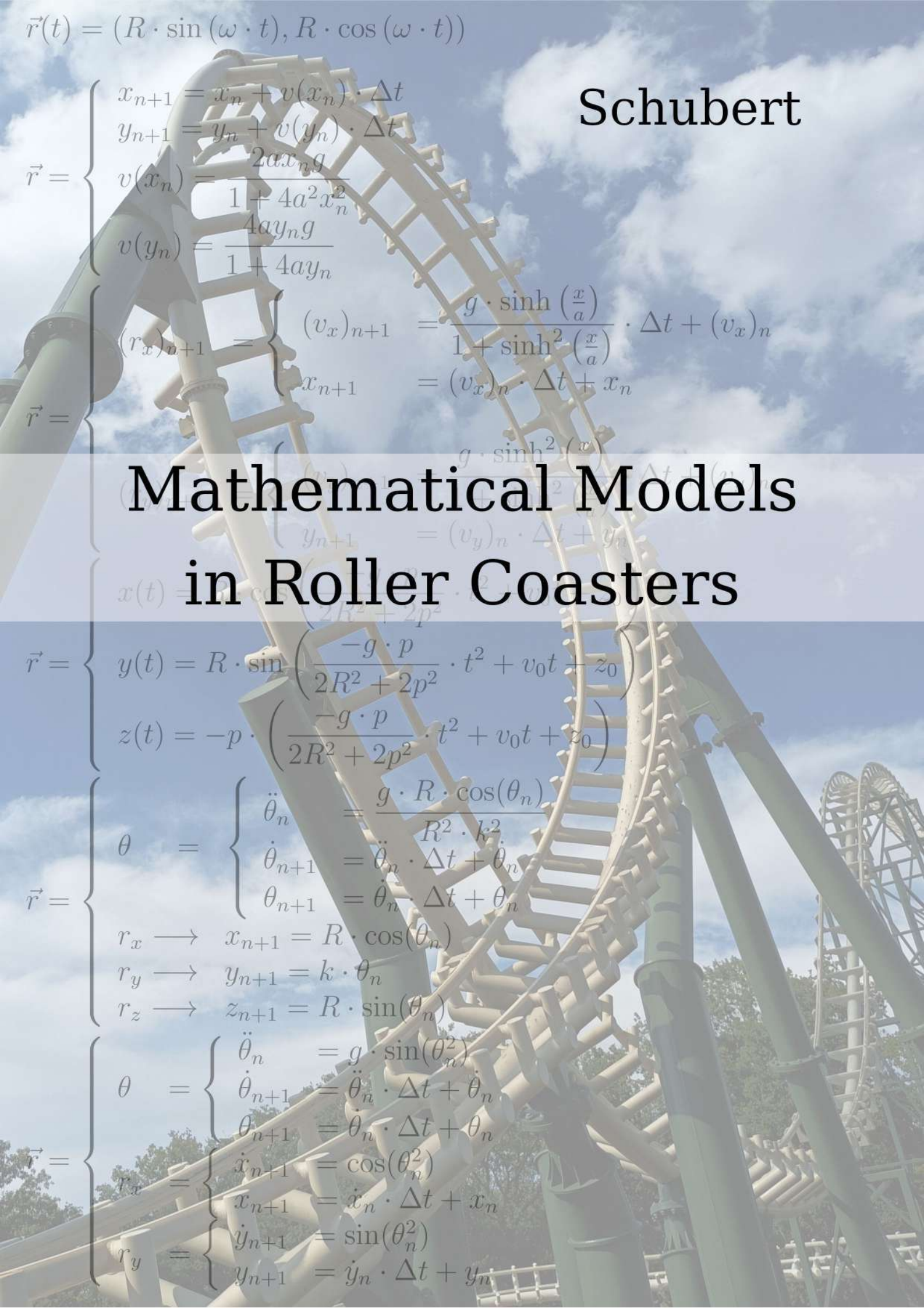


$$\vec{r}(t) = (R \cdot \sin(\omega \cdot t), R \cdot \cos(\omega \cdot t))$$

$$\vec{r} = \begin{cases} x_{n+1} = x_n + v(x_n) \cdot \Delta t \\ y_{n+1} = y_n + v(y_n) \cdot \Delta t \\ v(x_n) = \dfrac{2ax_n g}{1 + 4a^2 x_n^2} \\ v(y_n) = \dfrac{4ay_n g}{1 + 4ay_n} \end{cases}$$

$$\vec{r} = \begin{cases} (r_x)_{n+1} = \begin{cases} (v_x)_{n+1} = \dfrac{g \cdot \sinh\left(\frac{x}{a}\right)}{1 + \sinh^2\left(\frac{x}{a}\right)} \cdot \Delta t + (v_x)_n \\ x_{n+1} = (v_x)_n \cdot \Delta t + x_n \end{cases} \\ (r_y)_{n+1} = \begin{cases} (v_y)_{n+1} = \dfrac{g \cdot \sinh^2(x)}{\cdots} \cdot \Delta t + (v_y)_n \\ y_{n+1} = (v_y)_n \cdot \Delta t + y_n \end{cases} \end{cases}$$

$$\vec{r} = \begin{cases} x(t) = R \cdot \cos\left(\dfrac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0\right) \\ y(t) = R \cdot \sin\left(\dfrac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0\right) \\ z(t) = -p \cdot \left(\dfrac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0\right) \end{cases}$$

$$\vec{r} = \begin{cases} \theta = \begin{cases} \ddot{\theta}_n = \dfrac{g \cdot R \cdot \cos(\theta_n)}{R^2 \cdot k^2} \\ \dot{\theta}_{n+1} = \ddot{\theta}_n \cdot \Delta t + \dot{\theta}_n \\ \theta_{n+1} = \dot{\theta}_n \cdot \Delta t + \theta_n \end{cases} \\ r_x \longrightarrow x_{n+1} = R \cdot \cos(\theta_n) \\ r_y \longrightarrow y_{n+1} = k \cdot \theta_n \\ r_z \longrightarrow z_{n+1} = R \cdot \sin(\theta_n) \end{cases}$$

$$\vec{r} = \begin{cases} \theta = \begin{cases} \ddot{\theta}_n = g \cdot \sin(\theta_n^2) \\ \dot{\theta}_{n+1} = \ddot{\theta}_n \cdot \Delta t + \dot{\theta}_n \\ \theta_{n+1} = \dot{\theta}_n \cdot \Delta t + \theta_n \end{cases} \\ r_x = \begin{cases} \dot{x}_{n+1} = \cos(\theta_n^2) \\ x_{n+1} = \dot{x}_n \cdot \Delta t + x_n \end{cases} \\ r_y = \begin{cases} \dot{y}_{n+1} = \sin(\theta_n^2) \\ y_{n+1} = \dot{y}_n \cdot \Delta t + y_n \end{cases} \end{cases}$$

Mathematical Models in Roller Coasters

Schubert

Abstract

Català

Aquest treball de recerca és un estudi matemàtic dels diferents elements que formen els recorreguts de les muntanyes russes. És habitual que la física sigui el focus principal a l'hora d'analitzar el comportament de les muntanyes russes, i hi ha nombrosa informació sobre el tema. Aquest treball en canvi, té un enfoc matemàtic, tot cercant la difusió de les matemàtiques aplicades com a mitjà d'anàlisi i tenint com a objectiu final el de poder construir animacions de les diferents seccions d'una muntanya russa.

Per dur a terme aquesta tasca, s'introdueixen en primer lloc els conceptes i notació fonamentals que matemàticament es requereixen per aquesta construcció, per passar a continuació a descriure els elements més significatius que pot tenir una muntanya russa. Aquests elements poden ser de dos tipus: els que es poden definir de forma explícita (la corba, la paràbola i la catenària) i els que es defineixen amb equacions paramètriques (l'hèlix, el gir en línia, el bucle vertical, i el tirabuixó). Finalment s'inclouen les animacions programades resultat de les fórmules matemàtiques que descriuen els diferents elements.

També s'inclou un apartat en el que s'analitzen muntanyes russes existents amb dades enregistrades en temps real.

English

This research project is a mathematical study of the different elements which can be found in roller coasters. It is quite common for physics to be the focus in the analysis of the behavior of roller coasters, and there is a lot of information on this topic. In contrast, this research report is centered on the mathematical point of view while trying to popularize the use of applied math as a method of analysis and having as the final objective creating animations of the different sections of roller coasters.

To achieve this, the required mathematical concepts and the fundamental notation are first introduced before moving on to the description of each of the most important elements that a roller coaster can feature. These elements can be classified into two categories: the ones that can be written explicitly (the curve, the parabola, and the catenary) and the ones that must be written in a parameterized form (the helix, the in-line twist, the vertical loop and the corkscrew). Finally, the programmed animations of the mathematical formulas that describe each element are shown.

Additionally, there is a section in which existing roller coasters are analyzed by using data collected recorded in real time.

Acknowledgments

First of all, my deepest thanks to my supervisor [redacted], with whom I have had the great luxury of learning everything I needed for this project. I'd also like to thank my Math Olympiad teacher Josep Grané for spending so much of his time teaching a group of students like me, and for giving me motivation and support at the beginning of this project. Thanks to both and their advice I have been able to reach further in my research. I really can say that both have absolutely motivated me to go towards mathematics.

My sincere thanks to everyone who responded the questions I asked in on-line forums, with special mention to Dr. Lutz Lehmann from Berlin.

I'd also like to show my immense gratitude to all the roller coaster manufacturers who spent some of their time to answer the emails I sent them, especially Paul Lattin from S&S, for his comments on this project, and who even invited me to visit their facilities in the US.

Finally, I'd also like to thank my family, especially my younger brother, who has read my whole project document without complaining. With them, I have travelled across Europe to visit theme parks and roller coasters.

Contents

Page

List of Figures

1 Introduction

Roller coasters are perhaps the most iconic invention that humanity has created to amuse and thrill people. Their evolution along time, from the first Russian mountain slides to the actual mega-coasters we see nowadays, is an amazing example of the science behind the service of leisure.

Modern roller coasters can be really complex, but in the end they can all be divided in small fragments which can be studied individually. In the roller coaster industry, these small fragments are called ***elements***. By joining and evolving elements, new roller coasters can be built, providing all sorts of amusing experiences to the rider.

As the title of this report suggests, I will be using a mathematical point of view all along the project to analyze roller coasters and their elements, with brief inclusions of physical laws. But why math? Math is a crucial concept to the design of roller coasters, but from my point of view it isn't appreciated enough. In other research projects, people opt for using physics to analyze or create their own roller coasters, but I thought it would be more interesting to take on a mathematical and geometrical view. I also thought that it would be interesting to animate the different elements of a roller coaster that I analyzed, and having the mathematical functions that form the shape of these elements makes it much natural, rigorous, and precise.

This research project consists of the mathematical analysis of the different elements that can be used to build roller coasters. Note that in the analysis of the resulting forces, the force of friction is not considered unless it is said otherwise.

As is described later, elements may have several defining characteristics. For example, any element can be either inverting or non-inverting, depending on the vertical tilt angle. Once the tilt angle surpasses 135^{o}, an element is considered an inversion. There are some elements that may even have a double inversion, meaning that riders are inverted twice in a small interval. Some elements can also be divided into subsections of track, so it is considered a different element if there is an intermediate or connecting element to distinguish them, like a curve or a break run. In the case of the elements that are mentioned in this report, they all are different types of curves.

I have divided the elements into two main categories: explicit elements and parameterized elements. The explicit ones are the simplest and are solved in a relatively similar way. These elements only need the x, y and z coordinates to express the full route. Parameterized elements, on the other hand, are used when a curve cannot be written explicitly and are quite more complex and present more difficulties in the way they are solved. This is because a certain value of one of the axes can have more than one point or the curve self-intersects. For these shapes, I will also need the parameter θ to be able to express the full function.

As a general rule, each element analyzed is described using four main sections: an introduction and explanation with a drawing of the corresponding shape, the mathematical equation(s) and their procedures, a small explanation and representation of the resulting forces and finally a link to an animation that shows the element in action in a rendered video.

The final objective of this research project is to be able to animate the different shapes with respect to time. For the final animation, I have prioritized the conservation of energy over following the actual shape.

Stating that there is a lot of generic information on roller coasters already available to the general public through Internet and books, I have opted not to include a historical section. Although they could complement and contextualize this project, they aren't relevant to the topics covered. Despite that, I must admit that much of this information has been used to gain the knowledge to structure the document. So, I have selected some of the most interesting resources I have used and included them in the Bibliography section. Additionally, for some of the elements I have still used a comparison to previous models and ideas to be able to explain the peculiarities of the modern elements in a better and more comprehensive way.

As a side note, real roller coaster designers don't actually use all the math explained here for every single roller coaster. Some computer programs, such as the *NoLimits Roller Coaster Simulator*, allow designers to easily create realistic ride proposals to send to potential customers for a bidding on a project without the burden of calculating all the formulas and forces. As the focus of this report are the calculations themselves, I have not used any simulator as a such and instead, the animations have been built using *Blender 3D*, an open-source rendering software, by using the final formulas calculated. Thus, it is Blender which, through being programmed, solves the mathematical model of the equation of motion.

All the mentioned animations which I have programmed are published in my *YouTube channel* and this document itself can also be reached through the Internet, at my site [redacted], along with a link to the video, as I want to promote the use of math in roller coasters. This is also the reason why this research project is written in English. Also, all the source code is publicly available through GitHub. Thus, I have made all the contents of the project, from the document to the source code, available to other researchers under a CC BY-SA license.

Along the document, some links are included, which lead to the animation on YouTube of the several elements analyzed. These animations have been designed and rendered with Blender 3D using Python (see Appendix III for the corresponding source codes).

This document has been written in the mathematical text format LaTeX, as explained in Appendix I.

2 Notes on Mathematical Concepts and Notation

To make this report more readable, here are some concepts and notations used throughout this document that may not be known to the reader.

- **Main Variables**: First of all, the main variable that I'm going to use is time, represented as t. I'm going to use r to mean the whole vectorial position, which can be divided into the x, y and z components. The first and second derivatives of the position r, are velocity v, and acceleration a respectively. I'm going to use subscript to indicate a certain component of the velocity (v_x, v_y and v_z) or acceleration (a_x, a_y and a_z). For circular movement, I'm going to use their respective Greek letters: φ for angle, ω for angular velocity and α for angular acceleration. On the other hand, I'm going to use the letter g to represent gravity (for this project, $g = 9,8\ m/s^2$ as we are going to consider that we stay relatively close to the surface of the Earth) and m for mass (although I will be assuming that there is no force of friction, so I'm not going to talk much about mass).

- **Secondary Variables**: To represent angles, I won't be using α as it is already used for the angular acceleration and might cause confusion. Instead, I'm going to use β for slope angles, σ for the angle between vectors and θ for rotation angles (from a specified point of view and commonly used as an additional parameter in which the function can be parameterized). Any other parameter will be represented by using a λ. The slope of a curve will be represented using a p (which means that $\tan\beta = p$). Additionally, sometimes one of these variables will be equaled to k, which symbolizes that the variable is an arbitrary known constant, and therefore does not depend on any dynamic variable like position, angle, or time. Lastly, the variable n will be used to denominate an arbitrary step in Euler's method (explained below) by using $n + 1$ as the following step.

- **Energy**: In physics, the conservation of energy principle states that the total energy of a system always remains constant[1]. The conservation of energy is only true for conservative forces, which are the ones in which the path followed from one point to another doesn't affect the total mechanical energy. In this project, we are going to talk about three main types of energy: kinetic energy, potential energy and mechanical energy. Kinetic energy is the energy an object has when moving. The formula for the kinetic energy is $E_c = 1/2 \cdot m \cdot v^2$. As we can see, the kinetic energy depends on the velocity.

[1] This system has to be isolate, as there can't be energy entering or exiting the system to preserve this principle.

Potential energy is the energy an object has because of its location. Its formula (assuming a constant gravitational field) is: $E_p = m \cdot g \cdot h$. Note that the potential energy depends only on the height. Mechanical energy is the total energy (the sum of potential and kinetic energy): $E_M = E_c + E_p$.

- **Forces**: There are six main types of forces: weight, normal force, force of friction, net force, the centripetal acceleration, and any external forces.
  - The **weight** (represented using W) is produced by a gravitational force, usually from a planet and does therefore not require any contact between the two objects. This force is constant as long as we are staying at the same distance from the planet. This force varies slightly with the position of the object, and it is produced by both objects simultaneously, but as we are going to consider that one of the objects is planet Earth, the force produced by the object attracting the planet gets neglected. The weight is represented using the formula: $W = m \cdot g$.
  - The **normal**[2] force is the force generated by a surface holding an object. Therefore, the normal force counteracts any other force applied on the object (usually the weight) so that the acceleration in that axis is 0.
  - The **force of friction** is the resistance that a surface presents for moving an object. Therefore, it is always opposite to the direction of movement. There are two types of friction forces: static forces, which stop an object from gaining velocity, and dynamic forces, which stop the object once it has a certain velocity[3].
  - The **net force** is the vectorial sum of total forces: $F_{net} = \sum F$. It represents the resulting force on an object, as it takes into account the direction of each force, so that any opposite component of a pair of forces gets canceled out.
  - The **centripetal force** is the force used to keep an object in a circular trajectory. Although it is technically only the consequence of another force, the centripetal force can be calculated by using all the forces that go towards the circle. The centripetal acceleration is therefore always perpendicular to the track.
  - The rest of the forces are classified as **external forces** because they come from external sources such as a magnetic field (magnetic force), a motor (mechanical force) or a rope (tension). In this project, we will

[2] ↩In mathematics, the word normal is a synonym for perpendicular.

[3] ↩This force requires for the two objects to be in contact, but air also counts and produces resistance.

not consider any external forces acting on an object, as this does not happen in real roller coaster (except in the initial launches or lift hills).

- When talking about forces, they can easily be divided into two components: the vertical and the horizontal, but that's not always best way to do so. Sometimes, we are going to use the *tangential* (which is tangent to the curve: $F_\top$) and sometimes the *normal* (which is perpendicular to the curve $F_\perp$). When there is more than one force involved (which is usually the case), we will use the previously mentioned concept of *net force* to imply the vector sum of forces.

- It is important to remember the difference between *mass* and *weight*. When talking about the mass of an object (measured in kg), we are referring to the physical property of the object produced by its molecules. Weight is used to talk about the attractive force produced by two objects with mass. The connection between mass and weight is therefore: $\vec{W} = m \cdot \vec{g}$.

- There is another important distinction to be recalled regarding the terms *speed* and *velocity*. Firstly, speed is the time rate at which an object moves along a path, while velocity is the rate and direction of the movement of an object. Therefore, speed is a scalar value (in m/s), while velocity is a vector. So, we can say that speed represents the modulus of the velocity. When talking about acceleration, we will always be referring to the vector unless we explicitly say that we have the modulus of the acceleration.

- **Ordinary derivatives**: There are three ways in which we can represent derivatives. Newton's notation states that the symbol $\dot{y}$ is used to symbolize the derivative of the function (in this case y), but only with respect to time. If the derivative is not with respect to time, we can use one of the other notations. Leibniz's notation is written as $\frac{df(x)}{dx}$ (i.e., derivation of the function $f(x)$ respect to the variable x) and it is used mainly in physics and other applied mathematics. However, the most common notation for derivatives in theoretical mathematics is the use of Lagrange's notation, which uses primes to indicate the derivative: $f'(x)$ (this would represent the derivative of the function $f(x)$ with respect to x, which is the variable commonly used in equations). This way, $\dot{y}$ is the equivalent form of $\frac{dy}{dt}$ and $y'(t)$. The number of dots, the exponent or the number of primes indicate the order of the derivative. So, the third derivative with respect to time of the function y could be represented as: $\dddot{y} = \frac{d^3y}{dt^3} = y'''(t)$.

- **Partial derivatives**: Partial derivatives are used in some branches of mathematics. They are quite common, and are used when there are several independent variables, but we just want to integrate with respect to one of them. Partial derivatives are written using the following notation: $\frac{\partial y}{\partial x}$, which is similar to Leibniz's notation. Despite this, the $\frac{d}{dx}$ notation is also used to represent partial derivatives depending on the context.
- To avoid confusion, the symbol $|x|$ is used as the absolute value of the number, and $\|\vec{x}\|$ to denominate the modulus (length) of a vector.
- **Vectors**: It is important to spot another couple of different symbols used while talking about vectors. Mainly, $\vec{u}$ is used to represent the vector using the components, u_x and u_y. To represent the modulus of the vector, $\|\vec{u}\|$ is used. Therefore, according to the Pythagorean theorem, $\|\vec{u}\| = \sqrt{u_x^2 + u_y^2}$. Additionally, the unitary vector[4] of $\vec{u}$ is represented as $\hat{u}$, and it is calculated using $\hat{u} = \frac{\vec{u}}{\|\vec{u}\|} = \left(\frac{u_x}{\|\vec{u}\|}, \frac{u_y}{\|\vec{u}\|}\right)$. It is important to note that a variable may or may not have the vector above it, and it is considered a different variable depending on it, for example p and $\vec{p}$[5].
- **Multiplication of Vectors**: There are two main ways in which vectors can be multiplied: scalar product or vector product. When talking about a **scalar product**, the result of the multiplication is a scalar value, and has therefore no directorial dimensions. The formula for the scalar product is (where σ is the angle between the two vectors, $\vec{v}$ and $\vec{w}$):

  $$\vec{v} \cdot \vec{w} = \|\vec{v}\| \cdot \|\vec{w}\| \cdot \cos(\sigma)$$

  On the other hand, a **vector product** does result in another vector, and it can be calculated using the formula (where σ is the angle between the two vectors, $\vec{v}$ and $\vec{w}$, and $\hat{u}$ is a unit vector perpendicular to both $\vec{v}$ and $\vec{w}$ in an additional dimension):

  $$\vec{v} \times \vec{w} = \|\vec{v}\| \cdot \|\vec{w}\| \cdot \sin(\sigma) \cdot \hat{u}$$

- Lastly, we are going to consider that the object stays affixed to the shape, like in actual roller coasters. This means that we are going to try to fit the mathematics to the model, which will give us a clear result when animating the shape.

