuit board

Figure 3: The strain gauge sensor system in the *Mamba* snake robot. Courtesy of [6]
.

3.2.2 The Boa snake robot

After the problems that were present with *Mamba*, the HOAL team started developing a new snake robot called *Boa*. It is the third snake robot based on OAL created at NTNU, and is currently close to being in a runnable state and to be used for experimentation. The Boa snake in comparison to the previous snakes employs a new version of Pål Lilljebekks measurement system based on industrially created sensors instead of *Mambas* in house design.

Figure 4: The Boa snake robot, courtesy of [7]

The sensor system of the Boa snake robot
Boas measurement system similarly to Mambas still uses strain gauge measurement technology to measure the forces and torques, but has instead opted into using commercially available senors combined together. This sensor combination is something developed by the HOAL team in cooperation with [5] and uses two different sensors locked together by a 3D-printed linkage to achieve necessary measurements. Boa is however still under development, so the sensor system solution may have changed since the time of writing.

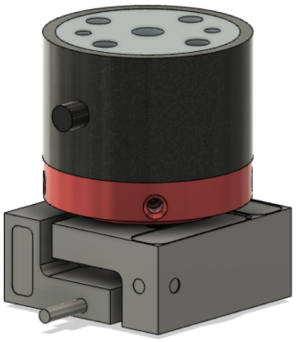


Figure 5: Boa sensor system, courtesy of [5]

3.3 Previous simulators created for snake robot research

The topic of creating a viable simulation enviorment at ITK for a snake robot has been attempted twice before. The first was created in 2018 called *Snakesim* shown in [8] and the other was a master thesis produced in 2019 called *Simulator for Obstacle Aided Locomotion in Snake Robots* shown in [9].

3.3.1 SnakeSim

The first simulator as mentioned was called Snakesim and was a ROS + Gazebo based simulator created to provide a virtual rapid-prototyping framework for *Perception Driven Obstacle-Aided Locomotion (POAL)*. It featured the ability to simulate a snake robot in environments cluttered with obstacles, aswell as the ability to add or remove sensors as necessary. A depiction of how the software system communicated can be seen in figure 6. According to the research articles conclusion the simulator worked very well and made it possible to swap out the sensor reading portion in the simulator with the sensors from the actual snake robot. In this way it was basically a plug and play system which could first asses how a snake robot would react to an enviorment and then experiment with the robot in the same enviorment.

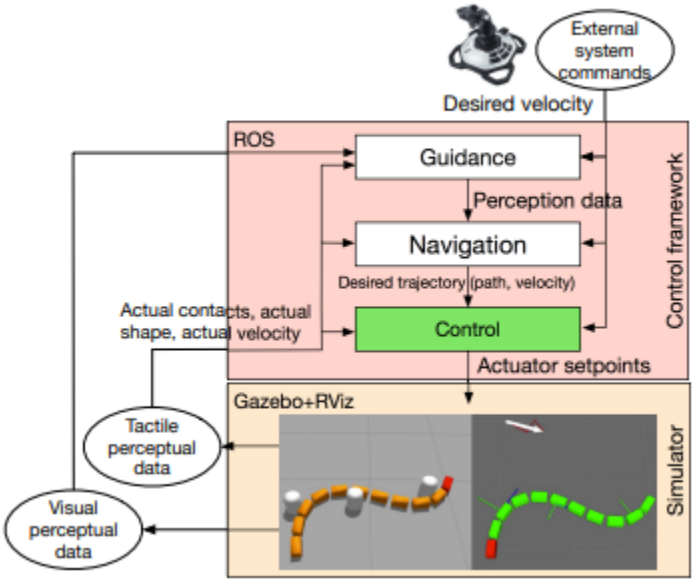


Figure 6: Snakesim software architecture, courtesy of [8]

3.3.2 Simulator for OAL

The second simulator created at NTNU for snake robot research was *Simulator for Obstacle Aided Locomotion in Snake Robots*. This simulator was created as part of a master thesis, and was created with guidance from one of the creators of Snakesim. Instead of using an existing simulator framework, this simulator was created solely in matlab implementing the physics functions directly. This allowed the simulator to be tailor made towards testing specific concepts, but as the paper discusses there were cases where the laws of energy conservation and momentum were violated. This means that it ended up with simulations that does not match the real world, likely due to not implementing a tested physics engine as a base line. The paper does however conclude that the premise of *HOAL* should be further researched.

3.3.3 Motivation for creating another novel snake simulator

As shown in chapter 3.3.1 and 3.3.2, both present a viable option to provide simulations for snake robot research. However due to different circumstances such as lack of version control and lack of git commits both are as per now either lost or non functional. They are by no means useless as they provide an abundance of information on how to develop a simulator, but it does require a new one to be created. This also underlines the importance of good version control for the coming master thesis.

3.4 The structure and theory of physics simulators

Another important subject, is the concept of what a simulator is, what it consists of and it's use cases. This is because a simulator is a widely used term to explain anything which digitally mimics a real world phenomena, but a simulator usually consists of multiple programs interacting. Here namely the physics engine and the rendering engine which works in unison as shown in figure 7.

3.4.1 The rendering engine

The rendering engine is the software responsible for controlling and rendering graphics in a simulation. It does this by continuously receiving information from the physics engine, and uses the positional data to move the objects in the simulation an appropriate amount. It is a useful tool to provide a more human interactable enviorment and varies from simulator to simulator. Some of the most common rendering engines are *OpenGL* used by MuJoCo and PyBullet and *OGRE* used by Gazebo and Isaac sim.

