Physics Simulation and Animation for Virtual Football

Master's thesis

July 2025

Riccardo Nicora

Technical University of Denmark (DTU)
M.Sc. in Human-Centered Artificial Intelligence

Abstract

This thesis presents a physics-based simulation framework for football motion, developed with the aim of achieving realistic behavior and responsiveness suitable for use in video games. The work is structured around two main components: the modeling of ball dynamics and the development of shot prediction systems.

The physical model captures key forces acting on the ball, including drag, lift, bounce, and ground friction, using empirical aerodynamic formulations. Several numerical integration schemes are evaluated—Explicit Euler, Semi-Implicit Euler, Velocity Verlet, and Runge-Kutta methods—based on their accuracy and computational performance in real-time contexts.

The second part of the thesis focuses on shot prediction systems designed to estimate the ball's initial parameters needed to reach a specific target. Two approaches are implemented: one that computes the optimal velocity for a given spin and speed, and another that determines both velocity and spin to guide the ball toward the target while also passing as close as possible to an intermediate constraint. Both models are evaluated in terms of accuracy and computational efficiency, and their respective limitations are also discussed The results demonstrate that the proposed framework delivers reliable, controllable, and physically plausible simulations, with strong potential for real-time gameplay integration.

Contents

Chapter 1

Introduction

Physics-based simulation plays a fundamental role in many interactive applications, from scientific computing to real-time video games. In the context of sports games, accurately modeling physical behavior is essential to achieving realistic, immersive, and responsive gameplay. This thesis focuses on the simulation of a football, with the goal of reproducing its motion in a way that is both physically plausible and controllable, while ensuring performance suitable for real-time applications Accurate physical simulations involve complex, nonlinear dynamics that cannot be solved analytically. As a result, numerical methods must be used to approximate the motion over time. These methods are inherently subject to instability and numerical error, and are often non-trivial to implement correctly. Achieving consistent and predictable behavior under real-time constraints thus represents a significant challenge.

This thesis is being developed in collaboration with GATTINI.games, a newly founded independent studio currently working on a football video game. The team is building their own engine and core systems, including a custom physics engine and a shot prediction system. While their development is already well underway, this thesis aims to provide complementary insights and explore alternative solutions. Their support has been valuable throughout the process, offering guidance, feedback, and testing resources that helped shape and evaluate the work presented here.

1.1 Background and Motivation

Unity is one of the most widely used game engines, valued for its accessible and flexible development tools. It includes a built-in physics engine, based on NVIDIA PhysX, which provides general-purpose rigid body simulation with features such as collision detection, drag, and friction. While sufficient for many gameplay scenarios, PhysX is not optimized for realistic physical modeling—particularly in the case of a spinning ball, where several important forces are not taken into account.

More advanced physics solutions in Unity, such as Unity Physics (DOTS) and Havok, offer better performance and scalability. However, they remain general-purpose engines and lack support for domain-specific forces that are crucial to accurately simulate football dynamics.

To address these limitations, this thesis proposes a custom, physically plausible simulation system tailored to football behavior. The goal is to strike a balance between realism and computational efficiency, enabling detailed yet controllable ball behavior suitable for gameplay.

In addition to simulation, the thesis also explores the problem of ball control. Specifically, it investigates trajectory prediction systems capable of computing the optimal initial velocity and spin needed to direct the ball toward a target. These systems are designed to support responsive and satisfying shooting mechanics.

1.2 Objectives

The main objective of this work is to develop a controllable and plausible physics simulation framework for football motion, suitable for use in video games. The project is structured around three core goals:

- **Physical Modeling:** Create a physical model that includes the main forces acting on a football while keeping the system flexible for tuning and extension.

- **Numerical Integration:** Select a numerical integration method that provides the best compromise between accuracy, stability, and computational efficiency for real-time simulation.
- **Prediction Systems:** Develop two prediction models capable of computing the initial conditions needed to reach a target. One model handles a single target constraint, while the other considers an additional intermediate constraint. Both systems should be fast, accurate, and reliable.

1.3 Methodology Overview

The development of this thesis follows a structured and iterative methodology, starting with an investigation into the physical forces acting on a football. The goal of this phase is to identify the key forces affecting a football's motion. Once the relevant forces are selected, they are modeled mathematically and implemented in C#. The ball's behavior, particularly its interaction with the environment, is also analyzed and abstracted into a state machine to manage transitions between different motion states.

Following the physical modeling, various numerical integration methods are studied and evaluated to determine which ones best suit real-time simulation. These methods are implemented and tested to analyze their performance, accuracy and stability.

With the simulation framework established, the focus shifts to designing and implementing a shot prediction system capable of estimating the initial parameters required to hit a given target. This predictor is first developed using a basic procedure, then refined through successive testing to improve accuracy and convergence. Two main models are created and compared with each other through performance and error-based evaluations. Additionally, tests are carried out against GATTINI.games' internal model to qualitatively assess the behavior and effectiveness of the proposed approach.

Finally, a more advanced prediction method is introduced that incorporates an additional constraint point between the ball's starting position and the final target. This system is developed following a similar process, from design to implementation and testing, with the same focus on accuracy and performance.

1.4 Related Work

Not many studies have been found regarding real-time simulations of football physics, as companies involved in the development of football video games do not publish technical reports detailing their methods or implementations. The academic works that are available focus primarily on sports analysis and training support rather than on real-time applications. For instance, the studies by Zhu et al.[2] and Javorova and Ivanov[3] aim to analyze ball trajectories for free kicks or general motion under controlled conditions, while Egoyan et al. [4] explore how environmental factors like wind and altitude affect the ball's path. These works contribute valuable insights but do not address computational constraints or practical considerations for interactive systems.

One notable exception is the study by Li et al. [1], which proposes a prediction system designed to estimate the initial values required to send the ball to a target. However, the method is described at a high level, and details regarding the actual numerical implementation are lacking.

Moreover, all these works focus exclusively on the flight phase of the ball, considering only aerodynamic effects such as drag and the Magnus force. They do not cover the interaction of the ball with the ground, such as friction, rolling, and bounce dynamics, nor do they discuss the numerical methods used for integration—an essential aspect for both accuracy and real-time performance.

In addition to these simulation-focused works, further research was carried out to understand the fundamental physics of ball motion, extending beyond the treatment found in the previously cited papers. Notably, works such as Goff [5] and Kiratidis and Leinweber [6] provide empirical models for drag and lift coefficients based on spin and velocity, allowing simulation of aerodynamic forces. Tuplin et al.[7] offer a simulation-based analysis of spin degradation due to air resistance—an effect often neglected in simpler models. Mencke et al.[8] explore both flight and bounce dynamics of spinning sports balls, offering valuable insight into the physics that occur upon contact with the ground.

Given the absence of ground interaction models in most ball trajectory simulations, additional references were studied to incorporate frictional behavior and bouncing more realistically. Works by Cross [9, 10] and Jia [11] provide models and experimental insights into the transition from sliding to rolling and the torque effects generated by ground friction. These models contributed to a more realistic representation of ball-ground interaction in this thesis, which was essential to simulate rolling, sliding, and the transition between these two states.

To support the accurate and stable simulation of these physical behaviors, further research was done on numerical integration methods, including Explicit Euler, Semi-implicit Euler, Velocity Verlet, and Runge-Kutta methods [12, 13, 14]. These were analyzed with attention to both their implementation feasibility and their computational characteristics under real-time constraints.

Another critical component of this research focused on quaternion mathematics, which is crucial for simulating the rotation of a rigid body like a football. Most prior work does not detail how orientation is updated in the simulation, or uses approximations that may fail for fast-rotating objects.

In summary, while several academic papers have explored football trajectory simulation, they are largely analytical and not intended for real-time applications. Their focus is on accurate modeling of flight dynamics, often neglecting interaction with the ground and computational feasibility. This thesis extends the existing literature by incorporating ground physics, analyzing and implementing integration techniques suitable for game environments, and employing quaternion-based rotational modeling. Moreover, it places particular emphasis on balancing realism and efficiency for use in interactive systems like video games—an area where academic literature is currently limited and industry methods are often proprietary or undocumented.

Chapter 2

Physics of a ball

In order to simulate a football realistically, it is essential to understand and model the physical principles governing its motion.

A ball in flight and in contact with the ground is subject to a wide range of forces and effects, from basic gravity to complex aerodynamic interactions. These forces influence not only the trajectory of the ball but also its spin, bounces, and ground interactions.

This chapter provides a detailed overview of the physical model used to simulate a football. It begins by describing the main forces acting on a ball moving in the air: gravity, buoyancy, drag, and the Magnus effect. It also includes less commonly modeled effects such as spin degradation due to air resistance.

Following the modeling of in-air dynamics, the chapter addresses ground interactions, including bouncing and frictional effects. The ball's behavior upon hitting the ground is described using impulse-based methods, while frictional effects, whether it slides or rolls, are modeled using directional friction approaches.

2.1 Ball Flight Mechanics

2.1.1 Gravity and Buoyancy

In a vacuum, the only force acting on a ball is gravity, which acts in the negative direction of the y-axis, following the equation:

$$F_G = -mg \tag{2.1}$$

However, when a ball moves through the air, additional forces become relevant. Any object immersed in a fluid experiences an upward force. According to Archimedes' principle, this force is equal to the weight of the fluid displaced by the object:

$$F_B = \frac{4}{3}\pi r^3 \rho\, g, \tag{2.2}$$

where ρ is the density of the air and g is the gravitational acceleration. The buoyant force is usually very small compared to other aerodynamic forces; however, in the case of a football, its value is around 1.5% of the ball's weight. This force is directed upwards and therefore acts to increase the ball's height. [8]

2.1.2 Drag force

As the ball moves through the air, it is slowed down by air resistance, a force opposing the motion, given by:

$$F_D = \frac{1}{2} C_D A |\vec{v}| \vec{v} \tag{2.3}$$

Here, C_D is the drag coefficient, $\vec{v}$ is the relative velocity of the ball with respect to the fluid and $A = \pi r^2$ is the cross-sectional area of the ball, approximated as a sphere. This formulation, that considers the relative velocity, allows to consider the effect of the wind [8].

The drag coefficient C_D depends on two aerodynamic parameters: the Reynolds number Re and the spin parameter Sp. The Reynolds number is defined as:

$$Re = \frac{vD}{\nu} \tag{2.4}$$

where ν is the kinematic viscosity of air. The spin parameter is defined as:

$$Sp = \frac{r\omega}{v} \tag{2.5}$$

It measures the ratio of the ball's tangential speed at the equator to its center-of-mass speed relative to the air.

Based on experimental data from[5], the drag coefficient was fitted with the following equation:

$$C_D = a + \frac{b}{1 + \exp\left(\frac{v - v_c}{v_s}\right)}, \tag{2.6}$$

where $a = 0.155$, $b = 0.346$, $v_c = 12.19$ m/s, and $v_s = 1.309$ m/s. Here, v_c is the critical speed at which the flow around the ball transitions from laminar to turbulent. This transition causes a significant drop in the drag coefficient. The scaling speed v_s determines how sharply the drag coefficient changes during this transition.

However, when spin is taken into account, C_D does not appear to depend strongly on Re for speeds above the critical value v_c. Therefore, for $v > v_c$ and $Sp > 0.05$, we use the alternative fitting function:

$$C_D = c\, Sp^d, \tag{2.7}$$

where $c = 0.4127$ and $d = 0.3056$. Otherwise, equation (2.6) is used [5].

2.1.3 Magnus force

Spin modifies the trajectory of the ball due to a force known as the Magnus force, which is responsible for the curve ball effect. This force arises from a pressure difference on opposite sides of the ball and acts perpendicular to both the aerodynamic velocity v and the angular velocity w, as described by the following formula:

$$F_M = \frac{1}{2} C_L A |\vec{v}| \vec{v}, \tag{2.8}$$

where C_L is the lift coefficient [8].

Similar to the drag coefficient C_D, the lift coefficient depends on the Reynolds number Re and the spin parameter Sp. The paper [6] models it using a spin-dependent power law:

$$C_L^{fit} = \alpha\, Sp^{\beta}, \tag{2.9}$$

with $\alpha = 1.15$ and $\beta = 0.83$. To reflect the strong inverse relationship between lift and drag, the lift coefficient is then scaled using the drag coefficient as follows:

$$C_L = C_L^{fit} \left(\frac{C_{D|Re=0} - \min(C_D, C_{D|Re=0})}{C_{D|Re=0} - C_D^{\text{ref}}} \right), \tag{2.10}$$

where $C_{D|Re=0}$ is the drag coefficient at $Re = 0$ (corresponding to the maximum value of C_D), and C_D^{ref} is a reference value representing the minimum drag in the turbulent regime. In this case, C_D^{ref} is taken as the asymptotic value of the sigmoid function (2.6) as v tends to infinity. The equation (2.10) shows that, as C_D increases (typically at lower Reynolds numbers), C_L decreases, and vice versa.

2.1.4 Spin degradation

The friction between the ball and the surrounding air particles not only opposes the ball's motion but also affects its rotation. This interaction generates a torque that resists the ball's angular velocity, causing the spin to decay over time [7].

The total tangential force acting due to this friction is given by:

$$F_T = \frac{1}{2} \rho\, U_T^2\, C_F\, S \tag{2.11}$$

Here, the surface area of the ball S is approximated as a uniform sphere $S = 4\pi r^2$. However, the model used simplifies the ball as consisting of two plates: one on the high-pressure side and one on the low-pressure side, each with area $A = B = \frac{1}{2}S$ [7].

The tangential velocity at each plate is estimated as:

$$\begin{aligned} U_{T+} &= U_\infty + r_h\omega \\ U_{T-} &= U_\infty - r_h\omega \end{aligned} \tag{2.12}$$

where U_∞ is the translational speed of the ball, ω is the angular velocity, and r_h is the radial distance from the spin axis at height h. In this model, $h = \frac{r}{2}$, so $r_h = \frac{r}{2}$ [7].

The skin friction coefficients on the two plates are given by:

$$C_{F\pm} = 0.0074 \left(\frac{U_{T\pm}\rho D}{\eta}\right)^{-\frac{1}{5}}, \tag{2.13}$$

where D is the diameter of the ball and $\eta = \nu\rho$ is the dynamic viscosity of air [7].

Substituting into equation (2.11), we obtain the total tangential force:

$$F_T = \rho\pi r^2 \left[U_{T+}^2\, C_{F+} + U_{T-}^2\, C_{F-}\right] \tag{2.14}$$

The resulting torque [7], estimated at the center of the plates, is

$$T = F_T\,\frac{r}{2}. \tag{2.15}$$

2.2 Bounce Response and Restitution

The bounce of the ball can be calculated using an impulse-based approach, applying the linear and angular impulse-momentum theorems along with directional coefficients of restitution, in the following way:

$$\begin{cases} v_{x,2} = \frac{1-\alpha e_x}{\alpha+1} v_{x,1} - \frac{\alpha(1+e_x)}{\alpha+1} r\omega_{z,1} \\ v_{y,2} = -e_y v_{y,1} \\ v_{z,2} = \frac{1-\alpha e_z}{\alpha+1} v_{z,1} + \frac{\alpha(e_z+1)}{\alpha+1} r\omega_{x,1} \end{cases} \tag{2.16}$$

$$\begin{cases} \omega_{x,2} = \frac{\alpha-e_z}{\alpha+1}\omega_{x,1} + \frac{e_z+1}{\alpha+1}\frac{v_{z,1}}{r} \\ \omega_{y,2} = \omega_{y,1} \\ \omega_{z,2} = \frac{\alpha-e_x}{\alpha+1}\omega_{z,1} - \frac{e_x+1}{\alpha+1}\frac{v_{x,1}}{r} \end{cases} \tag{2.17}$$

Here, e_y is the coefficient of restitution in the vertical direction, while $e_{x,z}$ are the coefficients of restitution in the horizontal directions. The parameter α is the dimensionless constant in the moment of inertia $I = \alpha m r^2$, and for a hollow ball, $\alpha = 2/3$ [8].

The vertical coefficient e_y only influences the vertical velocity after the collision, while the horizontal coefficients $e_{x,z}$ affect both linear and angular velocities due to tangential interactions. However, the cited paper approximates the ball as a sphere, and thus does not include energy loss from spin around the vertical axis. To account for this dissipation, a new restitution coefficient e_{ω_y} is added to the second equation of (2.17):

$$\omega_{y,2} = e_{\omega_y}\omega_{y,1}. \tag{2.18}$$

The use of an impulse-based method to handle collisions is particularly effective in this type of simulation. Not only is it simple to implement, but it also produces results comparable to force-based approaches, which are often more complex and can lead to instability [18].

2.3 Ground Friction

When a ball stops bouncing after being kicked, one of two things can happen: it may begin sliding on the ground, or it may start rolling. If it slides, it will eventually transition to rolling once the contact point of the ball comes to rest relative to the ground. In these two distinct cases, the forces acting on the ball differ.

2.3.1 Sliding

When the ball is sliding on the ground, the contact point has a relative velocity given by

$$s = v + r\cos\theta\,\hat{z} \times \omega, \tag{2.19}$$

where θ is the angle between the spin axis and the ground.

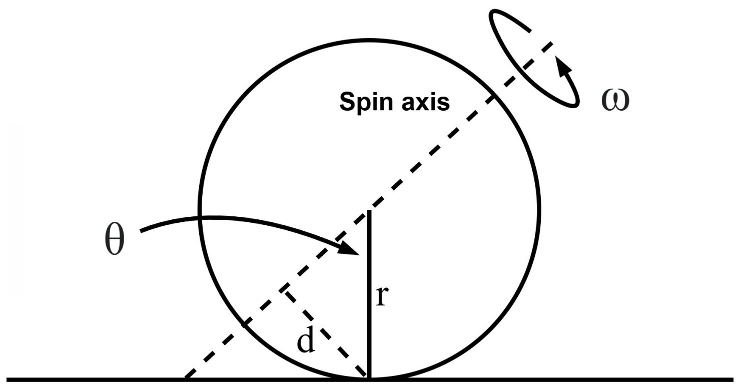

Figure 2.1: Diagram of a spinning ball sliding on a surface. The spin axis is tilted by an angle θ with respect to the ground. The contact point lies at a distance $d = r\cos\theta$ from the rotation axis [9].

The ball experiences a kinetic friction force:

$$F_s = -\mu_s N \hat{s}, \tag{2.20}$$

where N is the normal force, $\hat{s} = \frac{s}{\|s\|}$ is the direction of sliding, and μ_s is the coefficient of sliding friction.

This friction results in the following linear and angular accelerations:

$$\dot{v} = -\frac{\mu_s N}{m}\hat{s}, \tag{2.21}$$

$$\dot{\omega} = \frac{\mu_s N}{I}\,\hat{z} \times \hat{s}, \tag{2.22}$$

as derived from [10, 11].

2.3.2 Pure Rolling

Once the sliding stops and the contact point is momentarily at rest with respect to the ground, the ball enters a pure rolling regime, characterized by the no-slip condition $s = 0$. In this state, the velocity and angular velocity satisfy $v = r\omega$, and the friction force becomes:

$$F_r = -\mu_r N \hat{v}, \tag{2.23}$$

where μ_r is the coefficient of rolling friction, and $\hat{v} = \frac{v}{\|v\|}$ is the direction of motion [11]. This generates an acceleration of the ball given by:

$$\dot{v} = -\frac{\mu_r N \hat{s}}{m}, \tag{2.24}$$

Moreover, to satisfy the no-slip condition $v = \omega R$, which leads to $a = \alpha R$, the angular acceleration is:

$$\dot{\omega} = \frac{\|\dot{v}\|}{R}\,\hat{z} \times \hat{s} = -\frac{\mu_r N}{mR}\hat{z} \times \hat{s}, \tag{2.25}$$

In addition, to account for the dissipation of spin around the vertical axis during rolling, similar to the damping introduced during bounces, a torque-like damping force is applied, acting against the vertical spin:

$$M_{\omega_y} = -\mu_{\omega_y}\omega_y, \tag{2.26}$$

where μ_{ω_y} is an introduced constant representing rotational friction with the ground.

2.4 Implementation

The ball simulation was developed within the Unity environment, which offers a built-in coordinate system and utility functions for handling vector and quaternion operations. These tools were used throughout the implementation, as they simplify transformations and integrate well with Unity's overall architecture.

The development begins with the creation of a state machine to describe the ball's dynamic behavior in different conditions. This is followed by the implementation of force and torque computations that drive the simulation. Finally, a method is introduced to determine precisely when the ball bounces, ensuring realistic collision responses within the environment.

2.4.1 State-Based Modeling of Ball Motion

The first step in the implementation was to model the behavior of the ball using a finite state machine. Since the ball is subject to different forces depending on its interaction with the environment, four primary states were defined: *Bouncing, Sliding, Rolling*, and *Stopped*.

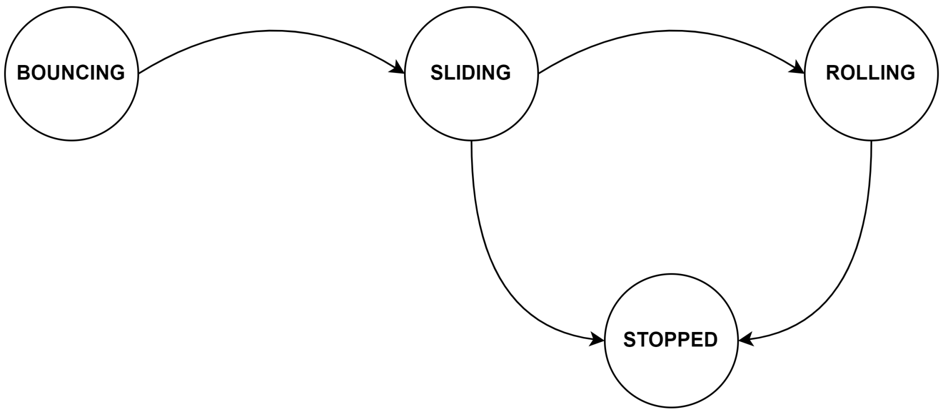

Figure 2.2: State machine diagram of the ball states.