[4] ↩A unitary vector is a vector that derives from another one but has modulus 1.

[5] ↩This is usually avoided by using other variables, but it is sometimes necessary. In the few cases that this may happen, the two values will probably have a relation in meaning.

Types of Equations

The following are the types of equations that will be mentioned in this project:

- **Algebraic equation**: The most common type of equations, in which the unknown variable or variables is present using only elementary functions. The degree of the equation is known by the highest exponent of the unknown variable. These are usually solved by finding a way to isolate the unknown variable.
- **Differential equation**: In a differential equation, the unknown function appears in conjunction with its derivatives. The order of this equation is determined by the highest derivative. These are more difficult to solve than the algebraic equations, as sometimes only the type of function that would solve the function itself can be figured out and additional values or conditions to find the final explicit equation may be necessary to be known.
- **Explicit equations**: In an explicit function, the output variable (dependent variable) can only be expressed directly in terms of the input variable (independent variable). Dependent and independent variables are commonly found in an explicit function. The values of the variables in an explicit function are easier to determine because of its more straightforward expression. On the contrary, an implicit function is one that cannot be expressed as one variable in terms of another variable.
- **Parametric equations**: A parametric equation is a group of functions of several independent variables called parameters that make up a curve or surface. Usually, all these parameters depend on a common variable. These type of equations are commonly used as coordinates (x, y). A lot of functions can be parameterized, but only some parametric equations can be transformed into simple functions. Almost all explicit functions (which are defined as: $y = f(x)$) can be parameterized easily by using a change of variable:
$$\begin{cases} x &= t \\ y &= f(t) \end{cases}$$
- **Discretized equation**: In this type of equation, variables are represented as a function with respect to the previous value. Because of this, it is necessary to set the initial conditions. In this report, they will be used as an alternative for finding the solution to a differential equation. The main downside of this type of equation is that only the value at a specific point can be calculated by using small increments from the start up to that value. The smaller the increments, the more precise the solution will be, as we get closer to an actual differential.

Equations can also have the following properties:

- **Order and degree of an equation**: Although they are similar, order and degree do not mean the same. On one hand, order is used to express the highest derivative and only makes sense in differential equations (as algebraic equations always have order 0). On the other hand, degree represents the highest power of an equation. The degree of an expression is mainly used in algebraic equations but can also be used in differential equations (differential equations with a degree higher than 1 tend to be harder to solve). There are some functions that receive a special classification according to their degree: linear equations are degree 1, quadratic equations have degree 2, cubic are degree 3, and so on.

- **Linearity**: A linear algebraic equation is the one that draws a line. Its formula is $y = m \cdot x + n$, where m and n are constants. In a linear equation, the variable is only multiplied by a constant value. Similarly, in differential equations each derivative is only multiplied by a constant value and isn't raised to a power. Generally, linear equations are solvable with relative ease, but most non-linear equations usually cannot be solved exactly and are the subject of much ongoing research. However, in some cases non-linear equations can be solved in specific ways, for example reducing the order of the equation or numerically discretizing the equation (like in initial value problems).

- **Ordinality**: An ordinary equation has a finite, also called discrete, set of variables. A partial equation has an unknown or infinite number of variables. To solve these differential equations, we approximate the partial differential equation into an ordinary differential equation.

- **Homogeneity**: An equation is considered homogeneous if the variable in which the function is derived appears by itself, without multiplying or dividing the function. This classification isn't used as often, as very few equations have specific solutions for being homogeneous. An equation cannot be linear and homogeneous at the same time, as the function or one of its derivatives would be multiplied by a number that isn't constant.

Newton's Laws of movement

First Law: Inertia

"An object at rest remains at rest, and an object in motion remains in motion at constant speed and in a straight line unless acted on by an unbalanced force."

Newton's first law states that every object will remain at rest or in uniform motion in a straight line unless compelled to change its state by the action of an external force which doesn't get balanced out. Balancing out the net force of an object is quite easy (all objects that are not moving are balanced out). This tendency to resist changes in a state of motion is inertia and is represented by the force of friction. If all the external forces cancel each other out, then there is no net force acting on the object. And if there is no net force acting on the object, then the object will maintain a constant velocity. [1]

This law of motion is essentially only theoretical, as there is always an unbalanced force acting on every object, usually the force of friction. As we are going to assume that there is no force of friction in this project, it is sometimes possible to apply this law, at least to one of the axes (usually the horizontal one, when the weight and the normal force cancel out).

Second Law: Force

"The acceleration of an object depends on the mass of the object and the amount of force applied."

Newton's second law defines that the sum of forces is equal to change in momentum[6] per change in time. This can be represented as:

$$\sum \vec{F} = \frac{d\vec{p}}{dt} = \frac{\vec{p}_1 - \vec{p}_0}{t_1 - t_0} = \frac{m \cdot \vec{v}_1 - m \cdot \vec{v}_0}{t_1 - t_0}$$

As we know that $\vec{a} = \dfrac{\vec{v}_1 - \vec{v}_0}{t_1 - t_0}$, we can rewrite the expression above as[7]:

$$\sum \vec{F} = m \cdot \vec{a}$$

This equation proves that an object subjected to a force will accelerate proportionally to the value of that force. It also tells us that the object will accelerate inversely proportional to the mass, as heavier object will experience less acceleration than a lighter object when applied under the same net force. This also means that there can't be an acceleration without an external force causing it.

[6] ↩Momentum, $\vec{p}$, is defined to be the mass m of an object times its velocity v: $\vec{p} = m \cdot \vec{v}$.

[7] ↩We are considering that the mass of an object is constant, otherwise this simplification would be impossible.

Newton's Second Law also proves his First Law, as if the net force is zero, the acceleration has to be zero, considering that every object has mass.

It is important to note that velocity, force, acceleration, and momentum are vector quantities, and have magnitude and direction.

Third Law: Action and Reaction

"Whenever one object exerts a force on another object, the second object exerts an equal and opposite on the first."

When an object receives a force from another object, it produces the same force on that object. For example, the gravitational pull that the earth exerts on an object is the same force that this object produces on the earth. This means that every force comes from the interaction between two objects, which can be either attractive or repulsive.

For objects to move freely, the Second Law show us that the acceleration that results from those forces is proportional to the mass of the object, which means that each object will accelerate differently.

This law justifies the normal force produced by an object holding another object and proves that the normal force is always perpendicular, as the reactive force produced to hold the object always points at it.

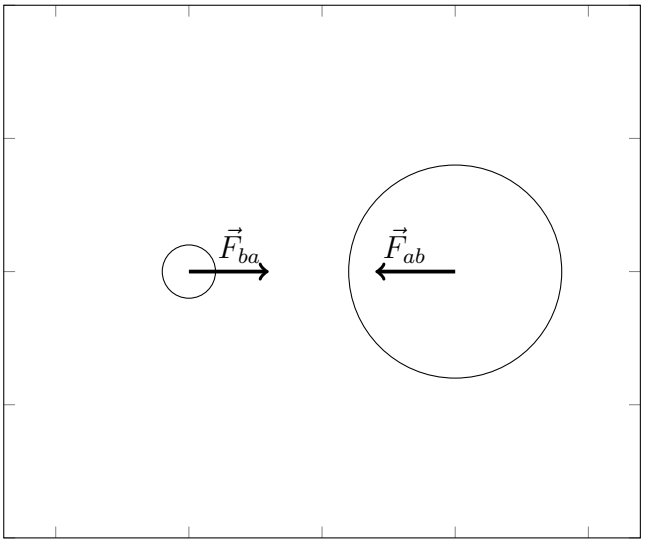


Figure 1: Chart displaying Newton's Third Law

3 General Method for Finding The Law of Motion

As a general rule, the first steps to obtain the formula that describes an object's motion with respect to time are always focused on finding a differential equation to solve. As in all mechanical systems, the expression of motion comes from the acceleration which is projected into the shape of the element through the forces and restrictions involved. In roller coasters, we know that the force of the track against the object, the normal force, is perpendicular to the curve itself, so we can calculate the total acceleration by canceling it out with the normal component of the gravitational force. In the chart below, we can see a representation of a function with a tangent vector showing the vertical and horizontal components. The angle between the tangential line and the horizontal axis is represented as the β angle.

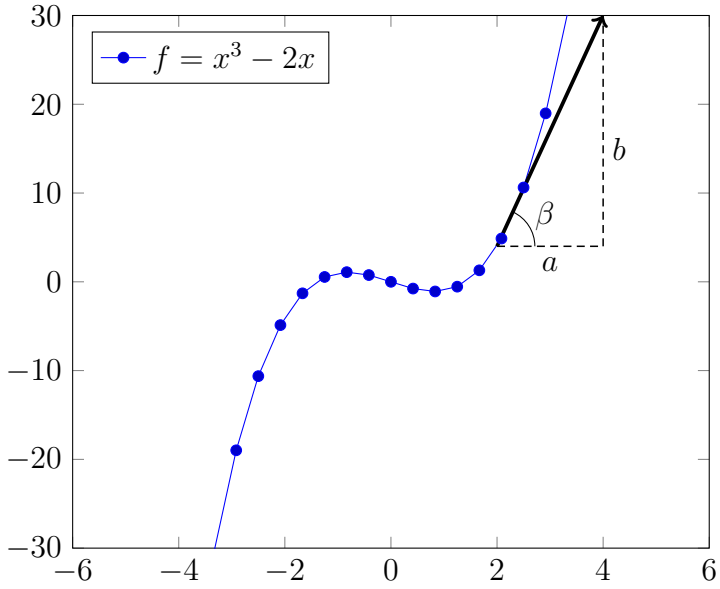


Figure 2: Representation of the components of the slope vector

This slope can be calculated using either the angle or the derivative: $p = \frac{b}{a} = \tan(\beta) = f'(x)$. The slope will be used to select the component of the gravitational force which gets projected onto the element.

There are three ways in which we will be able to obtain the resulting total acceleration, which are: the **trigonometric projection**, the **vectorial projection with respect to a variable** and the **vectorial projection with respect to a parameter**. The trigonometric projection allows us to use the angle of the slope to calculate the effect of gravity on the object. In contrast, the vectorial projection uses vectors to do the same. We are going to decide on one or the other depending

on which variables we have and which of those stay constant along the trajectory, and also on which variables change with time.

The most important variable we must consider is the slope at each point, as the trigonometric projection method is mostly useful when the angle of the slope remains constant. This is because in our formulas, the slope of the curve will be used, and we will therefore have the angle of the slope inside a trigonometric function, which will not be solved easily.

Trigonometric projection

Firstly, we are going to project the gravitational pull to the shape of the element so that all the other forces get canceled out. This can be done using the cosine function of the angle as follows:

$$\|\vec{a}_T\| = g \cdot \cos\theta$$

The vertical acceleration can then be calculated exacly the same way, as we can now use the total tangential acceleration and the slope angle to find its vertical component.

$$\|\vec{a}_Z\| = \|\vec{a}_T\| \cdot \cos\theta = g \cdot \cos^2\theta$$

Vectorial projection with respect to a variable

Another method is to project the gravitational pull using vectors. These avoid using the slope of the curve by using the derivative function. To do so, we define two vectors: one for the full vertical gravitational force, which is $\vec{g} = (0, g)$ and one for the tangent, which is $\vec{u} = (1, f'(x))$ for any point $(x, f(x))$.

Now we are going to use the scalar product, which is defined as:

$$\vec{u} \cdot \vec{g} = \|\vec{u}\| \cdot \|\vec{g}\| \cdot \cos\sigma$$

By using a unit vector for $\vec{u}$, the formula becomes:

$$\hat{u} \cdot \vec{g} = \|\vec{g}\| \cdot \cos(\sigma)$$

Notice how $\|\vec{g}\| \cdot \cos(\sigma)$ is the decomposition of $\vec{g}$ in the direction of $\vec{u}$. This gives us the tangent component of the gravity that doesn't cancel out with the normal force. If we had that $\|\vec{u}\| = 1$, the scalar product of the two vectors would equal the projection of $\vec{g}$ to the tangent component of the parabola. To get the unit vector, we divide the vector by its modulus: $\hat{u} = \dfrac{\vec{u}}{\|\vec{u}\|}$

Next, we are going to transform the $\vec{u}$ vector into a unitary vector using the corresponding formulas:

$$\hat{u} = \frac{\vec{u}}{\|\vec{u}\|} = \frac{(1, f'(x))}{\sqrt{1 + f'^2(x)}}$$

This unitary vector is always tangent to the curve, which will allow us to find the vector that is tangent to the shape but has the modulus of the tangent acceleration produced by gravity. The tangential acceleration indicates how much the velocity changes at each interval, and it is the direct representation of the gravitation force with respect to the slope at that point.

$$\|\vec{g}\| \cdot \cos(\sigma) = \hat{u} \cdot \vec{g} = \frac{(1, f'(x))}{\sqrt{1 + f'^2(x)}} \cdot (0, g) = \frac{g \cdot f'(x)}{\sqrt{1 + f'^2(x)}}$$

Now that we have the modulus of the acceleration, we are going to multiply it by the unitary vector again to obtain the total vector that defines the acceleration:

$$\vec{a} = \underbrace{\frac{g \cdot f'(x)}{\sqrt{1 + f'^2(x)}}}_{\text{Modulus of the acceleration}} \cdot \underbrace{\frac{(1, f'(x))}{\sqrt{1 + f'^2(x)}}}_{\hat{u}} = \left(\frac{g \cdot f'(x)}{1 + f'^2(x)}, \frac{g \cdot f'^2(x)}{1 + f'^2(x)} \right) = (\vec{a}_x, \vec{a}_y)$$

Vectorial projection with a parameter

Similarly to the vectorial projection for explicit curves, the objective this time is to use the unit vector to project the gravity towards direction of movement. Now, the unit vector can be found using the following formula (for this explanation, I'm going to assume that we have a three-dimensional curve defined by the parameter θ):

$$\hat{u} = \frac{\vec{u}}{\|\vec{u}\|} = \frac{(x'(\theta), y'(\theta), z'(\theta))}{\sqrt{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}}$$

$$\|\vec{g}\| \cdot \cos(\sigma) = \vec{g} \cdot \hat{u} = (0, 0, g) \cdot \frac{(x'(\theta), y'(\theta), z'(\theta))}{\sqrt{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}} = \frac{g \cdot z'(\theta)}{\sqrt{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}}$$

$$\vec{a} = \underbrace{\frac{g \cdot z'(\theta)}{\sqrt{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}}}_{\text{Modulus of the acceleration}} \cdot \underbrace{\frac{(x'(\theta), y'(\theta), z'(\theta))}{\sqrt{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}}}_{\hat{u}}$$

$$= \left(\frac{g \cdot z'(\theta) \cdot x'(\theta)}{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}, \frac{g \cdot z'(\theta) \cdot y'(\theta)}{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)}, \frac{g \cdot z'^2(\theta)}{x'^2(\theta) + y'^2(\theta) + z'^2(\theta)} \right)$$

3.1 The Differential Equation

These three methods have now given us an expression to find the acceleration. Using this acceleration, we can now find the position at any point in time. As we surely don't have a formula with respect to time which we could derive, we have another intermediate parameter (usually an angle of rotation). If we can form a relation between this parameter and time, we can use this in our original equation to get the final formula.

On one side of the differential equation, there is usually the acceleration, expanded using the chain rule and the product rule, and on the other side of the differential equation, there is usually the acceleration found using one of the methods above.

Although these methods can help us solve some of the elements, there are some others that require a small step further to solve the resulting differential equation. This is mainly the case of the shapes in which there are two different values for a certain point in the x, y or z axis. Another problem these shapes might have is that the slope of the curve is completely vertical and therefore it is not derivable. The quickest workaround for both problems is to introduce a new variable, such as the arc length or the angle of rotation from a certain point of view, and finding an equation for each axis with respect to that new variable. This should produce a new differential equation that can be solved using one of the previous methods.

We know that the acceleration is the second derivative of the position. If we can equal the tangential component of the gravity to the second derivative of the position, we might find a differential equation which we can solve. We know that our position isn't with respect to time, so we won't be able to find $\ddot{r}$ easily. Including a parameter in the derivative is quite easy with the chain rule (explained below).

We would now equal the two expressions we have and then obtain a differential equation. In case of having a higher derivative, we would have to use the product rule, which would result in a bigger expression which probably cannot be solved as easily. For some differential equations, it is not necessary to solve them analytically. Instead, we can solve them numerically using some theorems like Euler's method or the midpoint method.

3.2 Euler's Method

Euler's method is a procedure designed for solving ordinary differential equations using small splines[8]. As this method is based on the polygonal approximation of a certain curve at small increments, the precision of this method is inversely proportional to the number of segments, and we should therefore try to make the reduce the increments as much as possible.

This method allows solving first-order differential equations numerically. For a higher order, this method can be used twice at each step. Because of this, the precision of the shape is going to decrease exponentially over time, as differential equations are usually going to be second-degree ODE (equations which include the second derivative of the function).

Euler's method is quite simple, but we need some specific conditions. Firstly, we need to have an expression for the first derivative of the function. This expression is therefore an ordinary differential equation and should be expressed as a formula which uses only constant values and the variable of the function itself (which can't be in a derivative form). In our case, the expression we have won't depend on time (otherwise we would simply integrate the derivatives of the function until we get our formula with respect to time), but on other parameters, such as the position x.

As the shape satisfies the given differential equation, we can think of it as a formula by which the slope of the tangent line to the shape can be calculated at any point by knowing only the value of the function at that point. Sometimes, the value of a derivative is also necessary, but that can also be calculated by using Euler's method as long as the degree of this derivative is strictly lower than the degree of the differential equation itself. As the derivative shows us the rate of change of a certain function, we can use small increments to determine the direction of the function at that moment, and therefore, its position. Considering that Δx is the step size, also represented as the change in the variable x of the function, from n to $n+1$: $\Delta x = x_{n+1} - x_n$, we can express Euler's method as:

$$\boxed{f(x)_{n+1} = f'(x)_n \cdot \Delta x + f(x)_n}$$

As each position is calculated from the previous one, we need to have the initial values of position and velocity for this method to work. This is why Euler's method is considered an initial value problem, along with similar structured theorems.

In the case where we have a formula for the second derivative of a function, we can still use the same formula, but this time we will obtain the first derivative at each instant. Using Euler's method again with the value of the first derivative, we can then obtain the solution of the function.

[8] ↩ Splines is the segmentation of a curve using straight lines.

Therefore, Euler's method allows to obtain a system of discrete equations, which result in an expression for the result of the function at a certain moment by knowing the value of the derivative and the function of the previous step like this:

$$\begin{cases} f(x)_{n+1} = f'(x)_n \cdot \Delta x + f(x)_n \\ f'(x)_{n+1} = f''(x)_n \cdot \Delta x + f'(x)_n \end{cases}$$

As we are going to be using time, velocity, and acceleration, we can transform the formula into:

$$\vec{r} = \begin{cases} x_{n+1} = v_n \cdot \Delta t + x_n \\ v_{n+1} = a_n \cdot \Delta t + v_n \end{cases}$$

For parametric equations, we also use Euler's formula more than once, one for each component. Sometimes we can establish a relation between the two independent variables, but we are going to use both components of the parametric equation because we will prioritize the conservation of energy over following the actual shape.

3.3 General Rules

Chain rule

The states that if we have a composition of functions, the derivative can be calculated by using the following formula (considering the function $x(y(t))$):

$$\frac{dx}{dt} = \frac{dx}{dy} \cdot \frac{dy}{dt}$$

Integration by parts

This rule is used to solve some integrals which have a specific format. If we have two functions, $f(x)$ and $g(x)$, we can write the following formula:

$$\int f(x)g'(x)\ dx = f(x)g(x) - \int f'(x)g(x)\ dx$$

Leibniz product rule

The product rule states that we can calculate the derivative of a product of functions by using the following formula:

$$\frac{d}{dx}(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

In our case, we want to derivate a function with respect to time. For example, if we have the function $y(\theta)$ and we derivate it with respect to time once by using the chain rule, we get:

$$\dot{y}(\theta) = y'(\theta) \cdot \dot{\theta}(t)$$

Now that we have a multiplication of independent functions, if we want the second derivative with respect to time of that function, we are going to find the derivative using the product rule:

$$\ddot{y}(t) = \frac{d}{dx}\left(y'(\theta) \cdot \dot{\theta}(t)\right) = y''(\theta) \cdot \dot{\theta}^2(t) + y'(\theta) \cdot \ddot{\theta}(t)$$

Leibniz integral rule

In calculus, the Leibniz integral rule is a method that allows us to differentiate any integral of the form:

$$\int_{a(x)}^{b(x)} f(x, \lambda)\ d\lambda$$

The formula for Leibniz's integral rule states the following:

$$\begin{aligned}\frac{d}{dx}\left(\int_{a(x)}^{b(x)} f(x,\lambda)\ d\lambda\right) = \quad & f(x,b(x))\cdot\frac{d}{dx}b(x) \\ & -f(x,a(x))\cdot\frac{d}{dx}a(x) \\ & +\int_{a(x)}^{b(x)}\frac{\partial}{\partial x}f(x,\lambda)\ d\lambda\end{aligned}$$

Frequently, some components of this sum will get canceled, because either $a(x)$, $b(x)$ or $f(x,\lambda)$ is constant, and it's derivative is therefore 0.