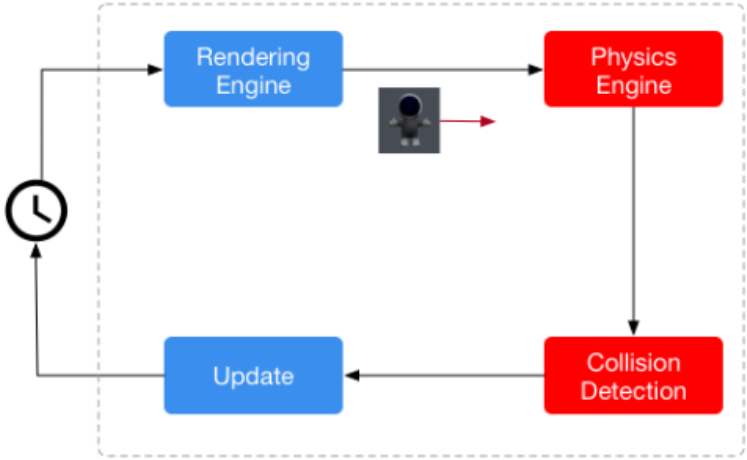


Figure 7: The simulation loop where the rendering engine is marked blue and the physics engine is marked red. Courtesy of [10]

3.4.2 The physics engine

The physics engine is the part of the simulator which calculates how the simulation should proceed, from one time step to the next. It is essential in any real time simulation system, and there exists many different versions. The physics engine can be further broken down into two parts namely *collision detection* and the *physical engine calculation* part. These two work together to both calculate and check for collisions continuously in the simulation which makes the movement and behavior of objects in the virtual world appear more realistic.

3.5 Contemporary physics engines

In today's marked, there exists many viable physics engines. There are however some that stand out, and appear more often than others in both research and games. Four of these which are relevant to the researched simulators in chapter 4 are presented here.

3.5.1 Bullet

Bullet is at the time of writing one of the more implemented physics engines within robotic research. It is an open source project built in 2013 and is offered by many of the front-runners within simulation, such as Gazebo, CoppeliaSim and alot of video game platforms. It is written in C/C++ and is continuously updated by the community to help it both evolve and stay up to date with current needs. It's latest release at the time of writing is Bullet 3.2.4 released on the 25. April 2022.

3.5.2 ODE

ODE is another one of the more used physics engines similarly to Bullet. It is present inn almost all widely used robotics simulators such as Gazebo, Webots, CoppeliaSim and is also open source. It is written in C/C++ and was first released on the 8. may 2001 making it quite old in the technological perspective. Its last stable release was July 30, 2020 meaning it is still being updated too this day.

3.5.3 MuJoCo

MuJoCo is one of the newer physics engines which is starting to make more and more appearances in simulator usage. It is known to be very good at specifically collision physics [11] which makes it interesting for the premise of OAL. It is likewise as the others written in C/C++ and is open source. MuJoCo was first released in 2015, but re-released

as MuJoCo 2.0 in 2018. Its latest stable release is MuJoCo 2.3.0 released on the 18. Oct 2022 and is continuously being updated through community contributions.

3.5.4 PhysX

PhysX is an open-source real time physics engine developed by Nvidia as part of their Nvidia GameWorks software suite. It was originally released as just PhysX in 2003 but has since then been updated all the way to the newest PhysX 5 which has its latest stable release on October 12, 2021. It is mostly known as a video game engine but in NVIDIA's newest project Isaac sim, it has been included as the sole physics engine meaning Nvidia is trying to make it more applicable to research aswell. It has interfaces with both C and Python.

3.5.5 Key differences between the physics engines

As can be seen from the chapter 3.5.1, 3.5.2, 3.5.3 and 3.5.4 they all seem to have similar features and are hard to tell apart. There are however a couple key differences between them based on how they approach physics calculations. The first key difference between these physics engines is the type of equations they use to simulate dynamics. Bullet uses a Lagrangian approach, which is based on the principle of least action. ODE uses a Newtonian approach, which is based on Newton's laws of motion. MuJoCo uses a Newton-Euler approach, which is also based on Newton's laws but includes additional terms to model the rotational motion of objects. PhysX uses a similar Newtonian approach to MuJoCo.

Another key difference is the type of collision detection algorithms they use. Bullet uses an algorithm, which is optimized for high-speed collision detection. ODE uses a more general-purpose algorithm that can handle a wider range of collision scenarios. MuJoCo and PhysX use similar algorithms for collision detection which are both more general purpose.

3.6 Known issues with physics engines

One of the main known problems with all existing physics engines in general is that of collision detection and it's problems with high velocity objects. It is something that stems from the very nature of physics engines and how they use time steps to move the physics forward. This means that if two objects manage to collide between time steps, they are not caught by the physics engine leading to objects intertwining. This further leads to physics breaking down and as a result can ruin simulations. The physics engines that use this kind of collision detection are called DCD (*Discrete collision detection*). There does however exist a sort of solution to this problem, which is called CCD (*Continuous*

collision detection). CCD instead of just checking for collisions at each time step also tries to predict the movement of the object, until the next time step occurs. In this way it is able to prevent the collision problems, but does in turn require alot more computational power meaning that it is not always a viable solution when real time is of the essence. It is worth mentioning that ODE, Bullet, MuJoCo and PhysX all offer CCD.

