Vrije Universiteit Amsterdam

BSc. Artificial Intelligence
Intelligent Systems Track

Simulating Robotics in an Underwater Environment using Gazebo and Revolve

Author:
Ravi Bossema (2640682)

Supervisor:
Dr. Eliseo Ferrante

Second reader:
Dr. Karine Miras

March 13, 2022

Abstract

Due to budget and resource limitations, the development and prototyping of underwater robotics is not viable in real-life environments. Instead, a number of simulators have been developed to build and train autonomous underwater vehicles. This paper aims to add to that list of simulators by implementing underwater capabilities into the **R**obot **Evolve** Toolkit. These capabilities will be centered around drag, thrust, and buoyancy physics with more to be implemented on a later date.

To test the accuracy of the implemented physics, an experiment was conducted with a free falling robot model with and without the underwater physics. The falling accelerations were calculated using height measurements taken at regular intervals. The expected buoyancy of a robot sinking in water was calculated and compared to the buoyancy calculated using Newton's Second Law of motion. The measured result shows an agree with the expected results, indicating that the physics were implemented accurately.

Contents

1 Introduction

In recent times there has been an increasing interest in automation and robotics. Many mundane and monotone tasks, such as packaging goods and manufacturing products, are being automated using various forms of robotics, speeding up the process and allowing humans to focus their time and energy on more creative endeavors. An arguably more pressing matter however, is the use of robotics in possibly dangerous fields such as construction, search and rescue missions, and exploration. The latter is especially dangerous - and expensive - in the case of underwater exploration.

This is causing a trend in underwater robotics that leans towards the use of autonomous underwater vehicles (**AUV**'s) as opposed to manned submersibles and remotely operated vehicles. The use of **AUV**'s greatly reduces the risk and costs of experimentation since they do not require the deployment of a research team of a surface vessel or in a submarine. However, real-life experimentation with **AUV**'s is still challenging due to the environment. Simple experiments and basic prototyping can be carried out in physical water tanks, but larger experiments such as navigation and seabed mapping require an open environment which involves high cost and significant time investments. Furthermore, bodies of water such as lakes and ponds provide little to no control over the conditions. Therefore, computer simulations are seen as the more viable alternative to physical experiments. By designing and developing a virtual prototype and analysing the performance in the real world, we can save a lot of time and resources spent on iterative development of physical prototypes and we can more accurately focus on particular vehicular control mechanisms.

This project aims to develop and outline underwater capabilities into the **R**obot **Evlove** (**Revolve**) toolkit [1], partially in an effort to aid Fuda van Diggelen in his research on the automatic design of morphologies and controllers for a range of applications and tasks in different environments. It will mainly be focused on four things:

- **Loading in underwater environments.**

  In order to properly visualize the underwater environments, a number of world models will have to be created that contain virtual representations of water and an ocean floor or bottom of a lake. The goal is to implement at least two underwater environments, such that the impact of the difference in environment on the performance of the robot can be tested.

- **Water resistance and drag.**

  A major physical property of water is an increased amount of resistance. A robot that is moving its body through the air will experience much less resistance than one moving its body through water. This resistance and drag, combined with the right kind of movement, can create thrust which the robot can use to propel itself forward. While this may not be very important for robots that mainly walk on the ocean floor, it is a vital part of the kinetics of swimming.

- **Buoyancy of the models.**

  The models and robots loaded into the underwater environment would realistically experience an amount of buoyant force, which is an upward force that any fluid exerts on objects that are partially or fully submerged in it. For the simulation to be adequately realistic compared to real life, this buoyant force needs to be calculated as accurately as possible.

- **Usability of the implementation.**

  Perhaps most importantly, it is vital that the additions that will be created during this project are easily readable, usable, and implementable by van Diggelen and any other researcher that wants to use or improve the underwater simulation. Therefore, best practices should be applied in the development of the implementation, and any additions should be properly documented and match the design of **Revolve**'s and **Gazebo**'s code.

2 Related Work

The field of robotic simulation, while relatively new, has already produced a number of useful programs that simulate realistic robotic behaviours. This project takes inspiration from some of these previous works, specifically on simulating underwater robotics, and builds on top of them to achieve the goals set in section 1.

2.1 Existing Simulators

There are a number of projects that have developed simulators in the field of **AUV** simulation. Two of the most widely used options for underwater simulation are UnderWater Simulator (**UWSim**) [2] and Unmanned Underwater Vehicle Simulator (**UUVSim**) [3], additionally, the Unity ROS Simulator (**URSim**) [4] is a promising new addition to this lineup. Each of these simulators are open source simulation platforms that offer a general purpose robotics solution for underwater environments. Where they differ is which programs they use to facilitate the visual rendering of their underwater environment. **UWSim** uses the OpenSceneGraph [5] and osgOcean [6] libraries whereas **URSim** uses the cross platform real-time game engine Unity3D [7] and **UUVSim** functions as an extension of the open source robotics simulator **Gazebo** [8].

What each of these platforms have in common is that they use the Robot Operating System (**ROS**) [9] to drive and control the robots in their respective simulators. **ROS** is a framework for writing robot software that offers tools, libraries and conventions that aim to simplify the creation of complex and robust robot behaviour. It was designed to be developed in a collaborative manner where different researchers and institutes contribute to the development. However, it does not offer many tools for evolutionary algorithms.

