LONDON'S GLOBAL UNIVERSITY

Exploring the usage of Unreal Engine 4 as a planetary rover locomotion simulator

Candidate Number: NRSN8[1]

BSc Computer Science

Supervisors: Dimitrios Kanoulas, Rae Harbird,
Sebastian Friston, Simon Julier

Submission date: 21 April 2021

Abstract

In recent years, there has been an increasing number of space mission programs involving planetary exploration by means of robotic systems such as landers and rovers. The usage of rovers expands the exploration areas and potentially increases the scientific return of a mission and, to a large extent, the mission life and science objectives are determined by the robustness and capability of the rovers' mobility systems. However, designing and testing the aforementioned systems is difficult and resource-consuming, requiring a terrain simulant, a way of dealing with differences in gravity, and a physical robot model. Moreover, the terrain simulant properties change as a consequence of driving through it, making real testing often not reproducible. Therefore, an accurate, software-based simulation tool that can predict rover locomotion performance might reduce the costs associated with designing such a mobile robot and act as an operational risk reduction tool.

In collaboration with researchers from NASA's Ames Research Center, this project seeks to determine if Unreal Engine 4 can be used as a simulation tool for planetary rovers by verifying that it can perform physics simulation to an appropriate level of fidelity through a series of experiments that capture various aspects of its physics engine. The data obtained from running these experiments are validated against results gained by creating the same scenarios in Gazebo, one of the most used robotics simulators.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Rae Harbird, Simon Julier, Dimitrios Kanoulas, and Sebastian Fristion for providing me with an opportunity to work on a very exciting project that was born from the shared interest in robotics, and space exploration. Their continued involvement, advice, stimulating remarks, enthusiasm, and knowledge made this project possible and supported me throughout the entire endeavour.

I would like to thank members of NASA Ames Research Center, especially Michael Furlong, Arrno Rogg, and Massimo Vespignani for sharing their knowledge and their interest in this project from the very beginning. Their input had a strong influence on the direction of this project and provided invaluable feedback along the way.

Contents

1 Introduction **2**
- 1.1 Motivation . . . 2
- 1.2 Project Aims . . . 4
- 1.3 Evaluation Method . . . 4
- 1.4 Report Structure . . . 4

2 Background **5**
- 2.1 Computer Simulations in Science and Engineering . . . 5
- 2.2 Introduction to Terramechanics . . . 6
- 2.3 Survey of Terramechanics Simulators . . . 6
- 2.4 Technologies Used . . . 9
- 2.5 Summary . . . 10

3 Initial Evaluation **11**
- 3.1 General Approach Used to Obtain the Data . . . 11
  - 3.1.1 Obtaining the Unreal Results . . . 12
  - 3.1.2 Obtaining the Gazebo Results . . . 12
  - 3.1.3 Obtaining the Theoretical Results . . . 13
  - 3.1.4 Visualising the Results . . . 13
- 3.2 Gravity Drop Test . . . 13
- 3.3 Impulse Applied to a Cube . . . 17
  - 3.3.1 Neglecting Friction . . . 17
  - 3.3.2 Including friction . . . 19
- 3.4 Constant Force Applied to a Cube . . . 22
- 3.5 Force Patterns Applied to a Cube . . . 24
  - 3.5.1 Pattern One . . . 25
  - 3.5.2 Pattern Two . . . 26
  - 3.5.3 Pattern Three . . . 27
  - 3.5.4 Pattern Four . . . 28
- 3.6 Summary . . . 29

4 Adapting an Existing Physics Benchmark **30**
- 4.1 Obtaining the Data . . . 30
- 4.2 Theoretical Description . . . 31
- 4.3 Simple Scenario . . . 33
- 4.4 Complex Scenario . . . 35
- 4.5 Summary . . . 36

5 Simulating a Basic Rover Model **37**
5.1 Cylindrical Wheels Model . . . 37
5.2 Spherical Wheels Model . . . 39
5.3 Ignoring the Effects of Friction . . . 40
5.4 Summary . . . 41

6 Conclusions **42**
6.1 Achievements . . . 42
6.2 Evaluation . . . 42
6.3 Future Work . . . 43
6.3.1 URDF Importer . . . 43
6.3.2 Further Simulation . . . 43
6.4 Final Considerations . . . 44

A User Manual **48**
A.1 Unreal Manual . . . 48
A.2 Gazebo Manual . . . 49
A.3 Theoretical Solutions Manual . . . 49
A.4 Data Plotting . . . 50

B Source Code Repository **51**

C Code Listing **52**
C.1 Obtaining the Theoretical Results . . . 52
C.2 Chapter 3 Unreal Code . . . 55
C.2.1 Gravity Drop Test Cube - header file . . . 55
C.2.2 Gravity Drop Test Cube - source file . . . 57
C.2.3 Impulse Test Cube - header file . . . 59
C.2.4 Impulse Test Cube - source file . . . 60
C.2.5 Force Patterns Test Cube - header file . . . 63
C.2.6 Force Patterns Test Cube - source file . . . 65
C.3 Chapter 3 Gazebo Code . . . 74
C.3.1 Plugin for the Force Patterns . . . 74
C.3.2 World File . . . 81

Chapter 1

Introduction

1.1 Motivation

In the past two decades, a large number of space exploration missions devoted to planetary surface exploration using mobile robots, or rovers, have been launched by space agencies around the world. NASA's Mars Exploration Rovers, Spirit, and Opportunity proved through their extended mission life that as the duration of rover missions increases, a greater variety of terrain types will eventually be accessed. However, replicating each type of terrain is often not feasible or extremely challenging from a physics perspective since there is a wide range of differences in soil properties between astronomical bodies. Therefore, the ability to predict rover locomotion performance based on different terramechanics models that can take into account variances in soil properties is critical during the design, validation, planning. and operations phases of a planetary robotic mission and can serve as a risk mitigation effort for future missions [1].

As a consequence, several software simulation tools with various complexities have been created. For example, Patel et al. developed a rover chassis evaluation tool called RMPET [2], Jain et al. [3] and Yen et al. [4] describe the development of a virtual rover simulator called ROAMS, Harnisch and Lach [5] describe another simulation tool called 'Off Road Systems Interactive Simulation' (ORSIS). Moreover, two NASA research centers developed such simulation tools: NASA JPL center developed Artemis, a software tool developed to simulate rigid-wheel planetary rover traverses across natural terrain surfaces [6], whilst NASA Ames created the VIPER simulator, a tool which links plan execution, rover simulation, and a high-fidelity, realistic environment [7]. However, despite the variety of simulators available and the time and effort invested into building them, shortcomings are still present, particularly related to features that are not necessary for an accurate simulation but that might provide long-term value, such as advanced graphics processing. Specifically, some of the existing simulators can only load relatively small digital elevation models, which are 3D computer graphics descriptions of elevation data that are used to represent the simulation terrain, therefore curtailing the simulation operations that can be executed, whilst others are limited in terms of lighting and shadows rendering, allowing only one light source in the simulation environment.

Unreal Engine 4 is one of the most advanced game engines available, currently used not only for game development but for a wide range of applications in domains such as architecture, automotive, and transport industry or for training and simulation purposes. Although the usage

of Unreal Engine 4 in the fields of robotics is relatively modest, there are a couple of successful examples, such as USARSim [8], a robot simulator used for education and research. Another successful example is Microsoft's Aerial Informatics and Robotics platform [9], which is built on top of Unreal Engine 4 and provides high-fidelity simulation that can be used for the rapid training of data-driven robotic systems, based on inputs from a wide variety of sensors. Figure 1 captures a subset of these capabilities by showing a scene used to simulate a UAV flying through an urban environment, along with the input from the front-facing camera attached to it.

Figure 1: Snapshot from Microsoft's Aerial Informatics and Robotics platform showing a UAV flying through an urban environment, along with the depth image stream, the materials property view stream, and the front camera image (acquired from [9])

Moreover, the prospect of using Unreal Engine 4 for detailed physics simulation is supported by existing software that achieves accurate physics modelling, but for different purposes. For example, CARLA [10] is a simulator implemented as an open-source layer over Unreal Engine 4 that is heavily used for the development, training, and validation of autonomous driving systems. Similarly, CARLA provides various sensing modalities, many of which can be used for training computer vision algorithms used for autonomous driving systems.

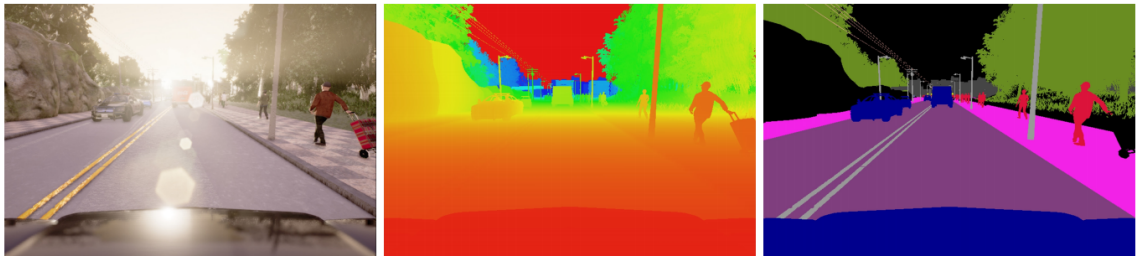


Figure 2: Snapshot from the CARLA simulator displaying three of the sensing modalities provided: the normal vision camera, the ground-truth depth, and the ground-truth semantic segmentation (acquired from [10])

1.2 Project Aims

This project aims to explore the possibility of using Unreal Engine 4 as a planetary rover simulator that can solve the shortcomings discussed in the previous section, such as problems in modelling complex illumination effects, whilst preserving the accuracy of existing solutions. Due to the amount of attention it received from the game development industry, Unreal Engine 4 offers state-of-the-art graphics processing and photorealistic rendering, effectively solving the graphics processing issues of other simulators out-of-the-box. Furthermore, the prospect of using Unreal Engine as a simulator is of interest for researchers from NASA's Ames Research Center, which guided the direction of this project by pointing out the most important aspects required for a planetary rover simulator to be successful. Therefore, based on input from NASA researchers, it has been determined that the performance of Unreal's real-time physics engine is the most important factor in deciding whether it can be comprehensively used as an alternative to existing simulators.

1.3 Evaluation Method

In this project, the evaluation of Unreal's physics engine, PhysX, is performed by creating a series of experiments that capture various aspects of the physics engines in both Unreal Engine 4 and Gazebo, and comparing the results from the two systems against a theoretical solution of each scenario. Gazebo is one of the most used robotics simulators and serves as a basis for the previously-mentioned VIPER tool. The decision of using Gazebo as a validation method was motivated by the lack of access to physical experimental data, and by the fact that it served as a basis for the simulator used by researchers from the Ames Research Center, which offered detailed insights on the results obtained throughout this project.

1.4 Report Structure

This section offers a brief overview of the report structure and its content:

- Chapter 1 introduced the reader to the broad context of this project, its aims, and motivations.
- Chapter 2 will delve deeper into the background work and research that was performed prior to starting the project by presenting a literature review. It will include a more in-depth discussion of the problem and existing solutions, followed by a presentation of the technologies used.
- Chapter 3 will describe the initial physics scenarios tested, accompanied by theoretical considerations. It will include all information needed to recreate the same scenarios in any other simulator and will contain a discussion of the results and the method of obtaining them.
- Chapter 4 will present the process and results of adapting an existing benchmark used by robotics researchers to compare physics engines.
- Chapter 5 will discuss the challenges encountered in trying to simulate a basic rover model in Unreal and Gazebo.
- Chapter 6 will summarise the findings of this project and propose a roadmap for the development and implementation of future research projects in this area.

Chapter 2

Background

This chapter introduces the reader to the scientific area related to computer simulations, focusing on wheel-soil interaction, provides a review of existing simulators, and presents the technologies used in this project.

2.1 Computer Simulations in Science and Engineering

Broadly speaking, a computer simulation represents the use of a computer to predict the behaviour or the outcome of a real-world or physical system, based on the implementation of a mathematical model. Since they provide a method of validating mathematical models, computer simulations have become an important tool for scientific research soon after the emerging of computers, following World War II. Whilst initially used for predicting weather and modelling nuclear detonations, computer simulations are currently extensively employed in a myriad of domains, including theoretical physics, climate science, biology, epidemiology, and many others.

However, a significant drawback of many computer simulations is the use of numerical methods to attempt an approximation of mathematical models for which closed-form analytic solutions are either impossible or extremely challenging to compute and represent. One example that is often encountered is given by ordinary differential equations, which typically do not have analytical solutions, and solving them is based on numerical approximations. This makes even obtaining a correct prediction of the position of a body whose motion is governed by Newton's second law of motion prone to numerical integration errors, as shown in [11]. Furthermore, as described by Paul Bratley et al. in [12], errors can be introduced by simplifying assumptions of the underlying mathematical model that are necessary to allow its implementation on a computer. For example, highly nonlinear functions are often approximated by simpler ones to reduce the computational complexity of the system.

Numerical errors introduced by approximations are even more likely to add up and cause significant differences when used in the context of physics engines. As introduced by Boeing and Bräunl in [13] and Erleben in [14], the typical physics engine contains a variety of sub-modules, each of which can introduce errors in the overall computations. For example, Kavan [15] and Hadap et al. [16] present the effects of collision detection and contact determination on the accuracy of a simulation, whilst Baraff [17], and Erleben [14] cover the main methods used in numerical integration and their potential effects on the numerical accuracy of the simulation.

Therefore, based on the observations made above, it is important to note that despite their complexity, computer simulations are often subject to numerical errors and that all simulation results should be compared against experimental data to ensure their correctness. This approach is used in the experiments conducted in this project and shows that the results presented contain numerical noise caused by the various approximations used by physics engines.

2.2 Introduction to Terramechanics

In the context of this project, terramechanics represents one of the most relevant domains in which computer simulations are heavily employed. In this section, the reader is introduced to the domain of terramechanics and to the main simulators that have been developed for applications in this field.

Since the beginning of the 20th century, the usage of powered off-road vehicles has emerged in various domains, ranging from agriculture and construction to military operations and planetary exploration. Despite the increasing importance of cross-country vehicles, their development has not been treated within a comprehensive theoretical context until the late 1950s. The publication of Mieczysław Gregory Bekker's work, 'Theory of land locomotion' [18], 'Off-the-road locomotion' [19], and 'Introduction to terrain–vehicle systems' [20], led to the establishment of the principles of land locomotion mechanics, laying the foundation of 'Terramechanics'. The aim of this relatively new field of applied mechanics is to study the performance of a machine in relation to the terrain it is operating on and to provide theoretical fundamentals for the design, development, and evaluation of different off-road vehicles.

The continuous interest of space agencies around the world in the exploration of the Moon, Mars, and beyond have further stimulated advancements in terramechanics and its applications to extraterrestrial vehicles. However, despite creating increasingly complex terramechanics models, the lack of an adequate analytical tool for evaluating vehicle wheels with sufficient detail can lead to inaccurate predictions of performance when compared to real experimental data [21]. Therefore, creating or validating an existing tool that can provide sufficiently detailed simulations, which represents the main goal of this project, is crucial to the advancement of the application of terramechanics in the field of extraterrestrial vehicles and spatial exploration.

2.3 Survey of Terramechanics Simulators

Since the theoretical establishment of terramechanics, a myriad of simulators have been created, providing an extremely diverse range of features. This section provides an overview of some of the simulators that appear most often in the literature.

Patel and al. describe the development of a simulation tool named 'Rover mobility performance evaluation tool (RMPET)' [2]. RMPET is a simulator created specifically for exploring the performance of planetary rovers for the European Space Agency. It is based on Bekker's classical theory, introduced in the papers mentioned in the previous section, and according to its developers, it is capable of providing preliminary analysis for various configurations of a rover. RMPET includes a terrain generator tool, MarsGen, which can provide a two-dimensional representation of terrain similar to Mars' surface. When compared to other simulators, RMPET offers a very common and

user-friendly interface, as can be seen in Figure 3. However, no detailed results of simulations are shared by its developers, and a comparison with experimental data is not provided, therefore its accuracy is hard to contrast against other tools.

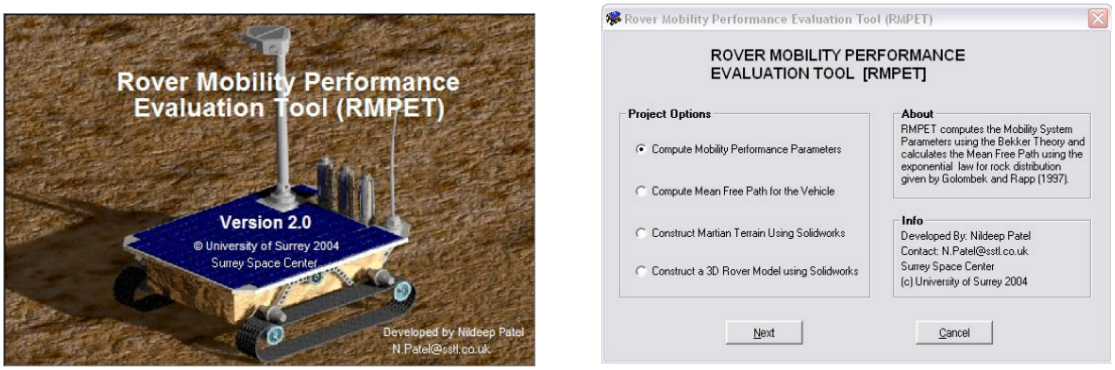


Figure 3: RMPET start-up screen and task selection screen (acquired from [2])

Another important tool is the 'Planetary Surface Rover Simulation Environment' or ROAMS [3]. Initially developed for NASA's 2009 Mars Science Laboratory mission, ROAMS contains models for various components of a robotic vehicle. Apart from the wheel-soil interactions, it includes models for mechanical subsystems, electrical subsystems, sensors, and on-board control software, allowing 'operator-in-the-loop' simulations, which involve a human controlling the rover, providing the opportunity to train operators in an environment similar to a real mission. Even though it provides complex models for many subsystems of a robotic vehicle, the terramechanics performance of ROAMS has not been validated against experimental data.

A different approach is taken by the developers of ORSIS [5]. The ORSIS simulator is not aimed specifically at rovers, but its vehicle representation method and wheel-soil model can accurately accommodate the simulation of a wheeled vehicle on extraterrestrial terrain. Similar to many of the simulators investigated, ORSIS is based on Bekker's theorem, and it also offers real-time capabilities, allowing the simulation of 'operator-in-the-loop' scenarios in a similar manner with ROAMS. Another common point between ORSIS and ROAMS is given by ORSIS's lack of testing against experimental data.

The VIPER simulator described in [7] is based on the open-source robotics simulator Gazebo but contains a series of proprietary plugins added by its developers. Interestingly, VIPER is focused on visually simulating the Lunar environment, motivated by the experience from Apollo astronauts that described difficulties with orientation and distance estimation [22]. The importance of visual similarity is also highlighted by a complex feature that is particular to VIPER, a shader that creates the wheel tracks left by the simulated rover in the powdery regolith that covers the surface of the Moon. Another interesting aspect of VIPER is its ability to model complex illumination effects from natural sources and rover lighting, creating accurate data that can be used for training and evaluation of computer vision algorithms used in rover navigation. VIPER's developers noted the difficulty in creating these visual features, therefore it is compelling to consider whether using a game engine that is better suited for visual features than Gazebo could have significantly reduced the cost of VIPER's development. According to its developers, VIPER is not perfectly suited for

simulating the wheel-soil interaction but offers 'relatively close' results to experimental data. The developers mention the possibility of increasing VIPER's fidelity by replacing its current wheel-soil interaction model with a discrete element method(DEM) model. However, using a DEM model is extremely expensive from a computational point of view [23], and the DEM model lacks validations for the lunar terrain.

Perhaps the most accurate simulator from a terramechanics perspective is described by Zhou, Iagnemma, and others in [6]. ARTEMIS is a simulation tool initially developed using validated mechanical models of NASA's Spirit, Opportunity, and Curiosity rovers, along with realistic soil properties taken from planetary missions led by NASA. Unlike the previously described tools, ARTEMIS doesn't provide as many features, but its main focus is placed on accurately simulating the wheel-soil interactions. Its terramechanics module is based on Bekker's theory and yields impressive results when compared to experimental data and even against real mission data. ARTEMIS's wheel-soil interaction model has been validated using single-wheel tests in deformable soil and by rover experiments at NASA's Jet Propulsion Laboratory. Furthermore, real data from the Opportunity rover mission were used to validate ARTEMIS. For example, Figure 4 shows wheel slip and rover pitch received through telemetry from the Opportunity rover, plotted against results obtained from an ARTEMIS simulation. One can observe the impressive accuracy that the simulator manages to achieve. However, it is important to note that there are still differences between the results of the simulated drive and the real drive. Interestingly, based on these differences, the authors manage to conclude that Opportunity's drive encountered two types of surfaces, and further analysis led to a very detailed description of the two soil types encountered.

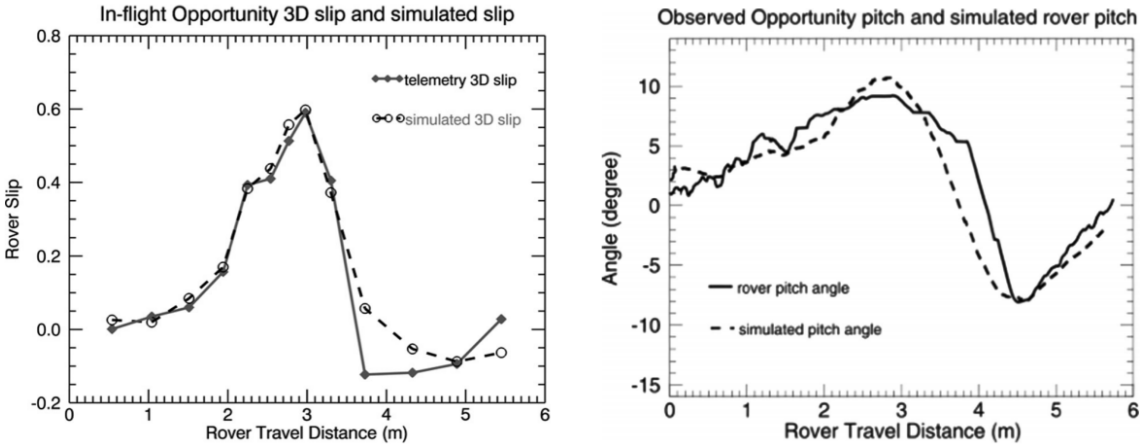


Figure 4: ARTEMIS results compared to data from the Opportunity rover (acquired from [6])

In summary, researchers have invested a considerable amount of effort into creating a variety of simulators with different capabilities. However, most of these tools are focused on a very specific subset of functionalities. Whilst the VIPER simulator seems to be the most general, it is based on Gazebo, which has a series of limitations specifically in the graphics processing domain. These limitations may suggest that Unreal Engine can represent a strong candidate for a general planetary rover simulator. Moreover, the related literature does not contain many studies of repurposing existing tools that are capable of handling the challenges that planetary rover simulations impose. Specifically, the usage of game engines for this simulation task has not been investigated, despite

the increasing complexity of this kind of software. A possible explanation for this research gap is given by the fact that the development of the aforementioned simulators started when game engines were not complex enough to be considered as a potential simulating tool.

2.4 Technologies Used

Considering the research gap introduced in the previous section, the choice of technologies for this project mainly consisted in deciding which game engine to investigate and how to validate it. At the time of writing this paper, the most advanced game engines available to the general public are Unity and Unreal Engine 4. Whilst both engines offer similar capabilities, the open-source approach taken by Epic Games, the developers of Unreal Engine 4, was the most important factor in deciding to use Epic's solution instead of Unity.

Unreal Engine 4 is a game engine created by Epic Games [24], first showcased in 1998. Although it was initially built solely for game development, it has evolved into a complete suite of development tools for working with real-time technology in a myriad of fields, including architectural and automotive visualization, the film industry, and more recently for simulation purposes. In terms of its physics model, Unreal is based on PhysX, an open-source real-time physics engine developed by Nvidia, which is used in many existing game engines. Furthermore, according to Epic Games, Unreal Engine 4 offers photorealistic rendering and dynamic physics and effects, which could potentially achieve the visual accuracy requirement of the VIPER simulator out-of-the-box. To test one of the requirements that VIPER needed, the ability to load large-scale maps, a DEM acquired from NASA's FROST dataset [25], specifically the Death Valley Location 1, was loaded into Unreal Engine 4. The engine was capable of handling the DEM without any modifications, as can be seen in Figure 5.

