Alma Mater Studiorum · Università di Bologna

School of Engineering
Electrical, Electronic, and Information Engineering "Guglielmo Marconi" - DEI
Master Degree in Automation Engineering

VIRTUALIZATION OF WIRING HARNESS MANIPULATION TASKS THROUGH THE PYCHRONO SIMULATION ENGINE

Supervisor:
Chiar.mo Prof.
PALLI GIANLUCA

Co-Advisor:
GALASSI KEVIN

Author:
GANGEMI JACOPO MARIA

Academic Year 2021-2022 - Session II

Abstract

The purpose of this thesis work is the study and creation of a harness modelling system. The model needs to simulate faithfully the physical behaviour of the harness, without any instability or incorrect movements. Since there are various simulation engines that try to model wiring's systems, this thesis work focused on the creation and test of a 3D environment with wiring and other objects through the PyChrono Simulation Engine.
Fine-tuning of the simulation parameters were done during the test to achieve the most stable and correct simulation possible, but tests showed the intrinsic limits of the Engine regarding the collisions' detection between the various part of the cables, while collisions between cables and other physical objects such as pavement, walls and others are well managed by the simulator.
Finally, the main purpose of the model is to be used to train Artificial Intelligence through Reinforcement Learnings techniques, so we designed, using OpenAI Gym APIs, the general structure of the learning environment, defining its basic functions and an initial framework.

Contents

List of Figures

List of Tables

Introduction

Various applications in mechanical and electric engineering require wiring in order to transmit the electrical energy from the logical interfaces to the users, that can be motors, sensors, ecc..
This kind of deformable materials can be found everywhere, and their behaviour changes drastically from one object to the other. Since every object has its own mechanical characteristics, most of the handling and assembly is done through humans. One of the main works that needs human interventions is the wiring of mechanical machines, using thin cables.
Since this process is repetitive, the use of a robotic arm suits well the task but, in order to automatize this process, a precise model of the wiring needs to be known, and its mechanical characteristics need to be precisely defined.
Introducing robotic manipulators to handle such flexible objects tends to lessen physical burden on workers, and also grants tremendous economical benefits. Many of the techniques used for the manipulation of rigid objects cannot be applied directly to deformable objects, since rigid object manipulation considers mostly the control of the grasped object's pose, but, when dealing with deformable objects, one should consider also their deformations. The deformable objects considered in this work of thesis are wiring and cables, uniparametric objects that have no compression strength.[1] Several tasks that need to manipulate DLOs require an accurate estimate of the object's shape, hence the creation of a correct model is fundamental.
One of the approaches used to model the wiring is through thin cables, each of one has a given diameter, length, and physical properties. This cables can be modeled through usage of spline functions and Lagrange's Equations[2], and the resulting mathematical model is used to create various simulation engines, each of one with its pros and cons.
Thin cables are part of the Deformable Linear Objects models(DLO), that are characterized by having a dimension bigger than the other two. DLO models are used for modeling in a simulated environment the evolution in time of the physical configuration of the cable. This model can then be used by a robotic arm, assisted by an artificial intelligence, in order to simulate the various steps to reach a desired final configuration.
This project of thesis deals with the design of the mechanical model of a wiring using modern programming language and packages. This model will be used to train an arti-

ficial intelligence using Reinforced Learning techniques
The tools used to define the model are inside the package *PyChrono*, a Multi-Physics Simulation Engine written in Python, that is computationally efficient and achieve stable results in the simulation. Some other functions were inside *NumPy*, but the general structure of the model is written starting from *PyChrono* modules.
There are several other tools that can be used to simulate wiring and cables, such *Blender's Physics*.

Chapter 1

DLO Model

In this chapter we describe the base theory behind the definition of the model for Deformable Linear Objects. Such model is defined using the Lagrange's equations and the definition of B-Spline, and is based on the work done by Theettena et all in the paper Geometrically exact dynamic splines[2]. Lagrange Equations are used in order to formally define the energy equations based on some spatial control points, and B-Spline curves are used to create a geometrically exact formal expression of the curve.

1.1 B-Spline

In this section, we will explain the basic theory behind the B-spline, used throughout the theory of the DLO model. Generally, a spline is a function consisting of a set of connected polynomials, whose purpose is to interpolate a set of points, also known as **nodes**, in an interval, so that the function is continuous at least up to a given order of derived at any point in the interval.

1.1.1 B-spline basis functions

The Model uses a particular version of a spline curves, called B-Spline, that are distinguished from splines because they are actually a special case in which Beziér curves are used as polynomial curves. B-Spline of order k is the union of several pieces of polynomials of order *(k-1)* with, at most, $C^{(k-2)}$ continuity at the breakpoints.

Such breakpoints are defined in a non-decreasing order, forming the so called *knot vector*:

$$T = [t_0, t_1, ..., t_m], \quad t_0 \leq t_1 \leq t_2 \leq .... \leq t_m \tag{1.1}$$

The vector T defines the characteristics of the basis functions.

The corresponding B-spline basis functions can be defined as:

$$N_{i,1}(t) = \begin{cases} 1 \; for \; t_i \leq t \leq t_{i+1} \\ 0 \; otherwise \end{cases} \tag{1.2}$$

where $k = 1$ and

$$N_{1,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t) \; for \; k > 1 \; and \; i = 0, 1, .., n \tag{1.3}$$

The B-spline Basis functions have the following properties:

- $N_{i,k}(t) > 0$ for $t_i < t < t_{i+k}$ called **Positivity**
- $N_{i,k}(t) = 0$ for $t_0 \leq t \leq t_i$ and $t_{i+k} \leq t \leq t_{n+k}$ called **Local support**
- $\sum_{i=0}^{n} N_{i,k}(t) = 1$ for $t \epsilon [t_0, t_m]$ called **Partition of unity**
- $N_{i,k}(t)$ has C^{k-2} continuity at each simple knot

The following figure is an example of an order four B-spline.

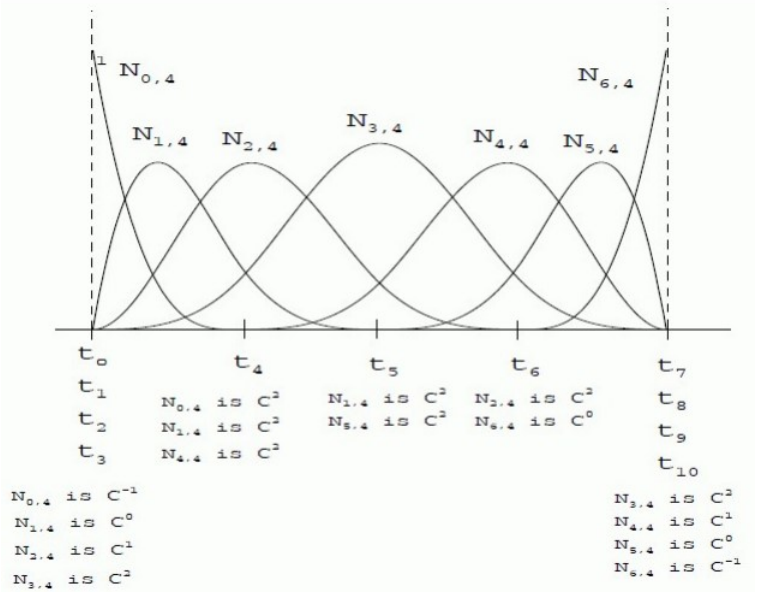

Figure 1.1: Order 4 B-spline with uniform knot vector

1.1.2 B-spline curve

A B-spline curve is defined as linear combination of the control point p_i and the spline basis function $N_{i,k}(t)$. This leads to:

$$r(t) = \sum_{i=0}^{n} p_i N_{i,k}(t) \; k-1 \leq n \; t\epsilon[t_{k-1}, t_{n+1}] \tag{1.4}$$

The control points are called de Boor points. The knot vector T contains $n+k+1$ elements, where n+1 is the number of control points and k is the order of the curve. Each knot span is mapped into a polynomial curve between two successive $r(t_i)$ and $r(t_{i+1})$.

1.1.3 B-spline curve properties

The B-spline curve has this following properties:

- *Geometry invariance property*: the shape of the B-spline curve is invariant to translation and rotation thanks to the **Partition of unity**
- *End points geometric property*: Usually, B-spline curves do not pass through the two end control points. The curve has C^{k-p-1} continuity at a knot that has multiplicity $p(\leq k)$. In order to assure that the curve passes through the end points, we need to repeat the knots at the two end of the knot vector *k times*. Thus, the knot vector will become $T = (t_0, t_0, .., t_0, ...t_{k+1}, ..., t_{n+k}, ...t_{n+k})$. Such knot vectors are known as *clamped*. A clumped B-spline curve is illustrated in figure 1.2.

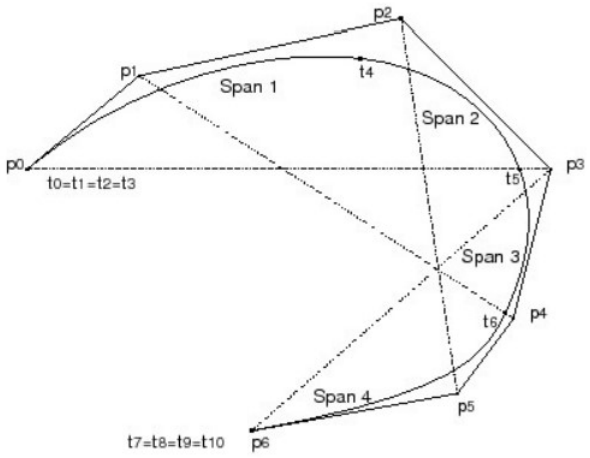

Figure 1.2: Clamped cubic B-spline

- A particular property of clamped B-spline curves is that they are tangent to the control polygon at their endpoints. This is given by the fact that

$$\dot{r}(t) = \sum_{i=0}^{n-1}(k-1)(\frac{p_{i+1}-p_i}{t_{i+k}-t_{i+1}})N_{i,k-1}(t)$$

where the knot vector is obtained by dropping the first and last knot element and

$$\dot{r}(0) = \frac{k-1}{t_k - t_1}(p_1 - p_0)$$

$$\dot{r}(1) = \frac{k-1}{t_{n+k-1}-t_n}(p_n - p_{n-1})$$

1.1.4 De Boor's Algorithm

The B-spline can be evaluated in a specific parameter $\bar{t}$ using the De Boor's algorithm[3]. This algorithm computes the spline with an equivalent recursion formula. Starting from:

$$r(t) = \sum_{i=0}^{n+j} p_i^j N_{i,k-j}(t) \quad j = 0, 1, ..., k-1$$

Where

$$p_i^j = (1 - \alpha_i^j)p_{i-1}^{j-1} + \alpha_i^j p_i^{j-1} \quad j > 0$$

with

$$\alpha_i^j = \frac{\bar{t}-t_i}{t_{i+k-j}-t_i} \qquad \text{and} \qquad p_j^0 = p_j$$

For $j = k-1$, the spline basis function reduces to $N_{l,1}$ for $t\epsilon[t_l, t_{l+1}])$ and p_l^{k-1} coincides with the curve

$$r(\bar{t}) = p_l^{k-1}$$

The figure below shows the De Boor's algorithm graphically.

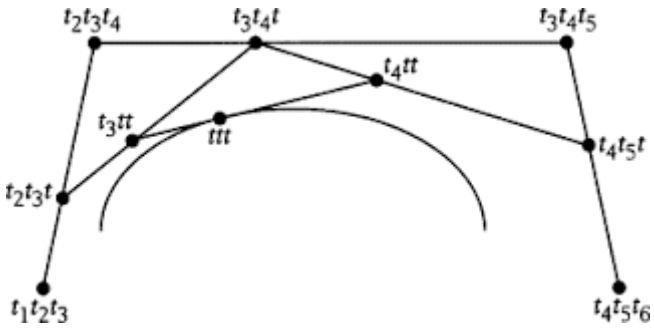

Figure 1.3: De Boor's algorithm

1.2 Lagrange Equations

Lagrange equations are differential equations of the second order and can describe a mechanical conservative system, and yields to the equation of motion. The fundamental theorem states that Lagrange equations are identical to the second principle of dynamics, that relates position and velocity of any other element of the system.
These equations are based on a defined control point, usually the position of the element of the system, and its first derivative. These equations involve the kinetic energy T and potential energy U of the system itself.

