

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Andrei Moissejev 143691

MAGNETIC LEVITATION SYSTEM SIMULATION IN VIRTUAL REALITY

Master's thesis

Supervisor: Aleksei Tepljakov
PhD

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Andrei Moissejev 143691

**MAGNETILISE LEVITATSIOONI
SÜSTEEMI SIMULATSIOON
VIRTUAALREAALSUSES**

Magistritöö

Juhendaja: Aleksei Tepljakov
PhD

Tallinn 2018

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Andrei Moissejev

Abstract

Magnetic Levitation System Simulation In Virtual Reality

This work documents the process of development of an interactive and dynamic magnetic levitation system that operates within the virtual reality environment.

The first part of the thesis provides a definition of the tasks this project is meant to accomplish as well as an introductory overview of the software tools used during the development of this application : primarily *Unreal Engine* and *Simulink*.

Most of the thesis documents the development process of the application and the thought process behind certain design decisions. The reader is also introduced to the variety of inherent software limitation problems faced when attempting to build an accurate dynamic model within a game engine environment.

Throughout the course of the development, some existing code solutions for particular problems from similar projects by other university students were borrowed and implemented within this application.

The majority of the work was done within the *Unreal Engine* game engine, which has inbuilt support for virtual reality. This, in turn, allows the end user to interact with the control system in a capacity similar to reality.

This thesis is written in English and is 51 pages long, including 9 chapters, 34 figures and 2 tables.

Kokkuvõte

Magnetilise Levitatsiooni Süsteemi Simulatsioon Virtuaalreaalsuses

Antud teos dokumenteerib interaktiivse dünaamilise magnetilise levitatsiooni süsteemi arenduskäigu, mis toimib virtuaalses keskkonnas.

Lõputöö esimeses osas on defineeritud antud projekti eesmärk ning lisatud põgus ülevaade kasutatud tarkvaradest, mille abil antud rakendust arendati. Peamiselt on nendeks *Unreal Engine* ja *Matlab*.

Enamik lõputööst käsitleb antud rakenduse arendustööd ja vajalikku eelnevat mõttetööd. Ühtlasi saab lugeja hea aimduse tarkvara piiratusest, mis ilmnis siis kui autor üritas luua realistlikku mudelit arvutimängumootori keskkonnas.

Arendustöö käigus laenas autor teatud probleemide lahendamiseks koodilõike teiste ülikoolide õpilaste sarnastest töödest ning kohandas neid antud projekti tarbeks vastavalt vajadusele.

Enamik tööst on tehtud mängumootori *Unreal Engine* abil, mis toetab virtuaalreaalsuse keskkonna loomist. Tänu sellele saab lõppkasutaja juhtida antud süsteemi tegelikkusega võrreldavas ulatuses.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 51 leheküljel, 9 peatükki, 34 joonist, 2 tabelit.

List of abbreviations and terms

UE	Unreal Engine
VR	Virtual Reality
MLS	Magnetic Levitation System
CMD	Command Prompt
A-Lab	Alpha Control Laboratory

Table of Contents

1 Introduction.....	11
2 Project Overview.....	12
2.1 Project Objectives.....	12
2.2 Completion Strategy.....	12
2.3 Unreal Engine Overview.....	13
2.3.1 Code Compiler.....	13
2.3.2 Blueprint Visual Scripting System.....	13
2.3.3 Physics Simulation.....	14
2.4 Matlab & Simulink Overview.....	14
3 Mathematical Model Implementation.....	15
3.1 Velocity Acceleration Implementation.....	17
3.2 Coil Current Acceleration Implementation.....	20
4 PID Controller Implementation.....	24
4.1 Proportional Term.....	26
4.2 Integral Term.....	26
4.3 Derivative Term.....	28
4.4 Term Summation.....	29
5 Network Interface Implementation.....	30
5.1 Simulink Network Interface.....	30
5.2 Unreal Engine Network Interface.....	34
6 System Validation.....	36
6.1 Coil Current Acceleration Modifier Model Selection.....	36
6.2 Unreal Engine Magnetic Levitation System Validation.....	39
7 Virtual Reality Implementation.....	45
8 Level Design.....	46
9 Conclusion.....	49
References.....	50

List of Figures

Figure 1. Example of PhysX particle physics.....	14
Figure 2. Physical model of the MLS.....	16
Figure 3. “set_distance” function code, it applies force to the ball.....	19
Figure 4. Coil current simulation model within the game level.....	21
Figure 5. “set_amperage” function code, it regulates amperage simulation.....	23
Figure 6. MLS Simulink diagram.....	24
Figure 7. Ball position over time in PID and FOPID simulations.....	25
Figure 8. PID controller block internal diagram.....	25
Figure 9. “p_control” function code, it outputs proportional term value.....	26
Figure 10. “i_control” function code, it outputs integral term value.....	26
Figure 11. PID delta_time variable assignment blueprint.....	27
Figure 12. Integrator block options.....	27
Figure 13. “d_control” function code, it outputs derivative term value.....	28
Figure 14. Derivative components blueprint flow.....	28
Figure 15. “set_voltage” function code, it outputs voltage value.....	29
Figure 16. PID gain factors initialization blueprint.....	29
Figure 17. Packet Input block setup.....	31
Figure 18. Project frame rate settings in Unreal Engine.....	32
Figure 19. Task Manager details.....	32
Figure 20. CMD command “netstat -a -o -n” result.....	33
Figure 21. “Andmed.h” structure variable declaration code portion.....	34
Figure 22. Network components initialization blueprint.....	35
Figure 23. Simulink diagram voltage and coil current characteristics.....	36
Figure 24. Unreal Engine MLS voltage and coil current characteristics.....	37
Figure 25. Voltage and coil current during simulation.....	38
Figure 26. MagLev Model block parameters.....	39
Figure 27. Simulink and UE MLS simulation results.....	40
Figure 28. UE MLS simulation results with different initial conditions.....	41

Figure 29. Simulink and UE MLS simulation results after modification.....	42
Figure 30. UE modified MLS simulation results with different initial conditions.....	43
Figure 31. Modified UE MLS response to a minor and major disturbance.....	44
Figure 32. Finished MLS level environment.....	46
Figure 33. Imported ball mesh resting on a platform.....	47
Figure 34. Collision columns during run-time and development.....	48

List of Tables

Table 1. Acceleration modifier value calculated by distance.....	18
Table 2. Amperage modifier value calculated by distance.....	22

1 Introduction

Education is one of the most heavily regulated sectors of human endeavour. The excessive regulation and reluctance to change can produce consistent results but this inertia can also work to inhibit progress in efficiency that technological innovation generally introduces to an existing system.

Because it is difficult to change the existing structural content that comprises education, what has recently been attempted is introduction of entertainment value into the educational process, otherwise known as edutainment.

The entertainment sector, in contrast, is something that is very prone to change and unpredictability. One of the more recent introductions into the market is virtual reality technology which is mostly supported by a niche audience, but is increasingly gaining more support and interest from the wider public [1].

This push for interactive content that is both educational and entertaining can certainly benefit from the rising popularity of virtual reality. It can be seen as a perfect vector for delivering content that provides additional dimension of interaction that is yet to become mainstream.

This project in particular focuses on simulating a magnetic levitation system which can be used as an object of study in control theory. Introduction of virtual reality will allow students to interact with the control system in a remote virtual environment.

2 Project Overview

2.1 Project Objectives

The main objective of this practical project is to create a dynamic magnetic levitation system for use in interactive virtual reality applications, the built model must adequately represent how the system functions in reality.

The primary tools to accomplish this project would be the following software :

- *Unreal Engine* – a game engine;
- *Simulink* – a graphical programming environment.

2.2 Completion Strategy

The following steps were taken to bring the project to completion :

1. Build a levitation system in *Unreal Engine* that is based on the mathematical model of the levitation control system provided by *A-Lab*;
2. Implement a control algorithm in *Unreal Engine* that is based on a functional *Simulink* PID control system provided by *A-Lab*;
3. Implement a network interface between *Unreal Engine* and *Simulink* that would allow a running *Simulink* model to validate the built system;
4. Validate the system using the existing *Simulink* PID control system to ensure that an adequate level of accuracy and realism has been achieved. Adjust the *Unreal Engine* model parameters and control algorithm as deemed necessary until system performance is sufficiently acceptable;

5. Introduce virtual reality controller support into the application;
6. Design the simulation level environment using the provided assets.

Every step is thoroughly described in a dedicated section of this document, various implementation design decisions that were necessary to make throughout the project are also sufficiently justified where mentioned.

2.3 Unreal Engine Overview

Unreal Engine is a game engine developed by Epic Games [2]. Though primarily used for building games, it is also a perfect tool for realizing a project such as this. *UE* is free to use, supports *VR* and is highly customizable. In addition to this, the engine functionality is very well documented [3].

2.3.1 Code Compiler

The engine code for *UE* is written in *C++*. For the purposes of writing custom code and compiling it, an external integrated development environment is used – *Microsoft Visual Studio* [4]. The free version of the development environment will be satisfactory for a project of this scope [5].

2.3.2 Blueprint Visual Scripting System

One of the more unique elements of *UE* is its' *Blueprint Visual Scripting* system [6]. It is a scripting system that allows to forego traditional literal coding and replace it with a node-based interface to create gameplay elements from within *Unreal Editor*.

This is a system that is very much reminiscent of module driven development wherein interactions between objects are modelled by a human operator on an abstract layer and the software works to translate built models and relationships into literal code [7].

This *Blueprint Visual Scripting* system is very flexible and allows user to use virtually the full range of concepts and tools generally only available to programmers, it also allows for implementation and use of any custom written *C++* code.

2.3.3 Physics Simulation

Unreal Engine uses the *PhysX* 3.3 physics engine to drive its physical simulation calculations and perform all collision calculations [8]. In video games, *PhysX* is most noticeable when used for graphical fidelity and realism : generating particles and debris.



Figure 1. Example of *PhysX* particle physics.

2.4 Matlab & Simulink Overview

Matlab is a software environment primarily used by scientists and engineers for the purposes of analysis and design of various systems, primarily those that involve computational data [9]. It is very convenient for use in analysis of large data sets, the code language of *Matlab* is easy to learn and can be integrated with other languages.

Simulink is an environment for simulation and model-based design of dynamic and embedded systems [10]. It is integrated with *Matlab* and allows to create various sorts of complex system simulations through its' graphical block diagram building.

3 Mathematical Model Implementation

The mathematical model of the system is provided by an academic paper authored by Aleksei Tepljakov, Eduard Petlenkov, Juri Belikov and Emmanuel A. Gonzalez [11].

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\frac{c(x_1)}{m} \frac{x_3^2}{x_1^2} + g$$

$$\dot{x}_3 = \frac{fip_2}{fip_1} \frac{i(u) - x_3}{e^{-\frac{x_1}{fip_2}}}$$

Where x_1 is the position of the sphere, x_2 is the velocity, x_3 is the coil current,

$$fip_1 = 1.1165 \times 10^{-3} \text{ m/s} ,$$

$$fip_2 = 26.841 \times 10^{-3} \text{ m} ,$$

$$c(x_1) = 3.9996 x_1^4 + 3.9248 x_1^3 - 0.34183 x_1^2 + 0.007058 x_1 + 2.9682 \times 10^{-5} ,$$

$$i(u) = -0.3 u^2 + 2.6 u - 0.047 ,$$

$$m = 0.0585 \text{ kg} ,$$

$$g = 9.81 \text{ m/s}^2 ,$$

$$e = 2.71828 .$$

It should be noted that the voltage control signal is normalized and has the range of $u \in [0, 1]$ and this corresponds to the pulse-width modulation duty cycle 0 ... 100%. In addition to this, the deadzone in control is $u_{dz} \in [0, 0.0182]$.

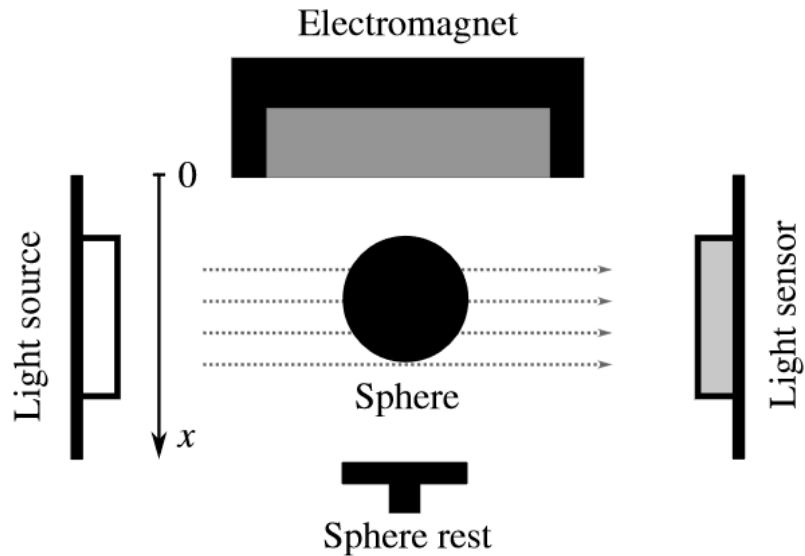


Figure 2. Physical model of the *MLS*.

There were a total of two implementation options that were worth consideration :

1. Calculate the value of x_1 – the sphere position to directly set the position of the sphere object within the simulation :
 - Allows maximum degree of control over the system,
 - The relationships between position, velocity, acceleration and their influence on each other must be manually calculated,
 - User interaction, gravity and dynamic response to a physical disturbance must be manually programmed for the system to properly behave;
2. Calculate the value of x_2' – the sphere velocity acceleration and use *PhysX* functionality to apply force to the object within the simulation :
 - No direct control over the sphere position, only the force applied to it,
 - Defer all physics-related calculations to *PhysX* : gravity, physical disturbances, user interactions, relationship between position, velocity and acceleration.