4 Researching, presenting and selecting simulators

This section covers the work done in researching, presenting and selecting simulators to proceed experimenting with. A recent review [12] on simulators for robotic applications provides a good overview of the current state of the field. The following section is heavily based on the aforementioned review.

An important first step in this project was to ascertain what makes a simulator useful for snake robot research. To achieve this a literature search was conducted using google scholar which yielded a couple of interesting articles, especially [12]. This research paper provides both a definition of what a simulator should contain aswell as graphs and tables comparing information. One of these tables can be shown in figure 8 which shows how many times relevant simulators have been cited in research articles found on google scholar. This provides insight into what simulator platforms are preferred by the research community which may be a good indicator on which platforms should be researched further.

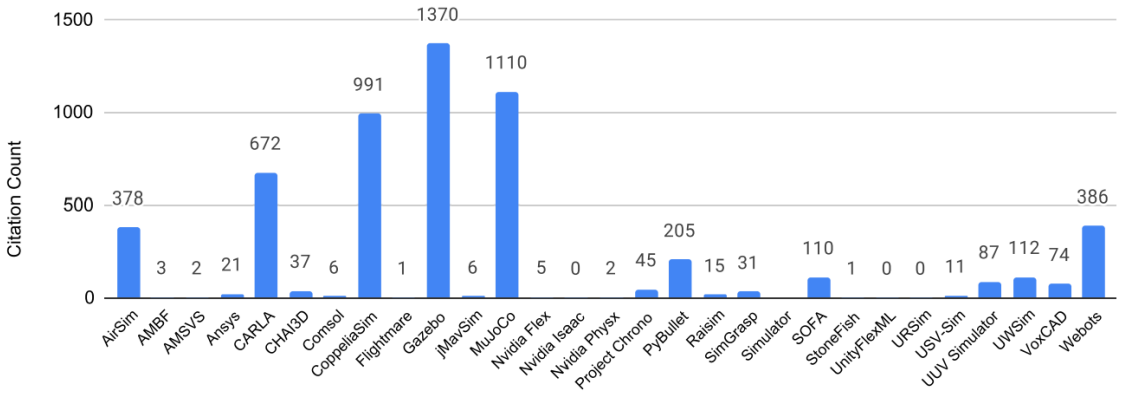


Figure 8: Citation count from 2016 to 2020 for reviewed simulators. Courtesy of [12]

Based on the number of citations in the graph presented in figure 8, it was decided too present and compare most of the simulators that had 200 citations or more and that were deemed usable for mobile robotic research by [12]. This is because it narrows down the amount that has to be researched as part of this preliminary project aswell as meaning that the simulators are most likely widely used. Including this some other simulator platforms that were found interesting have been included aswell.

4.1 Contemporary physics simulators

This chapter provides an overview over investigated simulator frameworks for this research paper. A structural overview can be seen in appendix 9.1. All logos in this chapter are taken from their respective homepages and links can be found in references [13], [14], [15], [16], [17] and [18].

4.1.1 Gazebo

As seen from figure 8, Gazebo is one the most mentioned simulator platforms in google scholar meaning that it is one of if not the most used simulator platform for robotic research. It was firstly developed in 2004 through something called *The player project* and has since then become fully standalone and is now supported by OSRF (Open-source Robotics foundation) for future development. It is designed to run on the Linux OS called Ubuntu and is like many other simulation tools written in C++. It provides API support in C++ aswell making it possible to run simulations from external programs. Gazebo also provides many different physics engines such as ODE, Bullet, Simbody and Dart making it versatile in what the user values when completing simulations, either speed, realism or other factors. Lastly it contains a large repertoire of sensors and objects which makes rapid development of robotic simulations both quick and easy.

4.1.2 CoppeliaSim

Coppelia sim (previously called V-rep) is another popular choice when simulating ground based mobile robots such as snake robots. It is possible to run on all major operating systems such as Windows, Linux and Mac OS making it easy to work with regardless of the available computer. It is built around having both Python and Lua scripts that it gives to each object in the simulation, aswell as API's in many different programming languages which one can use to interact with the simulation. It also provides many different physics engines such as ODE, Bullet, Newton, Vortex and MuJoCo making it very versatile based on what the simulation should focus on. Lastly and equally to Gazebo it also contains a large repertoire of available sensors and prebuilt objects helping with rapid development.

4.1.3 Webots

Webots was developed by the Swiss Federal institute of Technology in 1996 and has since then been continuously updated with newer releases. It has also since 2018 been released as free and open source under the Apache 2 license. Webots runs equal to Coppelia Sim on all major operating systems such as Windows, Linux and MAC OS and is written in C++. It has API support for almost all major programming languages within robotics such as C, C++, Python, Matlab and Java making it quite versatile as long as the user knows more or less any programming language. Webots does however only allow for the ODE physics engine making it less versatile. Lastly Webots also sports a large variety of existing robots, actuators and sensors which is useful for rapid development.

4.1.4 Isaac-sim

Isaac-sim is NVIDIA's attempt at creating a competitive and viable robotic simulator within their own framework. It is not used for robotic research by the majority of the community as per now 8 but this is something that might change in the future. It is available on both Linux and Windows and has it's main programming interface through either C or Python. Isaac sims main simulator is the PhysX 5 and it does per now not allow other physics engines to be run within it's simulation framework. It does however benefit from being able to use Ray-tracing to create very real looking and life like simulation environments aswell as many state of the art possibilities within cloud computing and machine learning systems. It's latest release as per writing is 01.01.2022 but gets continuously updated by NVIDIA.