We define the Lagrangian as:

$$L = T - U \tag{1.5}$$

Performing the derivative of the Lagrangian we obtain:

$$\frac{d}{dt}(\frac{dL}{d\dot{q_i}}) - \frac{dL}{dq_i} = F_i \tag{1.6}$$

q_i is the generalized coordinate and F_i are corresponding generalized forces. Rewriting it in terms of T and U:

$$\frac{d}{dt}\frac{dT}{d\dot{q_i}} - \frac{d}{dt}\frac{dU}{d\dot{q_i}} - \frac{dT}{dq_i} + \frac{dU}{dq_i} = F_i \tag{1.7}$$

Since potential energy does not depend on velocity nor time, the term $\frac{d}{dt}\frac{dU}{d\dot{q_i}}$ is null and, in our case, the kinetic energy does not depend on position. The final equation become:

$$\forall\ i\ \epsilon\ 1, ..., n,\ \frac{d}{dt}\frac{dT}{d\dot{q_i}} = F_i - \frac{dU}{dq_i} \tag{1.8}$$

F_i is the sum of the generalized external forces at q_i

1.3 Geometrically Exact Dynamic Spline

The Geometrically Exact Dynamic Spline is the results of the combination of the Lagrange Equations with the B-Spline representation of the control points. In this way we obtain a geometrically exact model of the cable.

1.3.1 Beam geometry definition

In order to define the Dynamic Splines, consider a cross-section of diameter D and section S as shown in Fig.1.4.

We define as *neutral fiber* f the oriented curve of length L that passes through the center of every cross-section. The whole volume defined by every cross-section creates

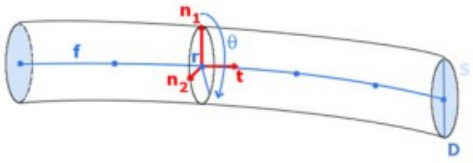

Figure 1.4: Scheme of a small with with its geometrical parameters

the *beam*.
We can describe the beam configuration using two different fields: the first one, called *position field* $\mathbf{r} = (x, y, z)$, describes the natural fiber position in space; the second one, called *rotation field* θ describes the roll of the natural fiber.
Defined the two fields, they can be combined in order to describe a unique field

$$\mathbf{q} = (\mathbf{r}, \theta) = (x, y, z, \theta) \tag{1.9}$$

Each resulting spline is the obtained by

$$\mathbf{q}(u) = \sum_{i=1}^{n} b_i(u)\mathbf{q}_i \tag{1.10}$$

In equation (1.3), b_i are the *ith* spline basis functions of the control points q_i, and u is the value between 0 and L, that is the natural fiber length. Given the resulting spline, the ***jth*** derivative of q with respect to u is given by:

$$\mathbf{q}^{(j)}(u) = \sum_{i=1}^{n} b_i^{(j)}(u)\mathbf{q}_i \tag{1.11}$$

We will denote arc length and derivatives of point $\mathbf{q}$, position $\mathbf{r}$ and roll θ as, respectively, s, $\mathbf{q}'$, $\mathbf{r}'$ and θ'. Displacement elements ds and du are interrelated by $ds = \|r'\|du$. Control points are important to define the Lagrange equations by considering them as degrees of freedom, since they define completely the position of the spline and the orientation of the cross section.

1.3.2 Elastic Domain and Strain Energy

To obtain the motion of control points with the Lagrange formulations, we need first to define deformation energies starting from the physical parameters. After this passage, we need to differentiate with respect of degrees of freedom.
We can express every action on the natural fiber f as forces and torques in the local frame, since they are proportional to stress and easier to manipulate. We can define a vector F made of three different forces:

- F_S is the *normal force* to the cross-section. The resulting motion is the streching of the natural fiber.
- F_T is the *torsional torque*, responsible of the rotation of the cross-section.
- F_B denotes the *bending force*, that corresponds to the oriented curvature of the natural fiber.

So the vector F can be defined as:

$$F = \begin{bmatrix} F_S \\ F_T \\ F_B \end{bmatrix} \tag{1.12}$$

By the usage of the *Kirchhoff assumption*, we can presume that the cross-sections are stiff and only that the neutral axis is distorted. This yields to the *Rayleigh model*. Force F is related to the strain ϵ. Their elastic relationship is described by Courbon[4]. The rest strain is denoted by ϵ^0.
To facilitate calculations of strain energies, we work under the *small strain* assumptions. The result of these considerations is the following equation, derived from Hooke's law:

$$F = H(\epsilon - \epsilon^0) = \begin{bmatrix} ES & 0 & 0 \\ 0 & GI_0 & 0 \\ 0 & 0 & EI_S \end{bmatrix} (\epsilon - \epsilon^0) \tag{1.13}$$

I_0 is the polar momentum of inertia, I_S is the cross-section momentum of inertia, and ES, GI_0 and EI_S are the streching, twisting and bending rigidities. The whole matrix define the H Hooke matrix.[4]
Since the assumptions state that the cross-section is circular and the diamenter is costant, we can rephrase the Hooke matrix as:

$$H = \frac{D^2\pi}{4} \begin{bmatrix} E & 0 & 0 \\ 0 & \frac{GD^2}{8} & 0 \\ 0 & 0 & \frac{ED^2}{16} \end{bmatrix} \tag{1.14}$$

Strain energy U can be formulated by the following integration along the beam:

$$U = \frac{1}{2} \int_0^L (\epsilon - \epsilon^0)^t F ds \tag{1.15}$$

Changing the expression of F with (1.13), we obtain:

$$U = \frac{1}{2} \int_0^L (\epsilon - \epsilon^0)^t H (\epsilon - \epsilon^0) ds \tag{1.16}$$

Now that we have all the mechanics necessary to determine our motion, we will study the two terms of the Lagrange equations.

1.3.3 Twisting in dynamic splines

The kinetic energy of the object comprises translational and rotational energy, since our one-dimensional object is specified by position and rotation. In order to formulate kinetic energy, first we need to define *inertia matrix* J, which is the same everywhere along the spline, since the diameter is costant:

$$J = \begin{bmatrix} \mu & 0 & 0 & 0 \\ 0 & \mu & 0 & 0 \\ 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & I_0 \end{bmatrix} \tag{1.17}$$

μ corresponds to the linear density. We can define now the kinetic energy as:

$$T = \frac{1}{2} \int_0^L \frac{d\mathbf{q}^t}{dt} J \frac{d\mathbf{q}}{dt} ds \tag{1.18}$$

To obtain the left term of the Lagrange equations, we need to differentiate the kinetic energy with respect to the control point $\mathbf{q}_i$. This yields to:

$$\frac{d}{dt}\frac{\delta T}{\delta \dot{q}_i} = \frac{1}{2} \int_0^L \frac{d}{dt} \frac{\delta \frac{d\mathbf{q}^t}{dt} J \frac{d\mathbf{q}}{dt}}{d\dot{\mathbf{q}}_i} ds \tag{1.19}$$

By replacing $\mathbf{q}$ with expression (1.10), yields to:

$$\frac{d}{dt}\frac{\delta T}{\delta \dot{q}_i} = \sum_{j=1}^{n} J \int_0^L (b_i(s) b_j(s)) ds \frac{d^2\mathbf{q}_j}{dt^2} \tag{1.20}$$

We can define a more compact version of the expression above considering a matrix M with components $M_{i,j} = J \int_0^L (b_i(s) b_j(s))$ and a vector A of components $\mathbf{A}_j = \frac{d^2\mathbf{q}_j}{dt^2}$. This yields to:

$$\frac{d}{dt}\frac{\delta T}{\delta_i} = \sum_{j=1}^{n} M_{i,j} \mathbf{A}_j \tag{1.21}$$

Considering all degrees of freedom, this sum can be written as a matrix-vector product:

$$M\mathbf{A} \tag{1.22}$$

1.3.4 Energy Evaluation

Now we need to express the right term of the Lagrange equation. The derivative of the potential energy has a more complicated expression:

$$P^i = -\frac{\delta U}{\delta \mathbf{q}_i} = -\frac{1}{2}\int_0^L \frac{\delta(\epsilon - \epsilon^0)^t H(\epsilon - \epsilon^0)}{\delta \mathbf{q}_i} ds \tag{1.23}$$

This term can be generalized by sum three forces:

- P_S, that is the stretching force;
- P_t, that is the twisting force;
- P_b, that is the bending force.

We need to express these generalized forces with respect to the position $\mathbf{r}$ up to the third derivative $\mathbf{r}^{'}$, $\mathbf{r}^{''}$, $\mathbf{r}^{'''}$, and also the spline basis function $\mathbf{b}_i$ up to its third derivative $\mathbf{b}_i^{'}$, $\mathbf{b}_i^{''}$, $\mathbf{b}_i^{'''}$.

For compactness of equations, we introduce the following variables:

- $C = \mathbf{r}^{'} \times \mathbf{r}^{''}$;
- $P = \frac{\delta \mathbf{r}^{'} \times \mathbf{r}^{''}}{\delta \mathbf{r}_i}$;
- $T = Cb_i^{'''} - P \times \mathbf{r}^{'''} - 2\tau(CxP)$.

$\times$ denotes the cross product. Now we can express the generalized forces described above.

- Stretching Force
  In small strains, the streching strain is defined by $\epsilon_s = 1 - ||\mathbf{r}^{'}||$. This leads to:

$$P_S^i(\mathbf{r}) = -\frac{\pi E D^2}{4}\int_0^L (1 - \frac{||\mathbf{r}_0^{'}||}{||\mathbf{r}^{'}||})\mathbf{r}^{'} b_i^{'} ds \tag{1.24}$$

  We can also assert that:

$$P_S^{'}(\theta) = 0 \tag{1.25}$$

  since stretching strain energy U_s does not depend on θ.

- Twisting Force
  To define the twisting force, we need to consider the *Frenet twisting* τ and the *roll* θ. Frenet or geometrical twisting is due to the bending of the neutral fiber, whereas roll corresponds to the rotation of the cross-section of the material. As described

by Chouaieb[5], twisting is the sum of Frenet twisting and a rotation about the tangent. The twisting result in the following expression:

$$\epsilon_t = \theta^{'} + \tau$$

$\tau = \frac{\mathbf{r}^{'} \times \mathbf{r}^{''} \cdot \mathbf{r}^{'''}}{||\mathbf{r}^{'} \times \mathbf{r}^{''}||^2} = \frac{C \cdot \mathbf{r}^{'''}}{||C||^2}(1.26)$ We can now express the two contribution. The geometrical twisting yields to:

$$P_t^i(\mathbf{r}) = -\frac{\pi G D^4}{32} \int_0^L (\epsilon_t - \epsilon_t^0) \frac{T}{||C||^2} ds \tag{1.27}$$

The roll contribution is:

$$P_t^i(\theta) = -\frac{\pi G D^4}{64} \int_0^L (\epsilon_t - \epsilon_t^0)(\frac{b_i^{'}}{||r^{'}||}) ds \tag{1.28}$$

- Bending Force
  The bending force is function of the scalar Frenet curvature k, which is equal to the bending strain:

$$\epsilon_b = k = \frac{||\mathbf{r}^{'} \times \mathbf{r}^{''}||}{||\mathbf{r}^{'}||^3} = \frac{||C||}{||\mathbf{r}^{'}||^3} \tag{1.29}$$

  The bending force term P_b^i yields:

$$P_b^i(\mathbf{r}) = -\frac{\pi G D^4}{64} \int_0^L \frac{\epsilon_b - \epsilon_b^0}{||r^{'}||^2} (\frac{C \times P}{||C||||\mathbf{r}^{'}||} - 3 k b_i^{'} r^{'}) ds \tag{1.30}$$

  Similarly to the stretching energy, the bending energy U_b does not depend on θ, so $P_b^i(\theta) = 0$

All the forces computed can be summed and described by a vector $P = P_s + P_t + P_b$.