2.2 Gazebo and Revolve

Revolve is a system that simulates the online evolution of robots, specifically robots that can be physically constructed [1]. In **Revolve**, the robots of each population can evolve their structure and controllers simulataneously in an online, embodied fashion. Importantly, **Revolve** also offers a traditional offline mode alongside the online evolution mode, allowing us to compare the two types of evolution. It is built using a set of Python and C++ libraries, that aid the researcher in setting up simulated experiments regarding the evolution of robotic bodies and minds/controllers. To maintain the performance required to simulate complex environments and experiments, the C++ language is used in the performance critical parts such as robot controllers and physics simulations. Unfortunately while C++ is a very fast and performant language, it can be quite tedious to write. Which is why the parts that are not as performance critical, such as the specification of robots and the world management, are written Python, which is much slower but much more developer-friendly.

Revolve was built on top of the previously mentioned general purpose simulation platform **Gazebo**, making use of the ability for high-performance C++ integration that it offers. **Gazebo** provides both physics simulation and visualisation of rigid body robots and their environment. It also provides integration for several well-established physics engines, allowing an experiment to be run using any of these physics engines by changing only a handful of parameters. The robots and environments are defined using the Simulation Description Format (**SDF**), an XML-based format, which is quite intuitive and easy to understand. Additionally, **Gazebo** has a messaging API which functions as a publisher/subscriber messaging system in which any component can subscribe to and publish on certain topics. This communication happens over TCP sockets and is specified in Google's *Protocol Buffers* format, allowing for straightforward access to the interface. Lastly, **Gazebo** provides a plugin infrastructure which makes it possible to load shared libraries as a plugin for several types of models that have access to the simulation using **Gazebo**'s C++ API. As these plugins are written in C++, they have a high performance and can therefore be used to, for example, add additional physics or logic controllers to the robots or the environments.

3 Designing Implementation

To achieve the goals set in section 1, a number of additions and changes have to be made to **Revolve**'s code base. While they are relatively minimal, a solid understanding of the structure and *'pipelines'* used to simulate the experiments is very beneficial - if not necessary - for understanding those changes and additions.

A simulation is run in **Revolve** by running the `revolve.py` script from the terminal and passing a number of parameters. The main parameters of interest are `--simulator-cmd`, `--world`, and `--manager`. The `--simulator-cmd` parameter determines whether the experiment should use **Gazebo** as its simulator, or the `gzserver`, a simulation only version of **Gazebo** which doesn't visualise the experiment, allowing it to run slightly faster and with more cores if the `--n-cores` parameter is passed as well. The `--world` parameter determines which world model to load, changing the environment accordingly. These world models are stored in `.world` files in the *worlds* folder. And lastly, the `--manager` parameter determines which manager file will be used for the experiment. The manager file is a vital file for running experiments since it - as the name suggests - manages the experiment.

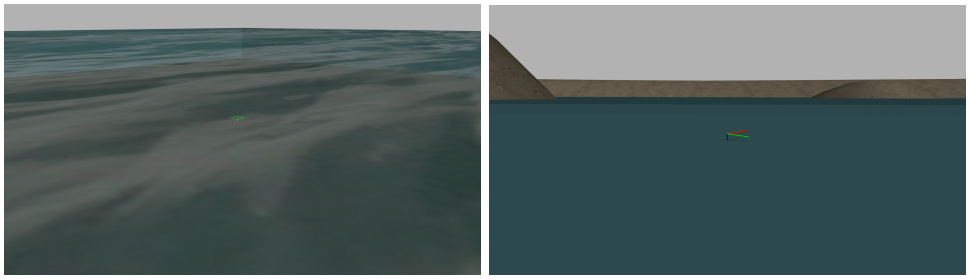
The manager file is connected with the rest of **Revolve**'s code base in various ways; 1) allowing it to set up the simulator, 2) define which robots should be loaded into the world, 3) set up and keep track of the experiment, 4) determine ending conditions, and much more. The manager's function to load the robots is particularly important for the work presented here as this *'pipeline'* can be used to turn underwater physics on for the robot. Not all experiments, after all, will make use of underwater simulation thus, in agreement with the fourth point from section 1, these functions should be made optional and accessible in an intuitive location, which suits the manager file perfectly. Robots are loaded from `.yaml` files using the `load_file` function from the `RevolveBot` class. The robot is then saved in **SDF** usding the `save_file` function from the `RevolveBot` class because **Gazebo** uses **SDF** for its models while the robots are written in YAML for better readability. To turn the `.yaml` file into a `.sdf` file, the `save_file` function makes use of the `revolve_bot_to_sdf` function from `revolve_bot_sdf_builder.py`, which constructs the `.sdf`. This same function is also used to insert the robot into the world model by the `insert_robot` function from the `World` class.