Bouncing

In this state, the simulation focuses on handling collisions and modeling the effects of aerodynamic forces. The ball continues bouncing as long as collisions with the ground result in a sufficiently high vertical velocity. When this vertical velocity drops below a small threshold ϵ, the state transitions to *Sliding*.

Sliding

In the Sliding state, sliding friction is applied both as a linear force opposing motion and as a torque that decelerates the ball's rotation. The vertical forces cancel out: the gravitational force is balanced by the normal force, and the Magnus effect is assumed to be zero due to the lack of airflow around the ball, while all the other aerodynamic forces are still applied. When the no-slip condition is met, the state transitions to *Rolling*.

Rolling

This state differs from Sliding in that the friction model changes from sliding to rolling friction. The point of contact between the ball and the ground is assumed to be stationary relative to the surface, satisfying the no-slip condition.

Stopped

The *Stopped* state is used when the system's kinetic energy becomes negligible. In this state, physics simulation is temporarily paused to avoid unnecessary computation. The ball remains in this state until a new impulse is applied.

2.4.2 Heuristic for Detecting No-Slip Contact

To ensure a correct transition from Sliding to Rolling, a custom method was developed for detecting when the no-slip condition is satisfied. Due to numerical inaccuracies in integration, it's impractical to expect the contact point velocity to reach exactly $v_{cp} = 0$. Moreover, a simple threshold check like $v_{cp} < \epsilon$ can fail—after a single time step h, the velocity might reverse direction and exceed ϵ, preventing the transition to Rolling from ever occurring.

To address this, a heuristic approach was used. The contact point velocity at the current step, v_{cp_k}, is compared with that of the previous step, $v_{cp_{k-1}}$. The transition is triggered if either of the following conditions is true:

$$v_{cp_k} > v_{cp_{k-1}} \tag{2.27}$$

or

$$v_{cp_k} < \epsilon \tag{2.28}$$

This method works well in practice and effectively accounts for the issue. It should be noted that even if v_{cp_k} is not greater than $v_{cp_{k-1}}$, the true relative velocity might still have reached zero during the step. In such cases, the ball may remain in the sliding state for one extra frame before transitioning to rolling. However, this small delay has negligible impact on the simulation and can be safely ignored

2.4.3 Force and Torque Computation

The implementation of the ball's physics was designed using a modular approach, where each force is encapsulated within its own black-box function, independent from the others. This structure allows for easy addition, removal, or modification of individual forces without affecting the rest of the system. Additionally, Unity's `Vector3` library proved particularly useful, as it provides essential operations for 3D vector calculations—such as the cross product and angle between vectors—which are fundamental to the simulation.

As described in the formulation of the problem in (3.5), we need to define how to compute the linear acceleration a and the angular acceleration α. These depend on the current linear velocity v, angular velocity ω, and the state of the ball. The core function used to compute both a and α is defined as follows:

```
(Vector3, Vector3) CalculateAccelerationAndAngularAcceleration(Vector3 velocity, Vector3
    angularVelocity, BallState state)
{
    Vector3 acceleration = CalculateTotalForces(velocity, angularVelocity, state) / ball.mass;
    Vector3 angularAcceleration = CalculateTotalTorque(velocity, angularVelocity, acceleration, state)
        / ball.InertialMomentum;

    return (acceleration, angularAcceleration);
}
```

Inside this function, we can see two more functions being called, one computing the total force acting on the ball and the other computing the total torque. Note that the second function also takes the acceleration as a parameter: this is because, when the ball is in the state of pure rolling, the no-slip condition $v = \omega R$ must be preserved, and the angular acceleration depends on the linear acceleration.

Forces

The total force F acting on the ball is equal to

$$F = F_G + F_B + F_N + F_D + F_M + F_F, \tag{2.29}$$

where F_G is the gravitational force, F_B is the buoyant force, F_N is the normal force, F_D is the drag force, F_M is the Magnus force and F_F is the friction force with the ground.

```
Vector3 CalculateTotalForces(Vector3 velocity, Vector3 angularVelocity, BallState state)
{
    Vector3 dragForce = CalculateDragForce(velocity, angularVelocity);
    Vector3 magnusForce = CalculateMagnusForce(velocity, angularVelocity, state);
    Vector3 normalForce = CalculateNormalForce(state);
    Vector3 frictionForce = CalculateFrictionForce(velocity, angularVelocity, state);

    //sum all forces
    Vector3 totalForce = gravitationalForce + buoyantForce + dragForce + magnusForce + normalForce +
         frictionForce;

    return totalForce;
}
```

We can observe that the gravitational force F_G and the buoyant force F_B are constant throughout the simulation, and are instantiated as follows:

```
gravitationalForce = Gravity * ball.mass * Vector3.down;
buoyantForce = 4f/3f * Mathf.PI * Mathf.Pow(ball.radius,3) * airDensity * Gravity * Vector3.up;
```

All other forces vary dynamically during the simulation, and therefore require specific functions to compute them. In particular, for the aerodynamic forces (drag and Magnus), which are derived from empirical models, the implementation includes a flag that determines whether to use the empirical formulas with variable coefficients or a simplified model using constant custom values. This flag allows flexibility: when enabled, the simulation uses editor-assigned constants; when disabled, it applies the more physically accurate models.

```
Vector3 CalculateDragForce(Vector3 velocity, Vector3 angularVelocity)
{
    Vector3 dragForce;

    if(ball.useConstantCoefficients)
        //use constant linear drag coefficient
        dragForce = - 0.5f * airDensity * ball.constantDragCoefficient * Mathf.PI * Mathf.Pow(ball.
            radius, 2) * velocity.magnitude * velocity;
    else
    //use empirical model
    {
        //spin rate parameter
        float s = angularVelocity.magnitude * ball.radius / velocity.magnitude;

        //compute drag coefficient
        float dragCoefficient;

        if (velocity.magnitude > ball.speedOfTransitionFromLaminarToTurbulentDrag && s > 0.05f)
            //if the ball is spinning fast enough and moving faster than critical speed
            dragCoefficient = 0.4127f * Mathf.Pow(s, 0.3056f);
        else
            //otherwise use the drag coefficient model depending on Reynolds numbers
            dragCoefficient = 0.155f + 0.346f / (1 + Mathf.Exp((velocity.magnitude - ball.
                speedOfTransitionFromLaminarToTurbulentDrag) / ball.dragTransitionSlope));

        dragForce = -0.5f * airDensity * dragCoefficient * Mathf.PI * Mathf.Pow(ball.radius, 2) *
            velocity.magnitude * velocity;
    }

    return dragForce;
}
```

The formula used for the Magnus force when a constant lift coefficient is applied differs from the one in the empirical model. In this case, the model presented in [8] is used, which explicitly includes the angular velocity in the force calculation, as it does not influence the lift coefficient in this formulation.

```
Vector3 CalculateMagnusForce(Vector3 velocity, Vector3 angularVelocity, BallState state)
{
    Vector3 magnusForce;

    if (ball.useConstantCoefficients)
    {
        //use constant linear drag coefficient
        magnusForce = ball.constantDragCoefficient * Mathf.PI * airDensity * Mathf.Pow(ball.radius, 3)
             * Vector3.Cross(angularVelocity, velocity);
    }
    else
    //use empirical model
    {
        //direction cross product
        Vector3 direction = Vector3.Cross(angularVelocity.normalized, velocity.normalized);

        //spin rate parameter, how strong the spin is compared to the speed
        float s = angularVelocity.magnitude * ball.radius / velocity.magnitude;

        float cdRe0 = 0.155f + 0.346f / (1 + Mathf.Exp(-ball.
            speedOfTransitionFromLaminarToTurbulentDrag) / ball.dragTransitionSlope);
        float cdReRef = 0.155f;

        //compute drag coefficient
        float dragCoefficient;

        if (velocity.magnitude > ball.speedOfTransitionFromLaminarToTurbulentDrag && s > 0.05f)
            //if the ball is spinning fast enough and moving faster than critical speed
            dragCoefficient = 0.4127f * Mathf.Pow(s, 0.3056f);
        else
            //otherwise use the drag coefficient model depending on Reynolds numbers
            dragCoefficient = 0.155f + 0.346f / (1 + Mathf.Exp((velocity.magnitude - ball.
                speedOfTransitionFromLaminarToTurbulentDrag) / ball.dragTransitionSlope));

        //compute lift coefficient
        liftCoefficient = 1.15f * Mathf.Pow(s,0.83f) * (cdRe0 -  dragCoefficient) / (cdRe0 - cdReRef);

        magnusForce = 0.5f * airDensity * liftCoefficient * Mathf.PI * Mathf.Pow(ball.radius, 2) *
            Mathf.Pow(velocity.magnitude, 2) * direction;
    }

    //magnus force vertical component is considered negligible when the ball is on the ground
    if (state != BallState.Bouncing)
        magnusForce.y = 0f;

    return magnusForce;
}
```

The normal force depends entirely on the state of the ball: when the ball is bouncing the normal force is null, while when the ball is on the ground the normal force acts upwards perpendicularly to the ground and it is calculated as:

```
Vector3 CalculateNormalForce(BallState state)
{
    return state != BallState.Bouncing ? -(gravitationalForce + buoyantForce) : Vector3.zero;
}
```

Finally, the friction force is implemented by distinguishing between the rolling and sliding states. In the sliding state, the velocity at the point of contact is computed and used to determine the direction of the sliding friction. In the rolling state, the contact point is at rest relative to the ground, so the friction force opposes the ball's velocity direction.

```
Vector3 CalculateFrictionForce(Vector3 velocity, Vector3 angularVelocity, BallState state)
{
    if(state == BallState.Bouncing)
        return Vector3.zero;

    Vector3 frictionForce;

    float normalForce = (gravitationalForce + buoyantForce).magnitude;

    //if the ball is sliding, use the sliding friction formula
    if(state == BallState.Sliding)
    {
        //inclination in of the ball axis against the up-axis in radians
        float theta = (90f - Vector3.Angle(angularVelocity.normalized, Vector3.up)) * Mathf.Deg2Rad;

        //compute direction of the velocity of the contact point
        Vector3 contactPointVelocity = (velocity + Vector3.Cross(ball.radius * Mathf.Cos(theta) *
            Vector3.up, angularVelocity)).normalized;

        frictionForce = -ball.coefficientOfSlidingFriction * normalForce * contactPointVelocity;
    }
    else
        //otherwise, rolling friction formula
        frictionForce = -ball.coefficientOfRollingFriction * normalForce * velocity.normalized;

    return frictionForce;
}
```

Torques

The same approach is applied to the computation of torque, where the total torque T acting on the ball is given by:

$$T = T_F + T_D, \tag{2.30}$$

where T_F represents the torque generated by friction with the ground, which includes the artificial damping term applied to the spin around the vertical axis, and T_D accounts for spin degradation caused by air resistance acting on the ball's surface.

```
Vector3 CalculateTotalTorque(Vector3 velocity, Vector3 angularVelocity, Vector3 acceleration,
     BallState state)
{
    Vector3 frictionTorque = CalculateFrictionTorque(velocity, angularVelocity, acceleration, state);
    Vector3 spinDegradationTorque = CalculateSpinDegradation(velocity, angularVelocity, state);
    Vector3 totalTorque = frictionTorque + spinDegradationTorque;

    return totalTorque;
}
```

The friction torque is computed in two steps. First, a damping torque is applied to reduce the angular velocity around the vertical axis. Then, depending on whether the ball is rolling or sliding, an additional torque is calculated using one of the two physical models introduced in Section 2.

```
Vector3 CalculateFrictionTorque(Vector3 velocity, Vector3 angularVelocity, Vector3 acceleration,
     BallState state)
{
    if(state == BallState.Bouncing)
        return Vector3.zero;

    float normalForce = (gravitationalForce + buoyantForce).magnitude;

    Vector3 frictionTorque = new Vector3(0, - ball.coefficientOfVerticalAxisSpinningDamping *
         angularVelocity.y, 0);

    //if the ball is rolling, the contact point is not moving with respect to the ground, therefore
         the is no other friction component acting on the ball
    if (state == BallState.Rolling)
    {
        frictionTorque += Vector3.Cross(acceleration / ball.radius, Vector3.down) * ball.
             InertialMomentum;
    }
    else
    {
        //inclination in of the ball axis against the up-axis in radians
        float theta = (90f - Vector3.Angle(angularVelocity.normalized, Vector3.up)) * Mathf.Deg2Rad;

        //compute direction of the velocity of the contact point
        Vector3 frictionDirection = velocity + Vector3.Cross(ball.radius * Mathf.Cos(theta) * Vector3.
             up, angularVelocity).normalized;

        //compute the axis on which the torque acts
        Vector3 axisOfTorque = Vector3.Cross(Vector3.up, frictionDirection).normalized;

        Vector3 angularAcceleration = ball.coefficientOfSlidingFriction * normalForce * ball.radius *
             axisOfTorque / ball.InertialMomentum;

        frictionTorque += angularAcceleration * ball.InertialMomentum;
    }

    return frictionTorque;
}
```

Lastly, the spin degradation, which is also based on an empirical model, implements both the empirical approach and a version using a constant coefficient.

```
Vector3 CalculateSpinDegradation(Vector3 velocity, Vector3 angularVelocity, BallState state)
{
    if(state != BallState.Bouncing || angularVelocity.magnitude < 1e-6)
        return Vector3.zero;

    float totalForce;

    if (ball.useConstantCoefficients)
    {
        //use constant coefficient
        totalForce =  airDensity * 2 * Mathf.PI * Mathf.Pow(ball.radius, 2) * Mathf.Pow(velocity.
             magnitude, 2) * ball.constantSpinDecayCoefficient;
    }
    else
    {
        //use empirical formula

        //calculate the relative velocity of both side of the ball with respect to the direction of
            the linear velocity of the ball
        float utP = Mathf.Abs(Vector3.Cross(velocity, angularVelocity.normalized).magnitude + ball.
            radius / 2 * angularVelocity.magnitude);
        float utM = Mathf.Abs(Vector3.Cross(velocity, angularVelocity.normalized).magnitude - ball.
            radius / 2 * angularVelocity.magnitude);

        //coefficient of each side
        float cfP = 0.0074f * Mathf.Pow(utP * airDensity * ball.radius * 2 / kinematicViscosityOfAir,
            -1f / 5f);
        float cfM = 0.0074f * Mathf.Pow(utM * airDensity * ball.radius * 2 / kinematicViscosityOfAir,
            -1f / 5f);

        totalForce = airDensity * Mathf.PI * Mathf.Pow(ball.radius, 2) * (Mathf.Pow(utP, 2) * cfP +
            Mathf.Pow(utM, 2) * cfM);
    }

    Vector3 spinDegradationTorque = -totalForce * ball.radius / 2 * angularVelocity.normalized;

    return spinDegradationTorque;
}
```

2.4.4 Collisions

Handling collisions accurately proved to be one of the more challenging aspects of the simulation. When the ball is bouncing, the system checks each frame to determine whether it has collided with the ground. However, due to the discrete nature of the simulation steps, a collision is usually detected after the ball has already penetrated the ground slightly. To correct for this and ensure a realistic bounce, a recursive algorithm is used to approximate the actual collision time as closely as possible.

Collision detection algorithm

When a collision is detected at the end of a time step, it indicates that the actual collision occurred sometime during that step. The algorithm's goal is to approximate the exact moment of collision, t_c, by performing a binary search within the time interval.

The process begins by checking whether the collision still occurs at half the original step size. Based on whether the collision happens or not, we narrow the interval and continue recursively, halving the step size each time until we reach a desired precision.

Example The recursive search continues until the step size δ reaches the stopping condition $\delta \leq \varepsilon$, where in this case $\varepsilon = \frac{\Delta t}{8}$. This ensures a sufficiently accurate estimate of the collision time t_c.

1. A collision is detected at time $t_1 = \Delta t$, meaning it actually occurred at some t_c such that $0 < t_c < \Delta t$. Step size: $\delta = \frac{\Delta t}{2}$.

2. We move back to $t_2 = \frac{\Delta t}{2}$ and check again. The collision still occurs, so $0 < t_c < t_2$. Step size: $\delta = \frac{\Delta t}{4}$.

3. We backtrack further to $t_3 = \frac{\Delta t}{4}$. The collision no longer occurs, so the interval narrows to $t_3 < t_c < t_2$. Step size: $\delta = \frac{\Delta t}{8}$.

4. We then proceed to $t_4 = \frac{3\Delta t}{8}$, where the step size now satisfies $\delta \leq \varepsilon$, and the process is therefore terminated at this point.

In this way, we know that the final approximation t_4 of the collision time differs from the real solution t_c by at most $\delta = \frac{\Delta t}{8}$.

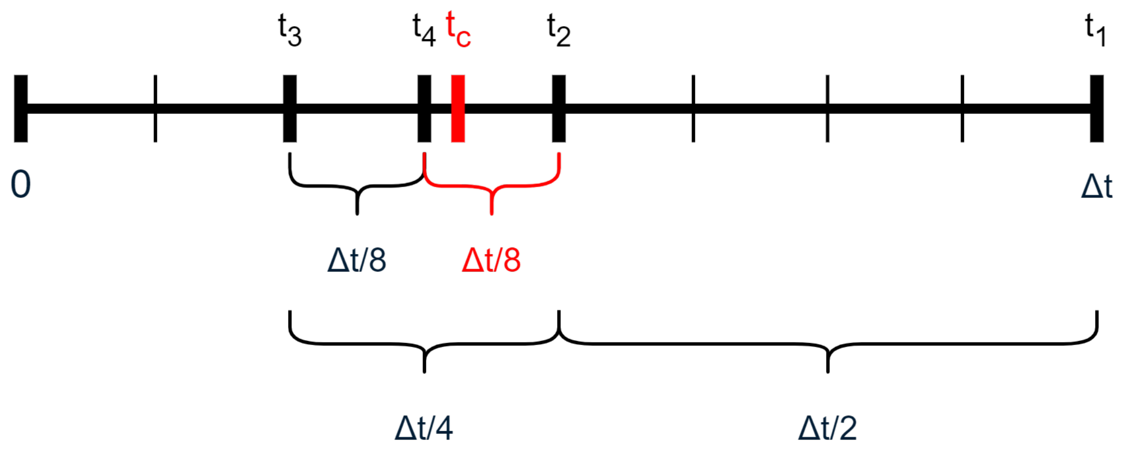

Figure 2.3: Representation of the recursive process.

```
float RecursiveCollisionDetection(float deltaTime, float step, Vector3 position, SysQuat orientation,
     Vector3 velocity, Vector3 angularVelocity, BallState state)
{
    //predict new position
    (Vector3 newPosition, _, _, _) = IntegrateMotion(deltaTime, position, orientation, velocity,
        angularVelocity, state);

    //adjust delta time: go back if below ground, forward if above
    float newDeltaTime = newPosition.y - ball.radius < 0f ? deltaTime - step / 2 : deltaTime + step /
        2;

    //stop if step is below precision threshold
    if (step < TimeStepThreshold)
        return newDeltaTime;

    //recurse with new delta time and halved step size
    return RecursiveCollisionDetection(newDeltaTime, step/2, position, orientation, velocity,
        angularVelocity, state);
}
```

Collision response

After the approximated collision time t_a is found, the ball's motion is re-integrated from the beginning of the frame using a time-step equal to t_a.

At this point, the remaining time in the frame is given by $t_r = h - t_a$. When the collision occurs, new values for the velocity and angular velocity are computed using equations (2.16) and (2.17).

Once the new velocities are obtained, the ball's motion is integrated again over the remaining time-step t_r to complete the frame.

Chapter 3

Numerical Integration

Accurately simulating the motion of a football requires solving the equations of motion over time. Due to the presence of nonlinear forces—such as aerodynamic drag, lift, collisions, and friction—these equations cannot be solved analytically. As a result, numerical integration methods are employed to approximate the evolution of the system state over discrete time steps.

This chapter presents and evaluates a set of numerical integration techniques used to simulate the dynamics of a football. The chosen methods were selected for their simplicity, efficiency, and suitability for real-time simulation. These include the Explicit Euler method, the Semi-Implicit Euler method, and the Runge-Kutta family of methods. Each technique is described in the context of its mathematical formulation and implemented. Their trade-offs in terms of accuracy, numerical stability, and performance are then discussed and compared through practical tests.

3.1 Numerical Formulation of Ball Dynamics

To resolve a physical process of the behaviour of ball we can use some methods from numerical analysis, that allow to give an approximation of the solution. There are typically two approaches: explicit and implicit methods. They differ for how the state of a system is calculated; explicit methods calculate the later state of the system from the state at the current time, while implicit methods find a solution by solving an equation in both the current state and the later one.

Let $y(t)$ be the current state and $y(t + \Delta t)$ the state at a later time. For an explicit method

$$y(t + \Delta t) = f(y(t)), \tag{3.1}$$

while for an implicit method, one solves an equation of the form

$$g\Big(y(t), y(t + \Delta t)\Big) = 0 \tag{3.2}$$

Implicit methods are more computationally expensive and can be more challenging to implement. For this reason, explicit methods are typically preferred in real-time simulations.