1.4 Model final expression

Considering all the equations expressed above, we can put the whole relation in a more compact matrix form, obtaining:

$$\begin{bmatrix} M & L^T \\ L & 0 \end{bmatrix} \begin{bmatrix} A \\ -\lambda \end{bmatrix} = \begin{bmatrix} F + P \\ E \end{bmatrix} \tag{1.31}$$

L is the matrix of constraints and λ are the Lagrange multipliers. At each step, the system is solved multiplying the equation by the pesudoinverse of the first matrix, and the obtained velocities and accelerations are integrated in order to find the new state of each control point, that is each position a velocity.

1.5 Project Chrono

The theory explained above led to various models, each of one created for a specific environment. The one used for our project of thesis is *Project Chrono*[6]. Project Chrono is an Open Source Multi-Physics Simulation Engine. The original project is written in C++, and it's based on a platform-independent open-source design. The libraries can be used in a project to simulate various physical systems, such as wheeled vehicles, robots, and many others mechatronic systems. The systems can be made of rigid and flexible elements, and each element has its own three-dimensional shape for collision detection. PyChrono[7] is the Python library that wraps the Chrono C++ simulation library. In our project, we used the version 7.0.3, the latest one. The advantages of this library are various, such as a fast comprehension of the code, and its ease of use.
One of the intrinsic limit of the PyChrono suite is the absence of the plastic deformations. Plastic deformations happen when on the body are applied excessive loads and, when the load is finally ceases, the final configuration of the wire is not the same as the initial one, with a permanent deformation ruled, in general, by the *stress-deformation curve*. A general curve is represented in the Fig. 1.5 below:

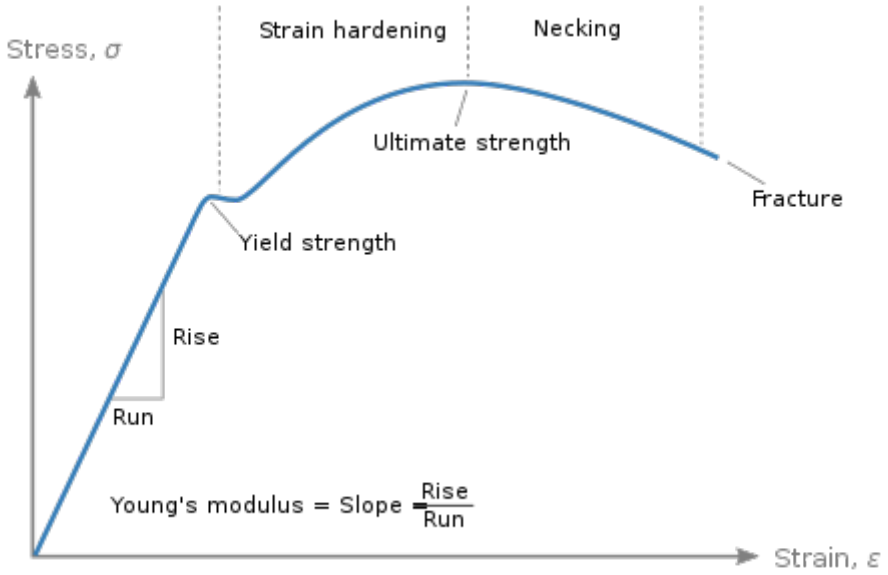

Figure 1.5: General stress-strain curve

In our model, the stress applied to the wire is not strong enough to reach the plastic deformation's region, so we can not take into account plastic deformations.

PyChrono's APIs are based on an alternative work, called A geometrically exact isogeometric beam for large displacements and contacts[8], where the model is based

on the shear-flexible Cosserat rod theory implemented in the context of Isogeometric Analysis.
Similarly to the theory explained above, the fiber of the beam is parametrized using splines and time integration of rotations is performed using the exponential map of quaternions.

1.6 OpenAI Gym

Gym[9] is an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API. Since its release, Gym's API has become the field standard for doing this.
In this work of thesis, Gym's API are used to build the skeleton for the reinforcement learning algorithms. The documentation can be found at *https://www.gymlibrary.dev/*.
Reinforcement learning is a machine learning technique that aims to create autonomous agents able to choose actions to be taken to achieve certain objectives through interaction with the environment in which they are immersed. It's one of the three principal paradigms of the automatic learning, and it takes care of sequential decisions' problems. These problems are characterized by the fact that an action to be taken depends on the initial state of the system and determines the future one.
The quality of the action is based on a reward-like numeric value, modelized through the Markov decision-making processes[10].
The training follows the *agent* paradigma, in which we call the objective we want to train as agent, and we uses some of its characteristics as *observation* parameter, such as its position or its velocity.

Chapter 2

Code Analysis

In this section, we will explain the tools used to create the model, using the programming language Python, and the reasons behind this choices.
The whole code is divided into three different parts, the *main.py*, the *GymEnv.py*, a class in which we created the main functions to implement the reinforced learning, and *CableModels.py*, the most important class, in which we create the models of the wires.
The function of the *main.py* is to create and manage the 3D environment, using *GymEnv.py* classes and functions. Inside the *main.py* we manage the simulation's steps and saves the informations needed to a further analysis.

2.1 PyChrono Implementation

In PyChrono, the cable can be defined through the Chrono API. Once the model is defined, the behaviour is studied using the Finite Element Analysis (FEA).
The cable is defined using the *pychrono.fea* module, in which we find all the functions used to define the spatial configuration of the cable, its physical properties and the collisions models. After importing the module fea module, we can begin to create the cable, starting from its spatial configuration and physical properties.
Before defining the cable, we need to define the physical properties of the cable through a *fea.ChBeamSectionEulerAdvanced*[11], in which you can assing all the physical variables, like the Young Modulus, the section of the beam and so on. In our specific cases, the section of the beam is set to circular, with a diameter that varies depending on the section of the cable we are considering.
The nodes of the cable are simply created through the API *fea.ChNodexyzrot*[12]. This function takes as an input a *chrono.ChFrameD*, an object made of a position vector *chrono.ChVectorD*, and a direction, defined as a quaterion. The positions can be defined in any way someone wants, and in the study cases, we defined the positions of the nodes in two different ways: as a mathematical function and from a input file. Nodes are stored

inside lists in Python. The resulting element in the enviroment is shown in figure 2.1.

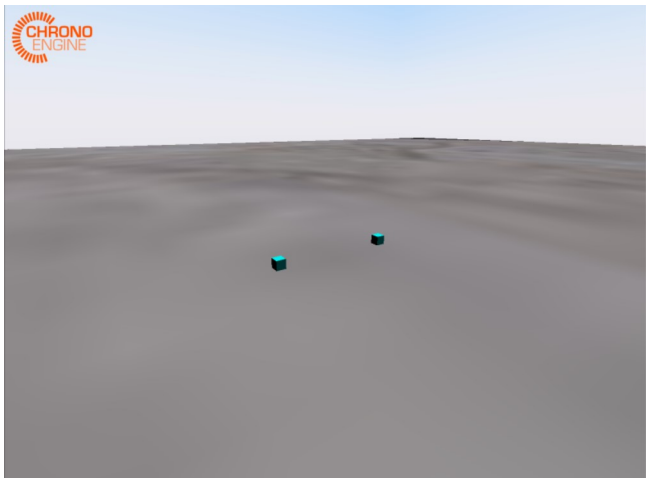

Figure 2.1: PyChrono Nodes

After the nodes are defined, the next thing we need to create is the cable itself. For the sake of simplicity, we will call as *cable element*, the section of the cable create from one node to the following. A cable element is created through the API *fea.ChElementBeamEuler*[13]. We used this function since is the one compatible with the Beam Section described above. To create the cable element, we just need to set the nodes and the section we want to use. Every cable element is stored, like the nodes, in lists. The resulting element in the enviroment is shown in figure 2.2.

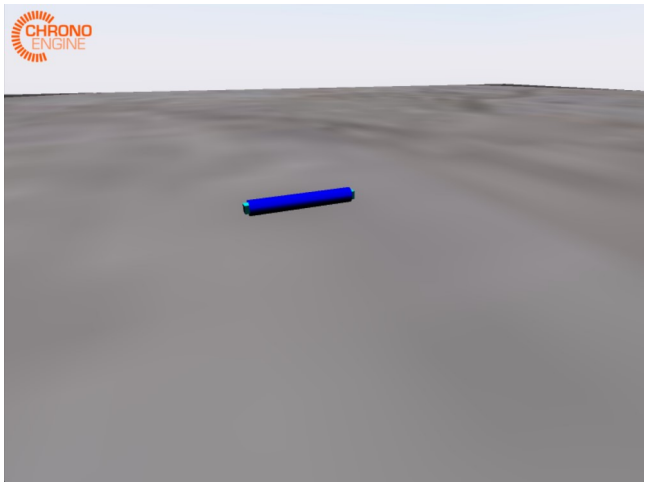

Figure 2.2: PyChrono Cable Element

In order to simulate collision and visualize the various element, we need to create the *mesh*, an object in which we will assign nodes and other elements.
The *mesh*[14] can be used to define the elements of the simulation that will interact with the other elements inside, or by mean of visualization. In our project of thesis, we decided to create three different mesh.
The first one, called simply *cable mesh*, is responsible only on the visualization of the cable elements, that is nodes and the resulting cable.
The second one, called in the code ***contact_mesh_node***, is responsible to detect the node-to-node collisions. In order to detect collisions, you have to assign the *fea* elements to a specific mesh, that will be used by the simulator during the simulation process.
The third one, called ***contact_mesh_cable***, is responsible to detect the cable-to-cable collisions. Similarly to the contact mesh node, we need to assign the cable elements.
From various tests, we decided to create two different mesh for the collisions to strengthen the simulation. Without the collision detection, the cable would just "go through itself" when touching another of its parts. This, of course, leads to errors in the final simulation.
Both ***contact_mesh_cable*** and ***contact_mesh_node*** requires some parameters, used only to manage the interpenetration between the two meshes. This parameters were modified during various tests, to find the ones that impede interpenetration and also bouncing between the elements.
The cable than can be moved inside the enviroment by applying forces or velocities to each node. The simulator, after a single step, computes the new positions of the nodes of the cable, taking into considerations the whole model.
Other than the cables, *PyChrono* provides simple APIs to create various 3D objects that can be used to simulate obstacles inside the simulation. In the project, we used *chrono.ChBodyEasyBox*[15], that takes as input the dimension of the box we want to create, its density and the material.
Since we used this function to create both the pavement, the walls and the obstacle, we defined two different materials through *chrono.ChMaterialSurfaceSMC*, a function that, similarly to *fea.ChBeamSectionEulerAdvanced*, allows us to assign the Young Modulus, restitution coefficient and so on to every 3D object we want.
The **Restitution coefficient** is defined as the ration between the *relative velocity* of the object with the other after and before the collision:

$$\mathbf{CoR} = \frac{|Relativevelocityaftercollision|}{|Relativevelocitybeforecollision|} \tag{2.1}$$

Another important parameter that can be assigned is the **Adhesion coefficient**, that is the ration of adhesive force to vertical load between two objects. It's generally defined for weels interacting with roads.
Lastly, the **Friction coefficient** is the ration between the resistive friction force and the

normal force:

$$fr = \frac{Fr}{N} \tag{2.2}$$

with **fr** friction coefficient, **Fr** resistive force and **N** normal force.
The first material created is called *pavementmaterial* and is used to define the properties of the pavement and the walls. We have set the Young Modulus to **0.5 MPa**, friction coefficient to **0.3** and restitution coefficient to **0.2**. This values are only assign based on the stability of the simulation. In particular, the Young Modulus needs to be not to high, in order to impede the cable to bounce inside the simulation, nor to low. Especially, if the Young Modulus is to low, the collision detection of the system doesn't work properly, and the cable will go through the pavement (or the walls) instead of being blocked.
The second and last one is the *cubematerial*. It used to define the properties of the obstacle (in our specific case a simple cube). We have set the Young Modulus to **1 GPa**, friction coefficient to **0.3** and restitution coefficient to **0.2**.
The 3D elements created are used to practically define the 3D environment in which our cable can move and interact. The collision-detection is automatically handled by the simulator, but one can decide to change its shape based on some preferences.