3.1 Worlds

Two different world models, with different contents, have been added to the *worlds* folder, namely `empty_underwater.world` and `lake.world`, along with real-time versions of each. Both models are largely based on the models of the same name from **UUVSim**. Additionally, some models needed to be added alongside to give the worlds the right content. The `ocean_box`, `ned_frame`, `lake`, `ocean`, `sand_heightmap`, `sea_bottom`, and `sea_surface` models have been added to the *models* folder. To provide the right look and texture the *Media* folder has also been added, containing a model for the sea surface, varying textures for clouds, sand, and water, and scripts and programs to load the right material and simulate small waves at the surface of the water.

While these world models are largely based on the models from **UUVSim**, their contents have been adjusted in order to implement them in the **Revolve** code. The world control plugin `libWorldControlPlugin.so` has been added to both models, making them able to interact with the **Revolve** code, and the `<scene>` and `<physics>` tags have been adjusted to fit those of the other world models included in the **Revolve** toolkit.

With these changes **Revolve** can load these two world models as seen in figure 1, and use them as environments to run experiments in. However, while the worlds now appear to contain water, the world models don't contain any physics plugins for underwater environments, this happens in the robot models.



(a) `empty_underwater.world` (b) `lake.world`

Figure 1: Images of the world files

3.2 Lift and Drag

In an underwater environment, the robots interaction with the fluid it is surrounded by weighs more heavily on the kinetics than in an above water environment due to the increased density of the fluid. At one atmospheric pressure and 15°Celsius, the density of water is $999.1026 kg/m^3$ while the density of air in those conditions is $1.225 kg/m^3$. Consequently, since drag is proportional to the density of the fluid, a robot submerged in water will experience nearly 1000 times as much drag as a robot that is above water. This alone would be enough reason to implement drag physics into the underwater simulation but, in addition to that, an increase in drag also causes an increase in thrust. As described by Newton's third law, thrust is a reaction force that an object experiences in one direction after expelling or accelerating mass in the opposite direction. It is most commonly explained as *"Every action has an equal and opposite reaction"*.

With these forces, a robot could swim by attaching one or more propellers that create thrust. Moreover a robot that would try to walk or crawl on the seabed would have to fight the drag in order to keep moving. Implementing the thrust and drag physics is therefore vital for the accuracy of the simulation. Luckily, **Gazebo** offers an easy solution to this problem in the form of the `LiftDragPlugin.so` model plugin [10]. This plugin simulates the forces on an object immersed in a fluid and applies the forces to the object's links directly, thus simulating fluid mechanics.

Before being able to access `LiftDragPlugin.so` from **Revolve**, it first needed to be placed in the *build/lib* folder. After placing it in the proper folder, since it is a model plugin - as opposed to a world, sensor, system, visual, or GUI plugin - the plugin needs to be loaded into the robot model. As stated in section 3, **Revolve** has a function called `revolve_bot_to_sdf` which constructs the `.sdf` file of the revolve bot. The *'pipelines'* from the manager file to this function have been adjusted to include the `underwater` parameter. This is a Boolean that causes the Lift-Drag plugin to be loaded into the robot when True. It is important to have the same value passed through both *'pipelines'*, the one going through the `save_file` function and the one going through the `insert_robot` function, because one creates a `.sdf` file of the robot and saves it in the specified folder but does *not* insert it into the simulation, while the other *does* insert the robot into the simulation but it does *not* save it to a `.sdf` file. With these changes in place, a researcher can now choose to let the robot experience the proper amount of drag and thrust (which is called 'Lift' by **Gazebo** [10]).

3.3 Buoyancy

Just like the Lift-Drag plugin, **Gazebo** also offers a model plugin for simulating buoyancy in the file `BuoyancyPlugin.so` [11], which can be implemented in the `revolve_bot_to_sdf` function. To do this, the C++ files, `BuoyancyPlugin.cc` and `BuoyancyPlugin.hh`, had to be compiled into a shared library (or `.so`) file using CMAKE, and then placed in the *build/lib* folder so that it can be accessed by **Revolve** . However, the default version of the buoyancy plugin is not sufficient for the goals set in section 1 for two reasons.

Firstly, the plugin assumes that the density of the fluid is the same in the entire world,

which is definitely not the case for the worlds described in section 3.1. As you can see in figure 1, the two worlds contain a body of water with a surface. To adjust for this discrepancy, the `surfaceHeight` parameter was added to the plugin and the C++ code was adjusted so that it would only calculate the buoyant force on a module if its Z position is below the `surfaceHeight`. The default value for the `surfaceHeight` is set to 0, making it easy to use in the two worlds as the water surface is set at a height of 0 for both.

The second problem is slightly more complicated as it involves many more parameters. Buoyant force is an upward force that a fluid exerts on an object that is partially or fully submerged in it. By Archimedes's' principle it can be defined as:

$$F_B = \rho_f \times V_m \times g$$

Where F_B is the buoyant force exerted on on a module, ρ_f is the density of the fluid, V_m is the volume of the module, and g is the gravitational acceleration. The buoyancy plugin gets the fluid density as a parameter or it takes the density of water ($999.1026 kg/m^3$) as a default and it gets the gravitational acceleration from the physics engine that the world uses. Thus these parameters should not cause any issues. However, the default version of `BuoyancyPlugin.so` uses the collision box of each module to calculate the volume. This would work if the collision boxes are defined accurately but in **Revolve**'s code, these collision boxes are simplified to be rectangular boxes. However, in reality the modules are shaped like the frames seen in figure 2, which means they have a much smaller volume than their corresponding collision boxes. It was considered to change the collision box to a collision mesh that accurately represents the shape and volume of each module. However, since the volume of the modules is only relevant for the underwater simulation, and since this change would likely come with a huge cost to performance, it was decided to solve the problem with a different approach.