4.1.5 Unity

Unity is a game engine created by Unity technologies in 2005. It is not designed to be used for robotics, but contains alot of the necessary tools to do so which is why it is presented here. It benefits from having a large world wide user base, aswell as continues updates to it's system. It is written in C++/C# and allow headless controll through a well established interface towards many

code editors. Including this Unity's personal edition is free to use and it's last stable release was on 28.07.2022.

4.1.6 MuJoCo (Standalone)

MuJoCo including being a physics engine also offers a standalone product which can be used for simulating robotics. It uses the OpenGL rendering engine and direct communication with the MuJoCo source code to complete simulations. Using a standalone variant like this gives alot more flexibility in controlling the simulations, but may lead to a longer development time as it lacks the user interfaces of the other simulators such as Gazebo, Coppelia Sim and Isaac-sim. For more direct information on the physics engine, look at chapter 3.5.3.

4.1.7 PyBullet (Standalone)

Similar to MuJoCo, Bullet also offers its own standalone version for simulation purposes. It uses the OpenGL rendering engine with direct communication to the Bullet source code for simulation. It does however allow a python interface towards the physics engine which makes it both easy to implement and use. For more direct information on the physics engine, look at chapter 3.5.1.

4.2 Selecting simulators for further study

After gathering several different and viable simulators, it was important to narrow them down even further. It was decided to continue with three simulators as experimenting with a simulator requires learning alot of specific details about each framework which again requires alot of work and time. To make it easier to sort out three simulators to experiment with, a tabular was created which listed the most desirable aspects for a snake robot research simulator.

Must have:

- Ability to accurately simulate collision physics (Meaning does not struggle with objects intertwining during simulations)
- Ability to run simulations without the rendering engine (headless mode) to help speed up simulations for machine learning
- API with good documentation

- Ability to import URDF, MJCF or other reasonable file formats
- Ability to log sensor measurements such as velocity, position, orientation, forces and accelerations

Nice to have:

- Open-source simulator and physics engine (meaning possible to see all the source code. This also implies that the they are free to use)
- Available on all major OS (Linux, Windows, MAC)
- Ability to implement ROS (Robot operating system)
- Compatibility with multiple programming languages

The point on accurately simulating collision physics is more related to the physics engines the simulators use and therefore has to be assessed through experimentation. The other parts however can be seen from the rest of chapter 4.
After comparing and discussing with the supervisor it was decided that the three simulators to continue testing with was **Coppelia Sim, Gazebo and Isaac sim**. Gazebo was selected due to being the most prevalent simulator for robotic research (see figure 8) aswell as the main simulator for the previous attempt named *Snakesim* shown in chapter 3.3.1. Coppelia Sim was selected due to having access to MuJoCo aswell as also being widely used by the research community. Lastly Isaac sim was chosen due to being brand new aswell as containing alot of interesting features such as cloud computing and machine learning gyms. Including these reasons all the chosen simulators also contain all the must have points aswell as alot of the nice to have from the tabular presented in this sub chapter.

5 Experiment method

After selecting three simulators to continue experimenting with, it was then time to design a test. The goal of the test as discussed in chapter 4.2 was to investigate each simulators ability to accurately simulate collision physics. This is because using collisions with external objects to create momentum is one of the main concepts of OAL, and as mentioned in chapter 3.6, collision physics is also one of the main issues with physics engines. A couple experimentation methods were proposed, but in the end it was settled on simply bouncing a ball while looking at its position and velocity data. This gives insight into the physics engines ability to rapidly swap momentum directions which is crucial for when a snake robot pushes of against an object. A simple test like this also helps make time for familiarization with the frameworks API which is another crucial part when selecting simulators in the end. It should be pointed out however that this will not explore the physics engines ability to handle large workloads or complex dynamics, but this is deemed somewhat unnecessary as snake robot collisions will mostly not push a simulators limits. The data used for the ball object in the test aswell important values for the physic engines are presented in tabular 1.

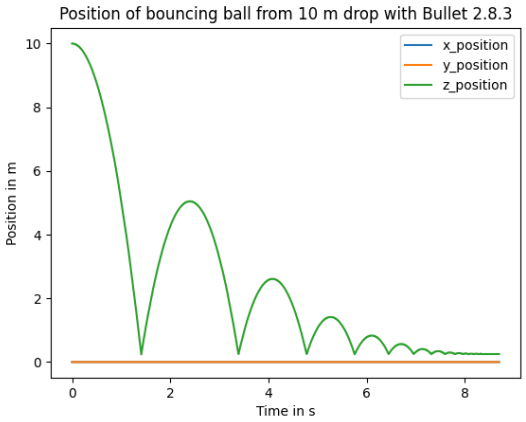
Experiment values	
Density	1.9 kg/m^3
Mass	1 kg
Radius	0.5 m
Drop height	10 m
Restitution	1.00
Gravity	-0.98 m/s^2
Time step	1 ms
Max velocity	100m/s

Table 1: Experiment values for ball object and physics engine used in experiments