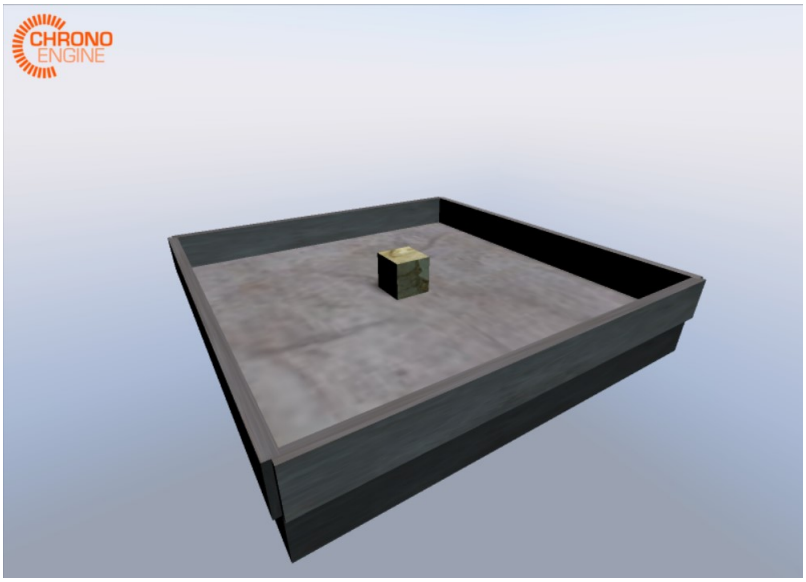

Figure 2.3: PyChrono 3D objects

PyChrono provides also a various number of solver, and we went with the **MinRes** solver because it provides good performances with a stable simulation. Such solver works with the Minimum Residual method, where it is a short-recurrence method with a constant memory requirement.[16] The parameters one can define are the number of step for every iteration, the time step of the single iteration and the minimum resolution.

We tested other solutions, such as the **Pardiso MKL** solver, that didn't work at all, since there were some conflicts with the installed drivers. In general, all the *Pardiso-based* solver proposed by PyChrono were not functioning. Another choice was the **Sparse QR** solver, a sparse left-looking rank-revealing QR factorization. This showed similar results to the **MinRes** solver, but with higher time needed per step. A particularity of the 3D environment of PyChrono is the axis definition. Generally, the plane $X - Y - Z$ has as vertical axis the Z-axis, but PyChrono takes as ground the $X - Z$ plane and as vertical axis the Y-axis.

2.2 CableModels.py

The functions described above are used inside *CableModels.py*, in which we created three different configurations to test the stability of the environment. These functions are then used inside *GymEnv.py* to then develop the reinforced learning techniques. The three configurations are developed as classes, that differentiate one from the other only on the initial spatial configuration of the cable, while the main structure of the classes remain the same. Below we will describe the different configurations for the cables, and the general structure of the class.
A schematic of *CableModels.py* is represented in the image below.

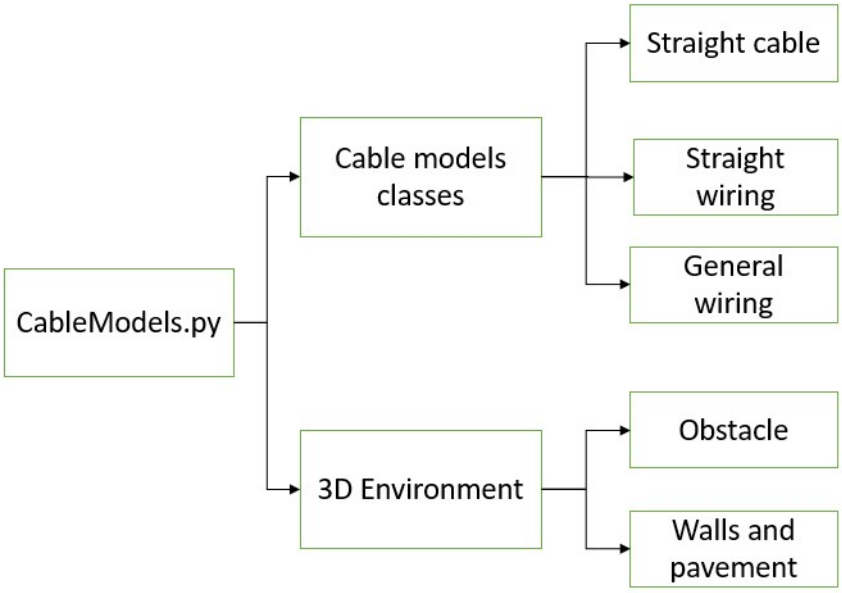

Figure 2.4: CableModels.py scheme

2.2.1 General Structure

CableModels.py is a module containing classes regarding the cable models, and also the main items used to create the 3D environment.
The cable models' classes are similar in the main structure, but differ one from the other only on the initial configuration. The main function of the cable models' classes is the *__init__*, in which we create the cable itself with the contact and visualization meshes. It takes as input only the meshes used to simulate the cable-to-cable contacts and the visualization. If we are creating a wiring that has multiple bifurcations, we need also to

input the *system* to create the constraints that the main cable has with its bifurcations. The cable model's class has also other auxiliary functions, used to define its behaviour inside the simulation, such as *SetVelocity* or *SetForce*, that both takes respectively as input the velocity or the force, the node in which we want to apply it, and the cable defined by its ID.
Since we need also to save and then process the state of the cable, we defined some functions, *CreateDataBase* and *AddDataFrame*, that allow us to save in a *NodePosition.csv* file the main informations about the cable. The way we do so is explained in a section below.
In order to simulate various kind of wiring, we created three different cable's models, that will be fully described in the sections below:

- **Straight Cable**: simple straight cable with fixed lenght and diameter
- **Straight Wiring**: straight wiring made of three different cables, all tied together to simulate a more complex wiring, with fixed length and diameter
- **General Wiring**: Wiring made with multiple cables with different length, diameter and physical conformation

Along with the classes to create the cable, there are also the functions used to create the obstacle, the walls and the pavement. This are used directly in the *GymEnv.py* class, that will be described after the different kind of cables.

2.2.2 Straight Cable

The ***Straight Cable*** is the simplest one. The class's name is ***StraightCable***. As the name states, it's a wiring made only of a straight cable, and so the nodes a created along the X-axis, using the default direction of the nodes when they are created.
Since it's the most basic one, it was used preliminary to check if the initial parameters of the objects inside the simulations where correct. Below there's the figure representing the cable inside the environment.

2.2.3 Straight Wiring

The ***Straight Wiring*** is similar to the *Straight Cable* but, instead of being made only of one cable, it's made of 3 different cables, all of them parallel to the other along the X-axis. The class's name is ***Straight3Cable***
The wiring can be ideally dived into the main cable and the two slave cables. Each slave cables' node has a constraint that force its position to follow the main cable's node evolution. Since we need to define the constraints, in this class we input also the *system* we created in the *GymEnv.py* environment.

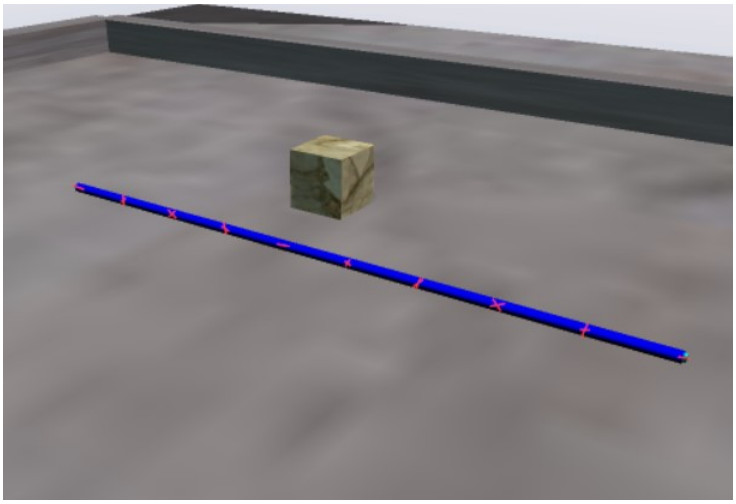

Figure 2.5: Straight cable

It was mainly used to find a way to initially create wirings, but the idea was set aside for some instability of the collision detection system, along with a better idea to create a wiring that was randomly generated.

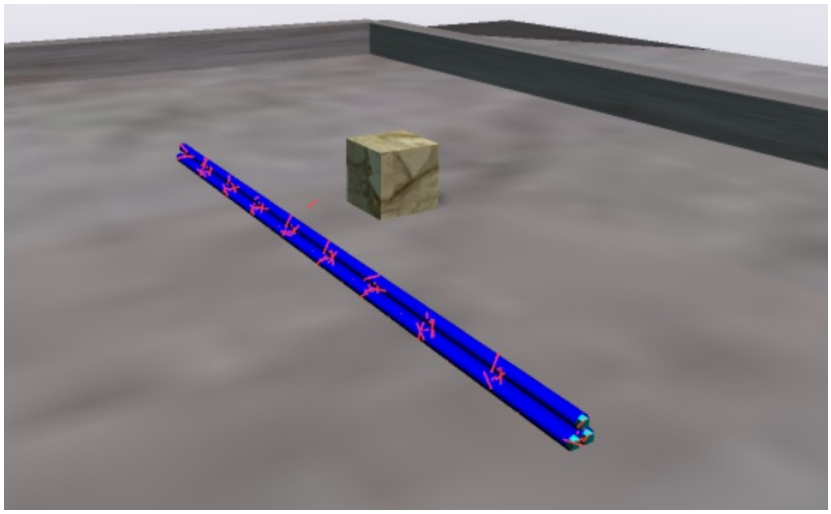

Figure 2.6: Straight wiring

2.2.4 General wiring

The most important model, what will be used for the reinforced learning, is the General wiring, that has a spatial conformation randomly.
The positions of the nodes are stored inside a *label.yaml* file, a dictionary divided in sections. The table below shows how is made a cable of the wiring stored inside the .yaml file.

Cable ID1:		
branches:	 id: Spline index: id: Spline index:	 3 99 XX XX
diameter:	0.025	
intersections:	 k1: k2: point:	 X1 X2 [XX, YY]
points:	 XX .. XX	 YY .. YY
spline:	 XX .. XX	 YY .. YY
Cable ID2:		
..	..	..

Table 2.1: Structure of label.yaml File

Each dictionary's section is defined by the **Cable ID**, that ranges from 0 to N, where N is the number of cables that composes the wiring. The subsections of the dictionary are defined as follows:

- **branches**: This section contains the id of the cable that's originated from the principal one and the index of the spline matrix where this bifurcation starts.
- **diameter**: This section contains the diameter of the cable
- **intersections**: This section contains the *point* where the principal cable, whose ID is defined by the variable *k1*, intersect with the cable whose ID is defined by *k2*

- **points**: This section contains the points where the given spline interpolation needs to go through
- **spline**: This is the most important section. Here we store the final points of the spline as a Matrix of 100 elements, each element contains the spatial coordinates of the point.

Since it's spatial coordinates are randomly generated, we needed also to define a way to give to its nodes the correct direction, in order to eliminate some initial instability given by the fact that, as a default parameter, the direction of the nodes is along the X-axis, and the evolution of the cables is random. To address this issue, we assigned as direction of the node the vector that connects the previous node to the next one, as schematized in the figure below.