Integrating over time a differential equation is equivalent to solve an initial value problem of the form

$$\dot{y} = f(t, y), \qquad y(t_0) = y_0 \tag{3.3}$$

where in this case

$$y(t) = \begin{bmatrix} \vec{p} \\ \vec{v} \\ \vec{q} \\ \vec{\omega} \end{bmatrix} \tag{3.4}$$

where $\vec{p}$ is the position, $\vec{v}$ is the velocity, $\vec{q}$ is the quaternion representing the orientation of the ball and $\vec{\omega}$ is the angular velocity and

$$\dot{y} = f(t, y) = \begin{bmatrix} \dot{\vec{p}} \\ \dot{\vec{v}} \\ \dot{\vec{q}} \\ \dot{\vec{\omega}} \end{bmatrix} = \begin{bmatrix} \vec{v} \\ \vec{a}(v, \omega) \\ \frac{1}{2}\vec{q}_\omega \otimes \vec{q} \\ \vec{\alpha}(v, \omega) \end{bmatrix} \tag{3.5}$$

where a and α are respectively the acceleration of the center of mass and the angular acceleration, and $\vec{q}_\omega$ is the quaternion of the angular velocity given by

$$\vec{q}_\omega = \begin{bmatrix} 0 \\ \vec{\omega} \end{bmatrix}. \tag{3.6}$$

The quaternion derivative $\dot{\vec{q}}$ is computed under the assumption that the angular velocity $\vec{\omega}$ is expressed in the global coordinate system, following the formulation presented by Solà [16].

3.2 Integration methods

3.2.1 Euler method

The Euler method is the most basic explicit method for numerical integration of ODEs (ordinary differential equations) and it is defined by

$$y_{n+1} = y_n + hf(t_n, y_n), \tag{3.7}$$

where h is the time-step. The Euler method is a first-order method, so the local error—the error gained at each step—is $\mathcal{O}(h^2)$, and the global error—the cumulative error after many steps—is $\mathcal{O}(h)$ [12]. Therefore, the smallest h is, the smallest the error will also be when integrating.

However, the Euler method is not energy-conserving, which can lead to issues in physical simulations. For example, when simulating a bouncing ball, the method can artificially increase the system's energy over time. As a result, even if the vertical velocity is dampened after each bounce, the energy may stop decreasing below a certain point, and the ball may continue bouncing indefinitely.

3.2.2 Semi-implicit Euler Method

To address the limitations of the standard explicit Euler method, a simple but effective modification is the semi-implicit Euler method. Unlike the explicit Euler approach, where the position is updated using the current velocity, the semi-implicit method first updates the velocity using the current acceleration, then uses this updated velocity to compute the new position [13]. This method is a symplectic integrator, a class of methods specifically designed to conserve energy over long time periods. Despite maintaining the same order of accuracy as the explicit Euler method, it provides improved stability in dynamical simulations.

For linear motion involving position $\vec{p}$ and velocity $\vec{v}$, the semi-implicit Euler update is defined as:

$$\begin{aligned} \vec{v}_{n+1} &= \vec{v}_n + h\,\vec{a}(v_n, \omega_n) \\ \vec{p}_{n+1} &= \vec{p}_n + h\,\vec{v}_{n+1} \end{aligned} \tag{3.8}$$

Similarly, for rotational motion the method updates as:

$$\begin{aligned} \vec{\omega}_{n+1} &= \vec{\omega}_n + h\,\vec{\alpha}(v_n, \omega_n) \\ \vec{q}_{n+1} &= \vec{q}_n + h\,\frac{1}{2}\,\vec{q}_{\omega_{n+1}} \otimes \vec{q}_n \end{aligned} \tag{3.9}$$

By updating velocity before position, the semi-implicit Euler method tends to conserve energy more effectively over time and results in more stable simulations.

3.2.3 Runge-Kutta methods

Runge-Kutta methods are a family of iterative techniques to solve ODEs.

Let s be an integer—the number of stages—and a_{21}, a_{31}, a_{32}, ..., a_{s1}, a_{s2}, ..., $a_{s,s-1}$, b_1, ..., b_s, c_2, ..., c_s be real coefficients. Then the method

$$
\begin{aligned}
k_1 &= f(t_0,\ y_0) \\
k_2 &= f(t_0 + c_2 h,\ y_0 + h a_{21} k_1) \\
k_3 &= f(t_0 + c_3 h,\ y_0 + h(a_{31} k_1 + a_{32} k_2)) \\
&\ \vdots \\
k_s &= f(t_0 + c_s h,\ y_0 + h(a_{s1} k_1 + \cdots + a_{s,s-1} k_{s-1})) \\
y_{n+1} &= y_n + h(b_1 k_1 + \cdots + b_s k_s)
\end{aligned}
\tag{3.10}
$$

is called an s-stage explicit Runge-Kutta method. The more stages the method has, the more accurate it will be, but also more computationally expensive. These methods, however, are generally not energy-conserving [14].

Runge-Kutta methods are usually represented compactly using a Butcher tableau. The general Butcher tableau is:

$$
\begin{array}{c|ccccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & \vdots & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
\tag{3.11}
$$

The simplest Runge-Kutta method, with 1 stage, is exactly the explicit Euler method, so the Runge-Kutta family can be seen as a generalization of it:

$$
\begin{array}{c|c}
0 & 0 \\
\hline
 & 1
\end{array}
\tag{3.12}
$$

which results in

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
y_{n+1} &= y_n + h k_1
\end{aligned}
\tag{3.13}
$$

Second-Order Runge-Kutta

The 2-stage Runge-Kutta explicit method—also known as RK2 or the Midpoint method—is the following:

$$
\begin{array}{c|cc}
0 & & \\
1/2 & 1/2 & \\
\hline
 & 0 & 1
\end{array}
\tag{3.14}
$$

From this, the RK2 method is:

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2} k_1\right) \\
y_{n+1} &= y_n + h \cdot k_2
\end{aligned}
\tag{3.15}
$$

It is a second-order method, so it provides better accuracy than Euler while remaining relatively inexpensive. It achieves a global error of $\mathcal{O}(h^2)$ and local error of $\mathcal{O}(h^3)$ [14].

Fourth-Order Runge-Kutta

The classical Runge-Kutta method, or RK4, is a 4-stage method. Its Butcher tableau is:

$$\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 2/6 & 2/6 & 1/6
\end{array} \tag{3.16}$$

Explicitly, the RK4 method is:

$$\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}\right) \\
k_3 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_2\right) \\
k_4 &= f(t_n + h, y_n + hk_3) \\
y_{n+1} &= y_n + h\left(\frac{1}{6}k_1 + \frac{2}{6}k_2 + \frac{2}{6}k_3 + \frac{1}{6}k_4\right)
\end{aligned} \tag{3.17}$$

It is a fourth-order accurate method, meaning it has local error $\mathcal{O}(h^5)$ and global error $\mathcal{O}(h^4)$ [14].

This makes it highly accurate and widely used in offline or high-precision simulations, although it is often too computationally expensive for real-time applications.

Runge-Kutta methods are not energy-conserving, which can result in energy drift over long simulations. However, both RK4 and RK2 offer significantly higher accuracy than the Euler method, which helps reduce the accumulation of numerical errors and mitigates the energy instability typically associated with simpler integrators.

3.2.4 Velocity-Verlet Method

During the exploration of suitable numerical integrators for the simulation, the Velocity-Verlet scheme [15] was considered. This method is second-order accurate, symplectic, and widely used in physics-based simulations due to its favorable energy conservation properties. However, it was ultimately discarded in this work due to fundamental limitations related to the nature of the problem being solved.

Velocity-Verlet is designed for systems where the acceleration depends solely on position, i.e.,

$$\vec{a} = \vec{a}(\vec{p}) \tag{3.18}$$

In the case of a spinning ball, however, the acceleration depends on both the linear and angular velocities:

$$\vec{a} = \vec{a}(\vec{v}, \vec{\omega}) \tag{3.19}$$

This introduces a dependency that breaks the assumptions underpinning the method.

Method Breakdown

The standard Velocity-Verlet formulation is:

$$\begin{aligned}
\vec{p}_{n+1} &= \vec{p}_n + \vec{v}_n \Delta t + \frac{1}{2}\vec{a}_n \Delta t^2 \\
\vec{v}_{n+1} &= \vec{v}_n + \frac{1}{2}(\vec{a}_n + \vec{a}_{n+1})\Delta t
\end{aligned} \tag{3.20}$$

This method assumes that the acceleration $\vec{a}_{n+1}$ can be computed directly from the new position $\vec{p}_{n+1}$. However, in this case:

$$\vec{a} = \vec{a}(\vec{v}, \vec{\omega}) \Rightarrow \vec{a}_{n+1} = \vec{a}(\vec{v}_{n+1}, \vec{\omega}_{n+1}) \tag{3.21}$$

This creates a circular dependency:

$$\vec{v}_{n+1} = \vec{v}_n + \frac{1}{2}\left(\vec{a}_n + \vec{a}(\vec{v}_{n+1}, \vec{\omega}_{n+1})\right)\Delta t \tag{3.22}$$

Since $\vec{a}_{n+1}$ depends on $\vec{v}_{n+1}$, the update becomes implicit and cannot be solved in closed form without iteration, which would increase the computational complexity significantly and violate the method's original appeal for real-time use.

Conclusion

The Velocity-Verlet method relies on the assumption that acceleration depends only on position. When acceleration depends on velocity or angular velocity, as in the case of a spinning football with drag, Magnus force, and torque, the method loses its explicit nature and requires iterative solvers. This defeats its primary advantage for use in real-time applications and thus made it unsuitable for this project.

3.3 Implemetation

3.3.1 Physics Update Loop and Time Stepping in Unity

In a video game simulation, accuracy and consistency are crucial. One key aspect of achieving this is ensuring that the physics simulation behaves reliably regardless of frame rate fluctuations. Unity provides two main update methods that serve different purposes: Update() and FixedUpdate().

The Update() method is called once per frame and is tied directly to the frame rate. This makes it suitable for handling gameplay logic that should react immediately to player actions. However, since the frame rate can vary depending on hardware and performance, using Update() for physics calculations can lead to inconsistent and unstable behavior.

To address this, Unity offers the FixedUpdate() method, which is called at consistent, fixed intervals independent of the frame rate. This makes it the preferred choice for applying physics-related logic. Using FixedUpdate() ensures smoother and more stable simulations across different hardware setups and performance conditions [21].

While this method simulates code execution at fixed time intervals, in practice, the intervals between FixedUpdate() calls are not always uniform. This is because a FixedUpdate() needs a frame to run in, and the duration of each frame may not perfectly match the fixed time step.

If the frame rate is higher than the fixed update rate, each frame will typically include one or no FixedUpdate() calls, as shown in Figure 3.1.

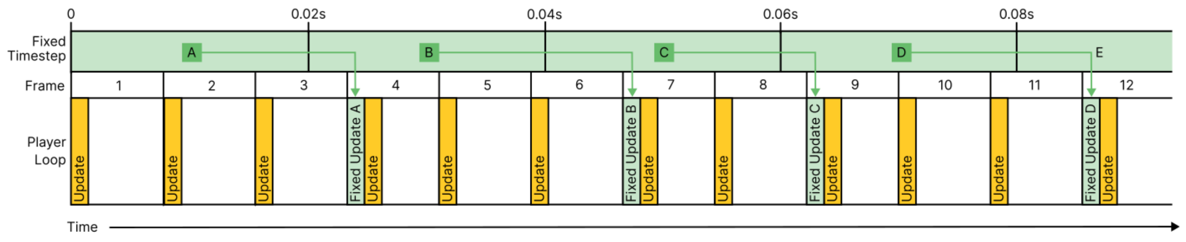

Figure 3.1: An example showing FixedUpdate running at 50 updates per second (0.02s per fixed update) and the Player Loop running at approximately 80 frames per second. Some frame updates (marked in yellow) have a corresponding FixedUpdate (marked in green) if a new complete fixed timestep has elapsed by the start of the frame. From reference [21].

Conversely, if the frame rate drops below the fixed update rate, a queue of fixed updates can accumulate. In this case, each frame may execute one or more FixedUpdate() calls to allow the physics simulation to catch up, as illustrated in Figure 3.2.

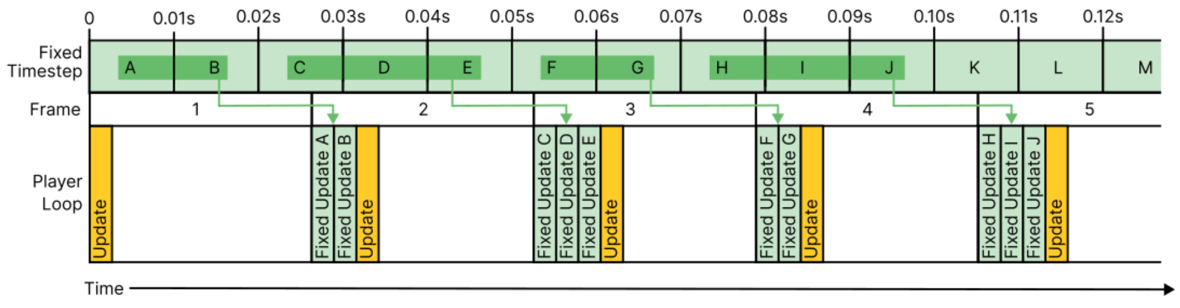

Figure 3.2: Example of Update running at 25 FPS and FixedUpdate running at 100 updates per second. In this case, four FixedUpdate calls occur during a single frame. From reference [21].

A lower time step value results in more frequent physics updates and more precise simulations, but also increases CPU load. The fixed time step can be accessed and modified via `Time.fixedDeltaTime`, provided by Unity's `Time` API.

3.3.2 Quaternion Integration and Orientation Handling

To update the orientation of the ball, operations on quaternions are required. In this context, `Unity.Quaternion` is not sufficient, as it does not support essential quaternion operations such as addition [19]. Therefore, the `System.Numerics.Quaternion` library was used instead, as it provides support for these operations. Additionally, utility functions were implemented to convert between `Unity.Quaternion` and `System.Numerics.Quaternion`, as well as a function to compute the derivative of a quaternion, as described in Equation (3.6):

```
SysQuat QuaternionDerivative(SysQuat quaternion, Vector3 angularVelocity)
{
    SysQuat angularVelocityQuaternion = new SysQuat(angularVelocity.x, angularVelocity.y,
        angularVelocity.z, 0f);

    SysQuat product = SysQuat.Multiply(angularVelocityQuaternion, quaternion);

    SysQuat quaternionDerivative = SysQuat.Multiply(product, 0.5f);

    return quaternionDerivative;
}
```

where `SysQuat` is an alias referencing `System.Numerics.Quaternion`

It is important to note that in the constructor of `System.Numerics.Quaternion`, the magnitude component is placed last, whereas in the mathematical formulation it appears first in the quaternion vector.

Moreover, each time the orientation is updated, the quaternion must be normalized to prevent drift or numerical errors during long simulations. However, this normalization must not be applied to the derivative, as it would alter the magnitude of the rotation.

3.3.3 Implementation of Euler Methods

The implementation of these two methods is very similar and was straightforward to create. The only difference lies in how the position `pNext` and the orientation `qNext` are updated. In the Explicit Euler method, they are updated using the velocity `vCurrent` and angular velocity `wCurrent` at the beginning of the frame. In contrast, in the Semi-implicit Euler method, they are updated using the newly calculated values `vNext` and `wNext`.

Explicit Euler

```
(Vector3, SysQuat, Vector3, Vector3) ExplicitEuler(float deltaTime,Vector3 pCurrent, SysQuat qCurrent,
      Vector3 vCurrent, Vector3 wCurrent, BallState state)
{
    //acceleration and angular acceleration at current state
    (Vector3 aCurrent, Vector3 alphaCurrent) = CalculateAccelerationAndAngularAcceleration(vCurrent,
        wCurrent, state);
    //velocity update
    Vector3 vNext = vCurrent + deltaTime * aCurrent;

    //angular velocity update
    Vector3 wNext = wCurrent + deltaTime * alphaCurrent;

    //position update
    Vector3 pNext = pCurrent + deltaTime * vCurrent;

    //orientation update
    SysQuat qDerivative = QuaternionUtils.QuaternionDerivative(qCurrent, wCurrent);
    SysQuat qDelta = SysQuat.Multiply(qDerivative, deltaTime);
    SysQuat qNext = SysQuat.Add(qCurrent, qDelta);
    qNext = SysQuat.Normalize(qNext);

    return (pNext, qNext, vNext, wNext);
}
```

Semi-Implicit Euler

```
(Vector3, SysQuat, Vector3, Vector3) SemiImplicitEuler(float deltaTime, Vector3 pCurrent, SysQuat
     qCurrent, Vector3 vCurrent, Vector3 wCurrent, BallState state)
{
    //acceleration and angular acceleration at current state
    (Vector3 aCurrent, Vector3 alphaCurrent) = CalculateAccelerationAndAngularAcceleration(vCurrent,
        wCurrent, state);

    //velocity update
    Vector3 vNext = vCurrent + deltaTime * aCurrent;

    //angular velocity update
    Vector3 wNext = wCurrent + deltaTime * alphaCurrent;

    //position update
    Vector3 pNext = pCurrent + deltaTime * vNext;

    //orientation update
    SysQuat qDerivative = QuaternionUtils.QuaternionDerivative(qCurrent, wNext);
    SysQuat qDelta = SysQuat.Multiply(qDerivative, deltaTime);
    SysQuat qNext = SysQuat.Add(qCurrent, qDelta);
    qNext = SysQuat.Normalize(qNext);

    return (pNext, qNext, vNext, wNext);
}
```

3.3.4 Implementation of Runge-Kutta Methods

Design of the Derivative Class

In Runge-Kutta methods, a key aspect is the computation of the intermediate derivatives k_n. To organize these derivatives, a class called `K` was created, containing four attributes representing the derivatives of position, velocity, orientation, and angular velocity:

```
class K
{
    public Vector3 V, W, P;
    public SysQuat Q;
}
```

Each attribute of this class corresponds to the derivative used when updating a specific component of the system. For example, to update the position, the corresponding value is accessed as follows:

```
pNext = pCurrent + deltaTime * k.P;
```

A helper method called `ComputeK` was defined to compute these intermediate derivatives. It returns an instance of type `K`:

```
K ComputeK(Vector3 velocity, Vector3 angularVelocity, SysQuat orientation, BallState state)
{
    K k = new K();

    (k.V, k.W) = CalculateAccelerationAndAngularAcceleration(velocity, angularVelocity, state);

    k.P = velocity;

    k.Q = QuaternionUtils.QuaternionDerivative(orientation, angularVelocity);
    return k;
}
```

With the defined function, the RK2 and RK4 methods could then be implemented directly by following the definitions from Section 3.2.3.

Second-Order Runge-Kutta (RK2)

```
(Vector3, SysQuat, Vector3, Vector3) Runge_Kutta_2(float deltaTime, Vector3 pCurrent, SysQuat qCurrent
    , Vector3 vCurrent, Vector3 wCurrent, BallState state)
{
    //k1 = f(t_n, y_n)
    K k1 = ComputeK(vCurrent, wCurrent, qCurrent, state);

    //k2 = f(t_n + h/2, y_n + k1 * h/2)
    K k2 = ComputeK(vCurrent + deltaTime * k1.V / 2f, wCurrent + deltaTime * k1.W / 2f, SysQuat.Add(
        qCurrent, SysQuat.Multiply(k1.Q, deltaTime / 2f)), state);

    //position update
    Vector3 pNext = pCurrent + deltaTime * k2.P;

    //velocity update
    Vector3 vNext = vCurrent + deltaTime * k2.V;

    //angular velocity update
    Vector3 wNext = wCurrent + deltaTime * k2.W;

    //orientation update
    SysQuat qDelta = SysQuat.Multiply(k2.Q, deltaTime);
    SysQuat qNext = SysQuat.Add(qCurrent, qDelta);
    qNext = SysQuat.Normalize(qNext);

    return (pNext, qNext, vNext, wNext);
}
```

Fourth-Order Runge-Kutta (RK4)

```
(Vector3, SysQuat, Vector3, Vector3) Runge_Kutta_4(float deltaTime,Vector3 pCurrent, SysQuat qCurrent,
     Vector3 vCurrent, Vector3 wCurrent, BallState state)
{
    //k1 = f(t_n, y_n)
    K k1 = ComputeK(vCurrent, wCurrent, qCurrent, state);

    //k2 = f(t_n + h/2, y_n + k1 * h/2)
    K k2 = ComputeK(vCurrent + deltaTime * k1.V / 2f, wCurrent + deltaTime * k1.W / 2f, SysQuat.Add(
        qCurrent, SysQuat.Multiply(k1.Q, deltaTime / 2f)), state);

    //k3 = f(t_n + h/2, y_n + k2 * h/2)
    K k3 = ComputeK(vCurrent + deltaTime * k2.V / 2f, wCurrent + deltaTime * k2.W / 2f, SysQuat.Add(
        qCurrent, SysQuat.Multiply(k2.Q, deltaTime / 2f)), state);

    //k4 = f(t_n + h, y_n + k3 * h)
    K k4 = ComputeK(vCurrent + deltaTime * k3.V, wCurrent + deltaTime * k3.W, SysQuat.Add(qCurrent,
        SysQuat.Multiply(k3.Q, deltaTime)), state);

    //velocity update
    Vector3 vNext = vCurrent + deltaTime * (k1.V + 2*k2.V + 2*k3.V + k4.V) / 6;

    //position update
    Vector3 pNext = pCurrent + deltaTime * (k1.P + 2*k2.P + 2*k3.P + k4.P) / 6;

    //angular velocity update
    Vector3 wNext = wCurrent + deltaTime * (k1.W + 2*k2.W + 2*k3.W + k4.W) / 6;

    //orientation update
    SysQuat weightedSum = SysQuat.Add(k1.Q, SysQuat.Multiply(k2.Q, 2f));
    weightedSum = SysQuat.Add(weightedSum, SysQuat.Multiply(k3.Q, 2f));
    weightedSum = SysQuat.Add(weightedSum, k4.Q);
    SysQuat deltaQ = SysQuat.Multiply(weightedSum, deltaTime / 6f);
    SysQuat qNext = SysQuat.Add(qCurrent, deltaQ);
    qNext = SysQuat.Normalize(qNext);

    return (pNext, qNext, vNext, wNext);
}
```

3.4 Results

To test and compare the implemented methods, three test cases were conducted. The first test measures the accuracy of each integration method by comparing the results to the analytical solution of projectile motion in a vacuum. The second test examines the consistency of each method by comparing results obtained at different time steps under full aerodynamic forces. The third test assesses the computational performance of each method by simulating realistic conditions and recording execution time. Finally, a summary of the results highlights the strengths and weaknesses of each approach.