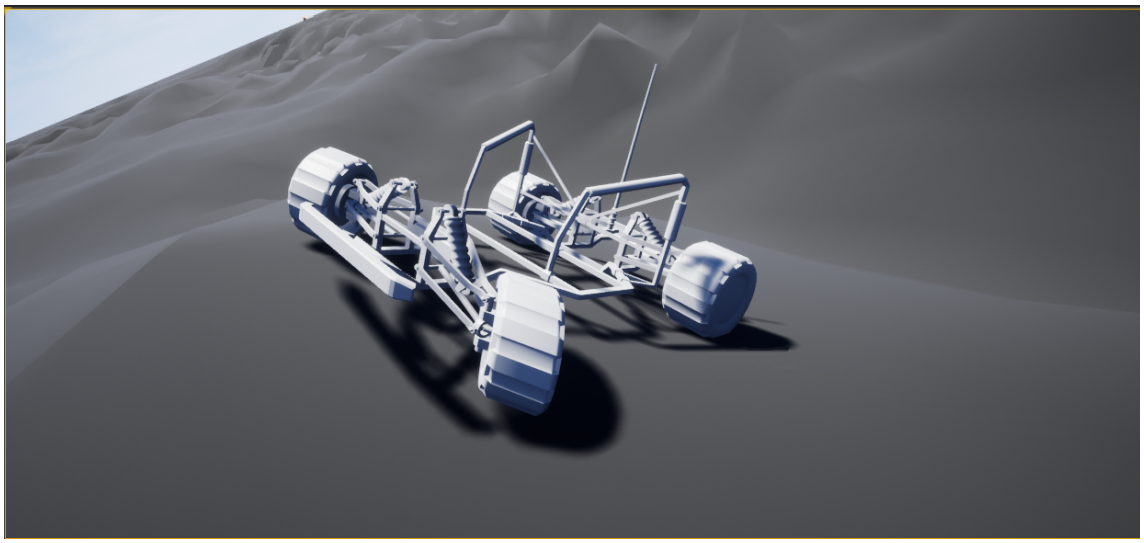


Figure 5: Vehicle model on FROST Death Valley Location 1 DEM in Unreal Engine 4

Having the confirmation that Unreal can satisfy at least one of the requirements of other simulators, the focus shifted on designing a methodology to perform a preliminary validation of its physics engine. Since obtaining experimental data was extremely challenging, it has been decided

to use another software tool to act as a validation mechanism. Based on discussions with individuals from NASA Ames Research Center that took place in several meetings in the initial phases of the project, the chosen approach was to create equivalent scenarios from a physics perspective in Unreal Engine 4 and another simulator and to compare the results obtained. However, most of the existing simulators are proprietary and not available at all to the open public. Therefore, Gazebo was chosen to act as the validation simulator, mainly because it is available as an open-source project and because it has been used as a basis for the VIPER simulator.

Gazebo [26] is an open-source robotics simulator that is actively developed and used. Since the beginning of its development in 2002, Gazebo has been used to simulate a large variety of robots in different environments [27–29]. Gazebo integrates multiple high-performance physics engines, such as Bullet, ODE, Simbody, and DART. Although it provides support for lighting, shadows, and textures, these features are not as complex as Unreal Engine's rendering pipeline.

Lastly, in order to process the data obtained from Unreal and Gazebo, the programming language Python was used, within the Jupyter Notebook framework. Analyzing the results was made possible through the usage of the visualization library Matplotlib, which is also based on Python. Python was chosen because it is fast, easy to use, and offers a variety of powerful visualization and data analysis libraries.

2.5 Summary

To sum up, this chapter introduced the reader to computer simulations in the field of terramechanics, presented a literature review of existing simulators, and identified the lack of research related to using game engines for high accuracy terramechanics simulations. In the last part of this chapter, the core technologies used to achieve this project were introduced, along with a motivation for the usage of each tool.

Chapter 3

Initial Evaluation

This chapter covers the description of the initial physics scenarios that were tested in Unreal and Gazebo. Each chosen scenario tests one of the main physics phenomena that would affect a rover as it crosses the surface of a planetary body. Therefore, gravity is initially tested, followed by how forces are modelled, which determines how the rover's wheels and body react to the force generated by its engine and transmission elements. The experiments that have been conducted are described in detail, and the results are discussed.

3.1 General Approach Used to Obtain the Data

For each experiment, there were three different data sources: the Unreal simulation, the Gazebo simulation, and the theoretical solution of each scenario. Hence, a consistent data format for storing the results constitutes a necessity for this project. The proposed solution represents the data as a series of observations, each containing the elapsed time in seconds from the beginning of the scenario, along with metrics such as the position, orientation, linear and angular velocity of each actor that was part of the experiment. The observations are stored in a CSV format, facilitating the comparison between the results and offering an efficient and flexible method of loading and visualising the data with Python. Figure 6 shows an example of this format, in the form of the raw results obtained by running one variation of the gravity drop test experiment in Gazebo.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Time	X Velocity	Y Velocity	Z Velocity	X Position	Y Position	Z Position	Roll	Yaw	Pitch	X Angular Velocity	Y Angular Velocity	Z Angular Velocity
2	0.001	0	0	-0.0098	0	0	9.99999	0	0	0	0	0	0
3	0.002	0	0	-0.0196	0	0	9.99997	0	0	0	0	0	0
4	0.003	0	0	-0.0294	0	0	9.99994	0	0	0	0	0	0
5	0.004	0	0	-0.0392	0	0	9.9999	0	0	0	0	0	0
6	0.005	0	0	-0.049	0	0	9.99985	0	0	0	0	0	0
7	0.006	0	0	-0.0588	0	0	9.99979	0	0	0	0	0	0
8	0.007	0	0	-0.0686	0	0	9.99973	0	0	0	0	0	0
9	0.008	0	0	-0.0784	0	0	9.99965	0	0	0	0	0	0
10	0.009	0	0	-0.0882	0	0	9.99956	0	0	0	0	0	0
11	0.01	0	0	-0.098	0	0	9.99946	0	0	0	0	0	0
12	0.011	0	0	-0.1078	0	0	9.99935	0	0	0	0	0	0
13	0.012	0	0	-0.1176	0	0	9.99924	0	0	0	0	0	0
14	0.013	0	0	-0.1274	0	0	9.99911	0	0	0	0	0	0
15	0.014	0	0	-0.1372	0	0	9.99897	0	0	0	0	0	0
16	0.015	0	0	-0.147	0	0	9.99882	0	0	0	0	0	0
17	0.016	0	0	-0.1568	0	0	9.99867	0	0	0	0	0	0
18	0.017	0	0	-0.1666	0	0	9.9985	0	0	0	0	0	0
19	0.018	0	0	-0.1764	0	0	9.99832	0	0	0	0	0	0
20	0.019	0	0	-0.1862	0	0	9.99814	0	0	0	0	0	0
21	0.02	0	0	-0.196	0	0	9.99794	0	0	0	0	0	0
22	0.021	0	0	-0.2058	0	0	9.99774	0	0	0	0	0	0
23	0.022	0	0	-0.2156	0	0	9.99752	0	0	0	0	0	0
24	0.023	0	0	-0.2254	0	0	9.9973	0	0	0	0	0	0
25	0.024	0	0	-0.2352	0	0	9.99706	0	0	0	0	0	0

Figure 6: Example of the general format of the data obtained from the experiments

It is important to specify that in order to obtain correct results an iterative approach was taken, where each time the behavior of the systems showed unexpected results, an analysis was conducted on the cause of the behavior, which then led to modifications of the simulated worlds. However, this process did not aim to synthetically create similar data that would give the appearance that the systems work correctly and show close results, but rather to ensure that equivalent scenarios were created in the systems, due to the complexity and the myriad of parameters available in Unreal and Gazebo. Furthermore, the approach taken for generating the results for each system makes it possible to run many variations of the same experiment relatively easy, by changing the values of different parameters, with no additional work being needed for processing and visualising the data. This aspect turned out to be extremely valuable throughout this project and could continue having a strong positive impact if this project is going to be continued. Note that the results presented in this report, along with the source code used to generate them, can be accessed by using the GitHub repository listed in Appendix B. The remainder of this section is going to present the development needed to create and visualize the results for each of the three data sources.

3.1.1 Obtaining the Unreal Results

The Unreal data was obtained by creating a project containing the four experiments, where each was represented in a separate map. Using different maps for every scenario provided the flexibility to modify each experiment without the risk of influencing the results of the others. In each map, the main actor was a cube, which was represented by an instance of a C++ class that, during its construction, created and attached a material to a cube shape. Subsequently, in every iteration of Unreal's physics engine, a series of calls to the PhysX API exposed by Unreal's wrapper was used to gather the value for each relevant metric. Figure 7 shows a snippet of code used to gather these values and save them into a string, which is then stored in an array of strings that is saved to a file in CSV format at the end of the experiment. Note that whilst this logging mechanism was used for each experiment, the forces required in experiments two, three, and four were applied by using the functions exposed by Unreal from the C++ code that represents each cube.

```
void AExperimentalCubeFour::AddObservation()
{
    // This function adds an Observation to the ObservationsArray.
    // Note that an Observations consists of the current time, the linear velocity, angular velocity, position and rotation of the cube and the force applied to the cube.

    FString Observation = "";
    FVector VelocityVector = CubeMesh->GetComponentVelocity();
    FVector PositionVector = GetActorLocation();
    FRotator RotationVector = GetActorRotation();
    FVector AngularVelocityVector = CubeMesh->GetPhysicsAngularVelocityInDegrees();

    // Note that there is a 0.5 seconds delay to account for any initialization that UE4 might perform and that might lead to unexpected behaviour.
    Observation += FString::SanitizeFloat(CrtTime - 0.5f) + ",";
    Observation += FString::SanitizeFloat(VelocityVector.X) + "," + FString::SanitizeFloat(VelocityVector.Y) + "," + FString::SanitizeFloat(VelocityVector.Z) + ",";
    Observation += FString::SanitizeFloat(PositionVector.X - DisplacementVector.X) + "," + FString::SanitizeFloat(PositionVector.Y - DisplacementVector.Y) + "," + FString::SanitizeFloat(PositionVector.Z - DisplacementVector.Z) + ",";
    Observation += FString::SanitizeFloat(RotationVector.Roll) + "," + FString::SanitizeFloat(RotationVector.Yaw) + "," + FString::SanitizeFloat(RotationVector.Pitch) + ",";
    Observation += FString::SanitizeFloat(AngularVelocityVector.X) + "," + FString::SanitizeFloat(AngularVelocityVector.Y) + "," + FString::SanitizeFloat(AngularVelocityVector.Z) + ",";
    Observation += FString::SanitizeFloat(ForceVector.X) + "," + FString::SanitizeFloat(ForceVector.Y) + "," + FString::SanitizeFloat(ForceVector.Z);

    ObservationsArray.Add(Observation);
}
```

Figure 7: Snippet of the code used for obtaining the Unreal data

3.1.2 Obtaining the Gazebo Results

In order to obtain the Gazebo results, a URDF model of a cube was created and used during each experiment. The first scenario used the built-in Gazebo tool to log the necessary metrics, whilst the results for the remaining three experiments were obtained by creating a C++ plugin similar to the one used in Unreal, which was then added into the *.world* file that Gazebo uses to represent its simulation worlds. Then, in each iteration of the physics engine, every relevant metric was stored in an array which was saved to an external file at the end of each experiment.

Note that a custom plugin was necessary since Gazebo only provides the logging functionality for a relatively small set of metrics, whilst the experiments in this project required more detailed observations. Furthermore, similar plugins were created to add the necessary forces for the last three experiments.

3.1.3 Obtaining the Theoretical Results

Obtaining the theoretical results was the least complex process from a programming point of view. It required implementing the equations described later in this chapter. Note that the sampling interval used between two successive observations was 0.001 seconds, the default duration of a Gazebo frame. Python was used, under the Jupyter Notebook framework, due to its flexibility and ease of use. Therefore, a main loop was created to represent the equivalent of a frame that has a fixed duration. The loop was run until the total experiment duration, which was particular to every experiment, elapsed. Each relevant metric was then stored in every iteration of the loop in a separate Python array, and then all metrics were combined into a single array that was then saved as a CSV file using the *pandas* [30] library from the Python environment.

3.1.4 Visualising the Results

Having obtained the results in a consistent format, the Python visualisation library, Matplotlib [31], was used to graph various measures of the data from the three sources on the same plot, creating the figures shown throughout this chapter. Note that due to the similarity of the data, markers are used in each figure to help the reader distinguish between the results from the three sources. It is worth specifying that the markers are plotted once every 100 iterations, implying that the total number of observations used to create the plots is significantly larger than the number of markers shown in each figure.

Moreover, in preparing the results for graphing, all observations were transformed into the international system of units equivalent. This transformation was required specifically for the Unreal results, which by default uses units of measurement smaller by two orders of magnitude. That is, the measure for distance in Unreal is a centimeter, compared to the international system of units equivalent for distance which is the meter. This consideration affects a variety of measures such as linear velocity, linear acceleration, and forces.

3.2 Gravity Drop Test

The first experiment aimed to test how Unreal and Gazebo model the effect of gravity, since it represents one of the forces that would continuously affect a planetary rover, and therefore differences in the modelling of gravity could quickly add up and create significant discrepancies in simulation results. Analysing the behaviour of gravity was performed by using an object that was subjected to a free fall. Even though the object's shape and mass do not affect its free-falling behaviour, in order to be consistent with the following experiments, a cube with a mass of 1 kilogram and dimensions of $1 \times 1 \times 1$ meters was used, as illustrated in Figure 8.

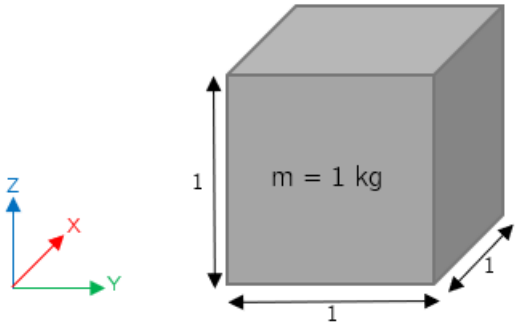


Figure 8: Cube used in the gravity drop test experiment

The cube was dropped from a height of 10 meters above the ground level, using Earth's gravitational acceleration. Furthermore, it was considered that the object falls in a vacuum, therefore all effects of friction were ignored. For clarity, a visual illustration of the entire experiment is shown below.

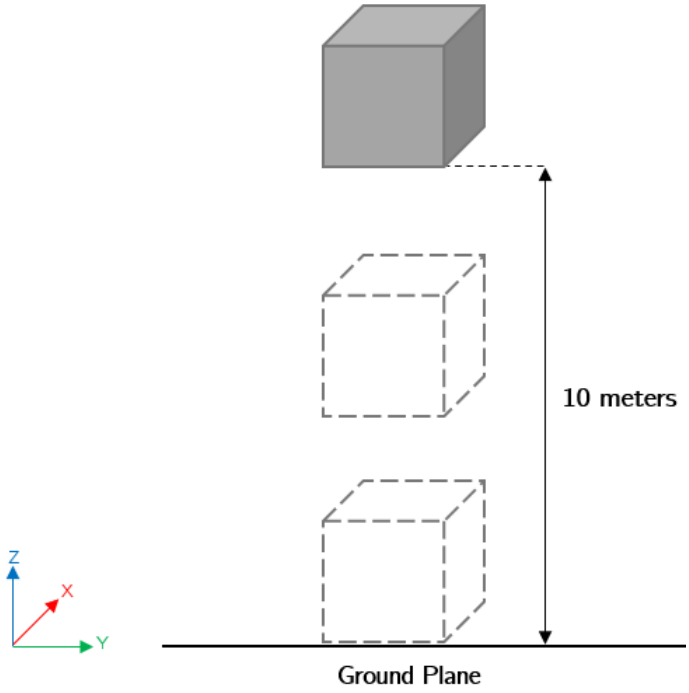


Figure 9: Illustration of the gravity drop test experiment

Under the previously described conditions, the movement of the cube is described by Newton's second law of motion. Hence, taking into consideration the coordinate system displayed in Figure 9, one can determine the velocity and location of the cube and capture the behaviour of the entire system by using the following set of equations:

Let: V = Velocity of the cube on the Z-axis in m/s

X = Displacement of the cube on the Z-axis in m

g = Earth's gravitational acceleration = -9.81 m/s^2

t = Time in seconds elapsed since the beginning of the experiment

Then, since the only force affecting the cube is gravity:

$$V = gt$$
$$X = \frac{gt^2}{2}$$

Therefore, one can obtain the expected minimum velocity of the cube by computing the time when the cube touches the ground, t_{stop}, and substituting the result into the formula for velocity. Since the cube has $1 \times 1 \times 1$ meter dimensions and uniform density, it is going to touch the ground when the Z-axis position of its center of mass equals 0.5 meters. Because the initial position of the cube's center of mass is 10 meters above the ground, a displacement of -9.5 meters is necessary for the cube to touch the ground. Solving for t_{stop} and replacing its value into the equation for V, one gets t_{stop} = 1.391, and the minimum velocity V_{min} = -13.64 m/s. Hence, the linear velocity of the cube is expected to decrease linearly from its initial value of 0 m/s to -13.62 m/s, and then return to 0 m/s immediately after 1.391 seconds. On the other hand, the cube's displacement is expected to decrease quadratically, causing the cube's position on the-Z axis to decrease quadratically. Note that since this experiment involves the cube falling without rotating, the Z-axis position and Z-axis linear velocity are the only two quantities of interest.

These theoretical considerations are closely followed by the results obtained from both Gazebo and Unreal. Figure 10 shows the cube's linear velocity on the Z-axis as a function of time from the Gazebo and Unreal simulations, along with the theoretical solution, plotted on the same figure. Note that each circle represents a marker that is used once in every 100 observations to help the reader distinguish between the 3 plots, which due to their similarity would be indistinguishable from each other if the markers were not used. Looking at the results, one can notice that the linear velocity decreases linearly from 0 m/s to -13.63 m/s after 1.39 seconds in Unreal, and respectively from 0 to -13.64 m/s after 1.392 seconds in Gazebo, being extremely close to the values obtained in the above paragraph.

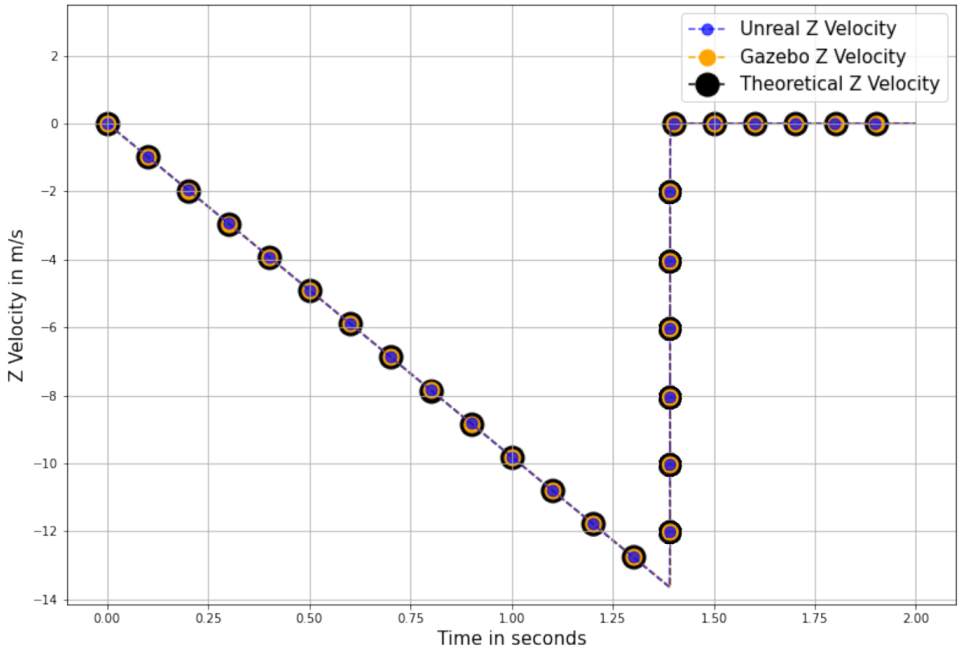


Figure 10: Z-linear velocity over time for the gravity drop test

Similarly, Figure 11 shows the Z-position of the cube over time from the three data sources. Once again, the two simulators show very accurate results, with the Z-position decreasing quadratically

from the initial value of 10 meters to the final value of 0.5 meters. Recall that the final value is expected to be 0.5 meters, due to the explanation previously given in the current section.

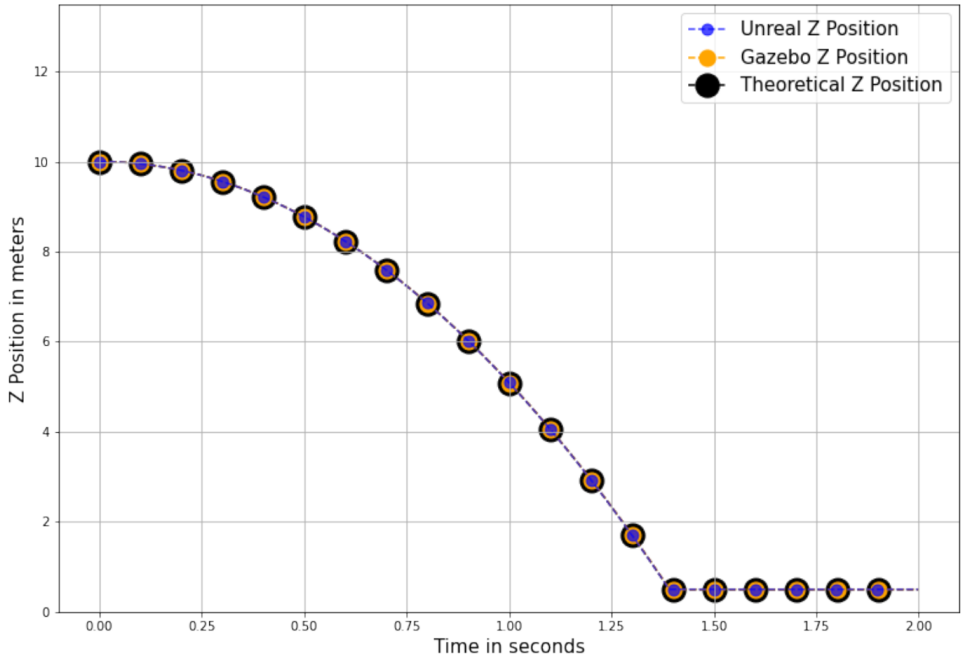


Figure 11: Z-linear position over time for the gravity drop test

It is worth noting that a close examination of Figures 10 and 11 shows that the data is not identical since the circles are not exactly centered. Since the number of observations is the same for all three sources, and the time between each observation is identical, one would expect the circles to be exactly centered if the data matched perfectly. Therefore, the lack of perfect centering of the markers implies that there are slight differences between the results. In order to check if the differences are caused by floating-point errors or by subtle integration errors which could eventually build up over time, the difference between the Unreal and the theoretical results, and the difference between Gazebo and the theoretical results have been plotted in Figure 12.

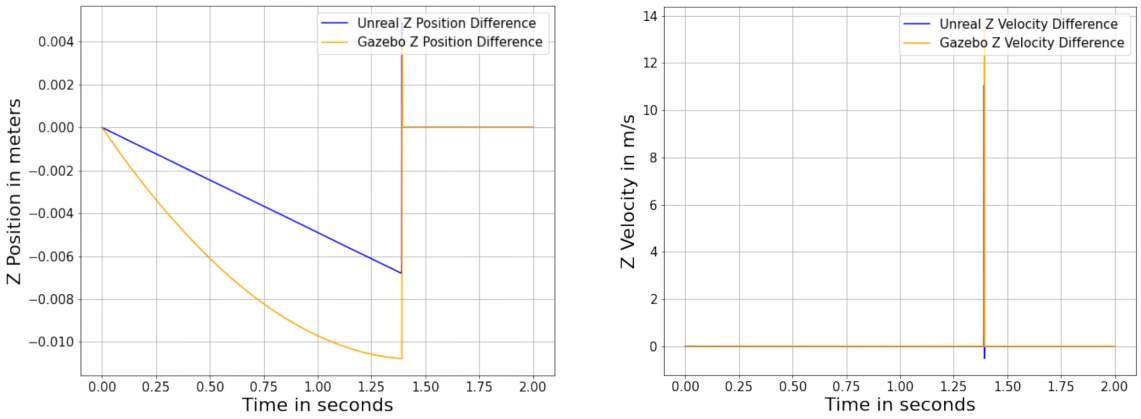


Figure 12: Plots showing the difference between the simulated results and the theoretical solutions for the gravity drop test. From left to right: Z-axis position error over time, Z-linear velocity error over time

Surprisingly, the Z-position results show that both engines tend to be biased, with the Unreal bias being always negative and decreasing linearly, whilst in Gazebo the bias starts from a negative

value and seems to decrease quadratically until the end of the experiment. In the case of the Z-velocity, both engines show a smaller error, up until the point when the cube touches the ground when Unreal seems to have several frames in which the velocity is extremely noisy. By looking solely at the results from Figure 12, one could argue that the errors seem to show underlying integration errors in both engines and could expect that the errors would increase with experiment duration. Based on this intuition, the experiment was recreated multiple times, with varying initial heights, and the same process was followed, plotting the differences between Unreal, Gazebo, and the theoretical results. The interested reader can refer to the GitHub repository listed in Appendix B, which contains both the source code and the results for each variation of this scenario. This analysis showed that the errors are probably caused by random noise and floating-point errors since varying the initial height of the cube did not seem to have a consistent effect on the size of the errors.