The plugin code can read information about the structure and code of the robot models through **Gazebo**'s C++ API. Specifically, it can access the names of the modules that make up the model, which all have a unique name assigned to them by **Revolve**. These names contain identifiers that show which type of module it is, thus, with a simple string search in the names, the buoyancy plugin could identify each module's type and assign a predetermined volume to that module. Finding the real volume seemed like a difficult task but it proved to be rather simple thanks to research done by Cha Zhang and Tsuhan Chen [12]. Their paper shows that the volume of a mesh can be calculated by taking the sum of the signed volumes of each of the triangles in the mesh. Giving us:

$$V_{total} = \sum_i V_i$$

$$V_i = \frac{1}{6}(-x_{i3}y_{i2}z_{i1} + x_{i2}y_{i3}z_{i1} + x_{i3}y_{i1}z_{i2} - x_{i1}y_{i3}z_{i2} - x_{i2}y_{i1}z_{i3} + x_{i1}y_{i2}z_{i2})$$

With i being a triangle consisting of (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3). Using this formula, the volumes of the modules have been calculated (see table 1) and implemented in the plugin code to calculate the accurate amount of buoyant force.

Module	Volume of Collision Box	Volume of Mesh	Mass of Module
CoreModule	3.564e-04 m^3	5.054e-05 m^3	0.090 kg
BrickModule	5.968e-05 m^3	1.070e-05 m^3	0.0102 kg
ActiveHingeServoHolder	1.062e-05 m^3	1.014e-05 m^3	0.0017 kg
ActiveHingeFrame	7.865e-06 m^3	3.032e-06 m^3	0.009 kg
TouchSensor	5.968e-05 m^3	2.735e-06 m^3	0.003 kg

Table 1: Volumes of each robot module

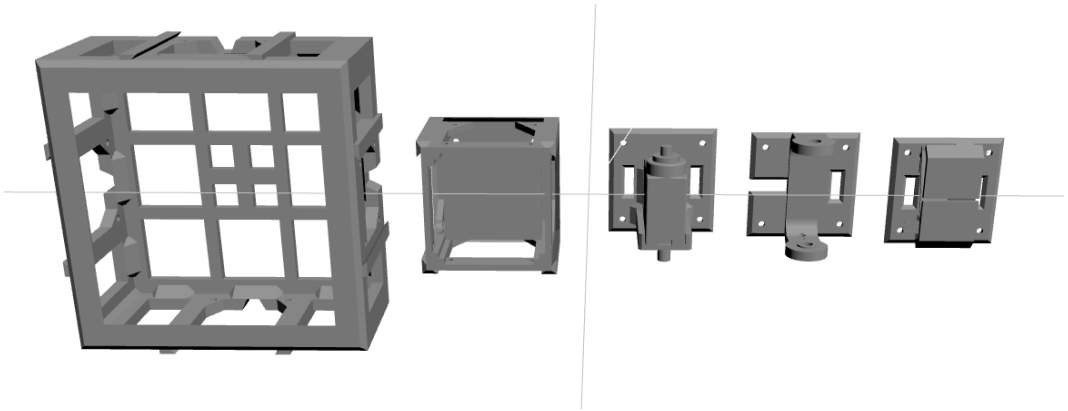


Figure 2: Meshes of the modules, with from left to right: CoreModule, BrickModule, ActiveHingeServoHolder, ActiveHingeFrame, and TouchSensor

4 Experimental Assessment

With the underwater world models and the Lift and Drag, and Buoyancy plugins properly implemented, **Revolve** should be capable of running an adequately accurate underwater simulation. That the world files work properly is plainly visible since they can be loaded in and display the right environment, however, the plugins are slightly more difficult to test. Since no changes have been made to the `LiftDragPlugin.so` code, it comes straight from **Gazebo**'s available plugins. Therefore this paper assumes that the Lift and Drag plugin works as intended given that the input parameters, explained in the tutorial [10], are properly set up in `revolve_bot_to_sdf`. To find out whether that is also the case for the Buoyancy plugin, which has undergone a number of significant changes, an experiment has to be conducted.

4.1 Experimental Setup

A falling object will continuously accelerate with the gravitational acceleration of its environment when it is in a vacuum. If an upward force is applied to the object, its acceleration will be reduced, become 0, or even be redirected upwards if the force is strong enough. Since the buoyant force is an upward force, it will oppose the gravitational acceleration of a falling object, making it fall slower. Using this principle, the buoyant force an object experiences in the simulation could be calculated by by turning off the Lift and Drag plugin, effectively creating a vacuum, and measuring the acceleration of an object that uses the Buoyancy plugin.