The second option was selected for this project, primarily due to the necessity of user interaction. Writing a custom physics framework within an engine that already has a functional physics engine would be wasteful. As a consequence, the system becomes a grey box model where there is partial understanding of the theoretical structure – which is the provided mathematical model, and data – which is the *Simulink* PID model that will be used for validation.

Though the first steps of the project are to recreate the mathematical and control system models provided, it is important to remember that the primary objective is to have the system perform as it does in reality – this means that the models that will operate within the engine will be freely adjusted upon necessity with little theoretical justification if the practical results produce a more accurate output. This is done, primarily, to compensate for the inability of *Unreal Engine* to perfectly replicate the results of a theoretical model for which the engine is not designed to work with in the first place.

3.1 Velocity Acceleration Implementation

$$\dot{x}_2 = -\frac{c(x_1)}{m} \frac{x_3^2}{x_1^2} + g$$

In order to observe how much of an impact the ball position can have on the value of acceleration the original formula will be transformed.

If all the constants in the formula were given their real values and the distance variable was accounted for in a separate function, the resulting equation would take this form :

$$\dot{x}_2 = y(x_1) \times x_3^2 ,$$

where y is the acceleration modifier the value of which depends entirely on the distance of the ball, x_3 is coil current, g is removed from this equation because *Unreal Engine* will automatically enforce gravitational pull on the ball object.

For the object to be lifted off the platform, the acceleration should be of greater value than the gravitational pull of the planet, because of this the acceleration modifier should

be of sufficient negative value. Coil amperage squared will always be a positive value that is relatively small considering that the current cannot exceed 2.25 amperes.

Sphere position has the range of $x_1 \in [0 , 0.0155]$, however, the model permits to calculate this position in two ways : position from the ceiling of the platform, or position from the bottom of the platform. The table below shows how the value of the acceleration modifier will change depending on how the ball position will be calculated.

Table 1. Acceleration modifier value calculated by distance.

Position, from floor	Acceleration Modifier	Position, from ceiling	Acceleration Modifier
0.0000	-4.68354E+30	0.0155	-5.10882818
0.0010	-622.2580991	0.0145	-5.877844932
0.0020	-181.462147	0.0135	-6.795944264
0.0030	-90.95123185	0.0125	-7.905294658
0.0040	-56.30014006	0.0115	-9.265159673
0.0050	-38.9192135	0.0105	-10.96130465
0.0060	-28.76403622	0.0095	-13.12222836
0.0070	-22.22017772	0.0085	-15.9486537
0.0080	-17.70693375	0.0075	-19.77055802
0.0090	-14.4356211	0.0065	-25.16630566
0.0100	-11.97329846	0.0055	-33.23714967
0.0110	-10.06442565	0.0045	-46.32709084
0.0120	-8.549319813	0.0035	-70.2828655
0.0130	-7.323496503	0.0025	-123.7662735
0.0140	-6.315944517	0.0015	-300.1948638
0.0150	-5.476841766	0.0005	-2265.027921
0.0155	-5.10882818	0.0000	-4.68354E+30

The modulo of the acceleration modifier is the lowest in the starting rest position and highest at the peak. If the coil current were to be constant, the acceleration of the ball

would reach its' peak when already at the ceiling while the acceleration modifier at starting position would be at its' lowest.

The acceleration modifier is at its' peak in the starting position and as the ball reaches higher towards its' intended position – the acceleration should naturally slow down. This would be the preferable distance model to use given the logic of the situation. To implement the equation and its' effect on the ball, a custom C++ function is written and implemented within the blueprint system.

```
float part1 =
    3.9996      * pow(distance, 4)
    + 3.9248    * pow(distance, 3)
    - 0.34183   * pow(distance, 2)
    + 0.007058  * pow(distance, 1)
    + 2.9682    * pow(10, -5);           //c(X1)

float part2 = -0.0585;                 //m
float part3 = pow(distance, 2);        //X1^2
float part4 = pow(amperage, 2);        //X3^2
float part5 = part1 / (part2 * part3);
float part6 = part4 * part5;

float force_strength = part6 * 10;      //for UE purposes
float force_radius = 55;               //force radius
float force_limit = 5000;              //force limit

if (force_strength > 0)
    force_strength = 0;

else if (force_strength < -force_limit)
    force_strength = -force_limit;

//GEngine->AddOnScreenDebugMessage(-1, 0, FColor::Blue,
//FString::Printf(TEXT("\n ball acceleration is %f"), force_strength));

reference_ball->AddRadialForce(force_origin, force_radius,
    force_strength, ERadialImpulseFalloff::RIF_Constant, true);
```

Figure 3. “set_distance” function code, it applies force to the ball.

Amperage is the value of coil current which is calculated by a different function, force origin is the location of one of the platforms within the simulation, it is fed into the function to determine where the centre of the force of attraction will be. Distance is the ball position that is calculated by one of the distance functions.

Force limit is the first variable introduced into the system that is there to facilitate proper functionality within the *Unreal Engine* specifically, it limits the amount of force

that can be applied to the object and is necessary due to potential clipping issues : because the acceleration modifier can reach very high values, it is possible for acceleration to be so large that the object bypasses any barriers set up to prevent movement, in reality such high acceleration cannot be achieved by the system in question, but the mathematical model allows it. The actual value of the variable will be estimated during the validation phase. Force radius determines the range in which attraction is possible and is set to encompass the area that is experimentally determined to be appropriate for the system. The force applied has no fall-off that depends on the distance from force origin – this is so because the force itself will be changing and such dynamics are not accounted for in the mathematical model.

The built-in *Unreal Engine* function used for simulating magnetism is *AddRadialForce* [12] . Several variables are used to facilitate its' work : *Origin* (Vector) to determine the location of the force, *Radius* (Float) to determine the radius of the force field, *Strength* (Float) to determine the strength of the force, *Falloff* (ERadialImpulseFalloff) which allows to control the strength of force as a function of distance from the force origin, *Accel Change* (Boolean) which if true, makes it so that the strength is taken as a change of acceleration instead of a physical force, meaning it will ignore mass – this variable is set to true since the mathematical model is designed to account for the ball acceleration and not the magnetic field.

The force applied is only allowed to be of negative value, meaning it will attract the object only, if the object must descend – attraction force is nullified and gravitational pull brings the object down.

3.2 Coil Current Acceleration Implementation

The mathematical model delivers the value of the acceleration of coil current and not the current itself. Instead of creating a separate mathematical function to calculate the value of current, a physics object is created inside the simulation level to model the behaviour of coil current based on its' acceleration.

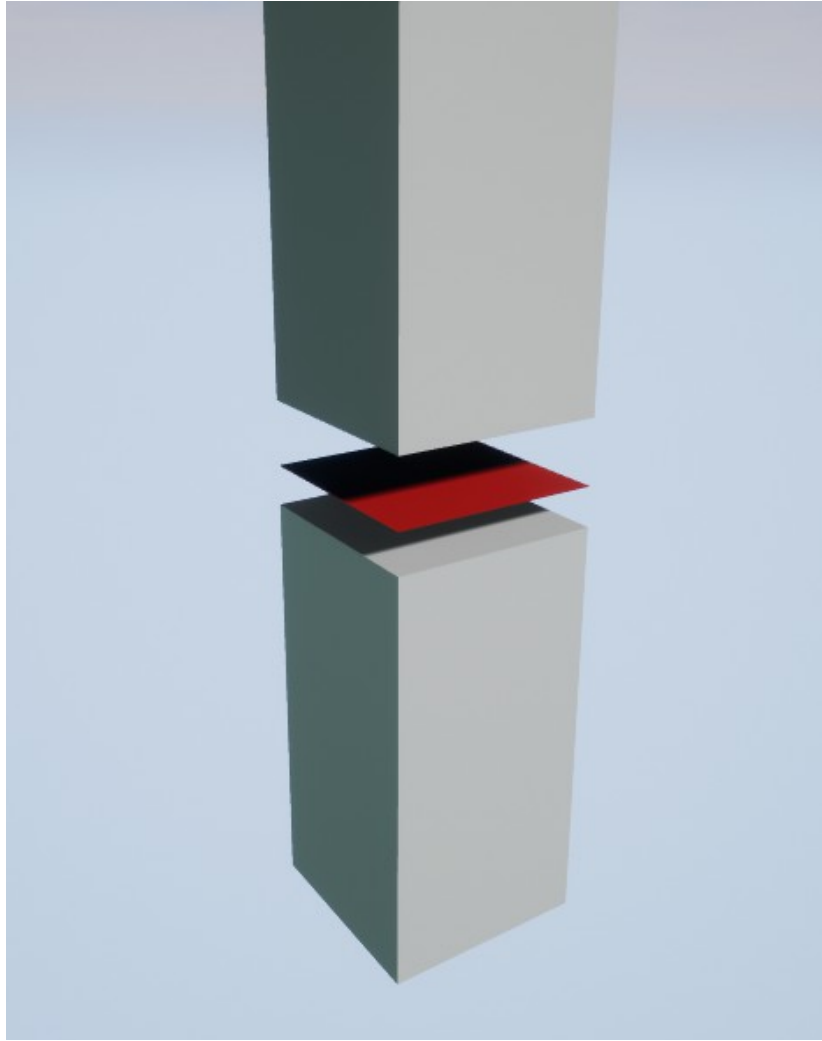


Figure 4. Coil current simulation model within the game level.

The coil current is modelled by the red plate physics object with no mass and no gravitational pull. The plate is placed between two columns with a space in between them for the plate to ascend and descend, the region in which the plate can move corresponds to the current range of 0 to 2.25 amperes.

Only the coil acceleration affects the movement of the plate, the function used to move the plate within the simulation is the same function used in magnetism simulation : *AddRadialForce*. The difference would be in that the coil current simulation does not involve radial force falloff and the force can also be of positive value to push the object downward. In short : the calculation of current based on its' acceleration is deferred to the engine and the position of the object in relation to the top of the bottom column would equal the value of the coil current if the system is properly set up.

$$\dot{x}_3 = \frac{fip_2}{fip_1} \frac{i(u) - x_3}{e^{-\frac{x_1}{fip_2}}}$$

To observe how much of an impact the ball position can have on the value of coil current acceleration the original formula will be transformed.

If all the constants in the formula were given their real values and the distance variable was accounted for in a separate function, the resulting equation would take this form :

$$\dot{x}_3 = z(x_1) \times (i(u) - x_3) \quad ,$$

where z is the coil current acceleration modifier – hereafter named amperage modifier, the value of which depends entirely on the distance of the ball, x_3 is coil current, $i(u)$ is a function of voltage that will be reviewed in this document later.

Similar to the previous section, both available distance models and their effect on acceleration modifier will be reviewed.

Table 2. Amperage modifier value calculated by distance.

Position, from floor	Amperage Modifier	Position, from ceiling	Amperage Modifier
0.0000	24.04030452	0.0155	42.82860186
0.0010	24.95285359	0.0145	41.26232006
0.0020	25.90004222	0.0135	39.75331863
0.0030	26.88318531	0.0125	38.2995028
0.0040	27.90364766	0.0115	36.89885436
0.0050	28.96284588	0.0105	35.54942893
0.0060	30.06225033	0.0095	34.24935325
0.0070	31.20338723	0.0085	32.99682255
0.0080	32.38784068	0.0075	31.79009805
0.0090	33.61725497	0.0065	30.6275046
0.0100	34.89333675	0.0055	29.50742827

0.0110	36.21785748
0.0120	37.59265587
0.0130	39.01964042
0.0140	40.50079205
0.0150	42.03816692
0.0155	42.82860186

0.0045	28.42831417
0.0035	27.38866428
0.0025	26.38703535
0.0015	25.42203693
0.0005	24.49232939
0.0000	24.04030452

The values the amperage modifier can take are much more consistent and mild than those present for the ball acceleration modifier, any differentiation in distance also produces a much smaller change. From the outset it is difficult to determine which of the distance models is more desirable, both will be tested during validation.

To impart acceleration unto the plate which models coil current, a custom C++ function is written and implemented within the blueprint system.

```
float part1 = (-0.3 * pow(voltage, 2) + 2.6 * voltage - 0.047 - amperage);
float part2 = pow(2.71, (distance / 0.026841));
float part3 = 24.04030452 * part2;
float part4 = part1 * part3;

float force_strength = part4 * -10; //special *-10 for UE functionality
float force_radius = 200;

reference_amperage->AddRadialForce(force_origin, force_radius,
    force_strength, ERadialImpulseFalloff::RIF_Constant, true);
```

Figure 5. “set_amperage” function code, it regulates amperage simulation.

Voltage is the value of voltage which is calculated by a different function, force origin is the location of the coil current platform within the simulation, it is fed into the function to determine where the centre of the force of attraction will be. Distance is the ball position that is calculated by one of the distance functions – which one specifically would be determined in the validation phase.

4 PID Controller Implementation

In the *MLS Simulink* diagram and mathematical model, voltage is considered the input signal. It is calculated using a PID controller based on the set point deviation – the proportional term, set point continuous history – the integral term, and the deviation rate of change – the derivative term.

A proportional-integral-derivative controller is a control loop feedback mechanism that continuously calculates an error value as the difference between a desired set point and a measured process variable and applies a correction based on the three terms, it is widely used in industrial control systems [13].

In the case of this project, the measured process variable is the ball position and the controller provides relevant voltage output for the system to correct any deviation. For the purposes of validation, a *Simulink* PID model has been provided by *A-Lab*.