To summarise, based on the results described throughout this section, one can conclude that although errors are present, both Unreal and Gazebo accurately simulate gravity, independent of the total experiment duration, implying that there are no integration errors that could potentially build up over time to cause a significant inaccuracy.

3.3 Impulse Applied to a Cube

The next tested scenario contained the same cube described in the previous section that was placed at rest on flat ground. A large force in the positive Y-direction was applied to the cube's center of mass for a small amount of time to analyse how the two engines model impulses. A complete illustration of this scenario is shown in Figure 13.

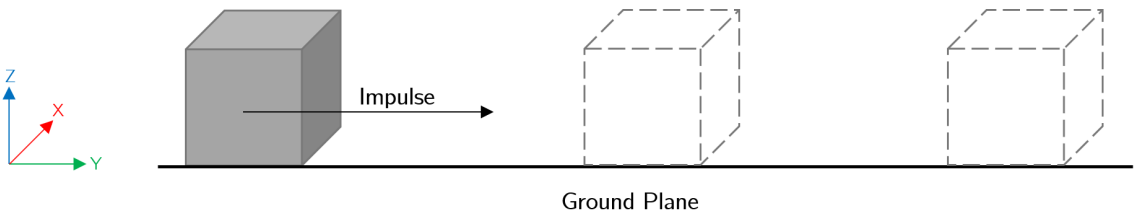


Figure 13: Illustration of the impulse experiment

3.3.1 Neglecting Friction

Initially, this experiment ignored the effects of friction, to ensure that both simulators apply the correct total amount of force. In this case, the behaviour of the system can easily be modeled by introducing the following notation and equations deriving from Newton's second law of motion and impulse definition:

Let: J = Impulse in Ns

F = The force applied to the cube in N

m = The mass of the cube in kg

t = Time in seconds elapsed since the beginning of the experiment

t_1 = The amount of time for which the force was applied in seconds

p_1 = Linear momentum after the force was applied in kgm/s

V = Velocity of the cube on the Y-axis in m/s

X = Displacement of the cube on the Y-axis in m

Then:

J=$\int_0^{t_1}$ F dt

From Newton's second law of motion, force is related to linear momentum p by:

$$F = \frac{\Delta p}{\Delta t}$$

However, since the cube was initially at rest and the initial time was 0, one gets:

$$F = \frac{p_1}{t_1} \Leftrightarrow$$

$$p_1 = Ft_1$$

From the definition of momentum, $p_1 = mV$

Because there is no friction, velocity is going to be constant and given by the formula:

$$V = \frac{p_1}{m} \Leftrightarrow V = \frac{Ft_1}{m}$$

And therefore:

$$X = Vt$$

Following the derivation presented above, one would expect the cube's velocity to increase nearly instantaneously from 0 to $\frac{Ft_1}{m}$ and then remain constant over the entire duration of the experiment. Also, the displacement should increase linearly over time, with an increase rate proportional to the velocity of the cube. It is worth noting that in this experiment, the value chosen for t_1 was 0.001 seconds, which is the default duration of an iteration of the physics engine in Gazebo. Furthermore, it can be observed that as opposed to the previous experiment, the mass of the cube starts playing an important role in its behaviour.

This experiment has been performed numerous times, covering a wide variety of forces and masses for the cube. The observed behaviour was nearly always the same, hence only the results from one set of parameters will be presented. Therefore, Figure 14 shows the results when using a cube of mass $m = 1kg$, a force $F = 1000N$, and a period of applying the force $t_1 = 0.001s$. Note that the force has a relatively large value because the time that it is applied for is extremely small. It can be seen that the expected behaviour is achieved by both simulators, with the velocity being constant and with a value of $1m/s = \frac{1000N*0.001s}{1kg}$, given from the formula $V = \frac{Ft_1}{m}$. Furthermore, the Y-position increases linearly with time. Similar to the previous experiment, one can notice that the data is not identical, and by using the same method of analysing the differences as presented in the previous section, it has been concluded that the differences represent noise, since once again no correlation was observed between varying the experiment parameters and the errors obtained.

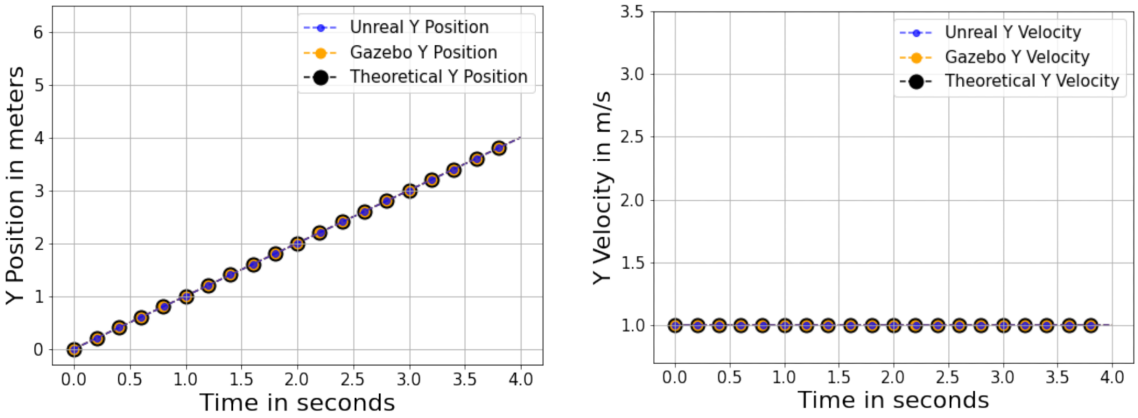


Figure 14: Plots showing the results from the impulse experiment when friction is ignored. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.3.2 Including friction

Next, the same experiment was repeated and friction was taken into consideration. Interestingly, even though both Unreal and Gazebo model friction based on a Coulomb friction model, there are different optimizations that the two engines perform which can lead to significant differences. Gazebo is approximating the 'friction cone' by a 'friction pyramid' [32], whilst Unreal uses an optimisation technique created by Nvidia and integrated into the PhysX engine that is not rigorously described by its developers, but one which yielded accurate results in all the experiments that this project examined.

Modelling friction increases the complexity of the theoretical solution of this scenario. However, one can compute the velocity of the cube as a function of time by adding four elements to the previously introduced notation:

Let: μ_k = The coefficient of kinetic friction
F_f = The kinetic friction force in N
v_0 = The velocity of the cube immediately after applying the force F in m/s
g = Earth's gravitational acceleration = -9.81 m/s^2

Recall that the cube was initially at rest, and a force F was applied to it for a period of time t_1. Since t_1 is extremely small, one can neglect the effect of friction for the duration given by t_1, and consider that the velocity of the cube immediately after applying the force F is given by the formula previously determined, $v_0 = \frac{Ft_1}{m}$. Since the force is applied exactly once, only friction directly affects the cube and its movement after t_1 seconds. In order to help the reader follow through the explanation, an illustration of all the forces that are affecting the cube after applying the force F is displayed in Figure 15, where G represents gravity, N is the normal force, F_f is the friction force, and v_0 is the velocity of the cube immediately after F was applied.

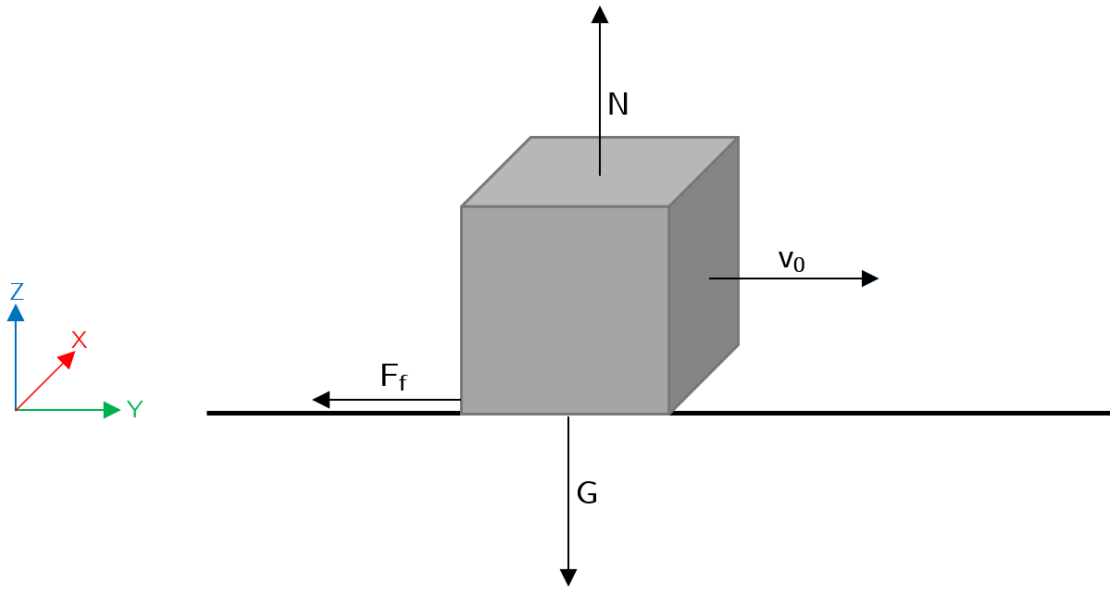


Figure 15: Illustration of the forces affecting the cube after the impulse has been applied

From Newton's second law, where a is the acceleration of the cube, and from the definition of kinetic friction, F_f can be expressed as:

$$F_f = ma$$

$$F_f = N\mu_k \Leftrightarrow F_f = -mg\mu_k$$

Equating the 2 expressions for F_f and using the definition of acceleration yields:

$$m\frac{\Delta V}{\Delta t} = -mg\mu_k \Leftrightarrow$$

$$\frac{V - v_0}{t - 0} = -g\mu_k \Leftrightarrow$$

$$V(t) = v_0 - g\mu_k t$$

Then:

$$X(t) = \int_0^t V(t)\, dt \Leftrightarrow$$

$$X(t) = \int_0^t (v_0 - g\mu_k t)\, dt$$

The last formula obtained makes it possible to compute the time required for the cube to come to a complete stop under the effect of friction by setting $V(t)$ to 0 and solving for t. In this manner, the stopping time, t_{stop}, is given by:

$$t_{stop} = \frac{v_0}{g}\mu_k$$

Similarly, the stopping distance can be computed by using the stopping time:

$$d_{stop} = \frac{1}{2}v_0 t_{stop} \Leftrightarrow$$

$$d_{stop} = \frac{1}{2}\frac{{v_0}^2}{g\mu_k}$$

Having obtained a formula for the velocity of the cube as a function of time, one can conclude that in this experiment, the Y-axis velocity of the cube is expected to drop linearly from its initial value, $v_0 = \frac{Ft_1}{m}$ to 0, in t_{stop} seconds, and that the cube should travel a total of d_{stop} meters during the entire experiment.

Similar to the previous subsection, only the results of one variation of parameters will be discussed. Figure 16 shows the Y-axis position and Y-linear velocity over time for a force $F = 10000N$ applied for $t_1 = 0.001s$, with a friction coefficient of $\mu = 1$, and the cube mass of $m = 1kg$. Unlike the previous results, the differences between Unreal and Gazebo seem to be significantly larger. However, in this experiment, the Unreal frame duration was significantly larger due to the increased computational cost that friction introduces. To be more specific, in this experiment, the Unreal observations are logged every 0.016 seconds, whilst Gazebo and theoretical results are based on a 0.001 sampling time. This causes the results showed in Figure 16 to seem significantly different because the markers are added at different time stamps. Therefore, if one corrects for this fact by computing the total error as a sum of the differences between the Unreal and theoretical results every 0.016 seconds, the total error is similar to the error computed between the Gazebo and the theoretical results every 0.016 seconds. Furthermore, both Unreal and Gazebo results respect the theoretical considerations previously determined: the cube in Unreal starts from the initial velocity of 9.89 m/s, and travels a total of 5.057 meters and stops after 1.118 seconds, whilst in Gazebo it starts from 9.99 m/s and travels 5.082 meters in 1.115 seconds. By introducing the parameters of this experiment in the formulas determined above, one can obtain the expected initial velocity of the cube $v_0 = \frac{10000*0.001}{1} = 10$, the expected stopping time $t_{stop} = \frac{10}{9.81} * 1 = 1.019$, and travel a total of $d_{stop} = 5.096$ meters. To conclude, both simulators showed good results, with Gazebo being more accurate in this particular experiment. However, the difference between the results was influenced by the significantly larger frame duration in Unreal.

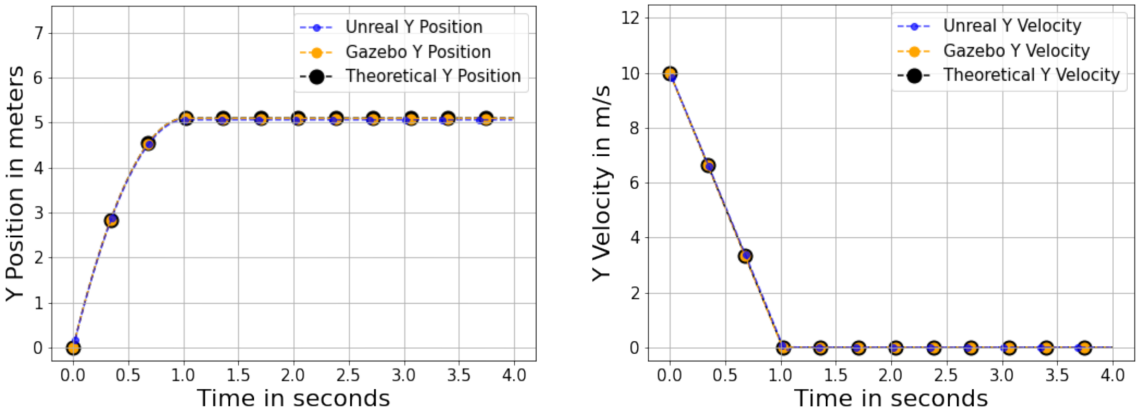


Figure 16: Plots showing the results from the impulse experiment when friction is taken into consideration. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.4 Constant Force Applied to a Cube

In this experiment, a constant force of various magnitudes was applied to the cube, in the positive Y-direction. The motivation of this scenario was to analyse how the two engines integrate forces over a longer period, since in the case of simulating a planetary rover, forces are going to be applied over long periods of times, not just in very short bursts similar to the scenario tested in Section 3.3. Compared to the previous experiment, friction was always modelled and the duration for which the force was applied was on the order of seconds. Since this experiment and the previous one are very related from a physics perspective, the reader can refer to Figures 13 and 14 for a visual illustration of this scenario.

In order to characterize the system from a theoretical perspective, one could split this experiment into two phases, governed by t_{force}, the duration for which the force is applied: the first phase starts from time 0 and lasts till t_{force} seconds, and the second ranges from t_{force} seconds until the end of the experiment. In the first phase, the behaviour of the cube can be determined by using a rationale similar to the one described in Section 3.3.2. For clarity, Figure 17 shows an example of the force applied to the cube over time, when choosing the value of the force to be 20N, t_{force} to be 5 seconds, and the total experiment duration 10 seconds.

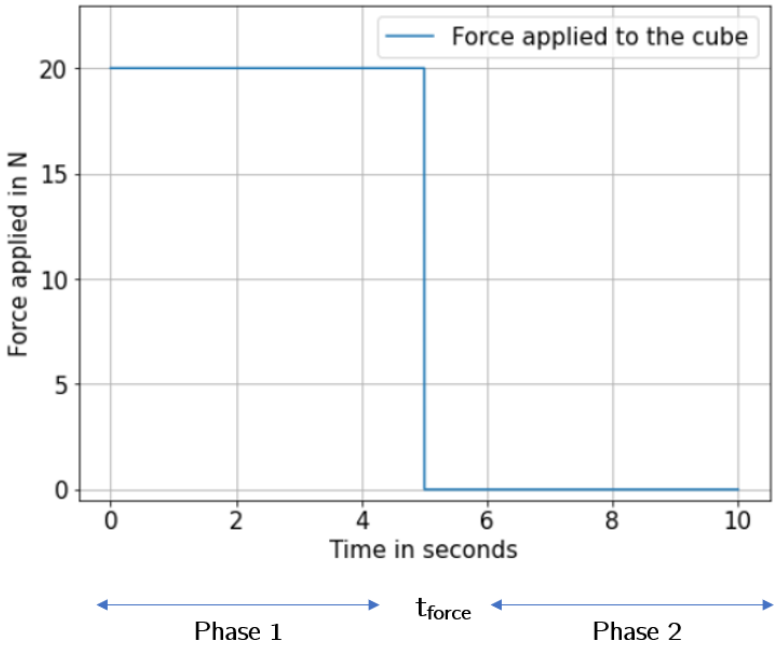


Figure 17: Constant force applied over time in one realisation of the third experiment

Keeping the same notation introduced throughout the previous section, one can compute the velocity of the cube as a function of time in the first phase as:

$$F - F_f = \frac{\Delta p}{\Delta t} \Leftrightarrow$$
$$F - mg\mu_k = \frac{\Delta p}{\Delta t}$$

Following the same rationale as in Section 3.2, one gets the velocity in phase one as:

$$V(t) = \frac{(F - mg\mu_k)\, t}{m}$$

Then, the distance travelled by the cube as a function of time in phase one is given by:

$$X(t) = \int_0^t V(t)\, dt \Leftrightarrow$$
$$X(t) = \int_0^t \frac{(F - mg\mu_k)\, t}{m}\, dt$$

The second phase is identical to the scenario described in Section 3.2.2, with the initial velocity of the cube being equal to the velocity at time t_{force}, which can be computed using the formula obtained above. Therefore, the initial velocity in phase two is $v_0 = \frac{(F - mg\mu_k)t_{force}}{m}$.

Combining the results from the two phases, one can reason that the velocity of the cube should increase linearly from the beginning of the experiment until t_{force} seconds and then start decreasing linearly for the next t_{stop} seconds. Moreover, the Y-position of the cube should increase for the duration of the entire experiment, with an increase rate proportional to the velocity. That is, it should increase more rapidly when the experiment time is closer to t_{force} seconds since due to the previous observation the velocity reaches its maximum at t_{force} seconds. The total distance traveled by the cube and the total time in which the cube is moving can also be easily computed by combining the two phases:

Let: X_{total} = The total distance travelled by the cube in meters

t_{total} = The total time for which the cube is moving in seconds

Then:

$$X_{total} = \int_0^{t_{force}} \frac{(F - mg\mu_k)\, t}{m}\, dt + \frac{1}{2}\frac{{v_0}^2}{g\mu_k} \Leftrightarrow$$
$$X_{total} = \int_0^{t_{force}} \frac{(F - mg\mu_k)\, t}{m}\, dt + \frac{\left((F - mg\mu_k)\, t_{force}\right)^2}{2m^2 g\mu_k}$$
$$t_{total} = t_{force} + \frac{v_0}{g\mu_k} \Leftrightarrow$$
$$t_{total} = t_{force} + \frac{(F - mg\mu_k)\, t_{force}}{mg\mu_k}$$

Figure 18 displays the results obtained by applying a force of 20N constantly over a period of 5 seconds, with a friction coefficient equal to 1, and a cube of 1kg. Once again, the results seem to differ due to the different sampling times for the markers, however, the actual data has extremely close values. The simulated results can be validated by using the formulas determined above and using the aforementioned values for the experiments' parameters. Therefore, the expected total distance travelled by the cube is given by $X_{total} = \int_0^5 (20 - 9.81)\, t\ dt + \frac{((20-9.81)*5)^2}{2*9.81} = 127.375+132.308 = 259.683$ meters, whilst the total time, $t_{total} = 5+\frac{(20-9.81)5}{9.81} = 5+5.193 = 10.193$ seconds. Both simulators respect all the theoretical aspects presented so far, with the cube in Unreal travelling a total of 259.716 meters in 10.202 seconds, and in Gazebo the cube moving for 260.204 meters in 10.231 seconds. Based on Figure 18 and a variety of variations of this experiment that showed similar behaviours, it can be concluded that both engines simulate applying a constant force accurately, with Unreal having a slight advantage in the overall accuracy, even though the frame duration was larger in Unreal and therefore the physics engine performed fewer iterations overall than in Gazebo.

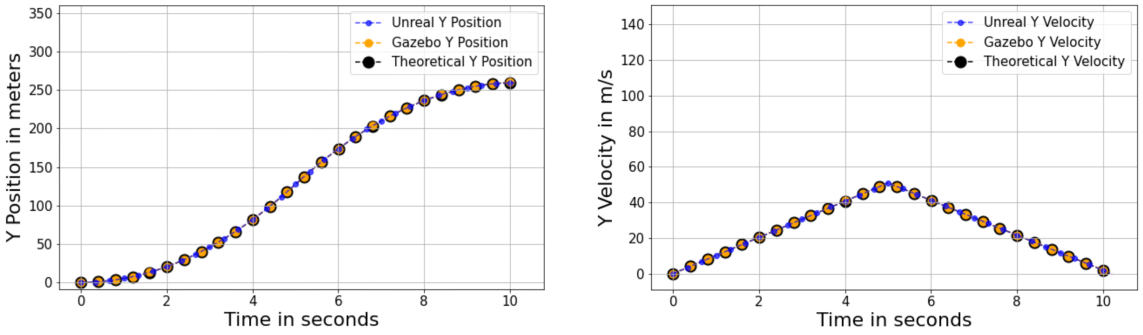


Figure 18: Plots showing the results from a variation of the constant force experiment. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.5 Force Patterns Applied to a Cube

The final experiment presented in this chapter is the most complex one. It involved using different force patterns that were applied to the same cube used so far. Similarly, the cube was placed on flat ground, and forces that varied with time were applied in its Y-positive direction, taking into consideration the effects of friction. This experiment had the goal of analysing how the two systems behave when the used forces vary over a period of time. Several force patterns were trialed, each trying to mimic the behaviour of a rover engine over time.

From a theoretical perspective, the behaviour of the cube for each of the patterns used can be determined by using the same set of equations and using a similar rationale to the previously described experiments. This section contains a set of assumptions based on the equations and rationale introduced so far in this chapter and analyses how well each of the two simulators manages to follow the theoretical assumptions.

In this experiment, the difference between the duration of the physics engine iteration of each simulator becomes significantly more important, as will be shown in the presentation of the results. Because the experiments were ultimately based on a fixed time amount, and Unreal frame duration is larger, this results in a smaller number of iterations that Unreal's physics engine will perform, which can lead to a significant decrease in the accuracy of the approximations resulting from the force integration performed by the engine. Two different approaches were taken in the effort to mitigate the effects of this dissimilarity. Initially, the experiments were not based on time, but rather on the number of iterations the physics engines in both Unreal and Gazebo performed. Therefore, the forces were not applied for a total of t seconds, but rather for a total of x frames. However, this approach was quickly proven to be flawed, since due to the difference in frame duration, the total amount of force applied to the cube in Unreal was significantly larger, causing the behaviour of the cube to be completely different than the Gazebo version. The second approach was to base the experiments entirely on time and to use the physics sub-stepping functionality available in Unreal, which lets the user perform multiple iterations of the physics engine in one single frame, in order to reduce the difference between the frame duration in Unreal and Gazebo. This approach was used to obtain the results presented throughout this section.

3.5.1 Pattern One

The first pattern consists of a linearly increasing force, starting at 0N and increasing to 10N, where it remains for one second, and then decreases at the same rate back to 0. This pattern was meant to reproduce an engine increasing its force output during the first 2 seconds, staying at maximum power, and then slowly going back down. Figure 19 shows an illustration of the pattern used in this case.