3.4.1 Accuracy Test

The first test considers the classical physical scenario of a projectile moving in a vacuum, where the only force acting on the body is gravity, and the horizontal velocity remains constant. This experiment has an exact analytical solution given by:

$$\begin{cases} y = y_0 + v_{y0}t - \frac{1}{2}gt^2 \\ x = x_0 + v_{x0}t \end{cases} \tag{3.23}$$

Here, y and x represent the vertical and horizontal positions respectively, y_0 and x_0 are the initial coordinates, and v_{y0} and v_{x0} denote the initial velocities in the vertical and horizontal directions.

In this experiment, a ball is kicked from the ground with a given initial velocity. The comparison between the numerical simulation and the analytical solution will be performed at the point where the ball collides back with the ground.

The initial position of the ball is defined as:

$$\begin{cases} x_0 = 0 \\ y_0 = 0 \end{cases} \tag{3.24}$$

The simulation is then run multiple times with random initial velocity components:

$$\begin{cases} v_{x0} \in [5, 20] \\ v_{y0} \in [5, 20] \end{cases} \tag{3.25}$$

To find the time t_f when the ball returns to the ground ($y = 0$), we solve the vertical motion equation:

$$0 = y_0 + v_{y0}t_f - \frac{1}{2}gt_f^2 = 10t_f - \frac{1}{2} \times 9.81 \times t_f^2 \tag{3.26}$$

which yields:

$$t_f = \frac{2v_{y0}}{g} \tag{3.27}$$

Then, the horizontal distance traveled at impact is:

$$x_f = x_0 + v_{x0}t_f \tag{3.28}$$

To evaluate the accuracy of the different numerical integration methods, 50 shots were simulated at three different time steps: 10 ms, 20 ms, and 40 ms. For each shot, the final landing position of the ball was compared to the corresponding analytical solution, and the average displacement error was computed. This process was repeated for each method. The collision detection system was configured with an accuracy of 1 μs to ensure precise measurement.

Comparison of Results

The results are summarized in the table below:

Method	$\Delta t = 40\,\mathbf{ms}$	$\Delta t = 20\,\mathbf{ms}$	$\Delta t = 10\,\mathbf{ms}$
Explicit Euler	588.870 mm	294.817 mm	147.541 mm
Semi-Implicit Euler	586.855 mm	293.985 mm	147.142 mm
Runge-Kutta 2 (RK2)	0.016 mm	0.027 mm	0.055 mm
Runge-Kutta 4 (RK4)	0.014 mm	0.025 mm	0.044 mm

Table 3.1: Average displacement of each method at specific time steps from the analytical collision point.

From the results, it is evident that the Explicit Euler and Semi-Implicit Euler methods are significantly less accurate, with errors exceeding 50 cm at a time step of 40 ms, and still performing poorly at 10 ms with errors around 15 cm. In contrast, the RK2 and RK4 methods demonstrate excellent accuracy across all three time steps, with errors consistently below a tenth of a millimeter highlighting their strong suitability for precise simulations.

Interestingly, the Runge-Kutta methods exhibit a somewhat counterintuitive behavior. While it is generally expected that reducing the time step leads to higher accuracy, in this case the results show slightly lower errors at larger time steps. This phenomenon may be attributed to the accumulation of floating-point rounding errors, which become more significant as the number of iterations increases with smaller time steps.

It is worth noting that the contribution of the collision detection error, although minimal under the current settings, must still be taken into account.

In conclusion, RK2 and RK4 methods demonstrate high accuracy for this test case, while Euler methods fail to produce sufficiently precise results. This distinction becomes even more relevant when considering a complete simulation with additional forces acting on the ball, which may introduce further sources of error.

3.4.2 Consistency test

The second test is designed to evaluate which integration method—between RK2 and RK4—maintains the most consistent and reliable results across varying time steps under full-force conditions. To perform this evaluation, 250 shots were simulated using randomized initial velocity and spin values. Each shot was run at three different time steps: 10 ms, 20 ms, and 40 ms, resulting in a total of 750 simulations per method.

For each shot, the landing position was recorded at each time step. The positional discrepancies across the different time steps were then computed for each method, allowing a comparison of their numerical stability and consistency.

Comparison of Results

The RK2 method yielded average deviations of 3.1 mm (10–20 ms), 7.1 mm (20–40 ms), and 7.0 mm (10–40 ms). The variance remained low, as confirmed by the histograms of the position differences.

From the distribution plots, we observe that the largest discrepancies occurred when comparing the 40 ms simulations to the others, reaching up to 5 cm in rare cases. However, comparisons between the 10 ms and 20 ms runs showed maximum deviations below 2 cm. Most results remained tightly clustered around the mean, indicating strong consistency overall.

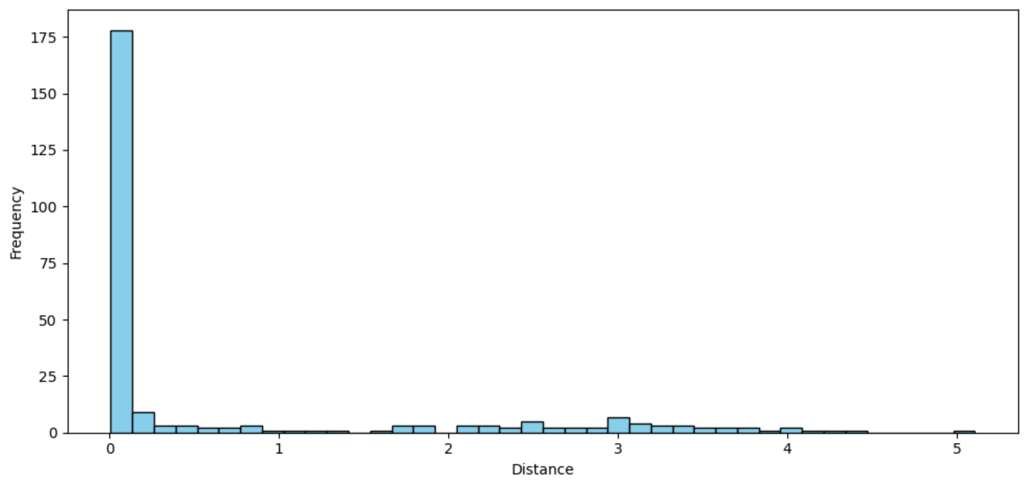

Figure 3.3: Histogram of final position differences for RK2: comparison between $\Delta t = 40\,\text{ms}$ and $\Delta t = 20\,\text{ms}$.

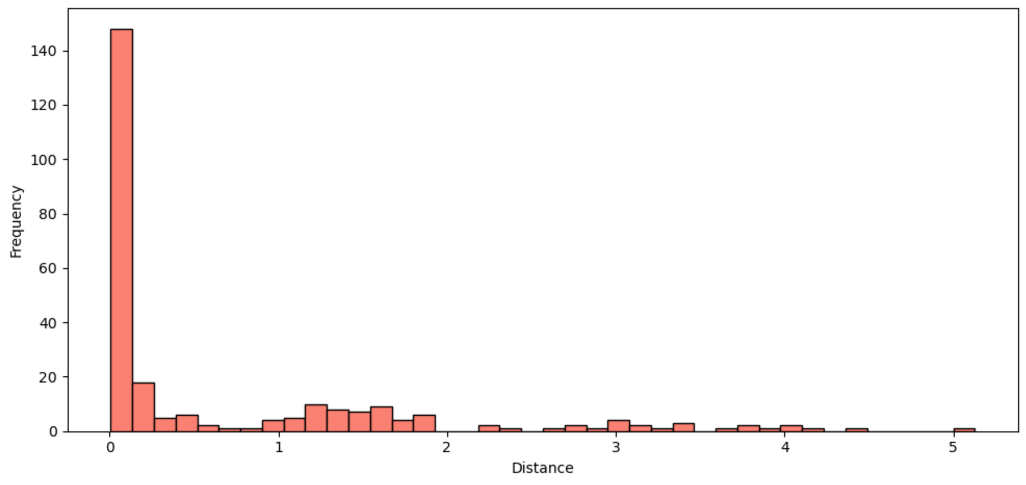

Figure 3.4: Histogram of final position differences for RK2: comparison between $\Delta t = 40\,\text{ms}$ and $\Delta t = 10\,\text{ms}$.

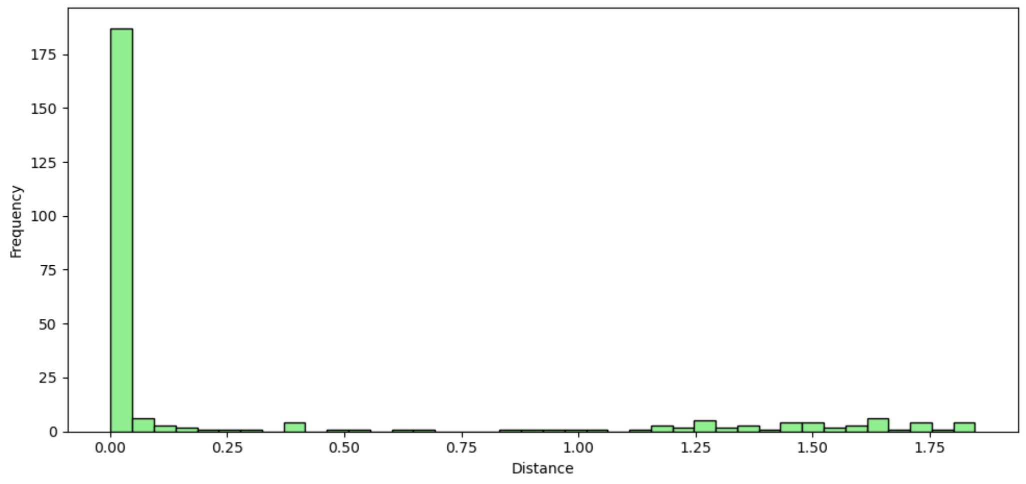

Figure 3.5: Histogram of final position differences for RK2: comparison between $\Delta t = 20\,\text{ms}$ and $\Delta t = 10\,\text{ms}$.

The RK4 method exhibited improved performance, confirming its higher accuracy. Average deviations were reduced to 1.8 mm (10–20 ms), 3.8 mm (20–40 ms), and 3.2 mm (10–40 ms). Additionally, the maximum observed differences were smaller: just over 1 cm between the shortest two time steps, and no more than 3 cm when comparing 40 ms to the others.

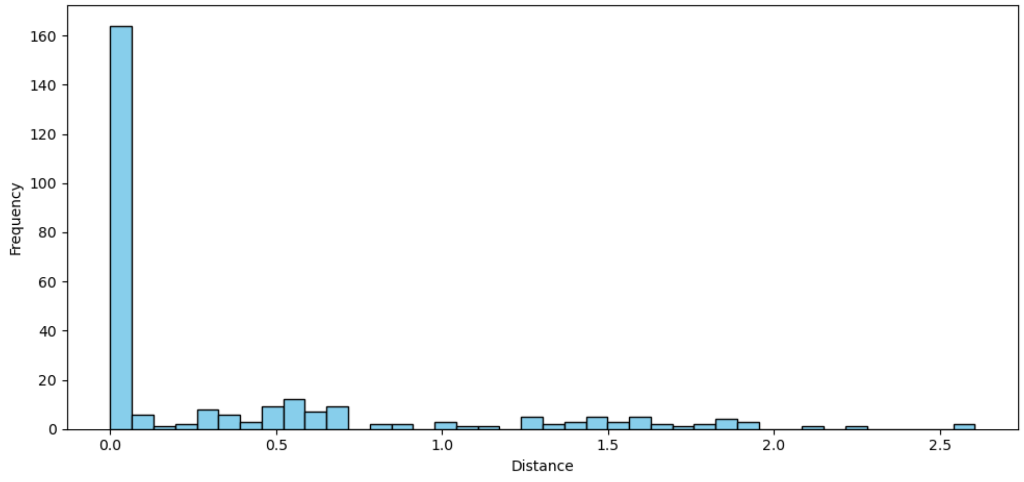

Figure 3.6: Histogram of final position differences for RK4: comparison between $\Delta t = 40\,\text{ms}$ and $\Delta t = 20\,\text{ms}$.

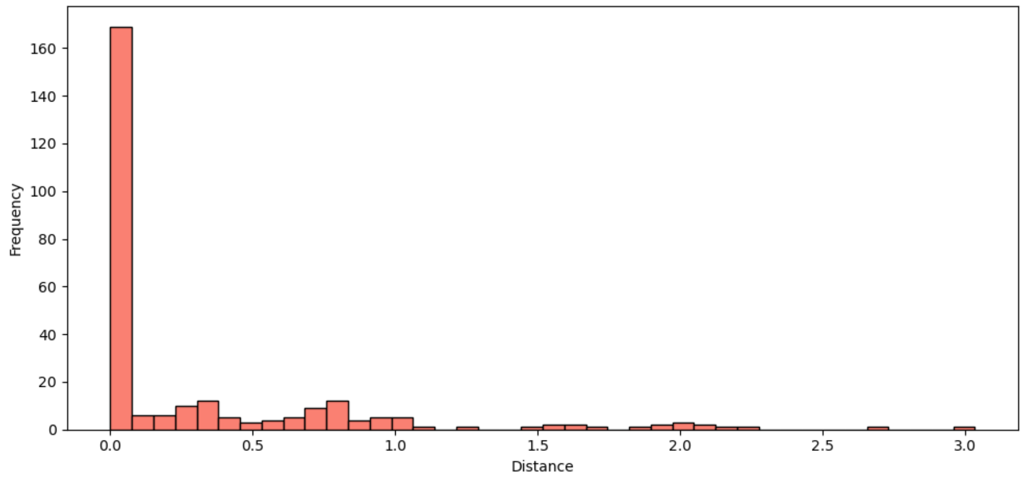

Figure 3.7: Histogram of final position differences for RK4: comparison between $\Delta t = 40\,\text{ms}$ and $\Delta t = 10\,\text{ms}$.

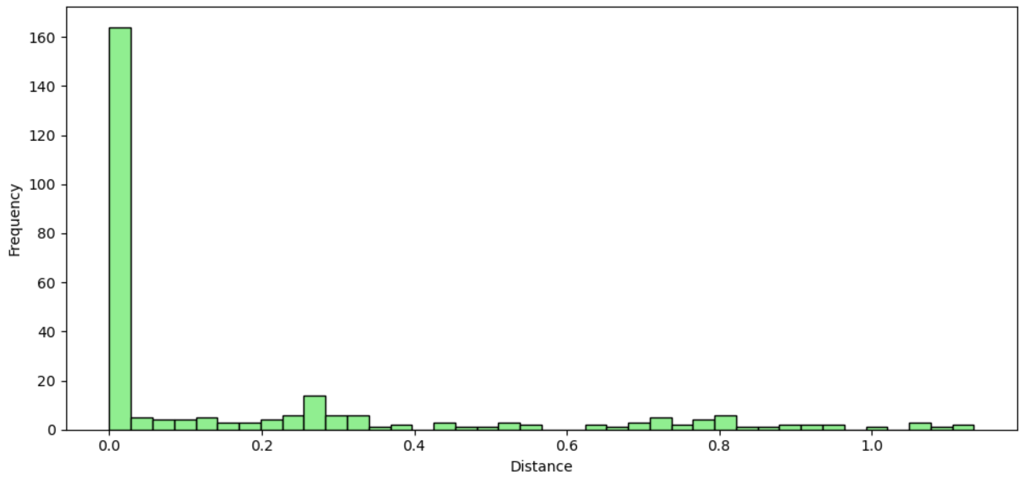

Figure 3.8: Histogram of final position differences for RK4: comparison between $\Delta t = 20\,\text{ms}$ and $\Delta t = 10\,\text{ms}$.

In conclusion, both RK2 and RK4 demonstrated high accuracy and numerical stability, even at relatively large time steps. While RK4 provided the most precise results, RK2 offered a better balance between accuracy and computational efficiency.

3.4.3 Performance Test

This third test focuses on evaluating the performance of the simulation while accounting for all forces acting on the ball. Each integration method was tested by launching the ball under identical initial conditions, both linear and angular velocities, to ensure that all aerodynamic forces were active. The simulation continued after the initial impact, allowing the ball to bounce and eventually come to a stop while rolling on the ground.

To ensure the results were as comparable as possible, 25,000 frames were runned for each method, but only the last 20,000 were used for performance analysis, as the first few hundred frames were affected by the CPU's cache and branch predictor not yet reaching stable performance. Additionally, any frames during which the ball collided with the ground were excluded, since the collision detection algorithm introduces significant computational overhead that could skew the results.

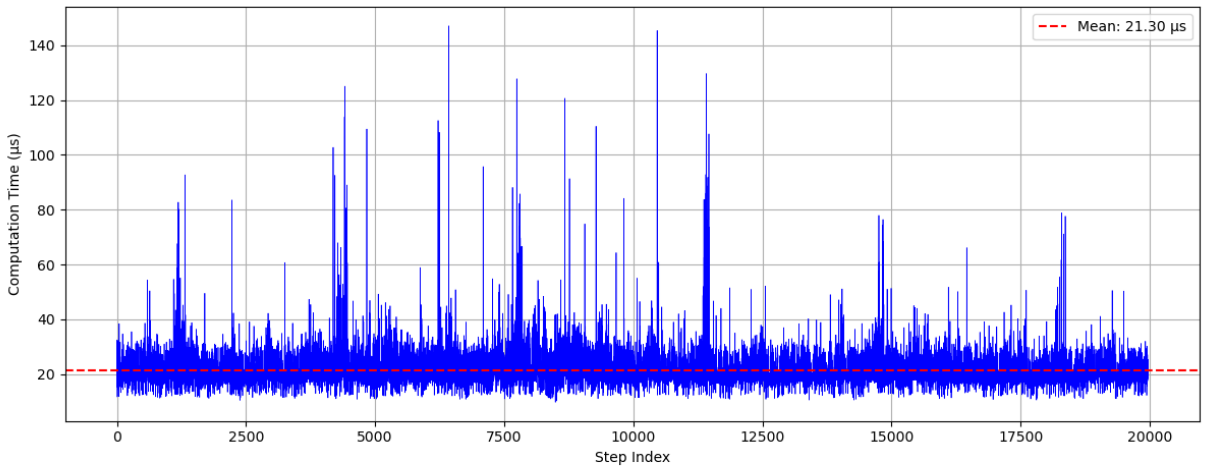

Figure 3.9: Execution time data for RK4 before filtering, showing spikes.

Despite the precautions taken, the data still contained noticeable spikes that could alter the analysis, as illustrated in Figure 3.9. These spikes are likely caused by background system processes or internal Unity operations. To mitigate their impact, the data was filtered to exclude all values above the 99th percentile. This filtering effectively removed the spikes, as shown in Figure 3.10, while preserving the majority of the data for a more accurate performance assessment.

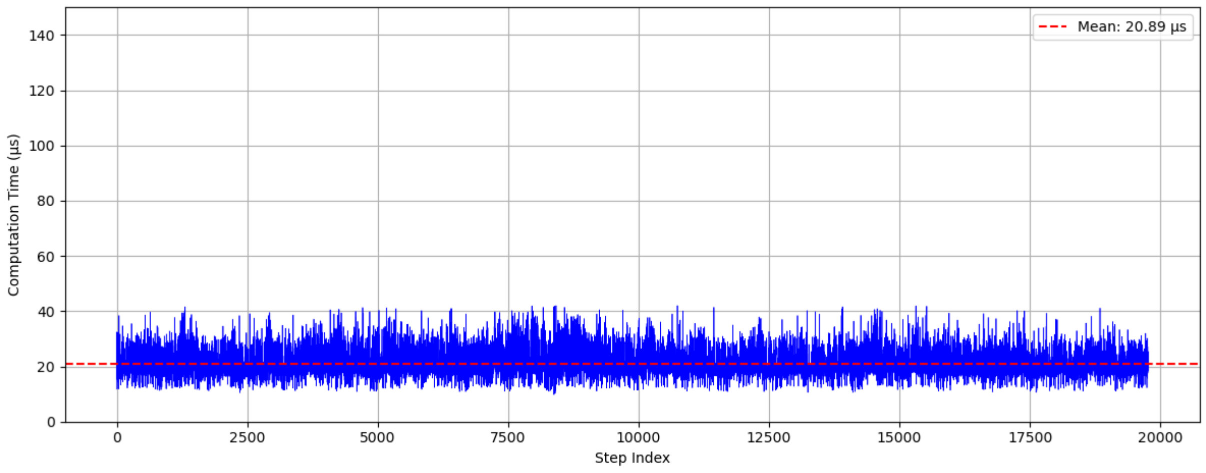

Figure 3.10: Execution time data for RK4 after filtering out values above the 99th percentile, removing spikes and improving data reliability.

Comparison of results

When comparing the performance of the integration methods, the results align with expectations. The Explicit Euler and Semi-Implicit Euler methods showed the fastest computation times, averaging around 11.6 μs per frame—11.2 μs for Explicit Euler and 12.0 μs for Semi-Implicit Euler (Figures 3.11 and 3.12)—but both exhibited the lowest accuracy among the tested methods.