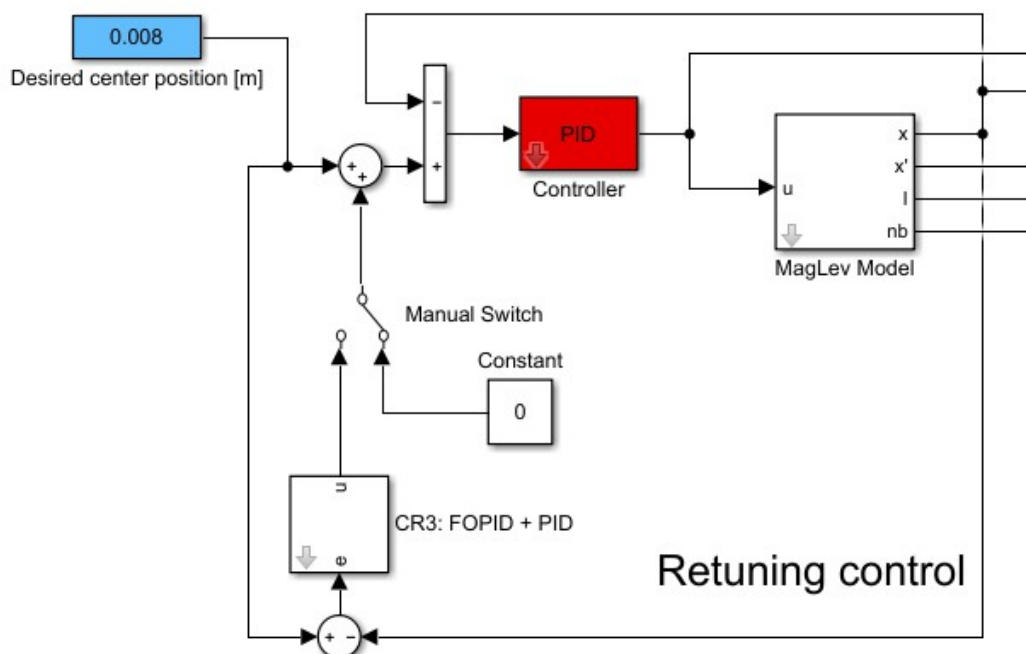


Figure 6. *MLS Simulink* diagram.

In addition to a PID controller, this diagram also contains a fractional order retuning PID controller block, which can be used to tune the set point in such a way that the PID controller output voltage would be more responsive. The figure below demonstrates this well, the simulation was run with a static set point of 0.008 m.

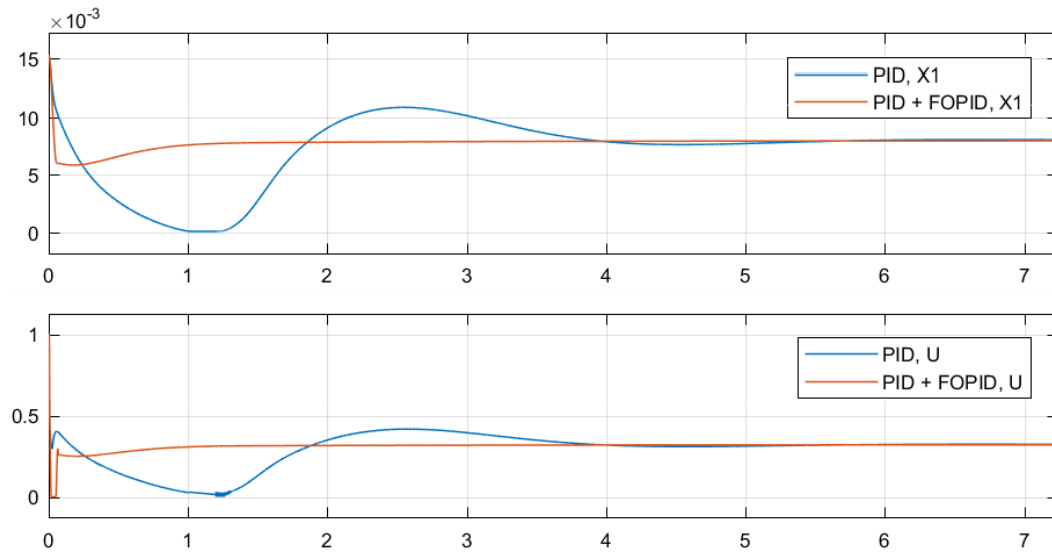


Figure 7. Ball position over time in PID and FOPID simulations.

For validation purposes, however, only the PID controller will be used. The internal logic of the controller is shown below.

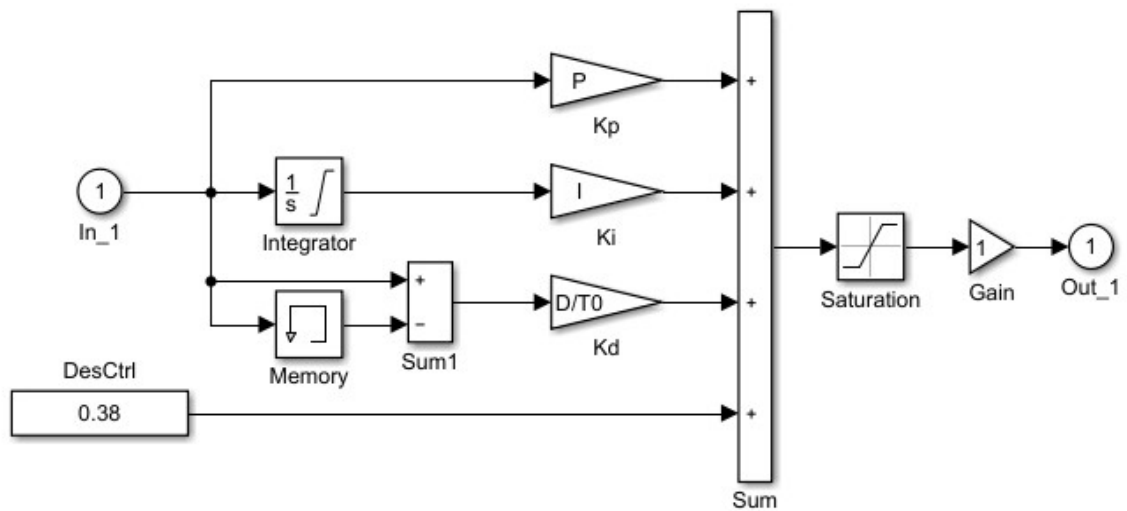


Figure 8. PID controller block internal diagram.

Individual terms are split into separate blueprint-accessible C++ functions which follow the internal diagram logic and are summarized thereafter.

4.1 Proportional Term

```
float P = setpoint_deviation;  
return P;
```

Figure 9. “p_control” function code, it outputs proportional term value.

The proportional component of the PID controller consists of straightforward multiplication of the set point deviation by the gain factor, however, this multiplication takes place in the summation portion of the code for the sake of function transparency.

4.2 Integral Term

To calculate the integral, current set point deviation is multiplied by delta time – the amount of time passed since the last integral calculation, the result is added to the current integral value. The integral, prior to multiplication by the gain factor, is bound via saturation between -1 and 1.

```
float addition = delta_time * setpoint_deviation;  
float I = integral + addition;  
  
if (I < -1)           //internal saturation from -1 to 1  
    I = -1;  
else if (I > 1)  
    I = 1;  
  
return I;
```

Figure 10. “i_control” function code, it outputs integral term value.

The integral calculation, and in fact all of the calculations within the simulation, are performed each time a frame is rendered. The frequency of frame rendering can be static or vary – in the case that it is varied, the *Unreal Engine* blueprint system provides the time passed between the current frame and the previous one – delta time.

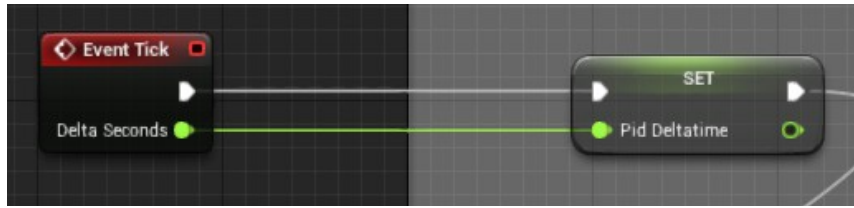


Figure 11. PID delta_time variable assignment blueprint.

In *Matlab*, the integrator block allows for internal signal saturation which limits the range of values that the signal can be [14] , it caps the signal and keeps it from exceeding the upper and lower saturation limits – the integral setup is shown below.

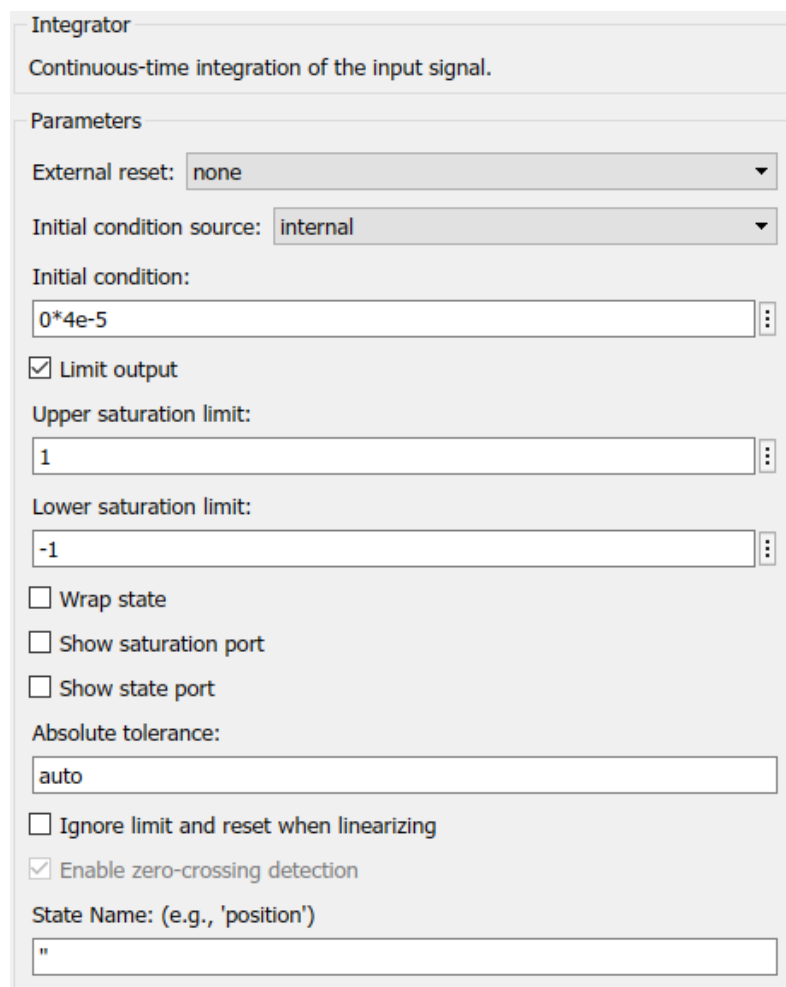


Figure 12. Integrator block options.

A blueprint variable is declared to represent the integral – it is external to the *C++* function because the function calculates only the addition that is added to the integral,

this variable is then fed into the function on each frame. Having relevant variables declared within the blueprint and not the C++ code also allows for greater visibility of the processes and reuse of those variables across different .

4.3 Derivative Term

To calculate the derivative, the change in the set point value must be visible throughout the process. To accomplish this, a new blueprint variable – set point history is used.

```
float D = (setpoint_deviation - setpoint_history) / delta_time;
return D;
```

Figure 13. “d_control” function code, it outputs derivative term value.

Calculation of a derivative can involve several steps wherein it would be necessary to introduce more variables to represent even more previous values, and there are different ways to proceed with the calculation when using those. In this case, the algorithm is simple and is concerned only with one previous value, as it is in the *Simulink* model.

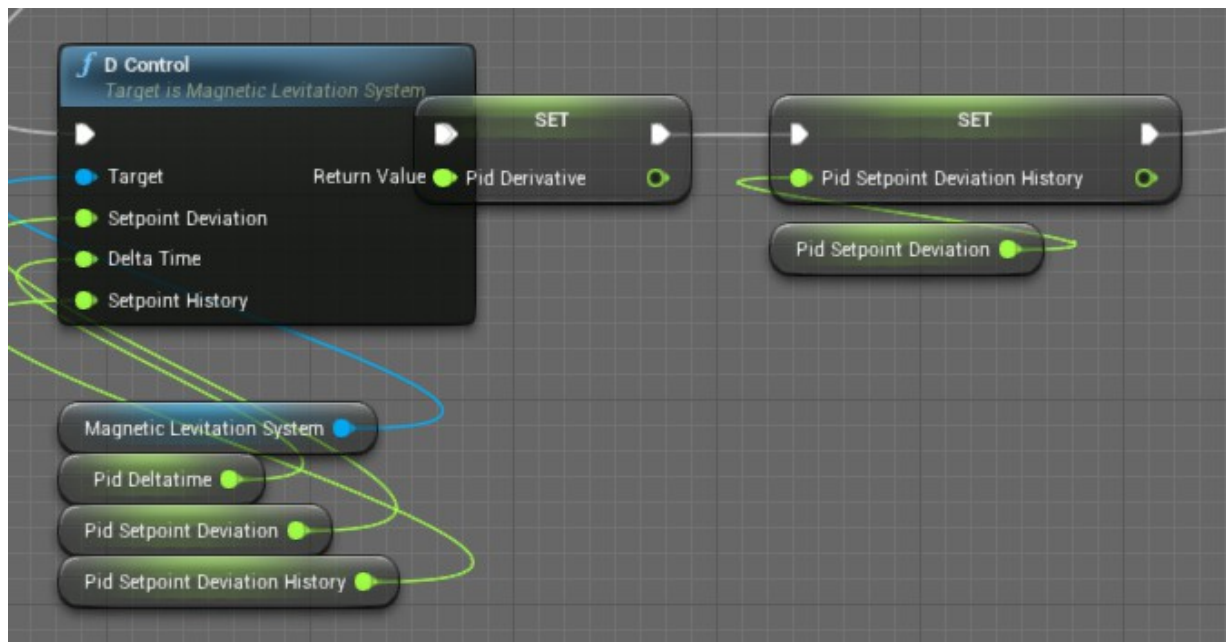


Figure 14. Derivative components blueprint flow.