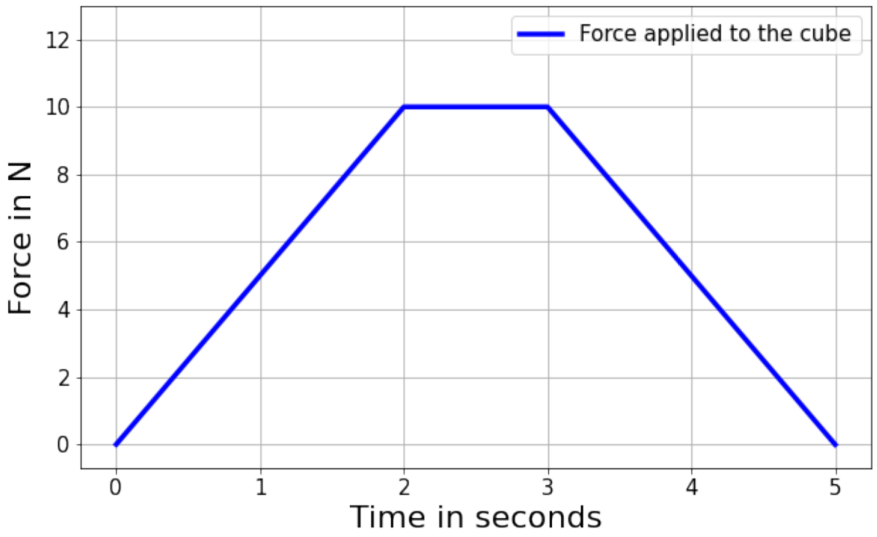


Figure 19: Force applied over time in pattern one

The results from pattern one were the first that showed a slightly more substantial discrepancy between the two simulators, with Unreal results being smaller than one would expect based on theoretical considerations, and the Gazebo results matching the expected total distance and maximum velocity gained from using the formulas introduced throughout this chapter. After eliminating a series of hypotheses related to the cause of these differences, it has been decided that a possible cause of Unreal's inaccuracy is the larger frame duration. Due to larger sampling intervals, Unreal can end up applying less force than expected, leading to the behaviour observed in Figure 20. Additionally, the reader should take into consideration that the plots might cause the difference between the final position in Unreal and Gazebo to seem higher than its actual value of 0.0106 meters.

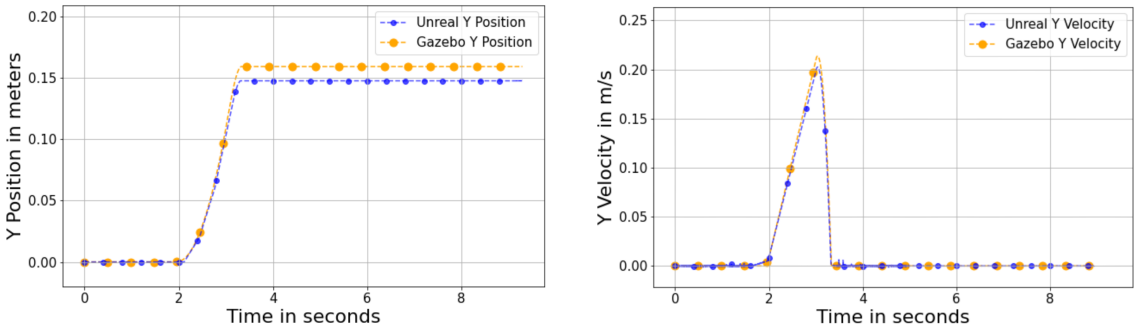


Figure 20: Plots showing the results from applying force pattern one. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.5.2 Pattern Two

Pattern number two is similar to the previous case, however, the force increases and decreases at a much larger rate, and remains constant at its maximum value of 10N for a longer period. The figure below shows the force applied to the cube as a function of time for the second pattern.

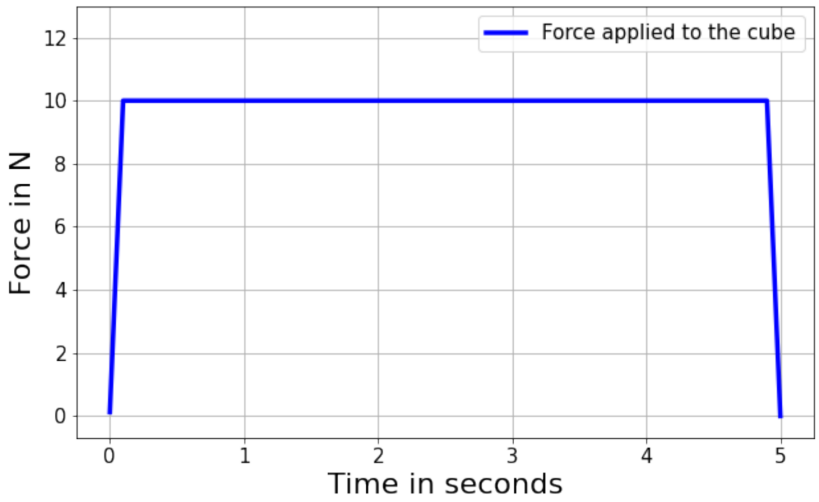


Figure 21: Force applied over time in pattern two

Since the force increases rapidly and then remains constant for the vast majority of this experiment, one would expect the results from Unreal and Gazebo to be extremely close, since the difference in frame duration should not have a significant impact, and the whole behaviour of the cube in this scenario should be similar to the constant force experiment described in Section 3.4. Figure 22 confirms these assumptions, showing that the results of the two simulators are once again nearly perfectly matched, with the difference most likely being caused by floating-point errors. Alike the constant force experiment, the Y-position of the cube increases quadratically over the period when the force is applied and then remains constant immediately after, whilst the velocity increases linearly for 5 seconds and then experiences a steep decrease. However, unlike the results presented in Figure 18, the Y-velocity does not seem to experience a linear decrease, since the maximum value of the velocity is considerably smaller and the friction coefficient stops the cube nearly instantly.

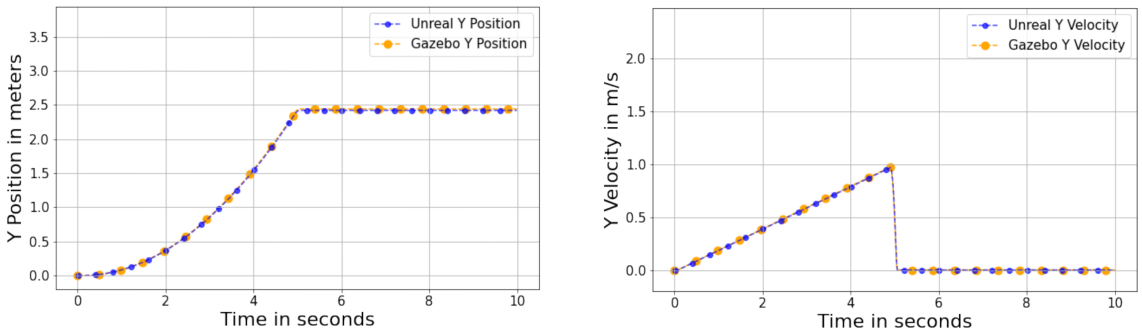


Figure 22: Plots showing the results from applying force pattern two. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.5.3 Pattern Three

In this scenario, the applied force was based on sequential repetitions of a scaled version of pattern one. Therefore, the force was significantly more varied than in the previous experiments, as shown in Figure 23.

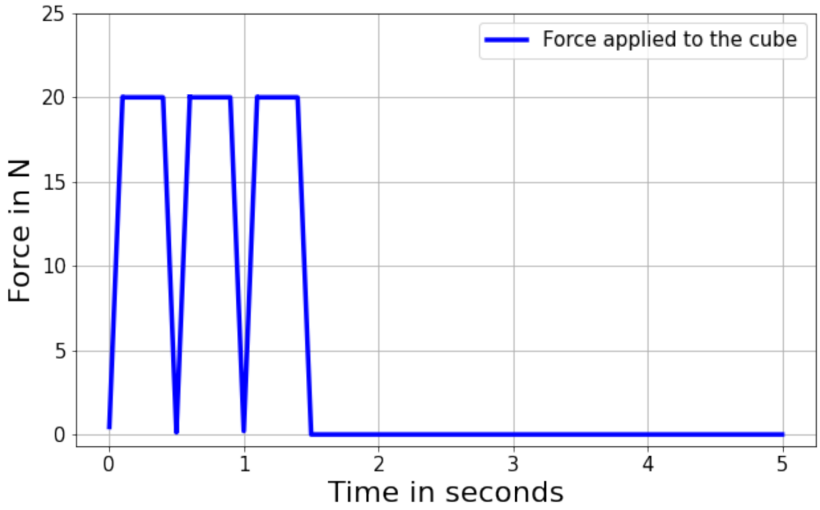


Figure 23: Force applied over time in pattern three

Based on the previous behaviour observed for pattern one and the shape of the current pattern, one can expect the results from this experiment to show significant differences between the simulators. Since the force fluctuates more violently and remains constant for a significantly lower duration, the effects of the frame duration difference should be more visible, leading to a poor accuracy from Unreal. Figure 24 exhibits this exact behaviour, where Unreal once again applies a smaller amount of force, translating in a smaller velocity over time, and therefore the distance travelled by the cube is significantly lower. Even though the results are very different, it is important to realise that they do not imply that Unreal is not correctly or accurately modelling the physics concepts that are tested, but rather that the pattern is not correctly recreated in Unreal, due to its larger frame duration. Furthermore, these results can serve in enforcing that Unreal is correctly modelling friction and force since its behaviour follows the overall trend of the Gazebo results, but with a smaller magnitude of the force, and implicitly of the velocity and displacement. This is the main reason for which this experiment was presented and considered relevant, despite the significant disagreement between the two systems.

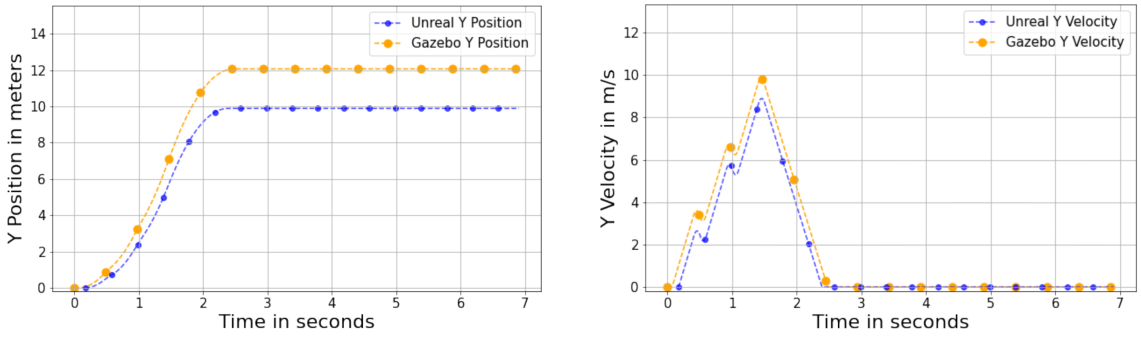


Figure 24: Plots showing the results from applying force pattern three. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.5.4 Pattern Four

The last pattern presented consists of a sine wave with an amplitude of 20N, and a frequency of 1. It is based on the same idea of applying the same sub-pattern in sequential repetitions over a fixed duration. A visual illustration of this pattern is shown in Figure 25.

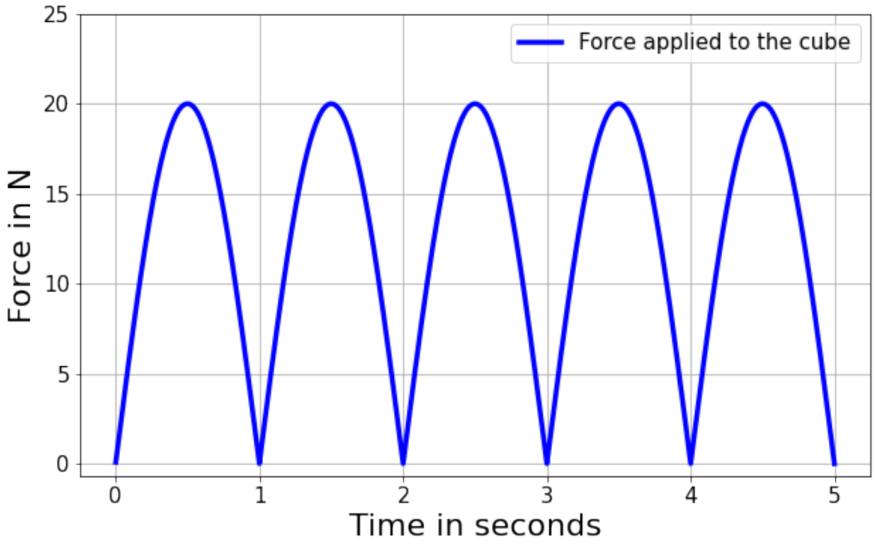


Figure 25: Force applied over time in pattern four

Since the last pattern is represented by a smooth function that does not have a large rate of change over time, the frame duration difference should not be significant in the resulting behaviour of the cube, and one would expect the two simulators to perform with a similar level of accuracy. Figure 26 confirms this expectation, with both Unreal and Gazebo results overlapping nearly perfectly. To reiterate, the markers shown on the plots are not perfectly overlapping due to the difference in frame duration and the number of observations taken before each marker is plotted. This might cause the reader to assume that the results are dissimilar, however, by analysing the raw data, one can notice that the differences are extremely small, for example, the maximum cube velocity in Unreal was 16.199 m/s, and in Gazebo 16.230 m/s.

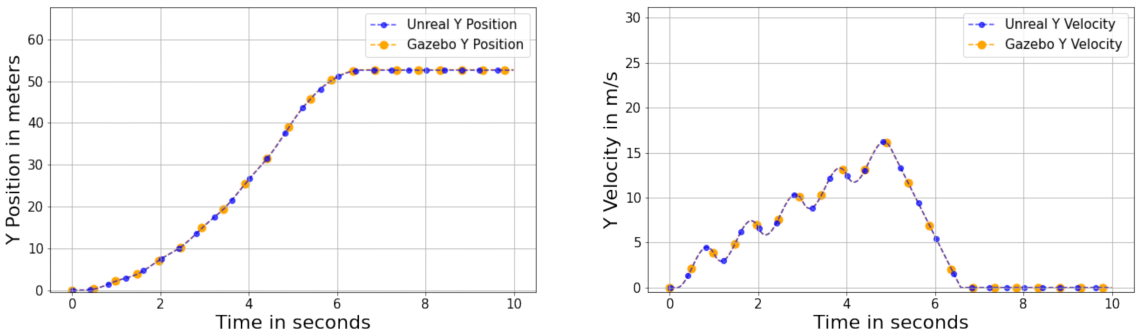


Figure 26: Plots showing the results from applying force pattern four. From left to right: Y-axis position over time, Y-axis linear velocity over time

3.6 Summary

To summarise, Chapter 3 provided the reader with a theoretical analysis of four of the scenarios tested throughout this project, offering enough information to allow the equivalent recreation of each experiment in any physics simulator. Firstly, the gravity model of Unreal and Gazebo was tested by having a cube free-fall from different heights. Secondly, the force integration of the two engines was analysed in a variety of ways, starting from applying an impulse of varying magnitudes, to different constant forces for a longer period of time, and finally to different force patterns.

Based on comparing and contrasting the results obtained from running the aforementioned experiments with a large variety of parameters, it can be concluded that the accuracy obtained by Unreal is similar to that which Gazebo currently offers and is a good match to the theoretical considerations and results.

Chapter 4

Adapting an Existing Physics Benchmark

This chapter contains the description of the work undertaken to adapt an existing benchmark that has been previously used by robotics researchers to compare several physics engines. It includes the theoretical description of the benchmark, along with a discussion of the results.

4.1 Obtaining the Data

The process of obtaining the results of this benchmark was based on an approach similar to the one described in Section 3.1. The data format is consistent with the previously described scenarios, but contains several extra entries for each observation, due to the more complex nature of this experiment. The Unreal results were obtained by using the same approach described in Chapter 3: creating an Unreal project with a map that holds the initial state of the benchmark, according to the theoretical description that will be provided, and logging the measures for each actor in the experiment in every iteration of the physics engine. Furthermore, the theoretical results were also obtained using Python, under the same aforementioned Jupyter Notebook framework, and the visualisation of data was once again made possible by the Matplotlib visualisation library.

In comparison, the process of obtaining the Gazebo results was more complex, since this benchmark is executed once for each physics engine that Gazebo supports. To be more specific, Gazebo offers support for four different physics engines to perform its calculations: ODE, Simbody, Bullet, and Dart. The first three engines were used to gather results for each variation of this benchmark. Due to the large number of parameters involved in this experiment, and the fact that the scenario had to be run three times, once for every physics engine under Gazebo, for each choice of parameters, a need to automate the process of obtaining the results arose. Therefore, the Google Test [33] library was used to run the Gazebo world with the different parameters and physics engines needed, and log the results in an automated manner. Moreover, using Google's testing framework facilitates the development of other benchmarks, since one would only have to change the initial Gazebo world, along with the initial parameters to obtain results for other physics experiments.

4.2 Theoretical Description

This experiment is based on a benchmark presented by Steven Peters and John Hsu in a talk from ROSCon [34]. It involves having 5 free-floating boxes with a specific initial linear and angular velocity. The response of the boxes as they are affected by the velocities is verified against analytical solutions that first neglect gravity, and then model it. From a physical perspective, the relevant principles that are tested by this benchmark are momentum conservation and the movement of a rigid body with three distinct principal moments of inertia, a result in classical mechanics which is described as the Dzhanibekov effect. The boxes used in this experiment have uniform density, a mass of 10 kilograms, and their dimensions are in the order of centimeters, as shown in Figure 27.

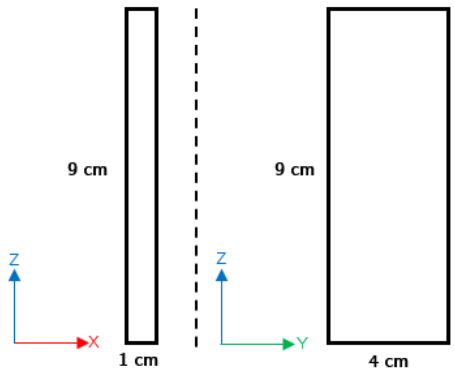


Figure 27: Dimensions of the cuboid used in the free-floating rigid bodies experiment

Furthermore, it is important to specify that although the cuboids were initially not rotated along any axis, the coordinate frame attached to the center of gravity of each cuboid was aligned to the world coordinate frame. This is why the quaternion introduced in the following paragraphs has the initial value of identity. To aid the reader in creating a better visual understanding of the experiment, Figure 28 shows the Unreal scene which contains the initial state of each of the 5 cuboids used in this scenario. Note that the world coordinate frame and the coordinate frame of one of the cuboids can be observed by looking at the bottom-left gizmo, and respectively at the selected cuboid's gizmo. Moreover, the initial position, rotation, and scale of one of the cuboids is shown in the blue frame in the right corner of the picture. Note that the default dimension of a cuboid in Unreal is $100 \times 100 \times 100$ centimeters, motivating the values of the object scale seen in the figure.

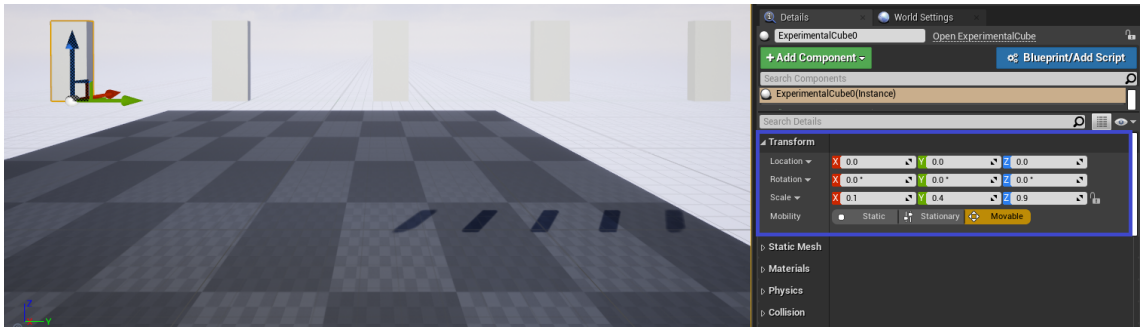


Figure 28: Unreal scene containing the initial state of the free-floating rigid bodies experiment

Since this experiment involves the cubes being rotated, it is important to specify their moments of inertia, which broadly speaking measures the extent to which the cubes resist rotational acceleration. Since the cubes are solid and have uniform density, the moments of inertia are given by the following formulas, where m represents the cube's mass, w is its width, h is the height of the cube, and d its depth:

$$J_h = \frac{1}{12} m \left(w^2 + d^2\right)$$
$$J_w = \frac{1}{12} m \left(d^2 + h^2\right)$$
$$J_d = \frac{1}{12} m \left(w^2 + h^2\right)$$

Then, by combining the above results, the cube's inertia matrix $\mathbf{J}$ is given by:

$$\mathbf{J} = \begin{bmatrix} J_h & 0 & 0 \\ 0 & J_w & 0 \\ 0 & 0 & J_d \end{bmatrix}$$

To describe the behaviour of a cube in this benchmark, more measures have to be characterized as a function of time, not solely linear velocity and linear position as in the previous experiments. The theoretical description of this scenario is based on Steven Peter's work that is available on GitHub [35] and contains modifications that aim to help the reader follow easily through the explanation. Note that Euler's notation is used to express differentiation with respect to time:

Let: O = An inertial frame

c_f = The coordinate frame attached to the center of gravity of the cube

m = The mass of the cube

$\mathbf{J}$ = The inertia matrix of the cube expressed in c_f

$\mathbf{g}$ = The gravity vector

$\mathbf{c}(t)$ = The position of the cube's center of gravity in the frame O as a function of time

$\dot{\boldsymbol{c}}(t)$ = The cube's linear velocity in frame c_f as a function of time

$\boldsymbol{\omega}(t)$ = The cube's angular velocity in frame c_f as a function of time

$q(t)$ = Quaternion holding the orientation of c_f with respect to frame O as a function of time

$\mathbf{R}(q)$ = A rotation matrix from O to c_f

$\mathbf{p}(\mathbf{t})$ = The cube's linear momentum in frame O as a function of time

$\mathbf{L}(\mathbf{t})$ = The angular momentum with respect to c_f as a function of time

$T(t)$ = The cube's kinetic energy as a function of time

$V(t)$ = The cube's potential energy as a function of time

$E(t)$ = The cube's total energy as a function of time

Lastly, denote the initial values of the measures introduced so far by the subscript 0.

For example, the initial linear velocity is denoted by $\dot{\mathbf{c}}_\mathbf{0}$.

Then, using the definition of the quantities introduced so far yields:

$$\mathbf{p}(\mathbf{t}) = m\dot{\mathbf{c}}(\mathbf{t})$$
$$L(\mathbf{t}) = \mathbf{R}^{\intercal}(q(t)) \cdot \mathbf{J} \cdot \boldsymbol{\omega}(t)$$
$$T(t) = \frac{1}{2m}(\dot{\mathbf{c}}^{\intercal}(t) \cdot \dot{\mathbf{c}}(t)) + \frac{1}{2}\boldsymbol{\omega}^{\intercal}(t) \cdot \mathbf{J} \cdot \boldsymbol{\omega}(t)$$
$$V(t) = -m(\mathbf{g}^{\intercal} \cdot \mathbf{c}(t))$$
$$E(t) = T(t) + V(t)$$

A theoretical solution starts from the observation that gravity is the only external force, therefore, from Newton's second law of motion, one can get an expression for the time derivative of the linear momentum and the linear acceleration of the cube as:

$$\dot{\mathbf{p}}(t) = m\mathbf{g}$$
$$\ddot{\mathbf{c}}(t) = \mathbf{g}$$

Based on the same observation, the cube's velocity is linear in time and is given by:

$$\dot{\mathbf{c}}(t) = \dot{\mathbf{c}}_{\mathbf{0}} + \mathbf{g}(t - t_0)$$

And the cube's position is given by a parabolic function of time:

$$\mathbf{c}(t) = \mathbf{c}_{\mathbf{0}} + \dot{\mathbf{c}}_{\mathbf{0}}(t - t_0) + \frac{1}{2}\mathbf{g}(t - t_0)^2$$

With no external torques affecting the cube, angular momentum is conserved and it is going to be constant in the fixed frame:

$$\mathbf{L}(t) = \mathbf{L}_{\mathbf{0}}$$
$$\mathbf{R}^{\intercal}(q(t)) \cdot \mathbf{J} \cdot \boldsymbol{\omega}(t) = \mathbf{R}^{\intercal}(q_0) \cdot \mathbf{J} \cdot \boldsymbol{\omega}_{\mathbf{0}}$$

Lastly, since gravity is a conservative force, the system has constant energy:

$$E(t) = E_0$$

4.3 Simple Scenario

In the simple scenario, gravity is not taken into consideration, therefore its value is set to 0. Apart from setting gravity to 0, the complete initial configuration for this scenario is given by:

$$\mathbf{c}_0 = [0, 0, 0]^{\intercal}$$
$$\mathbf{q}_0 = \mathbf{I}_3$$
$$\dot{\mathbf{c}}_{\mathbf{0}} = [-0.9, 0.4, 0.1]^{\intercal}$$
$$\boldsymbol{\omega}_{\mathbf{0}} = [0.5, 0, 0]^{\intercal}$$

From the equations presented above, setting gravity to 0 has a series of effects that lead to the simplification of the analytical solution. The linear velocity reduces to its initial value and the position of the cube's center of mass becomes a linear function of time. Furthermore, the potential energy becomes 0, which because the system is conservative, implies that the kinetic energy is also constant and equal to its initial value. Note that the angular velocity is aligned with the X-axis, which has the highest rotational stability since J_h is the largest among the inertia tensors. Based on these observations, one would expect the position of each cube in the simple scenario to follow a straight line with constant linear velocity. Additionally, the angular momentum and the angular velocity should remain constant, and the cumulative rotation angle should vary linearly with time.