For this experiment the example spider (`spider.yaml`) robot will be used. The spider robot consists of one CoreModule, eight BrickModules, and eight ActiveHinges with frames and servoholders which gives us:

$$m_{spider} = m_{CoreModule} + 8m_{BrickModule} + 8m_{ActiveHingeServoHolder} + 8m_{ActiveHingeFrame}$$

$$m_{spider} = 0.090 + 8 \times 0.0102 + 8 \times 0.0017 + 8 \times 0.009$$

$$m_{spider} = 0.2572kg$$

And

$$V_{spider} = V_{CoreModule} + 8V_{BrickModule} + 8V_{ActiveHingeServoHolder} + 8V_{ActiveHingeFrame}$$

$$V_{spider} = 5.054\text{e}{-05} + 8 \times 1.070\text{e}{-05} + 8 \times 1.014\text{e}{-05} + 8 \times 3.032\text{e}{-06}$$

$$V_{spider} = 2.415\text{e}{-04}m^3$$

Using the formula for buoyancy from section 3.3 and the volume of the spider, we can calculate the buoyant force the spider should experience:

$$F_{B,spider} = 999.1026 \times 2.415\text{e}{-}04 \times g$$

To find g, we need to calculate the falling acceleration of the spider that does not experience any buoyant force (S_V). In theory, the predefined gravitational acceleration used in the simulator could also be used. However, the data is gathered outside of the simulator, in the manager file, and there could possibly be a slight delay, or even a different time factor based on computation time. Therefore, to avoid these issues, g will be experimentally defined. With g measured, we can calculate the buoyant force that the other spider (S_B) should experience and compare it to the force we can calculate using the measured acceleration.

While **Revolve** does have a function called `measures.velocity`, figure 4 of the appendix shows that it only seems to calculate the horizontal velocity of the robots and is therefore not useful for calculating the falling velocity of the spiders. Instead, this experiment will be made by measuring the Z-position of the model at a constant time interval of 0.2 seconds and extrapolating the velocity and acceleration from this data. Assuming that the acceleration is constant during the fall, it can be defined as:

$$a = \frac{\Delta v}{\Delta t} = \frac{v_{end} - v_{start}}{\Delta t}$$

And the velocity at time-step i can be approximated by:

$$v_i = \frac{h_{i+1} - h_i}{0.2}$$

4.2 Results

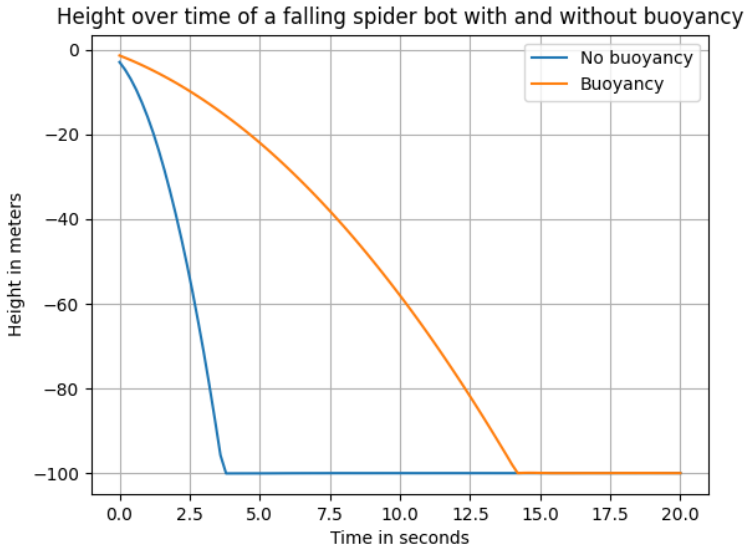


Figure 3: Heights of S_V and S_B plotted over time

Figure 3 shows the height over time of S_V and S_B, note that since the measurements are taken in the manager file in **Revolve**, the first measurement is taken a fraction of a second after the simulation is started in **Gazebo** causing the robots to start with a downward velocity and slightly below a height of 0.

From the figure it is immediately clear that the Buoyancy plugin is slowing down the fall of S_B, which is the expected result. However, to calculate the exact amount that S_B is being slowed down we need to calculate the accelerations of S_V and S_B. Thus, as described in

section 4.1 we first need to calculate v_{start} and v_{end} of S_V and S_B. With the data gathered from the experiment (table 2) this comes down to:

$$v_{start,S_V} = \frac{h_{1,S_V} - h_{0,S_V}}{0.2} = \frac{-4.74 - -2.94}{0.2} = -9.00m/s$$

$$v_{end,S_V} = \frac{h_{19,S_V} - h_{18,S_V}}{0.2} = \frac{-95.70 - -87.22}{0.2} = -42.40m/s$$

And

$$v_{start,S_B} = \frac{h_{1,S_B} - h_{0,S_B}}{0.2} = \frac{-1.92 - -1.38}{0.2} = -2.70m/s$$

$$v_{end,S_B} = \frac{h_{71,S_B} - h_{70,S_B}}{0.2} = \frac{-97.92 - -95.69}{0.2} = -11.15m/s$$

The velocities are all negative because the spiders fall downwards while the height is measured upwards from 0. However, the distance that they travel is the same amount whether they go up or down, therefore the negative signs can be ignored in further calculations.