Deviation history is equated to the outgoing derivative value from the function, when the next frame is rendered and the derivative calculation function is called from the

flow, ball position and set point deviation would already be altered, the previous set point history is fed into the function to calculate the new derivative value. Upon blueprint execution, the set point deviation history initial value is set to zero.

4.4 Term Summation

When all the terms have been calculated, they are first multiplied by their respective gain factors and then summarized along with an offset signal of 0.38 – which is shown in the *Simulink* diagram at the start of this chapter. Following the summation, the signal goes through a saturation block. The gain block that is used thereafter has the gain factor of 1 so it is not factored in the C++ function. The output of this summation function is the voltage input signal that is used in the mathematical model.

```
float voltage = P * Pk + I * Ik + D * Dk + offset;  
  
if (voltage > 1)           //output is saturated between 0 and 1  
    voltage = 1;  
else if (voltage < 0)  
    voltage = 0;  
  
//GEngine->AddOnScreenDebugMessage(-1, 0, FColor::Blue,  
//FString::Printf(TEXT("\n coil voltage is %f"), voltage));  
  
return voltage;
```

Figure 15. “set_voltage” function code, it outputs voltage value.

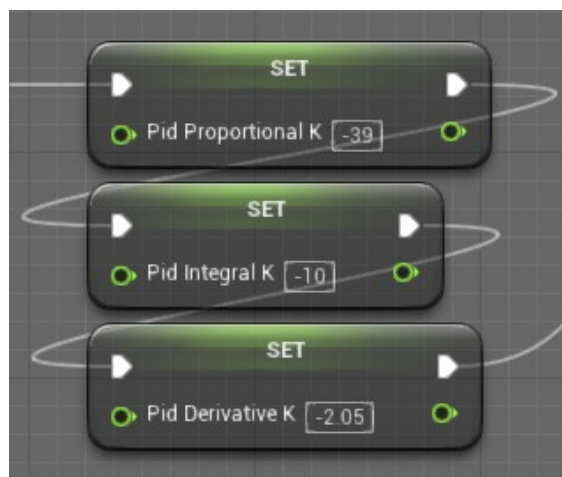


Figure 16. PID gain factors initialization blueprint.

5 Network Interface Implementation

The purpose of building a network interface between *Simulink* and *Unreal Engine* is to allow model validation : the necessary variables that represent the model inputs and outputs are sent to *Simulink* wherein they can be compared with the existing model. If the results are sufficiently divergent, the variables that govern the *Unreal Engine MLS* are adjusted until the system resembles the *Simulink* model results.

To assist in this endeavour an existing solution can be utilized. This solution is appropriated from a project that is titled : “Implementation of an Inverted Pendulum Model in Virtual Reality”. It was a part of a Bachelor's thesis authored by Aleksandr Kuzmin [15] from the same university as this project, the main purpose of this thesis was to simulate the behaviour of an inverted pendulum in virtual reality wherein the pre-existing mathematical model would be implemented wholly within *Matlab Simulink* with *Unreal Engine* being used for the graphical representation of the system and virtual reality interaction features.

5.1 Simulink Network Interface

The *Simulink* network interface would be an addition to the existing *MLS* diagram that would receive variable data from *UE* allowing real-time comparison of system inputs and outputs. The diagram would also allow to send variables to *UE* if is ever desirable to have the *Simulink* model control the *UE MLS*.

The *Simulink* environment has pre-existing diagram blocks for UDP communication protocol using defined UDP ports [16] . The configuration of this block within the *Simulink* diagram used is shown in the figure below.

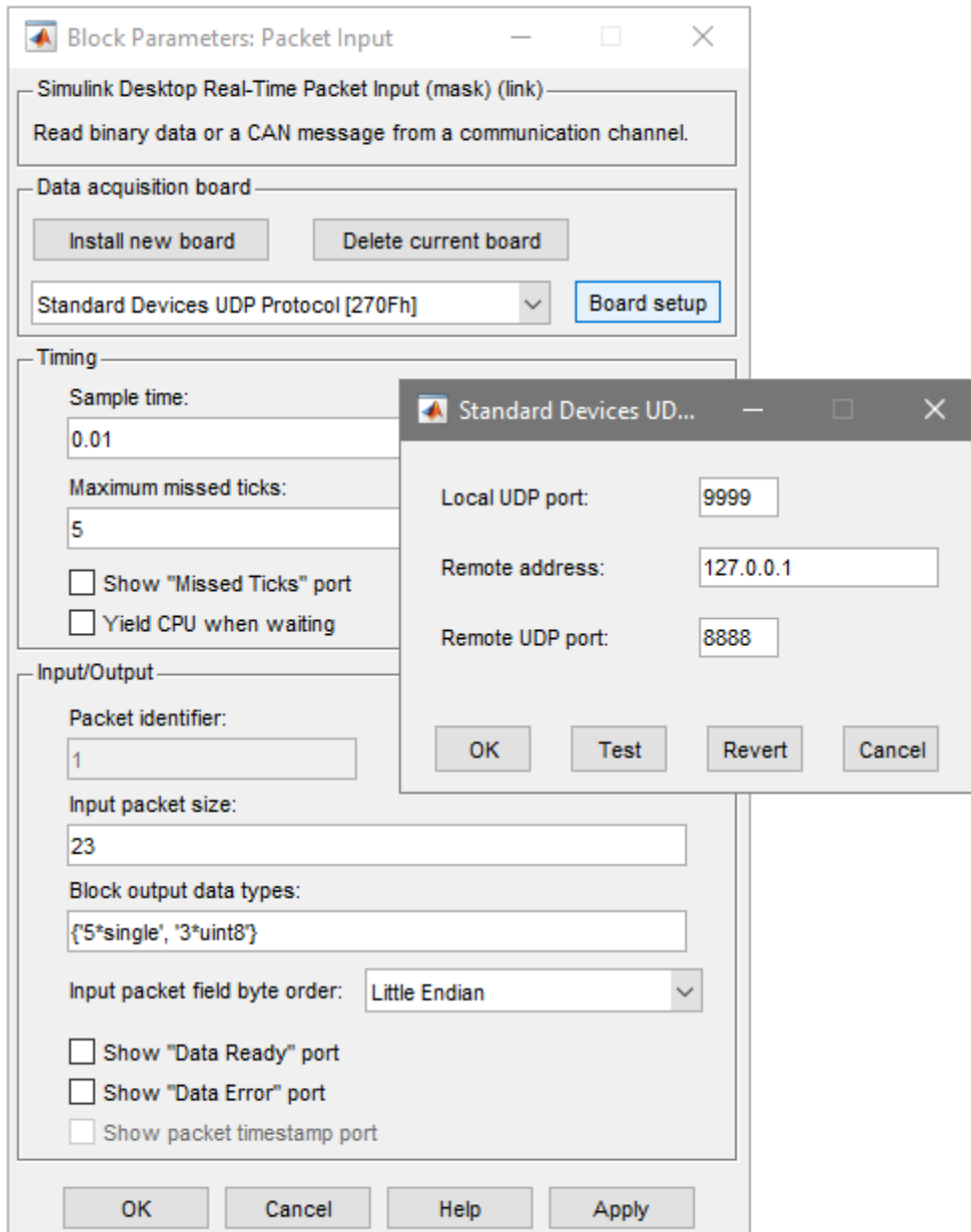


Figure 17. *Packet Input* block setup.

The important variables to consider are the local and remote UDP port numbers as well as the remote address. One particular variable that should be noted is the sample time which should correspond to the frame rate of the running *UE* simulation, for stability purposes it is best to set the simulation to run at a fixed frame rate, this can be set in the *Unreal Engine* project properties, engine general settings subcategory.

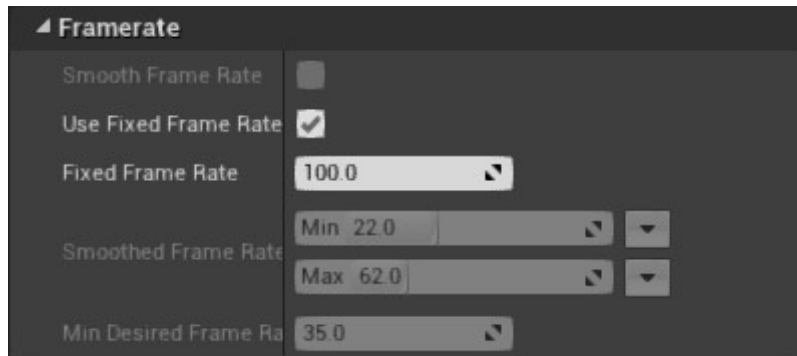


Figure 18. Project frame rate settings in *Unreal Engine*.

The frame rate of 100 frames per seconds corresponds to a sample time setting of 0.01 in the UDP packet input block. If the simulation frame rate does not correspond to the *Simulink* sample rate, data will either be lost or supplemented by empty values which reduces the accuracy of any reading. The block output data structure is inherited from Aleksandr Kuzmin's project and remains unchanged.

To test whether or not the network functionality works – on the *Windows* platform it is possible to view active computer connections via the *CMD* command : “netstat -a -o -n”. The additional command parameters are there to help identify the process responsible for an active connection – it will show the process identifier which can be traced in the *Task Manager* details tab, as shown in the figure below.

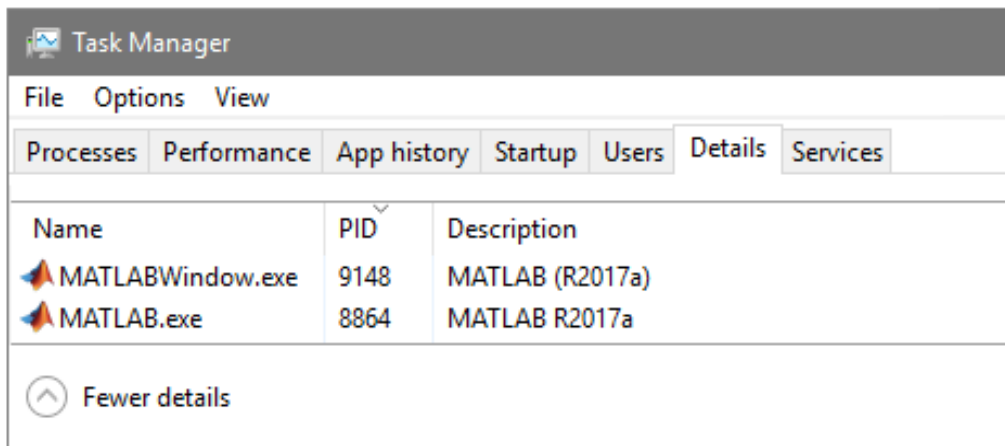


Figure 19. *Task Manager* details.


```
Active Connections
```

Proto	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	892
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:5432	0.0.0.0:0	LISTENING	3888
TCP	0.0.0.0:31415	0.0.0.0:0	LISTENING	8864
TCP	0.0.0.0:31515	0.0.0.0:0	LISTENING	8864
TCP	0.0.0.0:49664	0.0.0.0:0	LISTENING	584
TCP	0.0.0.0:49665	0.0.0.0:0	LISTENING	1292
TCP	0.0.0.0:49666	0.0.0.0:0	LISTENING	1368
TCP	0.0.0.0:49667	0.0.0.0:0	LISTENING	2036
TCP	0.0.0.0:49668	0.0.0.0:0	LISTENING	2788
TCP	0.0.0.0:49669	0.0.0.0:0	LISTENING	660
TCP	0.0.0.0:49691	0.0.0.0:0	LISTENING	668
TCP	127.0.0.1:5939	0.0.0.0:0	LISTENING	3368
TCP	127.0.0.1:5939	127.0.0.1:49680	ESTABLISHED	3368
TCP	127.0.0.1:31515	127.0.0.1:49701	ESTABLISHED	8864
TCP	127.0.0.1:49680	127.0.0.1:5939	ESTABLISHED	6836
TCP	127.0.0.1:49684	127.0.0.1:49685	ESTABLISHED	6836
TCP	127.0.0.1:49685	127.0.0.1:49684	ESTABLISHED	6836
TCP	127.0.0.1:49701	127.0.0.1:31515	ESTABLISHED	1628
TCP	192.168.0.179:139	0.0.0.0:0	LISTENING	4
TCP	192.168.0.179:49681	213.39.120.133:5938	ESTABLISHED	3368
TCP	192.168.0.179:49740	191.232.139.2:443	TIME_WAIT	0
TCP	192.168.0.179:49741	134.170.165.249:443	ESTABLISHED	6752
TCP	:::135	:::0	LISTENING	892
TCP	:::445	:::0	LISTENING	4
TCP	:::5432	:::0	LISTENING	3888
TCP	:::31415	:::0	LISTENING	8864
TCP	:::49664	:::0	LISTENING	584
TCP	:::49665	:::0	LISTENING	1292
TCP	:::49666	:::0	LISTENING	1368
TCP	:::49667	:::0	LISTENING	2036
TCP	:::49668	:::0	LISTENING	2788
TCP	:::49669	:::0	LISTENING	660
TCP	:::49691	:::0	LISTENING	668
TCP	:::1:31415	:::1:49729	TIME_WAIT	0
TCP	:::1:31415	:::1:49730	ESTABLISHED	8864
TCP	:::1:31415	:::1:49731	ESTABLISHED	8864
TCP	:::1:31415	:::1:49732	TIME_WAIT	0
TCP	:::1:31415	:::1:49738	TIME_WAIT	0
TCP	:::1:31415	:::1:49739	TIME_WAIT	0
TCP	:::1:49730	:::1:31415	ESTABLISHED	8864
TCP	:::1:49731	:::1:31415	ESTABLISHED	8864
TCP	:::1:49733	:::1:31415	TIME_WAIT	0
TCP	:::1:49734	:::1:31415	TIME_WAIT	0
TCP	:::1:49735	:::1:31415	TIME_WAIT	0
TCP	:::1:49736	:::1:31415	TIME_WAIT	0
TCP	:::1:49737	:::1:31415	TIME_WAIT	0
UDP	0.0.0.0:5050	*:*		8384
UDP	0.0.0.0:5353	*:*		2108
UDP	0.0.0.0:5355	*:*		2108
UDP	0.0.0.0:9999	*:*		8864
UDP	0.0.0.0:50440	*:*		5508
UDP	192.168.0.179:137	*:*		4
UDP	192.168.0.179:138	*:*		4
UDP	192.168.0.179:5353	*:*		3368
UDP	:::5353	*:*		2108
UDP	:::5355	*:*		2108
UDP	:::50447	*:*		3368
UDP	:::1:5353	*:*		3368
UDP	:::1:59684	*:*		3888

Figure 20. CMD command “netstat -a -o -n” result.