Analysing the results of the simple scenario in the two systems confirmed these assumptions. Moreover, based on the experience gained with the previous experiments that showed the influence of the frame duration on Unreal results, this scenario was tested for a variety of frame lengths. In order to aid visualisation of the data, the plots in the figures below show the maximum error from each physics engine compared to the theoretical results, for nine different frame lengths, as illustrated in Figure 29 that contains the maximum linear velocity and linear position errors obtained in the simple scenario.

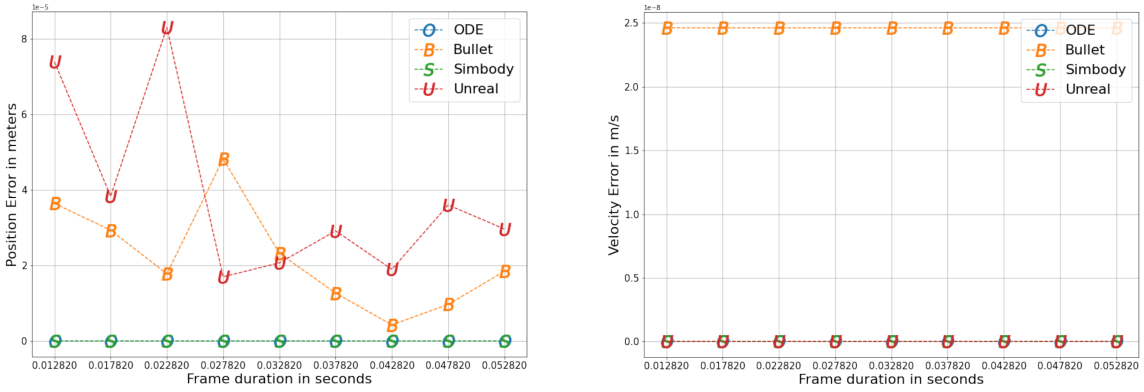


Figure 29: Plots showing the maximum errors from the simple scenario of the benchmark. From left to right: maximum X-axis position error over frame duration, maximum X-axis linear velocity error over frame duration

Figure 29 shows that both ODE and Simbody perfectly match the theoretical results, having no errors in either the X-position or the X-velocity, independent of the frame duration. This result is expected since linear velocity should remain constant at its initial value in the simple scenario. In the case of Unreal and Bullet, there are visible errors shown by the results, however, the magnitude of the errors is extremely small, in the order of 10^{-5} meters for the position, and respectively 10^{-8} m/s for the linear velocity. Moreover, the error shows no correlation to the frame duration, therefore it has been concluded that the observed discrepancies are caused by floating-point errors and do not imply the existence of any flaws in the underlying calculations performed by the physics engines. Similar results have been obtained for the angular momentum and the total energy in the simple scenario, as shown in Figure 30.

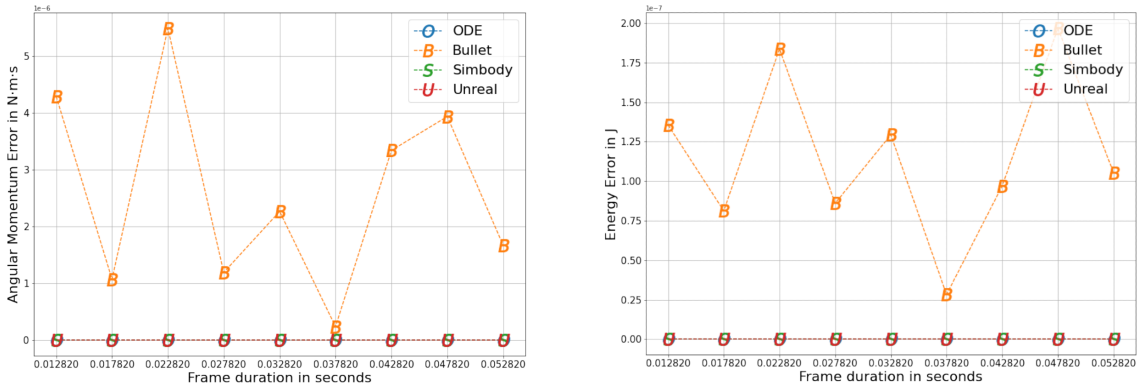


Figure 30: Plots showing the maximum errors from the simple scenario of the benchmark. From left to right: maximum angular momentum error over frame duration, maximum total energy error over frame duration

4.4 Complex Scenario

In the complex scenario, gravity is included and the complete description of the initial state is given by:

$$\mathbf{g} = [0, 0, -9.8]^\intercal$$
$$\mathbf{c}_0 = [0, 0, 0]^\intercal$$
$$\mathbf{q}_0 = \mathbf{I}_3$$
$$\dot{\mathbf{c}}_\mathbf{0} = [-2.0, 2.0, 8.0]^\intercal$$
$$\boldsymbol{\omega}_\mathbf{0} = [0.1, 5.0, 0.1]^\intercal$$

Notice that compared to the simple scenario, the angular velocity is aligned with the Y-axis and the initial velocities have significantly larger values. In this case, the expected trajectory of the cubes is parabolic, based on the equations derived above. Furthermore, angular momentum will remain constant, but this time it should manifest in a rapidly decreasing angular velocity. Moreover, because gravity is now set, the potential energy will have non-zero values, implying that kinetic energy will no longer be constant, as was the case in the simple scenario.

Similar to the simple scenario, the maximum errors of various measures have been analysed and plotted, as displayed in Figure 31. Unlike the simple scenario, the position error is significant and increases as the frame duration gets longer. It can be seen that Unreal follows the general pattern set by ODE and Bullet, whilst Simbody performs the best, having a significantly smaller error that does not seem to be affected as much by the frame length. In the case of the linear velocities, Unreal and Bullet underperform when compared to ODE and Simbody, but once again the observed errors are small and seem to be simply noise from the physics computations.

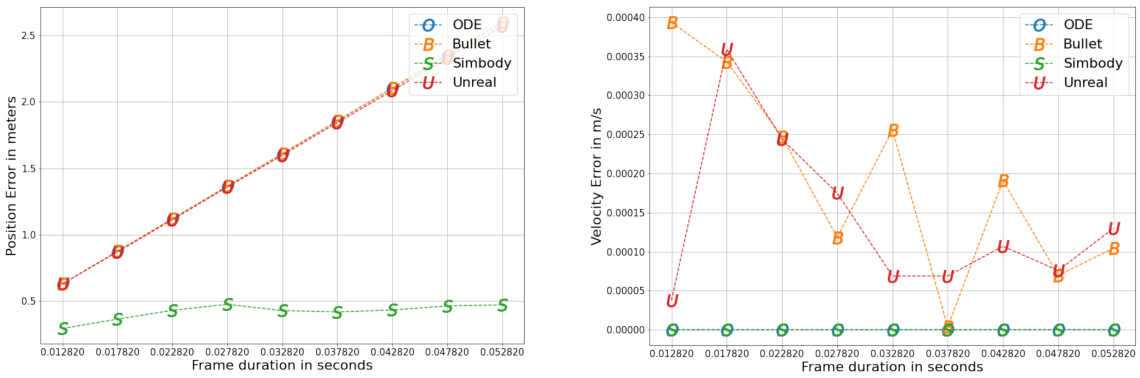


Figure 31: Plots showing the maximum errors from the complex scenario of the benchmark. From left to right: maximum X-axis position error over frame duration, maximum X-axis linear velocity error over frame duration.

On the other hand, Unreal has the best accuracy when it comes to angular momentum conservation, being closely followed by Simbody. Moreover, ODE shows an error that increases with frame duration, implying that the angular velocity and momentum calculations might be slightly inaccurate or even flawed. Lastly, the total energy error shows similar behaviours, with Unreal being consistent with ODE and Bullet, the error increasing linearly with frame duration, whilst

Simbody achieves the best results that seem to be independent of the frame length. These results are displayed in Figure 32.

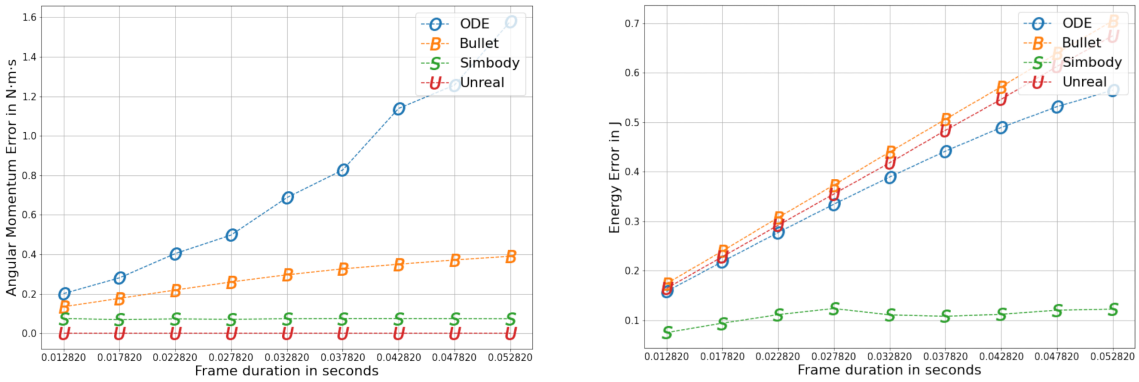


Figure 32: Plots showing the maximum errors from the complex scenario of the benchmark. From left to right: maximum angular momentum error over frame duration, maximum total energy error over frame duration

4.5 Summary

To summarise, an existing benchmark used by robotics researchers was adapted to include results from Unreal Engine. Additionally, a theoretical description of the two scenarios of this experiment, one neglecting the effects of gravity and one including them, has been offered in sufficient detail to allow the recreation of the benchmark in other simulators.

The results that have been obtained and presented throughout this chapter show that Unreal's performance is similar to both ODE and Bullet running in Gazebo, whilst Simbody seems to be the best performing physics engine in this particular scenario.

Chapter 5

Simulating a Basic Rover Model

This chapter contains a presentation of the steps taken in order to simulate a basic rover model in both Unreal and Gazebo. It includes the challenges that have been encountered when trying to create equivalent models in the two simulators, and the approaches taken in trying to overcome these challenges.

5.1 Cylindrical Wheels Model

The model used to represent the rover throughout this chapter was a very simplistic one, initially being represented by a rover body in the form of a cube, sitting on top of four cylindrical wheels, as shown in Figure 33.

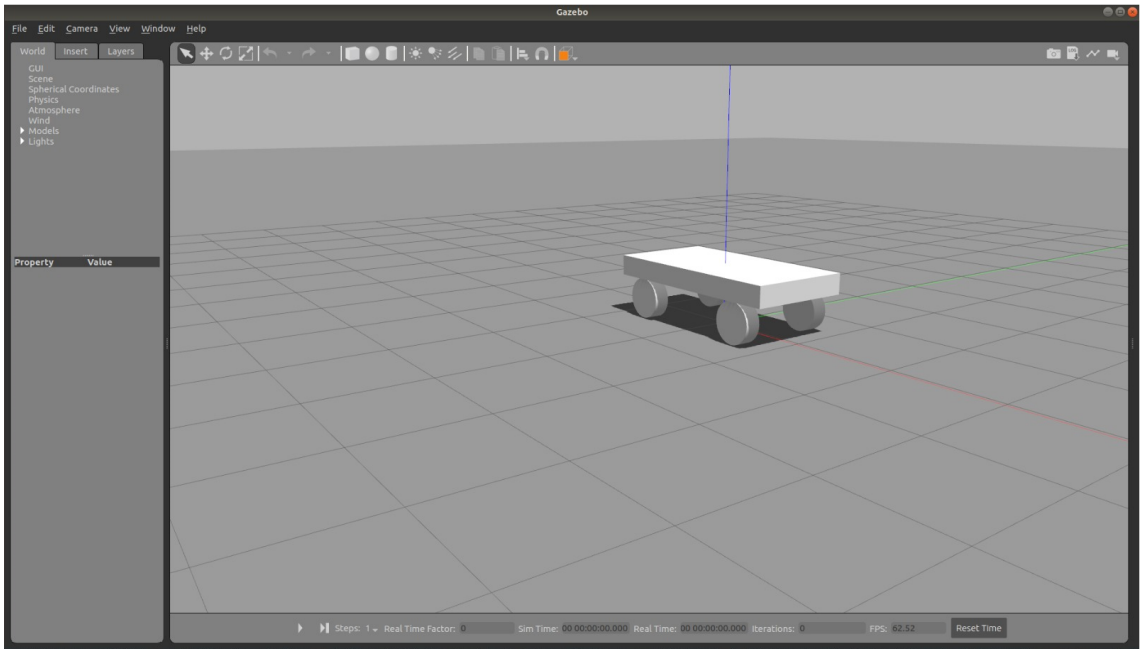


Figure 33: Basic rover model in a Gazebo world

Creating this rover model in Gazebo was a smooth process since Gazebo supports the usage of the unified robot description format (URDF) [36] out-of-the-box. URDF is one of the most popular formats for representing a robot model, being extensively used by Gazebo and the entire Robot

Operating System (ROS) [37] tool suite. Due to its popularity, URDF has become the main approach used for sharing model robots between different Gazebo worlds and projects. Therefore, in order to obtain the model shown in Figure 33, one only has to specify the primitives that create the rover, in this case, one cube and 4 cylinders, and their positions and scales. Next, the primitives can be connected together with dynamic and kinematic properties by the usage of URDF joints. Once again, creating these joints is a straightforward task, requiring only the choice of the type of joint, and the links that it should join, together with the position of the joint and parameters such as the joint's stiffness and movement limits.

In contrast, since Unreal Engine does not have a tool that can parse a URDF file and create a model in the Unreal scene, obtaining an equivalent model in Unreal represented a tedious and error-prone process, even for this very basic robot. Further, Unreal's physics engine, PhysX, does not include a cylinder collision primitive, which meant that in order to generate a cylinder in the Unreal scene, one would need to create a cylinder shape by using an external tool and then import it into Unreal. Based on the formats that Unreal can import into the scene, the tool used to create this shape was Blender [38], a widely used, free, and open-source 3D creation suite.

After creating the cylinder shape and importing it into an Unreal scene, obtaining an equivalent rover seemed to become an easy process. However, after scaling and positioning the elements correctly, along with creating physics constraints, which are relatively similar to Gazebo's joints, the behaviour of the rover model in Unreal was completely unrealistic. With cylindrical wheels, one would expect the rover to smoothly roll if it is placed on completely flat terrain and a force is applied to it. This behaviour was confirmed in Gazebo, but the Unreal rover was bouncing on top of the ground, which prevented it from rolling as expected. After a series of analyses that focused on the position, orientation, and scale of the rover's components, along with joint configuration parameters and terrain, the cause of the unexpected Unreal behaviour was once again caused by an underlying PhysX limitation. Since PhysX only allows 256 triangles for each shape, the mesh representing the cylinder object ends up being cut into multiple triangles, as illustrated in Figure 34, which in turn explains the behaviour of the rover in Unreal.

Figure 34: Cylindrical wheel collider in Unreal, note that the original mesh is represented by multiple triangle meshes

In order to overcome this limitation imposed by PhysX's approach of tackling collisions, one could change the rover model to use collision primitives that are available to PhysX, such as spheres or capsules, as discussed in the next section. It is worth noting that according to Epic Games, Unreal is migrating to a new physics system called 'Chaos Physics', which might solve this issue and could provide more support for vehicles and increased stability at lower iteration counts over PhysX.

5.2 Spherical Wheels Model

Based on the observations discussed in the previous section, the model was modified to encompass spherical wheels, as illustrated in Figure 35. Although the model looks unrealistic and spherical wheels are not often used in vehicles, it offers the two systems the possibility to simulate the new model in a more similar manner.

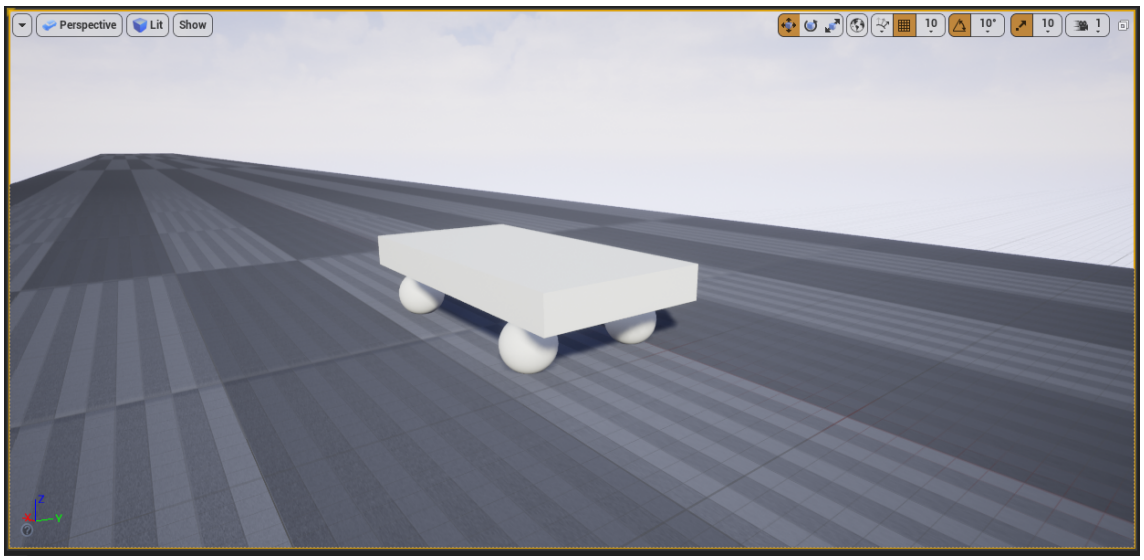


Figure 35: Spherical wheel rover model in an Unreal scene

Switching to the spherical wheels model, the observed behaviour in Unreal became closer to the expectations, with the rover starting to slide smoothly as soon as a force was applied to it. However, the two simulators showed significant discrepancies. Mainly, the rover model in Gazebo never stops after it starts rolling, implying that there is no friction being modelled between the spherical wheels and the ground. In contrast, the Unreal model stops after the force is applied, and the stopping time is correlated to the friction parameters used for the ground and the wheels of the rovers. Therefore, Unreal includes the friction between the spherical wheels and the ground, whilst Gazebo does not. Although intuitively one would expect the Unreal behaviour to be correct, the contrary is true. Since spherical wheels are being used in the model, the contact point between each wheel and the ground should be represented by the physics engines as a single point, not a continuous surface, which should translate into a lack of friction force between each wheel and the ground. Therefore, Unreal's behaviour implies that the sphere's point of contact is not a point, but rather a larger surface. This assumption was proved by a closer examination of the model which revealed that the collision of the default Unreal sphere mesh is not smooth, as illustrated in Figure 36.

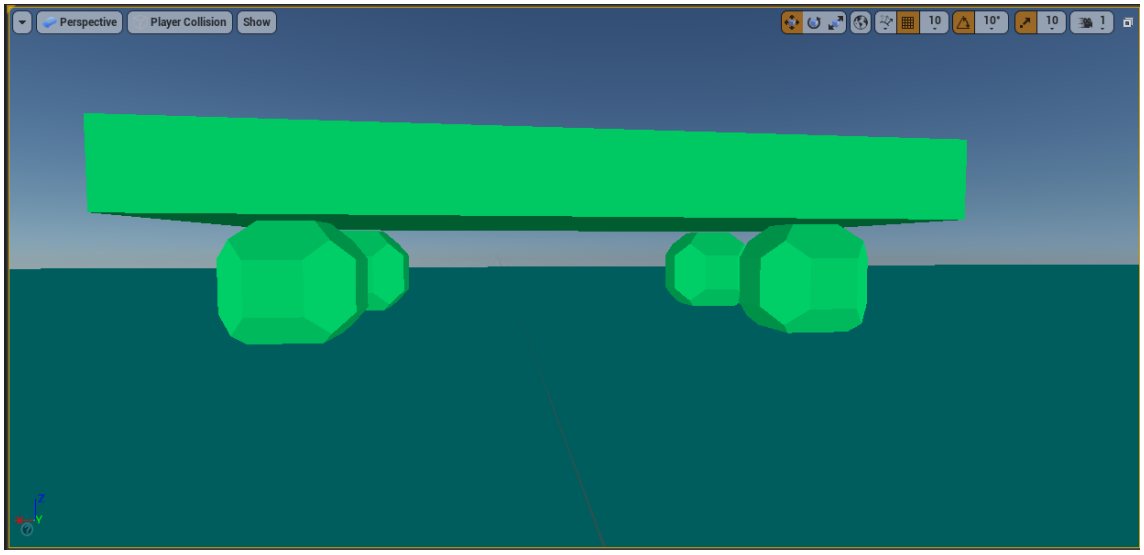


Figure 36: Spherical wheel collider in Unreal, note that the spherical mesh is not smooth

Once again, a custom shape was created in Blender and imported into Unreal in order to create a smooth collision for the wheels, with the result of this process shown in Figure 37.

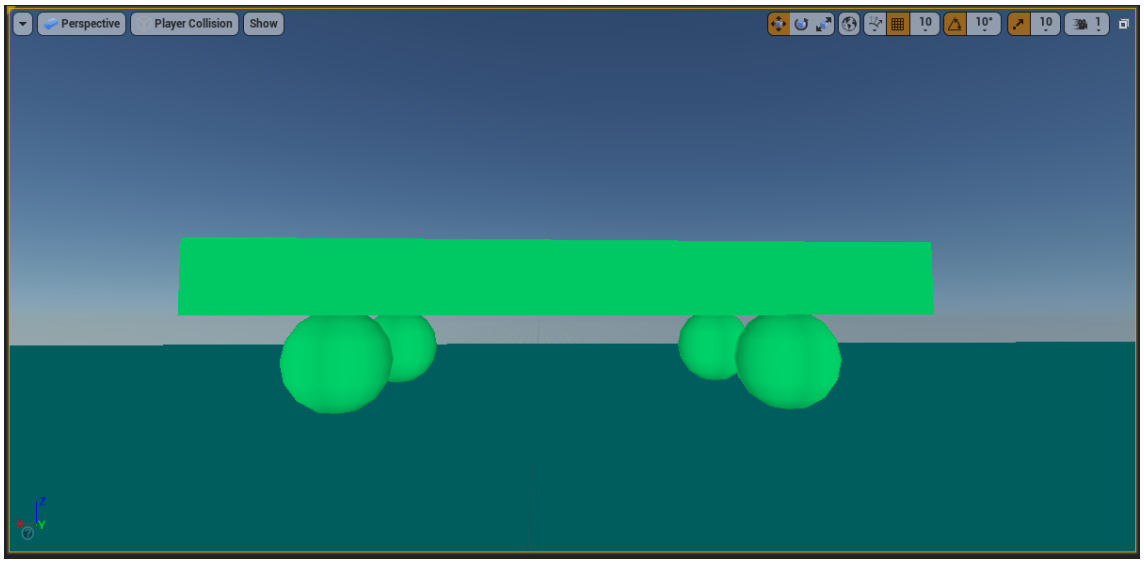


Figure 37: Custom spherical wheel collider in Unreal, note that the mesh is now smooth

5.3 Ignoring the Effects of Friction

Even after obtaining a correct spherical wheel, the Unreal rover was still influenced by friction between the wheels and the ground. Therefore, it was decided to investigate how the rover model performs when friction is completely ignored. Initially, the rover seemed to behave as expected, following a straight line with a constant velocity. However, a closer analysis of the results revealed that the rover tends to drift, translating into a linearly decreasing yaw of the rover body, as seen in Figure 38.