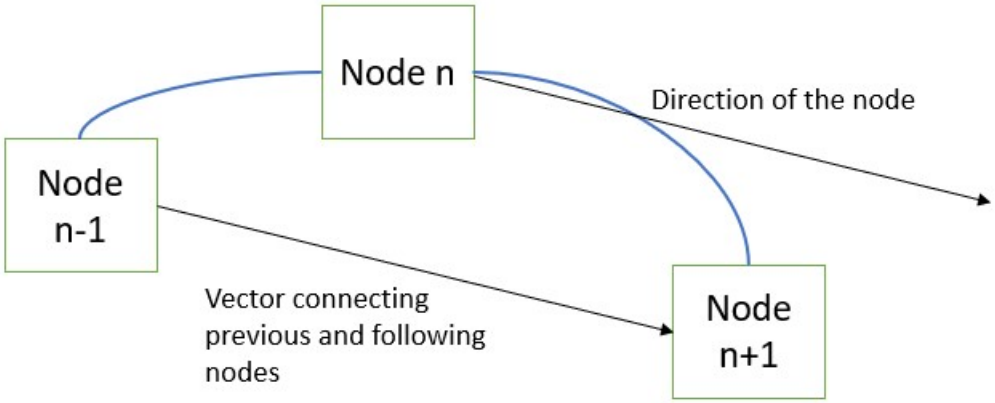

Figure 2.7: Direction of the nodes

To obtain the spatial coordinates of the wiring sections, points we created automatically starting from a Blender's script for the generation of synthetic datasets.
It's used initially the structure of the wiring, so we start from the first point of the main cable to then generate the various branches forcing the cables' interstactions. If not, the points would be purely casuals.
Once we've obtained the 2D points, we plot them in the 3D environment generating a volume of points around the guide at a distance equal to the diameter of the segment from which we obtain the pointcloud. The initial pointcloud and the final result are showed in the figure below.

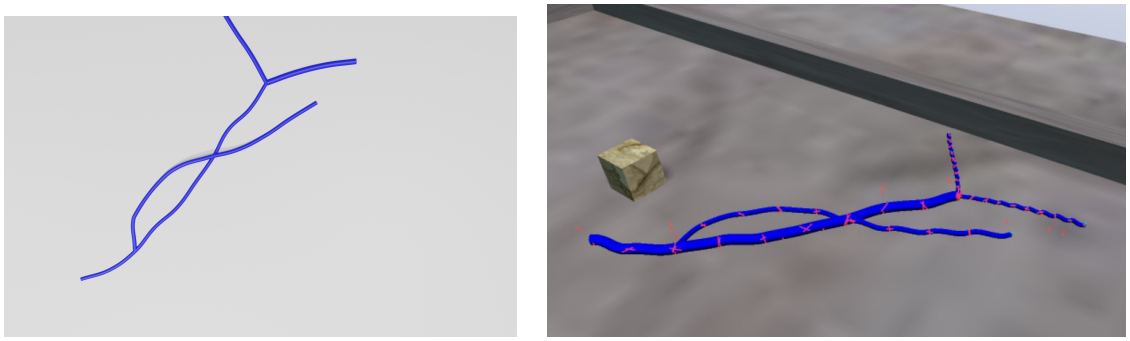

Figure 2.8: Pointcloud and final result

The class that implements this cable is called ***MultiCable***.
Since the number of points per cable obtained from the method described before is too high, we decided to use only a portion of them, by spanning the file by a fixed number for every wiring element. This of course reduces the precision of the resulting wiring, and it is well represented in the figure 2.8, where the 3D wiring obtained by the Blender's script differs slightly from the corresponding wiring in PyChrono.
This is due by the fact that the resulting cable elements obtained with a lesser number of nodes will have a different spatial evolution, but this is slightly mitigated by the fact that we change the nodes' direction in the way described above.

2.3 OpenAI Gym Implementation

Gym's API are used to create an initial structure for the reinforced learning section. Since it's created like a class, we can divide its structure into three different functions. The module is imported inside the *main.py*, and from the initial call we can create the 3D environment specifying only the cable that we want to use as a training.
The figure below schematize the class.

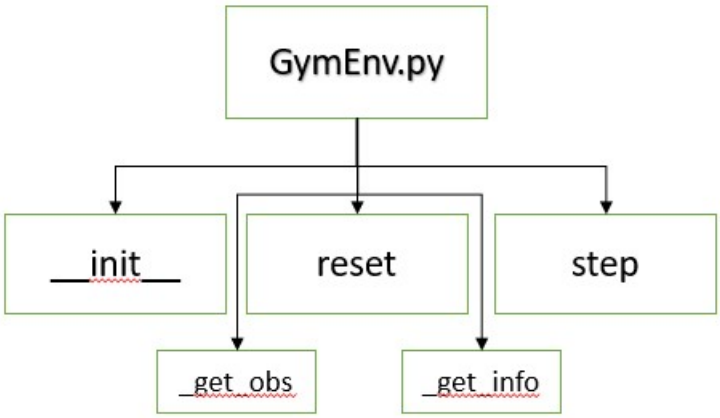

Figure 2.9: Class GymEnv.py scheme

In the *__init__* section we define the cable we want to use and also the action that our agent can take. Since we are using Gym's API, actions are defined through *gym.spaces.Space* superclass[17], that is used to define observation and action spaces. We use two specific functions of the class, namely *gym.spaces.Box*, a possibly unbounded box in IR^n, and *gym.spaces.Discrete*, a space consisting of finitely many elements.
The first one takes as an input a lower value, an higher value and the shape of the resulting matrix. The second one only takes as an input the number of the elements.

The action is divided into three sub-actions:

- The first action, called ***action_space_cable***, decides what wire we want to move. The choice is done using cable's IDs, a simple way to differentiate a cable from the other. In case we have only one cable, the default ID is set to 0.

- The second action is called ***action_space_node*** and decides which node we want to move. Since nodes are stored in a list, we can simply decide which node we want to move by using a simple index.

- The last action is called ***action_space_dir*** and is a *spaces.Box* action in which

we decide the direction of the movement, specified by the angle in the $X-Z$ plane, from 0 to 2π.

Also, in the *__init__* section we define the *system* in which we simulate the evolution of the cable, and we use it by practically "adding" to itself the items created, such as the pavement and the walls, and also the constraint that a section of the cable has with the other, in case we are working on a cable with various bifurcations.
The other main function of the *GymEnv.py* class is the ***reset*** function. We use this function inside the *main.py* to start anew the simulation, with different final position for a given cable, randomly selecting the specific node we want to move inside the 3D environment. This action must be performed every time we start the training to create random objectives.
Last main function of the *GymEnv.py* class is the ***step*** function. We call it every step in the main simulation loop and takes as input the kind of action we want to apply, so the direction, the **node** and the cable we want to move, a **timer** to specify the velocity of the node, and a flag **done** to check when the final position is reached.
Since this has to be a preliminary implementation for the reinforced learning techniques, we also defined some minor functions to check the state of the agent, called ***_get_obs***. It returns the position of the agent (a.k.a. node) and the final position it needs to reach. Similar to *_get_obs*, the function ***_get_info*** returns the distance between the agent and the objective.
All this functions will be used during a future training section.

2.4 Data Collection and Analysis

In order to analyze the evolution in time of the cable's model, we need to save the positions of every node in a format that can be analyzed with the use of Python's functions.
This is done using *Python's Pandas*[18] module, a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. The main use of this module is to create a DataFrame, Panda's way to define a data table, that can be spreadsheets and databases.
The rows of the DataFrame are composed of the nodes of each cable, divided by their respective IDs. The first column has the rest position and the following ones have the position at the give *Data Time*. The sampling time can be decided as a multiple of the *Time Step* of the simulation. If the sampling time is too similar to the Time Step, the fragmentation of the DataFrame becames too instable for the simulator. In general, during the experiments, it was set to 100 times bigger than the step time.
The DataFrame we created has the following conformation, represented in the table below.

	Rest Position	Time interval 1	Time interval 2
Cable ID			
Node ID0	Position [x,y,z]	Position [x,y,z]	Position [x,y,z]
Node ID1	Position [x,y,z	Position [x,y,z]	Position [x,y,z]
....	....	....	....
Node IDn	Position [x,y,z]	Position [x,y,z]	Position [x,y,z]
Cable ID			
Node ID0	Position [x,y,z]	Position [x,y,z]	Position [x,y,z]
Node ID1	Position [x,y,z	Position [x,y,z]	Position [x,y,z]
....	....	....	....
Node IDn	Position [x,y,z]	Position [x,y,z]	Position [x,y,z]

Table 2.2: DataFrame Schematic Structure

The DataFrame is then saved as a *NodePositions.csv* file, that can be opened in different scripts, all managed by Pandas functions.
Since Pandas saves the data as string, to analyze the evolution of the cable we created a script that converts every section of the DataFrame in rows of a single list. Every element of the row is converted into a 3x1 vector. As a section we refer to the rows between the IDs.
This data can then be further analyzed to ensure the correct behaviour of the simulation, and to extract specific information we want to use.

Chapter 3

Tests performed and results

In this chapter we will describe the tests we performed to validate the model, along with the parameters the simulation used.
The tests covered mainly the collision detections with other objects, which can be either with another wiring or with an obstacle. Furthermore, this tests were also used to modify the physical parameters of the other objects, in order to reach the most stable condition possible.

3.1 Simulation parameters and Velocity Profile

Similarly to the information about the wiring, the parameters of the simulation are stored inside a dictionary, namely *parameters.yaml*. This is done to ease the modification process of the simulation behaviour.
Pavement, walls and the obstacle practically behaves in the same way, so the physical parameters are equal, with the exception of the **Young's Modulus**, for the reasons explained in Chapter 2.1. During the tests, the model showed a particular instability regarding the gravity interactions between the wiring and the floor: in fact, if the wiring is too light, i.e. the diameter of the cable is too small, the cable itself would start bouncing without any kind of force or velocity applied. Modifying the Young's Modulus of only the pavement resulted in a better general behaviour of the simulation while standing still. Parameters used for the cube and the pavement are showed in the table below.

Pavement	
E: Young Modulus	0.5e3 Pa
R: Restitution	0.5
F: Friction	0.3
A: Adhesion	0.0
Cube	
E: Young Modulus	1e6 Pa
R: Restitution	0.5
F: Friction	0.3
A: Adhesion	0.0

Table 3.1: Pavement and Cube parameters

The most important parameters of the simulations are the one regarding the cable itself. The table 3.2 shows such parameters.

Cable	
nodes	10
E0: Young Modulus	3.5e6 Pa
E1: Young Modulus	4.5e6 Pa
E2: Young Modulus	5.5e6 Pa
F: Friction	0.3
R: Restitution	1
A: Adhesion	1
d: damping	0.05
ro1: density	2e3 Kg/m^3
ro1: density	3e3 Kg/m^3
ro1: density	4e3 Kg/m^3
length	1 m
diameter	0.015 m
starting pos:	[0, 0.06, 0]
max vel	0.5 m/s
acc	2.5 m/s^2
meshE0: Young Modulus	5e2 Pa

Table 3.2: Cable Parameters

Starting from the first, **nodes** decide the number of nodes each cable has. This number should be as high as possible, to ensure a granular representation of the cable. A drawback of having a high number of nodes is that the simulation becomes more and more unstable, and the computational load of the simulator becomes unbearable.
This is the reason behind having 10 nodes per cable. Going from 10 to 20 nodes per cable created a simulation too unstable, and the cable itself had a behaviour where its configuration started collapsing while doing nothing.
E0, **E1** and **E2** are three different Young's Modulus used through the tests. The values were selected starting from a previous experiment[19]. Similarly to the various Young's Modulus, also the different densities **ro1**, **ro2** and **ro3** were obtained starting from the experiment[19].
The parameters **max_vel** and **acc** describe the velocity profile we wanted to assign to the nodes when performing a single movement. The last parameter, **meshE0**, is used to define the collisions meshes behaviour, and is set to 500 Pa in order to avoid the interpenetrations during the contact. This values was found after several tests.
The decided velocity profile is the *trapezoidal* one, and its general behaviour is described by the figure below:

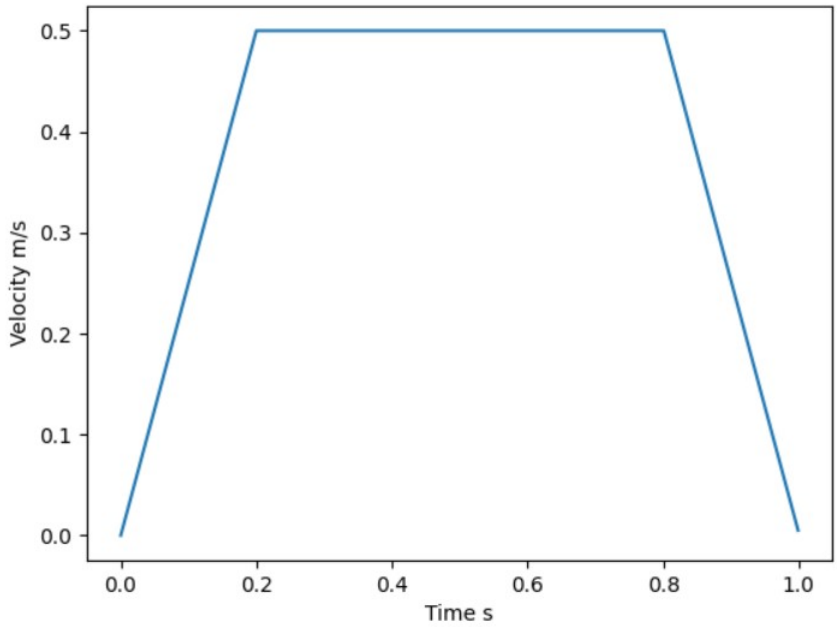

Figure 3.1: Velocity profile

Lastly, **starting_pos**, **diameter** and **length** are parameters only used by *StraightCable* and *Straight3Cable* models.