An active UDP connection on port 9999 has been initiated by process 8864, which corresponds to the PID of *Matlab Simulink*. The command should be entered while the simulation is running, otherwise there would be no active UDP connection to trace.

5.2 Unreal Engine Network Interface

Building a network interface in *Unreal Engine* is more complicated than *Simulink* as there aren't pre-existing assets to facilitate this sort of a connection. Standard *C++* code with network functionality dependencies is used to build independent functions that create network sockets and blueprints initialize these functions when the simulation is running. The relevant *C++* code that was appropriated consists of following files :

1. Andmed.h
Contains declaration of a custom *C++* data structure;
2. Iteraator.cpp and Iteraator.h
A counter;
3. Saatja.cpp and Saatja.h
Sends data to a network socket;
4. VastuVotja.cpp and VastuVotja.h
Receives data from a network socket.

“Andmed” is a custom data structure that contains relevant variable values, its' code was left unaltered in this project and is shown in the figure below.

```
float float1 = 0.0f;  
float float2 = 0.0f;  
float float3 = 0.0f;  
float float4 = 0.0f;  
float float5 = 0.0f;  
  
uint8 uint1 = 0;  
uint8 uint2 = 0;  
uint8 uint3 = 0;
```

Figure 21. “Andmed.h” structure variable declaration code portion.

The structure should correspond to the block output data type that was specified in the *Simulink* packet input and output blocks, it describes the inner construction of the sent packet, the actual variables to be packed into it can be adjusted within the *MLS* blueprint and within the *Simulink* diagram.

Previously, C++ functions would have been called from a blueprint directly, in this case, however, blueprint interfaces would be used. Blueprint interfaces are collections of functions in name only [17], they serve to expose functions and variables that are present within a blueprint to be called by other blueprints.

The two specific blueprint interfaces : InvpDSendInterface and InvpDDeliverInterface serve to expose the functions to send and receive data with relevant variables being passed throughout. These are necessary because the network functionality blueprints are segregated from the *MLS* blueprint, the interfaces are used for communication.

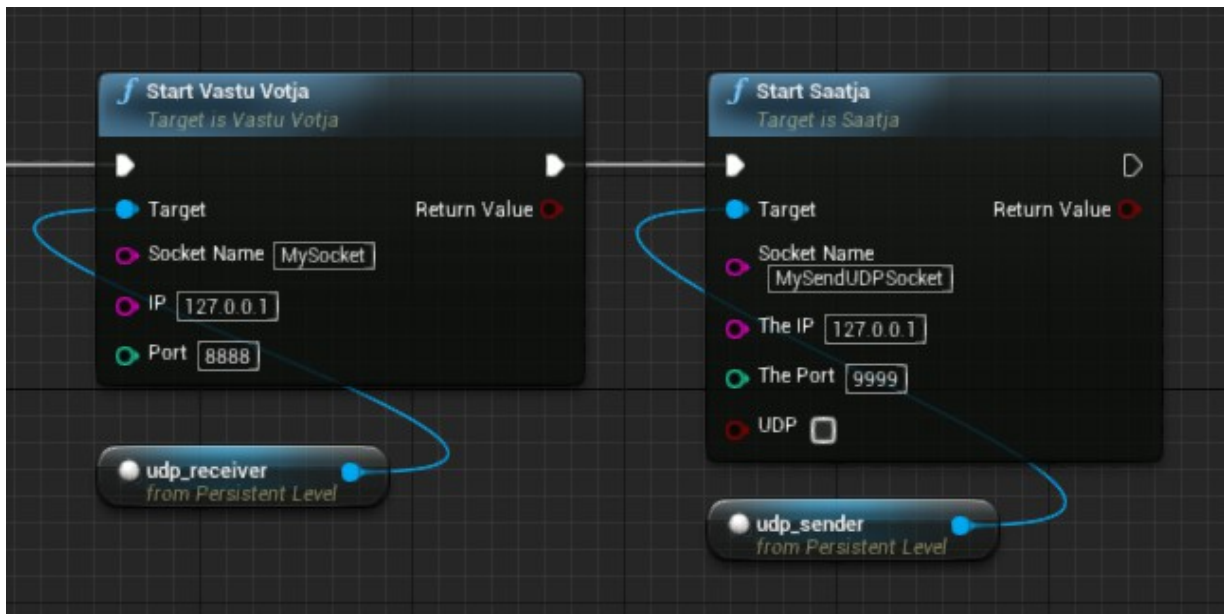


Figure 22. Network components initialization blueprint.

When the level is loaded, both of the network components are initialized from the level blueprint using the provided values, the networking functionality can be then tested when the simulation is running. Once the network interfaces work independently they can be tested in conjunction – this will not be shown in this document.

6 System Validation

The purpose of validation is to confirm the built system functions as intended. The adapted network interface allows for the data flow between the two software environments. Before testing for the validity of the model itself, it is important to revisit one of the previously unanswered questions : the coil current distance model.

6.1 Coil Current Acceleration Modifier Model Selection

Previously, there were two distance models proposed which determine the value of coil current modifier. To decide which of the two distance models is more suitable, current and voltage relationship must be observed – to do so the *Simulink* diagram is simulated.

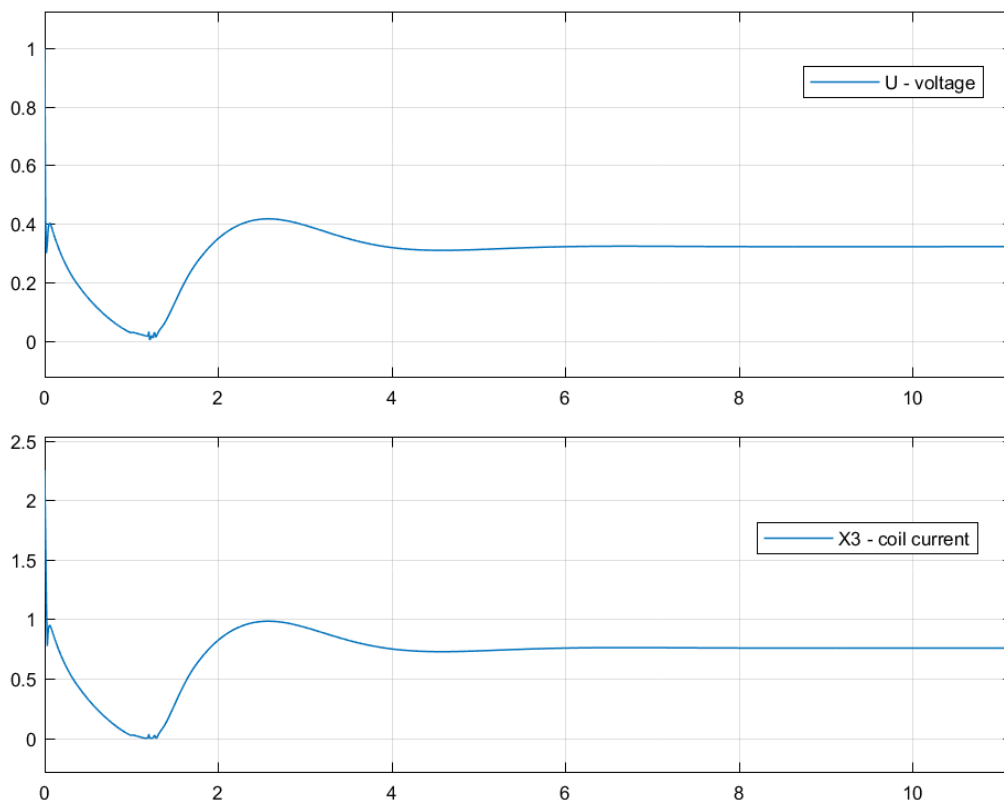


Figure 23. *Simulink* diagram voltage and coil current characteristics.

Based on the figure above, there is a great similarity in the characteristics of voltage and current with the difference being that of amplitude. Although it is difficult to see, at the starting moment of the simulation, there is a very short spike of both voltage and current towards their respective maximums : 1 V for voltage and 2.25 A for current.

The *Unreal Engine* model will be simulated next. The results are fed into *Simulink*. The first distance model to be simulated would be the ascending model wherein the ball position is calculated from the platform floor.

The similarity in characteristics is also observed within this simulation. At the beginning of the simulation, voltage is quite erratic but coil current is incapable of following such quick changes, it is also incapable of reaching its' maximum unless voltage is consistently high, like in the latter portion of the simulation.

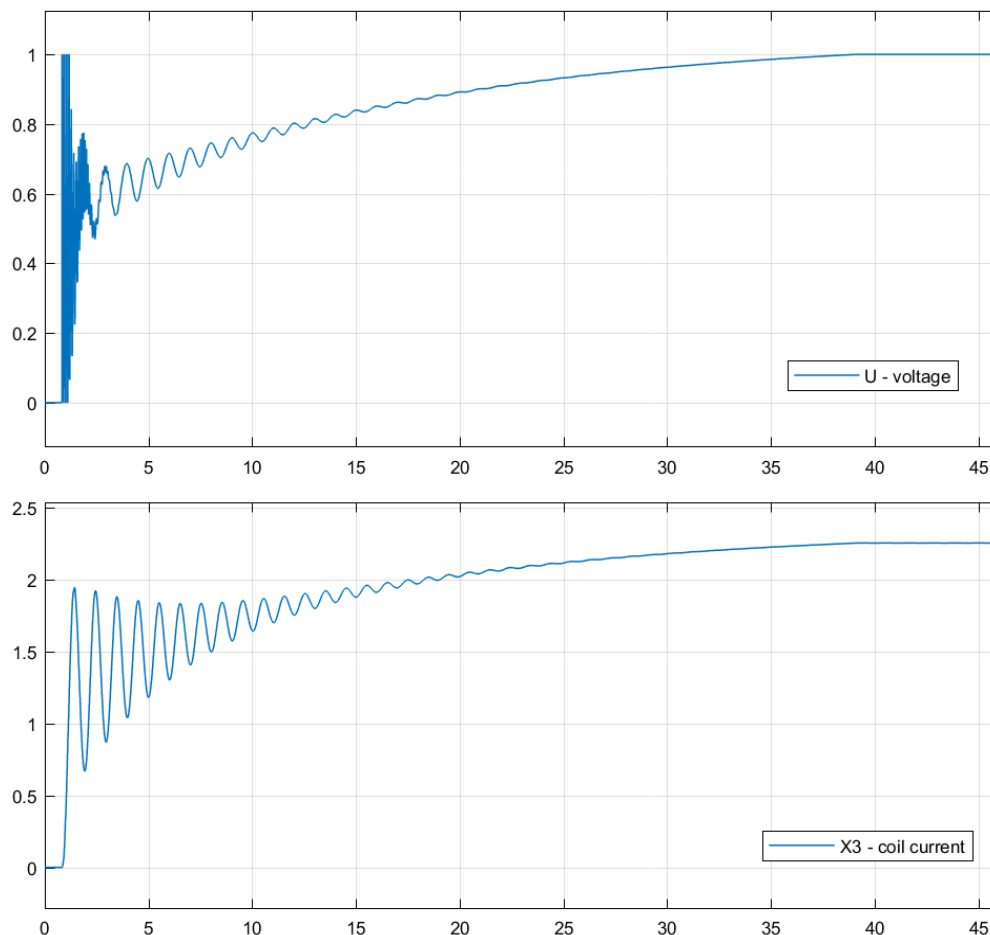


Figure 24. *Unreal Engine* MLS voltage and coil current characteristics.

The descending distance model is simulated next – ball position is calculated from the platform ceiling. This distance model ensures that the current acceleration modifier is at its' highest when the literal ball position is furthest from the ceiling.