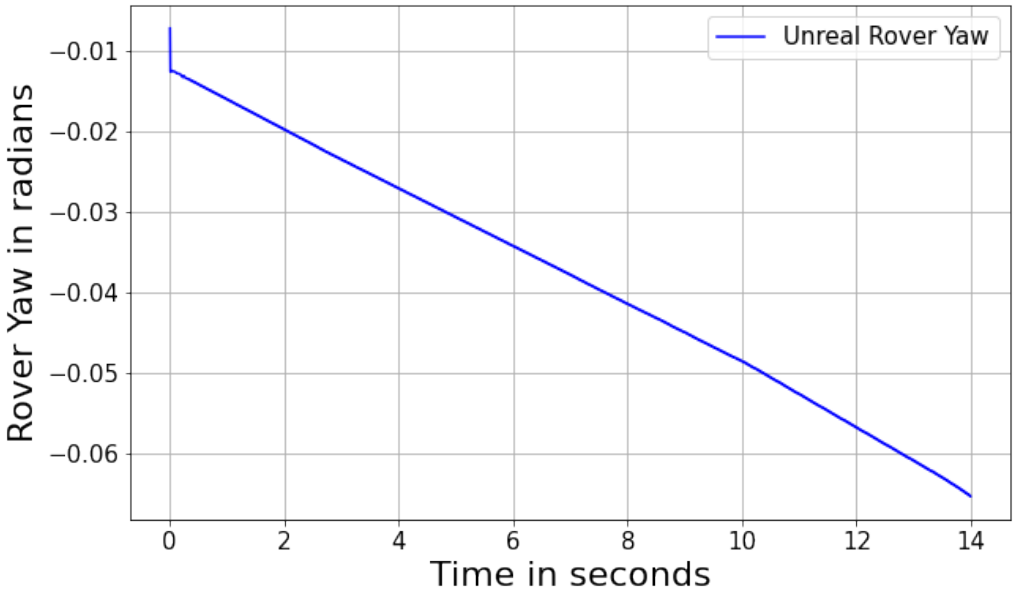


Figure 38: Unreal rover body's yaw over time

The drifting observed when ignoring the effects of friction implies that there are other issues with the rover model not caused by the wheels colliders or friction between them and the ground. Most likely, these issues are caused by the physics constraints that are used to join the wheels with the rover's body. However, the exact cause of this behaviour was not identified during this project, and the challenges encountered and presented during this chapter represent the main motivation of the proposed URDF importer, which is discussed as future work in Section 6.3.1.

5.4 Summary

To summarise, this chapter presented the reader with the challenges encountered when trying to create a basic rover model in the two simulators, highlighting the importance of a URDF importer tool, even for very simple models as the box placed on top of four cylindrical wheels. Although the cause of the discrepancy shown by the rover model in Unreal was not completely isolated, this chapter offers context and can provide useful insights if this project is continued.

Chapter 6

Conclusions

The final chapter of this report contains a summary of the project's achievements, an evaluation, and a series of proposals that could make Unreal a more suitable simulator for planetary rovers. Lastly, it will offer a conclusion including the final considerations about the entirety of the project.

6.1 Achievements

Recall that the main goal of this project was to perform a preliminary analysis on the possibility of using Unreal Engine 4 as a planetary rover simulator, and based on a series of discussions with researchers from NASA Ames Research Center, the main aspect that required a detailed analysis was Unreal's accuracy from a physics perspective. The results of the experiments described throughout this report have shown that Unreal manages to achieve similar accuracy to any of the physics engines that Gazebo supports. Therefore, it can be concluded that from a preliminary perspective, Unreal's physics engine can be accurate enough to allow the accurate simulation of planetary rovers, and represents a suitable candidate for a more thorough investigation of its physics engine capabilities.

Furthermore, this project has created a framework that allows the comparison of Unreal and Gazebo in a relatively effortless manner. Based on the work done so far, one can create and run multiple, potentially significantly more complex benchmarks, only by setting the correct initial state in both systems and using the functionalities that have been developed. Hence, this project can constitute the starting point of a more in-depth analysis between Unreal and Gazebo and could be easily extended to take into account any other simulators. One method of including other simulators is to create the same experiments used in the process of validating Unreal by using the thorough theoretical descriptions provided and logging the results in the format presented in Chapter 3.

6.2 Evaluation

Initially, the end goal of this project was to validate Unreal's physics engine for accurately simulating rovers and their complex interaction with deformable terrain. This project did not manage to achieve its final goal due to the complexity of the problem tackled, and steep learning curves for the technologies used. Using these technologies implied not only the understanding of how to use each simulator but also comprehending the underlying physics engine implementations, which is a

difficult and complex task. However, this project has managed to create a starting point in this direction, one that not only includes relevant experiments and results but also defines a methodology and a framework that can be used to further examine Unreal's capabilities as a planetary rover simulator. Furthermore, through the meetings and discussions that took place throughout this project with researchers from NASA Ames Research Center, a series of related projects could be created with guidance offered by NASA researchers, providing an extremely interesting opportunity for future robotics projects.

6.3 Future Work

Since Unreal seems to be suitable from a physics perspective to model the complex interactions of rovers with the terrain, two main directions can be followed to further enhance its usage as a planetary rover simulator: software quality of life developments and more detailed validation of its physics engine. The following subsections present one idea for each of these two directions.

6.3.1 URDF Importer

Creating a tool that can parse any URDF file and create a corresponding model in an Unreal scene would be a significant improvement for the purpose of using Unreal either as a general robotics simulator or in rover simulations. As shown in the discussion from Chapter 5, such a tool would provide tremendous value to users and would save considerable development time. Whilst creating such a tool is a complex task that is beyond the scope of this project, one could start by looking at the approach taken by URoboSim [39]. However, note that URoboSim is deprecated and used a relatively old version of Unreal, therefore the development methodology should be adapted accordingly. Furthermore, there is a custom tool developed by NVidia for their Isaac Sim [40] which offers the capability of importing certain URDF files into Unreal that could provide helpful insights for developing a more general URDF importer.

6.3.2 Further Simulation

Even though the experiments presented in this project provide a preliminary validation of Unreal's physics accuracy, a more thorough investigation of the behaviour of PhysX would be needed to completely validate Unreal as a planetary rover simulator. An approach that could be taken consists of the following steps:

1. Simulate a scenario in which a single rover wheel, without any suspension elements, behaves according to the widely-used terramechanics model introduced by Bekker in both Unreal and Gazebo. Run several simulations on this scenario, considering only the wheel slip, whilst varying the relevant parameters, and compare the results.

2. If Step 1 shows promising results, ask NASA Ames Research Center for physical experimental data obtained when testing the Center's simulators and compare the results against Unreal results.

3. Encompass a complete robot model into Unreal Engine and Gazebo and compare the results of the two related to wheel slip.

4. Modify the terramechanics model used in Step 1 to account for more complex effects such as bulldozing and multi-pass effects, or by introducing more advanced soil features, such as

embedded rocks that become unstable or break into multiple parts as the rover crosses over them.

The above four steps would constitute a complete validation of Unreal from a physics perspective, which in turn could transform Unreal Engine into the default simulator used by robotics researchers around the world, due to its myriad of features, especially related to complex graphics processing.

6.4 Final Considerations

To conclude, this project explored the usage of Unreal Engine 4 as a planetary rover simulator. It has shown Unreal's ability to solve shortcomings of existing simulators out-of-the-box due to its complex graphics processing functionalities. Furthermore, it has proved that Unreal's physics engine, PhysX, represents a suitable candidate for a more in-depth analysis since its behaviour on preliminary tests matched theoretical considerations and results of other physics engines used in rover simulations. It created a basis for a research question that is not thoroughly studied, the usage of game engines for accurate robotics simulations, and offered a starting point for future research on this topic. However, this project has also shown that Unreal Engine 4 is not currently perfectly suitable for robotics research, due to the lack of important quality of life features, such as a URDF importer, and that further development is needed for Unreal to achieve the role of the main tool used for robotics simulations.

Bibliography

[1] R. Bauer, W. Leung, T. Barfoot Development of a Dynamic Simulator Tool for the ExoMars Rover. *Proceedings of the International Symposium on Artificial Intelligence for Robotics and Automation in Space (iSAIRAS)*, Munich, Germany, Sept 5-9, 2005.

[2] N. Patel, A. Ellery, E. Allouis, M. Sweeting, L. Richter. Rover Mobility Performance Evaluation Tool (RMPET): A Systematic Tool for Rover Chassis Evaluation via Application of Bekker Theory. *8th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, Noordwijk, The Netherlands, 2-4 November, 2004.

[3] A. Jain, J. Guineau, C. Lim, W. Lincoln, M. Pomerantz, G. Sohl, R. Steele. ROAMS: Planetary Surface Rover Simulation Environment. *Proceedings of the International Symposium on Artificial Intelligence Robotics and Automation in Space (i-SAIRAS)*, Nara, Japan, May 19-23, 2003.

[4] J. Yen, A. Jain, J. Balaram. ROAMS: Rover Analysis, Modeling and Simulation. *Proceedings of the International Symposium on Artificial Intelligence Robotics and Automation in Space (i-SAIRAS)*, Noordwijk, The Netherlands, June 1999.

[5] C. Harnisch, B. Lach. Off Road Vehicles in a Dynamic Three Dimensional Realtime Simulation. *Proceedings of the 14th International Conference of the International Society for Terrain-Vehicle Systems*, Vicksburg, MS, US, October 20-24, 2002.

[6] F. Zhou, RE. Arvidson, K. Bennett, B. Trease, R. Lindemann, P. Bellutta, K. Iagnemma, C. Senatore. Simulations of Mars rover traverses. *Journal of Field Robotics 31*, 2014.

[7] M. Allan, U. Wong, P.M. Furlong, A. Rogg, S. McMichael, T. Welsh, I. Chen, S. Peters, B. Gerkey, M. Quigley, M. Shirley, M. Deans, H. Cannon, T. Fong. Planetary rover simulation for lunar exploration missions. *2019 IEEE Aerospace Conference (pp. 1-19)*, March, 2019.

[8] M. Lewis, S. Balakirsky, J. Wang. USARSim: a robot simulator for research and education. *IEEE International Conference on Robotics and Automation*, May, 2007.

[9] S. Shah, D. Dey, C. Lovett, A. Kapoor. Aerial Informatics and Robotics platform. *Technical Report MSR-TR-2017-9, Microsoft Research*, 2017.

[10] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun. CARLA: An Open Urban Driving Simulator. *1st Conference on Robot Learning*, Mountain View, United States, November, 2017.

[11] P. Humphreys. Computer Simulations. *Proceedings of the Biennial Meeting of the Philosophy of Science Association*, Volume Two: Symposia and Invited Papers, 1990.

[12] P. Bratley, B.L. Fox, L.E. Schrage. A Guide to Simulation. *Springer Science*, 2nd Edition, 1987.

[13] A. Boeing, T. Bräunl. Evaluation of real-time physics simulation systems. Jan. 2007.

[14] K. Erleben. Stable, Robust, and Versatile Multibody Dynamics Animation. *PhD Thesis, Department of Computer Science, University of Copenhagen*, Apr. 2005.

[15] L. Kavan. Rigid body collision response. *Proceedings of the 7th Central European Seminar on Computer Graphics*, Aug. 2007.

[16] S. Hadap, D. Eberle, P. Volino, M. C. Lin, S. Redon, C. Ericson. Collision detection and proximity queries. *ACM SIGGRAPH*, 2004.

[17] D. Baraff. Physically Based Modeling: Principles and Practice. *Carnegie Mellon University*, 1997.

[18] M.G. Bekker. Theory of land locomotion. *Ann Arbor, Michigan: The University of Michigan Press*, 1956.

[19] M.G. Bekker. Off-the-road locomotion. *Ann Arbor, Michigan: The University of Michigan Press*, 1960.

[20] M.G. Bekker. Introduction to terrain–vehicle systems. *Ann Arbor, Michigan: The University of Michigan Press*, 1969.

[21] V.M. Asnani, D.C. Delap, C.M. Creager. The development of wheels for the Lunar Roving Vehicle. *Journal of Terramechanics*, 2008.

[22] D. Vaniman, R. Reedy, G. Heiken, G. Olhoeft, W. Mendell. The lunar environment. *The lunar Sourcebook*, CUP, pages 27-60, 1991.

[23] R. Soni, B. Mishra, and R. Venugopal. A review on discrete element method (dem): Steps, methodologies, development and applications in simulation of granular flow. May, 2012.

[24] Epic Games Unreal Engine 2021, available at: https://www.unrealengine.com.

[25] U. Wong, O. Umurhan, A. Tardy, T. Welsh, M. Allan, L. Edwards. FROST Dataset: Features Relevant to Ocean Worlds Surface Terrain. *NASA Ames Research Center*, Dec, 2019, available at: https://ti.arc.nasa.gov/dataset/frost/.

[26] Open Source Robotics Foundation. Gazebo Simulator. 2014.

[27] S. Paepcke, L. Poubel. Whats new in gazebo? upgrading your simulation user experience! *ROSCon*, 2016.

[28] C.E. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J. Rivero, J. Manzo, et al. Inside the virtual robotics challenge: Simulating real-time robotic disaster response. *IEEE Transactions on Automation Science and Engineering*, pages 494-506, 2015.

[29] K.A. Hambuchen, M.C. Roman, A. Sivak, A. Herblet, N. Koenig, D. Newmyer, R. Ambrose. Nasa's space robotics challenge: Advancing robotics for future exploration missions. *AIAA SPACE and Astronautics Forum and Exposition*, page 5120, 2017.

[30] The pandas development team. pandas-dev/pandas: Pandas. *Zenodo*, Feb. 2020, available at: https://doi.org/10.5281/zenodo.3509134.

[31] J.D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science Engineering*, vol. 9, no. 3, pages 90-95, 2007.

[32] J.M. Hsu, S.C. Peters. Extending open dynamics engine for the darpa virtual robotics challenge. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 37-48, 2014.

[33] Google, Google Test, Oct. 2019, available at: https://github.com/google/googletest

[34] J.M. Hsu, S.C. Peters. Comparison of Rigid Body Dynamic Simulators for Robotic Simulations in Gazebo. *ROSCOn*, 2014.

[35] S.C. Peters. Physics benchmarks using the Gazebo Simulator. *Available at: https://github.com/scpeters/benchmark*.

[36] I. Sucan, J. Kay. Universal Robotic Description Format. Available at: https://github.com/ros/urdf.

[37] Open Source Robotics Foundation Robot Operating System. Available at: https://www.ros.org/.

[38] Blender Online Community. Blender - a 3D modelling and rendering package. *Blender Foundation*, 2018, available at: http://www.blender.org.

[39] A. Haidu, Y. Li, R. Yousaf, Y. Kim. URoboSim. Jul, 2018. Available at: https://github.com/robcog-iai/URoboSim.

[40] NVIDIA Development Team. Isaac Sim: Omniverse Robotics App. 2020. Available at: https://developer.nvidia.com/isaac-sim.

Appendix A

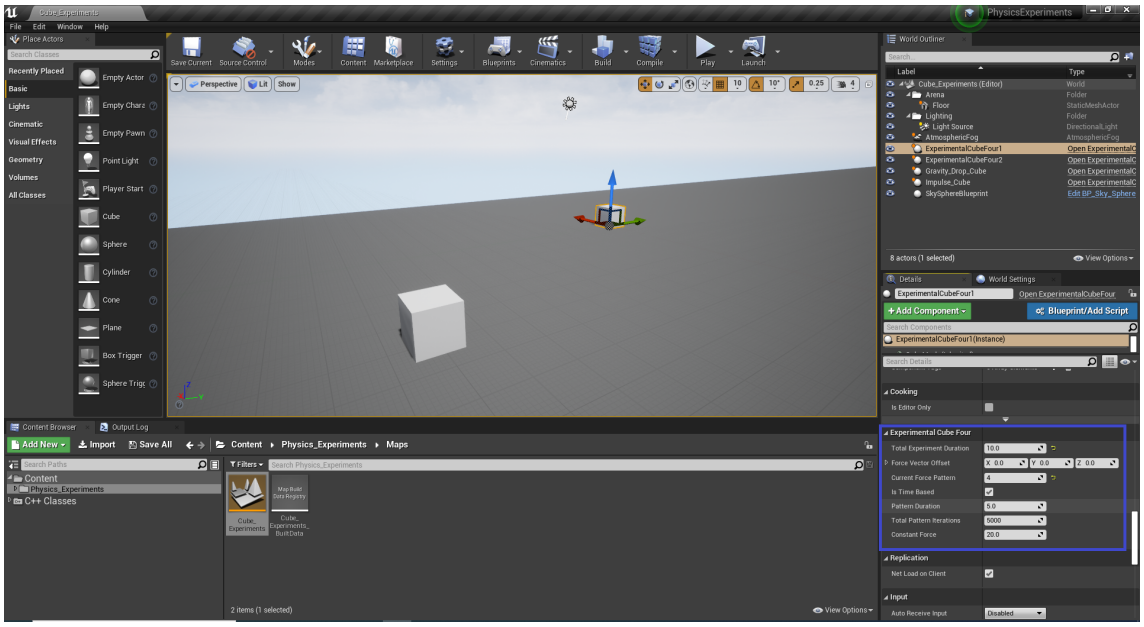
User Manual

This section will provide any potential users with details on how to reproduce the results obtained and presented throughout this project.

A.1 Unreal Manual

In order to obtain the results from Unreal Engine, one has to download the Unreal Engine source code from the following repository: *https : //github.com/EpicGames/UnrealEngine/tree/release*. Afterward, the source code should be built from source, with Epic Game's official documentation covering this process thoroughly.

After installing Unreal, the projects provided in the repository attached to this report should be opened and built. Next, the results can be obtained by placing the relevant cubes inside a map, change the parameters accordingly, and simply hit the Play button. For example, in order to obtain the force pattern results, an object of the class *ExperimentalCubeFour* has to be created in the scene. One method of doing this is to simply drag and drop the class into the scene using the Unreal Editor. Afterward, the parameters of the cube and the experiment can be modified either directly from the C++ source code, or from the Unreal Editor, as shown in the figure below.



Once the desired parameters have been chosen, the results of the scenario will be saved into a CSV file in the same directory that contains the Unreal project, after Total Experiment Duration seconds have passed. The results for all other experiments can be obtained in a similar manner.

A.2 Gazebo Manual

In order to obtain the Gazebo results, one has to run every Gazebo world provided in the aforementioned repository. To run a Gazebo world, one can use the following steps:

1. Install Gazebo version 9.0 or higher.
2. If Gazebo was installed from debians, ensure the Gazebo development files were also installed, this can be achieved by the following command:

   $ sudo apt-get install libgazebo9-dev.

   If the Gazebo version is other than 9, replace the 9 in the command above with the current Gazebo version.
3. Create a new folder, say Root. Inside this new folder copy the CMakeLists, along with the .world files and any .cc files needed.
4. Inside the Root folder, create a build directory:
   $ mkdir build.
5. Compile the code:
   $ cd build
   $ cmake ../
   $ make
6. Add the library path to GAZEBO_PLUGIN_PATH:
   $ export GAZEBO_PLUGIN_PATH=$GAZEBO_PLUGIN_PATH: /Root/build
7. Open the world with gzserver:
   $ cd ..
   $ gzserver -u name_of_the_world_file.world
8. In a new terminal window, start the client:
   $ gzclient

Once the world is open, the results for any experiment can be obtained either by using Gazebo's built-in plotting functionality or by using the .cc files, which have a logging functionality developed.

A.3 Theoretical Solutions Manual

One can obtain theoretical results by opening the file named *Theoretical_Solutions_Generator.ipynb* in the Jupyter notebook framework or any other solution that can load and run .ipynb files, such as Google Colab. However, one will have to make sure that *numpy* and *pandas* are installed inside the chosen framework. Next, each cell has to be run in a sequential order to obtain the results. The CSV files storing the data will be saved in the same directory as the .ipynb file used to generate them.

A.4 Data Plotting

Having obtained all the data, one can create plots similar to the ones presented in this report by running the *Main_Plotting.ipynb* file provided in the GitHub repository. Once again, *numpy* and *pandas* have to be installed, along with *matplotlib*. Moreover, the variables holding the paths to the data, such as *unreal_dir_name*, have to be modified accordingly.

Appendix B

Source Code Repository

The source-code used for this project is entirely available at the following GitHub repository:

https://github.com/AndreiL26/Exploring-the-usage-of-Unreal-Engine-4-asa-planetary-rover-locomotion-simulator

The repository also contains the raw data used to create the figures shown in this project. It is worth specifying that the Unreal projects were built under version 4.25.3, whilst the Gazebo version used was 9.0.

Appendix C

Code Listing

This section will provide a partial code listing, specifically the one used to obtain the results presented in Chapter 3, whilst the entirety of the code is available in the repository mentioned in the previous section. The code used for Chapter 3 is listed here since it's considerably smaller in size and easier to understand and follow than the code used for Chapter 4, and respectively the attempt to create a rover model in Unreal.

C.1 Obtaining the Theoretical Results

```
import numpy as np
import pandas as pd

# Experiment 1 -  Gravity Drop-Test
def run_gravity_drop_test(intial_z_position, total_experiment_time, destination_file_name):
    # Initial state
    observation_times = [0]
    z_velocities = [0]
    z_positions = [intial_z_position]

    g = -9.81
    crt_time = 0.0

    # Simulate the cube falling until it touches the ground.
    # Note that since the dimensions of the cube are 1x1x1, in order to touch the ground, the cube's
    ↪  center of mass Z position should be 0.5
    while crt_time < total_experiment_time:
        if intial_z_position + (g*(crt_time**2))/2 <= 0.5:
            break
        crt_time += 0.001
        z_velocities.append(g * crt_time)
        z_positions.append(intial_z_position + (g*(crt_time**2))/2)
        observation_times.append(crt_time)

    # Record the stopping time and maximum linear velocity of the cube.
    stopping_time = crt_time
    maximum_velocity = z_velocities[-1]

    crt_time += 0.001
    # Complete the observations array to be consistent with the simulation duration in Gazebo and
    ↪  Unreal.
    while crt_time < total_experiment_time:
```

```
        z_velocities.append(0)
        z_positions.append(0.5)
        observation_times.append(crt_time)
        crt_time += 0.001

    # Load the data into a pandas data frame to add column labels.
    data_frame = pd.DataFrame.from_dict({'Time': np.transpose(observation_times),
                                'Z Velocity': np.transpose(z_velocities),
                                'Z Position': np.transpose(z_positions)})

    data_frame.to_csv(destination_file_name, index=False)

run_gravity_drop_test(10, 2, '10M_Gravity_Drop.csv')
run_gravity_drop_test(50, 4, '50M_Gravity_Drop.csv')
run_gravity_drop_test(75, 4, '75M_Gravity_Drop.csv')
run_gravity_drop_test(100, 5, '100M_Gravity_Drop.csv')

# Experiment 2 - Impulse Applied to a Cube
## Experiment 2.1 - Neglecting dynamic friction

applied_forces = [1000, 10000]
impulse_duration = 0.001 # This represents the duration for which the impulse is applied.
total_experiment_time = 4.0
cube_mass = 1


# Get the theoretical solutions for all the forces considered.
for crt_force in applied_forces:
    # Initial state
    observation_times = []
    y_velocities = []
    y_positions = []
    crt_time = 0

    # Avoid overstepping the total time by 0.001
    while crt_time < total_experiment_time - 0.001:
        crt_time += 0.001

        crt_y_velocity = (crt_force * impulse_duration)/cube_mass
        y_velocities.append(crt_y_velocity)
        y_positions.append(crt_y_velocity * crt_time)
        observation_times.append(crt_time)

    data_frame = pd.DataFrame.from_dict({'Time': np.transpose(observation_times),
                            'Y Velocity': np.transpose(y_velocities),
                            'Y Position': np.transpose(y_positions)})

    data_frame.to_csv(str(crt_force) + "N Impulse_No_Friction.csv", index=False)

## Experiment 2.2 - Including dynamic friction

def compute_position_integration(initial_velocity, g, friction_coefficient, t):
    # This function computes the integral from 0 to t for the position of the cube.
    # From theoretical results presented in section 3.3.2., the position is given by:
    # Position = integral(0->t)(v0 - g *  * t)dt
    return initial_velocity * t - (g * friction_coefficient * (t**2))/2

applied_forces = [1000, 10000]
impulse_duration = 0.001 # This represents the duration for which the impulse is applied.
total_experiment_time = 4.0
```

```
cube_mass = 1
g = 9.81
friction_coefficient = 1

# Get the theoretical solutions for all the forces considered.
for crt_force in applied_forces:
    # Initial state
    observation_times = []
    y_velocities = []
    y_positions = []
    crt_time = 0

    # Stage 1: Compute the velocity and position immediately after applying the force.
    while crt_time < impulse_duration:
        crt_time += 0.001
        crt_y_velocity = (crt_force * impulse_duration)/cube_mass
        y_velocities.append(crt_y_velocity)
        y_positions.append(crt_y_velocity * crt_time)
        observation_times.append(crt_time)

    initial_velocity = y_velocities[-1]

    # Stage 2: Compute the velocity and position as friction starts acting
    # Avoid overstepping the total time by 0.001
    while crt_time < total_experiment_time - 0.001:
        crt_time += 0.001

        crt_y_velocity = initial_velocity - (g * friction_coefficient * crt_time)
        if crt_y_velocity > 0:
            y_velocities.append(crt_y_velocity)
            y_positions.append(compute_position_integration(initial_velocity, g,
            ↪  friction_coefficient, crt_time))
        else:
            y_velocities.append(0)
            y_positions.append(y_positions[-1])

        observation_times.append(crt_time)

    data_frame = pd.DataFrame.from_dict({'Time': np.transpose(observation_times),
                            'Y Velocity': np.transpose(y_velocities),
                            'Y Position': np.transpose(y_positions)})

    data_frame.to_csv(str(crt_force) + "N Impulse_With_Friction.csv", index=False)

# Experiment 3 - Constant Force Applied to a Cube

def compute_force_position_integration(F, g, friction_coefficient, m, t):
    # This function computes the integral from 0 to t for the position of the cube.
    # From theoretical results presented in section 3.4., the position is given by:
    # Position = integral(0->t)(1/m * (F- m * g * ) * t)dt
    return ((F - m * g * friction_coefficient) * (t **2))/(2 * m)

# Note that this implementation is extremely rudimentary and only works correctly for forces that
↪  are larger than m * g *
applied_forces = [10, 20]
total_experiment_time = 10.0
force_applying_time = 5.0

m = 1
g = 9.81
```

```
friction_coefficient = 1

for crt_force in applied_forces:
    observation_times = []
    y_velocities = []
    y_positions = []
    crt_time = 0

    # Phase 1 - apply the force constantly for 5 seconds.
    while crt_time < force_applying_time:
        crt_time += 0.001
        crt_y_velocity = 1/m * (crt_force - m * g * friction_coefficient)*crt_time
        y_velocities.append(crt_y_velocity)
        y_positions.append(compute_force_position_integration(crt_force, g, friction_coefficient, m,
        ↪  crt_time))
        observation_times.append(crt_time)

    # Phase 2 - no force being applied, velocity starts decreasing due to friction.
    initial_velocity = y_velocities[-1]
    position = y_positions[-1]
    while crt_time < total_experiment_time - 0.001:
        crt_time += 0.001
        crt_y_velocity = initial_velocity - (g * friction_coefficient * (crt_time -
        ↪  force_applying_time))
        if crt_y_velocity > 0:
            y_velocities.append(crt_y_velocity)
            y_positions.append(position + compute_position_integration(initial_velocity, g,
            ↪  friction_coefficient, (crt_time - force_applying_time)))
        else:
            y_velocities.append(0)
            y_positions.append(y_positions[-1])

        observation_times.append(crt_time)

    data_frame = pd.DataFrame.from_dict({'Time': np.transpose(observation_times),
                            'Y Velocity': np.transpose(y_velocities),
                            'Y Position': np.transpose(y_positions)})

    data_frame.to_csv(str(crt_force) + "N Constant Force.csv", index=False)
```