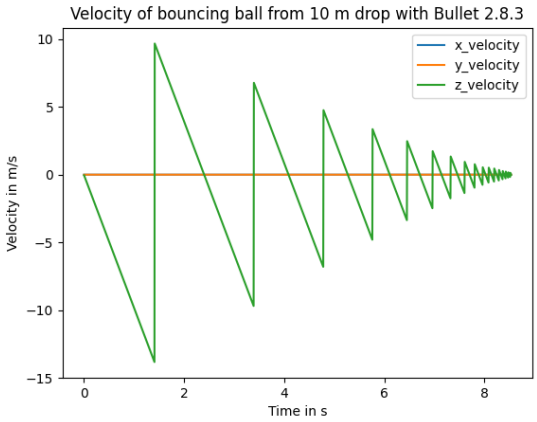
6 Experiment results

6.1 Testing collision in CoppeliaSim

The first simulator to be tested was Coppelia sim which is presented in chapter 4.1.2. Coppelia sim has access to two of the main physics engines that were considered to be useful for simulation purposes, namely Bullet and MuJoCo. It was therefore decided to run the tests with these two physics engines and look at both ease of implementation aswell as position and velocity results.

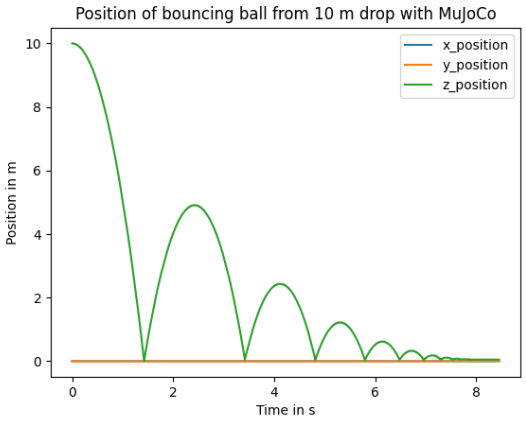


(a) Position graph for bouncing ball with Bullet

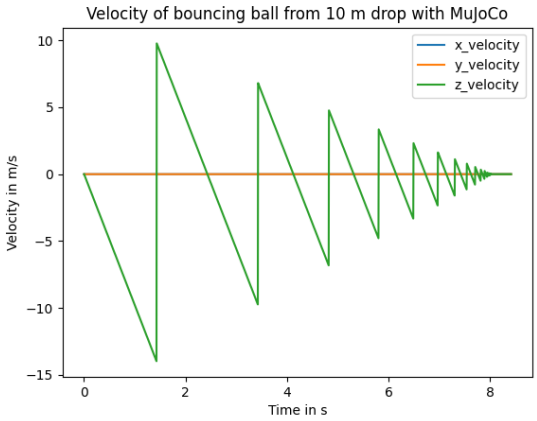


(b) Velocity graph for bouncing ball with Bullet

Figure 9: Plots for the Bullet engine



(a) Position graph for bouncing ball with MuJoCo

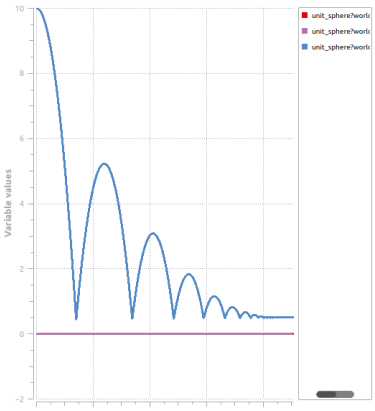


(b) Velocity graph for bouncing ball with MuJoCo

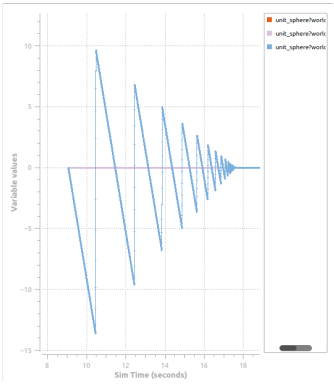
Figure 10: Plots for the MuJoCo engine

6.2 Testing collision in Gazebo

The second simulator to be tested was Gazebo which is presented in chapter 4.1.1. Gazebo also has access to multiple physics engines, but since Bullet was already tested with Coppelia sim, it was decided to test Gazebo with the ODE engine.



(a) Position graph for bouncing ball with ODE created using Gazebos built in graphing tool

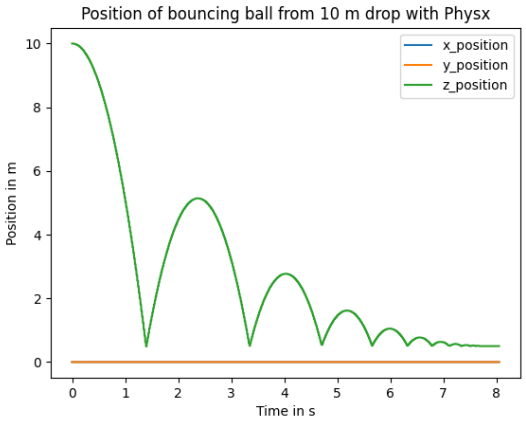


(b) Velocity graph for bouncing ball with ODE created using Gazebos built in graphing tool

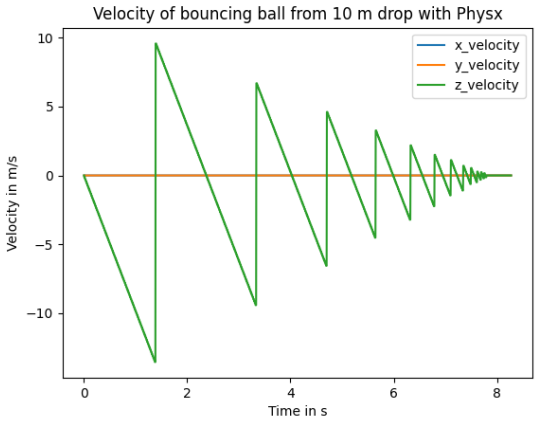
Figure 11: Plots for the ODE engine

6.3 Testing collision with Isaac sim

The last simulator to be tested was the Isaac sim simulator and it's physics engine PhysX.