Lastly, we needed to define solver's parameters, shown in the table below:

solver	
maxiterations	40
tolerance	1e-22
forcetolerance	1e-10
timestep	0.0001

Table 3.3: Solver Parameters

Mainly we used the default parameters with respect to the max iterations the solver could do before signaling an error, the tolerance and the force tolerance.
The one that needed tweeks is the **timestep**. This decide how much time occurs between two steps. One could think that a low value would result in a more precise simulation but, as we found out during the tests, the timestep's value shows two different behaviours:

- If the **timestep** is set to a high value, in the order of the tenth or hundredth of a second, the simulation is too unstable. This is due to the precision of the final result, that needs a low value for the time step
- If the **timestep** is set to a low value, in the order of 10^{-8} seconds and below, the simulation shows strange behaviours, where the wiring starts to move randomly.

The parameter also affect the computational load of the simulation, and its length in time.

3.2 Cable to Object collisions detection

The first thing tested is the behaviour of the cable when it collides with another physical object other than itself. This test was performed with a single straight wiring with diameter of 1.5 centimeters and a total length of 1 meter.
The object used to detect the collision is a simple cube of side of 10 centimetres and density of 5000 kg/m^3.
The figure below show some salient frames of the simulation, along with a schematic representation of nodes positions and the velocity of the one we moved.

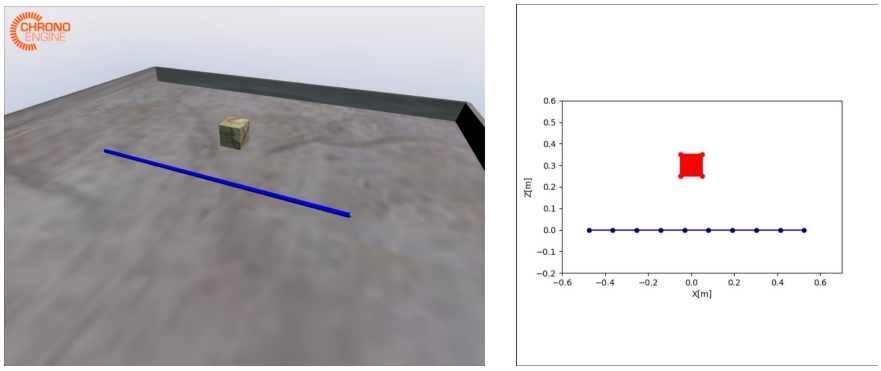

Figure 3.2: Cable-to-object collisions setup - Red object mesh - Blue cable mesh

The movement used to cause the collision is a simple linear translation with maximum velocity of 0.5 m/s along the Z direction. The **timestep** is set to *0.0001*.

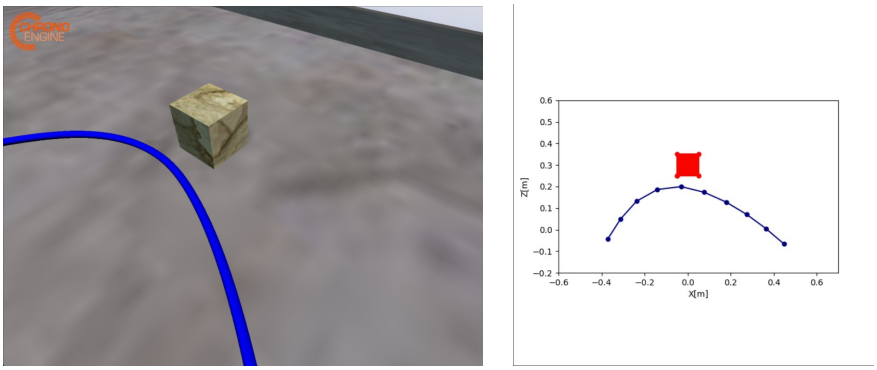

Figure 3.3: Cable-to-object collisions time instant 0.5s - Red object mesh - Blue cable mesh

After 0.5s, the cable still needs to reach the object. This is because the distance between the cube and the cable is of 25 centimetres.

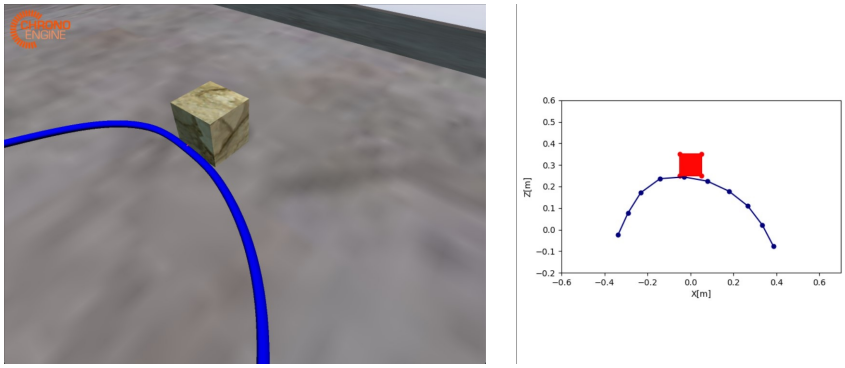

Figure 3.4: Cable-to-object collisions time instant 0.6s - Red object mesh - Blue cable mesh

After 0.6s, the collision between the cable and the object started and, as you can see by the frame, the node 5 stopped with 0 velocity.

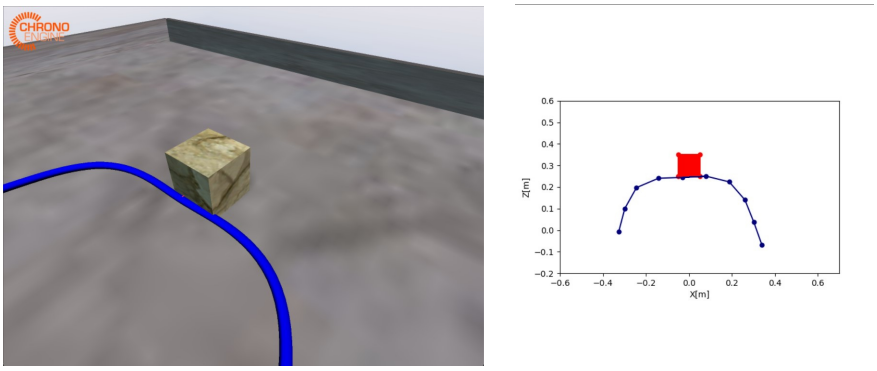

Figure 3.5: Cable-to-object collisions time instant 0.7s - Red object mesh - Blue cable mesh

After 0.7s, the node 5 is still attached to the cube, but other nodes continue moving. This is because the simulator tend to apply always the velocity to the node if you not say otherwise. So the node 5, that still has applied a velocity of 0.5 m/s, tends to stay stick to the object surface, while the other nodes, because of the inertia, continue to move along the Z direction until the friction stops them.
As you can see, the cable interact with the object in the correct way. Since the object is way heavier than the cable, it stops immediately the cable's movement. The Young Modulus is high enough to impede interpenetrations between the cable and the cube, and the contact happens without any bouncing of the cable.
Last figure shows the absolute velocity of the **4th** node. As you can see, until the node reaches the object, its velocity is 0.5 m/s but, as it collides, the velocity goes to 0 after some time. We can clearly see the deceleration phase.

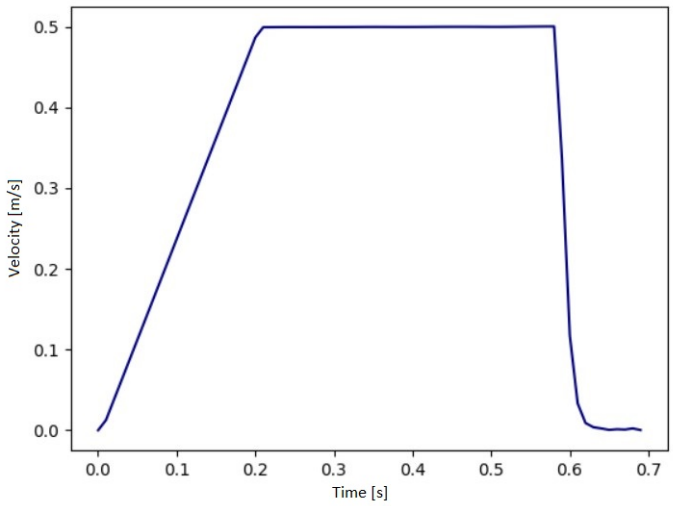

Figure 3.6: Node 4 absolute velocity profile

3.3 Cable to Cable collisions detection

The second set of tests performed aimed to show the stability of cable-to-cable collisions.In this case, we created the environment with two identical cables and no other object to interact with.
Both cables were made of 10 nodes each, 15 centimetres of diameter and 1 meter of length. As of velocities, the main cable was driven towards the second one with a velocity of $0.3\,m/s$ along the Z direction. The main cable was also shifted 5 centimetres towards the X-axis, to ensure that the collisions were detected also node-to-cable element and not only node-to-node. That's why we created two different meshes to detect collisions, one containing the nodes and one containing the tests.

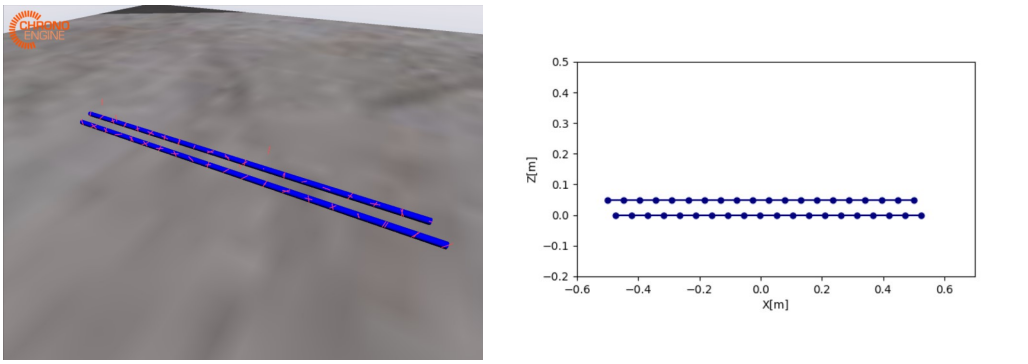

Figure 3.7: Cable-to-cable collisions setup

The main purpose of this test was to find the perfect parameters of the *contact meshes*. In fact, during the tests, we needed to heavily modify such parameters, like the **Young's modulus** of the contact mesh, since an high value caused a rebound between the cables, and a low value instead caused a interpenetration, resulting in an unstable simulation.

The following images shows some salient shots of the simulation.