Using the velocities from the previous calculations we can calculate the acceleration:

$$a_{S_V} = \frac{v_{end,S_V} - v_{start,S_V}}{\Delta t} = \frac{42.40 - 9.00}{0.2 \times 18} = 9.28m/s^2$$

$$a_{S_B} = \frac{v_{end,S_B} - v_{start,S_B}}{\Delta t} = \frac{11.15 - 2.70}{0.2 \times 70} = 0.60m/s^2$$

Since S_V falls in a vacuum, its acceleration should be equal to the gravitational acceleration of the simulation. Thus, we can fill it in for g in the formula for buoyant force on S_B:

$$F_{B,S_B} = 999.1026 \times 2.415\text{e}{-04} \times 9.28 = 2.24N$$

Now we know that the expected buoyant force is 2.24N. To calculate the measured buoyant force we have to calculate the gravitational force and the resulting downward force. And since the buoyant force is in an opposite direction of the gravitational force, the resulting force can be defined with the following equation:

$$F_{resulting} = F_{gravity} - F_{buoyant}$$

The gravity force can be calculated using Newton's Second Law of motion $F = m \times a$, and the gravitational acceleration:

$$F_{gravity} = 0.2572 \times 9.28 = 2.39N$$

And the resulting force on S_B can be calculated using Newton's Second Law of motion and the measured acceleration

$$F_{resulting} = 0.2572 \times 0.60 = 0.15N$$

These values can then be used to find the measured buoyant force using the previously mentioned equation:

$$F_{resulting} = F_{gravity} - F_{buoyant}$$

$$0.15 = 2.39 - F_{buoyant}$$

$$F_{buoyant} = 2.39 - 0.15 = 2.24N$$

This result lines up perfectly with the expected buoyant force, indicating that the buoyancy plugin accurately calculates the buoyant force.

4.3 Discussion

Since the height data of the spiders was gathered and saved in the manager file, it is possible that there was some sort of delay between the time of the measurement in **Gazebo** and the time of the measurement in the Python code. This in itself could cause some inaccuracy in the experiment but in addition to that, due to the reduced certainty that this possible delay causes, it was decided to limit the significant figures used from the measurements to two. Using all the significant figures from table 2 from the appendix, a difference of approximately $0.00873N$ can be measured. This does not necessarily mean that this is the actual inaccuracy since there is little certainty in the additional significant figures and they themselves might be inaccurate.

In addition to a possible delay in measurements, there is also a concern when it comes to differences in time factors. **Gazebo** has a parameter called *real time factor* which displays the amount of time that passes in-simulator, in relation to the real time. So a *real time factor* of 10 would mean that for every second that passes in real time, ten seconds pass in the simulator. In the experiment the *real time factor* was set to be 1.00 but it is possible that, due to computation time for example, the *real time factor* was actually slightly higher or lower than 1.00. For the sake of consistency, the gravitational acceleration was therefore also measured experimentally, with the assumption that the potential difference in *real time factor* would remain constant in each simulation. However, it could be the case that there was a difference in the *real time factor* of the two simulations due to the increased computational load that the buoyancy plugin introduces,

5 Conclusions

With the implementation in place and the experiment concluded, the goals set in section 1 should be revisited in order to asses how well they were met and whether or not this paper has succeeded in its attempt of outlining an implementation of underwater physics into the **Revolve** toolkit.

- **Loading in underwater environments.**

  As seen in section 3.1 and figure 1, two different world files containing underwater environments have been implemented. The worlds are adequately different, with one being a flat underwater plane and the other having a clear geographical, thus they could later be used to test the performance of underwater robots in different environments.

- **Water resistance and drag.**

  The resistance and drag are being handled by the `LiftDragPlugin.so` model plugin that **Gazebo** offers. It is assumed that this plugin works as intended and accurately calculates the drag and thrust (or lift) that the model experiences underwater.

- **Buoyancy of the models.**

  The experiment in section 4 shows that the accuracy of the buoyancy plugin is very close to, if not perfectly aligned with the expected result. It may need further experimentation and perhaps improvements but, for the scope of this project, the Buoyancy plugin is sufficiently accurate and therefore successfully implemented.

- **Usability of the implementation.**

  The usability of the implementation is more subjective than the previous goals and depends largely on the experience that a researches has with the **Revolve** toolkit, their ability to program in Python and C++, and their knowledge of the physics involved. That considered, best practices have been applied in the development and additions have been integrated to match the design of **Revolve**'s and **Gazebo**'s code. Most significant changes include proper documentation and are available in Ravi-Bossema's fork of the revolve repository [13]. Thus, it could be concluded that the implementation should be easily usable by experienced researchers.

All in all, the implementation meets the requirements set in section 1 and can hopefully be used by van Diggelen in his research on the automatic design of morphologies and controllers for a range of applications and tasks in different environments.

5.1 Further Developments

This paper offers what is possibly just the first step in what could prove to be a long road towards implementing realistic underwater simulations using the **Revolve** toolkit. The next steps to take fall outside of the scope of this project and might prove to be large enough to warrant their own projects and papers. However, some problems and some possibilities stood out particularly over the course of this project.