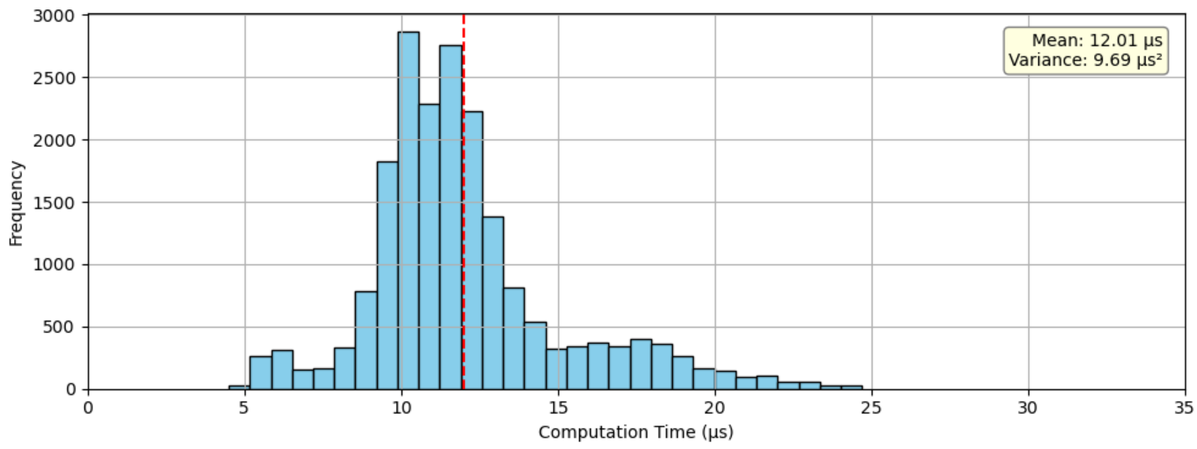

Figure 3.11: Performance profile of the Explicit Euler method.

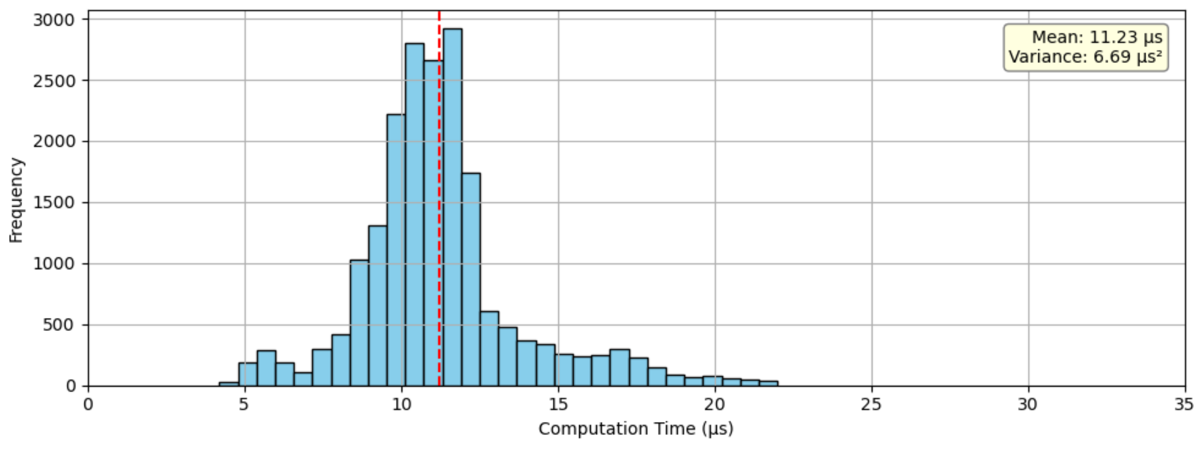

Figure 3.12: Performance profile of the Semi-Implicit Euler method.

The RK2 method, which is of second order, performed slightly slower, averaging around 14.7 μs respectively. These method strikes a good balance between computational efficiency and accuracy, making it a solid choice for real-time simulations (3.13).

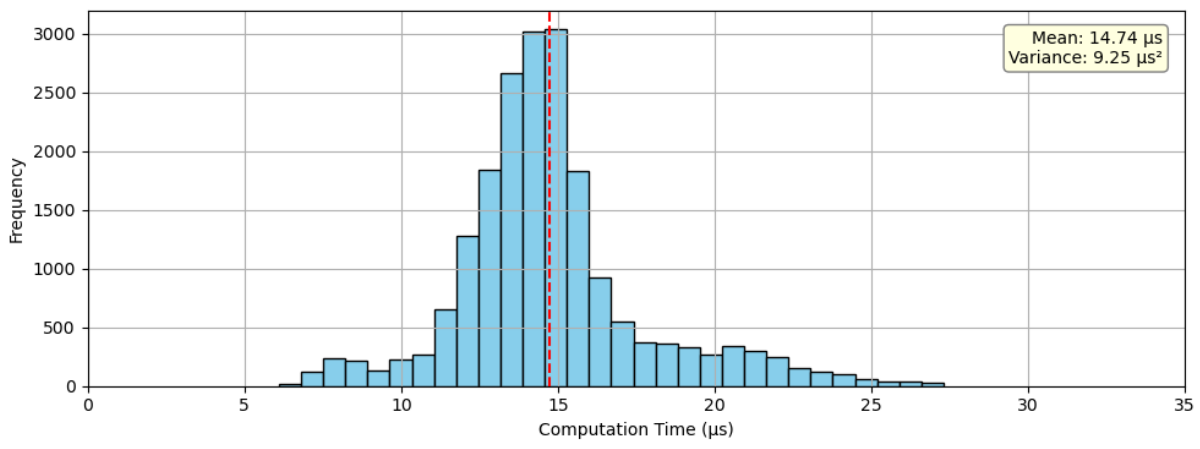

Figure 3.13: Performance profile of the RK2 method, offering good computational efficiency with improved accuracy.

Finally, the RK4 method, while offering the highest accuracy, was the slowest, with an average computation time of about 20.9 μs—nearly twice that of the Euler methods (Figure 3.14).

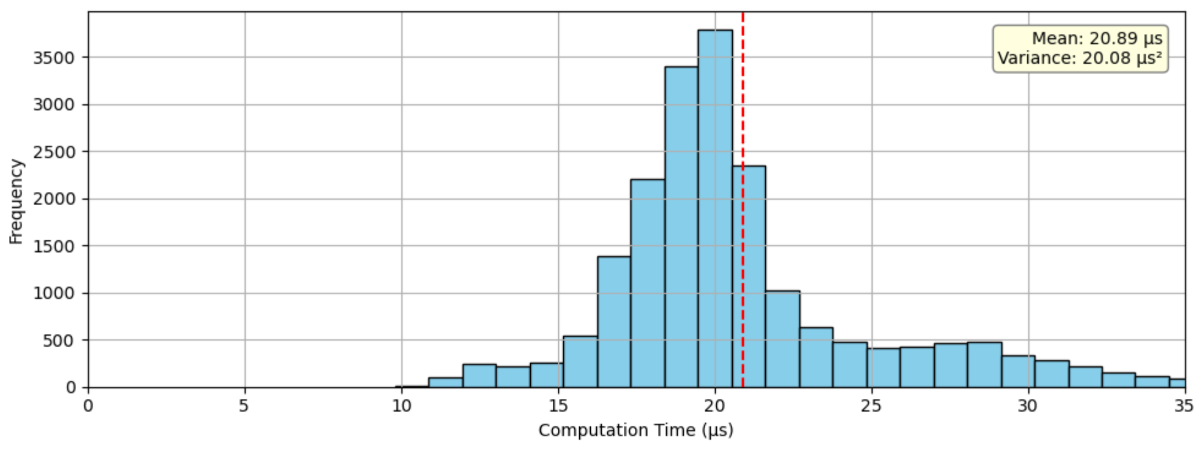

Figure 3.14: Performance profile of the RK4 method, providing the highest accuracy at the cost of computation speed.

3.4.4 Conclusion

Based on the test results, RK2 stands out as the most balanced method, providing a good compromise between accuracy and computational efficiency. While RK4 offers higher accuracy, this level of precision is often unnecessary for our purposes and comes at a significantly greater computational cost. On the other hand, Euler methods, despite their simplicity and low computational demands, showed noticeably worse accuracy, which makes them less suitable when a realistic and stable solution is needed. Therefore, RK2 was chosen as the primary integration method for the following chapter, as it effectively balances performance and accuracy.

Chapter 4

Shot Prediction Systems

The second part of this project focused on developing a system capable of predicting the initial conditions needed to send the ball from an initial position to a desired target. This problem arises frequently in gameplay scenarios—such as shots on goal, passes, or free kicks—and requires determining the correct combination of direction, speed, and spin to accurately reach the target under physical constraints.

Three methods were implemented:

- **Method A:** A fast and efficient predictor designed to compute the initial velocity vector required to reach a target, given a fixed speed and spin. This method is preferred for low elevation angle and during during dynamic gameplay.
- **Method B:** A more accurate but computationally heavier version of Method A. It iteratively adjusts the elevation angle of the velocity vector to improve precision using a binary search algorithm and it could be useful for static shots such as free kicks or corners, where higher arcs and fine control are more relevant.
- **Two-Point Constraint Method:** A more complex predictor that computes the optimal spin to apply to the ball, given a fixed speed and initial position. The goal is to curve the trajectory so that the ball passes as close as possible to an intermediate constraint point, typically a position above or around a defensive wall, and then reaches the final target. This method is especially suited for recreating realistic curved free kicks.

All three approaches are based on an iterative numerical procedure that solves an optimization problem. Given an objective function defined as the distance between the ball's closest approach and the target,

$$f(I) = \|\vec{p}_b - \vec{p}_t\| \tag{4.1}$$

where $\vec{p}_b$ is the closest point reached by the ball during the simulation and $\vec{p}_t$ is the target position, and I represents the initial parameters to optimize (e.g., velocity, spin, elevation), the goal is to find a solution such that:

$$f(I) < \epsilon \tag{4.2}$$

for some desired error threshold ϵ [1].

If the ball's speed is not sufficient to reach the target, the system will still try to minimize the error and reach the closest possible position. Each method uses different optimization strategies tailored to its constraints, balancing accuracy and performance to fit within a real-time environment.

4.1 Method A – Angle-Based Iterative Prediction

The idea behind this is that, given an input of spin and speed, the predictor will find the optimal velocity to send the object towards the target. This velocity can be described using spherical coordinates with two angles: the elevation angle θ (measured from the horizontal plane) and the azimuth angle ϕ (measured around the vertical axis).

The velocity vector $\vec{v}$ can then be expressed as:

$$\vec{v} = v \begin{pmatrix} \cos\theta_E \cos\theta_A \\ \sin\theta_E \\ \cos\theta_E \sin\theta_A \end{pmatrix} \tag{4.3}$$

where

- v is the speed (magnitude of the velocity),
- $\theta_E \in [a, b]$ represents the elevation angle, where $a \geq 0$ and $b \leq 45°$.
- $\theta_A \in [0, 2\pi)$ is the azimuth angle.

Therefore, since the speed v is given, the only objective is to find θ_E and θ_A.

4.1.1 Procedure

Initial Velocity Estimation

The first step is to compute an initial guess for the velocity. This initial estimate is directed toward the target, with the maximum elevation angle. This provides a starting point that maximizes flight time and arc height, increasing the likelihood of reaching the target.

To do this, we take the horizontal direction vector $\vec{d}_{\text{hor}}$ pointing from the ball's initial position $\vec{p}_b$ to the target position $\vec{p}_t$, constrained to the horizontal plane by setting the y-component to zero.

$$\vec{d}_{\text{hor}} = \begin{pmatrix} \vec{p}_{t,x} - \vec{p}_{b,x} \\ 0 \\ \vec{p}_{t,z} - \vec{p}_{b,z} \end{pmatrix} \tag{4.4}$$

This horizontal direction is then normalized obtaining:

$$\hat{d}_{\text{hor}} = \frac{\vec{d}_{\text{hor}}}{\|\vec{d}_{\text{hor}}\|} \tag{4.5}$$

Next, we elevate this direction vector by an angle $\theta_E = b$. We compute a rotation axis $\vec{a}$ as the cross product between the horizontal direction $\hat{d}_{\text{hor}}$ and the up vector $\hat{u} = (0, 1, 0)$:

$$\hat{a} = \hat{d}_{\text{hor}} \times \hat{u} \tag{4.6}$$

We then rotate $\vec{d}_{\text{hor}}$ around the axis $\hat{a}$ by angle θ_E, resulting in the elevated direction vector $\vec{d}$:

$$\left[0, \hat{d}\right] = q \otimes \left[0, \hat{d}_{\text{hor}}\right] \otimes q^*, \tag{4.7}$$

where

$$q = \left[\frac{\cos\theta_E}{2},\ \hat{a}\frac{\sin\theta_E}{2}\right]$$

is a quaternion and $\hat{q}^*$ is its conjugate [17]. The rotated vector $\hat{d}$ is then scaled by the desired speed magnitude v to obtain the full initial velocity vector:

$$\vec{v}_{\text{init}} = v \cdot \hat{d} \tag{4.8}$$

This method allows the generation of an elevated trajectory vector while preserving the initial horizontal direction and total speed magnitude. At this point, the ball's trajectory is simulated using the initial velocity guess, and the point on the path closest to the target p_g is identified.

Iterative Correction Process

The iterative refinement then begins: we compute two direction vectors, $\vec{u}_{\text{target}} = p_t - p_b$, pointing from the ball's initial position to the target, and $\vec{u}_{\text{guess}} = p_g - p_b$, pointing from the initial position to the closest point on the current trajectory. The angle θ between these vectors is then calculated (Figure 4.1).

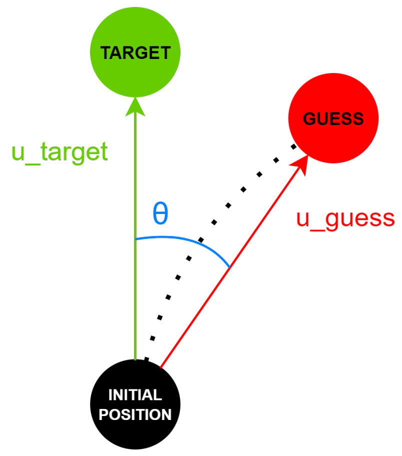

Figure 4.1: Angle θ between the vectors $\vec{u}_{\text{target}}$ and $\vec{u}_{\text{guess}}$. The dotted line represents the ball's trajectory.

The velocity vector $\vec{v}_{\text{old}}$ is then rotated by the angle θ around the axis needed to align $\vec{u}_{\text{guess}}$ with $\vec{u}_{\text{target}}$, obtaining a new velocity guess $\vec{v}_{\text{new}}$ (Figure 4.2). The trajectory is re-simulated with the updated velocity, and the new closest point is evaluated.

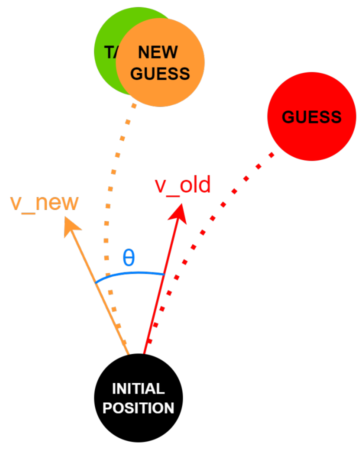

Figure 4.2: The velocity vector $\vec{v}_{\text{old}}$ is rotated by the angle θ, resulting in a new velocity $\vec{v}_{\text{new}}$. The updated trajectory brings the ball closer to the target, refining the guess.

This process repeats until the distance between the predicted impact point and the target falls below a specified tolerance ϵ.

Aligning the Spin with the Velocity Direction

The spin is represented in a 2D coordinate system (spin $= (x, y)$), where the y-component controls the sideways curvature of the ball, and the x-component affects the vertical lift or dip of the trajectory. However, the spin must be converted into a 3D vector to be used in the simulation. During each iteration, the horizontal component of the spin needs to be aligned with the horizontal direction of the velocity. This ensures that the spin is always applied relative to the ball's current direction of motion.

Let $\vec{v}_{\text{hor}}$ be the velocity projected on the horizontal plane. The aligned spin direction is computed as:

$$\vec{s}_{\text{hor}} = \text{spin}_x \cdot \left(\frac{\vec{v}_{\text{hor}}}{\|\vec{v}_{\text{hor}}\|} \times \vec{u} \right) \tag{4.9}$$

where $\vec{u} = (0, 1, 0)$ is the up vector. The total spin becomes:

$$\vec{s} = \vec{s}_{\text{hor}} + \text{spin}_y \cdot \vec{u} \tag{4.10}$$

This ensures that the spin always acts in the appropriate direction relative to the current velocity, maintaining consistent physical behavior throughout the simulation.

4.1.2 Implementation

Vector Elevation

As an initial guess for the velocity, a vector pointing toward the target and inclined at an elevation angle $\theta_E = b$ is proposed, where b is the maximum elevation angle. The steps described in Equations (4.4–4.8) were implemented in a function that takes a vector and an elevation angle as input parameters and returns a vector tilted above the horizontal plane, as shown below:

```
Vector3 ElevateVector(Vector3 vector, float elevationAngle)
{
    //horizontal direction of the vector
    Vector3 dirHor = new Vector3(vector.x, 0f, vector.z).normalized;

    //find rotation axis perpendicular to both the up-vector and the horizontal direction
    Vector3 rotationAxis = Vector3.Cross(dirHor, Vector3.up);

    //compute the quaternion corresponding to the rotation
    Quaternion rotation = Quaternion.AngleAxis(elevationAngle, rotationAxis);

    //apply the rotation
    Vector3 rotatedVector = rotation * dirHor;

    //rescale the vector
    Vector3 rescaledVector = rotatedVector * vector.magnitude;

    return rescaledVector;
}
```

To compute the rotation quaternion, the function `Quaternion.AngleAxis` is used. It takes an angle and a rotation axis as arguments and returns the corresponding quaternion. The rotation is then applied to the vector using the `*` operator, which performs the rotation as described in Equation 4.7.

Velocity Correction

The part of adjustement of the velocity direction is implemented this way:

```
Vector3 RotateVelocityTowardTarget(Vector3 initialPosition, Vector3 guessPosition, Vector3
     targetPosition, Vector3 velocity)
{
    Vector3 uGuess = guessPosition - initialPosition;
    Vector3 uTarget = targetPosition - initialPosition;

    //compute rotation axis, perpendicular to the plane oh which uGuess and uTarget lie
    Vector3 rotationAxis = Vector3.Cross(uGuess, uTarget).normalized;

    //compute angle between uGuess and uTarget
    float theta = Vector3.SignedAngle(uGuess, uTarget, rotationAxis);

    //compute the rotation needed to align uGuess with uTarget
    Quaternion rotation = Quaternion.AngleAxis(theta, rotationAxis);

    //apply rotation to velocity
    Vector3 rotatedVelocity = rotation * velocity;

    return rotatedVelocity;
}
```

In this case, another useful function from Unity was used: `Vector3.SignedAngle`. This function takes three vectors as parameters and computes the signed angle between the first two vectors, relative to the third vector, which represents the axis of rotation. The rotation is then computed as before and applied to the velocity vector in the same way as previously described.

Spin Alignment

Finally, the function that aligns the spin vector with the velocity direction is implemented. This function takes the current velocity and a 2D spin input as parameters, as described in Section 4.1.1.

```
Vector3 AlignSpinWithVelocityDirection(Vector3 velocity, Vector2 spin)
{
    //horizontal direction of velocity
    Vector3 direction = new Vector3(velocity.x, 0f, velocity.z).normalized;

    //direction to align spin
    Vector3 alignedSpinDirection = Vector3.Cross(Vector3.up, direction);

    //horizontal magnitude of spin
    float horizontalSpinMagnitude = spin.x;

    //rescale aligned direction to match original spin magnitude
    return alignedSpinDirection * horizontalSpinMagnitude + new Vector3(0f, spin.y, 0f);
}
```

Predictor

After defining the necessary functions to simulate the trajectory, the final prediction method can be implemented.

The function takes as input: the initial position, target position, initial speed, spin, and the time step used for the simulation. First, an initial velocity guess is computed, the trajectory is simulated, and a first velocity correction is applied, followed by an error evaluation. Next, the trajectory is evaluated with the minimum allowed elevation angle. This step is needed to determine whether the target is reachable within the defined elevation angle limits. By comparing the simulated trajectories at both maximum and minimum elevation, we can verify whether the current speed is too low or too high to reach the target.