4 Explicit Elements

4.1 Curve

The basic curve is an element that changes the direction of the track. It is produced by a force pulling the rider towards the center of the curvature, as the roller coaster would otherwise continue on a straight path. This force is called the centripetal force and may be caused by several factors: a slope, a cable, the track of a roller coaster itself, and so on. Sometimes, several of these factors are used at the same time to minimize the effort of each individual force. This implies that some forces applied to the object moving along the curve may change direction or be divided into components.

To allow the object to remain on the curve, the latter usually has a sideways slope pointing inwards at a certain degree. In roller coasters, there is no explicit need for this slope as the force of the tracks should be enough to produce the centripetal acceleration necessary, but it favors the conservation of energy and the smoothness of the rider's experience. On the other hand, some coasters use the absence of the slope, extreme banking, or an outward slope to add thrill to the ride. These variations receive the following names:

- **Banked curve**: This is just a slightly sloped curve (about 60 degrees). It is very common and sometimes imperceptible as it is mainly used to connect two pieces of track together that point in different directions.
- **Over-banked curve**: An over-banked curve is an element that consists of a turn or curve in which the track tilts beyond 90 degrees but less than 135 degrees.
- **Over-banked inversion**: It is similar to an over-banked curve, but it tilts more than 135^{o}, which is the angle from which an element is considered an inverting element.
- **Outward banked airtime hill**: Over-banked curve that tilts outwards instead of inwards. These are quite rare and produce a small sensation of being launched into the air.
- **Flat turns**: As the name says, these element curves are completely flat without any slope, giving to the rider the sensation of tipping over (found in Wild Mouse roller coasters, which are known for the sensation of tipping over they produce as they have several flat turns in a row, e.g., Matterhorn Blitz in Europa-Park). This type of element is quite old, as it was easy to design and build.

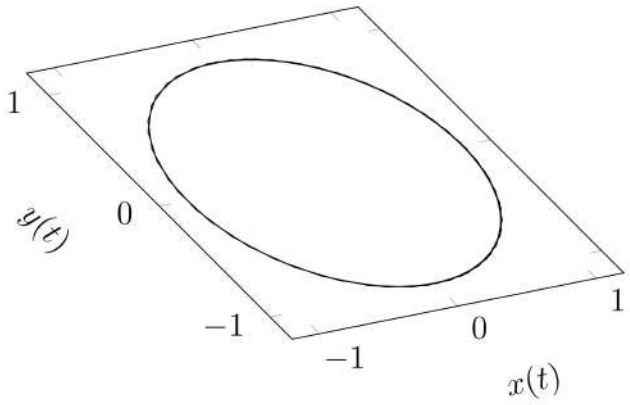


Figure 3: Graphical representation of the circle

Figure 4: Max and Moritz roller coaster at Efteling, Holland

Source: *By* [redacted], *CC BY-SA 3.0 (2022)*

4.1.1 Equation

First, we are going to consider a perfect curve in which the radius remains constant. This produces a perfect circle, which has the following equation:

$$x^2 + y^2 = R^2$$

Now, we must find an alternative for this formula to be able to draw the actual circle with respect to time, as we can't use the one above because there is no dynamic variable. There are a lot of curves that cannot be written as a single

equation in terms of only x and y. To deal with this, we are going to introduce a new parameter, φ, to produce a parametric equation. This new parameter would be the angle of rotation of the circle, which we are going to use to find a relation with time.

$$\frac{x^2}{R^2} + \frac{y^2}{R^2} = 1$$

By using the expression $\sin^2(x) + \cos^2(x) = 1$, we then know that:

$$\frac{x^{\not 2}}{R^{\not 2}} = \sin^{\not 2}(\varphi)$$

$$\frac{y^{\not 2}}{R^{\not 2}} = \cos^{\not 2}(\varphi)$$

In parametric form, we would write:

$$\vec{r}(\varphi) = \begin{cases} x(\varphi) &= R \cdot \sin\varphi \\ y(\varphi) &= R \cdot \cos\varphi \end{cases}$$

As the acceleration is zero, we know that there are no forces left that are perpendicular to the horizontal plane[9]. This is proven by Newton's second law:

$$a_z = 0 = \frac{\sum F_z}{m}$$

A similar principle can be applied to the horizontal axis, as the normal force produced by the tracks to keep the vehicle in the circular motion is applied as the centripetal force to keep the object in the circular trajectory. This means that we are assuming that we are in a perfect uniform circular motion.

We can now state that the total circular acceleration is equal to 0. This way we obtain a very simple differential equation:

$$\alpha(t) = \ddot{\varphi}(t) = 0$$

Now we have all the information we need. Solving this equation is very straightforward, we just have to integrate twice, and we will obtain the formula for the uniform circular movement:

$$\omega(t) = \omega_0 + \underbrace{a \cdot t}_{0}$$

[9] This is because the gravitational force and the vertical component of the normal force cancel out.

$$\varphi(t) = \varphi_0 + \omega \cdot t$$

Lastly, we substitute φ in the expression stated above to get the circular position with respect to time (note that the initial angle is omitted, as we are going to adjust our perspective so that the initial angle becomes negligible):

$$\boxed{\vec{r}(t) = (R \cdot \sin(\omega \cdot t), R \cdot \cos(\omega \cdot t))}$$

As a side note, most curves don't have a constant radius. These changing radiuses increase and decrease gradually and allow for the track to move towards another section of the ride. The change is not arbitrary, there are special types of variations that allow for a smooth experience and dampen any extreme G-forces that may be produced during the change in direction. The rate of change of a curve can vary (not just in a constant way, the rate of change of a curve can also change along the curve), producing the different types of curves that we find in roller coasters.

4.1.2 Resulting forces

In the flat circle, the weight and the vertical component of the normal force cancel out and the centripetal force, produced by the tracks of the roller coaster, keeps the object in a uniform circular motion. The centripetal force can be calculated by using Newton's second Law:

$$\vec{F_c} = m \cdot a_c = m \cdot \omega^2 \cdot r$$

Despite this, most curves that are intended to be used at high speeds, such as roller coaster or highways, are banked. Most banked curves are usually unnoticeable, but roller coasters may feature a wide range of banking as the vehicle is always kept on track. Outwards slopes are extremely rare and produce an outwards force, as every object tends to follow a straight line. Inwards banked curves are the most common, and make the movement smoother, as the only the perpendicular component of the weight vector cancels out with the normal force, allowing for the tangent component to push the wagon inwards.

For example, let's consider an inwards banked circle. From a front view, we can see that the slope allows for the object that is going through the curve to stay in the rotating motion. To simplify we are going to consider that there is no external force that is keeping the roller coaster in the circle. The only internal forces left are the gravitational pull and the normal force, and the resulting centripetal acceleration. The free-body diagram would be drawn as follows:

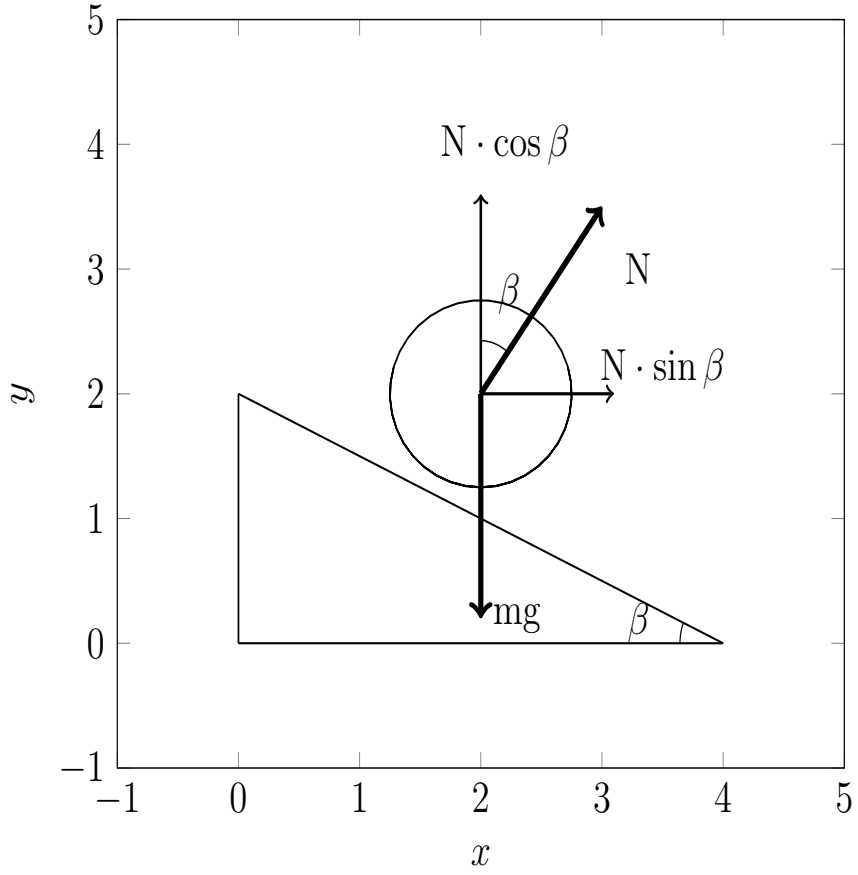


Figure 5: Graphical representation of the Free-Body diagram of a sloped curve

In this banked curve, the gravitational vector gets canceled out with the vertical component of the normal force, $mg = N \cdot \cos\beta$. Therefore, the centripetal force consists only of the horizontal component of the normal force, $N \cdot \sin\beta$. With these two formulas, we can easily calculate the inwards force exerted on the object and, therefore, the centripetal acceleration.

When in a roller coaster, the track is usually responsible for the rotating motion, so this horizontal component is not necessary. This is the case of the flat turns found in Wild Mouse roller coasters. Steeper curves allow for soft changes in forces and higher speeds with less risks.

4.1.3 Animation

This animation was quite easy to make, I just had to make time proportional to the angle traversed through the frames. This animation respected both the conservation of energy and the actual shape. I obtained the expected result. When rendering, I opted for having the camera above the circle to better illustrate the shape.
The code for the animation can be found Appendix III - Curve section.
The link to the animation on YouTube is as follows:

[redacted]

4.2 Camelback (Parabola)

Roller coasters often feature a section called camelbacks. These are just hills and valleys that resemble a parabola. A camelback is designed to lift riders out of their seats and provide a feeling of weightlessness, commonly known as *airtime*, produced by the negative G-forces. Some roller coasters are basically a series of camelbacks, built with huge steel structures that can reach impressive heights.

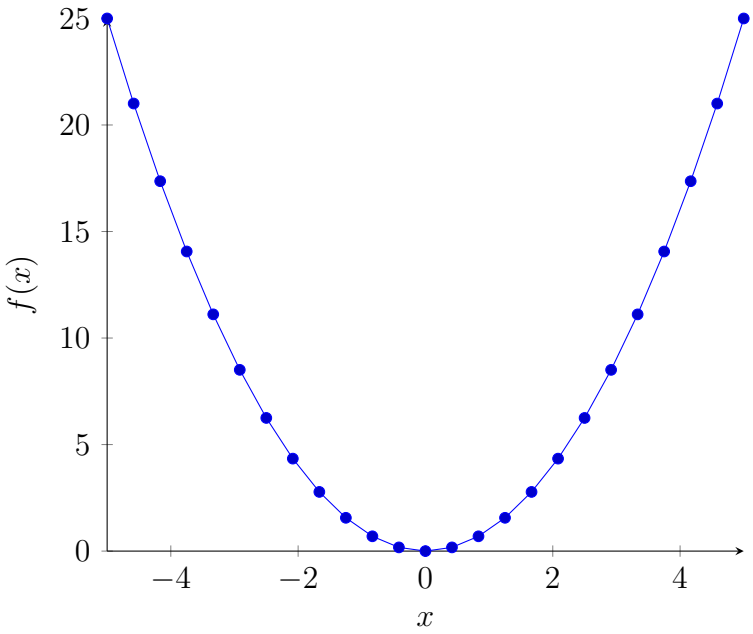


Figure 6: Graphical representation of the Parabola

Figure 7: Parabola in a Roller Coaster

Source: *WYFF4*

Figure 8: Parabola at the Shambhala Roller Coaster, Port Aventura, Tarragona

Source: *By Sotti - Own work, CC0, https://commons.wikimedia.org/*

4.2.1 Equation

To generalize, I'm going to consider that the camelback shape is produced by a perfect parabola. The static **parabola** is defined by the quadratic equation, which is as follows (with a, b and c constants):

$$y = ax^2 + bx + c$$

To simplify, we are going to consider the vertex of the parabola (the point where the parabola changes its direction), as the origin of coordinates $(0, 0)$. This means that we can neglect the terms $bx + c$, and only take into consideration $y = ax^2$. Also note how the parabola is placed vertically, but I am going to use the letters x and y for the coordinate system as it is most used in mathematics when working with two-dimensional shapes.

Notice how the variable a can be either positive or negative. The minus symbol allows for the parabola to make a hill. If a were positive, the parabola would form a valley-like shape, which would be located in between two camelbacks. Also note how the G-forces that result at the top and bottom of camelbacks also have opposite symbols.

In a free parabolic throw, the horizontal component remains constant (without acceleration), while the vertical component accelerates at a constant rate. This is due to the lack of forces horizontally and gravity being the only force vertically. Using Newton's second law ($\vec{F} = \frac{d\vec{p}}{dt}$, where $\vec{p} = m\vec{v}$), we know that $\sum \vec{F} = m \cdot \vec{a}$, where $\sum F$ is the sum of forces, m is the mass and a is the acceleration. As the only force is gravity, the sum of forces equals the weight: $\sum F = W$, which leads us to $m \cdot a = W = m \cdot g$, which means that $a = g = -9.8 m/s^2$. Mathematically, this behavior can be described by the following quadratic equations:

$$\vec{r} = \begin{cases} r_x(t) = (v_x)_0 \cdot t + x_0 \\ r_y(t) = \dfrac{a \cdot t^2}{2} + (v_y)_0 \cdot t + y_0 \end{cases}$$

Derivating these ones results in the formulas for the velocity components:

$$\vec{v} = \begin{cases} v_x(t) = (v_x)_0 \\ v_y(t) = a \cdot t + (v_y)_0 \end{cases}$$

Derivating them again we get the acceleration:

$$\vec{a} = \begin{cases} a_x(t) = 0 \\ a_y(t) = a \quad \text{(constant)} \end{cases}$$

Despite this, our parabola is produced by the force of the tracks, so the parabola doesn't follow a free trajectory, which is why the speed at its maximum height doesn't equal zero as it would in a free throw. In fact, due to the changes in the weight of the vehicle at each ride, the force of friction produces slight changes in the speed at the top of the camelback, so it wouldn't be certain whether the object would continue or if it would fall back. This is why roller coasters have a margin of speed, so that the object always continues forward.

To solve this, we are going to use the change in the slope of the parabola. We know that the derivative of a function is equal to the slope of the tangent line in that point: $f'(x) = p = \tan\beta$. This implies that $x = \dfrac{\tan\beta}{2a}$

$$\begin{cases} f(x) = ax^2 \\ f(y) = y \end{cases}$$

Next, we are going to use vectorial projection. To find the tangent vector, we are going to use the slope to figure out how much the vehicle advances vertically when moved one unit horizontally: $\vec{u} = (1, f'(x)) = (1, 2 \cdot a \cdot x)$. This gives us the tangent vector at the point $(x, f(x)) = (x, a \cdot x^2)$ on the shape. We are also going to consider the gravitational force as a vector with the following components: $\vec{g} = (g, 0)$.

We need to project the force of gravity to the tangent of the parabola to know the total acceleration as the perpendicular component of the gravitational pull gets canceled with the force of the track. The scalar product allows us to establish that relation without the need of the slope angle. The formula for the scalar product is $\vec{u} \cdot \vec{g} = \|\vec{u}\| \cdot \|\vec{g}\| \cdot \cos(\sigma)$ (with σ the angle formed between $\vec{u}$ and $\vec{g}$). Notice how $\|\vec{g}\| \cdot \cos(\sigma)$ is the projection of $\vec{g}$ in the direction of $\vec{u}$. This gives us the component of the gravity that doesn't cancel out with the normal force. If $\vec{u}$ were a unitary vector, meaning that $\|\vec{u}\| = 1$, the scalar product of the two vectors would equal

the projection of $\vec{g}$ to the tangent component of the parabola. To get the unit vector, we have to divide the vector by its modulus: $\hat{u} = \dfrac{\vec{u}}{\|\vec{u}\|}$

$$\|\vec{a}\| = \|\vec{g}\| \cdot \cos(\sigma) = \frac{\vec{u} \cdot \vec{g}}{\|\vec{u}\|} = \frac{(1, 2ax)}{\sqrt{1 + 4a^2x^2}} \cdot \frac{2axg}{\sqrt{1 + 4a^2x^2}} = \frac{(2axg, 4a^2x^2g)}{1 + 4a^2x^2}$$

Knowing that $\|\vec{a}\| = (\ddot{x}, \ddot{y})$, we can split the equation above into its two components:

$$\ddot{r}(t) = \begin{cases} \ddot{x}(t) = \dfrac{2ax(t)g}{1 + 4a^2x^2(t)} \\ \ddot{y}(t) = \dfrac{4ay(t)g}{1 + 4ay(t)} \end{cases}$$

This total acceleration is tangent to the parabolic shape at all times. This set of differential equations will allow us to isolate $\ddot{y}(t)$ and consequently $y(t)$. As we know that $y = ax^2$, if we solve the differential equation for one of the variables, we will get the other one easily. We will solve for $\ddot{y}$ as we have already been working on the y-axis.

To do this, Euler's method can be used to obtain a series of discretized solutions.

$$\begin{cases} \dot{v} = \dfrac{2axg}{1 + 4a^2x^2} \\ \dot{x} = v(x) \end{cases}$$

This way, we can calculate the velocity and position using the previous position and speed. The smaller we make the increments of t, the more accurate the animation will be.

$$\boxed{\vec{r} = \begin{cases} x_{n+1} = x_n + v(x_n) \cdot \Delta t \\ y_{n+1} = y_n + v(y_n) \cdot \Delta t \\ v(x_n) = \dfrac{2ax_ng}{1 + 4a^2x_n^2} \\ v(y_n) = \dfrac{4ay_ng}{1 + 4ay_n} \end{cases}}$$

I could have written y as a function of x, but I opted to calculate each one on their own to maintain the conservation of energy even though the object will slightly shift from the parabolic shape.

Brachistochrone curve

Sometimes, roller coaster manufacturers use a brachistochrone curve instead of a parabolic one. It is the fastest curve on the vertical plane (with loss of height) as it optimizes the velocity gained from the gravitational force initially and converting it into kinetic[10]. For the roller coaster industry, this means that the rides are shorter while gaining a lot of speed. Usually, a brachistochrone curve is used mainly on really wide curves with slow drops. Steeper drops may likely feature the parabolic curve. The brachistochrone curve is formed by the inversion of the cycloid curve, which is formed by analyzing the path of a point at the exterior of a smooth circumference as it rolls over a plane.

The formula for the cycloid curve is as follows:

$$\vec{r} = \begin{cases} x = R \cdot (\varphi - \sin\varphi) \\ y = R \cdot (1 - \cos(\varphi)) \end{cases}$$

4.2.2 Resulting forces

The resulting forces are all produced by the change in direction. The normal force is equal to the normal component of the weight. If the normal force and the weight were to produce a 90-degree angle, the object would fall freely, as the normal force would equal zero. Therefore, the normal force is never bigger than the weight and they are only equal to each other when they are opposite from each other.

When the vehicle in the roller coaster descends, the weight force favors the direction of movement and the velocity increases. The tangent component of the weight, which points downwards, is the only force left so the rider feels lighter than normally.

At the lowest point of the shape, the weight gets canceled out with the normal force, but the change in direction produces the inertia to push the vehicle down. This is one of the moments of most extreme forces and the rider feels then a lot heavier than usual.

Lastly, during the ascend, the direction of movement is opposite to the tangent component of the weight (the normal component gets canceled out again) and the rider feels heavier than usual as he/she gets pulled in the direction of the track. Although it's portrayed in the animation, roller coasters don't usually stop at the middle of the ride unless they change direction[11], and generally it is a controlled stop, not simply the action of the friction or weight. Theoretically, it would be

[10] In fact, the word Brachistochrone comes from Ancient Greek and it translates to "shortest time".

[11] This is the case of roller coasters which go backwards for a section of the ride. An example of this is Raik at Phantasialand, Germany or Stunt Fall at Parque Warner Madrid, Spain.

a sensation opposite to that produced at the vertex of the parabola, resulting in a sensation of weightlessness (also called airtime). Instead of going back, a lot of roller coasters feature another parabolic shape right after finishing the one before, and the rider can feel some airtime at the highest point.

4.2.3 Animation

To make this animation more visual, I decided to add a simple parabolic shape that shows where the ball is going through instead of drawing a path. Although the object goes slightly out of the shape, the conservation of energy principle is preserved, and the ball does therefore have the correct speed relative to the position at each moment.
The code for the animation can be found Appendix III - Parabola section.
The link to the animation on YouTube is as follows:

4.3 Catenary

The catenary curve is the curve formed by a **uniform thread** that is hanging from two points. A thread being uniform means that its density and diameter are equal at all its points.

In modern roller coasters, the catenary isn't used often, as the change in steepness isn't as smooth as in a parabola. Some Top Hat roller coasters (as seen in figure 10) feature a shape that resembles the catenary. Despite this, the main uses of catenaries are architecture and in zip lines.

In architecture, we can see a clear example of the usage of the catenary in the Sagrada Família by Gaudí. He built the model at a smaller scale upside down using hanging threads of different lengths.