First and foremost is the fact that, while the volumes of the meshes are much more accurate than those of the collision boxes, they do not contain any of the electronics and wiring used to control the robots. Therefore, the volumes of the modules, as calculated in section 3.3, are not completely accurate. It is unclear whether or not the mass of the electronics has been included in the masses of the modules but it is highly unlikely that they would have the exact same density as water meaning that the actual modules would, in reality, be less dense and therefore more buoyant than those in the simulation. This inaccuracy could be fixed by measuring the volume of the electronics and adding this to the volumes of the frames in the Buoyancy plugin.

Additionally, while researching related works, it was noticed that **UUVSim** has an Underwater Current plugin. To add this plugin to **Revolve**, changes will likely have to be made to the plugin code but it is definitively worthwhile to do so in a future project since any robot in a real lake or even a pool will encounter such underwater currents and they would have to be able to navigate through them.

Lastly, the underwater implementation, thus far, does not yet simulate any surface tension or partial submersion of modules, making the behaviour at the surface of the water quite unrealistic. This is not a problem when experimenting on the ocean floor or with fully submerged swimmers but, to build any robots that can swim on, or up to, the surface, it is imperative that these physics are implemented in the simulator at some point.

5.2 Future Experiments

The eventual goal of this paper is to implement underwater capabilities that can be used to run evolutionary experiments on **AUV**'s. If the previously mentioned physics issues are resolved, experiments could be set up to evolve robots that crawl on the bottom of a body of water or swimmers. The included world models should offer a decent amount of variety to the available environments but adding additional world models that fit a specific purpose should be rather straightforward due to the nature in which they are constructed.

Future projects with the underwater capabilities of **Revolve** could also be centered around the implementation of a thruster-like propeller module and experimentally determining its performance in the underwater environments.

References

[1] Elte Hupkes, Milan Jelisavcic, and A. E. Eiben. **Revolve: A Versatile Simulator for Online Robot Evolution**. *Applications of Evolutionary Computation Lecture Notes in Computer Science*, page 687–702, 2018. doi: 10.1007/978-3-319-77538-8_46.

[2] Mario Prats, Javier Perez, J Javier Fernandez, and Pedro J Sanz. **An open source tool for simulation and supervision of underwater intervention missions**. In *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*, pages 2577–2582. IEEE, 2012.

[3] Musa Morena Marcusso Manhães, Sebastian A Scherer, Martin Voss, Luiz Ricardo Douat, and Thomas Rauschenbach. **UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation**. In *OCEANS 2016 MTS/IEEE Monterey*, pages 1–8. IEEE, 2016.

[4] P. Katara, M. Khanna, H. Nagar, and A. Panaiyappan. **Open Source Simulator for Unmanned Underwater Vehicles using ROS and Unity3D**. In *2019 IEEE Underwater Technology (UT)*, pages 1–7, April 2019. doi: 10.1109/UT.2019.8734309.

[5] **OpenSceneGraph**, 2018. `http://www.openscenegraph.org/` [Accessed: 15-6-2021].

[6] **osgOcean**, 2009. `https://code.google.com/archive/p/osgocean/` [Accessed: 15-6-2021].

[7] **Unity3D**, 2021. `https://unity.com/` [Accessed: 15-6-2021].

[8] N. Koenig and A. Howard. **Design and use paradigms for Gazebo, an open-source multi-robot simulator**. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004. doi: 10.1109/IROS.2004.1389727.

[9] **Robot Operating System**, 2017. `ros.org` [Accessed: 15-6-2021].

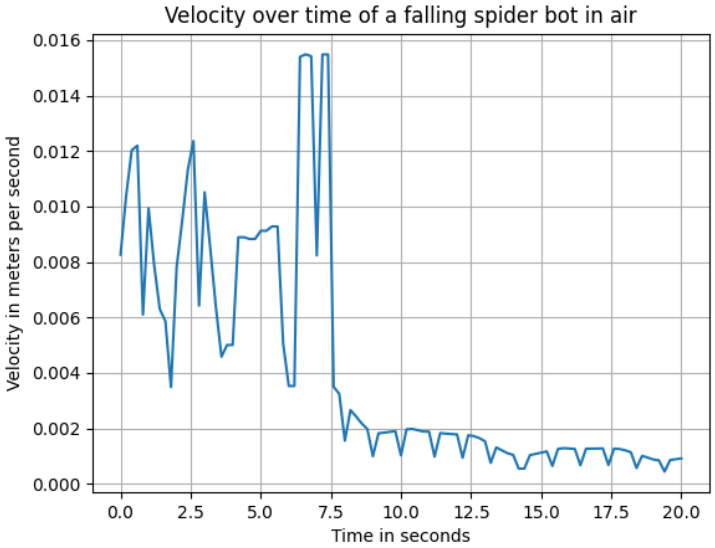
[10] **Aerodynamics**, 2014. `http://gazebosim.org/tutorials?tut=aerodynamics&cat=physics` [Acessed: 18-6-2021].

[11] **Hydrodynamics**, 2014. `http://gazebosim.org/tutorials?tut=hydrodynamics&cat=physics` [Acessed: 18-6-2021].