If the target is reachable, the iterative correction loop begins: if the error exceeds a predefined threshold, the velocity is adjusted and the trajectory re-simulated. This process repeats until the error falls below the threshold.

```
(Vector3, Vector3) PredictInitialValues(float timeStep, Vector3 targetPosition, Vector3
    initialPosition, float speed, Vector2 spin)
{
    //compute the initial guess of the velocity
    (Vector3 velocityEstimation, Vector3 alignedSpin) = InitialVelocityGuess(initialPosition,
        targetPosition, speed, elevationAngleMaximumLimit, spin);

    //compute the trajectory and find the closest point to the target
    (Vector3 guessPosition,_) = EstimateTrajectoryClosestPoint(timeStep, initialPosition,  SysQuat.
        Identity, velocityEstimation, alignedSpin, targetPosition);

    //calculate error
    float errorInCentimeters = Vector3.Distance(guessPosition, targetPosition) * 1e2f;

    //compute velocity with lowest elevation and simulate trajectory
    Vector3 minElevatedVelocity = ElevateVector(velocityEstimation, elevationAngleMinimumLimit);
    (Vector3 minElevationPositionGuess,_) = EstimateTrajectoryClosestPoint(timeStep, initialPosition,
        SysQuat.Identity, minElevatedVelocity, alignedSpin, targetPosition);

    //check the ball can reach the target
    if(guessPosition.y > targetPosition.y && minElevationPositionGuess.y < targetPosition.y)
    {
        while (errorInCentimeters > maxErrorInCentimeters)
        {
           //compute new guess of the velocity
            velocityEstimation = RotateVelocityTowardTarget(initialPosition, guessPosition,
                targetPosition, velocityEstimation);

            //spin is aligned with new velocity direction
            alignedSpin = AlignSpinWithVelocityDirection(velocityEstimation, spin);

            //compute the trajectory and find the closest point to the target
            (guessPosition, _) = EstimateTrajectoryClosestPoint(timeStep, initialPosition, SysQuat.
                Identity, velocityEstimation, alignedSpin, targetPosition);

            //calculate error
            errorInCentimeters = Vector3.Distance(guessPosition, targetPosition) * 1e2f;
        }
    }
    else
    {
        //set low elevation trajectory if it goes above the target, otherwise use high limit
        if (minElevationPositionGuess.y > targetPosition.y)
            velocityEstimation = minElevatedVelocity;
    }
    return (velocityEstimation, alignedSpin);
}
```

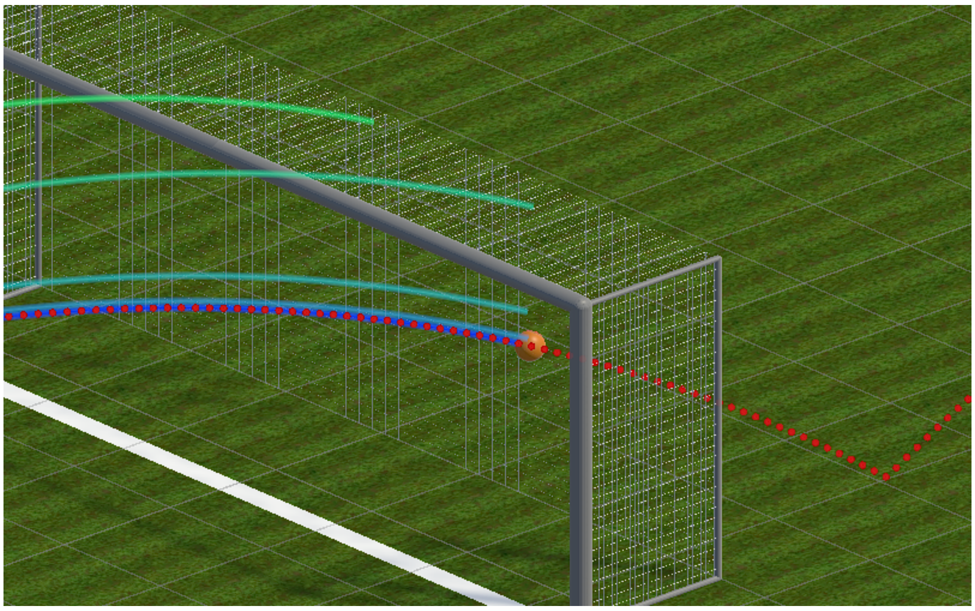

Figure 4.3: The image shows the predictor correcting the trajectories until the ball can reach the target close with enough accuracy

4.1.3 Results

The implemented method was tested to evaluate its accuracy and limitations. Since the method itself uses an approximate way to measure the error—by interpolating between the two closest simulated frames—a more accurate error computation method was implemented for testing purposes only.

Testing Setup

In the implemented prediction method, the closest point to the target is approximated by interpolating along the segment between the two closest simulated frames, assuming a roughly straight trajectory between them. While this provides a fast estimation, it is not fully accurate and may lead to accepting velocities that don't meet the desired precision threshold.

To rigorously evaluate the actual error, a more accurate method was introduced exclusively for testing. After the final velocity guess is computed, the trajectory is re-simulated. At each frame, three consecutive positions p_{n-2}, p_{n-1}, p_n are tracked. If the following condition is met:

$$\|p_{n-1} - p_t\| < \|p_{n-2} - p_t\| \quad \text{and} \quad \|p_{n-1} - p_t\| < \|p_n - p_t\|, \tag{4.11}$$

then the ball has reached its minimum distance to the target somewhere between frames $n-2$ and n. Depending on which endpoint is closer to the target, the algorithm refines the segment using a recursive binary search-like approach to find the most accurate point of minimum distance. This technique is similar to the method described in Section 2.4.4.

```
Vector3 RecursiveClosestPointSearch(float deltaTime, float step, Vector3 position, SysQuat orientation
    , Vector3 velocity, Vector3 angularVelocity, BallState state, Vector3 targetPosition, float
    previousFrameDistance, float nextFrameDistance)
{
    (Vector3 newPosition, _, _, _) = IntegrateMotion(deltaTime, position, orientation, velocity,
        angularVelocity, state);
    float newDistance = Vector3.Distance(newPosition, targetPosition);

    bool goingBackward = previousFrameDistance < nextFrameDistance;
    float newDeltaTime = deltaTime + (goingBackward ? -step / 2f : step / 2f);

    if (step < TimeStepThreshold)
    {
        (Vector3 closestPoint, _, _, _) = IntegrateMotion(newDeltaTime, position, orientation,
            velocity, angularVelocity, state);
        return closestPoint;
    }

    return RecursiveClosestPointSearch(newDeltaTime, step / 2f, position, orientation, velocity,
        angularVelocity, state, targetPosition, goingBackward ? previousFrameDistance : newDistance,
        goingBackward ? newDistance : nextFrameDistance);
}
```

This accurate error computation is used only in the testing phase, as including it in the prediction method would introduce a significant computational overhead.

Evaluation of Accuracy and Efficiency

A test was conducted by predicting multiple shots, with initial positions ranging from approximately $15\,m$ to $25\,m$ from the target. Each trajectory was simulated with RK2 using a time-step of $10\,ms$, an initial velocity of $25\,m/s$, and a fixed spin vector spin $= (20, 10)$. The predictor was configured with a stopping threshold of $1\,cm$ and a maximum elevation angle of $30°$.

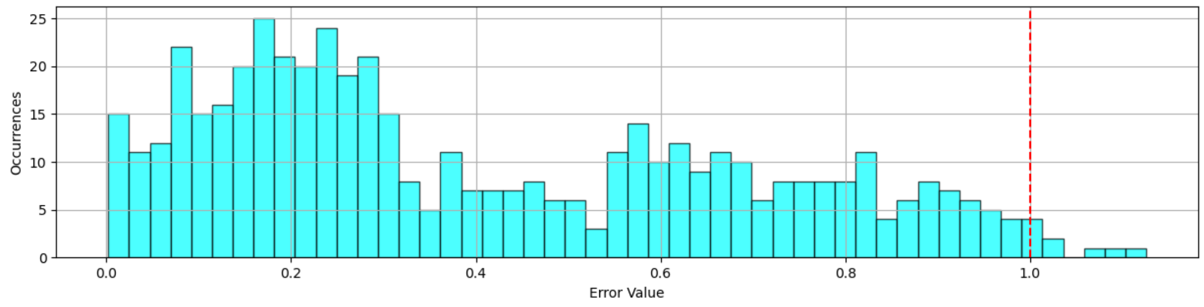

Figure 4.4: Histogram of the errors, using a time-step of $10\,ms$

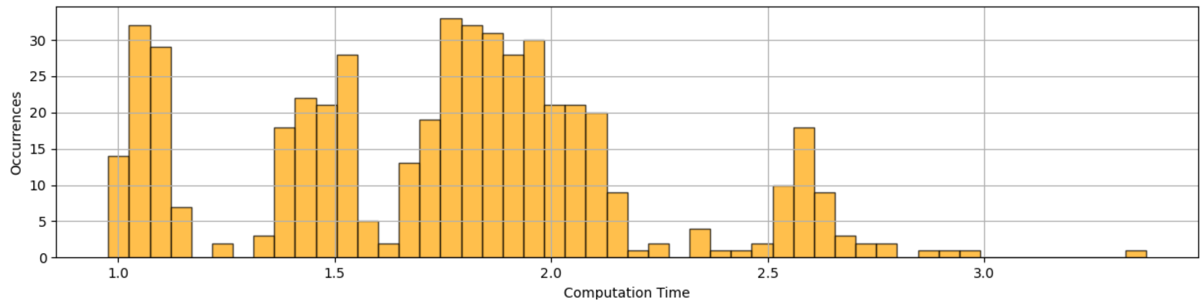

Figure 4.5: Histogram of the computation times, using a time-step of $10\,ms$

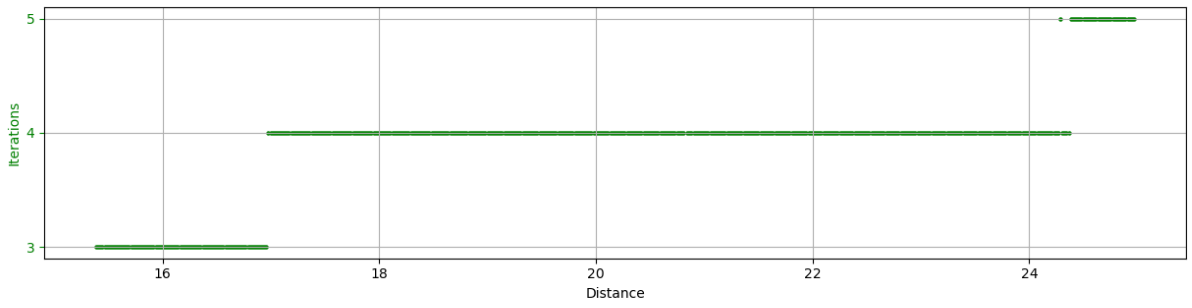

Figure 4.6: Iterations required to converge for shots at different initial distances from the target, using a time-step of $10\,ms$

The results show that the final error exceeded the threshold in only a few cases, and by no more than $2\,mm$. The average error across all simulations was approximately $4\,mm$, which can still be considered a very good outcome. The average computation time per prediction was around $1.75\,ms$, with occasional spikes likely due to background processing, as the number of iterations remained fairly constant across all runs.

The same test was repeated using a larger time-step of $40\,ms$, four times greater than the previous configuration, to evaluate whether the predictor's approximation remained viable when using longer simulation steps. The accurate error was still computed using $10\,ms$ steps to ensure consistency in the evaluation.

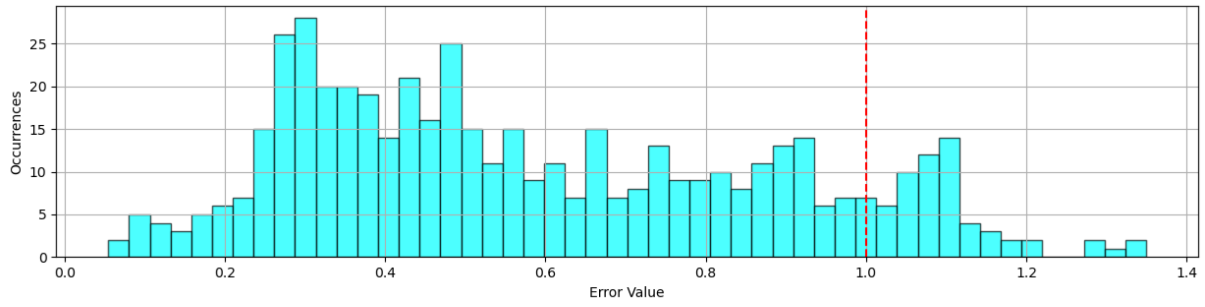

Figure 4.7: Histogram of the errors, using a time-step of $40\,ms$

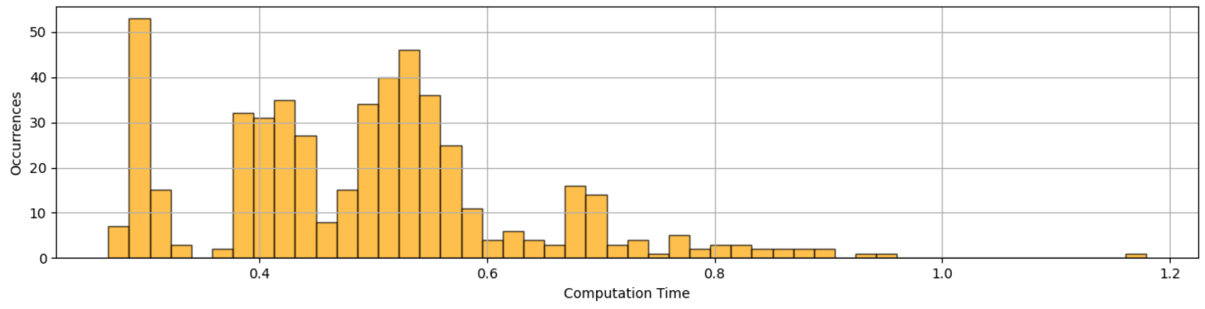

Figure 4.8: Histogram of the computation times, using a time-step of $40\,ms$

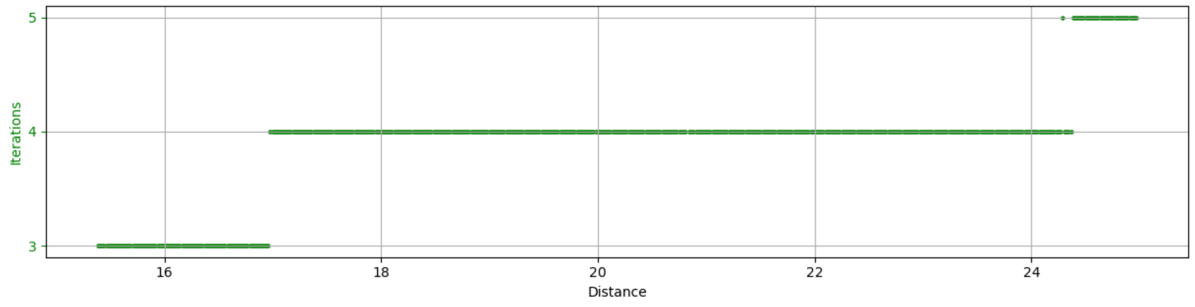

Figure 4.9: Iterations required to converge for shots at different initial distances from the target, using a time-step of $40\,ms$

As expected, the average error increased slightly to $5.1\,mm$, with more frequent violations of the $1\,cm$ threshold. However, the maximum excess remained below $4\,mm$. On the other hand, the average computation time was reduced significantly to $0.49\,ms$.

These results demonstrate that the method maintains good accuracy, even when using larger time-steps and a faster, approximate error measure. Although the method does not compute the most precise error during prediction, it still yields a very low actual error, validating its use in real-time applications where performance is a constraint.

Limitation at High Elevation Angles

To assess the robustness of the method under edge conditions, additional tests were conducted in which the ball was shot from increasingly distant positions, with a maximum elevation angle of 45°. In these scenarios, the initial speed and distance were set such that the target could not be reached directly, forcing the method to select high-arching trajectories in an attempt to maximize range.

These tests revealed a limitation of the approach: when the elevation angle exceeds approximately 30°, the number of iterations required for convergence increases rapidly, leading to significantly longer computation times, as shown in the following figures.

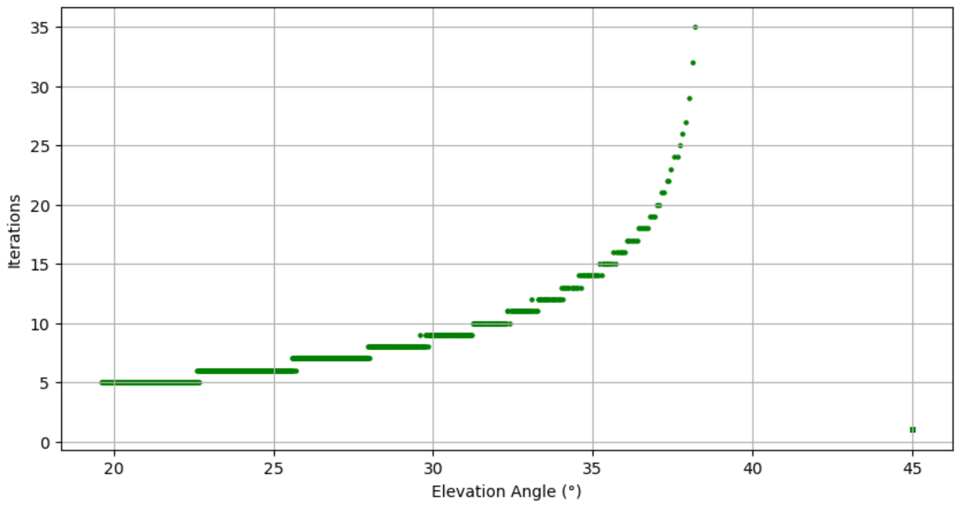

Figure 4.10: Number of iterations required to converge and resulting elevation angles. The iteration count increases significantly beyond 30°, then collapses to 1 at 45°, where the shot lacks sufficient speed to reach the target.

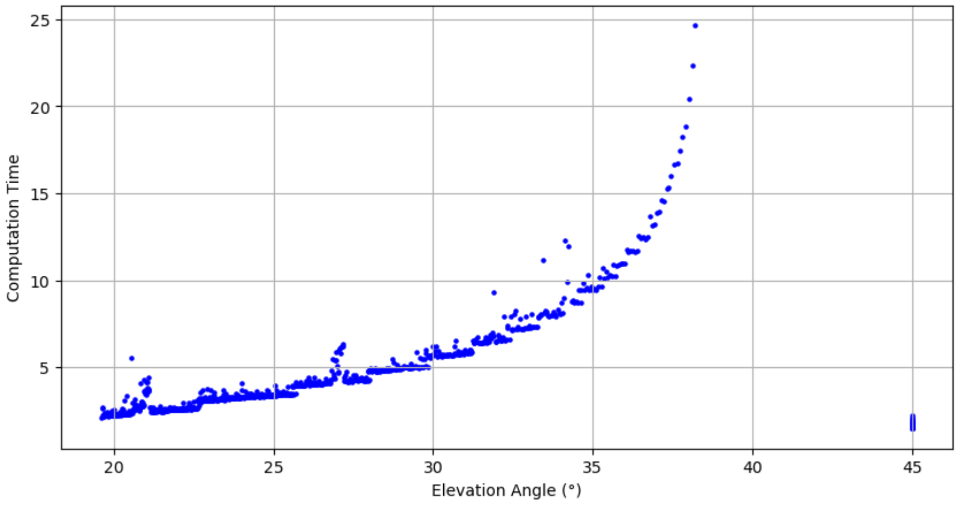

Figure 4.11: Computation times and final elevation angles. The computation time increases in parallel with the number of iterations, diverging notably beyond 30°.

This issue persists even in the absence of aerodynamic forces, indicating that it is not caused by drag or lift, but rather by an intrinsic property of projectile motion.

The cause lies in the mathematical behavior of the projectile range, given by:

$$R = \frac{v^2 \sin(2\theta_E)}{g}, \tag{4.12}$$

where θ_E is the elevation angle. The function $\sin(2\theta_E)$ has a nonlinear growth: its derivative is largest at low angles and flattens as θ_E approaches 45°. This implies that small changes in angle at high elevation result in relatively minor variations in range, making the trajectory less sensitive to corrections. Conversely, at low angles, similar changes lead to more significant differences in the outcome, which helps the method converge more efficiently. In short, as the angle increases, the method becomes less effective at adjusting the shot.

Comparison with GATTINI.games Prediction Model

This section presents a qualitative comparison between the prediction model developed in this thesis and the one implemented by GATTINI.games. While the previous tests employed Runge-Kutta integration methods, this comparison is designed to closely replicate the operational environment of the GATTINI.games system. Their model utilizes the Semi-Implicit Euler method, and therefore, the same integrator was adopted in this evaluation to ensure consistency. Furthermore, the simulation frame rate was matched to their setup, running at 60 Hz.

It is important to note that this comparison cannot be considered absolute, as the GATTINI.games model is implemented in C++ and operates within a different runtime environment. Moreover, the tests were conducted on different machines: the model developed in this thesis was run on a more recent and performant setup (11th Gen Intel Core i5-11400H @ 2.70 GHz, 6 cores / 12 threads), while the GATTINI.games model was tested on an older, less powerful system (Intel Core i7-8565U @ 1.80 GHz, 4 cores / 8 threads), running on battery power. To reduce interference from Unity Editor overhead, the thesis simulation was executed in a standalone build. While these differences prevent a strictly fair comparison, the evaluation still provides meaningful qualitative insights into the relative behavior and efficiency of the two models.

The GATTINI.games model achieves an average computation time between 200 μs and 300 μs, consistently reaching the target with a maximum error of 1 cm, and converging within 10 iterations.

To evaluate the performance of the model developed in this thesis, approximately 5000 shots were simulated with randomized initial conditions and target positions within the goal. The initial velocity magnitude ranged between 20 and 30 m/s, while spin values varied between -20 and 20rad/s on both axes. For this test, the simulation used the simplified physical model with constant coefficients for aerodynamic forces.

The results indicate an average computation time of 261 μs, with a variance of 11 μs. There are however very few occurencies of values around 1 ms, probably due to the garbage collector.

The vast majority of the shots achieved an error below 1 cm, with only a few exceptions slightly above this threshold. The overall average error was 3.7 mm. Moreover, the method converged in no more than 5 iterations across all tests.

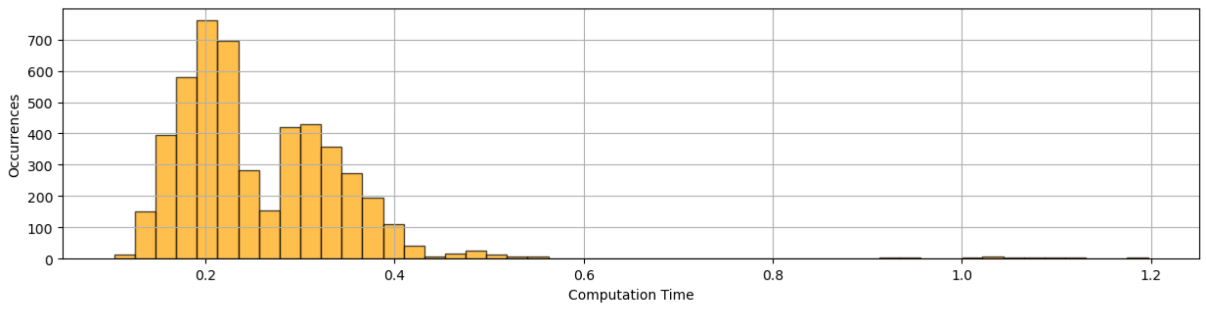

Figure 4.12: Computation time distribution for the shot prediction model.