(a) Position graph for bouncing ball with PhysX



(b) Velocity graph for bouncing ball with PhysX

Figure 12: Plots for the PhysX engine

7 Discussion

This section discusses results from the experiments conducted in chapter 6 and the implication of these. It also discusses findings for the tested simulator frameworks aswell as their strengths and weaknesses.

7.1 The physics engines

From the results in chapter 6, there are a few interesting observations to be made. The first is that the time it takes before the ball loses all momentum varies from simulator to simulator. This is because every physics engine has its own way of enabling restitution (bounciness) and friction, meaning that the slight differences in simulation time should be contributed to minor errors during setup. The second observation is that all physics engines in all simulators manage to accurately portray changes in both velocity and position for the ball. This is to be expected as simply dropping a ball requires little processing power aswell as providing only one set of collisions for the engine to calculate. It does however show that simple collisions work within the investigated simulators and should indicate that simple snake robotic simulations, which also requires few collisions, should be expected to work aswell. Including this simple test, a more in depth one of three physics engines can be found in *S M Longshaws* paper, *Numerical Modelling and Visualization of the Evolution of Extensional Fault Systems* [19]. This paper compared the frame rates of the three physics engines ODE, Bullet and PhysX while dropping 625 spheres in consecutive iterations. This shows that when the physics engines are put under more strain, some engines perform better than others as shown in figure 13. It should however be noted that the experiment was conducted in 2012 which in a technological perspective is quite old, meaning that the results could have changed by then.

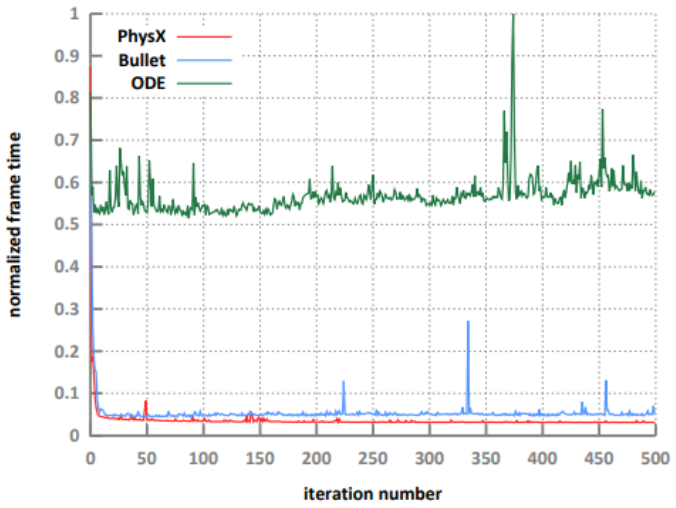


Figure 13: Frame rate comparison of PhysX, Bullet and ODE when dropping 625 spheres. Test conducted in 2012 which should be taken into consideration. Courtesy of [19]

7.2 The simulator frameworks

Including looking at the physics engines, it is also important to look at the simulator frameworks which hosts them. These do not directly affect how the physics work within a simulation, but controls everything around it from setup, programming, GUI and API. Each framework has some major benefits and some drawbacks which weighs in on what should be proceeded with.

Firstly Gazebo, Gazebo benefits mainly from being one of the most used simulator frameworks for robotic research as shown in figure 8. This provides it with a plethora of research papers, community tutorials and other useful documentation that made both the simulator and it's API easy to comprehend. It allows multiple physics engines and is open-source meaning it is free to use and allows the source code to be inspected. Including this there are no major drawbacks with Gazebo and the only thing this report could find were more personal preference differences such as lack of MJCF import formats and lack of variety in programming languages.

Coppelia sim as explained in chapter 4.1.2 benefits majorly from having access to the MuJoCo physics engine as this physics engine is known to be quite adept at collision physics [11]. It does however seem that in comparison with Gazebo and Isaac sim, Coppelia sims API contains less functions with more limited descriptions. This meant that it took more time during experimentation to figure out how to access objects and gather information, which can pose problems with further development. It does also not have the same community presence as Gazebo making both tutorials and research papers harder to locate. It should however be mentioned that these are minor drawbacks and Coppelia sim on it's own would still be plausible for snake robotic research.

Isaac sim benefits alot from being produced by Nvidia and it shows when using it's graphical user interface (GUI) and it's application interface (API). Everything is very well described and explained with well thought out tutorials which made understating it quite simple. Isaac-sim does also provide alot more options in it's GUI than Gazebo and Coppelia Sim such as a large repertoire of premade materials which can be added onto objects. Further more it has access to many other interesting software solutions such as machine learning gym and cloud computing presented in chapter 4.1.4. Isaac sim does however contain two drawbacks. Firstly Isaac sim is a commercial product meaning that it's code is closed source. This can make it difficult to understand how the software works, to customize the software if needed or to troubleshoot problems that may arise. Secondly Isaac sim requires the computer to contain a Nvidia Ray Tracing (RTX) graphics card to be able to run at all. These are usually quite expensive meaning that even though Isaac sim is free to use as per now, there still is a hidden cost behind it. This also makes simulations difficult to complete outside of a dedicated work computer and is something that should be considered before selecting Isaac sim.