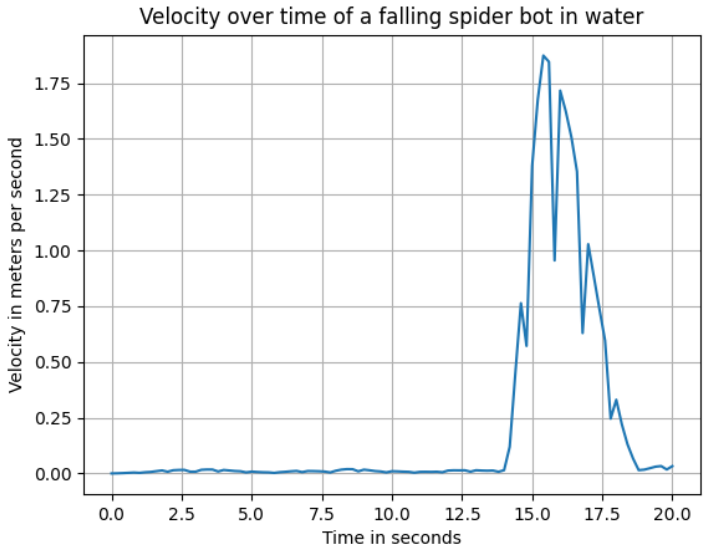
[12] Cha Zhang and Tsuhan Chen. **Efficient feature extraction for 2D/3D objects in mesh representation**. In *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, volume 3, pages 935–938. IEEE, 2001.

[13] **Fork of the Revolve repository**, 2021. `https://github.com/Ravi-Bossema/revolve` [Acessed: 1-7-2021].

A Appendix



(a) Not using buoyancy



(b) Using buoyancy

Figure 4: Plots of the `measures.velocity` of falling spiders

Time-step	Height of S_V	Height of S_B
1	-2.9431191136	-1.3847804052
2	-4.7408995621	-1.9154337879
3	-6.9384893584	-2.4649190347
4	-9.5081209958	-3.0447823503
5	-12.501135170	-3.6502166606
6	-15.858379566	-4.2811865909
7	-19.607567794	-4.9368965945
8	-23.792095984	-5.6096202835
9	-28.376473280	-6.3130388839
10	-33.360768456	-7.0409137895
11	-38.689794155	-7.7857498874
12	-44.410932833	-8.5620972883
13	-50.524218280	-9.3626794100
14	-57.029633808	-10.188001038
15	-63.998107094	-11.038668353
16	-71.291616135	-11.905896573
17	-79.055972696	-12.806586978
18	-87.220179546	-13.731568283
19	-95.697486105	-14.671647004
20	-99.992213128	-15.646183711
21	-99.988618693	-16.636140722
22	-99.985852900	-17.650855712
23	-99.983663505	-18.689552651
24	-99.981759797	-19.751213757
25	-99.980064093	-20.846883588
26	-99.978612347	-21.955920960
27	-99.975080126	-23.101304814
28	-99.969578992	-24.272361184
29	-99.962607117	-25.469276589
30	-99.955385743	-26.691830732
31	-99.949272535	-27.939112633
32	-99.944491632	-29.209747159
33	-99.941053719	-30.503235747
34	-99.938788011	-31.820729591
35	-99.937484254	-33.163294216
36	-99.936394681	-34.531399875
37	-99.933915377	-35.924823452
38	-99.929553934	-37.343245451
39	-99.926487365	-38.771680777
40	-99.925471309	-40.238512947
41	-99.925411355	-41.714911348
42	-99.926125544	-43.230790930
43	-99.927520054	-44.771867424
44	-99.927519924	-46.338235487
45	-99.927519878	-47.929672283
46	-99.927519829	-49.545850365
47	-99.926378212	-51.186609065
48	-99.925082727	-52.851649778
49	-99.923924424	-54.540625863
50	-99.923315260	-56.253608473

Time-step	Height of S_B
51	-57.9907496403
52	-59.7346723714
53	-61.5207685216
54	-63.3319019276
55	-65.1683759497
56	-67.0306003896
57	-68.9190967556
58	-70.8343277465
59	-72.7755356436
60	-74.7399414961
61	-76.7254231619
62	-78.7332729413
63	-80.7665179258
64	-82.8266658177
65	-84.9133992338
66	-87.0251100898
67	-89.1599293435
68	-91.3174983538
69	-93.4997293099
70	-95.6856954331
71	-97.9188879862
72	-99.9107195046
73	-99.8638381278
74	-99.8450143245
75	-99.8527226788
76	-99.8817257304
77	-99.8948612483
78	-99.9212274402
79	-99.9207444327
80	-99.9203757305
81	-99.9200276445
82	-99.9192104112
83	-99.9189546027
84	-99.9172384992
85	-99.9157156545
86	-99.9149716969
87	-99.9155586326
88	-99.9114952854
89	-99.9082112112
90	-99.9055259848
91	-99.9031423133
92	-99.9015555840
93	-99.9001874840
94	-99.8988781605
95	-99.8987241543
96	-99.9008681254
97	-99.9056132811
98	-99.9096201531
99	-99.9136841597
100	-99.9156387170

Table 2: Height values from the experiment