Zip lines are attractions in which the rider holds on to a small seat tied to the thread. Then, the rider coasts through the zip line until the breaks turn on and the rider is caught at the end by a net. An example of this is El Vol de l'Àliga (in Campdevànol, Girona), which is 270 meters long and has a 40-meter height difference.

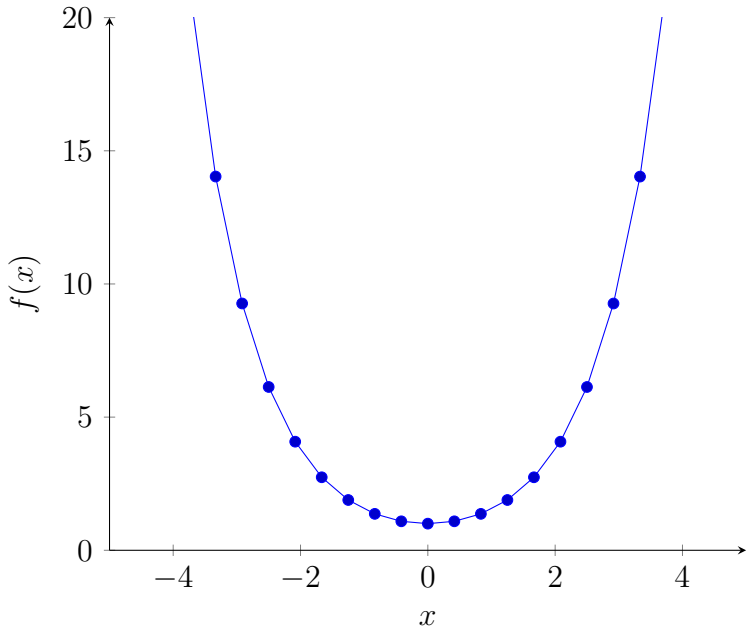


Figure 9: Graphical representation of a Catenary

Figure 10: Red Force Catenary at Ferrari Land (Port Aventura)

Source: *PA community*

4.3.1 Equation

We know that the equation for the catenary is the following:

$$f(x) = a \cdot \cosh\left(\frac{x}{a}\right) = \frac{a}{2} \cdot \left(e^{\frac{x}{a}} + e^{-\frac{x}{a}}\right)$$

Firstly, we are going to use Newton's Second Law to determine the vertical acceleration of the vehicle.

$$\sum F = m \cdot a \implies a = g \cdot \cos^2 \beta$$

We can easily find the derivative of the function by using the chain rule with the substitution $y = \cosh u$ and $u = \frac{x}{a}$:

$$f'(x) = a \cdot \frac{d}{dx} \cosh\left(\frac{x}{a}\right) = a \cdot \left(\frac{dy}{du}\right) \cdot \left(\frac{du}{dx}\right) = \frac{\cancel{a}}{\cancel{a}} \sinh(u) = \sinh\left(\frac{x}{a}\right)$$

Using the formula explained above, we obtain the following non-linear differential equation:

$$\vec{a} = \left(\frac{g \cdot f'(x)}{1 + f'^2(x)}, \frac{g \cdot f'(x)}{1 + f'^2(x)}\right) = \left(\frac{g \cdot \sinh\left(\frac{x}{a}\right)}{1 + \sinh^2\left(\frac{x}{a}\right)}, \frac{g \cdot \sinh^2\left(\frac{x}{a}\right)}{1 + \sinh^2\left(\frac{x}{a}\right)}\right)$$

Euler's method tells us that we can express this equation as:

$$\vec{r}_x = \begin{cases} \dot{v}_x = \frac{g \cdot f'(x)}{1+f'(x)} \\ \dot{x} = v_x \end{cases}$$

$$\vec{r}_y = \begin{cases} \dot{v}_y = \frac{g \cdot f'(x)}{1+f'(x)} \\ \dot{y} = v_y \end{cases}$$

If we add the derivative and integrate with respect to time, we obtain the formula (we are going to consider the constants to be the previous velocities and positions):

$$\vec{r}_{n+1} = \begin{cases} (r_x)_{n+1} = \begin{cases} (v_x)_{n+1} = \dfrac{g \cdot \sinh\left(\frac{x}{a}\right)}{1+\sinh^2\left(\frac{x}{a}\right)} \cdot \Delta t + (v_x)_n \\ x_{n+1} = (v_x)_n \cdot \Delta t + x_n \end{cases} \\ (r_y)_{n+1} = \begin{cases} (v_y)_{n+1} = \dfrac{g \cdot \sinh^2\left(\frac{x}{a}\right)}{1+\sinh^2\left(\frac{x}{a}\right)} \cdot \Delta t + (v_y)_n \\ y_{n+1} = (v_y)_n \cdot \Delta t + y_n \end{cases} \end{cases}$$

This is our final formula for the catenary. Notice how we could also solve the differential equation for only one axis and having a single pair of discretized equations. This would result in a more accurate animation in terms of the mathematical shape, but I have opted for using discretized equations on both axes to maintain the conservation of energy principle.

4.3.2 Resulting forces

The resulting forces applied on the track are almost identical to the parabola, but the rate of change is not as constant, as the first derivative is a hyperbolic function, not a linear one. As we are considering the mass to be constant, by Newton's second Law:

$$\sum \vec{F} = m \cdot \vec{a}$$

As the force and the acceleration are proportional to each other, the graphical representation of the acceleration is the same as the graphical representation of the resulting net force but scaled accordingly to the mass. To simplify, we are going to consider a test object that has mass 1. Therefore, the representation of the forces would be the second derivative.

$$f(x) = a \cdot \cosh\left(\frac{x}{a}\right) \implies f'(x) = \sinh\left(\frac{x}{a}\right) \implies f''(x) = \frac{1}{a} \cdot \cosh\left(\frac{x}{a}\right)$$

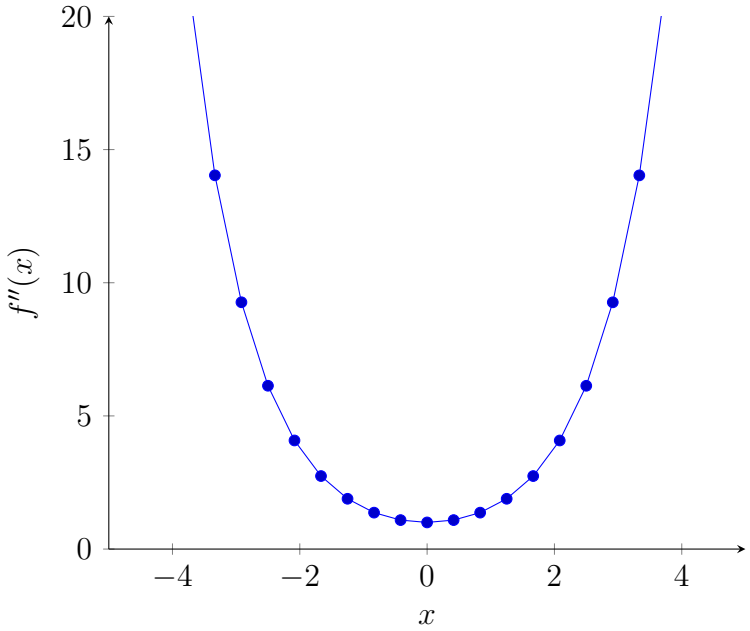


Figure 11: Graphical representation of the forces of a Catenary

4.3.3 Animation

Similarly to the parabola, the animation for the catenary worked well, although there is a bigger shift in the precision of the animation. This has been solved by decreasing the step size and therefore increasing the precision.
The code for the animation can be found Appendix III - Catenary section.
The link to the animation on YouTube is as follows:

5 Parameterized Elements

5.1 Helix

A helix is a type of curve in the three-dimensional space. It is characterized by the fact that the tangent line at any point makes a constant angle with the horizontal axis. Because of the force of gravity, every object that is going through the helix is supposed to increase speed on the downwards axis. As the position is fixed by the track, we experience a pronounced increase in the tangential speed instead.

Helices aren't common on roller coasters, and they aren't a specially thrilling or exciting element. This is because helices tend to have small changes in G-forces, which makes them unexciting, but it also means that they can be used to easily connect most of the other figures. This is the case of short ascending or descending helices, which are often used to connect two near pieces of track that are at a relatively close height. On the other hand, long ascending helices are used as a replacement for the chain lift hill in small roller coasters.

As the helix isn't a complicated element to build and doesn't involve much danger, older roller coasters feature this element more often in conjunction with flat turns. Similarly, some water attractions may also include these sections, as the increase of speed is much more noticeable since it starts with a very low speed and aren't strapped to a wagon, which means that the rider is thrown to the sides of the tube.

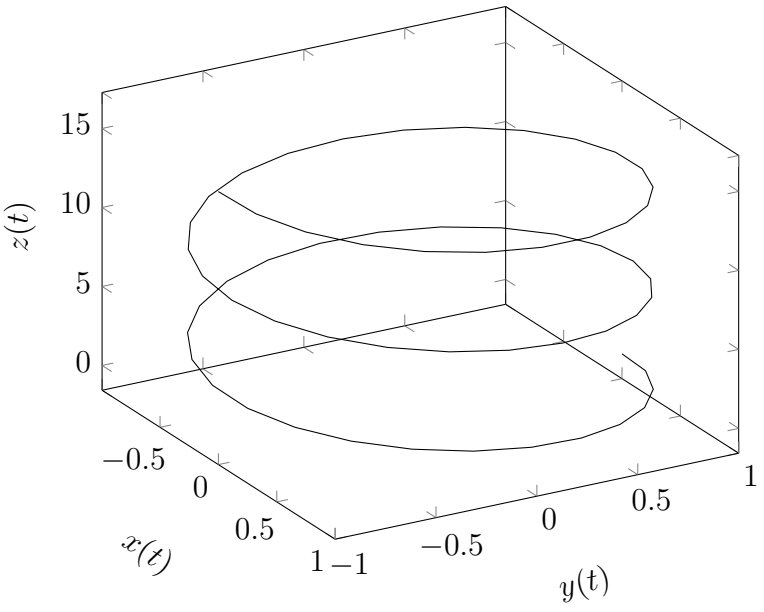


Figure 12: Graphical representation of a Helix

Figure 13: Toboggan roller coaster at Hersheypark

Source: *by Myselfalso, CC-BY-SA (2012)*

Figure 14: Helix at the Python roller coaster, Efteling

Source: *By* [redacted], *CC BY-SA 3.0 (2022)*

5.1.1 Equation

The static helix with respect to the rotation from above (as used in the chart 12) can be defined by the following parametric equations, where R is the radius, θ is the angle (seen from above) and p represents the inclination of the helix and is measured as a proportion between the vertical and horizontal movement:

$$\vec{r}(\theta) = \begin{cases} x(\theta) = R \cdot \cos(\theta) \\ y(\theta) = R \cdot \sin(\theta) \\ z(\theta) = -p \cdot \theta \end{cases}$$

This means that the helix has a circular shape when it is observed from above. Notice how this equation is written as $\vec{r}(\theta)$, so we cannot know the velocity just by taking the derivative directly, as we know that the velocity varies with time. Therefore, our goal is to rewrite this equation as a function with respect to time. Additionally, we know that θ also changes with time, which means that the equation from above could also be written as $\vec{r}(\theta(t))$, which will allow us to figure out the formula. A perfect helix has a constant radius and slope, so the only thing that varies with time is the angle.

As we only take into consideration two forces, the force of gravity and the force of the tracks that allows for the object to stay in its path (the normal force), we will take a closer look only to the vertical component of the equation above. As of now, the equation for the vertical component is written as z of θ ($z(\theta(t))$), but we want it to be z of t ($z(t)$).

First, we can express the slope using trigonometry considering that β is the angle formed between line tangent to the slope and the vertical axis:

$$\tan\beta = \frac{R\Delta\theta}{\Delta z} = \frac{-R}{\left(\frac{\Delta z}{\Delta\theta}\right)} = \frac{-R}{p}$$

Now we can take into account that the vertical acceleration is constant. This means that we know that the tangent acceleration is equal to the gravity times the cosine of the angle:

$\|\vec{a}_T\| = k = g \cdot \cos\beta$

By the same rule, the vertical acceleration is the tangent acceleration times the cosine of the angle.

$$\|\vec{a}_Z\| = \|\vec{a}_T\| \cdot \cos\beta = g \cdot \cos^2\beta$$

We also know the following: $a_Z = \dfrac{d^2 z(\theta(t))}{dt^2} = \ddot{z}(\theta)$, but our equation is currently $z(\theta)$, not $z(t)$. To solve this, we can multiply and divide the fraction by $d\theta$: $\dfrac{dz(\theta)}{dt} = \dfrac{dz(\theta)}{d\theta} \cdot \dfrac{d\theta}{dt} = \dfrac{dz(\theta)}{d\theta} \cdot \dot{\beta}(t)$

We can use the chain rule to derivate our formula, which results in:

$$\frac{d^2 z(\theta(t))}{dt^2} = \underbrace{\frac{d^2 z(\theta)}{d\theta^2} \cdot \dot{\theta}^2(t)}_{0} + \underbrace{\frac{dz(\theta)}{d\theta}}_{-p} \cdot \ddot{\theta}(t) = -p \cdot \ddot{\theta}(t)$$

We know that the second derivative of z is equal to $\|a_Z\|$. Adding this to the equation we obtained above, we get: $-p \cdot \ddot{\theta}(t) = g \cdot \cos^2(\beta)$, which is a second-degree differential equation.

Now we will simplify $\cos^2(\beta)$ in terms of other constant variables (this step isn't strictly necessary, but it allows for a clearer solution because our original equation also had the p as a slope, so β would depend on p anyway).

We will start using an expression we found before: $\tan(\beta) = -\frac{R}{p}\frac{\sin\beta}{\cos(\beta)} = -\frac{R}{p}$

We also know that: $\cos^2(\beta) + \sin^2(\beta) = 1$. As $\sin(\beta) = \sqrt{1-\cos^2\beta}$, we can replace it in the formula above: $\cos(\beta) = -\frac{p}{R} \cdot \sin\beta = -\frac{p}{R} \cdot \sqrt{1-\cos^2\beta}$

$$\cos^2\beta \cdot \left(1 + \frac{p^2}{R^2}\right) = \frac{p^2}{R^2}$$

$$\cos^2\beta = \frac{p^2}{R^2 + p^2}$$

When we then substitute this above, we get the following differential equation:

$$\ddot{\theta}(t) = -\frac{g \cdot p}{(R^2 + p^2)}$$

Lastly, we integrate the expression with respect to time to obtain the speed:

$$\dot{\theta}(t) = \frac{-g \cdot p}{R^2 + p^2} \cdot t + C_1$$

And yet again to obtain the position:

$$\theta(t) = \frac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + C_1 t + C_2$$

Here we can see that two constant of integration appear. They indicate the initial velocity and height (C_1 and C_2, respectively) of the object.

$$\boxed{\theta(t) = \frac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0}$$

If we substitute this in our original formula, we obtain:

$$\vec{r} = \begin{cases} x(t) = R \cdot \cos\left(\frac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0\right) \\ y(t) = R \cdot \sin\left(\frac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0\right) \\ z(t) = -p \cdot \left(\frac{-g \cdot p}{2R^2 + 2p^2} \cdot t^2 + v_0 t + z_0\right) \end{cases}$$

In this last formula, R is the radius, g is the force of gravity ($-9.8m/s^2$ on the surface of the Earth), p is the inclination, t is time, v_0 is the initial velocity and z_0, is the initial height.

5.1.2 Resulting forces

The helix is an easy shape in terms of the resulting forces as it is symmetric circularly. Firstly, there's obviously gravity, which produces the downwards acceleration. The centripetal force required to stay in the circular motion is produced by the component of the normal force which is perpendicular to the track. This normal force is also responsible for avoiding a free fall. As the weight doesn't fully cancel out, there is a small acceleration applied which means that the velocity increases while the object loses height (which respects the conservation of mechanical energy principle).

To reduce the strain on the track, helices tend to tilt inward to reduce the horizontal forces presented to the track by canceling them out with the normal force.

The centripetal acceleration depends on the velocity by the expression:

$$a_c = \frac{v^2}{r}$$

This means that, as the velocity is increasing along the helix, the vehicle gains centripetal acceleration, which implies that the centripetal force increases proportionally. This increase in the normal force is perceptible to the rider, as it may feel like the vehicle is launched away.

In this formula, we can also see that the radius of the helix is also involved. The wider the helix, the less force ends up being exerted on the vehicle.

5.1.3 Animation

The animation was produced using the formula above. Here are three snapshots of the animation seen form the side at the frames specified. Note that we can see a increase in speed between the second and third images with respect to the first and second image even though the distance in frames is the same.

(a) Frame 50 (b) Frame 150 (c) Frame 250

Figure 15: Sideview of the animated Helix (using Blender 3D)

The code for the animation can be found Appendix III - Helix section.
The link to the animation on YouTube is as follows:

[redacted]

5.2 In-line Twist

In an in-line twist, the track rotates around a central horizontal axis. This axis is usually located inside the track but can also be placed at the center of the riders' bodies, resulting in a **heartline roll**.

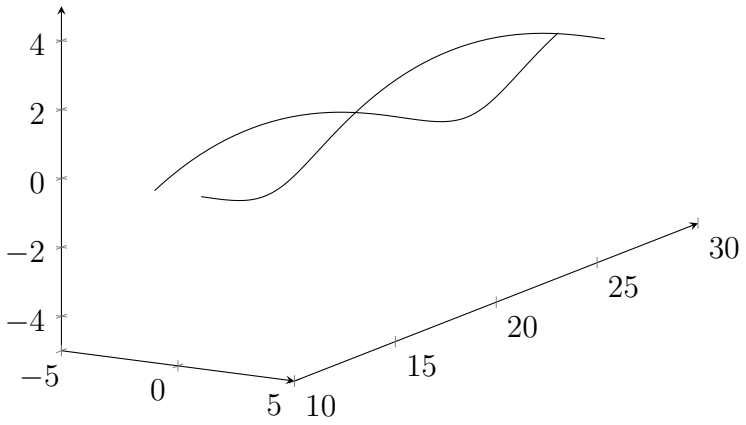


Figure 16: Graphical representation of an In-line Twist

Figure 17: In-line Twist of the Furius Baco coaster, Port Aventura

Source: *By PCG44 - Own work, CC BY-SA 4.0*

5.2.1 Equation

The basic formula for the in-line twist is quite similar to the one from the helix. Here, we can see that the variables y and z are swapped, so the vehicle won't always be moving parallel to gravity.

$$\vec{r}(t) = \begin{cases} x(t) &= R \cdot \cos(\theta) \\ y(t) &= k \cdot \theta \\ z(t) &= R \cdot \sin(\theta) \end{cases}$$

x, y and z represent position in all three axes, the variable R represents the radius of the helix and θ represents the angle traversed along the helix.

As we are going to use the unitary vector method, I'm going to analytically calculate the first and second derivative of the function above:

$$\vec{r}'(t) = \begin{cases} x'(t) &= -R \cdot \sin(\theta) \\ y'(t) &= k \\ z'(t) &= R \cdot \cos(\theta) \end{cases}$$

$$\vec{r}''(t) = \begin{cases} x''(t) &= -R \cdot \cos(\theta) \\ y''(t) &= 0 \\ z''(t) &= -R \cdot \sin(\theta) \end{cases}$$

Firstly, we are going to find the total acceleration. As $\vec{a}$ is also the second time derivative of $\vec{r}(\theta)$, By using the chain rule, we know that $\vec{a} = \vec{r}''(\theta) \cdot \dot{\theta}^2 + \vec{r}'(\theta) \cdot \ddot{\theta}$.

We also know that the vectors for $\vec{r}'$ and $\vec{r}''$ are orthogonal, which we know because, by the Pythagorean theorem, the modulus of $r'(\theta)$ is:

$$\sqrt{R^2 \cdot \sin^2(\theta) + k^2 + R^2 \cdot \cos^2(\theta)} = \sqrt{R^2 + k^2}$$

We know that both R and k are constants. Therefore the object doesn't increase the modulus of its derivative. As the object changes direction, it must have some acceleration which has to be at a right angle from the velocity. This means that we can affirm that $\cos(\sigma) = 0$. Consequently, $(\vec{r}''(\theta) \cdot \vec{r}'(\theta)) = 0$, which is proven using the dot product between two vectors:

$$\vec{r}''(\theta) \cdot \vec{r}'(\theta) = \|\vec{r}''\| \cdot \|\vec{r}'\| \cdot \cos(\sigma) = 0$$

Now we can use this formula to simplify the expression for the chain rule by multiplying it by $\vec{r}'(\theta)$.

$$\vec{a} \cdot \vec{r}'(\theta) = \underbrace{\vec{r}''(\theta) \cdot \vec{r}'(\theta) \cdot \dot{\theta}^2(t)}_{0} + \vec{r}'(\theta)^2 \cdot \ddot{\theta}(t)$$

This becomes:

$$\boxed{(\vec{a} \cdot \vec{r}'(\theta)) = \vec{r}'^2(\theta) \cdot \ddot{\theta}(t)}$$

Now, we are going to find another expression that will help us in solving this equation. We know that

$$(\vec{g} \cdot \vec{r}'(\theta)) = (0, 0, g) \cdot (-R\sin(\theta), k, R\cos(\theta)) = g \cdot R \cdot \cos(\theta)$$

As we know that the total acceleration produced in the tangential direction is the same as the projection of gravity over that tangent. Therefore, $\vec{a} \cdot \vec{r}'(\theta) = \vec{g} \cdot \vec{r}'(\theta)$. If we use this with the formulas we have obtained until now, we get the equality:

$$\vec{r}'^2(\theta) \cdot \ddot{\theta}(t) = g \cdot R \cdot \cos(\theta)$$

Using the Pythagorean theorem, we know that $\vec{r}'^2(\theta) = R^2 + k^2$, this results in the following second-order differential equation:

$$\ddot{\theta} = \frac{g \cdot R \cdot \cos(\theta)}{R^2 + k^2}$$

To solve this equation, we can apply Euler's method for the angle of rotation θ. Then we can finally add the initial vector for position, which will now return a position at a certain moment in time.

$$\vec{r} = \begin{cases} \theta = \begin{cases} \ddot{\theta}_n = \dfrac{g \cdot R \cdot \cos(\theta_n)}{R^2 \cdot k^2} \\ \dot{\theta}_{n+1} = \ddot{\theta}_n \cdot \Delta t + \dot{\theta}_n \\ \theta_{n+1} = \dot{\theta}_n \cdot \Delta t + \theta_n \end{cases} \\ r_x \longrightarrow x_{n+1} = R \cdot \cos(\theta_n) \\ r_y \longrightarrow y_{n+1} = k \cdot \theta_n \\ r_z \longrightarrow z_{n+1} = R \cdot \sin(\theta_n) \end{cases}$$

5.2.2 Resulting forces

Before entering the in-line twist, the object is completely vertical, which means that the weight and the normal force cancel out. Then, as the object starts the in-line twist, the normal force (which is always perpendicular to the track) gains a horizontal component and the object and the weight doesn't get canceled out, which means that the object's center of mass slowly starts to lose height, which gets transformed into velocity.