C.2 Chapter 3 Unreal Code

C.2.1 Gravity Drop Test Cube - header file

```
/*
        Copyright [2021] [-]

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at

                http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
```

```
        limitations under the License.
 */

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ExperimentalCubeOne.generated.h"

class UStaticMeshComponent;
class UMaterial;

/* This class is used to represent a Cube for the Gravity Drop Test experiment. */
UCLASS()
class PHYSICSEXPERIMENTS_API AExperimentalCubeOne : public AActor
{
        GENERATED_BODY()

public:
        AExperimentalCubeOne();

protected:
        virtual void BeginPlay() override;

public:
        virtual void Tick(float DeltaTime) override;

        /* Mesh component, used for rendering of the Cube. */
        UPROPERTY(VisibleAnywhere)
        UStaticMeshComponent* CubeMesh;

        /* Custom Material used for the Cube, note that in the original experiments, the Physical
        ↪  Material parameters were: Friction = 0.01, Restitution = 0.0.*/
        UPROPERTY(VisibleAnywhere)
        UMaterial* CustomMaterial;

        /* Variable holding the experiment duration, therefore deciding the time period for which
        ↪  Observations will be taken. */
        UPROPERTY(EditAnywhere)
        float TotalExperimentDuration = 2.0f;

        /* Variable holding the time elapsed since the experiment began. */
        float CurrentTime = 0.0f;

        /* Function that adds an Observation to the ObservationsArray. */
        void AddObservation();

        /* Function that adds Labels to the existing observations and saves the ObservationsArray to
        ↪  a .csv file. */
        void AddObservationLabels();

        /* Array holding all the Observations and the Labels, its content will be saved as a .csv
        ↪  file. */
        TArray<FString> ObservationsArray;

        /* Variable holding the name of the file in which the Observations will be saved. */
        FString FileName = "Gravity_Drop_Test_Observations.csv";

        /* Auxiliary boolean used to determine if labels were added or not. */
        bool bAddedLabels = false;
```

```
        /* Variable holding the displacement relative to the origin. This is needed so that all
        ↪  experiments can be ran at the same time in one scene, with the obtained results still
        ↪  being relative to the origin. */
        UPROPERTY(EditAnywhere)
        FVector DisplacementVector = FVector(800.0f, 0, 0);
};
```

C.2.2 Gravity Drop Test Cube - source file

```
/*
        Copyright [2021] [-]

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at

                http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
 */

#include "ExperimentalCubeOne.h"
#include "Components/BoxComponent.h"
#include "Components/StaticMeshComponent.h"
#include "UObject/ConstructorHelpers.h"
#include "Kismet/KismetSystemLibrary.h"
#include "Materials/Material.h"
#include "TextFileManager.h"

AExperimentalCubeOne::AExperimentalCubeOne()
{
        // Create and attach a Static Mesh to the Cube.
        CubeMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Cube mesh"));
        CubeMesh->SetupAttachment(RootComponent);
        CubeMesh->SetStaticMesh(ConstructorHelpers::FObjectFinder<UStaticMesh>
        (TEXT("/Game/Physics_Experiments/Shapes/Shape_Cube.Shape_Cube")).Object);

        // Initialize basic Physics properties for the Mesh.
        CubeMesh->SetSimulatePhysics(true);
        CubeMesh->SetMassOverrideInKg();

        PrimaryActorTick.bCanEverTick = true;
}

void AExperimentalCubeOne::BeginPlay()
{
        Super::BeginPlay();
}

void AExperimentalCubeOne::Tick(float DeltaTime)
{
        Super::Tick(DeltaTime);
```

```
	UE_LOG(LogTemp, Warning, TEXT("Frame duration:%f"), DeltaTime);

	if (CurrentTime >= TotalExperimentDuration)
	{
		if (!bAddedLabels)
		{
			AddObservationLabels();
		}
	}
	else
	{
		AddObservation();
	}

	CurrentTime += DeltaTime;
}

void AExperimentalCubeOne::AddObservationLabels()
{
	// This function adds labels to the ObservationsArray and saves the content of the
	↪ beforementioned array to a .csv file.
	FString ObservationLabels = FString("Time") + "," + FString("X Velocity") + "," + FString("Y
	↪ Velocity") + "," + FString("Z Velocity") + "," +
		FString("X Position") + "," +
		↪ FString("Y Position") + "," +
		↪ FString("Z Position") + "," +
		↪ FString("Roll") + "," +
		↪ FString("Yaw") + "," +
		↪ FString("Pitch") + "," +
		FString("X Angular Velocity") + ","
		↪ + FString("Y Angular Velocity")
		↪ + "," + FString("Z Angular
		↪ Velocity");

	ObservationsArray.Insert(ObservationLabels, 0);
	ATextFileManager::SaveArrayText(UKismetSystemLibrary::GetProjectDirectory(), FileName,
	↪ ObservationsArray, true);

	// Modify the value of the auxiliary so that labels are added only once.
	bAddedLabels = true;
}

void AExperimentalCubeOne::AddObservation()
{
	// This function adds an Observation to the ObservationsArray.
	// Note that an Observations consists of the current time, the linear velocity, angular
	↪ velocity, position and rotation of the cube.
	FString Observation = "";
	FVector VelocityVector = CubeMesh->GetComponentVelocity();
	FVector PositionVector = GetActorLocation();
	FRotator RotationVector = GetActorRotation();
	FVector AngularVelocityVector = CubeMesh->GetPhysicsAngularVelocityInDegrees();

	Observation += FString::SanitizeFloat(CurrentTime) + ",";
	Observation += FString::SanitizeFloat(VelocityVector.X) + "," +
	↪ FString::SanitizeFloat(VelocityVector.Y) + "," +
	↪ FString::SanitizeFloat(VelocityVector.Z) + ",";
	Observation += FString::SanitizeFloat(PositionVector.X + DisplacementVector.X) + "," +
	↪ FString::SanitizeFloat(PositionVector.Y + DisplacementVector.Y) + "," +
	↪ FString::SanitizeFloat(PositionVector.Z + DisplacementVector.Z) + ",";
```

```
        Observation += FString::SanitizeFloat(RotationVector.Roll) + "," +
        ↪  FString::SanitizeFloat(RotationVector.Yaw) + "," +
        ↪  FString::SanitizeFloat(RotationVector.Pitch) + ",";
        Observation += FString::SanitizeFloat(AngularVelocityVector.X) += "," +
        ↪  FString::SanitizeFloat(AngularVelocityVector.Y) + "," +
        ↪  FString::SanitizeFloat(AngularVelocityVector.Z);

        ObservationsArray.Add(Observation);
}
```

C.2.3 Impulse Test Cube - header file

```
/*
        Copyright [2021] [-]

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at

                http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
 */

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ExperimentalCubeTwo.generated.h"

class UStaticMeshComponent;
class UMaterial;

/* This class is used to represent a Cube for experiments in which a force is applied only once. */
UCLASS()
class PHYSICSEXPERIMENTS_API AExperimentalCubeTwo : public AActor
{
        GENERATED_BODY()

protected:
        virtual void BeginPlay() override;

public:
        AExperimentalCubeTwo();

        virtual void Tick(float DeltaSeconds) override;

        /* Mesh component, used for rendering of the Cube. */
        UPROPERTY(VisibleAnywhere)
        UStaticMeshComponent* CubeMesh;

        /* Custom Material used for the Cube, note that in the original experiments, the Physical
        ↪  Material parameters were: Friction = 0.01, Restitution = 0.0.*/
        UPROPERTY(VisibleAnywhere)
```

```
        UMaterial* CustomMaterial;

        /* Function that adds an Observation to the ObservationsArray. */
        void AddObservation();

        /* Function that adds Labels to the existing observations and saves the ObservationsArray to
        ↪  a .csv file. */
        void AddObservationLabels();

        /* Array holding all the Observations and the Labels, its content will be saved as a .csv
        ↪  file. */
        TArray<FString> ObservationsArray;

        /* Variable holding the name of the file in which the Observations will be saved. */
        FString FileName = "Impulse_Observations.csv";

        /* Auxiliary boolean used to determine if labels were added or not. */
        bool bAddedLabels = false;

        /* Variable storing the vector representing the force that will be applied to the Cube*/
        UPROPERTY(EditAnywhere)
        FVector ForceVector;

        /* Variable storing an offset from the center of mass of the Cube, deciding where relative
        ↪  to the center of mass the force will be applied. */
        UPROPERTY(EditAnywhere)
        FVector ForceVectorOffset = FVector(0,0,0);

        /* Variable holding the experiment duration, therefore deciding the time period for which
        ↪  Observations will be taken. */
        UPROPERTY(EditAnywhere)
        float TotalExperimentDuration = 5.0f;

        /* Variable holding the time elapsed since the experiment began. */
        float CrtTime = 0.0f;

        /* Auxiliary boolean used to determine if the force was yet applied. */
        bool bHasAppliedForce = false;

        /* Auxiliary boolean used to determine if the experiment is based on the impulse approach or
        ↪  on the force approach. */
        UPROPERTY(EditAnywhere)
        bool bIsImpulseBased = false;

        /* Variable holding the displacement relative to the origin. This is needed so that all
        ↪  experiments can be ran at the same time in one scene, with the obtained results still
        ↪  being relative to the origin. */
        UPROPERTY(EditAnywhere)
        FVector DisplacementVector = FVector(0, 0, 0);
};
```

C.2.4 Impulse Test Cube - source file

```
/*
        Copyright [2021] [-]

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
```

```
        You may obtain a copy of the License at

                http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
 */

#include "ExperimentalCubeTwo.h"
#include "Components/BoxComponent.h"
#include "Components/StaticMeshComponent.h"
#include "UObject/ConstructorHelpers.h"
#include "TextFileManager.h"
#include "Kismet/KismetSystemLibrary.h"
#include "Engine/StaticMesh.h"

AExperimentalCubeTwo::AExperimentalCubeTwo()
{
        // Create and attach a Static Mesh to the Cube.
        CubeMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Cube mesh"));
        CubeMesh->SetupAttachment(RootComponent);
        CubeMesh->SetStaticMesh(ConstructorHelpers::FObjectFinder<UStaticMesh>(TEXT("/Game/Physics_Experiments/Shapes/Sha

        // Initialize basic Physics properties for the Mesh.
        CubeMesh->SetSimulatePhysics(true);
        CubeMesh->SetMassOverrideInKg();

        PrimaryActorTick.bCanEverTick = true;
}

void AExperimentalCubeTwo::BeginPlay()
{
        Super::BeginPlay();
        DisplacementVector = GetActorLocation();
}

void AExperimentalCubeTwo::AddObservation()
{
        // This function adds an Observation to the ObservationsArray.
        // Note that an Observations consists of the current time, the linear velocity, angular
        ↪  velocity, position and rotation of the cube and the force applied to the cube.
        FString Observation = "";
        FVector VelocityVector = CubeMesh->GetComponentVelocity();
        FVector PositionVector = GetActorLocation();
        FRotator RotationVector = GetActorRotation();
        FVector AngularVelocityVector = CubeMesh->GetPhysicsAngularVelocityInDegrees();

        // Note that there is a 0.5 seconds delay to account for any initialization that UE4
        ↪  performs.
        Observation += FString::SanitizeFloat(CrtTime - 0.5) + ",";
        Observation += FString::SanitizeFloat(VelocityVector.X) + "," +
        ↪  FString::SanitizeFloat(VelocityVector.Y) + "," +
        ↪  FString::SanitizeFloat(VelocityVector.Z) + ",";
        Observation += FString::SanitizeFloat(PositionVector.X - DisplacementVector.X) + "," +
        ↪  FString::SanitizeFloat(PositionVector.Y - DisplacementVector.Y) + "," +
        ↪  FString::SanitizeFloat(PositionVector.Z - DisplacementVector.Z) + ",";
```

```
	Observation += FString::SanitizeFloat(RotationVector.Roll) + "," +
	↪  FString::SanitizeFloat(RotationVector.Yaw) + "," +
	↪  FString::SanitizeFloat(RotationVector.Pitch) + ",";
	Observation += FString::SanitizeFloat(AngularVelocityVector.X) += "," +
	↪  FString::SanitizeFloat(AngularVelocityVector.Y) + "," +
	↪  FString::SanitizeFloat(AngularVelocityVector.Z) + ",";
	Observation += FString::SanitizeFloat(ForceVector.X) += "," +
	↪  FString::SanitizeFloat(ForceVector.Y) + "," + FString::SanitizeFloat(ForceVector.Z);

	ObservationsArray.Add(Observation);
}

void AExperimentalCubeTwo::AddObservationLabels()
{
	// This function adds labels to the ObservationsArray and saves the content of the
	↪  beforementioned array to a .csv file.
	FString ObservationLabels = FString("Time") + "," + FString("X Velocity") + "," + FString("Y
	↪  Velocity") + "," + FString("Z Velocity") + "," +
	                                                    FString("X Position") + "," +
	                                                    ↪  FString("Y Position") + "," +
	                                                    ↪  FString("Z Position") + "," +
	                                                    ↪  FString("Roll") + "," +
	                                                    ↪  FString("Yaw") + "," +
	                                                    ↪  FString("Pitch") + "," +
	                                                    FString("X Angular Velocity") + ","
	                                                    ↪  + FString("Y Angular Velocity")
	                                                    ↪  + "," + FString("Z Angular
	                                                    ↪  Velocity") + "," +
	                                                    FString("X Force Applied") + "," +
	                                                    ↪  FString("Y Force Applied") + ","
	                                                    ↪  + FString("Z Force Applied");

	ObservationsArray.Insert(ObservationLabels, 0);
	ATextFileManager::SaveArrayText(UKismetSystemLibrary::GetProjectDirectory(), FileName,
	↪  ObservationsArray, true);

	// Modify the value of the auxiliary so that labels are added only once.
	bAddedLabels = true;
}

void AExperimentalCubeTwo::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
	FVector Inertia = CubeMesh->GetInertiaTensor();
	UE_LOG(LogTemp, Warning, TEXT("Inertia:%f,%f,%f"), Inertia.X, Inertia.Y, Inertia.Z);

	// Note that there is a 0.5 seconds delay to account for any initialization that UE4
	↪  performs.
	if (CrtTime >= TotalExperimentDuration + 0.5f)
	{
		if (!bAddedLabels)
		{
			AddObservationLabels();
		}
	}
	else
	{
		if (CrtTime > 0.5f)
		{
			if (bHasAppliedForce) {
```

```
                        AddObservation();
                }

                if (/*CrtTime > 1.0f &&*/ !bHasAppliedForce)
                {
                        if (bIsImpulseBased)
                        {
                                // The constant 0.001 comes from the frame duration of
                                ↪  Gazebo, so that the actual forces applied match in both
                                ↪  Engines.
                                CubeMesh->AddImpulseAtLocation(0.001 * ForceVector,
                                ↪  CubeMesh->GetCenterOfMass() + ForceVectorOffset);
                        }
                        else
                        {
                                CubeMesh->AddForceAtLocation(ForceVector,
                                ↪  CubeMesh->GetCenterOfMass() + ForceVectorOffset);
                        }

                        // Modify the value of the auxiliary so that the force is applied
                        ↪  only once.
                        bHasAppliedForce = true;
                }
        }
    }

    CrtTime += DeltaTime;
}
```

C.2.5 Force Patterns Test Cube - header file

```
/*
    Copyright [2021] [-]

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
 */

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ExperimentalCubeFour.generated.h"

class UStaticMeshComponent;

/* This class is used to represent a Cube for experiments in which force patterns are applied to it.
↪  */
UCLASS()
```

```
class PHYSICSEXPERIMENTS_API AExperimentalCubeFour : public AActor
{
	GENERATED_BODY()

protected:
	virtual void BeginPlay() override;

public:
	AExperimentalCubeFour();

	virtual void Tick(float DeltaSeconds) override;

	/* Mesh component, used for rendering of the Cube. */
	UPROPERTY(VisibleAnywhere)
	UStaticMeshComponent* CubeMesh;

	/* Function that adds an Observation to the ObservationsArray. */
	void AddObservation();

	/* Function that adds Labels to the existing observations and saves the ObservationsArray to
	↪  a .csv file. */
	void AddObservationLabels();

	/* Variable holding the name of the file in which the Observations will be saved. */
	FString FileName = "Force_Observations.csv";

	/* Array holding all the Observations and the Labels, its content will be saved as a .csv
	↪  file. */
	TArray<FString> ObservationsArray;

	/* Auxiliary boolean used to determine if labels were added or not. */
	bool bAddedLabels = false;

	/* Variable holding the experiment duration, therefore deciding the time period for which
	↪  Observations will be taken. */
	UPROPERTY(EditAnywhere)
	float TotalExperimentDuration = 7.0f;

	/* Variable holding the time elapsed since the experiment began. */
	float CrtTime = 0.0f;

	/* Variable storing the vector representing the force that will be applied to the Cube*/
	FVector ForceVector;

	/* Variable storing an offset from the center of mass of the Cube, deciding where relative
	↪  to the center of mass the force will be applied. */
	UPROPERTY(EditAnywhere)
	FVector ForceVectorOffset = FVector(0, 0, 0);

	/* Variable holding the force pattern in use. Note that it can be modified from the Editor,
	↪  therefore running different patterns do not require building the project multiple times.
	↪  */
	UPROPERTY(EditAnywhere)
	int CurrentForcePattern = 3;

	/* Auxiliary boolean deciding if the experiment is time or tick based. Note that it can be
	↪  modified from the Editor, switching between time and tick-based experiments do not
	↪  require building the project multiple times*/
	UPROPERTY(EditAnywhere)
	bool bIsTimeBased = true;
```

```
        /* Function that computes the Force that needs to be added at every timestamp, bsased on the
        ↪  CurrentForcePattern. */
        FVector ComputeForceVector(int PatternNumber, float CurrentTime);

        /* Variable holding the duration of the CurrentForcePattern, deciding how long a force is
        ↪  going to be applied to the Cube. */
        UPROPERTY(EditAnywhere)
        float PatternDuration = 5.0f;

        /* Function that computes the Force that needs to be added at every tick, based on the
        ↪  CurrentForcePattern. */
        FVector ComputeTickBasedForce(int PatternNumber, int CurrentIteration);

        /* Function used for Physics Sub-Stepping and tick-based implementation of the force
        ↪  patterns. */
        virtual void PhysicsTick_Implementation(float SubstepDeltaTime);

        /* Delegate used for Physics Sub-stepping. */
        FCalculateCustomPhysics OnCalculateCustomPhysics;

        /* Function used for Physics Sub-Stepping. */
        UFUNCTION(BlueprintNativeEvent)
        void PhysicsTick(float DeltaTime);

        /* Function used for Physics Sub-stepping and tick-based implementation of the force
        ↪  patterns. */
        void CustomPhysics(float DeltaTime, FBodyInstance* BodyInstance);

        /* Variable holding the number of iterations executed since the experiment began. */
        int CrtIteration = 0;

        /* Variable holding the total number of iterations of the CurrentForcePattern, deciding how
        ↪  many ticks a force is going to be applied to the Cube. */
        UPROPERTY(EditAnywhere)
        int TotalPatternIterations = 5000;

        /* Variable holding the displacement relative to the origin. This is needed so that all
        ↪  experiments can be ran at the same time in one scene, with the obtained results still
        ↪  being relative to the origin. */
        FVector DisplacementVector = FVector(0, 0, 0);

        /* Variable holding the constant force to apply when performing the constant force
        ↪  experiment. */
        UPROPERTY(EditAnywhere)
        float ConstantForce = 20;
};
```