8 Conclusion

After researching both physics engines and simulator frameworks for this project assignment, it has become clear that there are both positive and negative aspects with every physics engine and simulator framework. However, it has also become apparent that every physics engine and simulator framework are more or less suited to be used for further snake robot research as shown by experiments in section 6. It is therefore much up to personal preference aswell as future potential that a simulator was selected.

In conclusion this project recommends that simulator construction and implementation continues on Isaac sims platform. Isaac sim even with it's drawbacks also have alot of the same positives that Gazebo had, but due to future potential such as cloud computing, machine learning gyms aswell as alot of other nice to have features Isaac sim seems to be the better choice. There is also not much research completed on robotics in Isaac sim meaning that continuing with Isaac sim can provide the added benefit of researching the usefulness of the simulator. However in the case that the user does not have access to an RTX graphics card, the project recommends Gazebo instead.

9 Appendix

9.1 A structural comparison of physics simulators

	Gazebo	Coppelia Sim	Webots	Isaac sim	Unity	MuJoCo	Pybullet
General info	Open Source Robotics Foundation	Coppelia robotics	Cyberbotics Ltd	NVIDIA	Unity Technologies	Deep Mind	Bullet physics
Supported OS	GNU/Linux	GNU/Linux, Mac-OS, Windows	GNU/Linux, Mac-OS, Windows	Ubuntu, Windows	GNU/Linux, Mac-OS, Windows	GNU/Linux, Mac-OS, Windows	GNU/Linux, Windows, MAC-OS
Last release	11.0.0 (30.01.2020)	4.4 (22.08.2022)	R2022 (13.09.2022)	01.01.2022	28.07.2022	2.3.0 (18.10.2022)	3.2.4 (25.04.2022)
Programming language	C++	Lua, Python	C++	C, Python	C++/C#	C/C++	Python
API support	C++	C, C++, Python, Matlab, Java	C, C++, Python, Matlab, Java	C, Python	C++/C#	C	Python
Physics engines	ODE, Bullet, Simbody, Dart	Bullet, ODE, Vortex, Newton, MuJoCo	ODE	PhysX 5	PhysX	MuJoCo	Bullet
CAD files support	SDF, URDF, OBJ, STL, Collada	OBJ, STL, DXF, 3DS, Collada, URDF	WBT, VRML, X3D	MJCF, XML, URDF	FBX, Collada, DXF, OBJ	MJCF, XML, URDF	MJCF, URDF, SDF
Licences	Open-source	Free educational use (Students and research)	Open-source and commercially available	Local free use with cloud upgrades possible	Open-source and commercially available	Open-source	Open-source
Supports Force/Torque sensor	✓	✓	✓	✓	✓	✓	✓

References

[1] Pål Liljebäck. *Modelling, development, and control of snake robots*. Vol. 2011:70. Doktoravhandlinger ved NTNU (trykt utg.) Norwegian University of Science, Technology, Faculty of Information Technology, Mathematics, and Electrical Engineering, Department of Engineering Cybernetics, 2011. ISBN: 9788247126677.

[2] A. A. Transeth et al. "Snake Robot Obstacle-Aided Locomotion: Modeling, Simulations, and Experiments". In: *IEEE Transactions on Robotics* 24.1 (2008), pp. 88–104. DOI: 10.1109/TRO.2007.914849.

[3] Aksel Andreas Transeth and Kristin Ytterstad Pettersen. "Developments in Snake Robot Modeling and Locomotion". In: *2006 9th International Conference on Control, Automation, Robotics and Vision*. 2006, pp. 1–8. DOI: 10.1109/ICARCV.2006.345142.

[4] T. Yoshikawa and A. Sudou. "Dynamic hybrid position/force control of robot manipulators-on-line estimation of unknown constraint". In: *IEEE Transactions on Robotics and Automation* 9.2 (1993), pp. 220–226. DOI: 10.1109/70.238286.

[5] Victor Melhuus Joel Mörlin and Oscar Mørk. *Intrinsic Force-Torque Sensor System for a Next Generation Snake Robot*. 2021. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2781042 (visited on October 12, 2022).

[6] Fredrik Veslum. "Assessment of the Mamba snake robot sensor system". In: (2020).

[7] Jostein Løwer, Irja Gravdahl, Damiano Varagnolo, and Øyvind Stavdahl. "Proprioceptive contact force and contact point estimation in a stationary snake robot". In: (2022).

[8] Filippo Sanfilippo, Øyvind Stavdahl, and Pål Liljebäck. "SnakeSIM: a ROS-based control and simulation framework for perception-driven obstacle-aided locomotion of snake robots". In: *Artificial Life and Robotics* 23 (August 2018). DOI: 10.1007/s10015-018-0458-6.

[9] Atussa Koushan. *Simulator for Obstacle Aided Locomotion in Snake Robots*. 2019. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2780900/no.ntnu%3Ainspera%3A56990118%3A20964868.zip?sequence=2 (visited on October 12, 2022).

[10] Harold serrano. *How a simulator works*. 2019. URL: https://www.haroldserrano.com/blog/the-heart-of-a-game-engine-the-game-engine-loop (visited on November 30, 2022).