As it starts going around the twist, the vehicle gains acceleration until it has gone a quarter of the rotation. At that point, the weight and the normal force are completely perpendicular. Then, the acceleration starts to decrease, although the velocity is still increasing.

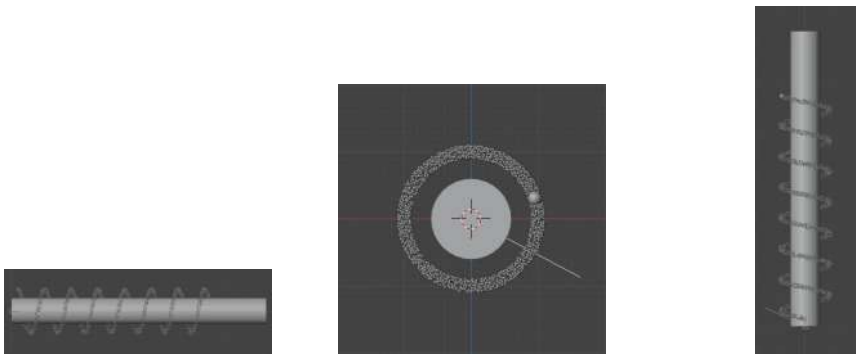
At the bottom of the twist, the centripetal acceleration equals to zero and changes sign, which means that the velocity starts to decrease.

While the object is going up, the vehicle is slowed down due to the gravitational pull, until it's standing vertically again. During this ascend, the acceleration also increases and decreases due to the change in the angle formed with the normal force.

When the vehicle is back up, it has returned to its initial speed. This is because all forces are conservative, and the mechanical energy of the vehicle is constant.

The formula for the mechanical energy $\Delta E_M = 0$, which means that the amount of energy cannot vary, only it's function (potential or kinetic).

5.2.3 Animation



(a) X-Axis (b) Y-Axis (c) Z-Axis

Figure 18: Three different views of the animated In-line Twist (using Blender 3D) at the frame 900

The code for the animation can be found Appendix III - In-line Twist section. The link to the animation on YouTube is as follows:

5.3 Vertical Loop (Clothoid)

These are the most common and popular type of inversions. They used to be manufactured in a circular shape, but they currently use an inverted teardrop shaped to reduce the resulting forces during the change in direction. There are several shapes that can produce the necessary damping, according to the objective. There are two properties that can be dampened: the centripetal acceleration and the G-forces.

A curve with a constant centripetal acceleration has the following set of discretized equations [2]:

$$\vec{r} = \begin{cases} \theta_{n+1} &= \theta_n + \dfrac{C}{\left(\dfrac{v_0^2}{g} - 2 \cdot y_n\right)} \cdot \Delta s \\ x_{n+1} &= x_n + \cos(\theta_n) \cdot \Delta s \\ x_{n+1} &= x_n + \cos(\theta_n) \cdot \Delta s \end{cases}$$

However, there is still a strong change of the forces at the beginning and at the end of the loop. This is where clothoids appear. Clothoids are usually used to connect two circular objects in a smooth trajectory. In a clothoid, the curvature changes proportionally to the length. A full clothoid is as follows:

Figure 19: Vertical Loop of the Blue Fire Megacoaster, Europa Park

Source: *By* [redacted], *CC BY-SA 3.0 (2022)*

5.3.1 Equation

We can use a piece-wise function to define a loop as two opposite clothoids, with a semicircle between them.

Firstly, we will analyze the clothoid curve. In a **clothoid**, the radius of curvature is indirectly proportional to the arc traversed ($s \cdot R = k$). These are used because they minimize the forces that are applied. The formula for the clothoid curve is written using the Fresnel Integrals:

$$\vec{r}(\theta) = \begin{cases} C(\theta) &= \int_0^{\theta} \cos\lambda^2 \; d\lambda \\ S(\theta) &= \int_0^{\theta} \sin\lambda^2 \; d\lambda \end{cases}$$

This integral cannot be derivated directly as the variable of integration is one of the limits, so we need to use Leibniz's integration rule, which states that:

$$\begin{aligned} \frac{d}{dx}\left(\int_{a(x)}^{b(x)} f(x,\lambda)\; d\lambda\right) = \; & f(x,b(x)) \cdot \frac{d}{dx} b(x) \\ & - f(x,a(x)) \cdot \frac{d}{dx} a(x) \\ & + \int_{a(x)}^{b(x)} \frac{\partial}{\partial x} f(x,\lambda)\; d\lambda \end{aligned}$$

Substituting results in the following:

$$\begin{aligned} \frac{d}{dx}\left(\int_0^{\theta} \cos(\lambda^2)\; d\lambda\right) = \; & \cos(\theta^2) \cdot \cancelto{1}{\frac{d}{dx}\theta} \\ & - \cancelto{0}{\cos(0^2) \cdot \frac{d}{dx} a(x)} \\ & + \cancelto{0}{\int_{a(x)}^{b(x)} \frac{\partial}{\partial x} \cos(\theta^2)\; d\lambda} \end{aligned}$$

Doing the same with the vertical axis, we now know that the derivative of the clothoid function is:

$$r'(\theta) = \begin{cases} C'(\theta) &= \cos(\theta^2) \\ S'(\theta) &= \sin(\theta^2) \end{cases}$$

This can be derivated again easily by using substitution:

$$r''(\theta) = \begin{cases} C''(\theta) = -2 \cdot \theta \cdot \sin(\theta^2) \\ S''(\theta) = 2 \cdot \theta \cdot \cos(\theta^2) \end{cases}$$

Firstly, we are going to find the total acceleration. To do this, we know that $\vec{a} = \ddot{r}(\theta)$. We can use the chain rule to transform this equality into the following:

$$\dot{r}(\theta) = \vec{r}'(\theta) \cdot \dot{\theta}(t)$$

Now we use the product rule to derivate again with respect to time:

$$\vec{a} = \vec{r}''(\theta) \cdot \dot{\theta}^2(t) + \vec{r}'(\theta) \cdot \ddot{\theta}(t)$$

In the initial formula, $C(\theta)$ and $S(\theta)$ are the corresponding horizontal and vertical positions, which use the abstract parameter θ as a value at each point.

We also know that the vectors for $\vec{r}'$ and $\vec{r}''$ are orthogonal because the modulus of $\vec{r}'$ is constant, which means that $\cos(\sigma) = 0$ and consequently, $(\vec{r}''(\theta) \cdot \vec{r}'(\theta)) = 0$. This is proven using the scalar product between two vectors:

$$\vec{r}''(\theta) \cdot \vec{r}'(\theta) = \|\vec{r}''\| \cdot \|\vec{r}'\| \cdot \cancelto{0}{\cos 90} = 0$$

If we multiply the formula for $\vec{a}$ by $\vec{r}(\theta)$ and using the property that we just described, we then get:

$$\vec{a} \cdot \vec{r}'(\theta) = \underbrace{\vec{r}''(\theta) \cdot \vec{r}'(\theta) \cdot \dot{\theta}^2(t)}_{0} + \vec{r}'(\theta)^2 \cdot \ddot{\theta}(t)$$

$$(\vec{a} \cdot \vec{r}'(\theta)) = \vec{r}'^2(\theta) \cdot \ddot{\theta}(t)$$

If we project the gravitational vector $\vec{g}$ over $\vec{r}'(\theta)$, we get the expression for:

$$(\vec{g} \cdot \vec{r}'(\theta)) = (0, g) \cdot (\cos(\theta^2), \sin(\theta^2)) = g \cdot \sin(\theta^2)$$

As we know that the total acceleration produced in the tangential direction is the same as the projection of gravity over that tangent. Therefore, $\vec{a} \cdot \vec{r}'(\theta) = \vec{g} \cdot \vec{r}'(\theta)$. If we use this with the formulas we have obtained until now, we get the equality:

$$\ddot{\theta}(t) \cdot \vec{r}'^2(\theta) = g \cdot \sin(\theta^2)$$

As we know that $\vec{r}'^2(\theta) = \sin^2(\theta^2) + \cos^2(\theta^2) = 1$, this results in the following second-order differential equation:

$$\boxed{\ddot{\theta} = g \cdot \sin(\theta)}$$

We can now solve this differential equation numerically by applying Euler's method for the parameter θ.

$$\theta = \begin{cases} \ddot{\theta}_n & = g \cdot \sin(\theta_n^2) \\ \dot{\theta}_{n+1} & = \ddot{\theta}_n \cdot \Delta t + \dot{\theta}_n \\ \theta_{n+1} & = \dot{\theta}_n \cdot \Delta t + \theta_n \end{cases}$$

Now we can use this in our original equation to find the position with respect to time, but because the purpose of this is to animate the movement with Python, we have to avoid using integrals[12].

$$\vec{r} = \begin{cases} \theta & = \begin{cases} \ddot{\theta}_n & = g \cdot \sin(\theta_n^2) \\ \dot{\theta}_{n+1} & = \ddot{\theta}_n \cdot \Delta t + \dot{\theta}_n \\ \theta_{n+1} & = \dot{\theta}_n \cdot \Delta t + \theta_n \end{cases} \\ r_x & = \begin{cases} \dot{x}_{n+1} & = \cos(\theta_n^2) \\ x_{n+1} & = \dot{x}_n \cdot \Delta t + x_n \end{cases} \\ r_y & = \begin{cases} \dot{y}_{n+1} & = \sin(\theta_n^2) \\ y_{n+1} & = \dot{y}_n \cdot \Delta t + y_n \end{cases} \end{cases}$$

5.3.2 Resulting forces

At the beginning of the loop, we have a change of forces and direction of movement. There can be no completely circular loops, as the forces needed for these changes would be too strong. The G-forces are going to allow us to analyze the vertical loop.

The G forces compare the net force with the gravitational acceleration:

$$G = \frac{F}{m \cdot g}$$

As a reference, most people can only take $6G$ for a second. Here is a chart in which we can see how long a roller coaster can apply a certain G-force according to the roller coaster safety regulations.

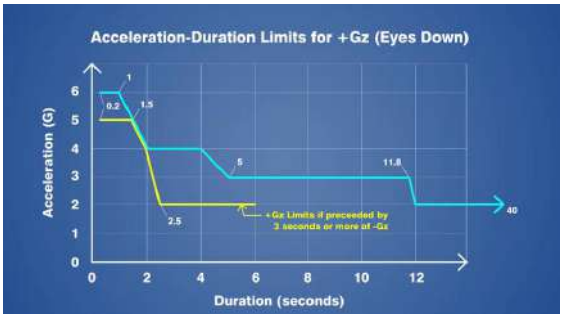


Figure 20: ASTM standards for G-force

Source: *Research Gate*

[12] It is probably possible to use integrals in Python, but I'm going to avoid it because it is not necessary and may be even more complicated than using Euler's method again.

The use of the clothoid shape allows for a gradual change in the balance between the weight and the normal force to avoid all these extreme changes in the G-forces. Thanks to the math involved in the creation of roller coasters, the number of accidents and injuries decreases, which makes for more thrilling and safe rides.

When the vehicle arrives to the circular section, the normal force becomes perpendicular to the weight, which means that the acceleration is equal to the force of gravity. At the highest point of the loop, the direction of movement and the gravitational pull have the same direction, so the vehicle starts accelerating again. Since the normal force ends up aligning with the weight again at the end of the loop, we can see that the acceleration decreases. Thus, we have the maximum acceleration at the sides of the loop.

5.3.3 Animation

This animation ended up being the worst one, because, as the static motion is defined by a set of integrals, the shift in Euler's method is clearly perceptible.
The code for the animation can be found Appendix III - Vertical Loop section.
The link to the animation on YouTube is as follows:

5.4 Corkscrew

A corkscrew is a type of inversion that resembles a vertical loop that has been stretched so that the entrance and exit points are a certain distance away from each other. In most roller coasters, the direction in which the corkscrew is entered is parallel to the direction in which the corkscrew is exited. At the moment of inversion, the riders produce a 90-degree angle from the incoming and exiting tracks when seen from above.

Figure 21: Matterhorn Blitz roller coaster at Europa Park, Germany

Source: *By Coasterman1234 at en.wikipedia, CC BY-SA 3.0*

5.4.1 Equation

Now that we have the in-line twist and the vertical loop, we can combine them to form the corkscrew, which I will divide into three segments: the clothoid used to enter, the middle circular section and the clothoid used to exit the corkscrew. Notice that we are also adding a third axis to avoid any type of overlapping that could be produced, as the shape would be impossible otherwise. This is the only figure in which we are actually going to use algebraic and geometric properties to work out the solution to the formula, as we need to know the position at which the curve changes. This could also be done analytically using python, but the explanation would be the same, only the numbers could be scaled proportionally. This is also the reason why I am going to make a lot of assumptions in this element.

Firstly, the corkscrew starts with a clothoid shape to accommodate for the change in the radius of curvature. We know that the clothoid section ends once the corkscrew has a perfectly vertical slope, which is a point that we can calculate. Then, the twist section starts, so we need to match the curvature of both sections to fit with each other. To do this, we can take advantage of the circle's constant

curvature and the clothoids curvature, which is directly proportional to the arc length. As we have calculated before, the formula for the parameter in the solved clothoid shape is:

$$\boxed{\vec{r} = \begin{cases} \theta & = \begin{cases} \ddot{\theta}_n & = g \cdot \sin(\theta_n^2) \\ \dot{\theta}_{n+1} & = \ddot{\theta}_n \cdot \Delta t + \dot{\theta}_n \\ \theta_{n+1} & = \dot{\theta}_n \cdot \Delta t + \theta_n \end{cases} \end{cases}}$$

To be able to transition smoothly into the next section, we are going to use the variation of the parameter θ to advance in the y-axis. This makes for a more realistic link between the two different shapes.

To connect the clothoid with the following section, we are going to have to calculate the point at which the clothoid ends and the circle starts. We can do this easily, because we know that the derivative of the horizontal component of the clothoid at that point is 0. Thus, we get the following equation:

$$\cos(\theta^2) = 0$$

Solving this trigonometric equation gives us an unlimited number of solutions, but we'll keep the one that's closest to zero and is positive, as that will be the first time that the clothoid changes direction on the horizontal axis.

$$\theta = \sqrt{\arccos(0)} = \sqrt{\frac{\pi}{2}}$$

As we now know the arc length, we can now calculate the point at which the curve changes. We're going to call this point P_1.

$$P_1 = \vec{r}\left(\sqrt{\frac{\pi}{2}}\right) = \begin{cases} x\left(\sqrt{\frac{\pi}{2}}\right) & = \int_0^{\sqrt{\frac{\pi}{2}}} \cos\lambda^2 \, d\lambda \\ y = 0 & \\ z\left(\sqrt{\frac{\pi}{2}}\right) & = \int_0^{\sqrt{\frac{\pi}{2}}} \sin\lambda^2 \, d\lambda \end{cases}$$

This result can be scaled up or down, depending on how we want to draw the clothoid. At this point, the type of curve changes from a clothoid to a horizontal spiral. When programming, we are going to gain precision by simply shifting the result that we obtain from the twist.

The definition of clothoid also states that the radius of curvature (R) is inversely proportional to the arc length (θ) with respect to a constant of curvature. Then, the formula for the radius of curvature at the point of horizontal inflection is:

$$R = \frac{c^2}{\theta} = \frac{c^2}{\sqrt{\frac{\pi}{2}}} = c^2 \cdot \sqrt{\frac{2}{\pi}}$$

The middle section of the corkscrew consists of a series of perfectly circular loops that advance in the horizontal direction. Much like he in-line twist, these can be described using three coordinates, but now we can't choose the coordinate $(0, 0, 0)$ as a starting point. We solve this by shifting the whole function by a known constant value. As we considered for the clothoid to be completely vertical, we know that $\theta = 0$ (we know this because the twist sections starts from the sides and goes upwards). We have just calculated the starting point for the other two values, so now we would have to make sure that $\vec{r}(\theta_0) = P_1$.

This middle section can be defined using the equation for the in-line twist:

$$\vec{r}(\theta) = \begin{cases} x(\theta) = R \cdot \cos(\theta) \\ y(\theta) = k \cdot \theta \\ z(\theta) = R \cdot \sin(\theta) \end{cases}$$

As all energy is conserved, the number of loops is irrelevant, as the vehicle will travel through them at the same intervals and speed (thus conserving the mechanical energy). Therefore, I will only consider one and a half rotations through the loop and I'm going to calculate where that point is.

The angle at which the inline twist ends can be calculated, as we know that it depends on the number of rotations we want to complete, but it always has to be in the shape of: $\theta = n \cdot 2\pi + \pi$, where $n \in \mathbb{N}$ is the number of rotations. The half rotation at the end is to ensure that the vehicle starts towards the correct direction. If we substitute this in our equation fo the in-line twist, we can obtain the final position of this section, which we'll consider to be the initial position for the last section of the corkscrew.

Finally, we would need another clothoid going downwards to complete the corkscrew. This last clothoid reduces its curvature until the vehicle is flat again. To do this, we are going to need to inverse Euler's method. Instead of adding the corresponding slope each time, we are going to subtract it, as the second derivative depends only on the position.

We know that the vertex for this last clothoid is symmetrical to the beginning of the corkscrew with respect to the center of the twist, which is also equal to the position at which the first two sections of the corkscrew connect minus the radius of curvature. This is also the point at which the object crosses the horizontal plane again.

To add the inclination in this last clothoid, we are going to use a separate method: we can use the same formula used in the first section, but storing the

frames in the inverse order. To do this, we need a way to know how long we have to calculate this for. We can do this easily by storing the number of frames in another variable and using this as a countdown.

5.4.2 Resulting forces

The resulting forces depend on the position of the wagon at a certain moment in time. As the vehicle is constantly in a circular motion (with changing radius), most of the forces that end up on the rider are constantly changing direction.

At the beginning of the corkscrew, while the object is still flat on the ground, the normal force gets canceled out with the gravitational force. Then, as the object starts the curvature, the centripetal force appears as the sum of all unbalanced forces that make the change in direction happen. The gravitational force is always pulling the object down, while the normal force is pointing inwards, allowing for the object to change direction.

5.4.3 Animation

The code for the animation can be found Appendix III - Corkscrew section.
The link to the animation on YouTube is as follows:

6 Animations

The purpose of finding the formulas was to be able to create an animation of each of the roller coaster elements by building a simulation that numerically solves the differential equation. The complete animation, with all the elements included, can be found under the following link:

[redacted]

Following the formulas, these animations have been programmed to accurately respect speed and acceleration, even if it means that the object may slightly shift away from its path. In these cases, the shift is gradual, increasing linearly over time. This is mainly due to the expected loss of accuracy of the discretized form of the equations, as they are calculated in small increments. Animations get more accurate the smaller the increments are, but a balance between accuracy and the time needed to generate and render the figures is also important.

All the animations have been made using Blender 3D, version 2.90, which can be found here: https://www.blender.org/

Blender is an open-source app designed to create 3D figures and animations, but it can also be used to manipulate images and videos or even program games. Although the graphical user interface is probably its main strength, I have instead opted for creating the animations with Python, which is integrated in Blender, as using this programming language allows to precisely repeat the render as many times as needed.

To be able to input the formulas into Blender, instead of manually programming each frame of the animation I have used the Python API console, which is already integrated into Blender and allows to code a scene using the Python language. The main library used to program Blender using Python is the `bpy` package, which is specific to Blender and contains a lot of custom commands that can't be used in Python anywhere else. I have also used the standard Python `math` library to be able to input more complex formulas such as the trigonometric or hyperbolic ones.

The process of coding the animation is divided into two phases: initializing the editor and objects and coding the actual movement and effects. I must say that surprisingly the first part of the process is noticeably longer than the second part.

In the first section of the code, the meshes (which are the representation of an objects with physical properties using a set of finite elements) are generated and the colors and shadings are set. Most of these lines start with the `bpy` command, from the package of the same name, as these settings are specific to the blender engine and cannot be found on the standard `Python` libraries. This package has pre-programed structures to be able to manipulate the objects, materials, animations, and effects easily. However, the code is only replicating what a person would do in the user interface, so it still has limitations when manipulating several objects. Therefore, at the beginning of the code, I have added two commands that delete

every object in the scene, so I can add the correct ones again with the properties, position, and effects I want them to have.