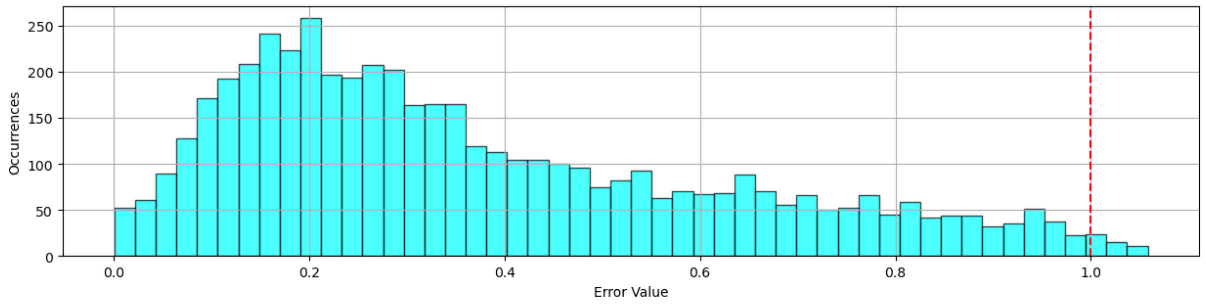

Figure 4.13: Prediction error distribution for randomized shots.

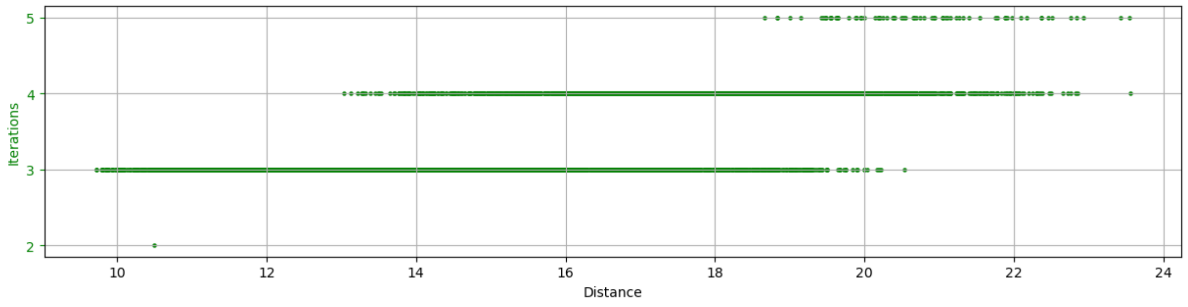

Figure 4.14: Number of iterations required to converge.

4.2 Method B – Binary Search over Elevation

To address the limitations of the previous model, an alternative approach was implemented. While the overall process remains similar, the key difference lies in how the vertical angle is adjusted—precisely the part responsible for the convergence issues.

Instead of adjusting the elevation angle by an amount equal to the angle between the velocity vector and the vector pointing to the target, this version uses a binary search strategy. Rather than adjusting the elevation angle incrementally, the method iteratively reduces the interval between predefined minimum and maximum bounds to converge on the optimal angle.

4.2.1 Binary Search Algorithm

Given a minimum and maximum elevation angle ($\theta_{E,min}$ and $\theta_{E,max}$), the algorithm computes a new trajectory using an elevation of

$$\theta_E = \frac{\theta_{E,\text{max}} + \theta_{E,\text{min}}}{2}, \tag{4.13}$$

i.e., the midpoint between the current bounds.

Then, it checks whether the vertical position of the closest point on the trajectory is above or below the target. If $p_{g,y} < p_{t,y}$, the trajectory is too low, so the lower bound is updated:

$$\theta_{E,min} = \frac{\theta_{E,max} + \theta_{E,min}}{2}. \tag{4.14}$$

Otherwise, the trajectory is too high, and the upper bound is adjusted:

$$\theta_{E,max} = \frac{\theta_{E,max} + \theta_{E,min}}{2}. \tag{4.15}$$

This process continues until the desired accuracy is reached.

Example

An example of this process is illustrated in Figure 4.15.

- We start with an initial guess with elevation $\theta_1 = \frac{\theta_{\max}+\theta_{\min}}{2}$. After computing the trajectory, we observe that the ball lands below the target.
- Consequently, we update the lower bound and compute a new angle $\theta_2 = \frac{\theta_{\max}+\theta_1}{2}$. This time, the ball lands above the target.
- We then refine the estimate with $\theta_3 = \frac{\theta_2+\theta_1}{2}$. At this point, the resulting trajectory reaches the desired accuracy, and the process terminates.

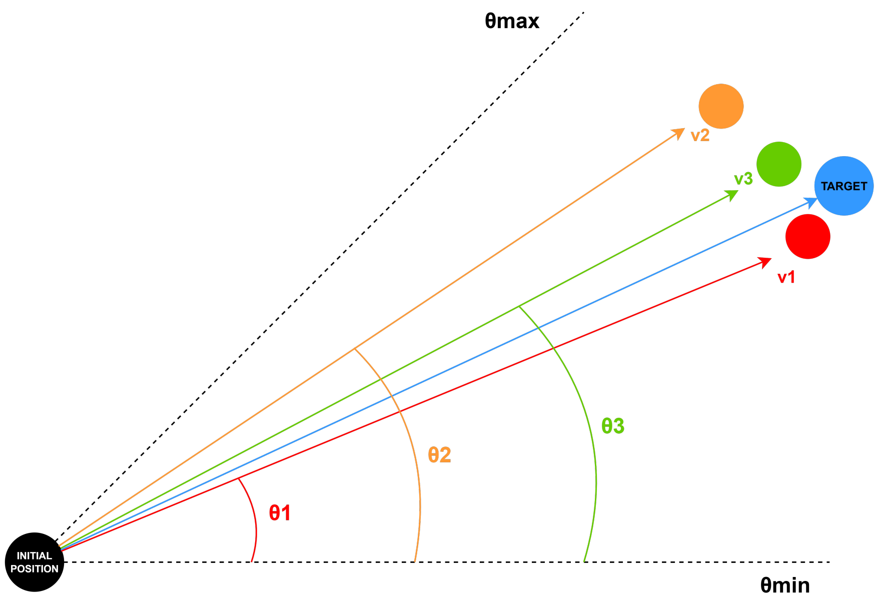

Figure 4.15: Example of the binary search process used to adjust the elevation angle.

4.2.2 Implementation

The main difference in the implementation of this prediction method lies within the while loop, where the elevation angle is iteratively updated, and the elevation of the velocity vector is adjusted accordingly, as explained earlier.

```
while (errorInCentimeters > maxErrorInCentimeters)
{
    //calculate the iteration elevation angle
    float elevationAngle = (elevationAngleLowLimit + elevationAngleHighLimit) / 2;

    //compute new guess of the velocity
    Vector3 velocityEstimation = RotateVelocityTowardTarget(initialPosition, guessPosition,
        targetPosition, velocityEstimation);

    //velocity is elevated
    velocityEstimation = ElevateVector(velocityEstimation, elevationAngle);

    //spin is aligned with new velocity direction
    alignedSpin = AlignSpinWithVelocityDirection(velocityEstimation, spin);

    //compute the trajectory and find the closest point to the target
    (guessPosition, _) = EstimateTrajectoryClosestPoint(timeStep, initialPosition, SysQuat.Identity,
        velocityEstimation, alignedSpin, targetPosition);

    //calculate error
    errorInCentimeters = Vector3.Distance(guessPosition, targetPosition) * 1e2f;

    //update one of the two limits of the elevation angle
    if (guessPosition.y < targetPosition.y)
        elevationAngleLowLimit = elevationAngle;
    else
        elevationAngleHighLimit = elevationAngle;
}
```

4.2.3 Results

If we perform again the test described in 4.1.3, which evaluated the first method at high elevation angles, but now apply it to the new method, we observe that the new method performs better. The number of iterations remains stable at around 10, with a maximum of 11 in the worst cases. Regarding computation times, they no longer diverge after a certain point; instead, they increase slowly and approximately linearly, which is likely due to the longer trajectories requiring more frames to be computed.

Similar to the previous test with the old method, we notice a jump in the elevation angle from about 38° to 45°. This behavior shows that even with the new method, after a certain elevation angle, small changes have less effect, causing the predictor to jump to the maximum possible value.

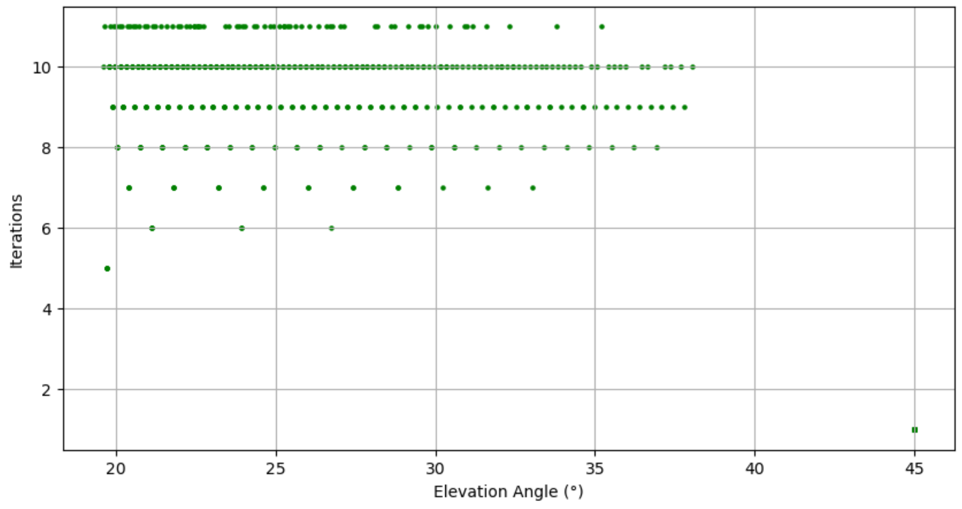

Figure 4.16: Number of iterations required for convergence and the resulting elevation angles with the new model. The number of iterations remains stable at around 10 across all angles.

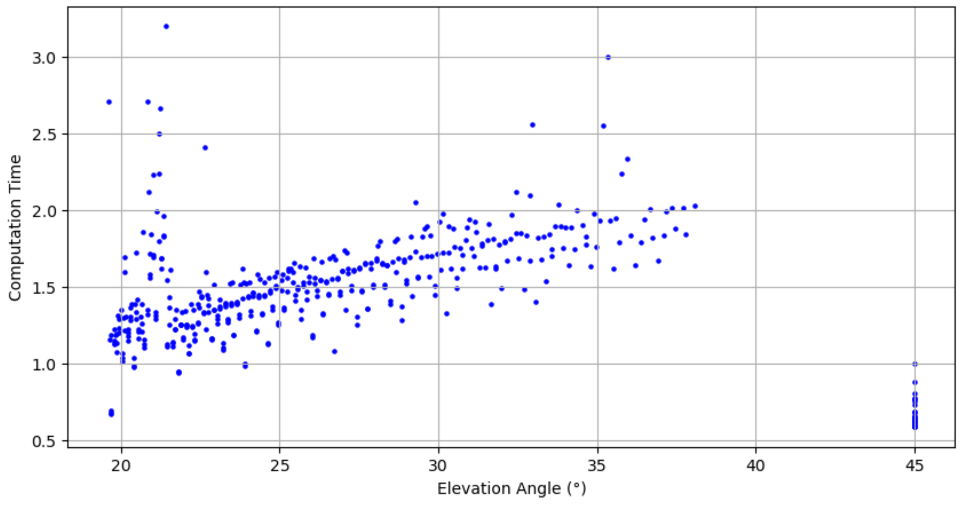

Figure 4.17: Computation times and final elevation angles using the new model. Unlike the previous method, the computation times do not increase exponentially; instead, they appear to grow approximately linearly and remain reasonable.

If instead we repeat the initial test for evaluating the model's accuracy, we observe some different results. By simulating the trajectories with a 10 ms time step, the average error remains roughly the same, 4.5 mm compared to 4 mm with the previous method. However, unlike before, no values exceed the error threshold.

On the other hand, we observe an increase in average computation time, reaching 3.45 ms, almost than double that of the previous method. This is due to the high number of iterations required by the new method, which rarely drops below 8.

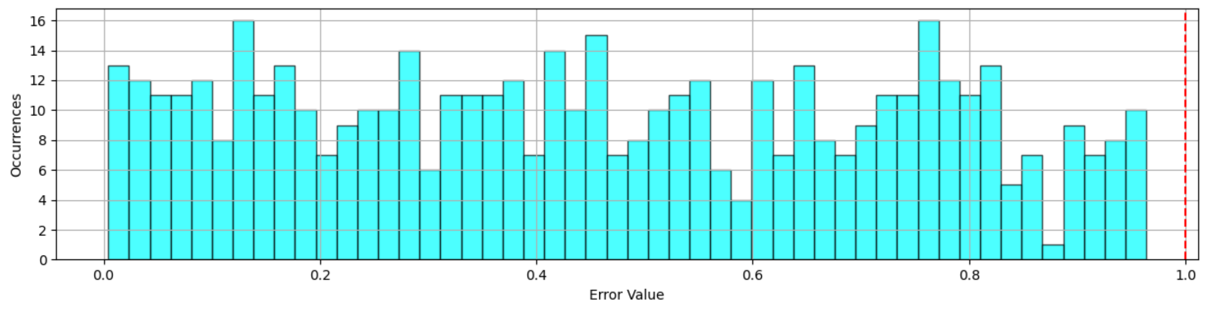

Figure 4.18: Histogram of the errors, using a time-step of 10 ms

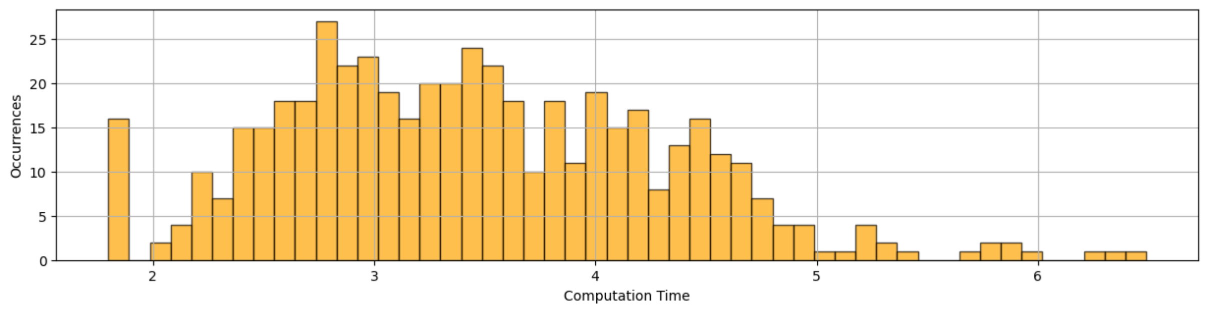

Figure 4.19: Histogram of the computation times, using a time-step of 10 ms

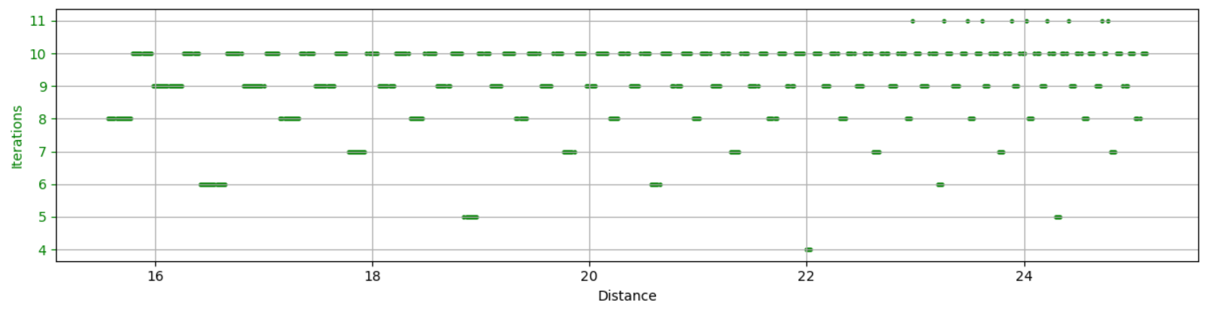

Figure 4.20: Iterations required to converge for shots at different initial distances from the target, using a time-step of 10 ms

The simulation with a 40 ms time step shows similar results in terms of average computation time, which is again roughly double that of the previous method, at around 0.95 ms. The error shows a slight improvement in this case, with an average of 5 mm and fewer occurrences above the threshold compared to the previous method. The number of iterations remains stable, as observed with the 10 ms time step.

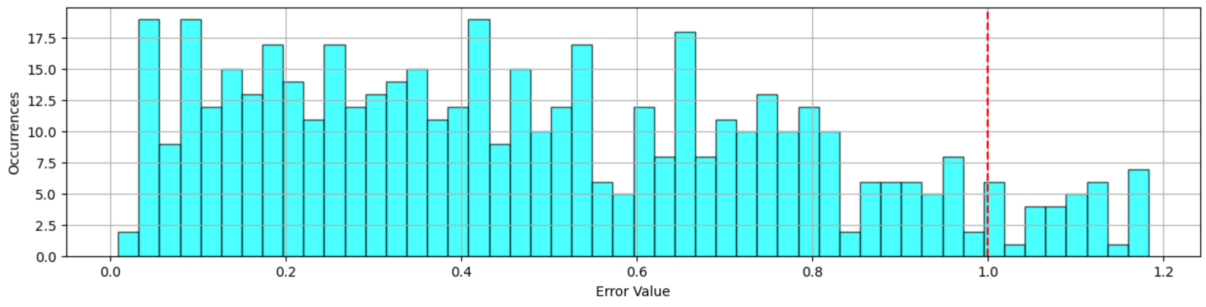

Figure 4.21: Histogram of the errors.

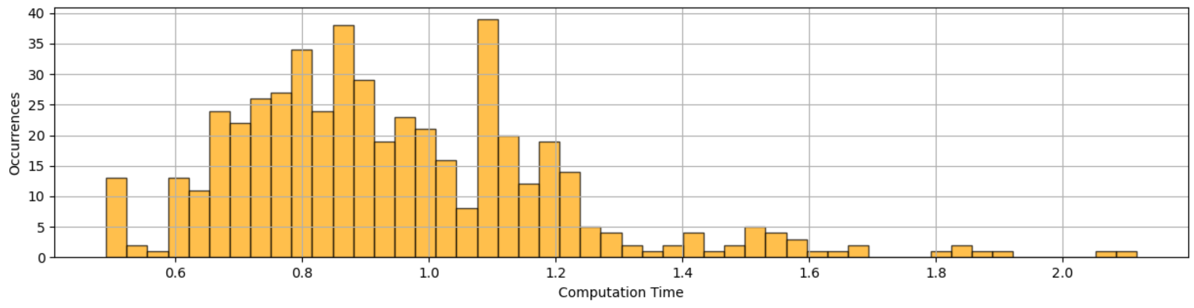

Figure 4.22: Histogram of the computation times.

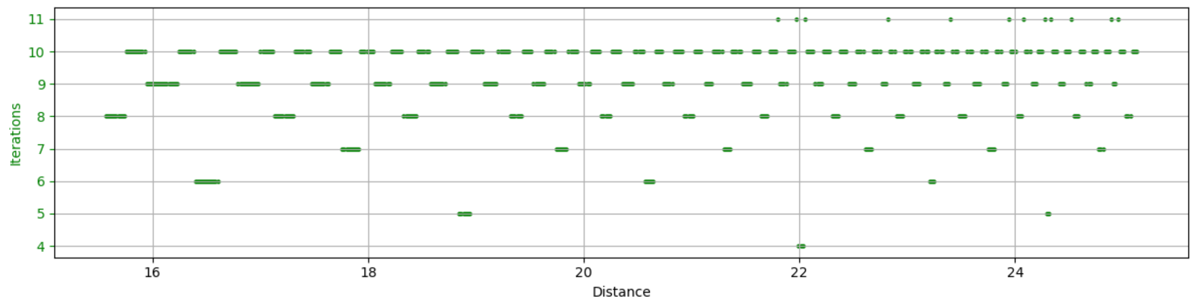

Figure 4.23: Iterations required to converge.

4.3 Two-point targeting

The second predictor implemented in this thesis differs significantly from the first. While the first system estimated both the initial velocity and spin needed to reach a target, this second predictor addresses a more complex task: given only the magnitude of the initial velocity, it aims to find the optimal velocity direction and spin vector that allows the ball to reach a target while passing as close as possible to an intermediate point. This method is particularly designed for simulating free kicks that curve over a defensive wall before reaching the goal.

The problem formulation includes two new unknowns: the horizontal and vertical spin components applied to the ball upon kicking. These are bounded within arbitrary maximum limits:

- $\omega_x \in [-\omega_{x,\max}, \omega_{x,\max}]$
- $\omega_y \in [-\omega_{y,\max}, \omega_{y,\max}]$

Unlike the previous method, this predictor is an original contribution. No reference to similar approaches was found in the literature, and the algorithm was iteratively designed and refined during development.

4.3.1 Ideation

The approach reuses many of the helper functions from the previous predictor, such as `RotateVelocityTowardTarget` and `AlignSpinWithVelocity`. The intuition is that the velocity should initially aim toward the intermediate constraint point—typically positioned between the ball and the final target—while the spin should curve the trajectory to allow the ball to reach the final destination.