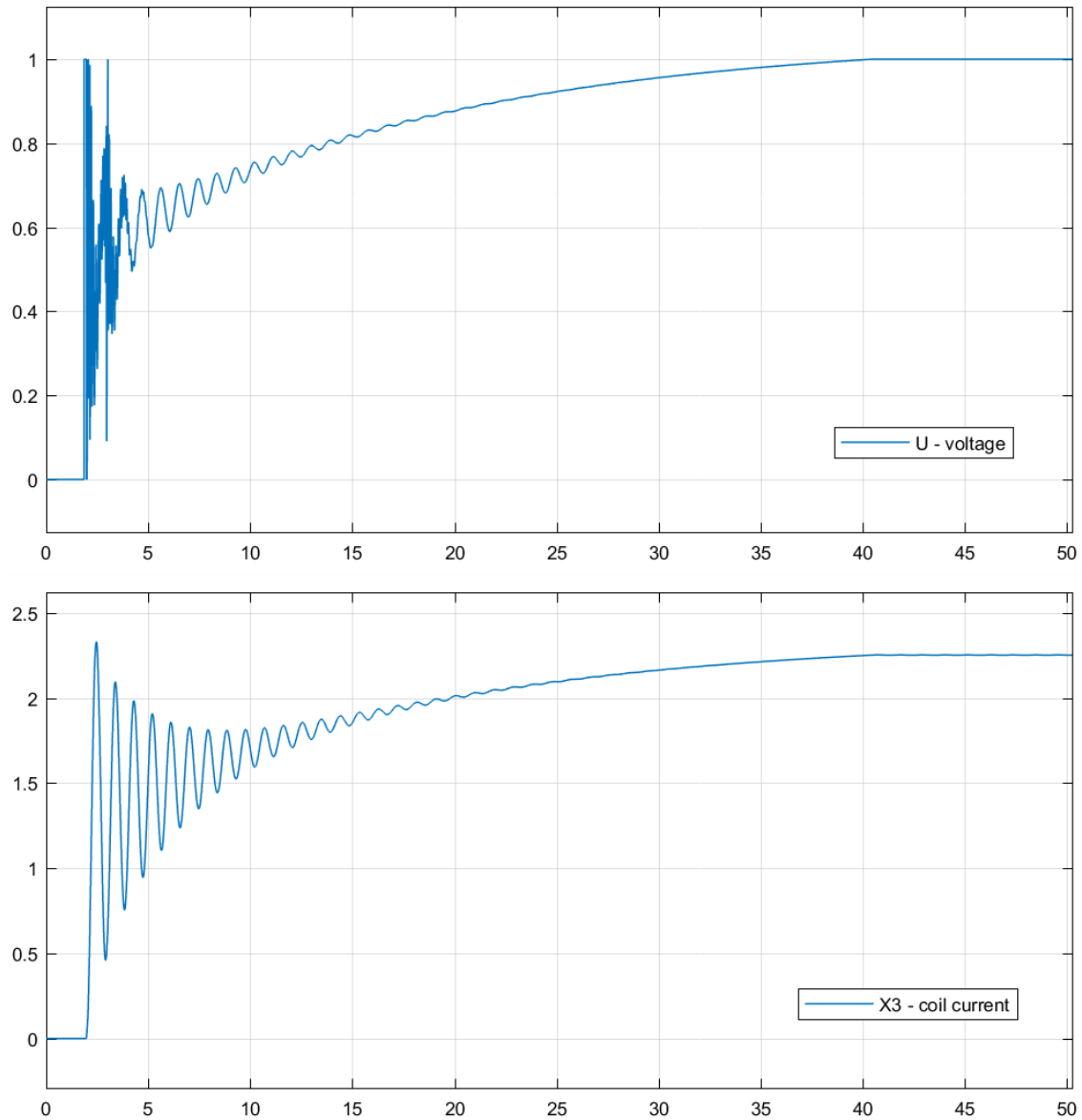
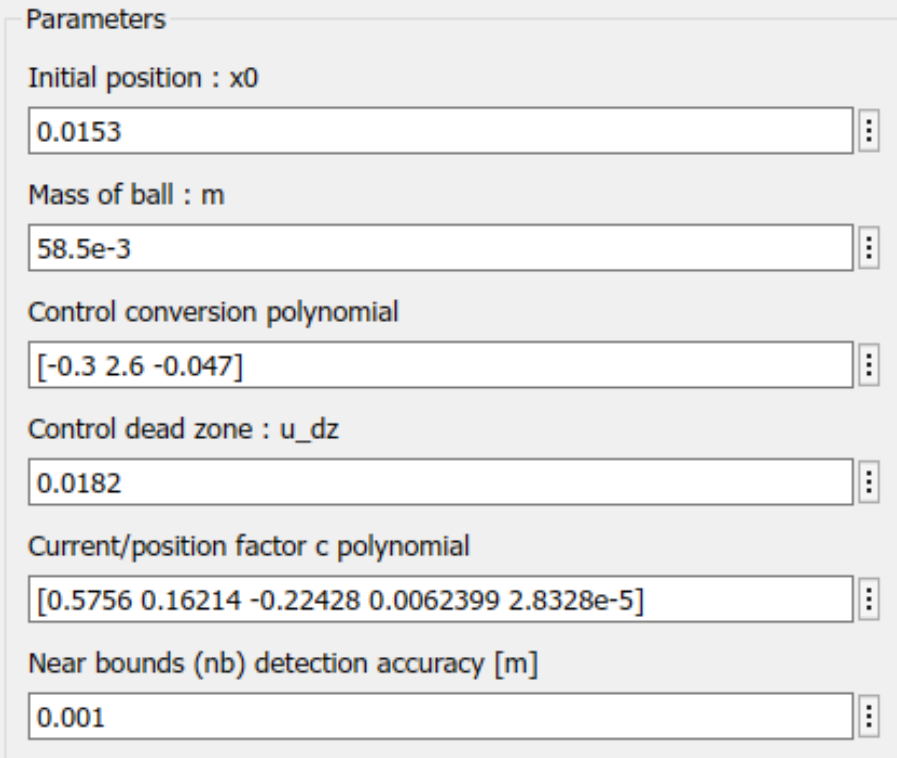


Figure 25. Voltage and coil current during simulation.

Results are generally similar, but coil current is more responsive – it has managed to reach its' maximum at simulation start. Because of this, this distance model would be used for calculating coil current acceleration.

6.2 Unreal Engine Magnetic Levitation System Validation

Before beginning *MLS* validation, the first step would be to replicate the conditions of the simulation in both environments – the relevant parameters for the *Simulink* model are shown below.



The image shows a screenshot of the 'Parameters' section for a 'MagLev Model' block in Simulink. The parameters are listed as follows:

Parameter Name	Value
Initial position : x0	0.0153
Mass of ball : m	58.5e-3
Control conversion polynomial	[-0.3 2.6 -0.047]
Control dead zone : u_dz	0.0182
Current/position factor c polynomial	[0.5756 0.16214 -0.22428 0.0062399 2.8328e-5]
Near bounds (nb) detection accuracy [m]	0.001

Figure 26. MagLev Model block parameters.

The primary parameter of interest is the initial position of the ball, the *Simulink* diagram is built to have the ball close to the top of the platform ceiling when simulation starts.

The figure below shows the results of simulation of both *Simulink* and *Unreal Engine MLS* so that the results can be compared. *Unreal Engine MLS* is also simulated with the starting ball position on the platform floor. The *Simulink* diagram, however, cannot be simulated with this ball position – the diagram would not simulate properly.

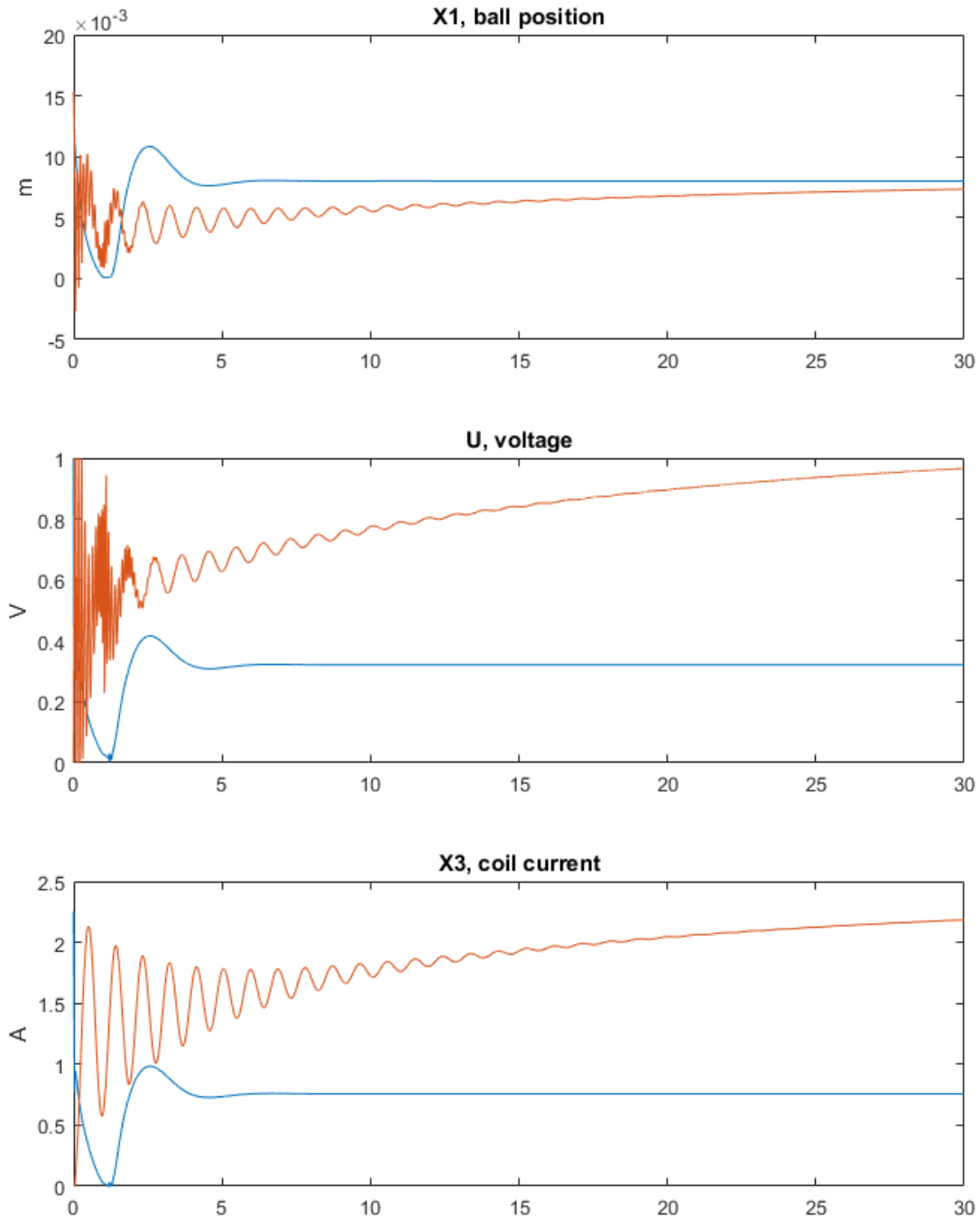


Figure 27. *Simulink* and *UE MLS* simulation results.

In the figure above, *Simulink* results are in blue, *UE MLS* – in orange. This is the first time the *MLS* is actually run and while the results are not great, they are not disastrous.

In the figure below, the *UE MLS* is simulated by itself with the starting ball position on the platform floor.

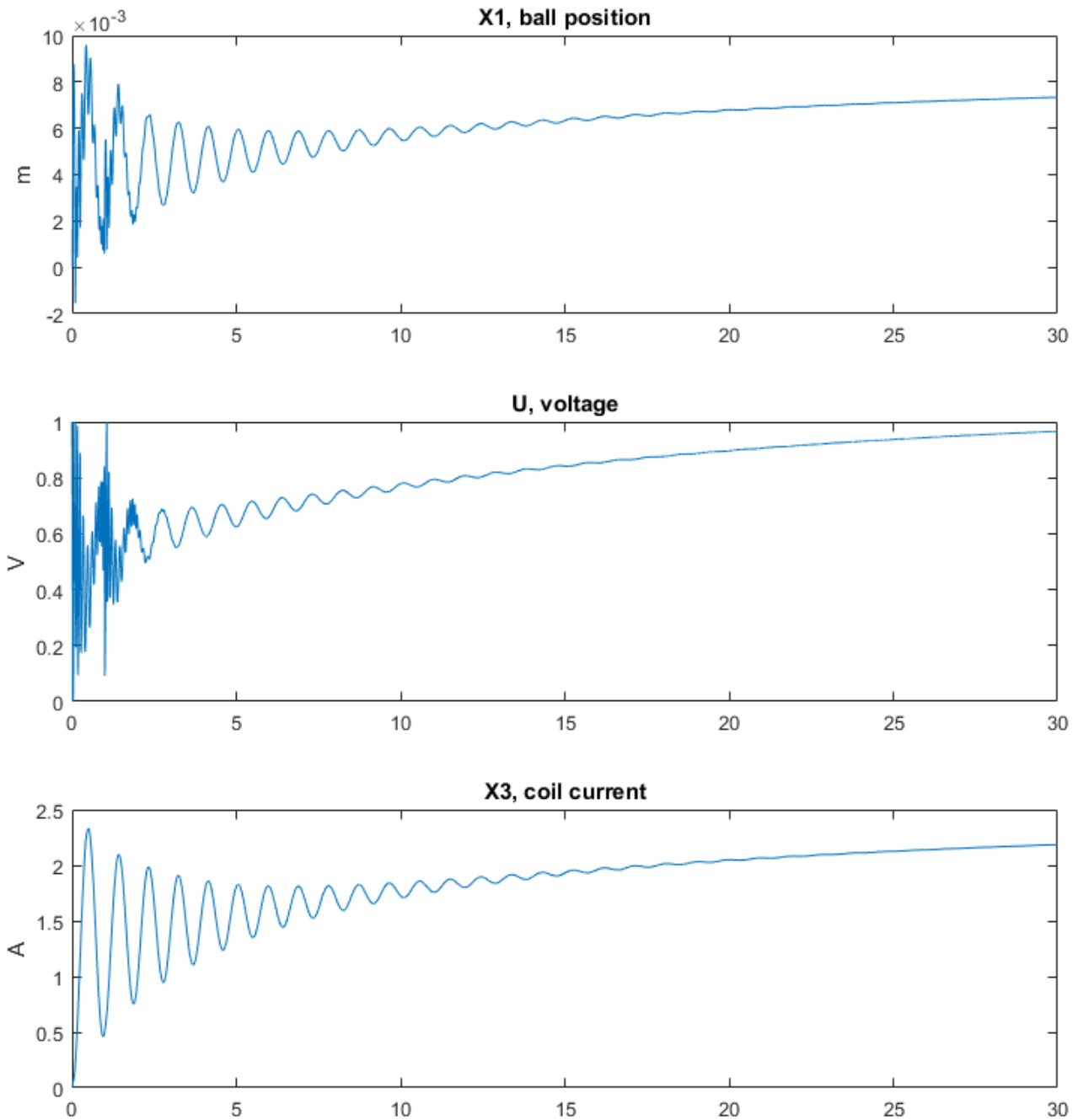


Figure 28. *UE MLS* simulation results with different initial conditions.