The second section of the code expresses the actual formulas and sets the way that the objects must move and behave. Using a `for`-loop, each frame of the animation is stored after having obtained the needed calculations. For some figures, this isn't precise enough, so the program does thousands of calculations at smaller incremental intervals which don't get stored into frames. Storing and animating all these calculations between frames would also increase the size of the video and the frames per second. It would be ideal to be able to perform the calculations at differential intervals, but that is physically impossible although I try to get close. This increase in the number of calculations largely affects the speed in which the program executes. Because of this, all formulas are calculated when the code is executed, not when the animation is rendered.

Finally, the last part of the code resets the editor to its default values which may have been altered in the code execution, and also applying some additional changes in the settings, for example the number of frames per second is adjusted and the animation is reset to start at the first frame.

To be able to use Euler's method, I define all variables twice, for n and for $n+1$ by naming them `x0` and `x1` respectively. Then, I define the step size Δt using the variable `t` and setting it to a very small number. I also use this variable to convert from seconds to frames by using the frames per second (`fps`) and the number of steps in between frames (seen in the second `for`-loop and defined using the variable `j`). This is done by setting the step size as `t = 1 / (fps * j)`.

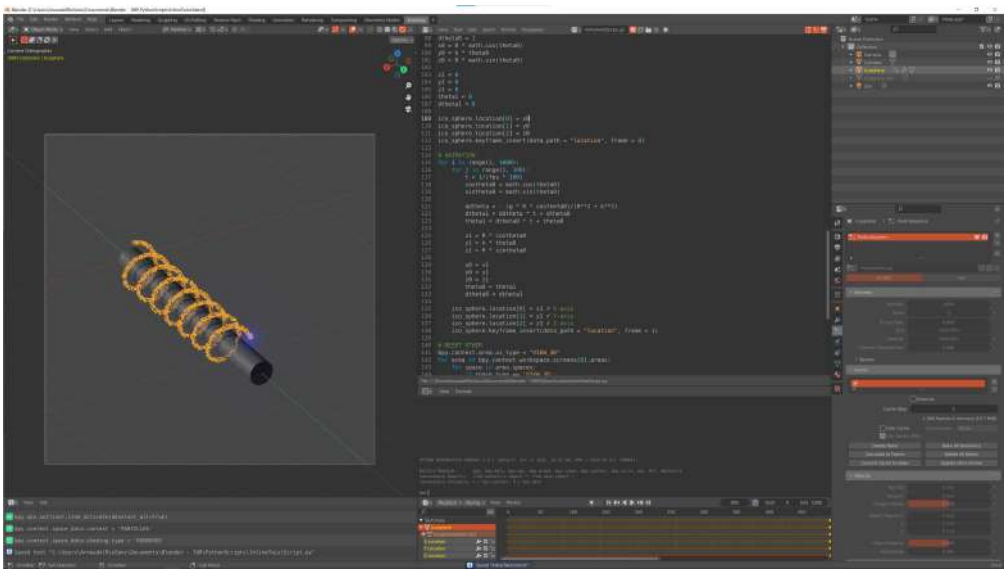


Figure 22: Capture of Blender

7 Data Collection from Real Roller Coasters

To complement this research project, I wanted to add section closer to real roller coasters. Thus, I have collected data from different roller coasters around Europe. Three roller coasters have been chosen and their elements have been analyzed. For each roller coaster, a chart has been built from the recording of the acceleration measurements of the coaster, while riding it using the `Arduino Science Journal` app. The recorded variables are:

- The acceleration in the x-axis
- The acceleration in the y-axis
- The acceleration in the z-axis
- A compass (used for referencing between the x and y axis).

The chart has been broken down in sections, each one of them corresponding to each of the elements of the coaster. The listing of the elements of the ride is shown after the chart, enumerating in the correct order which elements compose the coaster.

Also, a link to the POV[13] for each of the roller coaster included in this chapter has been added to the Bibliography. This enables the reader to be able to view by himself the elements of each roller coaster, as the video travels along them. The elements of each of the roller coasters are enumerated in its corresponding section, so it should be easy to follow along.

See Appendix II for more information on some of the most important roller coasters in Europe, some of which are further explained here.

Once the data is analyzed, the results are the different accelerations with respect to time. With this information, we can figure out the velocity, and consequently the position, at each moment (adding the initial conditions as the constants of integration). As we compare the data with the theory explained in the previous sections, we can see that the information is slightly off, and the values always tend to lower, as the force of friction must be included, and therefore, there is no conservation of energy. Despite this, the loss of energy due to friction is relatively small, so I'm not going to consider it when analyzing the data.

By using Euler's method on the numerical data, we can figure out the velocity at each moment, and therefore the position. For this, we can identify in the chart when the coaster is in the station and set the velocity and position at that point as 0.

[13] A POV is short for "Point of View", and it is the video of a ride from the rider's perspective.

7.1 Silver Star

Here is a picture of the accelerations measured on a real coaster (Silver Star Hypercoaster in Europa-Park, Rust, Germany). This roller coaster consists of a series of camelbacks, ended by a small one-turn helix.

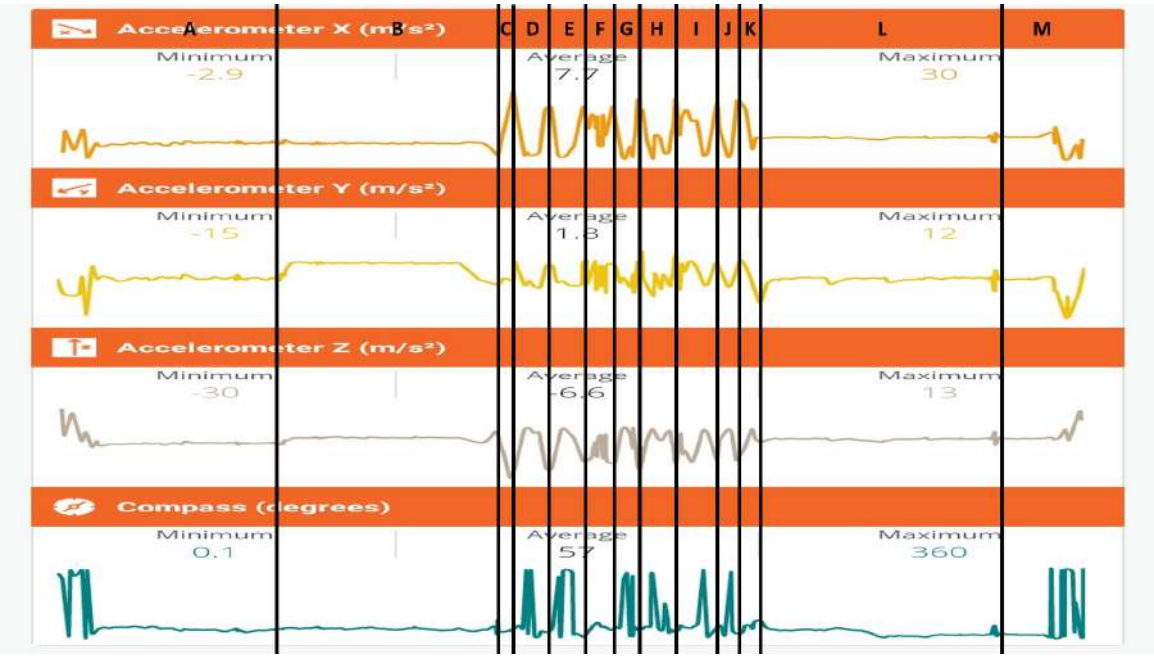


Figure 23: Acceleration of the Silver Star (01.09.2022)

A Station ***B*** Curve + Chain lift ***C*** First drop ***D*** Camelback 1 + Curve ***E*** Camelback 2 ***F*** Camelback 3 + 180º Curve ***G*** Camelback 4 ***H*** Camelback 5 + Security zone ***I*** Curve + Small drop ***J*** Camelback 6 ***K*** Double curve + Deceleration zone ***L*** Curve + Waiting zone ***M*** Station

Figure 24: Picture of the whole Silver Star

7.2 Python

The following charts corresponds to the Python roller coaster in Efteling, Netherlands (do not confuse with the Python language which has been mentioned previously). This roller coaster has an initial lift hill, followed by two vertical loops and two corkscrews. At the end, right before getting to the station, there is a small helix.

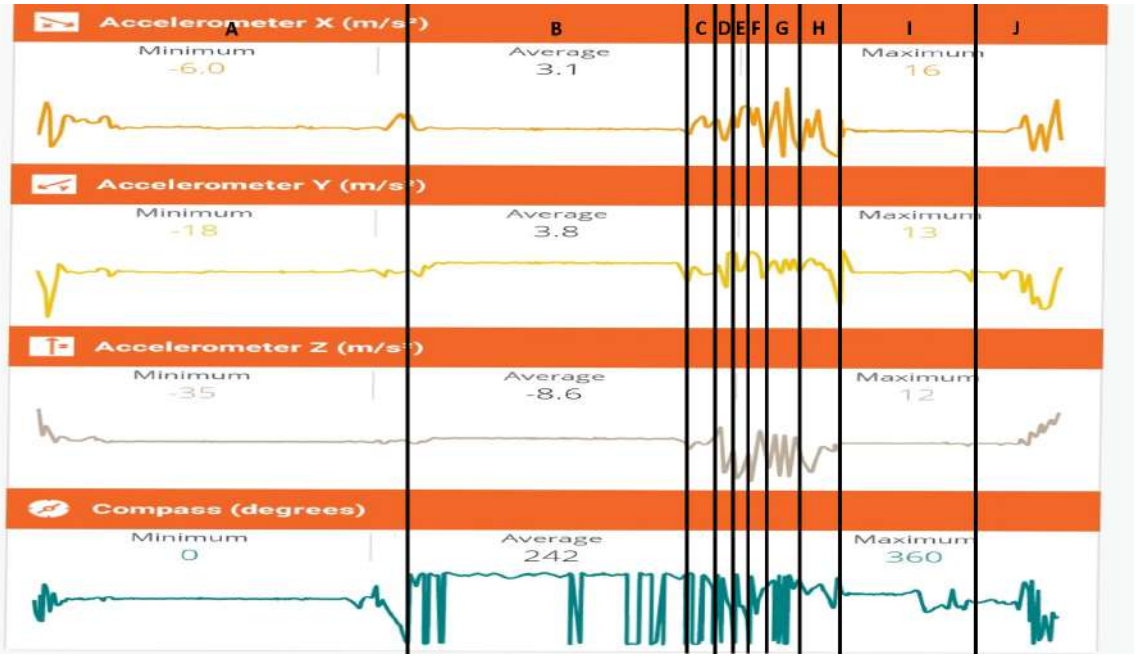


Figure 25: Acceleration of the Python (24.08.2022)

A Station ***B*** Chain lift ***C*** Small drop + Curve ***D*** Main drop ***E*** Loop 1 ***F*** Loop 2 ***G*** Curve + Double corkscrew ***H*** Helix ***I*** Deceleration zone ***J*** Station

Figure 26: Picture of the Python

7.3 Baron 1898

The following charts are the capture of the acceleractions measured in the Baron 1898 roller coaster in Efteling, Netherlands. This is a small roller coaster, but at the same time it has several of the elements that have been analyzed in this report, including a loop, an in-line twist and a helix.

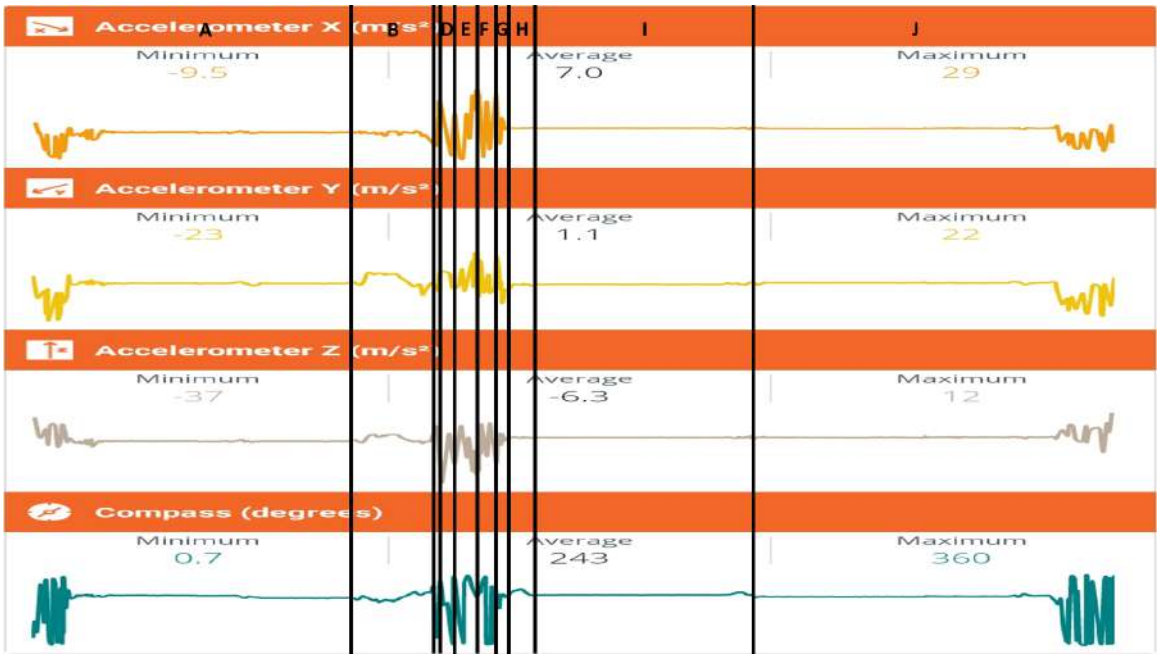


Figure 27: Acceleration of the Baron 1898 (24.08.2022)

A Station ***B*** Chain lift + Stop ***C*** Main drop ***D*** Loop ***E*** In-line Twist in a small camelback ***F*** Helix ***G*** Small camelback ***H*** Curve***I*** Deceleration zone ***J*** Station

Figure 28: Picture of the Baron 1898

8 Conclusions

This report shows the mathematical work needed to build the animations of the most common roller coaster elements.

Even though a lot of simplifications and assumptions have been made, the mathematical formulas obtained are more complex than I had initially thought, considering that in the end they simply calculate what happens to an object along a known path given the total forces applied. I can't imagine the struggle that would be doing this same project but considering for example the force of friction and the wind drag. At several points in the making of this project I tried simplifying even further or making specific assumptions which made the problems look easier, but even that resulted sometimes in a challenge.

Throughout the project, I have tried to find the elements that could most likely be solved in an analytical way, but all the formulas quickly turned into elaborate differential equations. However, I was able to solve all these equations by using a wide range of methods.

At some points, I had to explore different paths to mathematically try and solve the differential equations, and I found it strange that each shape presented a specific and complex problem that some of the other shapes didn't have. To help me solve each problem, I used complementary tools like `Geogebra` or `WolframAlpha` to check the formulas visually.

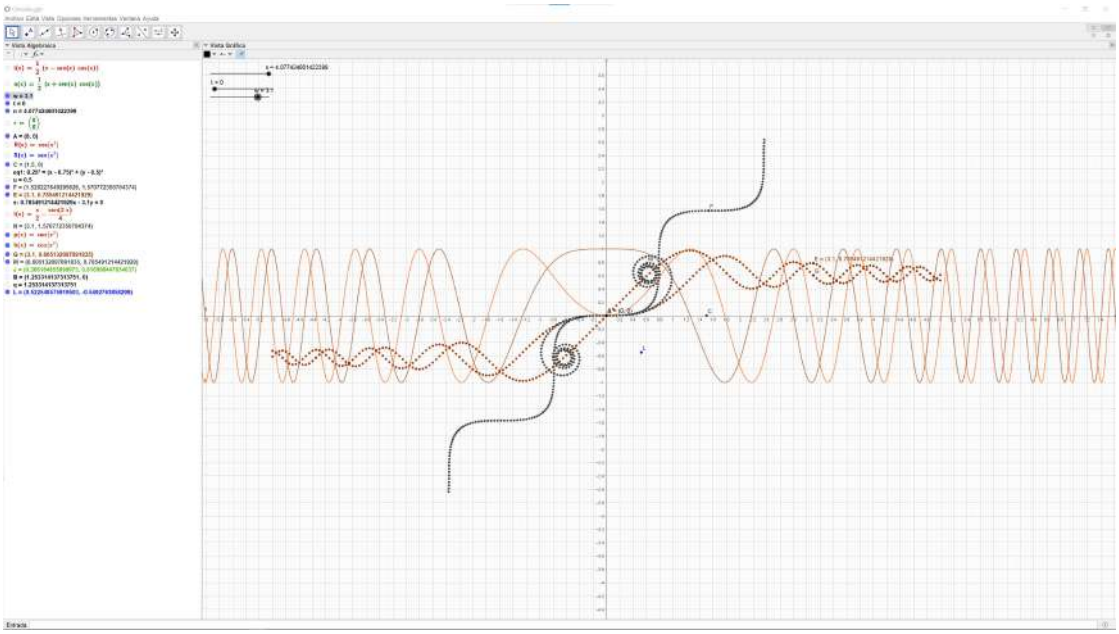


Figure 29: Capture of GeoGebra drawing different curves related to the clothoid

Finally, to review specifically the elements that this report covers:

- **Curve**: The curve was an easy element as the forces got neglected with each other.
- **Camelback (Parabola)**: The camelback was a first challenge as I had to learn how to solve differential equations by using Euler's method.
- **Catenary**: This element was quite easy to solve, as it was similar to the parabola.
- **Helix**: This was the first element I solved. Luckily, the differential equation ended up being the easiest one.
- **In-line twist**: This element was surprisingly similar to the vertical loop, so I ended up solving it first.
- **Vertical Loop**: When starting this element, I thought that having a parameter wouldn't complicate the resolution, but I was wrong. I ended up using a whole different approach than I had imagined to solve it.
- **Corkscrew**: Once I had the animation for the in-line twist and the vertical loop, I just had to add them together. Finding the points of intersection and the curvature there wasn't difficult, as I already knew some conditions about those points.

As I had already used Blender before, I was already familiar with the way it worked, but I hadn't experimented with the integrated Python console yet. Translating and finding each command was the hardest part about using python to reproduce certain animations.

I didn't think that it would take me so much time to complete this project, but it has really broadened by vision towards mathematics and the different branches of it, especially in differential calculus. I have actually learned a lot about geometrical analysis and the way that derivatives and integrals are used and solved in a certain situation.

Beyond the mathematical aspects, I have been able to use and apply new tools and knowledge that I am sure will help me in other areas of my career: using LaTeXand TeXstudio to build the document, using Python for programming or storing the document versions and the project website in GitHub. I had already been using Blender before starting this project, but I had never used the Python interface which is something I also had to learn.

In conclusion, this project has been a hard but pleasant journey to applying math to roller coasters. Journeys like mine have allowed engineers to improve the

experience and safety of roller coasters. Thanks to math nowadays riders don't faint or die! I hope my work to be a small contribution that may help others reach further in this amazing field.

Overall, I'd like to end stating that I really have enjoyed working on this project.

9 Bibliography and References

Books and Articles

[3] Nick Weisenberger. *Coasters 101: An Engineer's Guide to Roller Coaster Design*. 3rd ed. 2012.

[4] Raph Levien. "The Euler spiral: a mathematical history". In: (2008). URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-111.html.

[5] Dourmashkin Kleppner and Ramsey. *Quick Calculus: A Self-Teaching Guide*.

[6] Olivia Briggs. "Roller Coaster Acceleration". In: (2021). URL: https://vc.bridgew.edu/honors_proj/459.

Math Forums

[7] *Mathematics stackexchange*. URL: https://math.stackexchange.com.

[8] *Mathoverflow*. URL: https://mathoverflow.net.

Math Forums Questions

[9] Lutz Lehman (answer) [redacted] (question). *Doubts on the solution of a differential equation*. URL: https://math.stackexchange.com/questions/4597447/doubts-on-the-solution-of-a-differential-equation.

[18] Michael Seifert (answer) Archimedes (question). *Solving a second-order and second-degree differential equation*. 2018. URL: https://math.stackexchange.com/questions/2378818/how-to-solve-a-second-order-and-second-degree-differential-equation.

[19] Jake (answer) skvery (question). *Can PGF plot the integral of any specified function?* 2013. URL: https://tex.stackexchange.com/questions/103569/can-pgf-plot-the-integral-of-any-specified-function.

Videos

[2] Art of Engineering. *The Real Physics of Roller Coaster Loops*. 2019. URL: https://youtu.be/4q2W5SJc5j4.

[34] MasterWuMathematics. *Integrating the Fresnel integrals*. 2014. URL: https://youtu.be/fR4yd6pB5co.

[35] Matt Anderson. *Professor Matt Anderson's YouTube Channel*. URL: https://www.youtube.com/@yoprofmatt.

[36] *Silver Star POV On-Ride*. URL: https://www.youtube.com/watch?v=fFmuCmku4Z0.

[37] *Python POV On-Ride*. URL: https://www.youtube.com/watch?v=j74EvfW2QiU.

[38] *Baron 1898 POV On-Ride*. URL: https://www.youtube.com/watch?v=dCGkm_LpRGM.

Internet References

[1] Nancy Hall. *Newton's Laws of Motion*. URL: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/newtons-laws-of-motion/.

[10] *Outline of Calculus, Wikipedia (disambiguation)*. URL: https://en.wikipedia.org/wiki/Outline_of_calculus.

[11] Paul Dawkins. *Parametric Equations And Curves*. 2022. URL: https://tutorial.math.lamar.edu/classes/calcii/parametriceqn.aspx.