[11] Tom Erez, Yuval Tassa, and Emanuel Todorov. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4397–4404. DOI: 10.1109/ICRA.2015.7139807.

[12] Anthony Vanderkop Jack Collins Shelvin Chand and David Howard. *A Review of Physics Simulators for Robotic Applications*. 2021. URL: https://ieeexplore.ieee.org/document/9386154 (visited on November 1, 2022).

[13] Gazebo. *Gazebo logo*. URL: https://gazebosim.org/home (visited on November 17, 2022).

[14] CoppeliaSim. *CoppeliaSim logo*. URL: https://www.coppeliarobotics.com/ (visited on November 17, 2022).

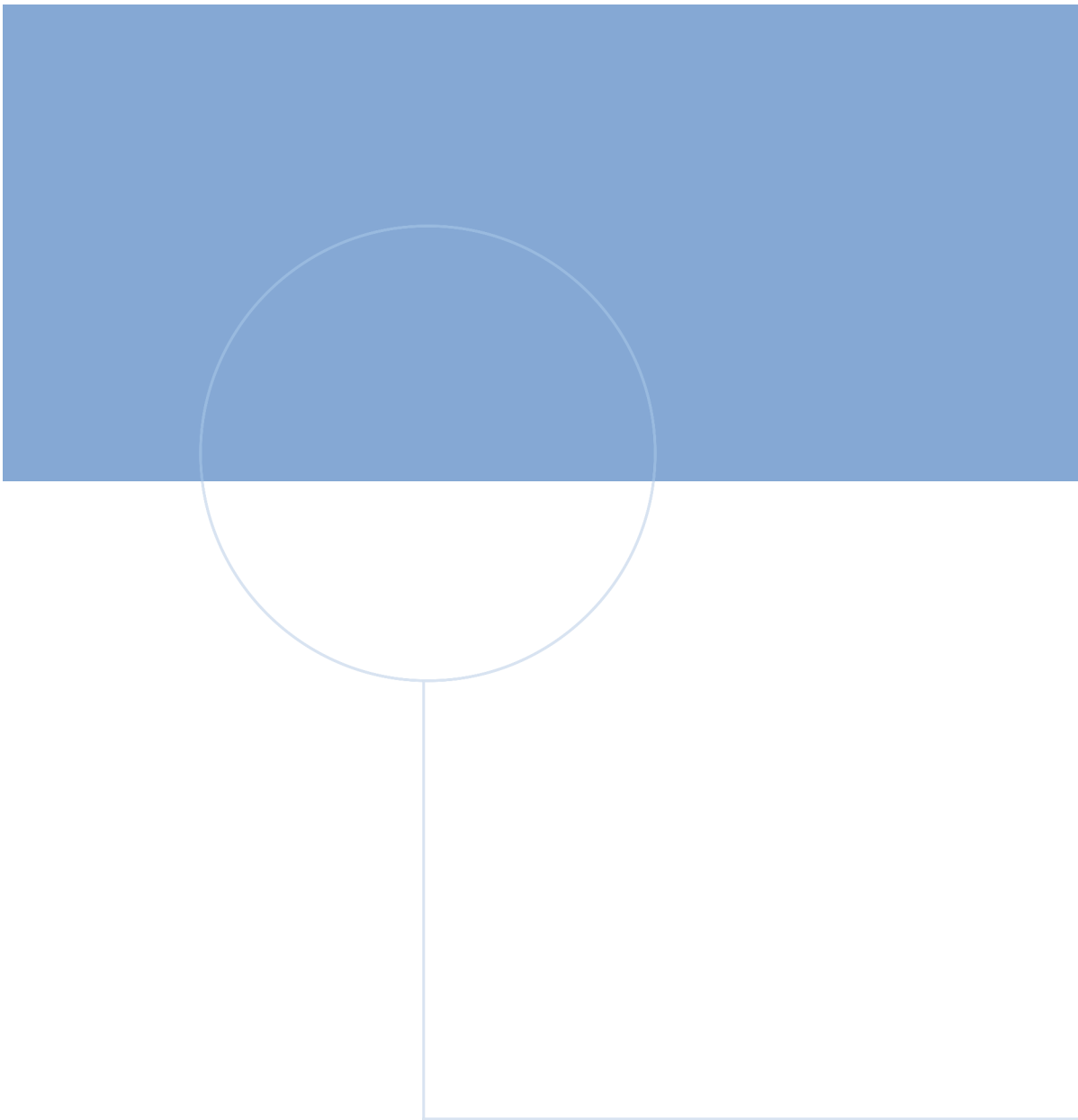
[15] Webots. *Webots logo*. URL: https://www.cyberbotics.com/ (visited on November 17, 2022).

[16] Nvidia. *Nvidia logo*. URL: https://www.nvidia.com/en-gb/about-nvidia/legal-info/logo-brand-usage/ (visited on November 19, 2022).

[17] Unity. *Unity logo*. URL: https://unity.com/legal/branding-trademarks (visited on November 19, 2022).

[18] MuJoCo. *MuJoCo logo*. URL: https://mujoco.org/ (visited on November 19, 2022).

[19] S M Longshaw. "Numerical Modelling and Visualization of the Evolution of Extensional Fault Systems". English. PhD thesis. United Kingdom: The University of Manchester, October 2011.



NTNU

Norwegian University of
Science and Technology