The system response is quite slow and it can be seen in both figures that over an extended period of time the ball position is slowly ascending.

To improve performance, there are several key variables that can be improved upon : force modifier – which is simply used to increase the amount of force exerted upon the

ball for a given value of coil current which is in turn reliant upon voltage, and PID controller gain factors – these will determine the amount of voltage generated depending on how the set point deviation behaves.

After numerous experiments of changing the variables and measuring the results, an acceptable model was found – the results are shown in the figure below.

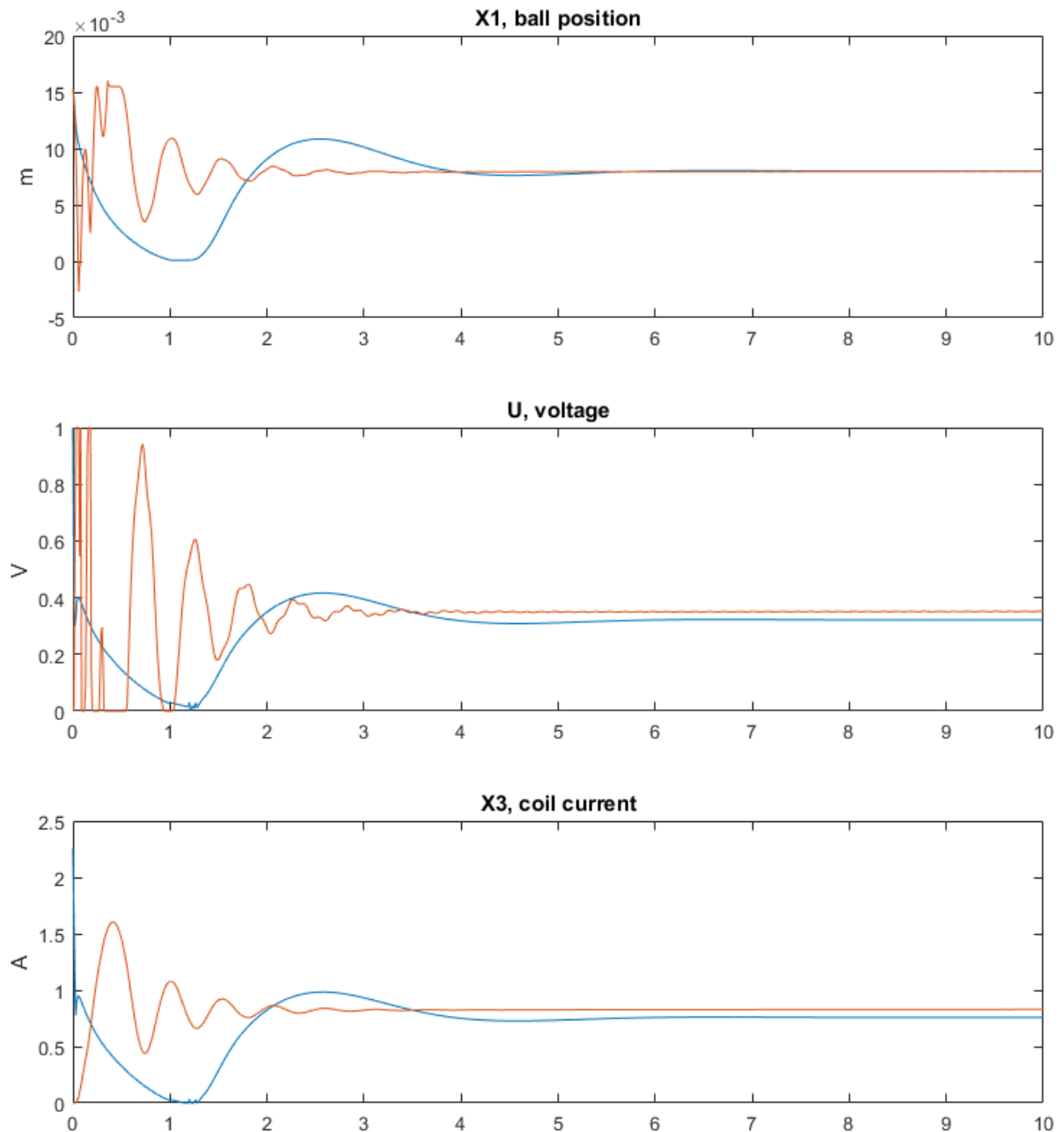


Figure 29. *Simulink* and *UE MLS* simulation results after modification.

Once again, the *Simulink* model results are in blue, *UE MLS* – in orange. The time it takes for the modified system to reach the set point is actually lower than the *Simulink* model, albeit this comes at the cost of a more sensitive PID controller – this can be seen by the voltage signal characteristic. The figure below demonstrates *MLS* performance when the initial ball position is the platform floor.

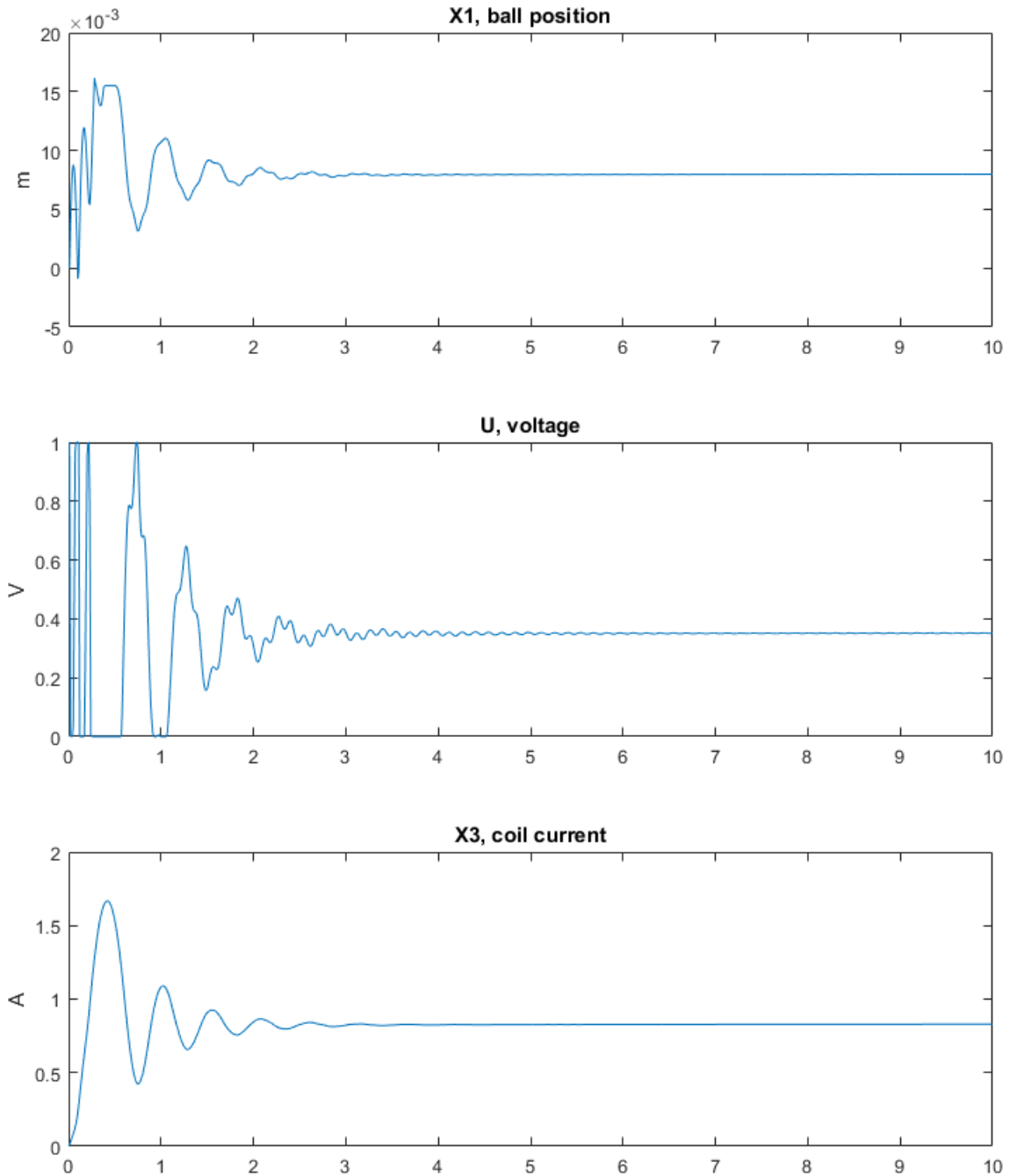


Figure 30. *UE* modified *MLS* simulation results with different initial conditions.

In the modified *UE MLS*, the magnetic force applied was magnified by a factor of 8. This is justified when looking at *Simulink* simulation results : the amount of current that is necessary to generate enough attraction to hold the ball at a set point of 0.008 is much less than the current the unmodified *UE MLS* was providing. In the following figures that demonstrate the output of the modified system, it can be seen that the current and voltage of *Simulink* and *UE* are much more similar. Further variables that were modified were the PID gain factors : the proportional component gain factor was changed from -39 to -120, the integral component gain factor was changed from -10 to -30, the derivative component gain factor was changed from -2.05 to -4.

The change in the mentioned variables affects the degree of impact they can have upon each other and the system output, however, the fundamental mathematical model that describes the relationship between them remains unchanged.

As a short stress test, the system was subjected to a minor and major disturbances – the figures below demonstrate the response.

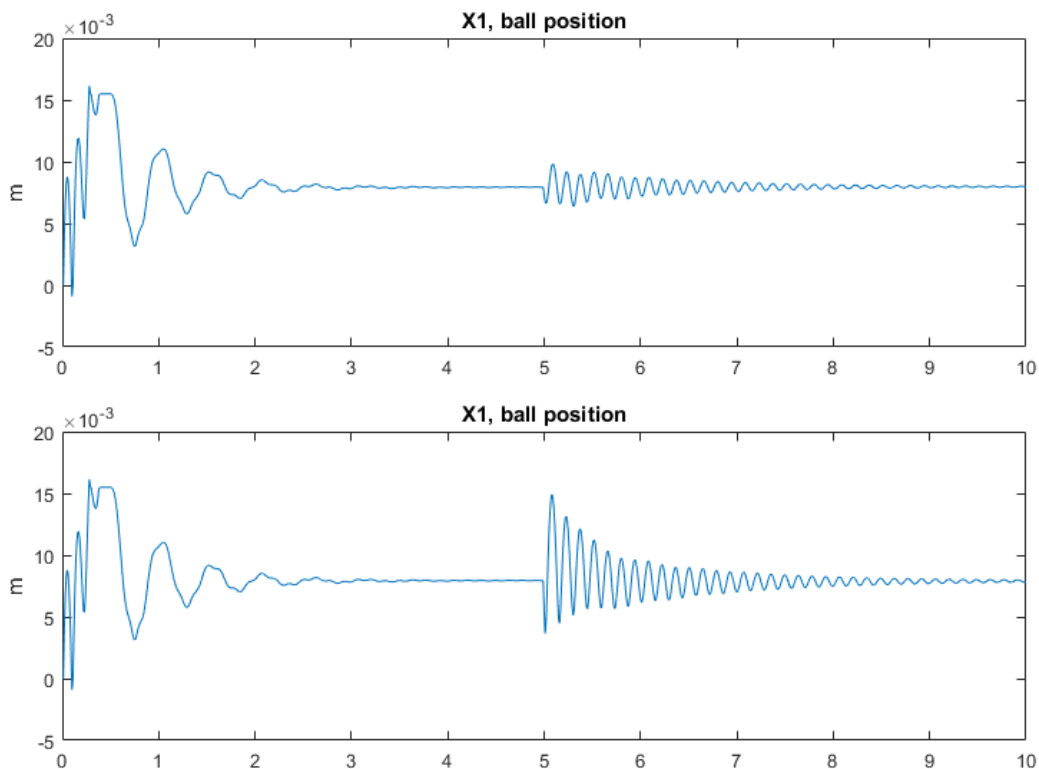


Figure 31. Modified *UE MLS* response to a minor and major disturbance.