C.2.6 Force Patterns Test Cube - source file

```
/*
        Copyright [2021] [-]

        Licensed under the Apache License, Version 2.0 (the "License");
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at

                http://www.apache.org/licenses/LICENSE-2.0
```

```
        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
 */

#include "ExperimentalCubeFour.h"
#include "Components/BoxComponent.h"
#include "Components/StaticMeshComponent.h"
#include "UObject/ConstructorHelpers.h"
#include "TextFileManager.h"
#include "Kismet/KismetSystemLibrary.h"
#include "Engine/StaticMesh.h"
#include <math.h>

AExperimentalCubeFour::AExperimentalCubeFour()
{
        // Create and attach a Static Mesh to the Cube.
        CubeMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Cube mesh"));
        CubeMesh->SetupAttachment(RootComponent);
        CubeMesh->SetStaticMesh(ConstructorHelpers::FObjectFinder<UStaticMesh>(TEXT("/Game/Physics_Experiments/Shapes/Sha

        // Initialize basic Physics properties for the Mesh.
        CubeMesh->SetSimulatePhysics(true);
        CubeMesh->SetMassOverrideInKg();

        PrimaryActorTick.bCanEverTick = true;

        // Bind the delegate used for Physics Sub-Stepping.
        OnCalculateCustomPhysics.BindUObject(this, &AExperimentalCubeFour::CustomPhysics);
}

void AExperimentalCubeFour::BeginPlay()
{
        Super::BeginPlay();
        DisplacementVector = GetActorLocation();
}

void AExperimentalCubeFour::AddObservationLabels()
{
        // This function adds labels to the ObservationsArray and saves the content of the
        ↪  beforementioned array to a .csv file.
        FString ObservationLabels = FString("Time") + "," + FString("X Velocity") + "," + FString("Y
        ↪  Velocity") + "," + FString("Z Velocity") + "," +
                                                                FString("X Position") + "," +
                                                                ↪  FString("Y Position") + "," +
                                                                ↪  FString("Z Position") + "," +
                                                                ↪  FString("Roll") + "," +
                                                                ↪  FString("Yaw") + "," +
                                                                ↪  FString("Pitch") + "," +
                                                                FString("X Angular Velocity") + ","
                                                                ↪  + FString("Y Angular Velocity")
                                                                ↪  + "," + FString("Z Angular
                                                                ↪  Velocity") + "," +
                                                                FString("X Force Applied") + "," +
                                                                ↪  FString("Y Force Applied") + ","
                                                                ↪  + FString("Z Force Applied");
```

```
        ObservationsArray.Insert(ObservationLabels, 0);
        ATextFileManager::SaveArrayText(UKismetSystemLibrary::GetProjectDirectory(), FileName,
        ↪  ObservationsArray, true);

        // Modify the value of the auxiliary so that labels are added only once.
        bAddedLabels = true;
}

void AExperimentalCubeFour::AddObservation()
{
        // This function adds an Observation to the ObservationsArray.
        // Note that an Observations consists of the current time, the linear velocity, angular
        ↪  velocity, position and rotation of the cube and the force applied to the cube.
        FString Observation = "";
        FVector VelocityVector = CubeMesh->GetComponentVelocity();
        FVector PositionVector = GetActorLocation();
        FRotator RotationVector = GetActorRotation();
        FVector AngularVelocityVector = CubeMesh->GetPhysicsAngularVelocityInDegrees();

        // Note that there is a 0.5 seconds delay to account for any initialization that UE4 might
        ↪  perform and that might lead to unexpected behaviour.
        Observation += FString::SanitizeFloat(CrtTime - 0.5f) + ",";
        Observation += FString::SanitizeFloat(VelocityVector.X) + "," +
        ↪  FString::SanitizeFloat(VelocityVector.Y) + "," +
        ↪  FString::SanitizeFloat(VelocityVector.Z) + ",";
        Observation += FString::SanitizeFloat(PositionVector.X - DisplacementVector.X) + "," +
        ↪  FString::SanitizeFloat(PositionVector.Y - DisplacementVector.Y) + "," +
        ↪  FString::SanitizeFloat(PositionVector.Z - DisplacementVector.Z) + ",";
        Observation += FString::SanitizeFloat(RotationVector.Roll) + "," +
        ↪  FString::SanitizeFloat(RotationVector.Yaw) + "," +
        ↪  FString::SanitizeFloat(RotationVector.Pitch) + ",";
        Observation += FString::SanitizeFloat(AngularVelocityVector.X) += "," +
        ↪  FString::SanitizeFloat(AngularVelocityVector.Y) + "," +
        ↪  FString::SanitizeFloat(AngularVelocityVector.Z) + ",";
        Observation += FString::SanitizeFloat(ForceVector.X) += "," +
        ↪  FString::SanitizeFloat(ForceVector.Y) + "," + FString::SanitizeFloat(ForceVector.Z);

        ObservationsArray.Add(Observation);
}

FVector ComputeTickPatternOne(int CurrentIteration, int TotalIterations)
{
        // This function computes and returns the force vector that should be applied every
        ↪  iteration of the physics engine according to Pattern 1.
        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentIteration < TotalIterations)
        {
                float yForce = 0.0f;

                if (CurrentIteration <= 2000)
                {
                        yForce = 0.005 * CurrentIteration;
                }
                else if (CurrentIteration <= 3000)
                {
                        yForce = 10;
                }
                else if (CurrentIteration <= 5000)
                {
```

```
                        yForce = 10 - 0.005 * (CurrentIteration - 3000);
                }

                AppliedForceVector = FVector(0, yForce, 0);
        }

        return AppliedForceVector;
}

FVector ComputeTickPatternTwo(int CurrentIteration, int TotalIterations)
{
        // This function computes and returns the force vector that should be applied every
        ↪  iteration of the physics engine according to Pattern 2.

        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentIteration < TotalIterations)
        {
                float yForce = 0.0f;

                if (CurrentIteration < 100)
                {
                        yForce = CurrentIteration * 1.0f / 10;
                }
                else if (CurrentIteration < 4900)
                {
                        yForce = 10;
                }
                else if (CurrentIteration < 5000)
                {
                        yForce = 10 - (CurrentIteration - 4900) * 1.0f / 10;
                }

                AppliedForceVector = FVector(0, yForce, 0);
        }

        return AppliedForceVector;
}

FVector ComputeTickPatternThree(int CurrentIteration, int TotalIterations)
{
        // This function computes and returns the force vector that should be applied every
        ↪  iteration of the physics engine according to Pattern 3.

        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentIteration < TotalIterations)
        {
                float yForce = 0.0f;

                // Note that Pattern 3 consists of applying the same sub-pattern 3 times.
                // Therefore, it can be computed more easily by 'shifting' the CurrentIteration and
                ↪  only providing the rules for the first time the sub-pattern is executed.
                if (CurrentIteration >= 500 && CurrentIteration < 1000)
                {
                        // Second repetition of the same sub-pattern.
                        CurrentIteration -= 500;
                }
                else if (CurrentIteration >= 1000 && CurrentIteration < 1500)
                {
```

```
                        // Third repetition of the same sub-pattern.
                        CurrentIteration -= 1000;
                }

                // Rules for the execution of the sub-pattern.
                if (CurrentIteration < 100)
                {
                        yForce = 0.2 * CurrentIteration;
                }
                else if (CurrentIteration < 400)
                {
                        yForce = 20;
                }
                else if (CurrentIteration < 500)
                {
                        yForce = 20 - 0.2 * (CurrentIteration - 400);
                }

                AppliedForceVector = FVector(0, yForce, 0);
        }

        return AppliedForceVector;
}

FVector ComputeTickPatternFour(int CurrentIteration, int TotalIterations)
{
        // This function computes and returns the force vector that should be applied every
        ↪  iteration of the physics engine according to Pattern 4.

        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentIteration < TotalIterations)
        {
                float yForce = 20 * abs(sin((CurrentIteration * PI / 1000)));

                AppliedForceVector = FVector(0, yForce, 0);
        }

        return AppliedForceVector;
}

FVector ComputeTickConstantForce(int CurrentIteration, int TotalPatternIterations, float
↪  ConstantForce)
{
        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentIteration < TotalPatternIterations)
        {
                AppliedForceVector =  FVector(0, ConstantForce, 0);
        }

        return AppliedForceVector;
}

FVector AExperimentalCubeFour::ComputeTickBasedForce(int PatternNumber, int CurrentIteration)
{
        // This function computes and returns the force vector that should be applied every
        ↪  iteration according to any of the 4 existing patterns.
        switch (PatternNumber)
        {
```

```
                case 0:
                        return ComputeTickConstantForce(CurrentIteration, TotalPatternIterations,
                        ↪  ConstantForce);
                case 1:
                        return ComputeTickPatternOne(CurrentIteration, TotalPatternIterations);
                case 2:
                        return ComputeTickPatternTwo(CurrentIteration, TotalPatternIterations);
                case 3:
                        return ComputeTickPatternThree(CurrentIteration, TotalPatternIterations);
                case 4:
                        return ComputeTickPatternFour(CurrentIteration, TotalPatternIterations);
                default:
                        return FVector(0, 0, 0);
        }
}

void AExperimentalCubeFour::PhysicsTick_Implementation(float SubstepDeltaTime)
{
        // This functions applies the force vector to the cube.
        ForceVector = ComputeTickBasedForce(CurrentForcePattern, CrtIteration);
        CubeMesh->AddImpulseAtLocation(ForceVector * SubstepDeltaTime, CubeMesh->GetCenterOfMass() +
        ↪  ForceVectorOffset);
        CrtIteration++;
}

void AExperimentalCubeFour::CustomPhysics(float DeltaTime, FBodyInstance* BodyInstance)
{
        // Note that there is a 0.5 seconds delay to account for any initialization that UE4
        ↪  performs and that might lead to unexpected behaviour.
        if (CrtTime > 0.5f)
        {
                UE_LOG(LogTemp, Warning, TEXT("Frame duration:%f\n"), DeltaTime);
                PhysicsTick(DeltaTime);
                AddObservation();
        }
        CrtTime += DeltaTime;
}

FVector ComputeForcePatternOne(float CurrentPatternTime, float PatternDuration)
{
        // This function computes and returns the force vector that should be applied at every
        ↪  timestamp according to Pattern 1.

        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentPatternTime <= PatternDuration)
        {
                float yForce = 0.0f;

                if (CurrentPatternTime <= 2.0f)
                {
                        yForce = 5 * CurrentPatternTime;
                }
                else if (CurrentPatternTime <= 3.0f)
                {
                        yForce = 10;
                }
                else if (CurrentPatternTime <= 5.0f)
                {
                        yForce = 10 - 5 * (CurrentPatternTime - 3.0f);
```

```
                }

                AppliedForceVector = FVector(0, yForce, 0);
        }

        return AppliedForceVector;
}

FVector ComputeForcePatternTwo(float CurrentPatternTime, float PatternDuration)
{
        // This function computes and returns the force vector that should be applied at every
        ↪  timestamp according to Pattern 2.

        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentPatternTime <= PatternDuration)
        {
                float yForce = 0.0f;

                if (CurrentPatternTime < 0.1f)
                {
                        yForce = 100 * CurrentPatternTime;
                }
                else if (CurrentPatternTime < 4.9f)
                {
                        yForce = 10;
                }
                else if (CurrentPatternTime <= 5.0f)
                {
                        yForce = 10 - 100 * (CurrentPatternTime - 4.9f);
                }

                AppliedForceVector = FVector(0, yForce, 0);
        }

        return AppliedForceVector;
}

FVector ComputeForcePatternThree(float CurrentPatternTime, float PatternDuration)
{
        // This function computes and returns the force vector that should be applied at every
        ↪  timestamp according to Pattern 3.

        FVector AppliedForceVector = FVector(0, 0, 0);

        if (CurrentPatternTime <= PatternDuration)
        {
                float yForce = 0.0f;

                // Note that Pattern 3 consists of applying the same sub-pattern 3 times.
                // Therefore, it can be computed more easily by 'shifting' the CurrentPatternTime
                ↪  and only providing the rules for the first time the sub-pattern is executed.
                if (CurrentPatternTime >= 0.5f && CurrentPatternTime < 1.0f)
                {
                        // Second repetition of the same sub-pattern.
                        CurrentPatternTime -= 0.5f;
                }
                else if (CurrentPatternTime >= 1.0f && CurrentPatternTime < 1.5f)
                {
                        // Third repetition of the same sub-pattern.
```

```
                CurrentPatternTime -= 1.0f;
        }

        // Rules for the execution of the sub-pattern.
        if (CurrentPatternTime < 0.1f)
        {
                yForce = 200 * CurrentPatternTime + 0.2;
        }
        else if (CurrentPatternTime < 0.4f)
        {
                yForce = 20;
        }
        else if (CurrentPatternTime < 0.5f)
        {
                yForce = 20 - 200 * (CurrentPatternTime - 0.4f);
        }

        AppliedForceVector = FVector(0, yForce, 0);
    }

    return AppliedForceVector;
}

FVector ComputeForcePatternFour(float CurrentPatternTime, float PatternDuration)
{
    // This function computes and returns the force vector that should be applied at every
    ↪  timestamp according to Pattern 4.

    FVector AppliedForceVector = FVector(0, 0, 0);

    if (CurrentPatternTime <= PatternDuration)
    {
        float yForce = 20 * abs(sin(PI * CurrentPatternTime));
        AppliedForceVector = FVector(0, yForce, 0);
    }

    return AppliedForceVector;
}

FVector ComputeTimeConstantForce(float CurrentTime, float PatternDuration, float ConstantForce)
{
    FVector AppliedForceVector = FVector(0, 0, 0);

    if (CurrentTime < PatternDuration)
    {
        AppliedForceVector =  FVector(0, ConstantForce, 0);
    }

    return AppliedForceVector;
}

FVector AExperimentalCubeFour::ComputeForceVector(int PatternNumber, float CurrentTime)
{
    // This function computes and returns the force vector that should be applied every
    ↪  timestamp according to any of the 4 existing patterns.

    switch (PatternNumber)
    {
        case 0:
```

```
                        return ComputeTimeConstantForce(CurrentTime, PatternDuration,
                        ↪  ConstantForce);
                case 1:
                        return ComputeForcePatternOne(CurrentTime, PatternDuration);
                case 2:
                        return ComputeForcePatternTwo(CurrentTime, PatternDuration);
                case 3:
                        return ComputeForcePatternThree(CurrentTime, PatternDuration);
                case 4:
                        return ComputeForcePatternFour(CurrentTime, PatternDuration);
                default:
                        return FVector(0, 0, 0);
        }
}

void AExperimentalCubeFour::Tick(float DeltaTime)
{
        Super::Tick(DeltaTime);
        UE_LOG(LogTemp, Warning, TEXT("Frame duration:%f"), DeltaTime);
        // Note that there is a 0.5 seconds delay to account for any initialization that UE4
        ↪  performs.
        if (CrtTime >= TotalExperimentDuration + 0.5f)
        {
                if (!bAddedLabels)
                {
                        AddObservationLabels();
                }
        }
        else
        {
                if (bIsTimeBased)
                {
                        CrtTime += DeltaTime;
                        if (CrtTime > 0.5f)
                        {
                                AddObservation();

                                // Note that the CrtTime - 0.5 seconds is passed as argument because
                                ↪  of the 0.5 seconds delay mentioned throughout the code.
                                ForceVector = ComputeForceVector(CurrentForcePattern, CrtTime -
                                ↪  0.5f);
                                CubeMesh->AddForceAtLocation(ForceVector,
                                ↪  CubeMesh->GetCenterOfMass() + ForceVectorOffset);
                        }
                }
                else
                {
                        CubeMesh->GetBodyInstance()->AddCustomPhysics(OnCalculateCustomPhysics);
                }
        }
}
```

C.3 Chapter 3 Gazebo Code

C.3.1 Plugin for the Force Patterns

```
/*
    Copyright [2021] [-]

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
 */

#include <functional>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include <ignition/math/Vector3.hh>
#include <fstream>
#include <math.h>
#define PI 3.14159265

namespace gazebo
{
  class ForcePattern : public ModelPlugin
  {

    public: void Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/)
    {
      // Store a pointer to the model
      this->model = _parent;

      // Listen to the update event. This event is broadcast every
      // simulation iteration.
      this->updateConnection = event::Events::ConnectWorldUpdateBegin(
          std::bind(&ForcePattern::OnUpdate, this));

      AddForceObservationLabels();
    }

    // Called by the world update start event
    public: void OnUpdate()
    {
        const physics::WorldPtr& worldPtr = model->GetWorld();
                common::Time simulationTime = worldPtr->SimTime();
                double simTime = simulationTime.Double();

        if(isTimeBased)
        {
            ComputeForceVector(currentForcePattern, simTime);
        }
        else
```

```
    {
        ComputeTickBasedForceVector(currentForcePattern);
        crtIteration ++;
    }

    this->model->GetLink("base_link")->AddForce(appliedForceVector);
    AddForceObservation(simTime);

    if (simTime >= totalExperimentDuration)
    {
        outputFile.close();
    }
}

public: void AddForceObservationLabels()
{
    // This function creates a .csv file and adds the first row consisting of the observations
    ↪  labels to that file.
    outputFile.open("appliedForce.csv");
    outputFile<<"sim_time, applied_force_x, applied_force_y, applied_force_z\n";
}

public: void AddForceObservation(double simTime)
{
    // This function adds a new observation to the .csv file previously opened.
    // Note that an observation simply consists of the current simulation time and the values of
    ↪  the appliedForceVector.
    outputFile << simTime << "," << appliedForceVector[0] << "," << appliedForceVector[1] << ","
    ↪  << appliedForceVector[2] << "\n";
}

private: void ComputeForceVector(int patternNumber, double currentTime)
{
    switch(patternNumber)
    {
        case 0:
            ComputeTimeConstantForce(currentTime);
            break;
        case 1:
            ComputeForcePatternOne(currentTime);
            break;
        case 2:
            ComputeForcePatternTwo(currentTime);
            break;
        case 3:
            ComputeForcePatternThree(currentTime);
            break;
        case 4:
            ComputeForcePatternFour(currentTime);
            break;
        default:
            appliedForceVector = ignition::math::Vector3d(0, 0, 0);
            break;
    }
}

private: void ComputeTickBasedForceVector(int patternNumber)
{
    // This function computes and returns the force vector that should be applied every
    ↪  iteration according to any of the 4 existing patterns.
```

```
    switch (patternNumber)
    {
        case 0:
            ComputeTickConstantForce();
            break;
        case 1:
            ComputeTickPatternOne();
            break;
        case 2:
            ComputeTickPatternTwo();
            break;
        case 3:
            ComputeTickPatternThree();
            break;
        case 4:
            ComputeTickPatternFour();
            break;
        default:
            appliedForceVector = ignition::math::Vector3d(0, 0, 0);
            break;
    }
}

private: void ComputeTimeConstantForce(double currentPatternTime)
{
    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (currentPatternTime <= patternDuration)
    {
        appliedForceVector = ignition::math::Vector3d(0, constantForceValue, 0);
    }
}

private: void ComputeForcePatternOne(double currentPatternTime)
{
    // This function computes and returns the force vector that should be applied at every
    ↪  timestamp according to Pattern 1.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);
    if (currentPatternTime <= patternDuration)
    {
        float yForce = 0.0f;

        if (currentPatternTime <= 2.0f)
        {
            yForce = 5 * currentPatternTime;
        }
        else if (currentPatternTime <= 3.0f)
        {
            yForce = 10;
        }
        else if (currentPatternTime <= 5.0f)
        {
            yForce = 10 - 5 * (currentPatternTime - 3.0f);
        }

        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}
```

```
private: void ComputeForcePatternTwo(double currentPatternTime)
{
    // This function computes and returns the force vector that should be applied at every
    ↪  timestamp according to Pattern 2.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (currentPatternTime <= patternDuration)
    {
        float yForce = 0.0f;

        if (currentPatternTime < 0.1f)
        {
            yForce = 100 * currentPatternTime;
        }
        else if (currentPatternTime < 4.9f)
        {
            yForce = 10;
        }
        else if (currentPatternTime <= 5.0f)
        {
            yForce = 10 - 100 * (currentPatternTime - 4.9f);
        }

        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

private: void ComputeForcePatternThree(double currentPatternTime)
{
    // This function computes and returns the force vector that should be applied at every
    ↪  timestamp according to Pattern 3.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (currentPatternTime <= patternDuration)
    {
        float yForce = 0.0f;

        // Note that Pattern 3 consists of applying the same sub-pattern 3 times.
        // Therefore, it can be computed more easily by 'shifting' the currentPatternTime and
        ↪  only providing the rules for the first time the sub-pattern is executed.
        if (currentPatternTime >= 0.5f && currentPatternTime < 1.0f)
        {
            // Second repetition of the same sub-pattern.
            currentPatternTime -= 0.5f;
        }
        else if (currentPatternTime >= 1.0f && currentPatternTime < 1.5f)
        {
            // Third repetition of the same sub-pattern.
            currentPatternTime -= 1.0f;
        }

        // Rules for the execution of the sub-pattern.
        if (currentPatternTime < 0.1f)
        {
            yForce = 200 * currentPatternTime + 0.2;
        }
        else if (currentPatternTime < 0.4f)
```

```
        {
            yForce = 20;
        }
        else if (currentPatternTime < 0.5f)
        {
            yForce = 20 - 200 * (currentPatternTime - 0.4f);
        }

        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

private: void ComputeForcePatternFour(double currentPatternTime)
{
    // This function computes and returns the force vector that should be applied at every
    ↪  timestamp according to Pattern 4.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (currentPatternTime <= patternDuration)
    {
        float yForce = 20 * abs(sin(PI * currentPatternTime));
        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

private: void ComputeTickConstantForce()
{
    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (crtIteration < totalIterations)
    {
        appliedForceVector = ignition::math::Vector3d(0, constantForceValue, 0);
    }
}

private: void ComputeTickPatternOne()
{
    // This function computes and returns the force vector that should be applied every
    ↪  iteration of the physics engine according to Pattern 1.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (crtIteration < totalIterations)
    {
        float yForce = 0.0f;

        if (crtIteration <= 2000)
        {
            yForce = 0.005 * crtIteration;
        }
        else if (crtIteration <= 3000)
        {
            yForce = 10;
        }
        else if (crtIteration <= 5000)
        {
            yForce = 10 - 0.005 * (crtIteration - 3000);
        }
```

```
        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

private: void ComputeTickPatternTwo()
{
    // This function computes and returns the force vector that should be applied every
    ↪  iteration of the physics engine according to Pattern 2.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (crtIteration < totalIterations)
    {
        float yForce = 0.0f;

        if (crtIteration < 100)
        {
            yForce = crtIteration * 1.0f / 10;
        }
        else if (crtIteration < 4900)
        {
            yForce = 10;
        }
        else if (crtIteration < 5000)
        {
            yForce = 10 - (crtIteration - 4900) * 1.0f / 10;
        }

        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

private: void ComputeTickPatternThree()
{
    // This function computes and returns the force vector that should be applied every
    ↪  iteration of the physics engine according to Pattern 3.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (crtIteration < totalIterations)
    {
        float yForce = 0.0f;

        // Note that the observation made below, together with the fact that crtIteration is a
        ↪  variable of the class leads to the need of an auxiliary variable.
        int auxiliary = crtIteration;

        // Note that Pattern 3 consists of applying the same sub-pattern 3 times.
        // Therefore, it can be computed more easily by 'shifting' the crtIteration and only
        ↪  providing the rules for the first time the sub-pattern is executed.
        if (auxiliary >= 500 && auxiliary < 1000)
        {
            // Second repetition of the same sub-pattern.
            auxiliary -= 500;
        }
        else if (auxiliary >= 1000 && auxiliary < 1500)
        {
            // Third repetition of the same sub-pattern.
            auxiliary -= 1000;
        }
```

```
        // Rules for the execution of the sub-pattern.
        if (auxiliary < 100)
        {
            yForce = 0.2 * auxiliary;
        }
        else if (auxiliary < 400)
        {
            yForce = 20;
        }
        else if (auxiliary < 500)
        {
            yForce = 20 - 0.2 * (auxiliary - 400);
        }

        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

private: void ComputeTickPatternFour()
{
    // This function computes and returns the force vector that should be applied every
    ↪  iteration of the physics engine according to Pattern 4.

    appliedForceVector = ignition::math::Vector3d(0, 0, 0);

    if (crtIteration < totalIterations)
    {
        float yForce = 20 * abs(sin((crtIteration * PI / 1000)));

        appliedForceVector = ignition::math::Vector3d(0, yForce, 0);
    }
}

// Pointer to the model.
private: physics::ModelPtr model;

// Pointer to the update event connection.
private: event::ConnectionPtr updateConnection;

// Variable holding the force pattern in use.
private: int currentForcePattern = 0;

// Variable holding the number of iterations executed since the experiment began.
private: int crtIteration = 0;

// Variable holding the total number of iterations of the CurrentForcePattern, deciding how many
↪  ticks a force is going to be applied to the Cube.
private: int totalIterations = 5000;

// Variable holding the duration of the currentForcePattern, deciding how long a force is going
↪  to be applied to the Cube.
private: double patternDuration = 5.0f;

// Variable holding the experiment duration, therefore deciding the time period for which
↪  Observations will be taken.
private: double totalExperimentDuration = 7.0f;

// Variable storing the vector representing the force that will be applied to the Cube.
private: ignition::math::Vector3d appliedForceVector = ignition::math::Vector3d(0, 0, 0);
```

```
    // Stream representing the output file in which Observations are written.
    private: std::ofstream outputFile;

    // Auxiliary boolean deciding if the experiment is time or tick based.
    private: bool isTimeBased = false;

    // Variable holding the value of the constant force that will be applied.
    private: float constantForceValue = 0;
  };

  // Register this plugin with the simulator
  GZ_REGISTER_MODEL_PLUGIN(ForcePattern)
}
```

C.3.2 World File

```
<!--
    Copyright [2021] [-]

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
-->

<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">
    <physics type='ode'>
      <!-- Note that the default Gazebo frame duration is 0.001, but the new value comes from the
      ↪  UE4 sub-stepping frame duration -->
      <max_step_size>0.001224</max_step_size>
      <ode>
        <solver>
          <!-- Note that the default solver is quick, but world is more accurate, removing the
          ↪  unexpected behaviour noticed during the experiments -->
          <type>world</type>
        </solver>
      </ode>
    </physics>

    <!-- Ground Plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- Using a sun as a source of light -->
    <include>
      <uri>model://sun</uri>
    </include>
```

```
    <!-- Model used to represent the Cube object that force is being applied to -->
    <model name="box">
      <!-- Note that the box is placed to sit on the ground plane, that is why the 0.5 displacement
      ↪  is needed -->
      <pose>0 0 0.5 0 0 0</pose>
      <link name="base_link">
        <inertial>
          <inertia>
            <!-- The inertia matrix is computed based on the a cube with a mass of 1kg -->
            <ixx> 1666.666382</ixx>
            <ixy>0</ixy>
            <ixz>0</ixz>
            <iyy>1666.666382</iyy>
            <iyz>0</iyz>
            <izz>1666.666382</izz>
          </inertia>
        </inertial>
        <collision name="collision">
          <geometry>
            <box>
              <!-- Note that the size of the box is 1x1x1 meters -->
              <size>1 1 1</size>
            </box>
          </geometry>
          <surface>
            <friction>
              <ode>
                <mu>1</mu>
                <mu2>1</mu2>
              </ode>
          </friction>
          </surface>
        </collision>

        <visual name="visual">
          <geometry>
            <box>
              <size>1 1 1</size>
            </box>
          </geometry>
        </visual>
      </link>

      <plugin name="force_pattern" filename="libforce_pattern.so"/>
    </model>
  </world>
</sdf>
```