[12] *CoasterPedia.net*. 2009. URL: https://coasterpedia.net/wiki/.

[13] *Roller Coaster element*. 2022. URL: https://en.wikipedia.org/wiki/Roller_coaster_element.

[14] *Geogebra*. URL: https://www.geogebra.org/.

[15] *Roller Coaster Data Base*. URL: https://www.rcdb.com/.

[16] *WolframAlpha*. URL: https://www.wolframalpha.com/.

[17] *Second Order Differential Equations*. 2021. URL: https://www.mathsisfun.com/calculus/differential-equations-second-order.html.

[20] *OverLeaf Documentation LaTeX*. URL: https://www.overleaf.com/learn.

[21] Erik Neumann. *Classifying Differential Equations*. 2004. URL: https://www.myphysicslab.com/explain/classify-diff-eq-en.html.

[22] *Mathematics of Circular Motion*. 1996. URL: https://www.physicsclassroom.com/class/circles/Lesson-1/Mathematics-of-Circular-Motion.

[23] Wikipedia. *Conservation of Energy Principle*. URL: https://en.wikipedia.org/wiki/Conservation_of_energy.

[24] *Energy*. URL: https://www.physicsclassroom.com/class/energy.

[25] Wikipedia. *Euler's method*. URL: https://en.wikipedia.org/wiki/Euler_method.

[26] Alejandro Blumentals. *Computing Clothoid Curves in five lines of Python*. 2019. URL: https://medium.com/@alejandro.blumentals/computing-clothoid-curves-in-five-lines-of-python-3ea762debaa1.

[27] Nick Berry. *Why Roller Coaster Loops Are Never Circular*. 2014. URL: https://gizmodo.com/why-roller-coaster-loops-are-never-circular-1549063718.

[28] Alicia Cypress. *A brief history of the roller coaster*. 1997. URL: https://www.washingtonpost.com/archive/1997/08/13/a-brief-history-of-the-roller-coaster/4490a0f9-6a82-451d-86b7-f36a7bc0fbbf/.

[29] *Roller Coaster Loop Shape*. URL: http://physics.gu.se/LISEBERG/eng/loop_pe.html.

[30] *Europa Park*. URL: https//www.europapark.de/en/park.

[31] Efteling. *Efteling*. 2015. URL: https://www.efteling.com/en.

[32] PortAventura. *PortAventura*. URL: https://www.portaventuraworld.com/ca.

[33] Tibidabo. *Tibidabo*. URL: https://www.tibidabo.cat/ca.

10 Appendix I - How this document was made

This document has been formatted with LaTeXusing TeXstudio, including all formulas and charts. The LaTeXformat is considered the "standard" for any math-related document. Much like Python, LaTeXuses libraries to add different specific functionalities, such as adding graphs or changing the text style.

One of the most useful packages has been the `Pythonhighlight` package. This package has been used to show the Python syntax using text that resembles code. The code shown in the Appendix III was imported directly from the file generated by blender using the `\inputpyhon` command.

Another useful library was used to insert charts and images. The charts were displayed using two libraries: `pgfplots` (mainly) and `pstricks` (for more complex formulas like the Vertical Loop).

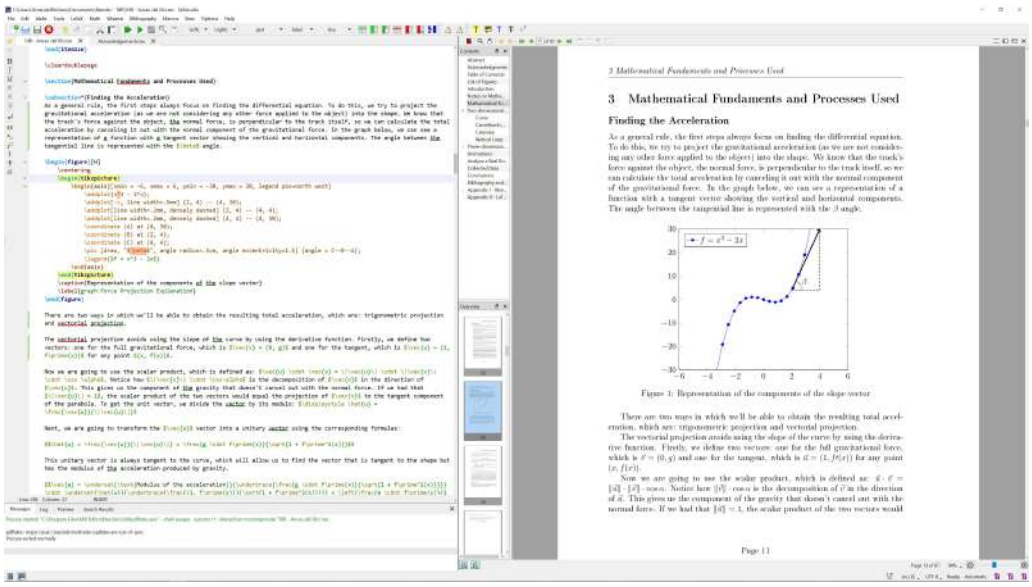


Figure 30: Capture of the editor used

As I said earlier, this document can also be accessed online at the following address: [redacted]. This website is made using GitHub pages, and the corresponding LaTeXsource code can also be found in the same repository, along with the Python source code.

11 Appendix II - Roller Coasters I Rode while Making this Research Project

Here is the main information on some of the most important roller coasters in Europe. These tables include the name of the roller coaster, the manufacturer, the model (as described by the manufacturer), the scale (which indicates the level), the year of opening and the number of times I've ridden it since the beginning of the project.

The main manufacturers are the following (Arrow Dynamics and Custom Coasters International are now defunct, and have mainly been bought by S&S):

Table 1: Manufacturers

Name	Abbreviation	Country	Web-page
Arrow Dynamics	Arrow	United States	https://www.s-s.com/
Mack Rides	Mack Rides	Germany	https://mack-rides.com/
Vekoma	Vekoma	Netherlands	https://www.vekoma.com/
Great Coasters International	GCI	United States	https://greatcoasters.com/
Custom Coasters International	CCI	United States	https://www.s-s.com/
Bolliger & Mabillard	B&M	Switzerland	https://www.bolliger-mabillard.com/
Sansei Technologies	S&S	United States	https://www.s-s.com/
KumbaK	KumbaK	Netherlands	https://www.kumbak.nl/
ART Engineering GmbH	ART	Germany	https://artengineering.de/
Intamin Amusement Rides	Intamin	Switzerland	https://www.intamin.com/
Zamperla	Zamperla	Italy	https://www.zamperla.com/

Roller Coasters

Table 2: Efteling Roller Coasters

Name	Manufacturer	Model	Scale	Opened	#
Baron 1898	B&M	Floorless	Extreme	2015	2
Joris en de Draak	GCI	Wood Twin	Thrill	2010	1
Max + Moritz	Mack Rides	Twin	Family	2020	2
Python	Vekoma	Sit Down	Extreme	1981	2
Vliegende Hollander	KumbaK	Dark Ride	Thrill	2007	1
Vogel Rok	Vekoma	Enclosed	Thrill	1998	1

Table 3: Europa Park Roller Coasters

Name	Manufacturer	Model	Scale	Opened	#
Alpenexpress Enzian	Mack Rides	Grottenblitz	Family	1984	1
Arthur	Mack Rides	Inverted	Family	2014	1
Atlantica SuperSplash	Mack Rides	Water Coaster	Thrill	2005	1
Ba-a-a-Express	ART	Oval	Kiddie	2016	1
Blue Fire Megacoaster	Mack Rides	Launch Coaster	Extreme	2009	1
Euro Mir	Mack Rides	Spinning Coaster	Thrill	1997	2
Eurosat - CanCan Coaster	Mack Rides	Enclosed Coaster	Thrill	1989	3
Matterhorn Blitz	Mack Rides	Wild Mouse	Thrill	1999	3
Pegasus	Mack Rides	Youngstar Coaster	Family	2006	2
Poseidon	Mack Rides	Water Coaster	Thrill	2000	2
Schweizer Bobbahn	Mack Rides	Bobsled	Thrill	1985	4
Silver Star	B&M	Hypercoaster	Extreme	2002	5
Wodan Timbur Coaster	GCI	Wood	Extreme	2012	1

Table 4: Phantasialand Roller Coasters

Name	Manufacturer	Model	Scale	Opened	#
Black Mamba	B&M	Inverted	Extreme	2006	1
Colorado Adventure	Vekoma	Mine Train	Thrill	1996	1
Crazy Bats	Vekoma	Enclosed	Thrill	1988	1
F.L.Y.	Vekoma	Flying	Extreme	2020	2
Raik	Vekoma	Boomerang	Family	2016	1
Taron	Intamin	Launch Coaster	Extreme	2016	1

Table 5: PortAventura Park Roller Coasters

Name	Manufacturer	Model	Scale	Opened	#
Tren de la Mina	Arrow	Mine Train	Thrill	1995	1
Dragon Khan	B&M	Sitting Coaster	Extreme	1995	1
Furius Baco	Intamin	Wing Launch Coaster	Extreme	2007	1
Shambala	B&M	Hypercoaster	Extreme	2012	1
Stampida	CCI	Wood - Twin	Thrill	1997	1
Tami-Tami	Vekoma	Junior Coaster	Family	1998	1
Tomahawk	CCI	Wood	Family	1997	1

Table 6: Tibidabo Roller Coasters

Name	Manufacturer	Model	Scale	Opened	#
Muntanya Russa	Vekoma	Terrain Custom	Thrill	2008	1
Tibidabo Express	Zamperla	Powered Coaster	Family	1990	1

12 Appendix III - Blender source code

Here is the code used to generate the animations. This code can also be accessed through GitHub on the same repository as the web-page:

[redacted]

The code for the Blender files is as follows (written in Python):

12.1 Curve Animation

```
#----------------------------------------------
#
# DOCUMENT Blender_circle_animation.blend
# AUTHOR   (Anonymized)
# DATE     20-09-2022
#
#----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                          location=(0, 0, 35), rotation=(0,
                           0, 0), scale=(1, 1, 1))

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                          location=(0, 0, 0), scale=(1, 1,
                          1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821

# FLOOR
bpy.ops.mesh.primitive_plane_add(size=25)

# FLOOR COLOR
mat_name = "Base"
bpy.ops.material.new()
```

```
bpy.data.materials[-1].name = mat_name
mat = bpy.data.materials.get(mat_name)
mat.use_nodes = False
bpy.data.materials[mat_name].diffuse_color = (0.1, 0.1, 0.1, 1)
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# SPHERE
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                                      False, location=(0, 0, 10))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                                      ShaderNodeEmission")
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                      Color"].default_value = (0, 0, 1,
                                       1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                      Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                      ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                          outputs[0],
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                          ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
bpy.ops.object.particle_system_add()

bpy.data.particles["ParticleSettings"].name = "ParticleSettings"
bpy.data.particles["ParticleSettings"].type = 'EMITTER'
bpy.data.particles["ParticleSettings"].frame_end = 200
bpy.data.particles["ParticleSettings"].mass = 0
bpy.data.particles["ParticleSettings"].lifetime = 300
bpy.data.particles["ParticleSettings"].normal_factor = 0
if(bpy.data.particles["ParticleSettings.001"]):
```

```
    bpy.ops.object.particle_settings_remove()


# VARIABLES
w = 10
phi0 = 0
R = 5

# ANIMATION
for i in range(251):
    t = i/60
    phi = phi0 + w * t

    ico_sphere.location[0] = R * math.cos(phi) # X-axis
    ico_sphere.location[1] = R * math.sin(phi) # Y-axis
    ico_sphere.location[2] = 2
    ico_sphere.keyframe_insert(data_path = "location", frame = i)

# RESET OTHER
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = 30
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```

12.2 Camelback Animation

```
#-----------------------------------------------
#
# DOCUMENT Blender_camelback_animation.blend
# AUTHOR   (Anonymized)
# DATE     04-11-2022
#
#-----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.data.objects["Icosphere"].select_set(True)
bpy.ops.object.delete(use_global=False)
bpy.data.objects["Camera"].select_set(True)
bpy.ops.object.delete(use_global=False)
bpy.data.objects["Sun"].select_set(True)
bpy.ops.object.delete(use_global=False)
bpy.data.objects["BezierCurve"].select_set(True)
bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                          location=(0, 100, 10), rotation=(
                          math.radians(90), 0, math.radians
                          (180)), scale=(1, 1, 1))
bpy.context.object.data.type = 'ORTHO'
bpy.context.object.data.ortho_scale = 60

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                          location=(0, 0, 0), scale=(1, 1,
                          1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821

# SPHERE
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                          False, location=(0, 0, 10))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
```

```
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeEmission")
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Color"].default_value = (0, 0, 1,
                                     1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                        outputs[0],
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                        ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
bpy.ops.object.particle_system_add()
bpy.data.particles["ParticleSettings"].name = "ParticleSettings"
bpy.data.particles["ParticleSettings"].type = 'EMITTER'
bpy.data.particles["ParticleSettings"].frame_end = 200
bpy.data.particles["ParticleSettings"].effector_weights.gravity =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.all = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.force = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.magnetic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.harmonic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.charge = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    lennardjones = 0
bpy.data.particles["ParticleSettings"].effector_weights.wind = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    curve_guide = 0
bpy.data.particles["ParticleSettings"].effector_weights.texture =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.smokeflow
                                    = 0
```

```
bpy.data.particles["ParticleSettings"].effector_weights.turbulence
                                        = 0
bpy.data.particles["ParticleSettings"].effector_weights.drag = 0
bpy.data.particles["ParticleSettings"].effector_weights.boid = 0
bpy.data.particles["ParticleSettings"].lifetime = 300
bpy.data.particles["ParticleSettings"].normal_factor = 0

# VARIABLES
fps = 30
g = -9.8
a = 0.05
x = 20
v0 = 0

x0 = x
y0 = a * x**2
vx0 = v0
vy0 = 2*a*vx0
x1 = 0
y1 = 0
vx1 = 0
vy1 = 0

ico_sphere.location[0] = x0
ico_sphere.location[2] = y0

# ANIMATION
for i in range(1, 2001):
    for j in range(1, 10):
        t = 1/(fps * 10)
        vx1 = vx0 + ((2*a*x0*g)/(1 + 4*a**2*x0**2)) * t
        vy1 = vy0 + ((4*a*y0*g)/(1 + 4*a*y0)) * t
        x1 = x0 + vx0 * t
        y1 = y0 + vy0 * t
        x0 = x1
        y0 = y1
        vx0 = vx1
        vy0 = vy1

    ico_sphere.location[0] = x1 # X-axis
    ico_sphere.location[2] = a * x1**2 # Y-axis
    ico_sphere.location[2] = y1 # Y-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = i)

# SHAPE
number_of_points = 5
coords = [(-x, a * x**2, 0), (-x * 4/5, a * (x * 4/5)**2, 0), (-x
                                * 3/5, a * (x * 3/5)**2, 0), (-x
                                * 2/5, a * (x * 2/5)**2, 0), (-x/
```

```
                                    5, a * (x/5)**2, 0), (0, 0, 0), (
                                    x/5, a * (x/5)**2, 0), (x * 2/5,
                                    a * (x * 2/5)**2, 0), (x * 3/5, a
                                     * (x * 3/5)**2, 0), (-x * 4/5, a
                                     * (x * 4/5)**2, 0), (x, a * x**2
                                    , 0)]

#curveData = bpy.data.curves.new('myCurve', type='CURVE')
#curveData.dimensions = '3D'
#curveData.resolution_u = 2

#polyline = curveData.splines.new('BEZIER')
#polyline.bezier_points.add(len(coords))
#for i, coord in enumerate(coords):
#    x,y,z = coord
#    polyline.bezier_points[i].co = (x, y, z, 1)

# curveOB = bpy.data.objects.new('myCurve', curveData)

bpy.ops.curve.primitive_bezier_curve_add(enter_editmode=False,
                                    align='WORLD', location=(0, -2, 0
                                    ), rotation=(math.radians(90), 0,
                                     0))

my_curve = bpy.context.active_object

bpy.ops.object.editmode_toggle()
bpy.ops.transform.translate(value=(-1, -0, -0), orient_axis_ortho=
                                    'X', orient_type='GLOBAL',
                                    orient_matrix=((1, 0, 0), (0, 1,
                                    0), (0, 0, 1)),
                                    orient_matrix_type='GLOBAL',
                                    mirror=False,
                                    use_proportional_edit=True,
                                    proportional_edit_falloff='SMOOTH
                                    ', proportional_size=1,
                                    use_proportional_connected=False,
                                     use_proportional_projected=False
                                    )

#bpy.ops.curve.subdivide(number_cuts=number_of_points)
#my_curve.data.splines[0].bezier_points[3].select_control_point =
                                    True

bpy.ops.object.editmode_toggle()

# RESET OTHER
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
```

```
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = fps
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```

12.3 Catenary Animation

```
#-----------------------------------------------
#
# DOCUMENT Blender_catenary_animation.blend
# AUTHOR   (Anonymized)
# DATE     04-11-2022
#
#-----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.data.objects["Icosphere"].select_set(True)
bpy.ops.object.delete(use_global=False)
bpy.data.objects["Camera"].select_set(True)
bpy.ops.object.delete(use_global=False)
bpy.data.objects["Sun"].select_set(True)
bpy.ops.object.delete(use_global=False)
#bpy.data.objects["BezierCurve"].select_set(True)
#bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                          location=(0, 100, 20), rotation=(
                          math.radians(90), 0, math.radians
                          (180)), scale=(1, 1, 1))
bpy.context.object.data.type = 'ORTHO'
bpy.context.object.data.ortho_scale = 60

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                          location=(0, 0, 0), scale=(1, 1,
                          1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821

# SPHERE
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                          False, location=(0, 0, 10))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
```

```
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeEmission")
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Color"].default_value = (0, 0, 1,
                                     1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                        outputs[0],
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                        ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
bpy.ops.object.particle_system_add()
bpy.data.particles["ParticleSettings"].name = "ParticleSettings"
bpy.data.particles["ParticleSettings"].type = 'EMITTER'
bpy.data.particles["ParticleSettings"].frame_end = 800
bpy.data.particles["ParticleSettings"].effector_weights.gravity =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.all = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.force = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.magnetic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.harmonic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.charge = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    lennardjones = 0
bpy.data.particles["ParticleSettings"].effector_weights.wind = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    curve_guide = 0
bpy.data.particles["ParticleSettings"].effector_weights.texture =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.smokeflow
                                    = 0
```

```
bpy.data.particles["ParticleSettings"].effector_weights.turbulence
                                        = 0
bpy.data.particles["ParticleSettings"].effector_weights.drag = 0
bpy.data.particles["ParticleSettings"].effector_weights.boid = 0
bpy.data.particles["ParticleSettings"].lifetime = 800
bpy.data.particles["ParticleSettings"].normal_factor = 0
bpy.data.particles["ParitcleSettings"].mass = 0

# VARIABLES
fps = 30
g = -9.8
a = 10
x = 20
v0 = 0

x0 = x
y0 = a * math.cosh(x0/a)
vx0 = v0
vy0 = math.sinh(x/a)
x1 = 0
y1 = 0
vx1 = 0
vy1 = 0

ico_sphere.location[0] = x0
ico_sphere.location[2] = y0

# ANIMATION
for i in range (1, 2001):
    for j in range(1, 10):
        t = 1/(fps * 10)
        print(g*math.sinh(x0/a))
        vx1 = vx0 + ((g * math.sinh(x0/a))/(1 + math.sinh(x0/a)**2
                                            )) * t
        vy1 = vy0 + ((g * math.sinh(x0/a)**2)/(1 + math.sinh(x0/a)
                                            **2)) * t
        print(vx1)
        print(vy1)
        print(math.cosh(x1/a))
        print("--------------------------------------")
        x1 = x0 + vx0 * t
        y1 = y0 + vy0 * t
        x0 = x1
        y0 = y1
        vx0 = vx1
        vy0 = vy1

    ico_sphere.location[0] = x1 # X-axis
    ico_sphere.location[2] = a * math.cosh(x1/a) # Y-axis
```

```
#     ico_sphere.location[2] = y1 # Y-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = i)

# SHAPE
number_of_points = 5
coords = [(-x, a * x**2, 0), (-x * 4/5, a * (x * 4/5)**2, 0), (-x
                                * 3/5, a * (x * 3/5)**2, 0), (-x
                                * 2/5, a * (x * 2/5)**2, 0), (-x/
                                5, a * (x/5)**2, 0), (0, 0, 0), (
                                x/5, a * (x/5)**2, 0), (x * 2/5,
                                a * (x * 2/5)**2, 0), (x * 3/5, a
                                 * (x * 3/5)**2, 0), (-x * 4/5, a
                                 * (x * 4/5)**2, 0), (x, a * x**2
                                , 0)]

#curveData = bpy.data.curves.new('myCurve', type='CURVE')
#curveData.dimensions = '3D'
#curveData.resolution_u = 2

#polyline = curveData.splines.new('BEZIER')
#polyline.bezier_points.add(len(coords))
#for i, coord in enumerate(coords):
#    x,y,z = coord
#    polyline.bezier_points[i].co = (x, y, z, 1)

# curveOB = bpy.data.objects.new('myCurve', curveData)

bpy.ops.curve.primitive_bezier_curve_add(enter_editmode=False,
                                align='WORLD', location=(0, -2, 0
                                ), rotation=(math.radians(90), 0,
                                 0))

my_curve = bpy.context.active_object

bpy.ops.object.editmode_toggle()
bpy.ops.transform.translate(value=(-1, -0, -0), orient_axis_ortho=
                                'X', orient_type='GLOBAL',
                                orient_matrix=((1, 0, 0), (0, 1,
                                0), (0, 0, 1)),
                                orient_matrix_type='GLOBAL',
                                mirror=False,
                                use_proportional_edit=True,
                                proportional_edit_falloff='SMOOTH
                                ', proportional_size=1,
                                use_proportional_connected=False,
                                 use_proportional_projected=False
                                )

#bpy.ops.curve.subdivide(number_cuts=number_of_points)
```

```
#my_curve.data.splines[0].bezier_points[3].select_control_point =
                                        True

bpy.ops.object.editmode_toggle()

# RESET OTHER
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = fps
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```

12.4 Helix Animation