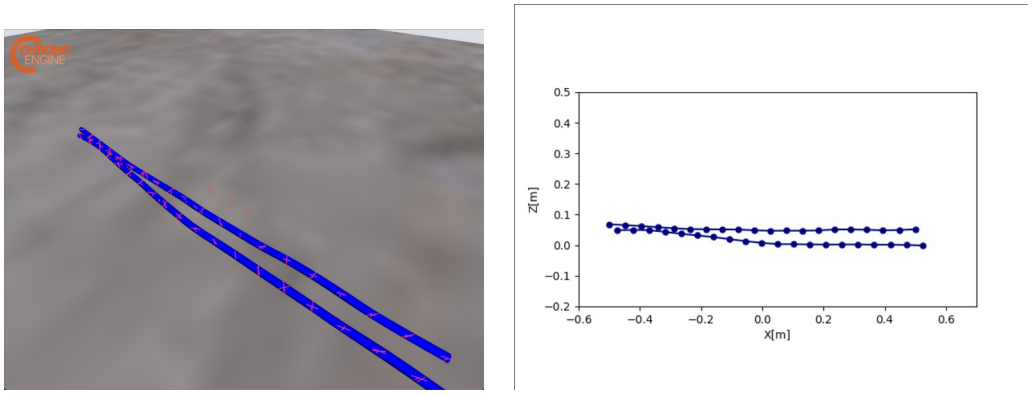

Figure 3.8: Cable-to-cable collision time instant 0.2s

As you can see from the figure 3.11, after 0.2s cables start the collision. The parameters are chosen in a manner that no bouncing happens between the two cables. The collision begins little before the real contact between the cable elements, maybe to assure a stable simulation.

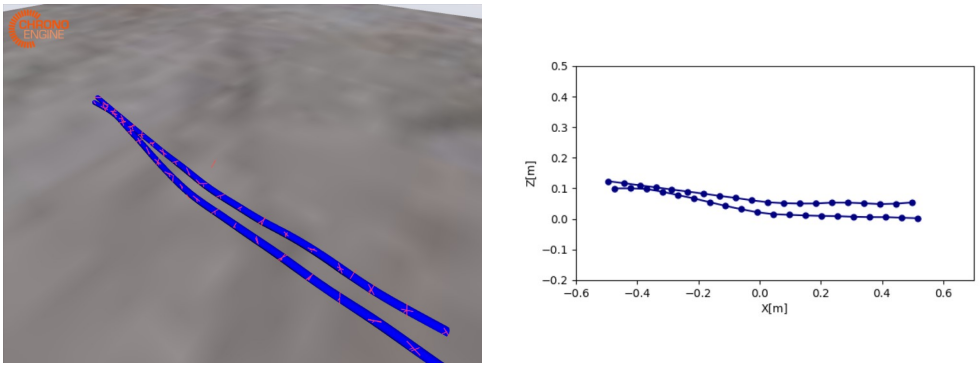

Figure 3.9: Cable-to-cable collision time instant 0.3s

After 0.3s, the collisions between the two cables is still going on. The collisions meshes, along with the cable elements, tend to interpenetrate a little. Anyway, this interpenetration is low enough to assure a stable simulation with a correct result.

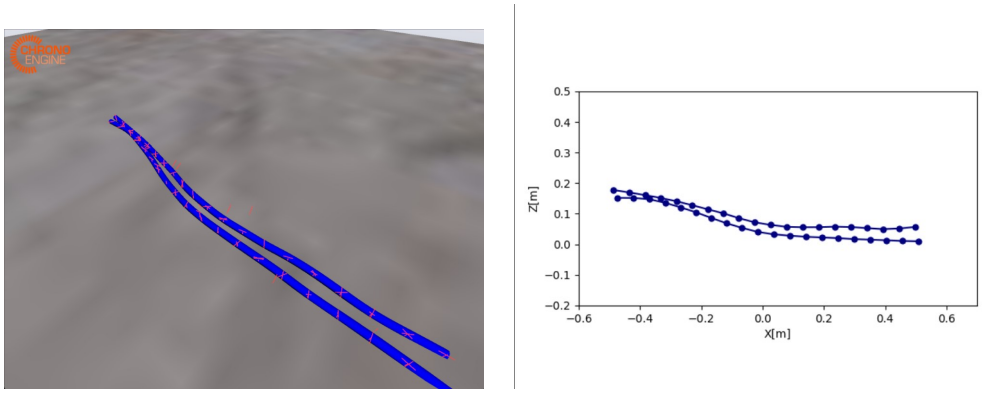

Figure 3.10: Cable-to-cable collision time instant 0.4s

After 0.4s, the movement is still going on and we can see that the collisions are well managed by the simulator. Both cables react with each other without any rebound, and the general stability of the simulation is ensured. We can see some bouncing of the end parts of the secondary cable, caused probably by some partial interpenetration of the cable with the pavement, but they eventually stops, without any interference with the main simulation.

Last figure shows the absolute velocity of the **second** Node. Its velocity is practically constant, with some fluctuations given by the collisions between the cable and the pavement, but the collision between the two cables doesn't affect such value.

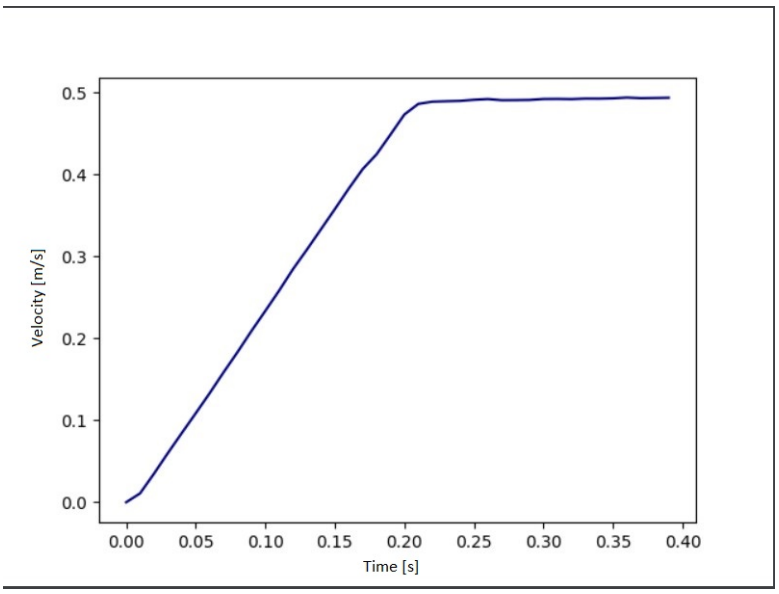

Figure 3.11: Node 2 absolute velocity profile

3.4 Straight wiring movement and collisions detection

One of the cable's models described in Chapter 2 had the purpose of testing how a single wiring made of multiple cables could be done. The results of the solution proposed were not satisfactory.

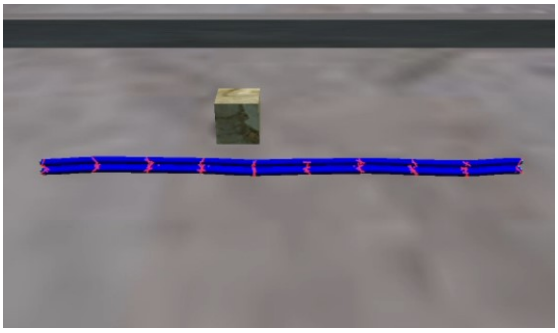

Figure 3.12: Straight wiring Time instant 0

As shown by the figure above, initially the wiring has a general structure not completely correct. We can see some waves in his shape, probably given by the fact that the simulator could not manage the whole collision detection.

After some time instants, the initial and intended shape is completely lost, as shown by the figure below.

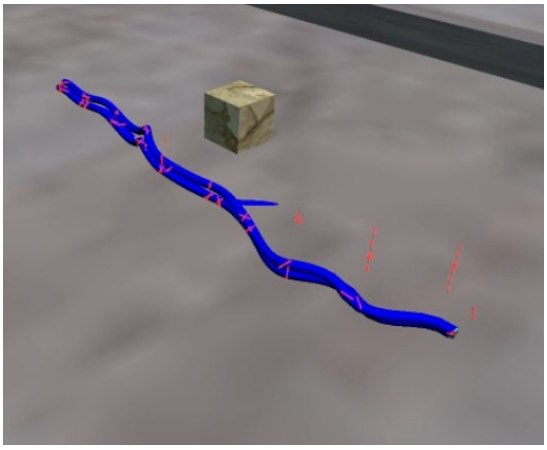

Figure 3.13: Straight wiring Time instant 0.1s

The instability of the whole simulation was so bad that the wiring could not even do a single step without the nodes shifting into random places. Some of them also inteprenetrated the pavement itself.

This got us to the conclusion that, in order to create a wiring made of multiple cables connected without any branch, the best solution was to just create a single cable with the resulting diameter congruous with the original wiring.

3.5 General wiring movement and collisions detection

Starting from the previous test, we found that the most simple way to create a compound cable is also the most efficient. The cable firstly used is the one described in chapter 2, but tests showed a limit of the simulator that can't be solved by simple means, but needs a completely rework on how we create the general cable.

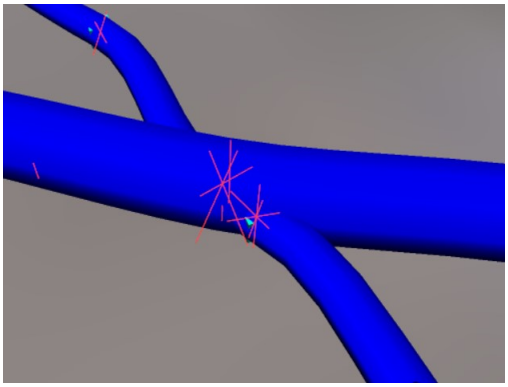

Figure 3.14: Wiring interpenetration close-up

As you can see from the image above, the trajectories of two of the cables composing the wiring intersect. This intersection, in the initial time instants of the simulation, needs to be resolved and, when the two sections of the cable divide, the general stability of the simulation become less, at point where the nodes of the general wiring starts bouncing in every direction possible.
This behaviour shows that, at the moment of the creation of a wiring, we need to eliminate all possibility of intersection between different part of the wiring.
In order to address this unexpected behaviour, we decided to design another cable, simpler from the physical configuration point of view. This is done because we need only to test if the simulation is stable enough to address this kind of wiring, but the way they are loaded into the simulation itself can be arbitrarily modified later on.
The new cable is similar to the more complex one, where it's made of a main and bigger structure, with a single branch starting from the middle, that has half it's diameter. It's represented in the figure below.

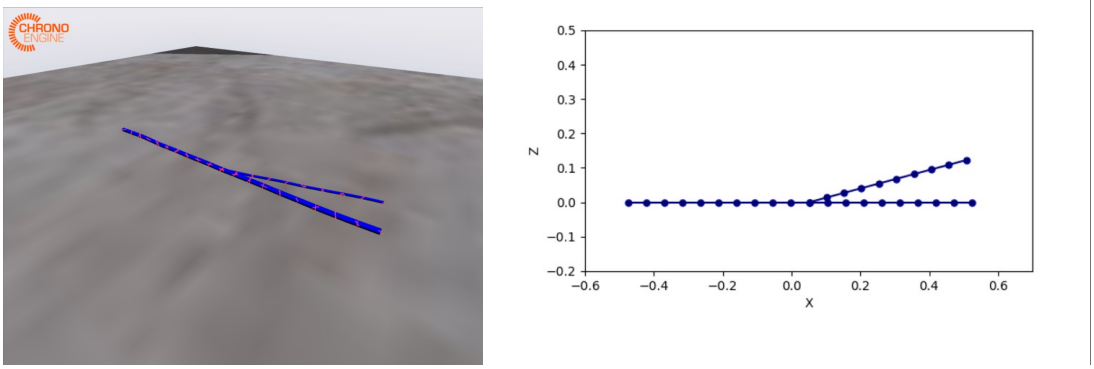

Figure 3.15: Wiring collisions setup

As the test itself, we decided to move the end of the main cable toward the secondary one, with a fixed velocity of 0.5 m/s over the Z direction, in order to test if this kind of configurations would work with PyChrono.

Following figures represent salient frames of the simulation.