7 Virtual Reality Implementation

Introduction of virtual reality into this project would imply this : the operator can observe and explore the level environment and interact with the physical objects and processes within it. With implementation of VR, it is generally necessary to reduce graphical fidelity or scene complexity to improve performance, but this would be unnecessary in such a small project.

To assist in a timely completion of this task, similar to the network interface implementation, the relevant blueprints can be migrated from Alexandr Kuzmin's project, no C++ code is used this time due to *Unreal Engine's* native support for VR.

In *Unreal Engine*, various components responsible for individual functionality build upon and attach to each other to provide features. Virtual reality in its' basic form is a camera component – that which determines what the player sees, a motion controller component – that which determines what the different buttons or manoeuvres with the motion controllers actually do, and a player pawn component – the abstract object within *Unreal Engine* which is controlled by the player or is the representation of the player, as such – the motion control and camera components attach to the player pawn.

Once the blueprints components are migrated to the project and compiled they can be tested, however, *Unreal Engine* requires for VR hardware to be connected and operational for testing any such features. Because this project has been in development in a remote location and with no access to *Virtual Reality* equipment – testing and implementation will be conducted at a later date and will not be documented here.

8 Level Design

With all of the functional components built and tested, attention is diverted towards the graphical component of the project, though there are also a few minor aspects related to functional performance that will be discussed in this section.

The assets for building the visual component of the level were provided by *A-lab*. Once placed, the components are easily arranged into the right order. The resulting level environment is shown below.



Figure 32. Finished *MLS* level environment.

One of the problems that arose with the imported assets is their incompatibility with *PhysX* – specifically, the collision detection, the figure below demonstrates this.

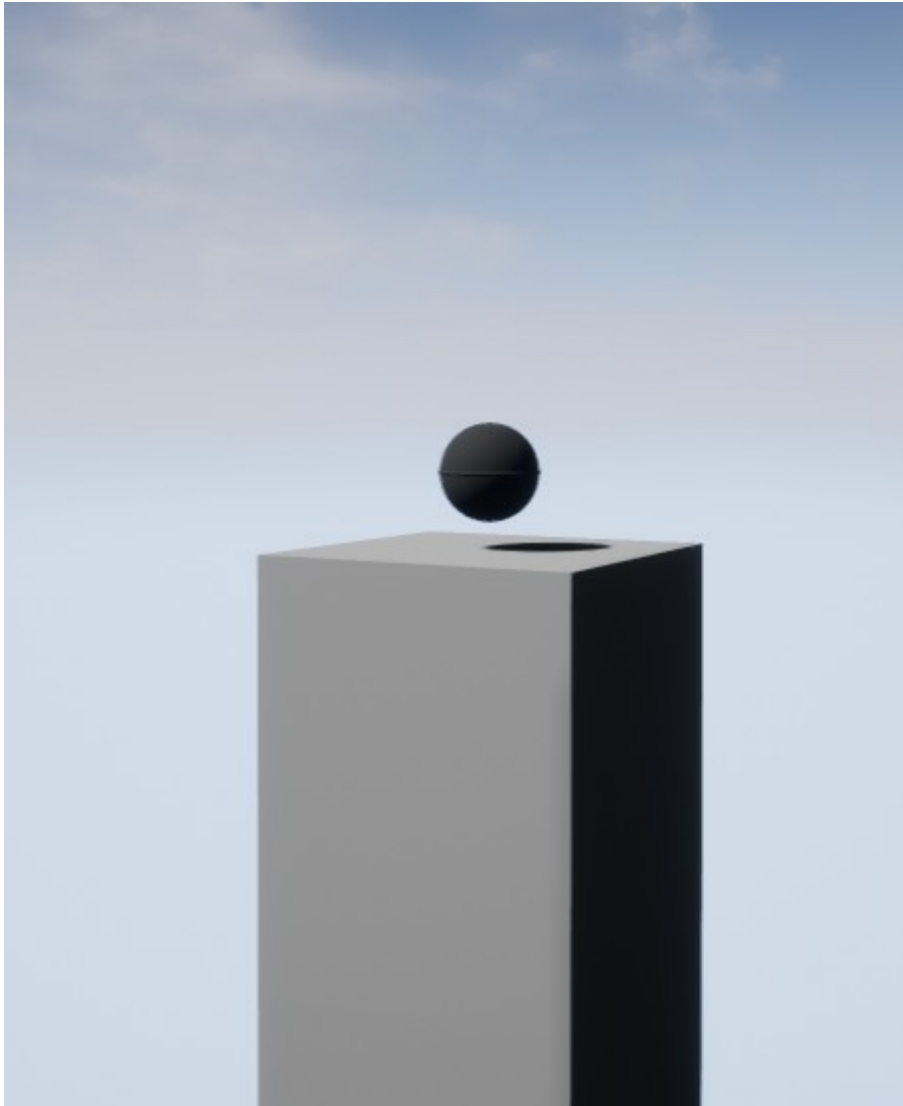


Figure 33. Imported ball mesh resting on a platform.

The imported asset is resting on a platform, the object is not suspended in air. *Unreal Engine's* collision detection finds the outer exterior of the asset to be larger than it is. This is not unique to the ball – all other assets suffer from the same problem. To correct such a problem, native *Unreal Engine* assets can be used instead : these are basic shapes that are included within the engine.

The ball asset is replaced with a basic sphere of the exact geometry and size which is native to the engine. There is a slight visual difference, but the collision detection is perfect and this is absolutely necessary for the sort of precise calculations performed within the system on a relatively small scale.

The ball must interact with the MLS platform, which is the yellow apparatus that actually imparts the magnetism. This platform is, however, an imported asset. To amend this, basic columns are added into the level that will act as collision barriers. These columns are carefully placed so that they align perfectly with the platform ceiling and floor, the MLS platform itself has collision disabled so that it does not interfere with the ball movement. At run-time, the columns are rendered invisible – this is shown below.

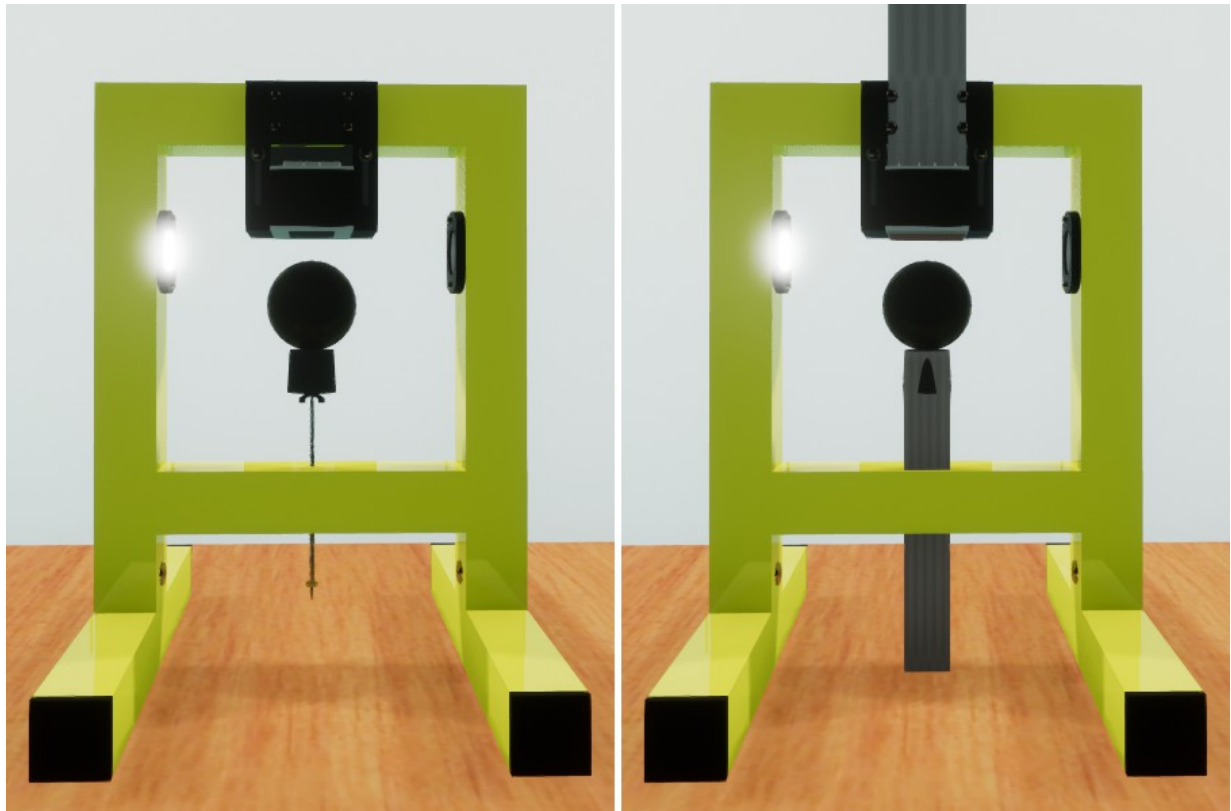


Figure 34. Collision columns during run-time and development.

9 Conclusion

At the current stage of the development, the project is available to be exported as a functional application. However, it would be optimal to test VR features when the opportunity arises. Further improvements can also be made to the control algorithm if more data is available, for example : connecting the virtual system to the real magnetic levitation system and performing experiments and stress tests to generate data for use in additional validation.

One of the difficulties of such a project was the non-iterative process of development. Whenever a piece of functionality was built – it was not possible to unit test the working condition independently because it was inherently reliant on other components to work properly. As a result, much of the functionality was only tested when the project was near completion and the multitude of issues that arose were more difficult to identify since there were several possible suspects. Migrating code and blueprint components between different projects in *Unreal Engine* is also problematic if the engine versions used in the projects are different, which they were.

The assets and components that were helpful in realizing this project were provided by Alpha Control Laboratory [18] .

The primary objective of this project was to create a dynamic interactive magnetic levitation system and this has been adequately accomplished.

References

- [1] Vasily Ryzhonkov, “The rise of VR & AR era. Why this time is different?”, [Online]. Available: <https://www.slideshare.net/VRyzhonkov/the-rise-of-vr-ar-era-why-this-time-is-different>. [Accessed 10.01.2018].
- [2] Official website for Epic Games, Inc., Unreal Engine, [Online]. Available: <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>. [Accessed 10.01.2018].
- [3] Official website for Epic Games, Inc., Unreal Engine documentation, [Online]. Available: <https://docs.unrealengine.com/latest/INT/>. [Accessed 10.01.2018].
- [4] Official website for Epic Games, Inc., Unreal Engine documentation, “Setting Up Visual Studio for UE4”, [Online]. Available: <https://docs.unrealengine.com/latest/INT/Programming/Development/VisualStudioSetup/>. [Accessed 10.01.2018].
- [5] Official website for Microsoft Corporation, Visual Studio, [Online]. Available: <https://www.visualstudio.com/vs/community/>. [Accessed 10.01.2018].
- [6] Official website for Epic Games, Inc., Unreal Engine documentation, “Blueprints Visual Scripting”, [Online]. Available: <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/> [Accessed 10.01.2018].
- [7] Official website for BigLevel Software, “Model-Driven Development and Product Line Engineering”, [Online]. Available: http://www.biglever.com/technotes/mdd_spl.html. [Accessed 10.01.2018].
- [8] Official website for Epic Games, Inc., Unreal Engine documentation, “Physics Simulation”, [Online]. Available: <https://docs.unrealengine.com/latest/INT/Engine/Physics/>. [Accessed 10.01.2018].
- [9] Official website for The Math Works, Inc., Matlab Documentation, [Online]. Available: <https://www.mathworks.com/help/matlab/>. [Accessed 10.01.2018].
- [10] Official website for The Math Works, Inc., Simulink Documentation, [Online]. Available: <https://www.mathworks.com/help/simulink/>. [Accessed 10.01.2018].
- [11] Tepljakov A., E. Petlenkov, J. Belikov, E. A. Gonzalez, “Design of retuning fractional PID controllers for a closed-loop magnetic levitation control system”, [Online]. Available: <https://a-lab.ee/research/publications/228>. [Accessed 10.01.2018].
- [12] Official website for Epic Games, Inc., Unreal Engine documentation, “Add Radial Force”, [Online]. Available: <https://docs.unrealengine.com/latest/INT/BlueprintAPI/Physics/AddRadialForce/>. [Accessed 10.01.2018].
- [13] Wikimedia Foundation, Inc., “PID controller”, [Online]. Available: https://en.wikipedia.org/wiki/PID_controller. [Accessed 10.01.2018].

- [14] Official website for The Math Works, Inc., Documentation, “Saturation”, [Online]. Available: <https://uk.mathworks.com/help/simulink/slref/saturation.html>. [Accessed 10.01.2018].
- [15] Aleksandr Kuzmin, “Implementation of an Inverted Pendulum Model in Virtual Reality”, [Online]. Available: <http://a-lab.ee/edu/theses/defended/1239>. [Accessed 10.01.2018].
- [16] Official website for The Math Works, Inc., Documentation, “Packet Input/Output”, [Online]. Available: <https://uk.mathworks.com/help/sldrt/examples/packet-input-output.html>. [Accessed 10.01.2018].
- [17] Official website for Epic Games, Inc., Unreal Engine documentation, “Blueprint Interface”, [Online]. Available: <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/Interface/>. [Accessed 10.01.2018].
- [18] Official website for Alpha Control Lab, [Online]. Available: <https://a-lab.ee/>. [Accessed 10.01.2018].