```
#----------------------------------------------
#
# DOCUMENT Blender_helix_animation.blend
# AUTHOR   (Anonymized)
# DATE     02-08-2022
#
#----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                          location=(40, -40, 35), rotation=
                          (1.22, -0, 0.785), scale=(1, 1, 1
                          ))

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                          location=(0, 0, 0), scale=(1, 1,
                          1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821


# CILINDER
bpy.ops.mesh.primitive_cube_add(size=1.5, enter_editmode=False,
                          align='WORLD', location=(0, 0, 0)
                          , scale=(1, 1, 1))
bpy.ops.object.modifier_add(type='SCREW')
bpy.context.object.modifiers["Screw"].screw_offset = 25
bpy.context.object.modifiers["Screw"].iterations = 1
bpy.context.object.modifiers["Screw"].angle = -12.566371
bpy.context.object.modifiers["Screw"].steps = 32
bpy.context.object.modifiers["Screw"].render_steps = 32
bpy.ops.object.modifier_apply(modifier="Screw")

# CILINDER COLOR
```

```
mat_name = "Material"
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
mat = bpy.data.materials.get(mat_name)
mat.use_nodes = False
bpy.data.materials[mat_name].diffuse_color = (0.1, 0.1, 0.1, 1)
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# FLOOR
bpy.ops.mesh.primitive_plane_add(size=25)

# FLOOR COLOR
mat_name = "Base"
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
mat = bpy.data.materials.get(mat_name)
mat.use_nodes = False
bpy.data.materials[mat_name].diffuse_color = (0.1, 0.1, 0.1, 1)
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# SPHERE
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                                      False, location=(0, 0, 10))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                                      ShaderNodeEmission")
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                      Color"].default_value = (0, 0, 1,
                                       1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                      Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                      ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                          outputs[0],
```

```
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                        ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
bpy.ops.object.particle_system_add()
bpy.data.particles["ParticleSettings"].name = "ParticleSettings"
bpy.data.particles["ParticleSettings"].type = 'EMITTER'
bpy.data.particles["ParticleSettings"].frame_end = 200
bpy.data.particles["ParticleSettings"].effector_weights.gravity =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.all = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.force = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.magnetic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.harmonic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.charge = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    lennardjones = 0
bpy.data.particles["ParticleSettings"].effector_weights.wind = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    curve_guide = 0
bpy.data.particles["ParticleSettings"].effector_weights.texture =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.smokeflow
                                    = 0
bpy.data.particles["ParticleSettings"].effector_weights.turbulence
                                     = 0
bpy.data.particles["ParticleSettings"].effector_weights.drag = 0
bpy.data.particles["ParticleSettings"].effector_weights.boid = 0
bpy.data.particles["ParticleSettings"].lifetime = 300
bpy.data.particles["ParticleSettings"].normal_factor = 0


# VARIABLES
g = -9.8
R = 2
p = 0.1
v0 = 0
z0 = 25
theta = (g * p)/(2 * R**2 + 2 * p**2)

# ANIMATION
```

```
for i in range(251):
    t = i/15
    ico_sphere.location[0] = R * math.cos(theta * t**2 + v0 * t +
                                          z0) # X-axis
    ico_sphere.location[1] = R * math.sin(theta * t**2 + v0 * t +
                                          z0) # Y-axis
#    ico_sphere.location[2] = 25 + (-9.8 * ((i/15)/(2 * math.pi))
                                   **2 / 2) # Z-axis
#    ico_sphere.location[2] = 25 + (g * math.sqrt(R**2 + k**2) * (
                                   i/15)**2)/(k*R*2) # Z-axis
#    ico_sphere.location[2] = 25 - (i/15) / R
    ico_sphere.location[2] = theta * t**2 + v0 * t + z0
    ico_sphere.keyframe_insert(data_path = "location", frame = i)

# RESET OTHER
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = 30
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```

12.5 In-line Twist Animation

```
#----------------------------------------------
#
# DOCUMENT Blender_in-line_twist_animation.blend
# AUTHOR   (Anonymized)
# DATE     04-11-2022
#
#----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                                location=(160, 350, 200),
                                rotation=(math.radians(60), 0,
                                math.radians(150)), scale=(1, 1,
                                1))
bpy.context.object.data.type = 'ORTHO'
bpy.context.object.data.ortho_scale = 150

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                                location=(0, 0, 0), scale=(1, 1,
                                1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821

# CILINDER
bpy.ops.mesh.primitive_cylinder_add(radius=2, depth=120,
                                enter_editmode=False, align='
                                WORLD', location=(0, 60, 0),
                                rotation=(1.5708, 0, 0), scale=(3
                                , 1, 3))

# CILINDER COLOR
bpy.ops.material.new()
bpy.data.materials["Material"].node_tree.nodes["Principled BSDF"].
                                inputs[0].default_value = (0.1, 0
```

```
                                    .1, 0.1, 1)

# SPHERE
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                                    False, location=(0, 0, 10), scale
                                    =(2, 2, 2))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeEmission")
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Color"].default_value = (0, 0, 1,
                                     1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                        outputs[0],
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                        ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
#C = bpy.data.objects['Icosphere']
#ps = C.particle_systems['ParticleSystem'].settings
bpy.ops.object.particle_system_add()
bpy.ops.object.ParticleSettings = "ParticleSettings"
#if not 'ParticleSettings':
if True:
    bpy.data.particles["ParticleSettings"].name = "
                                        ParticleSettings"
    bpy.data.particles["ParticleSettings"].type = 'EMITTER'
    bpy.data.particles["ParticleSettings"].frame_end = 1000
    bpy.data.particles["ParticleSettings"].effector_weights.
                                        gravity = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.all =
                                         0
```

```
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        vortex = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.force
                                         = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        vortex = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        magnetic = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        harmonic = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        charge = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        lennardjones = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.wind
                                        = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        curve_guide = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        texture = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        smokeflow = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.
                                        turbulence = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.drag
                                        = 0
    #bpy.data.particles["ParticleSettings"].effector_weights.boid
                                        = 0
    bpy.data.particles["ParticleSettings"].lifetime = 1000
    bpy.data.particles["ParticleSettings"].normal_factor = 0
    bpy.data.particles["ParticleSettings"].mass = 0

# VARIABLES
fps = 30
g = 9.8
R = 10
k = 2
ddtheta = 0

theta0 = math.radians(90)
dtheta0 = 1
x0 = R * math.cos(theta0)
y0 = k * theta0
z0 = R * math.sin(theta0)

x1 = 0
y1 = 0
z1 = 0
theta1 = 0
```

```
dtheta1 = 0

ico_sphere.location[0] = x0
ico_sphere.location[1] = y0
ico_sphere.location[2] = z0
ico_sphere.keyframe_insert(data_path = "location", frame = 0)

# ANIMATION
for i in range(1, 1000):
    for j in range(1, 100):
        t = 1/(fps * 100)
        costheta0 = math.cos(theta0)
        sintheta0 = math.sin(theta0)

        ddtheta = - (g * R * costheta0)/(R**2 + k**2)
        dtheta1 = ddtheta * t + dtheta0
        theta1 = dtheta0 * t + theta0

        x1 = R * costheta0
        y1 = k * theta0
        z1 = R * sintheta0

        x0 = x1
        y0 = y1
        z0 = z1
        theta0 = theta1
        dtheta0 = dtheta1

    ico_sphere.location[0] = x1 # X-axis
    ico_sphere.location[1] = y1 # Y-axis
    ico_sphere.location[2] = z1 # Z-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = i)

# RESET OTHER
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = fps
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```

12.6 Vertical Loop Animation

```
#----------------------------------------------
#
# DOCUMENT DOCUMENT Blender_vertical_loop_animation.blend
# AUTHOR   (Anonymized)
# DATE     02-08-2022
#
#----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                          location=(10, 100, 20), rotation=
                          (math.radians(90), 0, math.
                          radians(180)), scale=(1, 1, 1))
bpy.context.object.data.type = 'ORTHO'
bpy.context.object.data.ortho_scale = 60

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                          location=(0, 0, 0), scale=(1, 1,
                          1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821

# SPHERE
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                          False, location=(0, 0, 10))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                          ShaderNodeEmission")
```

```
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Color"].default_value = (0, 0, 1,
                                     1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                        outputs[0],
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                        ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
bpy.ops.object.particle_system_add()
bpy.data.particles["ParticleSettings"].name = "ParticleSettings"
bpy.data.particles["ParticleSettings"].type = 'EMITTER'
bpy.data.particles["ParticleSettings"].frame_end = 200
bpy.data.particles["ParticleSettings"].effector_weights.gravity =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.all = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.force = 0
bpy.data.particles["ParticleSettings"].effector_weights.vortex = 0
bpy.data.particles["ParticleSettings"].effector_weights.magnetic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.harmonic =
                                     0
bpy.data.particles["ParticleSettings"].effector_weights.charge = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    lennardjones = 0
bpy.data.particles["ParticleSettings"].effector_weights.wind = 0
bpy.data.particles["ParticleSettings"].effector_weights.
                                    curve_guide = 0
bpy.data.particles["ParticleSettings"].effector_weights.texture =
                                    0
bpy.data.particles["ParticleSettings"].effector_weights.smokeflow
                                    = 0
bpy.data.particles["ParticleSettings"].effector_weights.turbulence
                                     = 0
bpy.data.particles["ParticleSettings"].effector_weights.drag = 0
bpy.data.particles["ParticleSettings"].effector_weights.boid = 0
bpy.data.particles["ParticleSettings"].lifetime = 300
bpy.data.particles["ParticleSettings"].normal_factor = 0
```

```
# VARIABLES
fps = 30
g = - 9.8
ddtheta = 0

theta0 = 0
dtheta0 = 5
x0 = 0
y0 = 0
vx0 = math.cos(theta0**2)
vy0 = math.sin(theta0**2)

x1 = 0
y1 = 0
vx1 = 0
vy1 = 0
theta1 = 0
dtheta1 = 0

ico_sphere.location[0] = x0
ico_sphere.location[2] = y0
ico_sphere.keyframe_insert(data_path = "location", frame = 0)

# ANIMATION
for i in range(1, 2000):
    for j in range(1, 2000):
        t = 1/(fps * 20000)
        ddtheta = g * math.sin(theta0**2)
        dtheta1 = ddtheta * t + dtheta0
        theta1 = dtheta0 * t + theta0

        vx1 = math.cos(theta0**2)
        x1 = vx0 * t + x0
        vy1 = math.sin(theta0**2)
        y1 = vy0 * t + y0

        x0 = x1
        y0 = y1
        vx0 = vx1
        vy0 = vy1
        theta0 = theta1
        dtheta0 = dtheta1

    ico_sphere.location[0] = x1 * 150 # X-axis
    ico_sphere.location[2] = y1 * 150 # Y-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = i)

# RESET OTHER
```

```
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = fps
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```

12.7 Corkscrew Animation

```
#----------------------------------------------
#
# DOCUMENT Blender_corkscrew_animation.blend
# AUTHOR   (Anonymized)
# DATE     04-11-2022
#
#----------------------------------------------

import bpy
import math

# SHINE
bpy.context.scene.render.engine = 'BLENDER_EEVEE'
bpy.context.scene.eevee.use_bloom = True

# RESET
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False)

# CAMERA
bpy.ops.object.camera_add(enter_editmode=False, align='VIEW',
                          location=(0.43, 0.68, 0.2),
                          rotation=(math.radians(75), 0,
                          math.radians(150)), scale=(1, 1,
                          1))
bpy.context.object.data.type = 'PERSP'
mainCamera = bpy.context.object
mainCamera.data.lens = 50.0
mainCamera.data.clip_start = 0.00005
mainCamera.data.clip_end = 200

# LIGHT
bpy.ops.object.light_add(type='SUN', radius=1, align='WORLD',
                          location=(0, 0, 0), scale=(1, 1,
                          1))
bpy.context.object.data.energy = 2
bpy.context.object.rotation_euler[0] = 0.383972
bpy.context.object.rotation_euler[1] = 1.0821

# CILINDER
#bpy.ops.mesh.primitive_cylinder_add(radius=2, depth=120,
                          enter_editmode=False, align='
                          WORLD', location=(0, 60, 0),
                          rotation=(1.5708, 0, 0), scale=(3
                          , 3, 1))

# CILINDER COLOR
```

```
#bpy.ops.material.new()
#bpy.data.materials["Material"].node_tree.nodes["Principled BSDF
                                    "].inputs[0].default_value = (0.1
                                    , 0.1, 0.1, 1)

# SPHERE
#bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                                    False, location=(0, 0, 10), scale
                                    =(2, 2, 2))
bpy.ops.mesh.primitive_ico_sphere_add(radius=0.5, enter_editmode=
                                    False, location=(0, 0, 10), scale
                                    =(0.01, 0.01, 0.01))
ico_sphere = bpy.data.objects['Icosphere']

# SPHERE COLOR
mat_name = 'EmiMat'
bpy.ops.material.new()
bpy.data.materials[-1].name = mat_name
bpy.data.materials[mat_name].node_tree.nodes.clear()
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeEmission")
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Color"].default_value = (0, 0, 1,
                                     1)
bpy.data.materials[mat_name].node_tree.nodes["Emission"].inputs["
                                    Strength"].default_value = 50
bpy.data.materials[mat_name].node_tree.nodes.new("
                                    ShaderNodeOutputMaterial")
links = bpy.data.materials[mat_name].node_tree.links
links.new(
    bpy.data.materials[mat_name].node_tree.nodes["Emission"].
                                        outputs[0],
    bpy.data.materials[mat_name].node_tree.nodes["Material Output"
                                        ].inputs[0])
bpy.ops.object.material_slot_add()
mat = bpy.data.materials.get(mat_name)
ob = bpy.context.active_object
ob.data.materials[0] = mat

# TRAIL
#C = bpy.data.objects['Icosphere']
#ps = C.particle_systems['ParticleSystem'].settings
bpy.ops.object.particle_system_add()
bpy.ops.object.ParticleSettings = "ParticleSettings"
#if not 'ParticleSettings':
if True:
    bpy.data.particles["ParticleSettings"].name = "
                                        ParticleSettings"
    bpy.data.particles["ParticleSettings"].type = 'EMITTER'
```

```
    bpy.data.particles["ParticleSettings"].count = 2000
    bpy.data.particles["ParticleSettings"].frame_end = 500
    bpy.data.particles["ParticleSettings"].effector_weights.
                                        gravity = 0
    bpy.data.particles["ParticleSettings"].lifetime = 1000
    bpy.data.particles["ParticleSettings"].normal_factor = 0
    bpy.data.particles["ParticleSettings"].mass = 0

# VARIABLES
fps = 30 # frames per second
g = - 9.8 # gravity
R = 10 # radius
k = 0.05
ddtheta = 0
f = 0 # frame
h = 300 # frames in in-line twist
w = 0 # number of half-rotations in in-line twist
t = 1/(fps * 50000)
clothoidConstant = 0.5

theta0 = 0
dtheta0 = 5
vx0 = 0
vy0 = 0
vz0 = 0
x0 = 0
y0 = 0
z0 = 0

xPosSwitch = 0
yPosSwitch = 0
zPosSwitch = 0

x1 = 0
y1 = 0
z1 = 0
vx1 = 0
vy1 = 0
vz1 = 0
theta1 = 0
dtheta1 = 0

ico_sphere.location[0] = x0
ico_sphere.location[1] = y0
ico_sphere.location[2] = z0
ico_sphere.keyframe_insert(data_path = "location", frame = 0)

# ANIMATION - Loop
while (2 * theta0 * math.cos(theta0**2) >= 0):
```

```
    f = f + 1
    for j in range(1, 100):
        ddtheta = g * math.sin(theta0**2)
        dtheta1 = dtheta0 + ddtheta * t
        theta1 = theta0 + dtheta0 * t

        vx1 = math.cos(theta0**2)
        x1 = vx0 * t + x0
        y1 = k * theta0
        vz1 = math.sin(theta0**2) * 1.2
        z1 = vz0 * t + z0

        x0 = x1
        z0 = z1
        vx0 = vx1
        vz0 = vz1
        theta0 = theta1
        dtheta0 = dtheta1

    ico_sphere.location[0] = x1 # X-axis
    ico_sphere.location[1] = y1 # Y-axis
    ico_sphere.location[2] = z1 # Z-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = f)

#R = (clothoidConstant**2)/theta0
R = 0.08

lastTheta = theta0
lastdTheta = dtheta0

xPosSwitch = x1 - R
yPosSwitch = y1
zPosSwitch = z1

print("
                                    ----------------------------------------------------
                                    ")
print(str(f) + " | " + str(2 * theta0 * math.cos(theta0**2)) + " |
                                     " + str(dtheta0))
print(str(R) + " | " + str(theta0) + " ||| "+ str(xPosSwitch) + "
                                    | " + str(yPosSwitch) + " | " +
                                    str(zPosSwitch))

theta0 = 0
dtheta0 = 20

#for i in range(int(f), int(f + h) + 1):
# ANIMATION - Twist
while (w < 5):
```

```
    if w % 2 == 0:
        if (- math.sin(theta0) >= 0):
            w = w + 1
            print("PATH 1: " + str(f))
    else:
        if (- math.sin(theta0) <= 0):
            w = w + 1
            print("PATH 2: " + str(f))
    f = f + 1
#for i in range(41, 341):
    for j in range(1, 100):
        costheta0 = math.cos(theta0)
        sintheta0 = math.sin(theta0)

        ddtheta = (g * R * costheta0)/(R**2 + k**2)
        dtheta1 = ddtheta * t + dtheta0
        theta1 = dtheta0 * t + theta0

        x1 = R * costheta0
        y1 = k * theta0/2
        z1 = R * sintheta0

        x0 = x1
        y0 = y1
        z0 = z1
        theta0 = theta1
        dtheta0 = dtheta1

    ico_sphere.location[0] = x1 + xPosSwitch # X-axis
    ico_sphere.location[1] = y1 + yPosSwitch # Y-axis
    ico_sphere.location[2] = z1 + zPosSwitch # Z-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = f)
    #print("> " + str(x1) + " | " + str(y1) + " | "+ str(z1) + " |
                                        " + str(xPosSwitch) + " | "
                                       + str(yPosSwitch) + " | " +
                                       str(zPosSwitch))

theta0 = lastTheta
xPosSwitch = xPosSwitch - R
dtheta0 = lastdTheta

x0 = 0
y0 = 0
z0 = 0

print(str(f) + " | " + str(2 * theta0 * math.cos(theta0**2)) + " |
                                    " + str(dtheta0))
print(str(R) + " | " + str(theta0) + " ||| "+ str(xPosSwitch) + "
                                  | " + str(yPosSwitch) + " | " +
```

```
                                    str(zPosSwitch))

f1 = f

# ANIMATION - Loop
#while (z0 + zPosSwitch >= 0):
while(x1 <= xPosSwitch + 2*R):
    f = f + 1
    for j in range(1, 100):
        ddtheta = (g * math.sin(theta0**2))
        dtheta1 = dtheta0 - ddtheta * t
        theta1 = theta0 - dtheta0 * t

        vx1 = math.cos(theta0**2)
        x1 = vx0 * t + x0
        vz1 = math.sin(theta0**2) * 1.2
        z1 = -vz0 * t + z0

        x0 = x1
        z0 = z1
        vx0 = vx1
        vz0 = vz1
        theta0 = theta1
        dtheta0 = dtheta1

    ico_sphere.location[0] = x1 + xPosSwitch # X-axis
    ico_sphere.location[2] = z1 + zPosSwitch # Z-axis
    ico_sphere.keyframe_insert(data_path = "location", frame = f)

print(str(f) + " | " + str(2 * theta0 * math.cos(theta0**2)) + " |
                                    " + str(dtheta0))

theta0 = 0

yPosSwitch = y1 + 2*yPosSwitch
f1 = f

while (theta0 <= lastTheta):
    for j in range(1, 100):
        ddtheta = g * math.sin(theta0**2)
        dtheta1 = dtheta0 + ddtheta * t
        theta1 = theta0 + dtheta0 * t

        vx1 = math.cos(theta0**2)
        x1 = vx0 * t + x0
        y1 = - k * theta0
        vz1 = math.sin(theta0**2) * 1.2
        z1 = vz0 * t + z0
```

```
        x0 = x1
        z0 = z1
        vx0 = vx1
        vz0 = vz1
        theta0 = theta1
        dtheta0 = dtheta1

    ico_sphere.location[1] = y1  + yPosSwitch # Y-axis
    ico_sphere.keyframe_insert(data_path = "location", index = 1,
                                        frame = f1)
    f1 = f1 - 1

# RESET OTHER
bpy.context.area.ui_type = "VIEW_3D"
for area in bpy.context.workspace.screens[0].areas:
    for space in area.spaces:
        if space.type == 'VIEW_3D':
            space.shading.type = "RENDERED"
bpy.context.area.ui_type = "PROPERTIES"
bpy.data.scenes['Scene'].render.fps = fps
bpy.data.scenes['Scene'].frame_end = f
bpy.context.area.ui_type = "TIMELINE"
bpy.ops.anim.change_frame(frame=0)
bpy.context.area.ui_type = "TEXT_EDITOR"
```