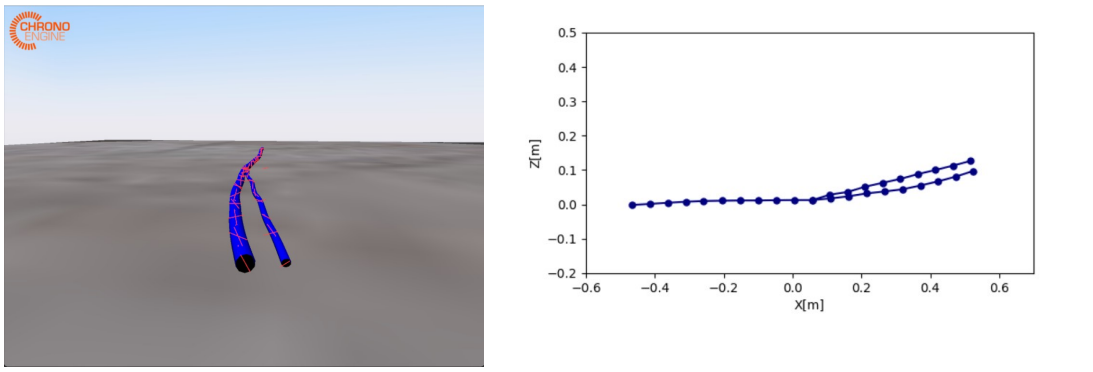

Figure 3.16: Wiring collisions time instant 0.3s

After 0.3s, the end of the main cable still has to reach the branch, but the physical position of the branch itself start moving along a little.

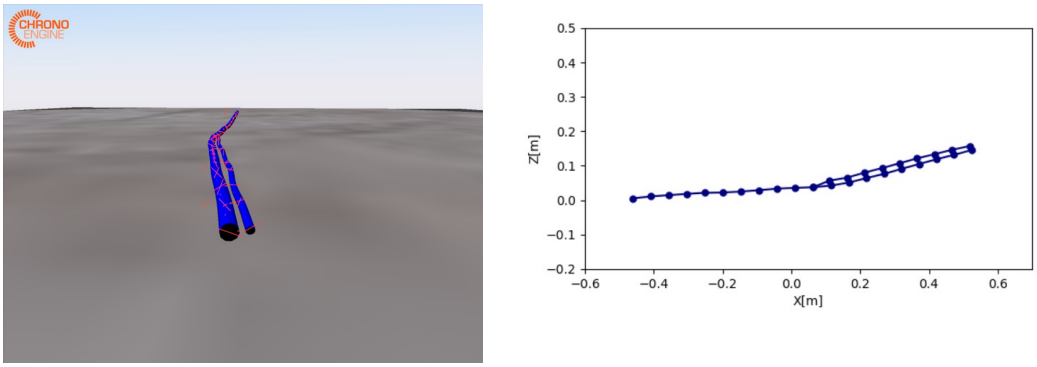

Figure 3.17: Wiring collisions time instant 0.4s

After 0.4s, the simulator starts to record the collision between the main cable and the branch, similarly to what happened in section 3.4. The branch modify its position along with the movement of the main cable's node.

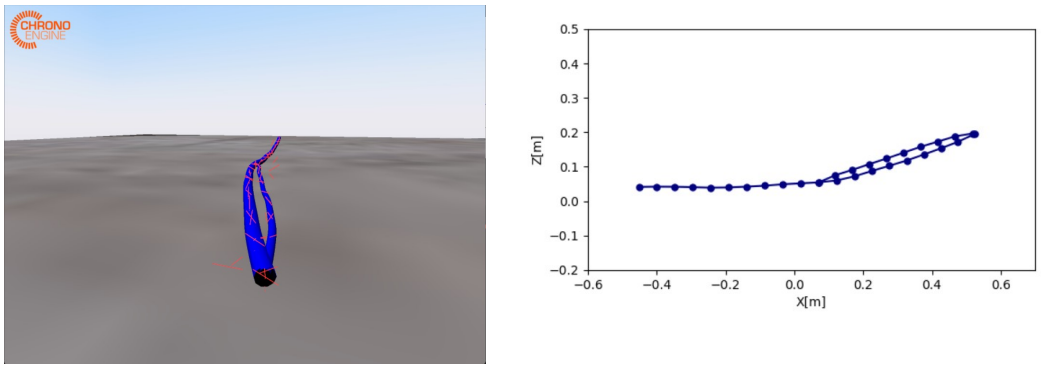

Figure 3.18: Wiring collisions time instant 0.5s

After 0.5s, the main cable starts to penetrate the branch, but this interpenetration doesn't destabilize the simulation, that continues to show a good behaviour of the collisions.

Lastly, similarly to the velocity profile of the **cable to cable collision**, the velocity of the last node of the main cable has a regular behaviour with some fluctuations given by the collisions with the pavement, while the collision with the branch doesn't creates much modifications in its profile.

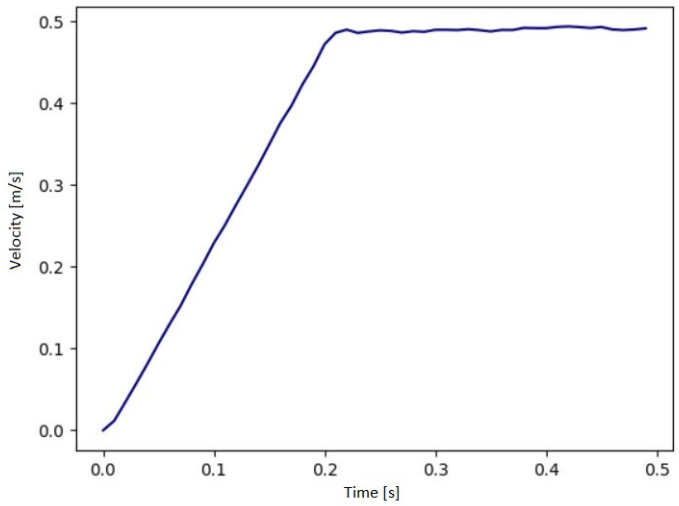

Figure 3.19: Main Cable Last Node absolute velocity profile

3.6 Test Results

Test results have highlighted the current limits of this approach, along with some needs in improving cable's parameters. The first experiment regarding collisions between cable and object showed that the collision is only registered between nodes and the 3D objects: this limit has a simple solution, that is increase the number of nodes, but an higher number of nodes makes the simulation too unstable or complex to compute. So there's a trade-off between the number of nodes and the stability-correctness of the simulation, in which we decided to stay in the range of 10 nodes per wiring section.
Furthermore, the high number of nodes creates more instability when dealing between cable to cable collisions. In fact, during the other experiments, the collisions are registered also when a node hits the cable element. This shows a correct behaviour of the model, so higher number of nodes doesn't mean a more correct management of the collision itself, but a more realistic representation of the cable spatial evolution under the conditions applied. Going from 10 nodes per cable to 20 nodes per cables shows in some way expected results: the general simulation slows down, since it has to compute twice the amount of collision per time instant, but the stability of the cables starts to fall

apart, since they start bouncing with the pavement and interpenetrate with it, as shown by the experiment dealing with the wiring made of three different cables bounded one to the other.
Another interesting result regards velocity profiles of the nodes. As you can see from the results regarding cable to cable collisions, the node that we move by injecting a specific velocity, doesn't show any change in behaviour even when colliding with the other cable. This shows that the velocity that the node has every time instant is not so much influenced by the collision with a similar object, such as another cable, but instead is only affected by the value we inject as an input. In fact, on the graphs we can see a little fluctuation to the downside, that is recovered slowly after.

Chapter 4

Conclusions

Through this work of thesis, we studied the effectiveness of a 3D environment in which we simulate and move a wiring using the PyChrono library. The model itself was created using PyChrono's API, starting from the physical structure of the wiring, and then designing the collision system between the wiring and other objects, and the wiring against other cables.
As shown by the results obtained during the tests, the model presents a stable behaviours when managing the collisions between a cable and an object, but shows some limits when dealing with cable-to-cable collisions. Such collisions present unstable behaviours, accentuated by the number of the nodes.
The main favorable feature of the PyChrono Simulation Engine is the process that allows us to create the environments. The ease of building new cables' configurations starting from spatial data reduces the time invested in developing specific functions for a given cable. Furthermore, the definition of the cable's physical behaviour is simplified by PyChrono's APIs.
The ease in developing new configurations for the cable brings also some drawback in managing interpenetration when creating the cable from scratch. The presence of interpenetrations between cable's meshes isn't something that the current simulator can manage, resulting in a simulation where the wiring itself is uncontrollable.
Furthermore, future developments will focus on a more specific algorithm that takes also in consideration such interpenetrations, in order to establish the final stability that this physical engine needs.

Bibliography

[1] Jose Sanchez et al. "Robotic manipulation and sensing of deformable objects in domestic and industrial applications: a survey". In: *The International Journal of Robotics Research* 37.7 (2018), pp. 688–716. DOI: 10.1177/0278364918779698. eprint: https://doi.org/10.1177/0278364918779698. URL: https://doi.org/10.1177/0278364918779698.

[2] A. Theetten, L. Grisoni, and B. Barsky C. Andriot. "Geometrically exact dynamic splines". In: (2008).

[3] Carl de Boor. "Package for Calculating with B-Splines". In: *SIAM Journal on Numerical Analysis* 14.3 (1977), pp. 441–472. ISSN: 00361429. URL: http://www.jstor.org/stable/2156696 (visited on 10/17/2022).

[4] Courbon J. *Theorie des poutres*. 1980.

[5] Chouaiebu. "Chouaieb N. Kirchhoff's problem of helical solutions of uniform rods and their stability properties". PhD thesis. 2004.

[6] University of Parma-Italy University of Wisconsin-Madison. *Project Chrono*. URL: https://projectchrono.org/.

[7] University of Parma-Italy University of Wisconsin-Madison. *PyChrono*. URL: https://projectchrono.org/pychrono/.

[8] Alessandro Tasora et al. "A geometrically exact isogeometric beam for large displacements and contacts". In: *Computer Methods in Applied Mechanics and Engineering* 358 (2020), p. 112635. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2019.112635. URL: https://www.sciencedirect.com/science/article/pii/S0045782519305195.

[9] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[11] University of Parma-Italy University of Wisconsin-Madison. *ChBeamSection documents*. URL: https://api.projectchrono.org/classchrono_1_1fea_1_1_ch_beam_section_euler_advanced.html.

[12] University of Parma-Italy University of Wisconsin-Madison. *ChNode documents*. URL: https://api.projectchrono.org/classchrono_1_1fea_1_1_ch_node_f_e_axyzrot.html.

[13] University of Parma-Italy University of Wisconsin-Madison. *ChBeam documents*. URL: https://api.chrono.projectchrono.org/classchrono_1_1fea_1_1_ch_element_beam_euler.html.

[14] University of Parma-Italy University of Wisconsin-Madison. *ChMesh documents*. URL: https://api.projectchrono.org/classchrono_1_1fea_1_1_ch_mesh.html.

[15] University of Parma-Italy University of Wisconsin-Madison. *ChBodyEasyBox documents*. URL: https://api.projectchrono.org/classchrono_1_1_ch_body_easy_box.html.

[16] C. C. Paige and M. A. Saunders. "Solution of Sparse Indefinite Systems of Linear Equations". In: *SIAM Journal on Numerical Analysis* 12.4 (1975), pp. 617–629. DOI: 10.1137/0712047. eprint: https://doi.org/10.1137/0712047. URL: https://doi.org/10.1137/0712047.

[17] Greg Brockman et al. *Gym Documentation*. URL: https://www.gymlibrary.dev/api/spaces/.

[18] Inc. Hosted by OVHCloud NumFOCUS. *Pandas*. URL: https://pandas.pydata.org/.

[19] Hongwang Du, Qinwen Jiang, and Wei Xiong. "Dynamic geometrical configuration predictions during robotic manipulation for automated cable assembly". In: *Journal of Manufacturing Systems* 64 (2022), pp. 121–132. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2022.06.001. URL: https://www.sciencedirect.com/science/article/pii/S0278612522000966.