The process starts by computing an initial guess with a velocity vector pointing toward the intermediate constraint and no spin. Then, the point at which the guessed trajectory intersects a plane passing through the target is determined. This plane has a normal vector pointing toward the ball's starting position and lies parallel to the ground. The lateral and vertical displacement of the intersection point relative to the target are then used to adjust the horizontal (ω_y) and vertical (ω_x) components of the spin separately.

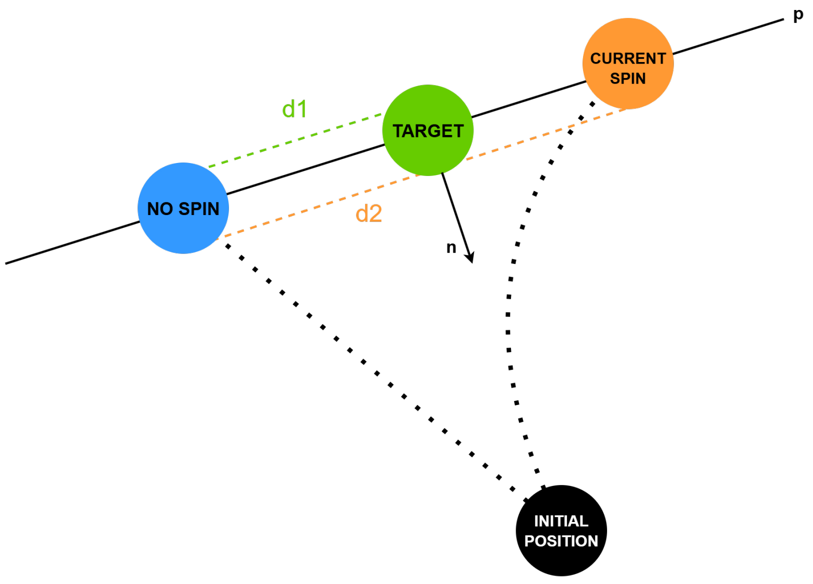

Figure 4.24: The image shows the shots with no spin, the target and the current spin guess, aligned on the plane that contains the target and with the normal vector pointing to the initial position.

At each iteration, two trajectories are compared: one with no spin on the axis that is being adjusted, and another with the current guess of spin. Let p_{noSpin} and $p_{\text{currentSpin}}$ be the closest points to the target from each respective shot. Then we compute:

- $d_1 = \|\vec{p}_{\text{noSpin}} - \vec{p}_{\text{currentSpin}}\|$
- $d_2 = \|\vec{p}_{\text{noSpin}} - \vec{p}_t\|$

The new spin magnitude is calculated using the proportion:

$$\omega_{\text{new}} = \min\left(\omega_{\text{prev}} \cdot \frac{d_1}{d_2}, \omega_{\max}\right) \tag{4.16}$$

After adjusting the spin, the trajectory is re-simulated, and the closest point to the intermediate constraint is recalculated. The velocity is then rotated to aim more precisely toward the constraint.

This adjustment process is first applied to the horizontal spin component, followed by the vertical one. While both spin components influence the trajectory shape, the horizontal component has a stronger effect on curvature, which indirectly impacts the height of the ball at the target due to the longer arc of curved trajectories.

The iterative loop continues until the trajectory satisfies both constraints within a given accuracy threshold. If the required spin to hit the target exceeds the maximum allowed value, the prediction process is stopped early. In this case, the partially adjusted spin and velocity vector are passed to the previously developed single-constraint predictor to ensure the ball still reaches the target accurately.

Although the trajectory may not pass close to the intermediate constraint, this fallback guarantees that the target remains the priority.

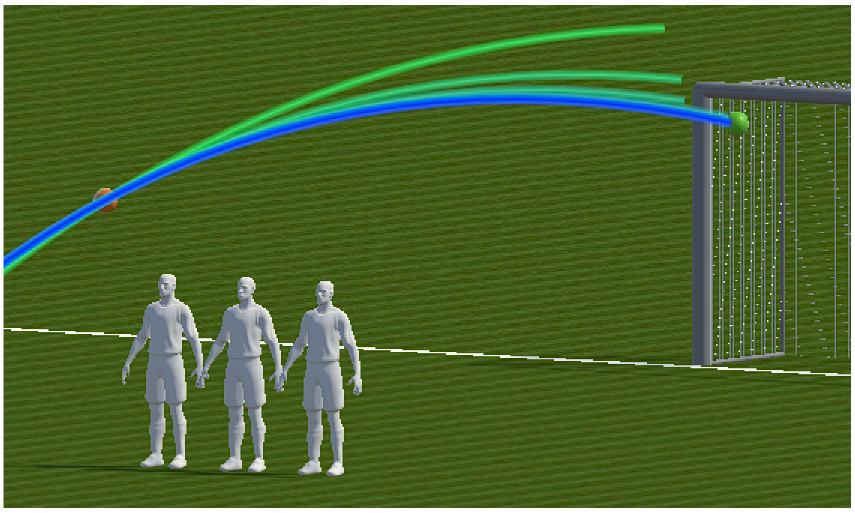
Figure 4.25: The predictor adjusting the trajectories. They pass through the constraint and the spin is iteratively adjusted.

4.3.2 Results

The method was tested to evaluate its overall accuracy. Differently from the tests presented in Section 4.1.3, these tests were conducted by keeping the ball's initial position fixed and varying the initial speed. The goal was to analyze how the required spin and the distances from both the target and the constraint evolved throughout the simulations.

Two scenarios were tested, both starting at about 25 meters from the goal in a central position:

- In the first scenario, the intermediate constraint was placed to the left side of the direct path to the target, simulating a typical free kick that curves around a defensive wall. The final target was located in the top-left corner of the goal.
- In the second scenario, the constraint was positioned in the center of the barrier, just above it, while the target was located in the bottom-left corner of the goal. This setup was used to demonstrate how the method handles descending trajectories.

These tests illustrate how the algorithm adjusts the spin components to satisfy both the target and the constraint. Both algorithms were run with a time-step of 40 ms, using RK2 as integration method. The stopping value for the target error was 1 cm, while for the constraint it was 10 cm. The initial speed was varied from 20 to 40 m/s throughout the simulations, and the maximum spin magnitude was set to 40 rad/s.

First Test

The first test demonstrated high precision with respect to the target, with the errors not exceeding $1.2\,cm$ and an average of $4.5\,mm$.

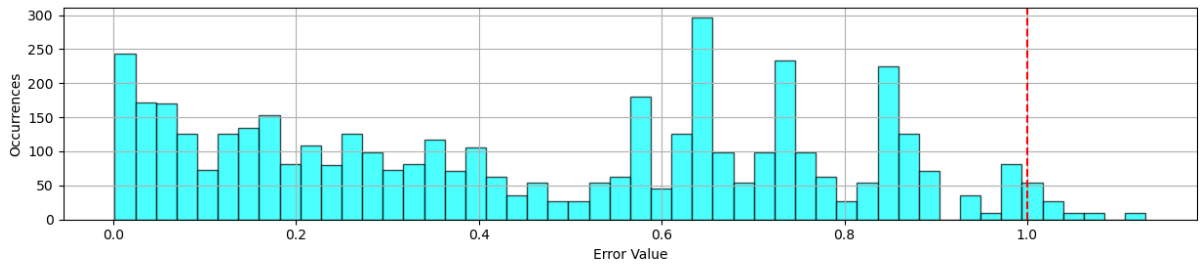

Figure 4.26: Target error in the first test.

The error from the constraint, instead, shows a different behavior. After an initially high error, due to the ball needing to pass above the constraint to reach the target, the trajectory gets closer and closer to it, until it starts moving away again.

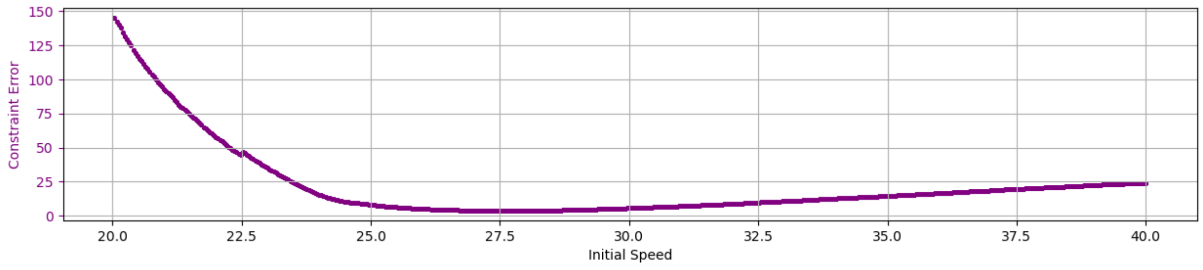

Figure 4.27: Constraint error in the first test.

This behavior is due to the spin reaching the maximum allowed values, and thus no longer being able to adjust the curvature. In this situation, the algorithm prioritizes reaching the target rather than passing near the constraint. This is evident in Figure 4.28, where the horizontal spin component saturates at its limit.

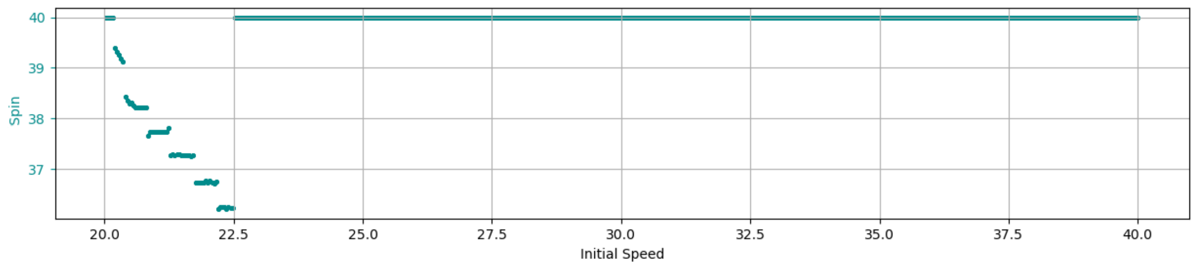

Figure 4.28: Horizontal spin saturation in the first test.

Regarding computation time, this method is slower than the models without constraints, with an average of $4.9\,ms$.

Second Test

The second test, like the first, also showed good accuracy with respect to the target, achieving an average error of $4.7\,mm$.

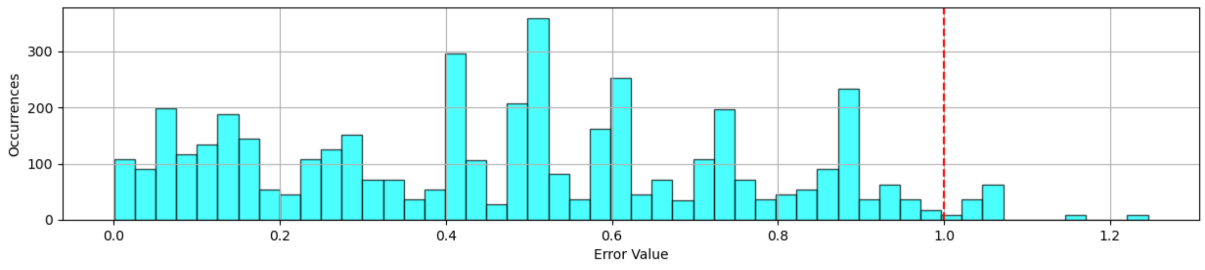

Figure 4.29: Target error in the second test.

The constraint error begins very low, remaining below the stopping threshold. However, once the speed becomes sufficiently high, the vertical spin component is no longer enough to bend the trajectory downward toward the constraint. As a result, the trajectory begins to deviate further from it.

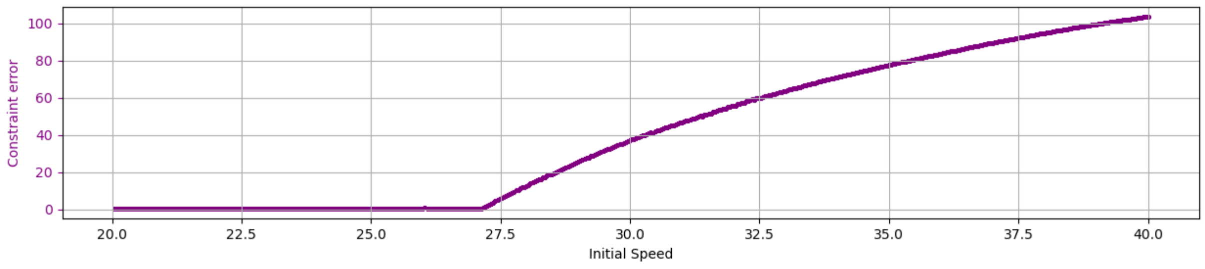

Figure 4.30: Constraint error in the second test.

This is confirmed by observing the evolution of the vertical spin, which gradually decreases until it reaches the minimum limit and remains constant for the rest of the simulation.

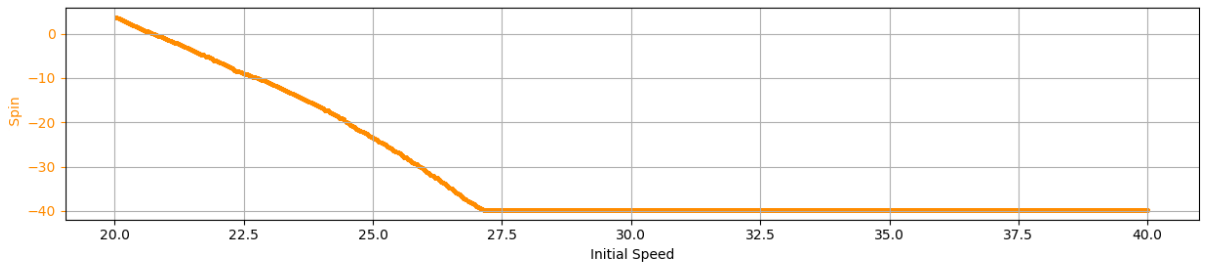

Figure 4.31: Vertical spin saturation in the second test.

Chapter 5

Discussion and Conclusion

This chapter reflects on the results and methods presented in this thesis, discussing the effectiveness and limitations of the simulation system, the integration techniques, and the prediction models. The discussion is structured around the three main components of the project: physics modeling, numerical integration, and trajectory prediction.

5.1 Physics Modeling

The implemented model captures a wide range of physical phenomena, including drag, lift, bounce, and rolling behavior. Unlike many works in the literature that focus solely on in-air dynamics, this model also includes post-bounce motion, spin decay, and ground friction, offering a more complete representation of a football's behavior.

The physics system integrates empirical aerodynamic models derived from experimental data, resulting in visually plausible and convincing ball motion. Crucially, the system is designed with a modular structure that emphasizes flexibility, maintainability, and clarity. Each physical phenomenon is encapsulated into separate components, making the overall simulation easier to understand, extend, and debug.

This modularity also allows developers to selectively enable or disable specific forces or torques and to incorporate simplified or alternative empirical models with minimal modification to the core system. As a result, the system allows the intrinsic properties of the ball to be adjusted, making it adaptable to simulate different types of footballs or other sports balls as well.

In its current state, the only object interaction handled is with the ground. In future work, the system could be extended to include interactions with additional elements such as goalposts. Moreover, a more physically accurate treatment of rolling behavior—particularly involving spin around near-vertical axis, could improve the realism of low-speed dynamics. Another possible enhancement would be to model the ground as a non-uniform surface with small irregularities or bumps, which could introduce slight, unpredictable changes in the ball's motion, thereby increasing the plausibility and visual realism of the simulation.

5.2 Integration Methods

Four integration schemes were tested: Explicit Euler, Semi-Implicit Euler, Runge-Kutta-2, and Runge-Kutta-4. Among these, the Runge-Kutta methods clearly outperformed the others in terms of both accuracy and stability, making them more suitable for simulating ball dynamics with aerodynamic forces.

Both RK2 and RK4 demonstrated strong numerical stability and consistent behavior even at larger time steps. This property made them particularly attractive for use in the shot prediction system, where reducing the number of simulation steps helps lower the overall computation time. While RK4 was the most accurate, its higher computational cost made it less practical for repeated use in real-time prediction routines. In contrast, RK2 provided a very good balance between precision and performance, and was therefore selected as the default method in the second part of the thesis focused on prediction.

Similarly to the physical model, the integration system was designed with modularity in mind. Each integration method is implemented as an interchangeable black-box function, which allows future extensions or testing of alternative schemes with minimal code restructuring. This flexibility makes the system adaptable for more complex or performance-optimized versions in future development.

5.3 Prediction Systems

The two main prediction systems developed, referred to as Method A and Method B, proved to be effective in computing initial parameters to accurately reach a target under varying conditions.

Method A is the faster of the two and demonstrated excellent performance in scenarios involving moderate elevation angles. Due to its speed and responsiveness, it is better suited for dynamic gameplay situations, such as during normal match play, where quick predictions are more important than high trajectory complexity. Moreover, such in-game shots typically do not require high elevation, making Method A a pragmatic and efficient choice.

A valuable insight emerged from the qualitative comparison with the prediction system developed by GATTINI.games. While their system is a commercial implementation written in native C++, the model developed in this thesis showed promising results in terms of both precision and responsiveness. This outcome highlights the potential of the approach and suggests that it could be worthwhile to implement the method in their native environment to further evaluate its performance and integration potential.

In contrast, Method B, which uses a binary search approach to refine the elevation angle, is better suited for static situations like free kicks and corner kicks. In these contexts, the ball is stationary, and slightly longer computation times are acceptable in exchange for more complex trajectories. Despite the higher iteration count and computation time, Method B reliably converges to accurate solutions and adds a layer of flexibility that Method A lacks.

Both methods can still be further optimized. While they currently strike a balance between speed and precision, there is room for improvement in terms of convergence speed and robustness, especially at high angles or near the limits of the simulation model.

Additionally, the prediction system with a second constraint in the middle, which computes both spin and velocity to guide the ball as close as possible to the intermediate constraint point before reaching the final target—, also showed promising results. Although this system was developed without direct reference to existing solutions, its performance was encouraging and suggests strong potential. Nevertheless, it would benefit from further testing to reveal any hidden edge cases or limitations that did not surface during the initial evaluation.

A notable limitation shared across all models is their sensitivity to the initial speed of the ball. If the speed is not sufficient to reach the target, particularly at high elevation angles, none of the methods can fully compensate through spin adjustments alone. As a possible direction for future work, an adaptive speed tuning algorithm could be integrated into the prediction process to refine the initial velocity based on the reachable region for a given elevation and spin configuration.

Another interesting area for further development involves simulating more realistic player interaction. In particular, it would be valuable to incorporate models that estimate initial speed and spin based on the point of contact between the foot and the ball, as explored in the paper *The curve kick of a football I: impact with the foot* [22].

5.4 Conclusion

This thesis has presented the development of a physically plausible simulation and prediction system for football dynamics. It successfully integrated various physical models, explored multiple integration techniques, and implemented novel prediction strategies. The results show that it is possible to achieve accurate, flexible, and real-time-compatible football simulations using numerical methods tailored to game environments.

Bibliography

[1] *Predicting Soccer Ball Target through Dynamic Simulation*, Ying Li, Junxian Meng and Qi Li, 2020

[2] *Simulation and Modeling of Free Kicks in Football Games and Analysis on Assisted Training*, Zhengqiu Zhu, Bin Chen, Sihang Qiu, Rongxiao Wang, Xiaogang Qiu, 2019

[3] *Study of Soccer Ball Flight Trajectory*, Juliana G. Javorova, Anastas Ivanov Ivanov , 2018

[4] *3D Computer Simulator to Examine the Effect of Wind and Altitude on a Soccer Ball Trajectory*, Alexander Egoyan, Ilia Khipashvili, Karlo Moistsrapishvili, 2017

[5] *Soccer Ball Lift Coefficients via Trajectory Analysis*, John Eric Goff, 2010

[6] *An Aerodynamic Analysis of Recent FIFA World Cup Balls*, Adrian L. Kiratidis, Derek B. Leinweber, 2018

[7] *The application of simulation to the understanding of football flight*, Simon Tuplin, Martin Passmore, David Rogers, Andy R Harland, Tim Lucas and Chris Holmes, 2012

[8] *Flight and Bounce of Spinning Sports Balls*, Jacob Emil Mencke, Mirko Salewski, Ole L. Trinhammer, Andreas T. Adler, 2020

[9] *Rolling Motion of a Ball Spinning About a Near-Vertical Axis* , Rod Cross, 2012

[10] *Transition from sliding to rolling in billiards and golf*, Rod Cross, 2021

[11] *Planning the Motion of a Sliding and Rolling Sphere*, Yan-Bin Jia, 2015

[12] *Solving Ordinary Differential Equations I Nonstiff Problems*, E. Hairer, S. P. Nørsett, G. Wanner, 2008

[13] *Stable Solutions Using the Euler Approximation Available to Purchase*, Alan Cromer, 1981

[14] *An Introduction To Numerical Analysis*, Kendall E. Atkinson, 1991

[15] *Geometric numerical integration illustrated by the Störmer-Verlet method*, Ernst Hairer, Christian Lubich, Gerhard Wanner, 2003

[16] *Quaternion Kinematics for the Error-state Kalman Filter*, Joan Solà, 2017

[17] *Quaternions, Interpolation and Animation*, Erik B. Dam, Martin Koch, Martin Lillholm, 1998

[18] *Game Physics Engine Development*, Ian Millington, 2007

[19] *Quaternion - Unity Scripting API*, Unity Technologies, `https://docs.unity3d.com/ScriptReference/Quaternion.html`,

[20] *System.Numerics.Quaternion Struct - .NET API Browser*, Microsoft `https://learn.microsoft.com/en-us/dotnet/api/system.numerics.quaternion`,

[21] *Fixed Timestep - Unity Manual*, Unity Technologies, `https://docs.unity3d.com/6000.1/Documentation/Manual/fixed-updates.html`,

[22] *The curve kick of a football I: impact with the foot*, T. Asai, M.J. Carrè, T. Akatsuka and S.J. Haake